Adrian-Horia Dediu
Henning Fernau
Carlos Martín-Vide (Eds.)

LNCS 6031

Language and Automata Theory and Applications

4th International Conference, LATA 2010
Trier, Germany, May 2010
Proceedings

Springer

Adrian-Horia Dediu   Henning Fernau
Carlos Martín-Vide (Eds.)

Language and Automata Theory and Applications

4th International Conference, LATA 2010
Trier, Germany, May 24-28, 2010
Proceedings

Springer

Volume Editors

Adrian-Horia Dediu
Carlos Martín-Vide
Universitat Rovira i Virgili
Research Group on Mathematical Linguistics
Avinguda Catalunya, 35, 43002 Tarragona, Spain
E-mail: {adrian.dediu; carlos.martin}@urv.cat

Henning Fernau
Universität Trier, Fachbereich IV - Informatik
Campus II, Behringstraße, 54286 Trier, Germany
E-mail: fernau@informatik.uni-trier.de

Library of Congress Control Number: 2010926873

CR Subject Classification (1998): F.1, J.3, I.2, I.5, I.4, F.2

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

ISSN	0302-9743
ISBN-10	3-642-13088-7 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-13088-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

These proceedings contain all the papers that were presented at the 4th International Conference on Language and Automata Theory and Applications (LATA 2010), held in Trier, Germany, during May 24–28, 2010.

The scope of LATA is rather broad, including: algebraic language theory; algorithms on automata and words; automata and logic; automata for system analysis and program verification; automata, concurrency and Petri nets; cellular automata; combinatorics on words; computability; computational complexity; computer 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; 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; neural networks; parallel and regulated rewriting; parsing; pattern matching and pattern recognition; patterns and codes; power series; quantum, chemical and optical computing; semantics; string and combinatorial issues in computational biology and bioinformatics; symbolic dynamics; term rewriting; text algorithms; text retrieval; transducers; trees, tree languages and tree machines; and weighted machines.

LATA 2010 received 115 submissions, many among them of good quality. Each one was reviewed by at least three Program Committee members plus, in most cases, by additional external referees. After a thorough and vivid discussion phase, the committee decided to accept 47 papers (which means an acceptance rate of 40.86%). The conference program also included four invited talks. 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 reviewers for their cooperation and Springer for the collaboration and publication.

February 2010

Adrian-Horia Dediu
Henning Fernau
Carlos Martín-Vide

Organization

LATA 2010 took place in Trier, Germany under the organization of the University of Trier and the Research Group on Mathematical Linguistics (GRLMC), Rovira i Virgili University, Tarragona, Spain.

Program Committee

Alberto Apostolico	Atlanta, USA
Thomas Bäck	Leiden, The Netherlands
Stefania Bandini	Milan, Italy
Wolfgang Banzhaf	St. John's, Canada
Henning Bordihn	Potsdam, Germany
Kwang-Moo Choe	Daejeon, Korea
Andrea Corradini	Pisa, Italy
Christophe Costa Florêncio	Leuven, Belgium
Maxime Crochemore	Marne-la-Vallée, France
W. Bruce Croft	Amherst, USA
Erzsébet Csuhaj-Varjú	Budapest, Hungary
Jürgen Dassow	Magdeburg, Germany
Volker Diekert	Stuttgart, Germany
Rodney G. Downey	Wellington, New Zealand
Frank Drewes	Umeå, Sweden
Henning Fernau (Co-chair)	Trier, Germany
Rusins Freivalds	Riga, Latvia
Rudolf Freund	Wien, Austria
Paul Gastin	Cachan, France
Edwin Hancock	York, UK
Markus Holzer	Giessen, Germany
Helmut Jürgensen	London, Canada
Juhani Karhumäki	Turku, Finland
Efim Kinber	Fairfield, USA
Claude Kirchner	Bordeaux, France
Brian Marcus	Vancouver, Canada
Carlos Martín-Vide (Co-chair)	Brussels, Belgium
Risto Miikkulainen	Austin, USA
Victor Mitrana	Bucharest, Romania
Claudio Moraga	Mieres, Spain
Sven Naumann	Trier, Germany
Chrystopher Nehaniv	Hatfield, UK

Maurice Nivat	Paris, France
Friedrich Otto	Kassel, Germany
Daniel Reidenbach	Loughborough, UK
Klaus Reinhardt	Tübingen, Germany
Antonio Restivo	Palermo, Italy
Christophe Reutenauer	Montréal, Canada
Kai Salomaa	Kingston, Canada
Jeffrey Shallit	Waterloo, Canada
Eljas Soisalon-Soininen	Helsinki, Finland
Bernhard Steffen	Dortmund, Germany
Frank Stephan	Singapore
Wolfgang Thomas	Aachen, Germany
Marc Tommasi	Lille, France
Esko Ukkonen	Helsinki, Finland
Todd Wareham	St. John's, Canada
Osamu Watanabe	Tokyo, Japan
Bruce Watson	Pretoria, South Africa
Thomas Wilke	Kiel, Germany
Sławomir Zadrożny	Warsaw, Poland
Binhai Zhu	Bozeman, USA

External Reviewers

Andy Adamatzky
Frédéric Alexandre
Marco Antoniotti
Paolo Baldan
Miklos Bartha
Henrik Björklund
Guillaume Blin
Benedikt Bollig
Olivier Bournez
Patricia Bouyer-Decitre
Roberto Bruni
Giusi Castiglione
Loek Cleophas
Thomas Colcombet
Carsten Damm
Stéphane Demri
Alberto Dennunzio
Chiara Epifanio
Patricia Evans
Fabio Farina
Dominik Freydenberger
Fabio Gadducci
Olivemarie Garland
Laura Giambruno
Stefan Haar
Peter Habermehl
Serge Haddad
Vesa Halava
Yo-Sub Han
Tero Harju
Ulrich Hertrampf
Mika Hirvensalo
Norbert Hundeshagen
Florent Jacquemard
Asher Kach
Akinori Kawachi
Derrick Kourie
Manfred Kudlek
Manfred Kufleitner
Marco Kuhlmann
Martin Kutrib
Anna-Lena Lamprecht
Jürn Laun
Cédric Lauradoux

Organizing Committee

Table of Contents

Complexity in Convex Languages

Janusz Brzozowski

David R. Cheriton School of Computer Science, University of Waterloo,
Waterloo, ON, Canada N2L 3G1
`brzozo@uwaterloo.ca`

Abstract. A language L is prefix-convex if, whenever words u and w are in L with u a prefix of w, then every word v which has u as a prefix and is a prefix of w is also in L. Similarly, we define suffix-, factor-, and subword-convex languages, where by subword we mean subsequence. Together, these languages constitute the class of convex languages which contains interesting subclasses, such as ideals, closed languages (including factorial languages) and free languages (including prefix-, suffix-, and infix-codes, and hypercodes). There are several advantages of studying the class of convex languages and its subclasses together. These classes are all definable by binary relations, in fact, by partial orders. Closure properties of convex languages have been studied in this general framework of binary relations. The problems of deciding whether a language is convex of a particular type have been analyzed together, and have been solved by similar methods. The state complexities of regular operations in subclasses of convex languages have been examined together, with considerable economies of effort. This paper surveys the recent results on convex languages with an emphasis on complexity issues.

Keywords: automaton, bound, closed, complexity, convex, decision problem, free, ideal, language, quotient, regular, state complexity.

1 Convex Languages

We begin by defining our terminology and notation. If Σ is a non-empty finite *alphabet*, then Σ^* is the free monoid generated by Σ. A *word* is any element of Σ^*, and the *empty word* is ε. A *language* over Σ is any subset of Σ^*.

If $u, v, w, x \in \Sigma^*$ and $w = uxv$, then u is a *prefix* of w, x is a factor of w, and v is a *suffix* of w. A prefix or suffix of w is also a factor of w. If $w = w_0a_1w_1 \cdots a_nw_n$, where $a_1, \dots, a_n \in \Sigma$, and $w_0, \dots, w_n \in \Sigma^*$, then $x = a_1 \cdots a_n$ is a *subword* of w. Every factor of w is also a subword of w.

Definition 1. *A language L is* prefix-convex *if $u, w \in L$ with u a prefix of w implies that every word v must also be in L if u is a prefix of v and v is a prefix of w; L is* prefix-free *if $w \in L$ implies that no prefix of w other than w is in L; L is* prefix-closed *if $w \in L$ implies that every prefix of w is in L; L is* converse prefix-closed *if $w \in L$ implies that every word that has w as a prefix is also in L.*

In a similar way, we define *suffix-convex*, *factor-convex*, and *subword-convex* languages, and the corresponding free, closed, and converse closed versions.

A language is *bifix-convex* (respectively, *bifix-free* or *bifix-closed*) if it is both prefix- and suffix-convex (respectively prefix- and suffix-free or prefix- and suffix-closed). The class of bifix-closed languages coincides with the class of factor-closed languages [1].

Definition 2. *A language $L \subseteq \Sigma^*$ is a* right ideal *(respectively,* left ideal*,* two-sided ideal*) if it is non-empty and satisfies $L = L\Sigma^*$ (respectively, $L = \Sigma^* L$, $L = \Sigma^* L \Sigma^*$). A two-sided ideal which satisfies the condition $L = \Sigma^* \sqcup\!\sqcup L = \bigcup_{a_1 \cdots a_n \in L} \Sigma^* a_1 \Sigma^* \cdots \Sigma^* a_n \Sigma^*$, where $\sqcup\!\sqcup$ is the shuffle operator, is called an* all-sided ideal*. We refer to all four types as* ideal languages *or simply* ideals*.*

Ideals and closed languages are related as follows [1]: A non-empty language is a right ideal (respectively, left, two-sided, or all-sided ideal ideal) if and only if its complement is not Σ^* and is prefix-closed (respectively, suffix-, factor-, or subword-closed).

Here is a brief history of the work on the class of convex languages and its subclasses; for more details see [1]. To avoid making many definitions of other authors' terms, we use our own terminology when discussing previous results.

Sets that are closed with respect to an arbitrary reflexive and transitive binary relation were introduced in 1952 by Higman [15]. His results were rediscovered several times in various contexts; a detailed account of this history was given by Kruskal [19] in 1972. Left and right ideals were studied by Paz and Peleg [22] in 1965 under the names "ultimate definite" and "reverse ultimate definite events". In 1969 Haines [12] examined subword-free, subword-closed and converse subword-closed languages, and also used all-sided ideals. Subword-convex languages were introduced by Thierrin [26] in 1973, who also studied subword-closed and converse subword-closed languages. Subword-free languages were studied by Shyr and Thierrin in 1974 under the name of hypercodes [25]. Suffix-closed languages were examined by Gill and Kou in 1974 [11]. Prefix-free and suffix-free languages (codes) were studied in depth in the 1985 book of Berstel and Perrin [2] and also in an updated version of the book [3]. In 1990, de Luca and Varricchio [20] observed that a language is factor-closed if and only if it is the complement of a two-sided ideal. In 1991 Jürgensen and Yu [17] studied codes that are free languages with respect to many binary relations, including the prefix, suffix, factor and subword relations; that paper also contains additional references to codes. More information about codes can be found in the 1997 article by Jürgensen and Konstantinidis [16]. In 2001 Shyr [24] studied right, left, and two-sided ideals and their generators in connection with codes.

Most of the material in this paper is based on recent work on convex languages [1,5,6,7,8,9,13,14,18,27,28]. The remainder of the paper is organized as follows: In Section 2 we define convex languages in terms of partial orders. Closure properties of convex languages are discussed in Section 3. The complexity of decision problems about convex languages is presented in Section 4. Quotient complexity (which is equivalent to state complexity) is defined in Section 5.

The quotient complexity of boolean operations in convex languages is then covered in Section 6. Section 7 summarizes the known results for the complexity of product, star, and reversal in convex languages. The special case of unary convex languages is treated in Section 8. The complexity of operations in closed and ideal language classes is discussed in Section 9, and Section 10 closes the paper.

2 Languages Defined by Partial Orders

For further details on the material in this section see [1]. The relations used in this paper to define classes of languages are all partial orders. Let $\trianglelefteq$ be an arbitrary partial order on Σ^*. If $u \trianglelefteq v$ and $u \neq v$, we write $u \triangleleft v$. Let $\trianglerighteq$ be the converse binary relation, that is, let $u \trianglerighteq v$ if and only if $v \trianglelefteq u$.

Definition 3. *A language L is* $\trianglelefteq$-convex *if $u \trianglelefteq v$ and $v \trianglelefteq w$ with $u, w \in L$ implies $v \in L$. It is* $\trianglelefteq$-free *if $v \triangleleft w$ and $w \in L$ implies $v \notin L$. It is* $\trianglelefteq$-closed *if $v \trianglelefteq w$ and $w \in L$ implies $v \in L$. It is* $\trianglerighteq$-closed *if $v \trianglerighteq w$ and $w \in L$ implies $v \in L$.*

For an arbitrary partial order $\trianglelefteq$ on Σ^* and a language $L \subseteq \Sigma^*$, define the *closure* ${}_{\trianglelefteq}L$ and *converse closure* $L_{\trianglelefteq}$ of L:

$$ {}_{\trianglelefteq}L = \{v \in \Sigma^* \mid v \trianglelefteq w \text{ for some } w \in L\}, $$

$$ L_{\trianglelefteq} = \{v \in \Sigma^* \mid w \trianglelefteq v \text{ for some } w \in L\}. $$

The following are consequences of the definitions:

Proposition 1. *Let $\trianglelefteq$ be an arbitrary partial order on Σ^*. Then*

1. *A language is $\trianglelefteq$-convex if and only if it is $\trianglerighteq$-convex.*
2. *A language is $\trianglelefteq$-free if and only if it is $\trianglerighteq$-free.*
3. *Every $\trianglelefteq$-free language is $\trianglelefteq$-convex.*
4. *Every $\trianglelefteq$-closed language and every $\trianglerighteq$-closed language is $\trianglelefteq$-convex.*
5. *A language is $\trianglelefteq$-closed if and only if its complement is $\trianglerighteq$-closed.*
6. *A language L is $\trianglelefteq$-closed ($\trianglerighteq$-closed) if and only if $L = {}_{\trianglelefteq}L$ ($L = L_{\trianglelefteq}$).*

The following special cases are of interest to us:

– Let $\leqslant$ denote the partial order "is a prefix of". If $\trianglelefteq$ is $\leqslant$, then we get prefix-convex, prefix-free, prefix-closed, and right ideal languages.
– Let $\preceq$ denote the partial order "is a suffix of". If $\trianglelefteq$ is $\preceq$, then we get suffix-convex, suffix-free, suffix-closed, and left ideal languages.
– Let $\sqsubseteq$ denote the partial order "is a factor of". If $\trianglelefteq$ is $\sqsubseteq$, then we get factor-convex, factor-free, factor-closed, and two-sided ideal languages.
– Let $\Subset$ denote the partial order "is a subword of". If $\trianglelefteq$ is $\Subset$, then we get subword-convex, subword-free, subword-closed, and all-sided ideal languages.

3 Closure Properties

This section is based on [1]. The following set operations are defined on languages: *complement* ($\overline{L} = \Sigma^* \setminus L$), *union* ($K \cup L$), *intersection* ($K \cap L$), *difference* ($K \setminus L$), and *symmetric difference* ($K \oplus L$). A general *boolean operation* with two arguments is denoted by $K \circ L$. We also define the *product*, usually called *concatenation* or *catenation*, ($KL = \{w \in \Sigma^* \mid w = uv, u \in K, v \in L\}$), *star* ($L^* = \bigcup_{i \geqslant 0} L^i$), and *positive closure* ($L^+ = \bigcup_{i \geqslant 1} L^i$). The reverse w^R of a word $w \in \Sigma^*$ is defined as follows: $\varepsilon^R = \varepsilon$, and $(wa)^R = aw^R$. The *reverse* of a language L is denoted by L^R and defined as $L^R = \{w^R \mid w \in L\}$. The *left quotient* of a language L by a word w is the language $L_w = \{x \in \Sigma^* \mid wx \in L\}$. The *right quotient* of L by w is the language $\{x \in \Sigma^* \mid xw \in L\}$.

It was shown in [1] that closure properties of convex languages can be studied in a common framework of binary relations. We now give some examples.

Example 1. In this example there are no conditions on the partial order $\trianglelefteq$. If $K, L \subseteq \Sigma^*$ are $\trianglelefteq$-convex ($\trianglelefteq$-free, or $\trianglelefteq$-closed), then so is $M = K \cap L$. It follows then that prefix-, suffix-, factor-, and subword-convex classes are closed under intersection, as are the corresponding free and closed versions.

Example 2. Here a condition on the partial order $\trianglelefteq$ is needed. A partial order $\trianglelefteq$ is *factoring* if $x \trianglelefteq y_1y_2$ implies that $x = x_1x_2$ for some $x_1, x_2 \in \Sigma^*$ such that $x_1 \trianglelefteq y_1$, $x_2 \trianglelefteq y_2$. If $\trianglelefteq$ is factoring and K and L are $\trianglelefteq$-closed, then so is KL. One then verifies that $\leqslant$, $\preceq$, $\sqsubseteq$ and $\Subset$ are factoring. From this it follows that the prefix-, suffix-, factor-, and subword-closed classes are closed under product.

Table 1 summarizes the closure results. In case closure holds only for some of the relations, these relations are specified in the table. In [1] it was incorrectly stated that free languages are closed under inverse homomorphism.

Table 1. Closure in classes defined by prefix, suffix, factor, and subword relations

	convex	*closed*	*converse closed*	*free*
intersection	yes	yes	yes	yes
union	no	yes	yes	no
complement	no	no	no	no
product	no	yes	yes	yes
Kleene star	no	yes	no	no
positive closure	no	yes	yes	no
left quotient	prefix subword	prefix subword	prefix subword	prefix subword
right quotient	suffix subword	suffix subword	suffix subword	suffix subword
homomorphism	no	no	no	no
inverse homomorphism	yes	yes	yes	no

4 Complexity of Decision Problems

The results of this section are from [8]. *Regular languages* over an alphabet Σ are languages that can be obtained from the *basic languages* $\emptyset$, $\{\varepsilon\}$, and $\{a\}$, $a \in \Sigma$, using a finite number of operations of union, product and star. Such languages are usually denoted by regular expressions. If E is a regular expression, then $\mathcal{L}(E)$ is the language denoted by that expression. For example, $E = (\varepsilon \cup a)^*b$ denotes $L = (\{\varepsilon\} \cup \{a\})^*\{b\}$. We use the regular expression notation for both expressions and languages. In contrast to other authors, we prefer to use the same operator symbol in expressions as in languages. Thus $\cup$ denotes union, juxtaposition denotes product, and * denotes the star.

A *deterministic finite automaton* (DFA) is a quintuple $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite, non-empty set of *states*, Σ is a finite, non-empty *alphabet*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*.

As usual, a *nondeterministic finite automaton* (NFA) is a quintuple $\mathcal{N} = (Q, \Sigma, \eta, S, F)$, where Q, Σ, and F are as in a DFA, $\eta : Q \times \Sigma \to 2^Q$ is the *transition function*, and $S \subseteq Q$ is the *set of initial states*. If η also allows ε as input, then we call $\mathcal{N}$ an ε-NFA.

The following questions were studied in [8]: If a regular language L is specified by a DFA, how difficult is it to decide whether L is prefix-, suffix-, factor-, or subword-convex, or -free, or -closed?

The most difficult case is that of factor-convexity. To solve this problem we test whether L is *not* factor-convex, in which case there exist $u, v, w \in \Sigma^*$, such that $u \sqsubseteq v \sqsubseteq w$, with $u, w \in L$ and $v \notin L$. Then there exist u', u'', v', v'' such that $v = u'uu''$ and $w = v'vv'' = v'u'uu''v''$.

Suppose $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ is a DFA accepting L. We define an ε-NFA $\mathcal{N} = (Q', \Sigma, \delta', \{q_0'\}, F')$, where $Q' = Q \times Q \times Q \times \{1, 2, 3, 4, 5\}$, $q_0' = [q_0, q_0, q_0, 1]$, $F' = F \times (Q \setminus F) \times F \times \{5\}$, and δ' is defined below. States of $\mathcal{N}$ are quadruples, where components 1, 2, and 3 keep track of the state of $\mathcal{N}$ as it is processing w, v, and u, (respectively). The last component represents the *mode* of operation of $\mathcal{N}$.

The ε-NFA starts with the first three components in the initial state q_0, and the fourth in mode 1. Recall that the input string we are examining has the form $w = v'u'uu''v''$. The automaton $\mathcal{N}$ operates in mode 1 for a while, reading the input only in component 1 using $\delta'([p, q_0, q_0, 1], a) = \{[\delta(p, a), q_0, q_0, 1]\}$, for all $p \in Q, a \in \Sigma$. Then it guesses nondeterministically that it has finished reading v', and switches to mode 2 using $\delta'([p, q_0, q_0, 1], \varepsilon) = \{[p, q_0, q_0, 2]\}$, for all $p \in Q$. In mode 2, component 1 continues to read the input $u'uu''v''$ from the state reached by v', while component 2 reads that input from state q_0 using $\delta'([p, q, q_0, 2], a) = \{[\delta(p, a), \delta(q, a), q_0, 2]\}$, for all $p, q \in Q, a \in \Sigma$. At some point $\mathcal{N}$ guesses that u' has been read and switches to mode 3 using the rule $\delta'([p, q, q_0, 2], \varepsilon) = \{[p, q, q_0, 3]\}$, for all $p, q \in Q$. Now all three components read the input $uu''v'$, the first starting from the state reached by $v'u'$, the second from the state reached by u' and the third from q_0, using $\delta'([p, q, r, 3], a) = \{[\delta(p, a), \delta(q, a), \delta(r, a), 3]\}$, for all $p, q, r \in Q, a \in \Sigma$. A guess is then made using $\delta'([p,q,r,3],\varepsilon) = \{[p,q,r,4]\}$, for all $p,q,r \in Q$, that u has been read. In mode 4, component 3 stops reading since its job was to read u, and that has been done. The first two components continue to read $u''v''$, using $\delta'([p,q,r,4],a) = \{[\delta(p,a),\delta(q,a),r,4]\}$, for all $p,q,r \in Q, a \in \Sigma$. Then $\mathcal{N}$ guesses that u'' has been read, using $\delta'([p,q,r,4],\varepsilon) = \{[p,q,r,5]\}$, for all $p,q,r \in Q$. Now component 2 has finished reading v, and only component 1 continues to read the input v'' in mode 5, to finish processing w; this uses the rule $\delta'([p,q,r,5],a) = \{[\delta(p,a),q,r,5]\}$, for all $p,q,r \in Q, a \in \Sigma$. The input w is accepted by $\mathcal{N}$ if and only if $u,w \in L$ and $v \notin L$. Hence L is factor-convex if and only if $\mathcal{N}$ accepts the empty language.

One verifies that $\mathcal{N}$ has $3n^3 + n^2 + n$ reachable states and $(3|\Sigma| + 2)n^3 + (|\Sigma|+1)(n^2+n)$ transitions, where $|\Sigma|$ is the cardinality of Σ. Since we can test for the emptiness of the language accepted by $\mathcal{N}$ using depth-first search in time linear in the size of $\mathcal{N}$, we can decide if a given regular language L accepted by a DFA with n states is factor-convex in $O(n^3)$ time, assuming $|\Sigma|$ is a constant.

To test for prefix-convexity of L, we construct an ε-NFA $\mathcal{N}$ that checks whether any word of the form $w = uu'u''$ is accepted, where $u,w \in L$ and $v = uu' \notin L$. Then L is prefix-convex if and only if $\mathcal{N}$ accepts the empty language. This construction is very similar to that for factor convexity, and the decision about prefix-convexity can also be done in $O(n^3)$ time.

For suffix convexity of L, we construct $\mathcal{N}$ to test whether $w = u''u'u$ is accepted, where $u,w \in L$ and $v = u'u \notin L$. This can be done in $O(n^3)$ time.

For subword convexity we use an NFA $\mathcal{N} = (Q', \Sigma, \delta', q_0', F')$, where $Q' = Q \times Q \times Q$, $q_0' = [q_0,q_0,q_0]$, $F' = F \times (Q \setminus F) \times F$, and

$$\delta'([p,q,r],a) = \{[\delta(p,a),q,r],\ [\delta(p,a),\delta(q,a),r],\ [\delta(p,a),\delta(q,a),\delta(r,a)]\},$$

for all $p,q,r \in Q$ and $a \in \Sigma$. The test can again be done in $O(n^3)$ time.

The properties of closure, freeness and converse closure can be decided in $O(n^2)$ time using similar methods.

A related problem studied in [8] is to find the length of a shortest word (*witness*) demonstrating that a language is not convex, not closed or not free. The results summarized in Table 2 are best possible, except in the case of subword convexity, where it is not known whether the bound can be reached.

The complexities of the convexity, closure, and freeness decision problems under the assumption that the language is specified by other means are also considered in [8]. For languages specified by NFA's or regular expressions, the

Table 2. Sizes of shortest witnesses denying convexity, closure and freeness

	convexity	closure	freeness
factor	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^2)$
prefix	$2n-1$	n	$2n-1$
suffix	$\Theta(n^3)$	$\Theta(n^2)$	$\Theta(n^2)$
subword	$3n-2$	n	$2n-1$

convexity and closure problems are PSPACE-complete, but for an NFA with n states and t transitions, freeness can be decided in $O(n^2+t^2)$ time. For additional references to these problems, see the bibliography in [8].

5 Quotient Complexity

For a detailed discussion of general issues concerning state and quotient complexity see [5,27] and the reference lists in those papers.

The *quotient complexity* of L is the number of distinct left quotients of L, and is denoted by $\kappa(L)$. From now on we refer to left quotients simply as quotients.

We now describe the computation of quotients of a regular language. First, the *ε-function* L^ε of a regular language L is equal to ε if $\varepsilon \in L$, and to $\emptyset$ otherwise. The quotient by a letter a in Σ is computed by structural induction: $b_a = \emptyset$ if $b \in \{\emptyset, \varepsilon\}$ or $b \in \Sigma$ and $b \neq a$, and $b_a = \varepsilon$ if $b = a$; $(\overline{L})_a = \overline{L_a}$; $(K \cup L)_a = K_a \cup L_a$; $(KL)_a = K_a L \cup K^\varepsilon L_a$; $(K^*)_a = K_a K^*$. The quotient by a word $w \in \Sigma^*$ is computed by induction on the length of w: $L_\varepsilon = L$; $L_w = L_a$ if $w = a \in \Sigma$; and $L_{wa} = (L_w)_a$. Quotients computed this way are indeed the left quotients of L [4,5]. A quotient L_w is *accepting* if $\varepsilon \in L_w$; otherwise it is *rejecting.*

The *quotient automaton* of a regular language L is $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, where $Q = \{L_w \mid w \in \Sigma^*\}$, $\delta(L_w, a) = L_{wa}$, $q_0 = L_\varepsilon = L$, $F = \{L_w \mid L_w^\varepsilon = \varepsilon\}$, and $L_w^\varepsilon = (L_w)^\varepsilon$. So the number of states in the quotient automaton of L is the quotient complexity of L. The *state complexity of a regular language* L is the number of states in the minimal DFA recognizing L. Evidently, the quotient complexity of L is equal to state complexity of L.

The *quotient complexity of a regular operation in* $\mathbb{Q}$ is defined as the worst case size of the quotient automaton for the language resulting from the operation, taken as a function of the quotient complexities of the operands in $\mathbb{Q}$.

Although quotient and state complexities are equal, there are advantages in using quotients. The following formulas [4,5] for quotients of regular languages can be used to establish upper bounds on quotient complexity of operations:

Proposition 2. *If K and L are regular languages, $u, v \in \Sigma^+$, $w \in \Sigma^*$, then*

$$(\overline{L})_w = \overline{L_w}, \quad (K \circ L)_w = K_w \circ L_w, \tag{1}$$

$$(KL)_w = K_w L \cup K^\varepsilon L_w \cup (\bigcup_{w=uv} K_u^\varepsilon L_v), \tag{2}$$

$$(L^*)_\varepsilon = \varepsilon \cup LL^*, \quad (L^*)_w = (L_w \cup \bigcup_{w=uv} (L^*)_u^\varepsilon L_v) L^* \text{ for } w \in \Sigma^+. \tag{3}$$

In the sequel we consider the quotient complexity of operations in various classes of convex languages.

6 Quotient Complexity of Boolean Operations

Table 3 shows the complexities of union and intersection. The results are from the following sources: regular languages [28] (for union of regular languages see also [21]), ideals [6], closed languages [7], prefix-free languages [14], suffix-free languages [13], and the other free languages [9]. Since the complexity of union for all four types of closed languages and for regular languages is mn, we have the same complexity for all four types of convex languages. Similarly, since the complexity of intersection for all four types of ideal languages and for regular languages is mn, we have the same complexity for all four types.

Table 3. Complexities of union and intersection

	$K \cup L$	$K \cap L$
right, two-sided, all-sided ideals	$mn-(m+n-2)$	mn
left ideals	mn	mn
prefix-, factor-, subword-closed	mn	$mn-(m+n-2)$
suffix-closed	mn	mn
prefix-free	$mn-2$	$mn-2(m+n-3)$
suffix-free	$mn-(m+n-2)$	$mn-2(m+n-3)$
bifix-, factor-, subword-free	$mn-(m+n)$	$mn-3(m+n-4)$
convex	mn	mn
regular	mn	mn

Table 4 shows the results for difference and symmetric difference.

Table 4. Complexities of difference and symmetric difference

	$K \setminus L$	$K \oplus L$
right, two-sided, all-sided ideals	$mn-(m-1)$	mn
left ideals	mn	mn
prefix-, factor-, subword-closed	$mn-(n-1)$	mn
suffix-closed	mn	mn
prefix-free	$mn-(m+2n-4)$	$mn-2$
suffix-free	$mn-(m+2n-4)$	$mn-(m+n-2)$
bifix-, factor-, subword-free	$mn-(2m+3n-9)$	$mn-(m+n)$
convex	mn	mn
regular	mn	mn

The sources for the difference operations are: regular languages [5], ideals [6], closed languages [7], and free languages [9]. Since the complexity of symmetric difference for all four types of closed languages and for regular languages is mn, we have the same complexity for all four types.

Since the complexity of difference for suffix-closed languages and for regular languages is mn, we have the same complexity for suffix-convex languages. This leaves the difference of the other three types of convex languages, which we now address.

Proposition 3. *If K and L are prefix-convex (respectively, factor-convex or subword-convex) with $\kappa(K) = m$ and $\kappa(L) = n$, then $\kappa(K \setminus L) \leqslant mn$, and this bound is tight if $|\Sigma| \geqslant 2$.*

Proof. Let $\Sigma = \{a, b\}$, $K = (b^*a)^{m-1}\Sigma^* = \Sigma^* ⧢ a^{m-1}$ and $\overline{L} = (a^*b)^{n-1}\Sigma^* = \Sigma^* ⧢ b^{n-1}$. Then K and $\overline{L}$ are all-sided ideals and L is subword-closed. Thus both K and L are subword-convex, and we know from [6] that $\kappa(K \cap \overline{L}) = mn$; thus $\kappa(K \setminus L) = mn$. □

In summary, the complexities of all four boolean operations in all four classes of convex languages are mn.

7 Complexities of Product, Star and Reversal

Table 5 shows the results for product, star, and reversal. In the product column, k denotes the number of accepting quotients of K. The sources are: regular languages [21,28], ideals [6], closed languages [7], prefix-free languages [14], suffix-free languages [13], and the other free languages [9]. The bounds for convex languages are still open.

Table 5. Complexities of product, star and reversal

	KL	L^*	L^R
right ideals	$m + 2^{n-2}$	$n + 1$	2^{n-1}
left ideals	$m + n - 1$	$n + 1$	$2^{n-1} + 1$
two-sided, all-sided ideals	$m + n - 1$	$n + 1$	$2^{n-2} + 1$
prefix-closed	$(m + 1)2^{n-2}$	$2^{n-2} + 1$	2^{n-1}
suffix-closed	$(m - k)n + k$	n	$2^{n-1} + 1$
factor-, subword-closed	$m + n - 1$	2	$2^{n-2} + 1$
prefix-free	$m + n - 2$	n	$2^{n-2} + 1$
suffix-free	$(m - 1)2^{n-2} + 1$	$2^{n-2} + 1$	$2^{n-2} + 1$
bifix-, factor-, subword-free	$m + n - 2$	$n - 1$	$2^{n-3} + 2$
regular	$m2^n - k2^{n-1}$	$2^{n-1} + 2^{n-2}$	2^n

8 Unary Convex Languages

Because product is commutative for unary languages, they have very special properties. Here, the concepts of prefix, suffix, factor, and subword all coincide. Therefore we can discuss convex unary languages in the terminology of prefix-convex languages. Let $\Sigma = \{a\}$, and suppose that $L \subseteq \Sigma^*$ is prefix-convex. If L is infinite and its shortest word is a^i, then $L = a^i a^*$. If L is finite, then $L = a^i \cup a^{i+1} \cup \cdots \cup a^j$, for $0 \leqslant i \leqslant j$.

If L is a unary right ideal with shortest word a^i, $i \geqslant 0$, then $L = a^i a^*$.

If L is unary and prefix-closed, then it is $L = a^*$, or $L = \{\varepsilon, a, \dots, a^i\}$, $i \geqslant 0$.

If L is unary and prefix-free, then it is $L = a^i$, for some $i \geqslant 0$.

Table 6 shows the complexities of boolean operations. Here $\kappa(K) = m$, and $\kappa(L) = n$. The results for the union and intersection of regular unary languages are from [27]; for difference and symmetric difference see [5]. The bounds for boolean operations on convex languages are all $\max(m,n)$. For example, if $K = a^i \cup a^{i+1} \cup \cdots \cup a^{m-2}$ and $L = a^j \cup a^{j+1} \cup \cdots \cup a^{n-2}$, then $\kappa(K) = m$, $\kappa(L) = n$, and $\kappa(K \cup L) = \max(m,n)$. If $K = a^{m-1}a^*$ and $L = a^{n-1}a^*$, then $\kappa(K) = m$, $\kappa(L) = n$, and $\kappa(K \cap L) = \max(m,n)$.

Table 6. Complexity of boolean operations on unary convex languages

	$K \cup L$	$K \cap L$	$K \setminus L$	$K \oplus L$
unary ideal	$\min(m,n)$	$\max(m,n)$	n	$\max(m,n)$
unary closed	$\max(m,n)$	$\min(m,n)$	m	$\max(m,n)$
unary free	$\max(m,n)$	$m = n$	m	$\max(m,n)$
unary convex	$\max(m,n)$	$\max(m,n)$	$\max(m,n)$	$\max(m,n)$
unary regular	mn	mn	mn	mn

The complexities of product, star and reversal are given in Table 7. The results for these operations on regular unary languages are from [28]. The bound for the star of a unary convex language is derived below.

Table 7. Complexity of product, star and reversal on unary convex languages

	KL	L^*	L^R
unary ideal	$m+n-1$	$n-1$	n
unary closed	$m+n-2$	2	n
unary free	$m+n-2$	$n-2$	n
unary convex	$m+n-1$	$n^2-7n+13$	n
unary regular	mn	n^2-2n+2	n

Proposition 4. *If L is a unary convex language with $\kappa(L) = n$, then $\kappa(L^*) \leqslant n^2 - 7n + 13$, and the bound is tight if $n \geqslant 5$.*

Proof. If $L = \emptyset$, then $\kappa(L^*) = 2$.

If L is infinite, then it has the form $L = a^{n-1}a^*$, and $\kappa(L) = n$. If $n = 1$, then $L = a^* = L^*$, and $\kappa(L^*) = 1$. If $n = 2$, then $L = aa^*$ and $L^* = a^*$ again. For $n > 2$, $L^* = \varepsilon \cup a^{n-1}a^*$, and $\kappa(L^*) = n$.

Now consider the case where L consists of only one word. If $L = \varepsilon$, then $\kappa(L) = 2$, $L^* = \varepsilon$, and $\kappa(L^*) = 2$. If $L = a$, then $\kappa(L) = 3$, $L^* = a^*$, and $\kappa(L^*) = 1$. If $L = a^{n-2}$, for $n \geqslant 4$, then $\kappa(L) = n$, $L^* = (a^{n-2})^*$, and $\kappa(L^*) = n - 2$.

Next suppose that L is finite, and contains at least two words. Then L has the form $L = a^i \cup a^{i+1} \cup \cdots \cup a^{n-2}$. If $a \in L$, then $L^* = a^*$, and $\kappa(L^*) = 1$.

Hence assume that $i \geqslant 2$, which implies that $n \geqslant 5$. The integers i and $j = i+1$ are relatively prime, and the largest integer k that cannot be expressed as a non-negative linear combination of i and j is the Frobenius number [23] of i and j, which is $ij - i - j = i^2 - i - 1$. This means that $a^{i^2-i-1} \notin (a^i \cup a^j)^* \subseteq L^*$, and $a^{i^2-i}a^* \subseteq L^*$. If we add words of length greater than j, the length of the longest word not in L^* can only decrease. Hence the worst case for the complexity of L^* occurs when $L = a^{n-3} \cup a^{n-2}$. Here $\kappa(L^*) = (n-3)(n-2) - (n-3+n-2) + 2 = n^2 - 7n + 13$. □

9 Closed and Ideal Language Classes

Let $\mathbb{L}$ be the class of left ideals, and let $\mathbb{L}_\emptyset = \mathbb{L} \cup \{\emptyset\}$. The class $\mathbb{L}_\emptyset$ has been studied by Paz and Peleg [22] under the name *ultimate definite events*. They observed that, if I is some index set and L_i are left ideals, then $\bigcup_{i \in I} L_i$ and $\bigcap_{i \in I} L_i$ left ideals as well. They also showed the following:

Proposition 5. *The algebra* $\mathcal{L} = (\mathbb{L}_\emptyset, \cup, \cap, \emptyset, \Sigma^*)$ *is a complete lattice with least element* $\emptyset$ *and greatest element* Σ^*. *Moreover,* $(\mathbb{L}_\emptyset, \cdot, \emptyset)$ *is a semigroup with zero* $\emptyset$, *and* $\mathbb{L}_\emptyset$ *is closed under positive closure.*

Now let $\mathbb{S}$ be the class of suffix-closed languages. Then we have:

Proposition 6. *The algebra* $\mathcal{L}' = (\mathbb{S}, \cap, \cup, \Sigma^*, \emptyset)$ *is a complete lattice, and it is isomorphic to* $\mathcal{L}$, *with complementation acting as the isomorphism. Moreover, the algebra* $(\mathbb{S}, \cdot, \{\varepsilon\}, \emptyset)$ *is a monoid with unit* $\{\varepsilon\}$ *and zero* $\emptyset$, *and* $\mathbb{S}$ *is closed under star.*

Proposition 7. *The algebra* $\mathcal{L}'' = (\mathbb{L}_\emptyset \cup \mathbb{S}, \cdot, \{\varepsilon\}, \emptyset)$ *is a monoid with unit* $\{\varepsilon\}$ *and zero* $\emptyset$, *and* $\mathbb{L}_\emptyset \cup \mathbb{S}$ *is closed under complementation.*

Proof. We need only verify that $\mathcal{L}''$ is closed under product. First, suppose that one of K and L is $\emptyset$; then $KL = \emptyset$ and $\emptyset$ is in $\mathbb{L}_\emptyset \cap \mathbb{S}$. Hence assume that K and L are non-empty. Because we know that $\mathbb{L}$ and $\mathbb{S}$ are closed under product, we only need to consider the cases KL, where K is a left ideal and L is suffix-closed or *vice versa*. In the first case, $KL = \Sigma^* KL$ is a left ideal. In the second case, since K is closed, it contains ε; thus $KL \supseteq L$. But we also have $KL \subseteq \Sigma^* L = L$. Hence $KL = L = \Sigma^* L$ is a left ideal. □

If $\mathbb{R}$ is the class of right ideals, let $\mathbb{R}_\emptyset = \mathbb{R} \cup \{\emptyset\}$, and let $\mathbb{P}$ be the class of prefix-closed languages. Then results analogous to Propositions 5–7 also hold for the algebras $(\mathbb{R}_\emptyset, \cup, \cap, \emptyset, \Sigma^*)$, $(\mathbb{R}_\emptyset, \cdot, \emptyset)$, $(\mathbb{P}, \cap, \cup, \Sigma^*, \emptyset)$, $(\mathbb{P}, \cdot, \{\varepsilon\}, \emptyset)$, and $(\mathbb{R}_\emptyset \cup \mathbb{P}, \cdot, \{\varepsilon\}, \emptyset)$. Similar statements hold for the class $\mathbb{T}$ of two-sided ideals with the class $\mathbb{F}$ of factor-closed languages, and the class $\mathbb{A}$ of all-sided ideals with the class $\mathbb{W}$ of subword-closed languages.

The close relationship between ideals and closed languages is reflected in quotient complexity. If we know the complexity $\kappa(K \cap L)$, where K and L are ideals, then we also know $\kappa(K \cap L) = \kappa(\overline{\overline{K} \cup \overline{L}}) = \kappa(\overline{K} \cup \overline{L})$, where $\overline{K}$ and $\overline{L}$ are closed

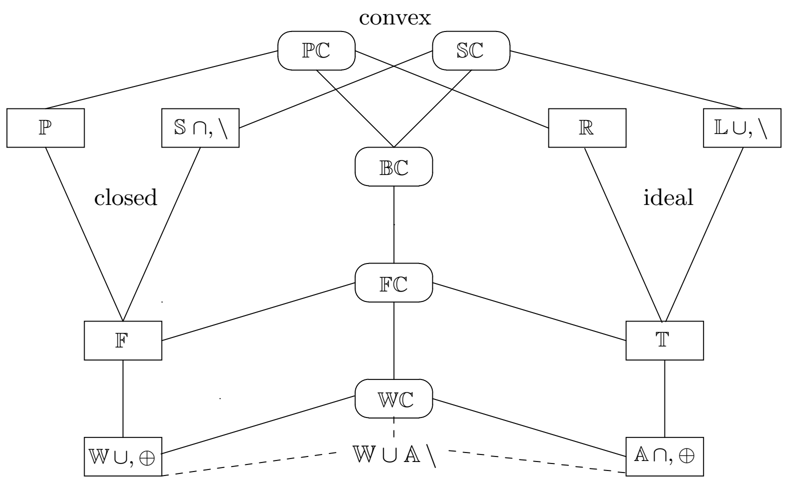


Fig. 1. Maximal complexities for boolean operations on convex languages

languages, and vice versa. Similar statements hold for the other boolean operations. Also, since reversal commutes with complementation, the complexities of the reversal of ideal languages are identical to those of closed languages. In Fig. 1, $\mathbb{PC}$, $\mathbb{SC}$, $\mathbb{BC}$, $\mathbb{FC}$, and $\mathbb{WC}$ stand for prefix-, suffix-, bifix-, factor-, and subword-convex languages, respectively.

The figure shows how the maximal bound mn for boolean operations can be reached in the convex hierarchy. For symmetric difference, it is reached at the bottom of both the closed and the ideal hierarchies. For union, it is reached in subword-closed languages in the closed hierarchy, but we have to go as high as left ideals in the ideal hierarchy. Intersection is dual to union, as shown in the figure. For difference, we can either go as high as suffix-closed languages or left ideals, or we can reach the bound in $\mathbb{W}\cup\mathbb{A}$, as witnessed by the languages in the proof of Proposition 3.

Since star and reversal are unary, the complexity of each of these operations in the union of a closed class with the corresponding ideal class is the maximum of the complexities in the closed and the ideal classes. This also holds for the product, as we now show.

Proposition 8. *If $K \subseteq \Sigma^*$ and $L \subseteq \Sigma^*$ are languages with $\kappa(K) = m$, $\kappa(L) = n$, and k is the number of accepting quotients of K; then the following hold:*

1. *If $K, L \in \mathbb{L}_\emptyset \cup \mathbb{S}$, then $\kappa(KL) \leqslant (m-k)n + k$, and the bound is tight in $\mathbb{S}$.*
2. *If $K, L \in \mathbb{R}_\emptyset \cup \mathbb{P}$, then $\kappa(KL) \leqslant (m+1)2^{n-2}$, and the bound is tight in $\mathbb{P}$.*
3. *If $K, L \in \mathbb{T}_\emptyset \cup \mathbb{F}$, then $\kappa(KL) \leqslant m+n-1$, and the bound is tight in $\mathbb{T}$ and $\mathbb{F}$.*
4. *If $K, L \in \mathbb{A}_\emptyset \cup \mathbb{W}$, then $\kappa(KL) \leqslant m+n-1$; the bound is tight in $\mathbb{A}$ and $\mathbb{W}$.*

Proof. The case when one of K and L is empty is trivial.

1. Consider the product of a left ideal K with a suffix-closed language L. It was shown in [7] that, for any language K and a suffix-closed language L, $\kappa(KL) \leqslant (m-k)n+k$, and that this bound is met by K and L that are both suffix-closed.

   In case K is suffix-closed and L is a left ideal, then $KL = L$, as we have shown in the proof of Proposition 7. Hence $\kappa(KL) = n$, which is less than the complexities in $\mathbb{L}$ and $\mathbb{S}$.
2. If K is a right ideal and L is prefix-closed, then $KL = K$ by an argument similar to that in the proof of Proposition 7. So $\kappa(KL) = m$, which is smaller than the bounds in $\mathbb{R}$ and $\mathbb{P}$.

   If K is prefix-closed and L is a right ideal, then $\varepsilon \in K$. Note that a prefix-closed language is either Σ^* or has $\emptyset$ as a quotient. In the first case, $KL = \Sigma^* L \Sigma^*$ and $\kappa(KL) \leqslant 2^{n-2}+1$ [6], which is smaller than the bounds in $\mathbb{R}$ and $\mathbb{P}$. Now assume that $K \neq \Sigma^*$, which implies that K has $m-1$ accepting quotients and $\emptyset$.

   If K_w is accepting, then so is K_u for every prefix u of w. From Equation (2), $(KL)_w = K_w L \cup L_w \cup (\bigcup_{w=uv} L_v)$. Then each quotient of KL is determined by a quotient of K and a union of quotients of L that always contains L. Because L has Σ^* as a quotient, 2^{n-1} unions are equal to Σ^*. Thus for each accepting quotient of K we have 2^{n-2} possible unions plus Σ^*. Altogether there are at most $(m-1)2^{n-2}+1$ quotients of KL of this type.

   If $K_w = \emptyset$ is the one rejecting quotient of K, then $(KL)_w$ is a union of quotients of L, which is non-empty because of L_w. Since unions that contain Σ^* have already been taken into account, we have $(2^{n-1}-1)$ additional quotients, for a total of at most $(m+1)2^{n-2}$. This bound is reached in $\mathbb{P}$ [7].
3. If K is a two-sided ideal and L is factor-closed, then $KL = K$. If K is factor-closed and L is a two-sided ideal, then $KL = L$. In either case, this bound is lower than the bound in $\mathbb{T}$ and $\mathbb{F}$.
4. The argument of the last case works here as well. □

Given any language L of complexity n, it is natural to ask what is the worst-case complexity of the prefix-closure of L and the converse prefix-closure of L, which is the right ideal generated by L. The same questions apply to the other closed languages. The results [6,18] for these two closures are summarized in Table 8.

The complexities of ideals in terms of their minimal generators and of minimal generators in terms of ideals are also discussed in [6].

Table 8. Complexities of closure and converse closure

	closure	converse closure
prefix relation	n	n
suffix relation	2^n-1	2^{n-1}
factor relation	2^{n-1}	$2^{n-2}+1$
subword relation	$2^{n-2}+1$	$2^{n-2}+1$

10 Conclusions

It has been demonstrated that the class of convex languages and its subclasses are interesting from several points of view, and that studying these classes together is very worthwhile.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada under Grant No. OGP000871.

I am grateful to Galina Jirásková for a very careful reading of the manuscript, and to Jeff Shallit for the reference to Frobenius numbers.

References

1. Ang, T., Brzozowski, J.: Languages convex with respect to binary relations, and their closure properties. Acta Cybernet. 19(2), 445–464 (2009)
2. Berstel, J., Perrin, D.: Theory of Codes. Academic Press, London (1985)
3. Berstel, J., Perrin, D., Reutenauer, C.: Codes and Automata. Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge (2010)
4. Brzozowski, J.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
5. Brzozowski, J.: Quotient complexity of regular languages. In: Dassow, J., Pighizzini, G., Truthe, B. (eds.) Proceedings of the 11th International Workshop on Descriptional Complexity of Formal Systems, Magdeburg, Germany, Otto-von-Guericke-Universität, pp. 25–42 (2009), Extended abstract at http://arxiv.org/abs/0907.4547
6. Brzozowski, J., Jirásková, G., Li, B.: Quotient complexity of ideal languages. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 208–221. Springer, Heidelberg (2010)
7. Brzozowski, J., Jirásková, G., Zou, C.: Quotient complexity of closed languages. In: Proceedings of the 5th International Computer Science Symposium in Russia, CSR. LNCS, Springer, Heidelberg (to appear, 2010), http://arxiv.org/abs/0912.1034
8. Brzozowski, J., Shallit, J., Xu, Z.: Decision problems for convex languages. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 247–258. Springer, Heidelberg (2009)
9. Brzozowski, J., Smith, J.: Quotient complexity of bifix-, factor-, and subword-free languages (In preparation)
10. Câmpeanu, C., Culik II, K., Salomaa, K., Yu, S.: State complexity of basic operations on finite languages. In: Boldt, O., Jürgensen, H. (eds.) WIA 1999. LNCS, vol. 2214, pp. 60–70. Springer, Heidelberg (2001)
11. Gill, A., Kou, L.T.: Multiple-entry finite automata. J. Comput. Syst. Sci. 9(1), 1–19 (1974)
12. Haines, L.: On free monoids partially ordered by embedding. J. Combin. Theory 6(1), 94–98 (1969)
13. Han, Y.S., Salomaa, K.: State complexity of basic operations on suffix-free regular languages. Theoret. Comput. Sci. 410(27-29), 2537–2548 (2009)

14. Han, Y.S., Salomaa, K., Wood, D.: Operational state complexity of prefix-free regular languages. In: Ésik, Z., Fülöp, Z. (eds.) Automata, Formal Languages, and Related Topics, pp. 99–115. University of Szeged, Hungary (2009)
15. Higman, G.: Ordering by divisibility in abstract algebras. Proc. London Math. Soc. 3(2), 326–336 (1952)
16. Jürgensen, H., Konstantinidis, S.: Codes. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, Word, Language, Grammar, vol. 1, pp. 511–607. Springer, Heidelberg (1997)
17. Jürgensen, H., Yu, S.S.: Relations on free monoids, their independent sets, and codes. Internat. J. Comput. Math. 40, 17–46 (1991)
18. Kao, J.Y., Rampersad, N., Shallit, J.: On NFAs where all states are final, initial, or both. Theoret. Comput. Sci. 410(47-49), 5010–5021 (2009)
19. Kruskal, J.B.: The theory of well-quasi-ordering: A frequently discovered concept. J. Combin. Theory A 13(3), 297–305 (1972)
20. de Luca, A., Varricchio, S.: Some combinatorial properties of factorial languages. In: Capocelli, R. (ed.) Sequences: Combinatorics, Compression, Security, and Transmission, pp. 258–266. Springer, Heidelberg (1990)
21. Maslov, A.N.: Estimates of the number of states of finite automata. Dokl. Akad. Nauk SSSR 194, 1266–1268 (1970) (Russian); English translation: Soviet Math. Dokl. 11, 1373–1375 (1970)
22. Paz, A., Peleg, B.: Ultimate-definite and symmetric-definite events and automata. J. ACM 12(3), 399–410 (1965)
23. Ramírez Alfonsín, J.L.: The Diophantine Frobenius Problem. Oxford Lectures Series in Mathematics and its Applications, vol. 30. Oxford University Press, Oxford (2005)
24. Shyr, H.J.: Free Monoids and Languages. Hon Min Book Co., Taiwan (2001)
25. Shyr, H.J., Thierrin, G.: Hypercodes. Information and Control 24, 45–54 (1974)
26. Thierrin, G.: Convex languages. In: Nivat, M. (ed.) Automata, Languages and Programming, pp. 481–492. North-Holland, Amsterdam (1973)
27. Yu, S.: State complexity of regular languages. J. Autom. Lang. Comb. 6, 221–234 (2001)
28. Yu, S., Zhuang, Q., Salomaa, K.: The state complexities of some basic operations on regular languages. Theoret. Comput. Sci. 125(2), 315–328 (1994)

Three Learnable Models for the Description of Language

Alexander Clark

Department of Computer Science,
Royal Holloway, University of London
Egham TW20 0EX
alexc@cs.rhul.ac.uk

Abstract. Learnability is a vital property of formal grammars: representation classes should be defined in such a way that they are learnable. One way to build learnable representations is by making them objective or empiricist: the structure of the representation should be based on the structure of the language. Rather than defining a function from representation to language we should start by defining a function from the language to the representation: following this strategy gives classes of representations that are easy to learn. We illustrate this approach with three classes, defined in analogy to the lowest three levels of the Chomsky hierarchy. First, we recall the canonical deterministic finite automaton, where the states of the automaton correspond to the right congruence classes of the language. Secondly, we define context free grammars where the non-terminals of the grammar correspond to the syntactic congruence classes, and where the productions are defined by the syntactic monoid; finally we define a residuated lattice structure from the Galois connection between strings and contexts, which we call the syntactic concept lattice, and base a representation on this, which allows us to define a class of languages that includes some non-context free languages, many context-free languages and all regular languages. All three classes are efficiently learnable under suitable learning paradigms.

1 Introduction

"Formal language theory was first developed in the mid 1950's in an attempt to develop theories of natural language acquisition."

This statement, in [18], may not be entirely accurate historically, but the point it makes is valid: language theory has its roots in the modelling of learning and of language. The very name of language theory betrays its origins in linguistics. Yet it has moved far from its origins – formal language theory is now an autonomous part of computer science, and only a few papers at the major conferences in Formal Language Theory (FLT) are directly concerned with linguistics. From time to time, the requirements of linguistics impinge on FLT – for example, the discovery of non-context free natural languages [32] inspired much study in mildly context sensitive grammar formalisms; Minimalist grammars are a direct offshoot of interest in Chomsky's Minimalist Program [5]; and so on.

Learnability is another aspect which has been put to one side. As a model of language acquisition, the original intention was for phrase-structure grammars to be learnable. The PSGs were meant to represent, at a suitable level of abstraction, the linguistic knowledge of the language that the child learner could infer from his or her exposure to the ambient linguistic environment. Chomsky [6] says (p. 172, footnote 15):

> The concept of "phrase structure grammar" was explicitly designed to express the richest system that could reasonable be expected to result from the application of Harris-type procedures to a corpus. . . .

The "Harris-type procedures" refer to the methods of distributional learning developed and partially formalised by Zellig Harris [17] following on the ideas of earlier American structuralist linguists. So PSGs in general, and CFGs in particular, were intended, were *designed*, to be learnable by distributional methods — but they weren't. Even the class of regular grammars, equivalent to non-deterministic finite state automata, is not learnable under very easy learning schemes [2]. This is not a problem for distributional methods, but rather a problem with these formalisms. The natural question is therefore whether there are other formalisms, different from the Chomsky hierarchy, that are learnable.

Before we answer this question it is a good idea to be clear about what we mean by learning, and why we think it is so important. Informally learning means that we want to construct our representations for the language from information about the language. From a formal point of view, this means that we wish to define representations, and algorithms for constructing those representations from a source of information about the language, and prove, under a suitable regime, that these algorithms will converge to the right answer. Given a language L, we want to prove that the hypothesised representation will converge to a representation G such that the language defined by G is equal to L. Our focus in this paper is on the computational complexity of learning and so we will assume that we have a very good source of information. We will assume that we have a source of examples of strings from the language, and that additionally the algorithm can query whether a particular string is in the language or not. In the terminology of grammatical inference we have positive data and membership queries. Under some assumptions this is sufficient to learn any class of languages if we neglect computational issues – here we will be considering algorithms that are efficient. For simplicity of exposition we will use a slightly inadequate formalisation: we will merely require that the algorithms use a polynomial amount of computation, in the size of the observed data, at each step of the algorithm.

Of course, not all potential application areas of language theory are linguistic. Linguistics might be considered a special case, where the representations are unknown, as they are cognitive structures and as yet we cannot hope to probe their natures directly. But it is worth considering where the grammars or representations of the languages come from in various other domains. There are broadly speaking two cases: one where we have information about the language, but not about the representation, and the second is where we have direct information about the representation. In almost all the engineering cases we are interested in, we will have some data that we want to model; in computational biology these might be strings of bases or amino acids, or in other areas they might represent sequences of actions by a robot or a human, or any other sequence of events. But we do not know what the representation is – in none of these cases do we have any direct information about the representation. Indeed, there is no "right" representation, unlike in linguistics. There are merely models that fit the data to a greater or lesser extent.

The only situation where this is not the case is where the language is a programming language or mark up language – in that case, we know what the structure of the language is. In almost all other cases, we will be using languages to represent some sequence of symbols or events, where the representation class is unknown. In these cases, just as in the case of linguistics, learnability is not a tangential property – it is absolutely essential. A representation without some plausible story about how it can be acquired from data is of limited interest. Efficient learning should, in our view, be as important a property of a representation class as efficient parsing.

One way of modelling the data is for a domain expert to construct it by hand. If they are written by a linguist then there are a number of clear desiderata. The grammars should be concise and make it easy for the linguist to express whatever generalisations he or she wishes; it should be easy for humans to reason with consciously, and ideally have a nice graphical representation, perhaps as a tree. Modularity is also a useful property. Following this path we end up with grammatical formalisms being effectively just special-purpose declarative programming languages: DATR [14] and to a lesser extent HPSG [29] are examples of this. There is no reason to think that such formalisms could be or should be learnable. If, on the other hand, we consider the grammar to be the output of a learning process, it need not be particularly convenient for humans to reason about consciously. The properties of a formal grammar are thus radically different, even diametrically opposed depending on whether we envisage them being produced manually or automatically by a process of inference.

2 How

If we accept this argument, then the goal becomes clear – we should construct representations that are intrinsically learnable. Putting this as a slogan: "Put learnability first!" We should design representations from the ground up to be learnable.

Here we present a number of different ways of doing this that follow the same basic strategy: they are objective or "empiricist". We define the representational primitives of the formalism in a language theoretical way. The basic elements of the formalism, whether they are states in an automaton, or non-terminals in a phrase structure grammar, must have a clear definition in terms of sets of strings, in a way that does not depend on the representation.

Rather than defining a representation, and then defining a function from the representation to the language, we should go backwards. If we are interested first in learnability, then we should start by defining the map from the language to the representation. For example, in a CFG, we define a derivation relation $\overset{*}{\Rightarrow}$; for each non-terminal N we define the set of strings that can be derived from that non-terminal: $Y(N) = \{w | N \overset{*}{\Rightarrow} w\}$. We then define the language as the set of strings derived from the start symbol or symbols. Thus we define a function from a non-terminal N, to a set of strings $Y(N)$, and thus from the set of context free grammars to the set of context free languages: from G to $L(G)$.

There is however an insurmountable obstacle to going in the reverse direction: $Y(N)$ is almost completely unconstrained. Suppose we have some context free language L, and a grammar G such that $L(G) = L$ and N is a non-terminal in G. What constraints are there on $Y(N)$? We can say literally nothing about this set, other than that it is a context free language. If we restrict the CFG so that all non-terminals are accessible, then it must be a subset of the set of substrings of L, but beyond that we can't say anything. Consider for example the degenerate case when L is Σ^*. There are clearly many CFGs with one non-terminal that define this language, but there are also infinitely many other ones, with non-terminals that correspond to arbitrary context-free subsets of Σ^*. We need some way of identifying some relevant subsets that will correspond to the primitive elements of our representation. We should start by defining some suitable sets of strings; only then can we construct a grammar. If we define some set of strings then the structure of the representation will follow: given an objective definition of the "reference" of the non-terminal or symbol, the derivation process will be fixed.

We will define three representations; we will start with a basic representation that is quite standard, and turns out to be equivalent to a subclass of DFAs; indeed to the canonical automata. In this case we base the representation on the right congruence classes, as used in the Myhill-Nerode theorem. The next step is to define context free grammars where the non-terminals correspond to the syntactic congruence classes of the language; the third and final representation uses what we call the syntactic concepts of the language – elements of a residuated lattice – as the representational primitives, and the resulting formalism can represent some non-context-free languages.

3 Canonical DFA

We will start by considering a classic case [16,4,11], where the learnability problems are well understood: the case of regular languages. We will end up with a class of representations equivalent to a subclass of deterministic finite automata. We present this standard theory in a slightly non-standard way in order to make the step to the next class of representations as painless as possible.

We will define our notation as needed; we take a finite non-empty aphabet Σ, and the free monoid Σ^*; we use λ to refer to the empty string. A language is a subset of Σ^*. Let L be some arbitrary language; not necessarily regular, or even computable. We define the residual language of a given string u as $u^{-1}L = \{w : uw \in L\}$.

Consider the following relation between strings: $u \sim_L v$ iff $u^{-1}L = v^{-1}L$. This is an equivalence relation and additionally a right congruence: if $u \sim_L v$ then for all $w \in \Sigma^*$, $uw \sim_L vw$.

We can consider the equivalence classes under this relation: we will write $[u]^R$ for the congruence class of the string u under this right congruence. It is better to consider these classes not just as sets of strings but as pairs $\langle P, S\rangle$ where P is a congruence class, and S is the residual language of all the strings in P. That is to say we will have elements of the form $\langle [u]^R, u^{-1}L\rangle$. One important such element is $\langle [\lambda]^R, L\rangle$.

Suppose we define a representation that is based on these congruence classes. Let us call these primitive elements of our representation *states*. The state $\langle [\lambda]^R, L\rangle$ we will denote by q_0. A few elementary observations: if $u \in L$ then every element of $[u]^R$ is also in L. We will call a state $\langle P, S\rangle$ such that $\lambda \in S$ a *final* state. Thus if we can tell for each string in the language which congruence class it is in, then we will have predicted the language. Our representation will be based on this idea: we will try to compute for each string w not just whether it is in L but which congruence class it is in.

We have now taken the first step: we have defined the primitive elements of the representation. We now need to define a derivation of some sort by exploiting the algebraic structure of these classes, in particular the fact that they are right congruence classes. Since it is a right congruence, if we have a string that we know is in the congruence class $[u]^R$ and we append the string v we know that it will be in the class $[uv]^R$. Thus we have a "transition" from the state $[u]^R$ to the state $[uv]^R$ labelled with the string v. It is clear that we can restrict ourselves to the case where $|v| = 1$, i.e. where the transitions are labelled with letters. We now have something that looks very like an automaton. We let Q be the, possibly infinite set of all these states, q_0 the initial state, δ a transition function defined by $\delta([u]^R, a) = [ua]^R$, and let F be the set of final states $\{[u]^R | u \in L\}$. We now define $\mathfrak{R}(L)$ to be this, possibly infinite, representation. We have defined a function from L to $\mathfrak{R}(L)$.

We extend δ recursively in the standard way and then define the function from the representation to the language $L(\mathfrak{R}(L)) = \{w : \delta(q_0, w) \in F\}$. Given this, we have that for any language L, $L(\mathfrak{R}(L)) = L$. In a strict sense, this "representation" is correct. Of course we are interested in those cases where we have a finite representation, and $\mathfrak{R}(L)$ will be finite if and only if L is regular, by the Myhill-Nerode theorem. Thus while we have that for any language this representation is correct, we can only finitely represent the class of regular languages.

It is possible to infer these representations for regular languages, using a number of different techniques depending on the details of the source of information that one has about the language. The recipe is then as follows:

– Define a set of primitive elements language theoretically – in this case the right congruence classes: $[u]^R$.
– Identify a derivation relation of some sort based on the algebraic structure of these elements: $[u]^R \rightarrow^a [v]^R$.

– Construct a representation $\mathfrak{R}(L)$ based on the language and prove that it is correct $L(\mathfrak{R}(L)) = L$.
– Identify the class of languages that can be finitely represented in this way: the class of regular languages. Happily, in this case, it coincides with an existing class of languages.

4 CFGs with Congruence Classes

We can now move on from representations that are regular to ones that are capable of representing context-free languages. We do this using the idea of distributional learning. These techniques were originally described by structuralist linguists who used them to devise mechanical procedures for discovering the structure of natural languages. As such, they are a reasonable starting point for investigations of learnable representations.

Some notation and basic terminology: we define a context to be a pair of strings (l, r) where $l, r \in \Sigma^*$. We combine a context with a substring so $(l, r) \odot u = lur$; We will sometimes refer to a context (l, r) with a single letter f. A string u occurs in a context (l, r) in a language if $lur \in L$. If L, R are sets of strings then we use (L, R), (L, r) etc to refer to the obvious sets of contexts: $L \times R$, $L \times \{r\}$ and so on. We define the distribution of a string in a language as the set of all contexts that it can occur in: $C_L(w) = \{(l, r)|lur \in L\}$. We will extend the notation $\odot$ to contexts: $(l, r) \odot (x, y) = (lx, yr)$, so $(f \odot g) \odot u = f \odot (g \odot u)$. We will also use it for sets in the obvious way.

We now define the crucial notion for this approach: two strings u and v are syntactically congruent iff they have the same distribution: $u \equiv_L v$, iff $C_L(u) = C_L(v)$. We write $[u]$ for the congruence class of u. We note here a classic result – the number of congruence classes is finite if and only if the language is regular.

Given the discussion in the previous section we hope that it is obvious what the next step is: our primitive elements will correspond to these congruence classes. Immediately this seems to raise the problem that we will be restricted to regular languages, since we are interested in finite representations, and thus can only represent a finite number of congruence classes. This turns out not to be the case, as we shall see.

Clearly the empty context (λ, λ) has a special significance: this context is in the distribution of a string if and only if that string is in the language; $(\lambda, \lambda) \in C_L(u)$ means that $u \in L$. So if we can predict the congruence class of a string, we will know whether the string is in the language or not. Given this fixed interpretation of these symbols, we can now proceed to determine what the appropriate derivation rules should be. We have mentioned that this is a congruence: this means that for all strings u, v, x, y if $u \equiv_L v$ then $xuy \equiv xvy$. This means that if we take any element of $[u]$ say u' and any element of $[v]$ say v', and concatenate them, then the result $u'v'$ will always be in the same congruence class as $[uv]$. This means that if we want to generate an element of $[uv]$ we can do this by generating an element from $[u]$ and then generating an element from $[v]$ and then concatenating the results. In other words, we have a context free production $[uv] \rightarrow [u][v]$. Additionally we know that for any string w we can generate an element of $[w]$ just by producing the string w. Given the previous productions, it is clearly sufficient just to have these productions for strings of length 1: i.e. to have productions $[a] \rightarrow a$.

Another way of viewing this is to note that the concatenation of the congruence classes is well defined – or alternatively that since the relation $\equiv_L$ is a monoid congruence, we can use the quotient monoid $\Sigma^*/\equiv_L$ which is the well known syntactic monoid. The production rules then can be viewed as saying that $X \rightarrow YZ$ is a production if and only if $X = Y \circ Z$, and that $X \rightarrow a$ iff $a \in X$. Here, we can see even more clearly that the structure of the representation is based on the structure of the language – in this case we have a CFG-like formalism that corresponds exactly to the syntactic monoid of the language. We therefore define our representation as follows: $\mathfrak{C}(L)$ consists of the, possibly infinite, set of congruence classes $[u]$ together with a set of productions consisting of $\{[uv] \rightarrow [u], [v] | u, v \in \Sigma^*\}$ and $\{[a] \rightarrow a | a \in \Sigma\}$ and $[\lambda] \rightarrow \lambda$. We identify a set of initial symbols $I = \{[u] | u \in L\}$, and we define derivation exactly as in a context free grammar.

It is then easy to see that given these productions, $[w] \overset{*}{\Rightarrow} v$ iff $v \in [w]$. Thus the productions are well behaved. We then define $L(\mathfrak{C}(L)) = \{w | \exists N \in I \text{ such that } N \overset{*}{\Rightarrow} w\}$. We can then prove that $L(\mathfrak{C}(L)) = L$, for any language L.

We have used the two schemas $[uv] \rightarrow [u][v]$ and $[a] \rightarrow a$; these are sufficient. But it is conceivable that we might want to have different schemas – we can have schemas like $[w] \rightarrow w$, $[lwr] \rightarrow l[w]r$, $[aw] \rightarrow a[w]$ or even $[uvw] \rightarrow [u][v][w]$, which will give us finite grammars, linear grammars, regular grammars and so on. All of these schemas maintain the basic invariant that they will only derive strings of the same congruence class.

We thus end up with something that looks something like a context free grammar in Chomsky normal form. It differs in two respects, one trivial and one extremely important. The trivial difference is that we may have more than one start symbol: we wish to maintain the nice map between the representation and the structure.

The second point is that the number of congruence classes will be infinite, if the language is not regular. Consider the language $L_{ab} = \{a^n b^n | n \geq 0\}$. This is a non-regular context free language. It is easy to see that we have an infinite number of congruence classes since a^i is not congruent to a^j unless $i = j$. It appears therefore that we have only achieved another representation for regular languages. We can consider this as the *canonical context free grammar* for a regular language.

Let us suppose we maintain the structure of the representation but only take a finite set of congruence classes V consisting of the classes corresponding to a finite set of strings K: $V = \{[u] | u \in K\}$. We will assume that K contains Σ and λ and finitely many others strings. The binary productions will thus be limited to the finite set $\{[uv] \rightarrow [u][v] | [u], [v], [uv] \in V\}$. This will then give us a finite representation, which we denote $\mathfrak{C}(L, K)$ We can prove that if we have only a subset of the productions, then $[w] \overset{*}{\Rightarrow} v$ implies $v \in [w]$, and therefore our representation will always generate a subset of the correct language: $L(\mathfrak{C}(L, K)) \subseteq L$.

The class that we can represent is therefore the set of all languages L such that there is some finite set of strings K such that $\mathfrak{C}(L, K)$ defines the correct language.

$$\mathcal{L}_{\mathrm{CCFG}} = \{L | \exists \text{ finite } K \subset \Sigma^* \text{ such that } L(\mathfrak{C}(L, K)) = L\}$$

This class clearly includes the class of regular languages. It also includes some non-regular context free languages. In the case of our example L_{ab} it is sufficient to have the following congruence classes: $[a], [b], [\lambda], [ab], [aab], [abb]$. Not all context free languages can be described in this way. The context free language $\{a^n b^m | n < m\}$ is not in $\mathcal{L}_{\mathrm{CCFG}}$, as the language is the union of an infinite number of congruence classes. $\mathcal{L}_{\mathrm{CCFG}}$ is therefore a proper subclass of the class of context free languages; by restricting the non-terminals to correspond exactly to the congruence classes, we lose a bit of representational power, but we gain efficient learnability. Note the very close relationship to the class of NTS languages [30]; indeed we conjecture that these classes may be equal. The first results on learning using this approach [9,7,10] have shown that various subclasses can be learned under various non-probabilistic and probabilistic paradigms. Note that we have lost the canonical nature of the formalism – there will often be more than one possible minimal choice of K or V. Nonetheless, given that the definition of the primitives is fixed this is not a problem: any sufficiently large set of congruence classes will suffice to define the same language.

4.1 Regular Languages

Let us briefly return to the case of regular languages. We know that the set of congruence classes is finite, but we can get some insight into the structure of this set by looking at the proof. Let A be the minimal deterministic finite automaton for a language L. Let Q be the set of states of this automaton; let $n = |Q|$. A string w defines a function from Q to Q: $f_w(q) = \delta(q, w)$. Clearly there are only n^n possible such functions. But if $f_u = f_v$ then $u \equiv_L v$, and so there can be at most n^n possible congruence classes. Indeed Holzer and Konig [19] show that we can approach this bound. This reveals two things: one that using one non-terminal per congruence class could be an expensive mistake as the number might be prohibitively large, and secondly, that there is often some non-trivial structure to the monoid.

Since these congruence classes correspond to functions from Q to Q it seems reasonable to represent them using some basis functions. Consider the set of partial functions that take $q_i \rightarrow q_j$: there are only n^2 of these. Each congruence class can be represented as a collection of at most n of these that define the image under f of each $q \in Q$.

If we represent each congruence class as a n by n boolean matrix T; where T_{ij} is 1 iff $f_u : q_i \mapsto q_j$, then the basis functions are the n^2 matrices that have just a single 1; and we represent each congruence class as a sum of n such basis functions.

Suppose the language is reversible [1]: i.e. $uv, u'v, uv' \in L$ implies $u'v' \in L$. Then for each state q_i we can define a pair of strings l_i, r_i that uniquely pick out that state: in the sense that $\delta(q_0, l_i) = q_i$ and only q_i has the property that $\delta(q_i, r_i)$ is a final state.

Thus we can represent the basis functions using the finite set of contexts (l_i, r_j) that represents the transition function $q_i \rightarrow q_j$. This gives us an important clue how to represent the syntactic monoid: If we have a class that maps $q_i \rightarrow q_j$ and another which maps $q_j \rightarrow q_k$ then the concatenation will map $q_i \rightarrow q_k$. Thus rather than having a very large number of very specific rules that show how individual congruence classes combine, we can have a very much smaller set of more general rules which should be easier to learn. This requires a representation that contains elements that correspond not just to individual congruence classes but to sets of congruence classes.

5 Distributional Lattice Grammars

The final class of representations that we consider are based on a richer algebraic structure; see [8] for a more detailed exposition. Note that the congruence classes correspond to sets of strings and dually to sets of contexts: a congruence class $[u]$ also defines the distribution $C_L(u)$ and vice versa. It is natural to consider therefore as our primitive elements certain ordered pairs which we write $\langle S, C\rangle$ where S is a subset of Σ^* and C is a subset of $\Sigma^* \times \Sigma^*$. Given a language L we will consider only those pairs that satisfy two conditions: first that $C \odot S$ is a subset of L, and secondly that both of these sets are maximal, while respecting the first condition. If a pair satisfies these conditions, then we call it a *syntactic concept* of the language.

We have chosen to define it in this way to bring out the connection to the Universal automaton [12,24], which has the same construction but using a prefix-suffix relation, rather than the context substring relation.

Another way is to consider the Galois connection between the sets of strings and contexts, which give rise to exactly the same sets of concepts. For a given language L we can define two polar maps from sets of strings to sets of contexts and vice versa. Given a set of strings S we can define a set of contexts S' to be the set of contexts that appear with every element of S.

$$S' = \{(l, r) : \forall w \in S \;\; lwr \in L\} \tag{1}$$

Dually we can define for a set of contexts C the set of strings C' that occur with all of the elements of C

$$C' = \{w : \forall (l, r) \in C \;\; lwr \in L\} \tag{2}$$

A concept is then an ordered pair $\langle S, C\rangle$ where $S' = C$ and $C' = S$. The most important point here is that these are closure operations in the sense that $S''' = S'$ and $C''' = C'$. This means that we can construct a concept by taking any set of strings S and computing $\langle S'', S'\rangle$, and similarly by taking any set of contexts C and computing $\langle C', C'' \rangle$. We will write $\mathcal{C}(S)$ for $\langle S'', S' \rangle$. This set of concepts have an interesting and rich algebraic structure, which gives rise to a powerful representation.

We will start by stating some basic properties of the set of concepts. The first point is that there is an inverse relation between the size of the set of strings S and the set of contexts C: the larger that S is the smaller that C is: in the limit there is always a concept where $S = \Sigma^*$; normally this will have $C = \emptyset$. Conversely we will always have an element $\mathcal{C}(\Sigma^* \times \Sigma^*)$. One particularly important concept is $\mathcal{C}(L) = \mathcal{C}((\lambda, \lambda))$: the language itself is one of the concepts.

The most basic structure that this set of concepts has is therefore as a partially ordered set. We can define a partial order on these concepts where:

$$\langle S_1, C_1 \rangle \leq \langle S_2, C_2 \rangle \text{ iff } S_1 \subseteq S_2.$$

$S_1 \subseteq S_2$ iff $C_1 \supseteq C_2$. We can see that $\mathcal{C}(L) = \mathcal{C}(\{(\lambda, \lambda)\})$, and clearly $w \in L$ iff $\mathcal{C}(\{w\}) \leq \mathcal{C}(\{(\lambda, \lambda)\})$.

Given this partial order we can see easily that in fact this forms a complete lattice; which we write $\mathfrak{B}(L)$, called the *syntactic concept lattice*. Here the topmost element is $\top = \mathcal{C}(\Sigma^*)$ bottom is written $\bot = \mathcal{C}(\Sigma^* \times \Sigma^*)$, and the two meet and join operations are defined as $\langle S_x, C_x \rangle \wedge \langle S_y, C_y \rangle$ is defined as $\langle S_x \cap S_y, (S_x \cap S_y)' \rangle$ and $\vee$ dually as $\langle (C_x \cap C_y)', C_x \cap C_y \rangle$.

Figure 1 shows the syntactic concept lattice for the regular language $L = \{(ab)^*\}$. L is infinite, but the lattice $\mathfrak{B}(L)$ is finite and has only 7 concepts.

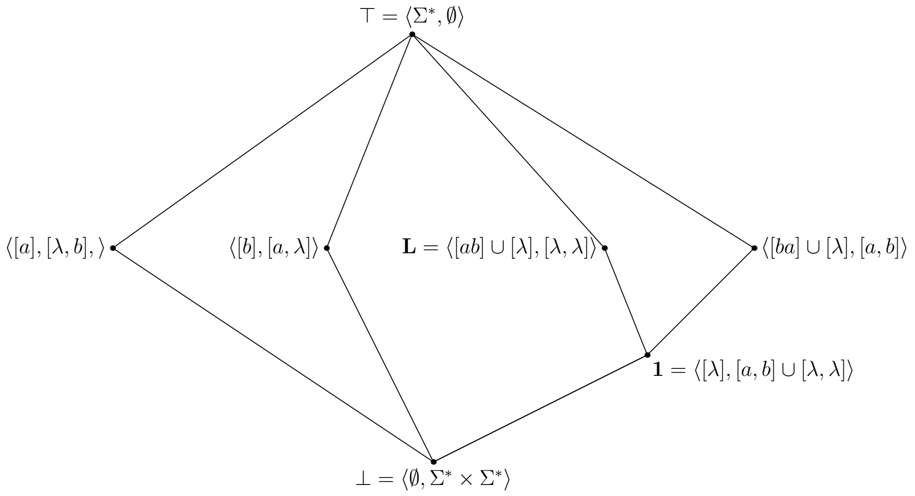

$\top = \langle \Sigma^*, \emptyset \rangle$
$\langle [a], [\lambda, b], \rangle$
$\langle [b], [a, \lambda] \rangle$
$\mathbf{L} = \langle [ab] \cup [\lambda], [\lambda, \lambda] \rangle$
$\langle [ba] \cup [\lambda], [a, b] \rangle$
$\mathbf{1} = \langle [\lambda], [a, b] \cup [\lambda, \lambda] \rangle$
$\bot = \langle \emptyset, \Sigma^* \times \Sigma^* \rangle$

Fig. 1. The Hasse diagram for the syntactic concept lattice for the regular language $L = \{(ab)^*\}$. Each concept (node in the diagram) is an ordered pair of a set of strings, and a set of contexts. We write $[u]$ for the equivalence class of the string u, $[l, r]$ for the equivalence class of the context (l, r).

Monoid Structure

Crucially, this lattice structure also has a monoid structure. We can define a binary operation over concepts using the two sets of strings of the concepts: define $\langle S_1, C_1 \rangle \circ \langle S_2, C_2 \rangle = \mathcal{C}(S_1 S_2)$. Note that this operation then forms a monoid, as it is both associative and has a unit $\mathcal{C}(\lambda)$. There is also an interaction between this monoid structure and the lattice structure. It is clearly monotonic in the sense that if $X \leq Y$ then $X \circ Z \leq Y \circ Z$ and so on, but there is a stronger relation. We can define two residual operations as follows:

Definition 1. *Suppose $X = \langle S_x, C_x \rangle$ and $Y = \langle S_y, C_y \rangle$ are concepts. Then define the residual $X/Y = \mathcal{C}(C_x \odot (\lambda, S_y))$ and $Y \backslash X = \mathcal{C}(C_x \odot (S_y, \lambda))$*

These satisfy the following conditions: $Y \leq X \backslash Z$ iff $X \circ Y \leq Z$ iff $X \leq Z/Y$. That is to say, given an element Z and an element X, $X \backslash Z$ is the largest element which when concatenated to the right of X will give you something that is less than Z.

With these operations the syntactic concept lattice becomes a residuated lattice. This gives some intriguing links to the theory of categorial grammars [23].

5.1 Maximal Elements

One important point of the residual operation is that we can use them to extract maximally general concatenation rules. So, suppose we have some concept Z. We can consider the set of all pairs of concepts (X, Y) such that their concatenation is less than Z.

$$H(Z) = \{(X, Y) | X \circ Y \leq Z\} \tag{3}$$

This is clearly a down set, in that if $(X, Y) \in H(Z)$ and $X' \leq X$ and $Y' \leq Y$ then $(X', Y') \in H(Z)$, and so it is natural to consider the maximal elements of this set. We can find these elements using the residuation operations. If $(X, Y) \in H(Z)$ then also we must have $(X, X \backslash Z)$ and $(Z/Y, Y)$ in $H(Z)$. But we can repeat this: we will also have $(Z/(X \backslash Z), X \backslash Z)$ and $(Z/Y, (Z/Y) \backslash Z)$ in $H(Z)$. We can prove that repeating the process further is not necessary – indeed all maximal elements of $H(Z)$ will be of this form.

An example: suppose $L = \{a^n b^n | n \geq 0\}$. Consider $H(\mathcal{C}(L))$. Clearly $\mathcal{C}(a) \circ \mathcal{C}(b) = \mathcal{C}(L)$, so $(\mathcal{C}(a), \mathcal{C}(b)) \in H(\mathcal{C}(L))$. Let us identify the two maximal elements that we get by generalising this pair. $\mathcal{C}(L)/\mathcal{C}(b) = \mathcal{C}(aab)$, and $\mathcal{C}(a) \backslash \mathcal{C}(L) = \mathcal{C}(abb)$. Repeating the process does not increase these so the two maximal elements above this pair are $(\mathcal{C}(a), \mathcal{C}(abb))$ and $(\mathcal{C}(aab), \mathcal{C}(b))$.

6 Representation

Having defined and examined the syntactic concept lattice, we can now define a representation based on this. Again, since the lattice will be infinite if the language is not regular, we need to consider just a part of it. We will start by considering how we might define a representation given the whole lattice.

Given a string w we want to compute whether it is in the language or not. Considering this slightly more generally we want to be able to compute for every string w, the concept of w, $\mathcal{C}(w)$. If $\mathcal{C}(w) \leq \mathcal{C}(L)$, then we know that the string is in the language. If we have the whole lattice then it is quite easy: since $\mathcal{C}(u) \circ \mathcal{C}(v) = \mathcal{C}(uv)$, we can simply take the list of letters that form w and concatenate their concepts. So if $w = a_1 \dots a_n$, then $\mathcal{C}(w) = \mathcal{C}(a_1) \circ \cdots \circ \mathcal{C}(a_n)$. It is enough to know the concepts of the letters, and of course of λ and the algebraic operation $\circ$ which is associative and well-behaved. In this case it effectively reduces to computing the syntactic monoid as before. The idea is very simple – we compute a representation of the distribution of a string, by taking the distributions of its parts and combining them.

However, if we have a non-regular language, then we will need to restrict the lattice in some way. We can do this by taking a finite set of contexts $F \subset \Sigma^* \times \Sigma^*$, which will include the special context (λ, λ) and constructing a lattice using only these contexts and all strings Σ^*. This give us a finite lattice $\mathfrak{B}(L, F)$, which will have at most $2^{|F|}$ elements. We can think of F as being a set of *features*, where a string w has the feature (context) (l, r) iff $lwr \in L$.

Definition 2. *For a language L and a set of context $F \subseteq \Sigma^* \times \Sigma^*$, the partial lattice $\mathfrak{B}(L, F)$ is the lattice of concepts $\langle S, C \rangle$ where $C \subseteq F$, and where $C = S' \cap F$, and $S = C'$.*

We can define a concatenation operation as before

$$\langle S_1, C_1 \rangle \circ \langle S_2, C_2 \rangle = \langle ((S_1 S_2)' \cap F)', (S_1 S_2)' \cap F \rangle$$

This is now however no longer a residuated lattice as the $\circ$ operation is no longer associative, there may not be an identity element, nor are the residuation operations well defined. We will now clearly not be able to compute things exactly for every language, but we should still be able to approximate the computation, and for some languages, and for some sets of features the approximation will be accurate.

We note some basic facts about this partial lattice: first it is no longer the case that $\mathcal{C}(u) \circ \mathcal{C}(v) = \mathcal{C}(uv)$. If we take a very long string in the language, we may be able to split it into two parts neither of which have any contexts ing F. For example, if we have the language of our running example $a^n b^n$, and a small set of short contexts, we could take the string $a^{100} b^{100}$ and split it into the two strings a^{100} and b^{100}. Neither of these two strings will have any contexts in F, and so $\mathcal{C}(a^{100}) = \mathcal{C}(b^{100}) = \top$; this means that $\mathcal{C}(a^{100}) \circ \mathcal{C}(b^{100}) = \top \circ \top = \top > \mathcal{C}(a^{100} b^{100}) = \mathcal{C}(L)$: they are not equal.

However we can prove that $\mathcal{C}(u) \circ \mathcal{C}(v) \geq \mathcal{C}(uv)$. This means that given some string, w, we can compute an upper bound on the $\mathcal{C}(w)$ quite easily: we will call this upper bound $\phi(w)$. This may not give us exactly the right answer but it will still be useful. If the upper bound is below $\mathcal{C}(L)$ then we definitely know that the string will be in the language: if $\mathcal{C}(w) \leq \phi(w)$ and $\phi(w) \leq \mathcal{C}(L)$, then $\mathcal{C}(w) \leq \mathcal{C}(L)$.

In fact we can compute many different upper bounds: since the operation is no longer associative, the order in which we do the computations matters. Suppose we have some string ab; our upper bound can just be $\mathcal{C}(a) \circ \mathcal{C}(b)$. But suppose we have a string of length 3: abb. Our upper bound could be $\mathcal{C}(a) \circ (\mathcal{C}(b) \circ \mathcal{C}(b))$ or $(\mathcal{C}(a) \circ \mathcal{C}(b)) \circ \mathcal{C}(b)$. We can now use the lattice structure to help: if $\mathcal{C}(abb) \leq X$ and $\mathcal{C}(abb) \leq Y$ then $\mathcal{C}(abb) \leq X \wedge Y$, since $X \wedge Y$ is a greatest lower bound. So in this case we can have our upper bound as $(\mathcal{C}(a) \circ (\mathcal{C}(b) \circ \mathcal{C}(b))) \wedge ((\mathcal{C}(a) \circ \mathcal{C}(b)) \circ \mathcal{C}(b))$. For longer strings, we have a problem: we cannot compute every possible binary tree and then take the meet over them all, since the number of such trees is exponential in the length.

However we can compute an even tighter bound using a recursive definition that can be computed by an efficient dynamic programming algorithm in $\mathcal{O}(|w|^3)$. For each substring of the word, we compute the lowest possible upper bound and then recursively combine them.

Given a language L and set of contexts F:

Definition 3. *we define $\phi : \Sigma^* \to \mathfrak{B}(L, F)$ recursively by*

- $\phi(\lambda) = \mathcal{C}(\lambda)$
- $\phi(a) = \mathcal{C}(a)$ *for all* $a \in \Sigma$, *(i.e. for all* $w, |w| = 1$*)*
- *for all* w *with* $|w| > 1$,

$$\phi(w) = \bigwedge_{u,v \in \Sigma^+ : uv = w} \phi(u) \circ \phi(v) \tag{4}$$

We can define the language generated by this representation to be:

$$\hat{L} = L(\mathfrak{B}(L, F)) = \{w | \phi(w) \leq \mathcal{C}((\lambda, \lambda))\} \tag{5}$$

We can show using a simple proof by induction that the computation ϕ will always produce an upper bound.

Lemma 1. *For any language L and set of contexts F, for any string w, $\phi(w) \geq \mathcal{C}(w)$*

This has the immediate consequence that:

Lemma 2. *For any language L and for any set of contexts F, $L(\mathfrak{B}(L, F)) \subseteq L$.*

As we increase the set of contexts, we will find that the language defined increase monotonically, until in the infinite limit when $F = \Sigma^* \times \Sigma^*$ we have that $L(\mathfrak{B}(L, \Sigma^* \times \Sigma^*)) = L$. This means that the problem of finding a suitable set of contexts is tractable and we can also define a natural class of languages as those which have representations as finite lattices. We will call this class of representations the Distributional Lattice Grammars.

The class of languages definable by finite DLGS is denoted $\mathcal{L}_{\text{DLG}}$.

$$\mathcal{L}_{\text{DLG}} = \{L : \exists \text{ a finite set } F \subset \Sigma^* \times \Sigma^* \text{ such that } L(\mathfrak{B}(L, F)) = L\} \tag{6}$$

First we note that $\mathcal{L}_{\text{DLG}}$properly includes $\mathcal{L}_{\text{CCFG}}$. Indeed $\mathcal{L}_{\text{DLG}}$includes some non-context free languages, including ones that are related to the MIX language

[3]. $\mathcal{L}_{\mathrm{DLG}}$is a proper subclass of the languages defined by Conjunctive Grammars [27]. $\mathcal{L}_{\mathrm{DLG}}$also includes a much larger set of context free languages than $\mathcal{L}_{\mathrm{CCFG}}$includingsome non-deterministic and inherently ambiguous languages.

A problem is that the lattices can be exponentially large. We can however represent them lazily using a limited set of examples, and only compute the concepts in the lattice as they are needed. This allows for efficient learning algorithms. An important future direction for research is to exploit the algebraic structure of the lattice to find more compact representations for these lattices, using maximal elements.

7 Discussion

The importance of reducing the under-determination of the grammar given the language has been noted before: de la Higuera and Fernau [15] argue that learnable representations must have a canonical form, and that equivalence should be computable.

On a historical note, it is important not to neglect the contribution of the Kulagina school, which was initiated by the seminal paper of [21]. The first work on the distributional lattice was by Sestier [31], and subsequent work was developed by Kunze [22]. However the concatenation and residuation properties seem not to have been noted. Important papers that follow this line of research include [20,13] as well as [26] and [25]. The ideas of basing representations on the congruence classes can be found in these works, but for some reason the rule schema $[uv] \rightarrow [u][v]$ seems not to have been discovered, instead exploration focussed on the linear grammar schema $[lur] \rightarrow l[u]r$, and on the development of contextual grammars [28]. We suspect that there are other related works in the Eastern European literature that we have not yet discovered.

Many other approaches can be recast in this form. For example the context-deterministic languages of [33], are exactly context free languages where the non-terminals have the property that they correspond to concepts, and where additionally the distributions of the non-terminals are disjoint: $\mathcal{C}(M) \vee \mathcal{C}(N) = \top$ for distinct non-terminals N and M.

There are a number of directions for future research that this approach suggests: probabilistic extensions of these algorithms, the development of algorithms for equivalence and decidability, and the use of these approaches for modelling transductions are all natural developments. Another extension would be to consider representations based on the relation $(x, y) \sim (u, v, w)$ iff $uxvyw \in L$. Some steps in this direction have recently been taken by Yoshinaka [34], which lead naturally to Multiple Context-Free Grammars. There are numerous language theoretic problems that need to be explored in the use of these models.

Finally, representations such as these, which are both efficiently learnable and capable of representing mildly context-sensitive languages seem to be good candidates for models of human linguistic competence.

Acknowledgements

I am very grateful to Rémi Eyraud and Amaury Habrard.

References

1. Angluin, D.: Inference of reversible languages. Communications of the ACM 29, 741–765 (1982)
2. Angluin, D., Kharitonov, M.: When won't membership queries help? J. Comput. Syst. Sci. 50, 336–355 (1995)
3. Boullier, P.: Chinese Numbers, MIX, Scrambling, and Range Concatenation Grammars. In: Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL), pp. 8–12 (1999)
4. Carrasco, R.C., Oncina, J.: Learning deterministic regular grammars from stochastic samples in polynomial time. Theoretical Informatics and Applications 33(1), 1–20 (1999)
5. Chomsky, N.: The Minimalist Program. MIT Press, Cambridge (1995)
6. Chomsky, N.: Language and mind, 3rd edn. Cambridge Univ. Pr., Cambridge (2006)
7. Clark, A.: PAC-learning unambiguous NTS languages. In: Sakakibara, Y., Kobayashi, S., Sato, K., Nishino, T., Tomita, E. (eds.) ICGI 2006. LNCS (LNAI), vol. 4201, pp. 59–71. Springer, Heidelberg (2006)
8. Clark, A.: A learnable representation for syntax using residuated lattices. In: Proceedings of the 14th Conference on Formal Grammar, Bordeaux, France (2009)
9. Clark, A., Eyraud, R.: Polynomial identification in the limit of substitutable context-free languages. Journal of Machine Learning Research 8, 1725–1745 (2007)
10. Clark, A., Eyraud, R., Habrard, A.: A polynomial algorithm for the inference of context free languages. In: Clark, A., Coste, F., Miclet, L. (eds.) ICGI 2008. LNCS (LNAI), vol. 5278, pp. 29–42. Springer, Heidelberg (2008)
11. Clark, A., Thollard, F.: PAC-learnability of probabilistic deterministic finite state automata. Journal of Machine Learning Research 5, 473–497 (2004)
12. Conway, J.: Regular algebra and finite machines. Chapman and Hall, London (1971)
13. Drášil, M.: A grammatical inference for C-finite languages. Archivum Mathematicum 25(2), 163–173 (1989)
14. Evans, R., Gazdar, G.: DATR: A language for lexical knowledge representation. Computational Linguistics 22(2), 167–216 (1996)
15. Fernau, H., de la Higuera, C.: Grammar induction: An invitation for formal language theorists. Grammars 7, 45–55 (2004)
16. Gold, E.M.: Complexity of automaton identification from given data. Information and Control 37(3), 302–320 (1978)
17. Harris, Z.: Distributional structure. In: Fodor, J.A., Katz, J.J. (eds.) The Structure of Language, pp. 33–49. Prentice-Hall, Englewood Cliffs (1954)
18. Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley, Reading (1978)
19. Holzer, M., Konig, B.: On deterministic finite automata and syntactic monoid size. In: Proc. Developments in Language Theory 2002 (2002)
20. Kříž, B.: Generalized grammatical categories in the sense of Kunze. Archivum Mathematicum 17(3), 151–158 (1981)

21. Kulagina, O.S.: One method of defining grammatical concepts on the basis of set theory. Problemy Kiberneticy 1, 203–214 (1958) (in Russian)
22. Kunze, J.: Versuch eines objektivierten Grammatikmodells I, II. Z. Zeitschriff Phonetik Sprachwiss. Kommunikat, 20-21 (1967–1968)
23. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly 65(3), 154–170 (1958)
24. Lombardy, S., Sakarovitch, J.: The universal automaton. In: Grädel, E., Flum, J., Wilke, T. (eds.) Logic and Automata: History and Perspectives, pp. 457–494. Amsterdam Univ. Pr. (2008)
25. Martinek, P.: On a Construction of Context-free Grammars. Fundamenta Informaticae 44(3), 245–264 (2000)
26. Novotny, M.: On some constructions of grammars for linear languages. International Journal of Computer Mathematics 17(1), 65–77 (1985)
27. Okhotin, A.: Conjunctive grammars. Journal of Automata, Languages and Combinatorics 6(4), 519–535 (2001)
28. Păun, G.: Marcus contextual grammars. Kluwer Academic Pub., Dordrecht (1997)
29. Pollard, C., Sag, I.: Head Driven Phrase Structure Grammar. University of Chicago Press, Chicago (1994)
30. Sénizergues, G.: The equivalence and inclusion problems for NTS languages. J. Comput. Syst. Sci. 31(3), 303–331 (1985)
31. Sestier, A.: Contribution à une théorie ensembliste des classifications linguistiques. In: Premier Congrès de l'Association Française de Calcul, Grenoble, pp. 293–305 (1960)
32. Shieber, S.: Evidence against the context-freeness of natural language. Linguistics and Philosophy 8, 333–343 (1985)
33. Shirakawa, H., Yokomori, T.: Polynomial-time MAT Learning of C-Deterministic Context-free Grammars. Transactions of the information processing society of Japan 34, 380–390 (1993)
34. Yoshinaka, R.: Learning mildly context-sensitive languages with multidimensional substitutability from positive data. In: Gavaldà, R., Lugosi, G., Zeugmann, T., Zilles, S. (eds.) ALT 2009. LNCS, vol. 5809, pp. 278–292. Springer, Heidelberg (2009)

Arbology: Trees and Pushdown Automata*

Bořivoj Melichar

Department of Theoretical Computer Science,
Faculty of Information Technology,
Czech Technical University in Prague,
Kolejni 550/2, 160 00 Prague 6, Czech Republic
melichar@fit.cvut.cz

1 Introduction

Trees are (data) structures used in many areas of human activity. Tree as the formal notion has been introduced in the theory of graphs. Nevertheless, trees have been used a long time before the foundation of the graph theory. An example is the notion of a genealogical tree. The area of family relationships was an origin of some terminology in the area of the tree theory (parent, child, sibling, ...) in addition to the terms originating from the area of the dendrology (root, branch, leaf, ...).

There are many applications of trees not only in the area of Computer Science. It seems that the reason for this is the ability of trees to express a hierarchical structure. The reader can find many applications of trees in his own area of interest.

Many algorithms were developed for operations on trees used in many applications. A number of them uses some way of traversing the tree. Such operations need a data structure to keep track on the possible continuation of the traversing. In many cases we can identify that the suitable data structure for this role is a pushdown store. The typical structure of these algorithms is based on the set of recursive procedures, where the pushdown store is used for saving return addresses.

Theory of tree automata has been developed as a tool for description of sets of trees (tree languages) and, moreover, for formal description of some operations with trees. Models of computation of this theory are various kinds of tree automata. The most researched kind of tree automata are finite tree automata, which recognize regular tree languages and their implementation is based on recursive procedures.

Some algorithms for operations on trees assume a linear notation (for example prefix, suffix, or Euler notations) of the input tree. We note that the linear notation can be obtained by the corresponding recursive traversing of the tree. It can be shown that the linear notation of a tree can be generated by a context–free grammar. Therefore, the pushdown automaton can be an appropriate model for algorithms processing linear notations of trees.

* This research has been partially supported by the Ministry of Education, Youth and Sports under research program MSMT 6840770014, and by the Czech Science Foundation as project No. 201/09/0807.

In [9] it is proved that the class of regular tree languages in postfix notation is a proper subclass of deterministic context-free string languages. Given a finite tree automaton it is proved that an equivalent LR(0) grammar can be constructed, which means that also an equivalent deterministic pushdown automaton can be constructed. Moreover, it is proved that deterministic pushdown automata are more powerful than finite tree automata: the class of tree languages whose linear notation can be accepted by deterministic pushdown automata is a proper superclass of regular tree languages.

These ways of reasoning led us to the decision to use pushdown automata and the corresponding underlying theory as a model of computation of tree processing algorithms. We call this branch of algorithmic study *arbology* [3] from the Spanish word *arbol*, meaning *tree*. The model and inspiration for building arbology can be found in stringology, which is an algorithmic discipline in the area of string processing. Stringology uses finite automata theory as a very useful tool. In arbology we try to apply the stringology principles to trees so that effective tree algorithms using pushdown automata would be created.

There are some differences between finite and pushdown automata theories. For every nondeterministic finite automaton there exists an equivalent deterministic finite automaton which can be constructed using well known algorithm. This does not hold generally for the case of pushdown automata – for some nondeterministic pushdown automata their equivalent deterministic versions do not exist. Examples of such pushdown automata are pushdown automata accepting palindromes of the form like ww^R. The reason is that an automaton reading the palindrome from left to right is not able to find the centre of it. Generally, it is not known how to decide for a given nondeterministic pushdown automaton whether there exists a deterministic equivalent or not. Nevertheless, we have identified three classes of pushdown automata for which such a determinisation is possible. These classes are called input–driven, visible [2] and heigth–deterministic pushdown automata [14].

Without regard to the above-mentioned problems some results have been achieved and are presented in the subsequent sections in brief. Section 2 contains basic definitions. Section 3 deals with linear notations of trees and some important properties of these notations are discussed in Section 4. Section 5 deals with determinisation of pushdown automata. Section 6 presents exact subtree matching. A complete index of subtrees and tree templates in the form of subtree and tree pattern pushdown automata is presented in Section 7. An application of these pushdown automata is shown in Section 8 where the way how to find repeats of subtrees is shown.

2 Basic Notions

We define notions on trees similarly as they are defined in [1,4,6,7].

2.1 Alphabet

An *alphabet* is a finite nonempty set of *symbols*. A *ranked alphabet* is a finite nonempty set of symbols each of which has a unique nonnegative *arity* (or *rank*).

Given a ranked alphabet $\mathcal{A}$, the arity of a symbol $a \in \mathcal{A}$ is denoted $Arity(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \dots, p$ are respectively called nullary (constants), unary, binary, ..., p-ary symbols. We assume that $\mathcal{A}$ contains at least one constant. In the examples we use numbers at the end of the identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a.

2.2 Tree, Tree Pattern, Tree Template

Based on concepts from graph theory (see [1]), a tree over an alphabet $\mathcal{A}$ can be defined as follows:

A *directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of lists of edges, where each element of R has the form $((f, g_1), (f, g_2), \dots, (f, g_n))$, $f, g_1, g_2, \dots, g_n \in N$, $n \geq 0$. This element will indicate that, for node f, there are n edges leaving f, entering node g_1, node g_2, and so forth.

A sequence of nodes $(f_0, f_1, \dots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \dots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. A *labelling* of an ordered graph $G = (N, R)$ is a mapping of N into a set of labels. In the examples we use a_f for a short declaration of node f labelled by symbol a.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in N$. By analogy, the *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in N$.

A *tree* is an acyclic connected graph. Any node of a tree can be selected as a *root* of the tree. A tree with a root is called *rooted tree*.

A tree can be *directed*. A *rooted and directed tree* t is a dag $t = (N, R)$ with a special node $r \in N$, called the *root*, such that

(1) r has in-degree 0,
(2) all other nodes of t have in-degree 1,
(3) there is just one path from the root r to every $f \in N$, where $f \neq r$.

A *labelled, (rooted, directed) tree* is a tree having the following property:

(4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet.

A *ranked, (labelled, rooted, directed) tree* is a tree labelled by symbols from a ranked alphabet and out-degree of a node f labelled by symbol $a \in \mathcal{A}$ is $Arity(a)$. Nodes labelled by nullary symbols (constants) are called *leaves*.

An *ordered, (ranked, labelled, rooted, directed) tree* is a tree where direct descendants $a_{f1}, a_{f2}, \dots, a_{fn}$ of a node a_f having an $Arity(a_f) = n$ are ordered.

The height of a tree t, denoted by *Height(t)*, is defined as the maximal length of a path from the root of t to a leaf of t.

To define a *tree pattern*, we use a special nullary symbol S, not in $\mathcal{A}$, which serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered ranked tree over ranked alphabet $\mathcal{A} \cup \{S\}$. By analogy, a tree pattern in prefix notation is defined as a labelled ordered ranked tree over ranked alphabet $\mathcal{A} \cup \{S\}$ in prefix notation. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. A tree pattern containing at least one symbol S will be called a *tree template*.

A tree pattern p with $k \geq 0$ occurrences of the symbol S *matches* an object tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of the tree t such that the tree p', obtained from p by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, k$, is equal to the subtree of t rooted at n.

2.3 Language, Grammar, Finite and Pushdown Automata

We define notions from the theory of string languages similarly as they are defined in [1].

A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Symbol $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly, for string $x \in \mathcal{A}^*$, symbol x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$ and $x^+ = x^* \setminus \{\varepsilon\} = \{x^m : m \geq 1\}$.

A *context-free grammar* (CFG) is a 4-tuple $G = (N, \mathcal{A}, P, S)$, where N and $\mathcal{A}$ are finite disjoint sets of *nonterminal* and *terminal (input) symbols*, respectively. P is a finite set of *rules* $A \rightarrow \alpha$, where $A \in N$, $\alpha \in (N \cup \mathcal{A})^*$. $S \in N$ is the *start symbol*. Relation $\Rightarrow$ is called *derivation*: if $\alpha A\, \gamma \Rightarrow \alpha\beta\gamma$, $A \in N$, and α, β, $\gamma \in (N \cup \mathcal{A})^*$, then rule $A \rightarrow \beta$ is in P. Symbols $\Rightarrow^+$, and $\Rightarrow^*$ are used for the *transitive*, and the *transitive and reflexive* closure of $\Rightarrow$, respectively. The language generated by a G, denoted by $L(G)$, is the set of strings $L(G) = \{w : S \ \Rightarrow^* w,\, w \in \mathcal{A}^*\}$.

A *nondeterministic finite automaton* (NFA) is a five-tuple $FM = (Q, \mathcal{A}, \delta, q_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, δ is a mapping from $Q \times \mathcal{A}$ into a set of finite subsets of Q, $q_0 \in Q$ is an initial state, and $F \subseteq Q$ is the set of final (accepting) states. A finite automaton FM is *deterministic* (DFA) if $\delta(q, a)$ has no more than one member for any $q \in Q$ and $a \in \mathcal{A}$. We note that the mapping δ is often illustrated by its transition diagram.

Every NFA can be transformed to an equivalent DFA [1]. The transformation constructs the states of the DFA as subsets of states of the NFA and selects only such accessible states (ie subsets). These subsets are called *d-subsets*. In spite of the fact that d-subsets are standard sets, they are often written in square brackets ([ ]) instead of in braces ({ }).

A *nondeterministic pushdown automaton* (nondeterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, G is a *pushdown store alphabet*, δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, and $F \subseteq Q$ is the set of final (accepting) states.

An *extended nondeterministic pushdown automaton* is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G^*$ into a set of finite subsets of $Q \times G^*$ and all other symbols have the same meaning as above.

Triple $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. We will write the top of the pushdown store x on its left hand side. The initial configuration of a pushdown automaton is a triple (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$. The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

A pushdown automaton M is a *deterministic* pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, Z)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A}$, $Z \in G$ and $\delta(q, \varepsilon, Z) = \emptyset$ or
2. $\delta(q, a, Z) = \emptyset$ for all $a \in \mathcal{A}$ and $|\delta(q, \varepsilon, Z)| \leq 1$.

An extended pushdown automaton M is an *deterministic* extended pushdown automaton (deterministic PDA), if it holds:

1. $|\delta(q, a, \gamma)| \leq 1$ for all $q \in Q$, $a \in \mathcal{A} \cup \{\varepsilon\}$, $\gamma \in G^*$.
2. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$ then α is not a suffix of β and β is not a suffix of α.
3. If $\delta(q, a, \alpha) \neq \emptyset$, $\delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a suffix of β and β is not a suffix of α.

A pushdown automaton is *input–driven* if each of its pushdown operations is determined only by the input symbol.

A language L accepted by a pushdown automaton M is defined in two distinct ways:

1. *Accepting by final state:*
$$L(M) = \{x : \delta(q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma) \wedge x \in \mathcal{A}^* \wedge \gamma \in G^* \wedge q \in F\}.$$
2. *Accepting by empty pushdown store:*
$$L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}.$$

If pushdown automaton accepts the language by empty pushdown store, then the set F of final states is the empty set.

3 Linear Notation of Trees

Every sequential computational process (sequential algorithm) visits nodes of a processed tree in some sequence. It means that it does some linearisation of a tree. This linearisation can be implicit or explicit. The implicit linearisation takes place if a tree is represented as a linked data structure. The explicit linearisation means that a tree is represented as a linear structure in the form of a sequence of nodes and some special symbols. This linear notation of a tree is then an input of a computational process.

The principles of linear notations are based on the notion of visits of nodes. A computational process visits every node one or more times. The linear notations of a tree can be divided according to the number of visits recorded in it:

1. every node is recorded during all visits,
2. nodes are recorded during more than one visit,
3. every node is recorded just once during some visit.

There are depth first or breath first oriented principles of tree traversing. In the following we take into account one–visit versions of the depth first oriented traversing of a tree during its linearisation.

3.1 One Visit Linear Notation

There are two standard one visit depth first oriented linear notations:

- prefix (also called preorder) notation and
- postfix (also called postorder) notation.

Definition 1. The *prefix notation* $pref(t)$ of a tree t is defined in this way:

1. $pref(t) = a$ if a_f is a leaf,
2. $pref(t) = a\ pref(b_1)\ pref(b_2) \ldots pref(b_n)$, where a is the root of the tree t and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

We note that in many papers on the theory of tree languages, such as [4,6,7], labelled ordered ranked trees are defined with the use of ordered ranked *ground terms*. Ground terms can be regarded as labelled ordered ranked trees in prefix notation.

Definition 2. The *postfix notation* $post(t)$ of a tree t is defined in this way:

1. $post(t) = a$ if a_f is a leaf,
2. $post(t) = post(b_1)\ post(b_2) \ldots post(b_n)\ a$, where a is the root of the tree t and $b_1, b_2, \ldots\ b_n$ are direct descendants of a.

Definitions of the basic prefix and postfix notations are very useful for ranked trees. If the tree is not ranked it is necessary to include information concerning the rank of every node. This can be done in two ways:

1. to represent a node in both notations as a pair $(a, Arity(a))$,
2. to use another principles of linearisation based on incorporation of some special symbols.

The second approach can be illustrated by a bracketted notation in which each subtree is enclosed in the left and the close bracket. The bar notations are based on the following observations:

1. there is always the root of a subtree just behind the left bracket in prefix notation,
2. there is always the right bracket just behind the root of a subtree in postfix notation.

It follows from this observation that there is the left (right) bracket redundant in prefix (postfix) bracketted notation. The bar notations in both cases reduces the number of symbols in both linear bracketted notations. Instead of different symbols, left or right brackets, the symbol bar (|) can be used in both cases.

Definition 3. The *prefix bar notation* $pref\_bar(t)$ and *postfix bar notation* $post\_bar(t)$ of a tree t are defined in this way:

1. $pref\_bar(a) = a \mid$ and $post\_bar(a) = \mid a$, respectively.
2. $pref\_bar(t) = a\ pref\_bar(b_1)\ pref\_bar(b_2) \dots pref\_bar(b_n) \mid$ and $post\_bar(t) = \mid post\_bar(b_1)\ post\_bar(b_2) \dots post\_bar(b_n)\ a$ for prefix and postfix bar notation, respectively, where a is the root of the tree t and $b_1, b_2, \dots\ b_n$ are direct descendants of a.

4 Properties of Linear Notations of Trees

In this section we prove some general properties of the one visit linear notations of trees. These properties are substantial for creating our arbology algorithms, which process trees using pushdown automata.

It holds for any tree that each of its subtrees in a one-visit linear notation is a substring of the tree in the linear notation.

Theorem 1. *Given a tree t and its notations $pref(t)$, $post(t)$, $pref\_bar(t)$ and $post\_bar(t)$, all subtrees of t in prefix, postfix, prefix bar and postfix bar notation are substrings of $pref(t)$, $post(t)$, $pref\_bar(t)$ and $post\_bar(t)$, respectively.*

Proof. In [10].

However, not every substring of a tree in a linear notation is the linear notation of its subtree. This can be easily seen from the fact that for a given tree with n nodes there can be $\mathcal{O}(n^2)$ distinct substrings, but there are just n subtrees – each node of the tree is the root of just one subtree. Just those substrings which themselves are trees in a linear notation are those which are the subtrees in the linear notation. This property is formalised by the following definitions and theorems.

Definition 4. Let $w = a_1a_2 \dots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the *arity checksum* $ac(w) = arity(a_1) + arity(a_2) + \dots + arity(a_m) - m + 1 = \sum_{i=1}^{m} arity(a_i) - m + 1$.

Theorem 2. *Let $pref(t)$ and w be a tree t in prefix notation and a substring of $pref(t)$, respectively. Then, w is the prefix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \geq 1$ for each w_1, where $w = w_1x$, $x \neq \varepsilon$.*

Proof. In [10].

We show the dual principle for the postfix notation.

Theorem 3. *Let $post(t)$ and w be a tree t in postfix notation and a substring of $post(t)$, respectively. Then, w is the postfix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \leq -1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$.*

Similarly for prefix and postfix bar notations.

Definition 5. Let $w = a_1 a_2 \dots a_m$, $m \geq 1$, be a string over $\mathcal{A} \cup \{|\}$. Then, the *bar checksum* is defined as follows:

1. $bc(a) = 1$, and $bc(|) = -1$.
2. $bc(wa) = bc(w) + 1$, and $bc(w|) = bc(w) - 1$.

Theorem 4. *Let pref_bar(t) and w be a tree t in prefix bar notation and a substring of pref_bar(t), respectively. Then, w is the prefix bar notation of a subtree of t, if and only if $bc(w) = 0$, and $bc(w_1) \geq 1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$.*

The dual principle holds for the postfix bar notation.

Theorem 5. *Let post_bar(t) and w be a tree t in postfix bar notation and a substring of post_bar(t), respectively. Then, w is the postfix bar notation of a subtree of t, if and only if $bc(w) = 0$, and $bc(w_1) \leq -1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$.*

We note that pushdown automata presented in the next sections compute arity or bar checksums by pushdown operations during the processing of trees. This computing of checksums is formally described by the following four theorems.

Theorem 6. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ be an input-driven PDA of which each transition from δ is of the form $\delta(q_1, a, S) = (q_2, S^i)$, where $i = arity(a)$. Then, if $(q_3, w, S) \vdash_M^+ (q_4, \varepsilon, S^j)$, where w is a tree in prefix notation, then $j = ac(w)$.*

Proof. In [10].

Theorem 7. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ be an input–driven PDA whose each transition from δ is of the form $\delta(q_1, a, S^i) = (q_2, S)$, where $i = Arity(a)$. Then, if $(q_3, w, \varepsilon) \vdash_M^+ (q_4, \varepsilon, S^j)$, where w is a tree in postfix notation, then $j = -ac(w) + 1$.*

Theorem 8. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ be an input–driven PDA whose each transition from δ is of the form $\delta(q_1, a, \varepsilon) = (q_2, S)$ or $\delta(q_1, |, S) = (q_2, \varepsilon)$. Then, if $(q_3, w, \varepsilon) \vdash_M^+ (q_4, \varepsilon, S^j)$, where w is a tree in prefix bar notation, then $j = bc(w)$.*

Theorem 9. *Let $M = (\{Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ be an input–driven PDA whose each transition from δ is of the form $\delta(q_1, a, S) = (q_2, \varepsilon)$ or $\delta(q_1, |, \varepsilon) = (q_2, S)$. Then, if $(q_3, w, \varepsilon) \vdash_M^+ (q_4, \varepsilon, S^j)$, where w is a tree in postfix bar notation, then $j = -bc(w)$.*

5 On Determinisation of Pushdown Automata

It is well known that in the theory of finite automata there exists the algorithm of transformation of any nondeterministic finite automaton to an equivalent deterministic one. The determinisation of a finite automaton contains a creation of sets of states of a nondeterministic automaton. These subsets play the role of states of an equivalent deterministic finite automaton. The number of states of resulting deterministic finite automaton is less or equal to 2^n, where n is the number of states of the original nondeterministic finite automaton.

Such a universal algorithm for the determinisation of pushdown automata does not exist. We identified three classes of nondeterministic pushdown automata for which exist algorithms for determinisation. They are called input–driven, visible [2] and heigth–deterministic pushdown automata [14]. Algorithms for determinisation are different for these classes.

The principle of determinisation of finite automata can be used for input–driven pushdown automata. A notion of pushdown operation will be frequently used in the following meaning.

Definition 6. Let $M = (Q, A, G, \delta, q_0, Z_0, F)$ be a pushdown automaton. Let $\delta(q, a, \alpha)$ contains pair (p, β) for $p, q \in Q$, $a \in A \cup \varepsilon$, $\alpha, \beta \in G^*$. Then the notation $\alpha \mapsto \beta$ is used for operation popping α from the top of the pushdown store and pushing β to the top of the pushdown store. This operation is called *pushdown operation.*

Definition 7. A pushdown automaton $M = (Q, A, G, \delta, q_0, Z_0, F)$ is an input–driven pushdown automaton if each pushdown operation $\alpha \mapsto \beta$ during every transition is explicitly determined by the input symbol. In more formal notation: For each $Q \in Q$ and $a \in \mathcal{A} \cup \{\varepsilon\}$ there exists the only mapping $\delta(q, a, \alpha) = \{(p_1, \beta), (p_2, \beta), ..., (p_m, \beta)\}$ for one pair $\alpha, \beta \in G^*$ and $p_1, p_2, \ldots, p_m \in Q$.

Given a nondeterministic input–driven PDA, it can be determinised as follows:

Algorithm 1. Transformation of an input–driven nondeterministic PDA to an equivalent deterministic PDA.
Input: Input–driven nondeterministic PDA $M_{nx}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$, where the ordering of its states is such that if $\delta(p, a, \alpha) = (q, \beta)$, then $p \leq q$.
Output: Equivalent deterministic PDA $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', q_I, S, \emptyset)$.
Method:

```
1. Let cpds(q'), where $q' \in Q'$, denote a set of strings over $\{S\}$. (The abbreviation cpds stands for Contents of the PushDown Store.)
2. Initially, $Q' = \{[0]\}$, $q_I = [0]$, $cpds([0]) = \{S\}$ and $[0]$ is an unmarked state.
3. (a) Select an unmarked state $q'$ from $Q'$ such that $q'$ contains the smallest possible state $q \in Q$, where $0 \leq q \leq n$.
   (b) If there is $S^r \in cpds(q')$, $r \geq 1$, then for each input symbol $a \in \mathcal{A}$:
       i. Add transition $\delta'(q', a, \alpha) = (q'', \beta)$, where $q'' = \{q : \delta(p, a, \alpha) = (q, \beta)$ for all $p \in q'\}$. If $q''$ is not in $Q'$ then add $q''$ to $Q'$ and create $cpds(q'') = \emptyset$. Add $\omega$, where $\delta(q', a, \gamma) \vdash_{M_{dx}(t)} (q'', \varepsilon, \omega)$ and $\gamma \in cpds(q')$, to $cpds(q'')$.
   (c) Set the state $q'$ as marked.
4. Repeat step 3 until all states in $Q'$ are marked. □
```

Theorem 10. *Given a input–driven nondeterministic PDA $M_{nx}(t) = (Q, \mathcal{A}, \{S\}, \delta, q_0, S, \emptyset)$, the deterministic PDA $M_{dx}(t) = (Q', \mathcal{A}, \{S\}, \delta', \{q_0\}, S, \emptyset)$ constructed by Alg. 1 is equivalent to PDA $M_{nx}(t)$.*

Proof. In [10].

6 Exact Tree Pattern Matching

A systematic approach to the construction of subtree pattern matchers by deterministic pushdown automata, which read subject trees in prefix and postfix notation, is presented in this section. The method is analogous to the construction of string pattern matchers: for a given pattern, a nondeterministic pushdown automaton is created and then it is determinised. The size of the resulting deterministic pushdown automata directly corresponds to the size of the existing string pattern matchers based on finite automata.

Definition 8. *Let s and pref(s) be a tree and its prefix notation, respectively. Given an input tree t, a subtree pushdown automaton constructed over pref(s) accepts all matches of tree s in the input tree t by final state.*

6.1 Subtree Matching

First, we start with a PDA which accepts the whole subject tree in prefix notation. The construction of the PDA accepting a tree in prefix notation is described by Alg. 1. The constructed PDA is deterministic.

Algorithm 2. Construction of a PDA accepting *pref*(t) by final state.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \{n\})$.
Method:

1. For each state i, where $1 \leq i \leq n$, create a new transition $\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Lemma 1. *Given a tree t and its prefix notation pref(t), the PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$, where $n = |t|$, constructed by Alg. 6, accepts pref(t).*

Proof. In [5].

We present the construction of the deterministic subtree matching PDA for trees in prefix notation. The construction consists of two steps. First, a nondeterministic subtree matching PDA is constructed by Alg. 3. This nondeterministic subtree matching PDA is an extension of the PDA accepting trees in prefix notation, which is constructed by Alg. 2. Second, the constructed nondeterministic subtree matching PDA is transformed to the equivalent deterministic subtree matching PDA by Alg. 1.

Algorithm 3. Construction of a nondeterministic subtree matching PDA for a tree t in prefix notation *pref*(t).
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: Nondeterministic subtree matching PDA $M_{nps}(t) = (\{0, 1, 2, \dots, n\}$, $\mathcal{A}$, $\{S\}, \delta, 0, S, \{n\})$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t)$ by Alg. 2.
2. For each symbol $a \in \mathcal{A}$ create a new transition $\delta(0, a, S) = (0, S^{Arity(a)})$, where $S^0 = \varepsilon$. □

Theorem 11. *Given a tree t and its prefix notation pref(t), the PDA $M_{nps}(t)$ constructed by Alg. 3 is a subtree matching PDA for pref(t).*

Proof. In [5].

For the construction of deterministic subtree matching PDA, we use the transformation described by Alg. 1.

Theorem 12. *Given a tree t with n nodes in its prefix or postfix notation, the deterministic subtree matching PDA $M_{pds}(t)$ constructed by Alg. 3 and 1 is made of exactly $n + 1$ states, one pushdown symbol and $|\mathcal{A}|(n + 1)$ transitions.*

Proof. In [5].

Theorem 13. *Given an input tree t with n nodes, the searching phase of the deterministic subtree matching automaton constructed by Algs. 3 and 1 is $\mathcal{O}(n)$.*

Proof. In [5]. □

6.2 Multiple Subtree Matching

Definition 9. *Let $P = \{t_1, t_2, \dots, t_m\}$ be a set of m trees and pref$(t_i), 1 \leq i \leq m$ be the prefix notation of the i-th tree in P. Given an input tree t, a subtree pushdown automaton constructed over set P accepts all matches of subtrees $t_1, t_2, \dots, t_m$ in the input tree t by final state.*

Similarly as in Subsection 6.1, our method begins with a PDA which accepts trees $t_1, t_2, \dots, t_m$ in their prefix notation. The construction of this PDA is described by Alg. 4.

Algorithm 4. Construction of a PDA accepting a set of trees $P = \{t_1, t_2, \dots, t_m\}$ in their prefix notation.
Input: A set of trees $P = \{t_1, t_2, \dots, t_m\}$ over a ranked alphabet $\mathcal{A}$; prefix notation *pref*$(t_i) = a_1a_2 \dots a_{n_i}$, $1 \leq i \leq m$, $n_i \geq 1$.
Output: PDA $M_p(P) = (\{0, 1, 2, \dots, q\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$.

Method:

1. Create PDAs $M_p(t_i) = (Q_i, \mathcal{A}, \{S\}, \delta_i, 0_i, S, F_i)$ by Alg. 2 for $i = 1, 2, \ldots, m$.
2. Create PDA $M_p(P) = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$, where
   $Q = \bigcup_{i=1}^{m} (Q_i \setminus \{0_i\}) \cup \{0\}$,
   $\delta(q, a, S) = \delta_i(q, a, S)$,
   $\delta(0, a, S) = \delta_i(0_i, a, S)$, where $q \in (Q \setminus \{0\})$, $i = 1, 2, \ldots, m$,
   $F = \bigcup_{i=1}^{m} F_i$. □

The correctness of the deterministic PDA constructed by Alg. 4, which accepts trees in prefix notation, is described by the following lemma.

Lemma 2. *Given a set of k trees $P = \{t_1, t_2, \ldots, t_m\}$ and their prefix notation pref(t_i), $1 \le i \le m$, the PDA $M_p(P) = (\{0, 1, 2, \ldots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, F)$, where $1 + min(|t_1|, |t_2|, \ldots, |t_m|) \le n \le 1 + \sum_{j=1}^{k} |t_j|$, constructed by Alg. 4 accepts pref(t_i), where $1 \le t_i \le m$.*

Proof. In [5].

The deterministic subtree matching PDA for multiple tree patterns in prefix notation can be constructed in a similar fashion to the subtree matching PDA for a single pattern. First, the PDA accepting a set of trees in their prefix notations, constructed by Alg. 4, is used to construct a nondeterministic subtree matching PDA by Alg. 5. The constructed nondeterministic subtree matching PDA is then transformed to the equivalent deterministic subtree matching PDA by Alg. 1.

Algorithm 5. Construction of a nondeterministic subtree matching PDA for a set of trees $P = \{t_1, t_2, \ldots, t_m\}$ in their prefix notation.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \ldots a_n$, $n \ge 1$.
Output: Nondeterministic subtree matching PDA $M_{nps}(t) = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t) = (Q, \mathcal{A}, \{S\}, \delta, 0, S, F)$ by Alg. 4.
2. For each symbol $a \in \mathcal{A}$ create a new transition $\delta(0, a, S) = (0, S^{Arity(a)})$, where $S^0 = \varepsilon$. □

Theorem 14. *Given a set of m trees $P = \{t_1, t_2, \ldots, t_m\}$ and their prefix notation pref(t_i), $1 \le i \le m$, the PDA $M_{nps}(P)$ constructed by Algs. 4 and 5 is a subtree matching PDA for tree patterns $t_1, t_2, \ldots, t_m$.*

Proof. In [5].

Theorem 15. *Given a set of m trees $P = \{t_1, t_2, \ldots, t_m\}$ over a ranked alphabet $\mathcal{A}$, the deterministic subtree matching PDA $M_{pds}(P)$ is constructed by Alg. 5 and 1 in time $\Theta(|\mathcal{A}|s)$, requires $\Theta(|\mathcal{A}|s)$ storage, where $s = \sum_{i=1}^{m} |t_i|$, and its pushdown store alphabet consists of one symbol.*

Proof. In [5].

Theorem 16. *Given an input tree t with n nodes, the searching phase of the deterministic subtree matching automaton constructed by Algs. 2 and 3 over a set of m trees P is $\mathcal{O}(n)$.*

Proof. In [5].

7 Indexing Trees

This section briefly describe subtree PDAs [8] and tree pattern PDAs [10] for trees in prefix notation. These PDAs are analogous to string suffix and factor automata, respectively. Subtree pushdown automata accept all subtrees of the tree. Tree pattern pushdown automata accept all tree patterns which match the tree. The presented pushdown automata are input–driven and therefore can be determinised. Given a tree with n nodes, the deterministic subtree and the deterministic tree pattern pushdown automaton represent a complete index of the tree, and the search phase of all occurrences of a subtree or a tree pattern, respectively, of size m is performed in time linear in m and not depending on n. This is faster than the time of the existing tree pattern matching algorithms, which depends on n. The total size of the deterministic subtree pushdown automaton is linear in n. Although the number of distinct tree patterns which match the tree can be exponential in n, for specific cases of trees the total size of the deterministic tree pattern pushdown automaton is linear in n.

Definition 10. *Let t and $pref(t)$ be a tree and its prefix notation, respectively. A* subtree pushdown automaton *for $pref(t)$ accepts all subtrees of t in prefix notation.*

First, we start with a PDA which accepts the whole subject tree in prefix notation by empty pushdown store, whose construction is described by Alg. 6. The constructed PDA is deterministic.

Algorithm 6. Construction of a PDA accepting $pref(t)$ by empty pushdown store.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. For each state i, where $1 \leq i \leq n$, create a new transition
   $\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Lemma 3. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_p(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$, where $n \geq 0$, constructed by Alg. 6 accepts $pref(t)$.*

Proof. In [10].

The construction of the deterministic subtree PDA for trees in prefix notation consists of two steps. First, a nondeterministic subtree PDA is constructed by Alg. 7. This nondeterministic subtree PDA is an extension of the PDA accepting tree in prefix notation, which is constructed by Alg. 6. Second, the constructed nondeterministic subtree PDA is transformed to the equivalent deterministic subtree PDA by Alg. 1.

Algorithm 7. Construction of a nondeterministic subtree PDA for a tree t in prefix notation $pref(t)$.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: Nondeterministic PDA $M_{nps}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. Create PDA $M_{nps}(t)$ as PDA $M_p(t)$ by Alg. 6.
2. For each state i, where $2 \leq i \leq n$, create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Theorem 17. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_{nps}(t)$ constructed by Alg. 7 is a subtree PDA for $pref(t)$.*

Proof. In [10].

To construct deterministic subtree or tree pattern PDAs from their nondeterministic versions we use the transformation described by Alg. 1.

Lemma 4. *Given a tree t with n nodes, the number of distinct subtrees of tree t is equal or smaller than n.*

Proof. In [10].

The deterministic subtree PDA has the only pushdown symbol S, and all its states and transitions correspond to the states and the transitions, respectively, of the deterministic suffix automaton constructed for $pref(t)$. Therefore, the total size of the deterministic subtree PDA cannot be greater than the total size of the deterministic suffix automaton constructed for $pref(t)$.

Theorem 18. *Given a tree t with n nodes and its prefix notation $pref(t)$, the deterministic subtree PDA $M_{dps}(t)$ constructed by Algs. 7 and 1 has just one pushdown symbol, fewer than $N \leq 2n + 1$ states and at most $N + n - 1 \leq 3n$ transitions.*

Proof. In [10].

We proceed with presenting tree pattern PDA.

Definition 11. *Let t and $pref(t)$ be a tree and its prefix notation, respectively. A* tree pattern pushdown automaton *for $pref(t)$ accepts all tree patterns in prefix notation which match the tree t.*

Given a subject tree, first we construct a so-called deterministic *treetop PDA* for this tree in prefix notation, which accepts all tree patterns that match the subject tree and contain the root of the subject tree. The deterministic treetop PDA is defined as follows.

Definition 12. *Let t, r and $pref(t)$ be a tree, its root and its prefix notation, respectively. A* treetop pushdown automaton *for $pref(t)$ accepts all tree patterns in prefix notation which have the root r and match the tree t.*

The construction of the treetop PDA is described by the following algorithm. The treetop PDA is deterministic.

Algorithm 8. Construction of a treetop PDA for a tree t in prefix notation $pref(t)$.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: Treetop PDA $M_{pt}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. Create $M_{pt}(t)$ as $M_p(t)$ by Alg. 6.
2. Create a set $srms = \{ i : 1 \leq i \leq n,\ \delta(i-1, a, S) = (i, \varepsilon),\ a \in \mathcal{A}_0\}$. The abbreviation $srms$ stands for Subtree RightMost States.
3. For each state i, where $i = n-1, n-2, \dots, 1$, $\delta(i, a, S) = (i+1, S^p)$, $a \in \mathcal{A}_p$, create a new transition $\delta(i, S, S) = (l, \varepsilon)$ such that $(i, xy, S) \vdash^{+}_{M_p(t)} (l, y, \varepsilon)$ as follows:
   If $p = 0$, create a new transition $\delta(i, S, S) = (i+1, \varepsilon)$.
   Otherwise, if $p \geq 1$, create a new transition $\delta(i, S, S) = (l, \varepsilon)$, where l is the p-th smallest integer such that $l \in srms$ and $l > i$. Remove all j, where $j \in srms$, and $i < j < l$, from $srms$. □

Theorem 19. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_{pt}(t)$ constructed by Alg. 8 is a treetop PDA for $pref(t)$.*

Proof. In [10].

The nondeterministic tree pattern PDA for trees in prefix notation is constructed as an extension of the deterministic treetop PDA.

Algorithm 9. Construction of a nondeterministic tree pattern PDA for a tree t in prefix notation $pref(t)$.
Input: A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: Nondeterministic tree pattern PDA $M_{npt}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$.
Method:

1. Create $M_{npt}(t)$ as $M_{pt}(t)$ by Alg. 8.
2. For each state i, where $2 \leq i \leq n$, create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$. □

Theorem 20. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_{npt}(t)$ constructed by Alg. 9 is a tree pattern PDA for $pref(t)$.*

Proof. In [10].

The nondeterministic tree pattern PDA $M_{npt}(t)$ is again an acyclic input-driven PDA, and therefore can be determinised by Alg. 1 to an equivalent deterministic tree pattern PDA $M_{dpt}(t)$.

8 Finding Repeats in Trees

Efficient methods of finding various kinds of repeats in a string can be based on constructing and analysing string suffix trees or string suffix automata. This section briefly presents a simple method of finding various kinds of all repeats of subtrees in a given tree by constructing and analysing the subtree pushdown automaton for the tree.

Given a tree, the problem is to find all repeating subtrees of the tree and to compute kinds and positions of all occurrences of these subtrees. All repeats of subtrees and their properties are summarised in a subtree repeat table, which is defined by Defs. 13, 14 and 15. We define two versions of the subtree repeat table: the first, basic, version of the table contains basic information on repeats and its size is linear to the number of nodes of the tree. The second one, an extended subtree repeat table, contains also further information such as all the repeating subtrees in prefix notation, which can result in a larger table.

Definition 13. Let t be a tree over a ranked alphabet $\mathcal{A}$. A *subtree position set* $\text{sps}(st, t)$, where st is a subtree of t, is the set $sps(st, t) = \{i : pref(t) = x\ pref(st)\ y,\ x, y \in \mathcal{A}^*, i = |x| + 1\}$.

Definition 14. Let t be a tree over a ranked alphabet $\mathcal{A}$. Given a subtree st of t, *list of subtree repeats* $lsr(st, t)$ is a relation in $sps(st, t) \times \{F, S, Q\}$ defined as follows:

- $(i, F) \in lsr(st, t)$ iff $pref(t) = x\ pref(st)\ y$, $i = |x| + 1$, $x \neq x_1\ pref(st)\ x_2$,
- $(i, S) \in lsr(st, t)$ iff $pref(t) = x\ pref(st)\ y$, $i = |x| + 1$, $x = x_1\ pref(st)$,
- $(i, G) \in lsr(st, t)$ iff $pref(t) = x\ pref(st)\ y$, $i = |x| + 1$, $x = x_1\ pref(st)\ x_2$, $x_2 \in \mathcal{A}^+$.

Abbreviations F, S, and G stand for First occurrence of the subtree, repeat as a Square, and repeat with a Gap, respectively. In comparison with kinds of repeats in string [12,13], repeats of subtrees have no kind which would represent the overlapping of subtrees because any two different occurrences of the same subtree cannot overlap.

Definition 15. Given a tree t, the *basic subtree repeat table* $BSRT(t)$ is the set of all lists of subtree repeats $lsr(st, t)$, where st is a subtree with more than one occurrence in the tree t. The *extended subtree repeat table* $ESRT(t)$ is the set of all triplets $(sps(st, t), pref(st), lsr(st, t))$, where st is a subtree with more than one occurrence in the tree t.

Algorithm 10. Construction of the basic subtree repeat table for a tree t in prefix notation $pref(t)$.
Input: A tree t; prefix notation $pref(t) = a_1a_2 \dots a_n$, $n \geq 1$.
Output: Basic subtree repeat table $BSRT(t)$.
Method:

```
1. Initially, $BSRT(t) = \emptyset$.
2. Create $M_{npt}(t) = (\{0, 1, 2, \dots, n\}, \mathcal{A}, \{S\}, \delta, 0, S, \emptyset)$ by Alg. 9.
3. Let $Q'$ denote a set of states. Let $pdsl(q')$, where $q' \in Q'$, denote a set of
   pairs of integers (the abbreviation pdsl stands for the number of symbols $S$
   in the PushDown Store, and the Length of the subtree.)
4. $Q' = \{[0]\}$, $pdsl([0]) = \{(1, 0)\}$ and $[0]$ is an unmarked state.
5. (a) Select an unmarked state $q'$ from $Q'$ such that $q'$ contains the smallest
       possible state $q \in Q$, where $0 \leq q \leq n$.
   (b) For each $(0, l) \in pdsl(q')$ to $BSRT(t)$ add pairs $(x, Z)$, where
       $x = r - l$, $r \in q'$ and:
         i. $Z = F$ if $x$ is the smallest such number $x$,
        ii. $Z = S$ if $x - 1 \in q''$,
       iii. $Z = G$ otherwise.
   (c) If there is $v > 0$, $(v, w) \in pdsl(q')$, then for each input symbol $a \in \mathcal{A}$:
       Compute state $q'' = \{q : \delta(p, a, \alpha) = (q, \beta)$ for all $p \in q'\}$.
       If $q''$ is not in $Q'$ and $|q''| > 1$, then add $q''$ to $Q'$ and create $pdsl(q'') = \emptyset$.
       Add pairs $(j, k + 1)$, where $(i, k) \in pdsl(q')$, $i > 0$, $j = i + Arity(a) - 1$,
       to $pdsl(q'')$.
   (d) Set the state $q'$ as marked.
6. Repeat step 5 until all states in $Q'$ are marked.   □
```

Theorem 21. *Given a tree t with n nodes, Alg. 10 correctly constructs the basic subtree repeat table $BSRT(t)$ in time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$.*

Proof. In [11]. □

9 Conclusion

Basic arbology principles and algorithms have been presented. In this paper, particular algorithms were presented for prefix notation of trees; these algorithms can be easily modified also for postfix, bar prefix and bar postfix notations in the following way: instead of the pair of Theorems (2, 6), the pairs of Theorems (3, 7), (4, 8), and (5, 9), respectively, can be considered and the pushdown operations can be changed accordingly. For more detailed information on arbology, see [3].

Acknowledgement. I would like to thank Jan Janoušek for his help with creating this paper.

References

1. Aho, A.V., Ullman, J.D.: The theory of parsing, translation, and compiling. Prentice-Hall, Englewood Cliffs (1972)
2. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) STOC, pp. 202–211. ACM, New York (2004)
3. Arbology: www pages (2010), http://www.arbology.org/
4. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (Release October 12, 2007), http://www.grappa.univ-lille3.fr/tata
5. Flouri, T., Janoušek, J., Melichar, B.: Subtree matching by pushdown automata. Computer Science and Information Systems, ComSIS Consortium (To appear 2010)
6. Gecseg, F., Steinby, M.: Tree languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 3, pp. 1–68. Springer, Berlin (1997) (Beyond Words)
7. Hoffmann, C.M., O'Donnell, M.J.: Pattern matching in trees. J. ACM 29(1), 68–95 (1982)
8. Janoušek, J.: String suffix automata and subtree pushdown automata. In: Holub, J., Žďáreks, J. (eds.) Proceedings of the Prague Stringology Conference 2009, pp. 160–172. Czech Technical University, Prague (2009), http://www.stringology.org/event/2009
9. Janoušek, J., Melichar, B.: On regular tree languages and deterministic pushdown automata. Acta Inf. 46(7), 533–547 (2009)
10. Janoušek, J., Melichar, B.: Subtree and tree pattern pushdown automata for trees in prefix notation. Submitted for publication (2009)
11. Janoušek, J., Melichar, B.: Finding repeats of subtrees in a tree using pushdown automata (in preparation, 2010)
12. Melichar, B.: Repetitions in text and finite automata. In: Cleophas, I.L., Watson, B.W. (eds.) Proceedings of the Eindhoven FASTAR Days 2004. TU Eindhoven, The Netherlands, pp. 1–46 (2004)
13. Melichar, B., Holub, J., Polcar, J.: Text searching algorithms, http://stringology.org/athens/ (release November 2005)
14. Nowotka, D., Srba, J.: Height-deterministic pushdown automata. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 125–134. Springer, Heidelberg (2007)

Analysis of Communicating Automata

Anca Muscholl

LaBRI, University Bordeaux, France

Abstract. This extended abstract is a survey of some of the recent developments in the area of automated verification dedicated to the analysis of communicating automata.

Communicating automata are a fundamental computational model for concurrent systems, where processes cooperate via asynchronous message passing using unbounded channels. They are a popular model for representing and reasoning about communication protocols, and they have been used to define the semantics of standardized specification languages such as SDL. However, from the algorithmic point of view communicating automata are more challenging than other true concurrent models such as Petri nets: indeed, this model is Turing equivalent, in particular it subsumes Post tag systems [21]. Therefore, basic questions arising in formal verification, such as the reachability problem, are intractable.

Solving the reachability problem is actually the first step in tackling the more general *model-checking* problem, that consists in verifying that the model, i.e. the communicating automaton, satisfies a given property, usually described in some logical formalisms such as e.g. temporal logics [18]. In this setting, reachability is used for validating safety properties, stating that no bad state can be reached from the initial state. A more challenging and difficult problem is *synthesis*: here, the aim is to compute a model from a given specification. In this survey we will only report on the *closed synthesis* problem, where the model that is to be computed does not interact with any environment. In contrast, synthesis of *open systems*, i.e. systems that are embedded in an unpredictable environment, is even more intricate. The reason why synthesis is challenging here is that communicating automata are a concurrent model, thus the simplest instance of synthesis (i.e., the closed case) already amounts to compute the distribution of a sequential object, i.e. the specification. In the open case, the situation is even more challenging, since we need to solve some sort of concurrent games. In both cases, the existing techniques are rather sparse.

Starting with the model-checking problem, an important line of research was devoted to identify structural or behavioral restrictions on communicating automata that make them amenable to algorithmic methods. A first example are lossy channel systems, or equivalently, communicating automata where any number of messages can be lost, at any time. Lossy channel systems are a particular instance of well-structured transition systems [1,9] , so reachability was shown to be decidable [2], albeit of non-primitive recursive complexity [22]. On the other hand, liveness properties were shown to be undecidable [1].

A second line of research aimed at describing the set of reachable configurations of communicating automata, resp. approximations thereof, by some form of extended finite-state automata (called *symbolic representations*). The idea here is to manipulate a possibly infinite set of configurations by means of finite objects, such as finite automata or some suitable extensions.

Whereas both previous approaches emphasize the symbolic representation of the set of reachable configurations, an orthogonal approach based on *partial orders*, has been developed more recently. The partial order approach emphasizes concurrency and, in particular, the partially ordered events executed by a communicating automaton. Here, we are mainly interested e.g. in the reachability of control states, so that the memory (i.e., the channel contents), is handled only implicitly. Notice that this kind of event-based reasoning arises very naturally when communicating automata are viewed as sequential automata synchronizing over communication events.

The partial order approach was successfully applied for obtaining both model-checking algorithms, as well as synthesis algorithms, for so-called *existentially-bounded* finite state communicating automata [13]. The main idea here is to assume unbounded channels, but to consider only executions that can be rescheduled on (uniformly) bounded ones. A simple example illustrating the idea is a pair of processes, a producer and a consumer, where the producer keeps sending messages to the consumer. Since there is no control on the relative speed of these two processes, the channel should be of unlimited size. However, often it suffices to reason on executions where messages can be consumed without delay, i.e. on executions that can be scheduled with a channel of size one.

The partial order approach has been actually inspired by the study of automata and logics over Mazurkiewicz traces ([20], see also [8] for a textbook of the topic). The deepest result in the area of Mazurkiewicz traces is Zielonka's construction of distributed (trace) automata from sequential automata [24,23]. This sophisticated construction is the building brick for the synthesis of existentially bounded finite-state communicating automata in [10].

1 Basics

Communicating automata follow the simple paradigm of a network of automata cooperating asynchronously over point-to-point, fifo communication channels. They arise naturally as models for peer-to-peer interaction, as occurring e.g. in distributed protocols using asynchronous message passing.

We consider systems described by means of a fixed *communication network*, consisting of a (usually finite) set of concurrent *processes* $\mathcal{P}$, together with a set of *channels* $\mathrm{Ch} \subseteq \{(p,q) \in \mathcal{P}^2 \mid p \neq q\}$, that stand for point-to-point links. Following the classical definition [6], we exclude multiple channels between a pair of processes, as well as self-linking channels. However, this has no severe impact on the kind of results we will present. At best, it has an impact when one aims at classifying networks w.r.t. decidability of various verification questions. In this model, processes act either by point-to-point communication or by local actions. A send action denoted as $p!q(m)$ means that process p sends a message with content m to process q. A receive action denoted as $p?q(m)$ means that p receives from q a message with content m. Whenever we write $p!q$ and $q?p$, we will assume that $(p,q) \in \mathrm{Ch}$. A local action m on process p is denoted as $\ell_p(m)$. For a given (finite) set M of message contents, resp. local actions, and a process $p \in \mathcal{P}$, we define the set of p-local actions as $\Sigma_p = \{p!q(m), p?q(m), \ell_p(m) \mid m \in M\}$ and set $\Sigma = \bigcup_{p\in\mathcal{P}} \Sigma_p$.

A *communicating automaton* (*CA* for short) is a tuple $\mathcal{A} = \langle (\mathcal{A}_p)_{p\in\mathcal{P}}, F\rangle$ where

– each $\mathcal{A}_p = (S_p, \rightarrow_p, s_p^0)$ is a labeled transition system (LTS for short) with state space S_p, transition relation $\rightarrow_p \subseteq S_p \times \Sigma_p \times S_p$, and $s_p^0 \in S_p$ as initial state;
– $F \subseteq \prod_{p\in\mathcal{P}} S_p$ is a set of *global* final states.

We denote the product $S := \prod_{p\in\mathcal{P}} S_p$ as set of *global states*.

The behavior of a CA is defined as the behavior of an infinite-state LTS, by considering the possible (local) transitions on the set of configurations of the CA. A *configuration* of the CA $\mathcal{A}$ consists a global state $s \in S$, together with a word $w_{p,q} \in M^*$ for each channel $(p,q) \in \mathrm{Ch}$. We write $C = \langle s = (s_p)_{p\in\mathcal{P}}, w = (w_{p,q})_{(p,q)\in\mathrm{Ch}}\rangle$ for a configuration with global state $s \in S$ and channel contents $w \in (M^*)^{\mathrm{Ch}}$. The set of all configurations of $\mathcal{A}$ is denoted $\mathcal{C}_{\mathcal{A}}$ (or simply $\mathcal{C}$ when there is no risk of confusion). For any two configurations $C = \langle s, w\rangle, C' = \langle s', w'\rangle$ and any action $a \in \Sigma_p$ of $\mathcal{A}$, we say that C' is a *successor* of C (and write $C \xrightarrow{a} C'$, or simply $C \longrightarrow C'$ or $C' \in \mathrm{post}(C)$, when the action does not matter), if

– $s_p \xrightarrow{a}_p s'_p$ is a p-local transition, and $s'_q = s_q$ for all $q \neq p$,
– Send action: if $a = p!q(m)$, then $w'_{p,q} = w_{p,q}m$ (message m is inserted into the channel from p to q) and $w'_{r,s} = w_{r,s}$ for all $(r,s) \neq (p,q)$ (all other channels are unchanged).
– Receive action: if $a = p?q(m)$, then $w_{q,p} = mw'_{q,p}$ (message m is deleted from the channel from q to p) and $w'_{r,s} = w_{r,s}$ for all $(r,s) \neq (q,p)$ (all other channels are unchanged).
– Local action: if $a = \ell_m$, then $w = w'$.

A *run* of a CA $\mathcal{A}$ is (finite or infinite) sequence of transitions: $\rho = C_0 \xrightarrow{a_0} C_1 \xrightarrow{a_1} C_2 \cdots$, with $C_i \in \mathcal{C}_{\mathcal{A}}$ configurations and $a_i \in \Sigma$ actions. For a run ρ as above, we also write $C_0 \xrightarrow{*} C_n$.

We define *accepting* runs in the usual way, by referring to the global states. A finite run $\rho = C_0 \xrightarrow{a_0} C_1 \xrightarrow{a_1} \cdots C_n$ is accepting if $C_0 = \langle s^0, \boldsymbol{\varepsilon}\rangle$ and $C_n = \langle f, w\rangle$, where $\boldsymbol{\varepsilon}_{p,q} = \varepsilon$ for all $(p,q) \in \mathrm{Ch}$, $f \in F$ and $w \in (M^*)^{\mathrm{Ch}}$.

The *reachability set* of a CA $\mathcal{A}$, denoted Reach($\mathcal{A}$), is the set

$$\mathrm{Reach}(\mathcal{A}) := \{w \in (M^*)^{\mathrm{Ch}} \mid C_0 \xrightarrow{*} \langle f, w\rangle \text{ for some } f \in F\}\,.$$

The *language* of a CA $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^*$, is the set

$$\mathcal{L}(\mathcal{A}) = \{a_0a_1\cdots a_{n-1} \mid C_0 \xrightarrow{a_0} C_1 \xrightarrow{a_1} C_2 \cdots \xrightarrow{a_{n-1}} C_n \text{ is an accepting run}\}\,.$$

Remark 1. Notice that we did not impose in the definition of a communicating automaton $\mathcal{A} = \langle (A_p)_{p \in \mathcal{P}}, F \rangle$ any restriction on the local LTS $\mathcal{A}_p$. In general, we might be interested in various kinds of (possibly infinite-state) LTS, such as pushdown automata. However, a basic kind of CA is obtained by requiring that every $\mathcal{A}_p$ is a finite-state automaton, and then we denote $\mathcal{A}$ as *communicating finite-state machine* (CFM for short). Most of the research done in the past 15 years on CAs focused on CFMs, and we will concentrate on them in the next sections.

2 Symbolic Representations

The basic idea when using symbolic representations is to approximate in a finitary way behavior of a CFM. Often, such a computation is intended to capture the effect of iterating single loops. This leads to define *meta-transitions* and to use them to accelerate the usual fixpoint computation defining $\text{Reach}_{\mathcal{A}}$:

$$X := \{C_0\}, \quad X := X \cup \text{post}(X)$$

Queue-content Decision Diagrams (QDD for short) were proposed in [3] for describing (possibly infinite) sets of configurations of CFM by finite automata. With such representations, one can answer to various questions such as boundedness of channels or reachability. But of course, the method only offers a semi-algorithm for the reachability problem.

Let us assume that our network has k channels, with a fixed order on Ch. The channel contents $w = (w_i)_{i=1}^k$ of a configuration $\langle s, w \rangle$ of the CFM can be represented by a word $w_1 \# w_2 \# \cdots w_k \#$ (assuming that $\# \notin M$). A QDD is then a finite automaton reading words from $(M^* \#)^k$. Computing the effect of a loop means iterating a sequence $\sigma \in S^*$, that leads from a global state $s \in S$ back to s. The general idea is to start with $s, \sigma, \mathcal{B}$, where $\mathcal{B}$ is a QDD, and to compute the effect of σ^* on the set of configurations $\{\langle s, (w_i)_{i=1}^k \rangle \mid w_1 \# w_2 \# \cdots w_k \# \in \mathcal{L}(\mathcal{B})\}$. The paper [4] characterizes those sequences σ that preserve regularity, i.e., QDD representations, as *non-counting* sequences. This roughly means that such loops cannot send on two different channels. This paper also suggests a semi-algorithm for model-checking LTL properties.

The paper [5] goes beyond regular representations, introducing *Constrained Queue-content Decision Diagrams* (CQDD for short). CQDDs are restricted finite automata, extended by linear constraints on the frequency of transitions in a run. The main result of [5] is that the CQDD representation is preserved by the iteration of arbitrary loops.

3 Faulty Channels

Assuming that channels are imperfect, at least two types of faults may seem natural for real systems. *Lossy* machines are such that channels can loose an arbitrary number of messages, at any time. For machines with *insertion errors*, new messages can be inserted in channels, at any time. Although these two models have different flavor, the techniques used to manipulate them are quite similar, so that we will only consider lossy CFMs in the following.

Lossy CFMs (or *lossy channel systems*) represent a special instance of a more general class of infinite-state systems, known as *well-structured transition systems* (WSTS for short), [2,9]. The basic idea behind a WSTS $\langle \mathcal{S}, \longrightarrow \rangle$ with state space $\mathcal{S}$ is to use a *well quasi-order* (wqo for short) on $\mathcal{S}$ in order to manipulate certain infinite subsets of $\mathcal{S}$ symbolically. A wqo $\preceq$ on $\mathcal{S}$ is a well-founded preorder with no infinite anti-chain. What makes a transition system $\langle \mathcal{S}, \longrightarrow \rangle$ a WSTS is *monotonicity*: for every $s' \in \text{post}(s)$ and every $s_1 \in \mathcal{S}$ with $s \preceq s_1$, it is required that some $s'_1 \in \text{post}(s_1)$ exists such that $s' \preceq s'_1$.

Two basic properties are crucial for WSTS. The first one is that every *upward-closed*[1] subset $X \subseteq \mathcal{S}$ can be described by a *finite* set of minimal elements. The second property is that the predecessor relation preserves upward-closed sets. That is, $\text{pre}(X) := \{x \mid x \longrightarrow y \text{ for some } y \in X\}$ is upward-closed whenever X is upward-closed. As a consequence, reachability of upward-closed sets X can be decided by a backward algorithm, that computes in a fixpoint manner $\text{pre}^*(X)$. Intersecting the result with the set of initial configurations solves the reachability problem.

For lossy CFMs, the choice for a wqo is very natural. One starts with the subword ordering: for two words $x, y \in M^*$, let $x \preceq y$ if $x = x_1 \cdots x_n$ and $y = y_0 x_1 y_1 \cdots y_{n-1} x_n y_n$ for some $y_i \in M^*$. This wqo easily extends to M^k and then to configurations of the CFM. For two configurations $C = \langle s, w \rangle$, $C' = \langle s', w' \rangle$ we set $C \preceq C'$ if $s = s'$ and $w \preceq w'$.

This technique allows to decide e.g. control-state reachability for lossy CFMs. More complex properties, such as repeated reachability of a control state, are undecidable [2]. For deciding termination from an initial configuration, a different technique is employed, based on the computation of a finite reachability tree (forward algorithm). However, the more general problem of termination from *any* initial configuration, is undecidable [19]. From a different perspective, a recent paper [7] considered mixed architectures, where some channels are lossy and others are error-free, and characterized architectures with a decidable reachability question.

4 Partial Order Approach

An early line of work considered *universally bounded* CFMs. This property amounts to say that there exists a uniform bound B such that every run can be executed with channels of size B, no matter how events are scheduled. Equivalently, the number of transitory messages is at most B, at any time. Since the size of the communication channels is fixed uniformly, this constraint turns a CFM into a finite state device. Checking that a CFM is universally bounded is undecidable, in general. However if the bound B is given, and if the CFM $\mathcal{A}$ is

[1] X is upward-closed if $X = \{y \mid x \preceq y \text{ for some } x \in X\}$.

deadlock-free, then we can check in polynomial space whether $\mathcal{A}$ is universally B-bounded [11].

Being universally bounded leads immediately to a decision procedure for the model-checking problem, since we can transform the CFM into an (exponentially larger) finite automaton.

An even more important result concerns closed synthesis. Suppose that we are given a regular language $L \subseteq \Sigma^*$, that satisfies the following properties for some $B > 0$ (notice that these properties are decidable for a given B):

1. For every prefix w of a word from L, and every $(p, q) \in \mathrm{Ch}$, we have $|w|_{p!q} - |w|_{q?p} \leq B$.
2. Every word in L can induce a fifo run, which leads to a configuration where all channels are empty.
3. Whenever $w \in L$, we can swap adjacent actions in w that (1) do not belong to the same process and (2) do not form a matching send/receive pair, and the resulting word is still in L.

The main result of [14], later extended to infinite runs in [16], is a construction for transforming a regular language satisfying the three properties above into a universally B-bounded CFM. As mentioned previously, the challenge for such constructions is to distribute a sequential object, e.g. the finite automaton describing L. The techniques make heavy use of Mazurkiewicz trace theory and, in particular, of Zielonka's construction for distributed automata.

The drawback of models with universally bounded channels is the limited expressive power. Intuitively, universal channel bounds require message acknowledgments, which can be difficult to impose in general. For instance, basic protocols of producer-consumer type (such as e.g. the USB protocol) are not universally bounded, since the communication is one-way and does not allow acknowledgments. Therefore, a relaxation of this restriction on channels was proposed in [13,10]. The idea is to require an *existential bound* on channels. This means roughly that every CFM run must have *some* scheduling of events that respects the given channel bound (other schedules might exceed the bound). In other words, runs can be executed with bounded channels, provided that we schedule the events fairly. For instance, in a producer-consumer setting, the scheduling alternates between producer and consumer actions. This requirement is perfectly legitimate in practice, since real life protocols must be executable with limited communication channels. When a channel overflow happens, then the sender stops temporarily until some message is consumed from the queue.

For channel systems with existential bounds, the fundamental Kleene-Büchi equivalence of automata, logics and regular expressions was shown to hold in [10]. Automata means here CFMs, whereas logics refers to monadic second-order logics over partial orders. Regular expressions refer to a visual formalism that is very popular for early design of communication protocols, namely *message sequence charts*. Regarding model-checking, the complexity for existentially-bounded CFMs remains the same as in the case of universal bounds [13].

5 Conclusion and Outlook

This survey focused on two basic questions related to the automated verification of communicating automata, namely *model-checking* and *(closed) synthesis*. We presented three different approaches that allow to tackle these questions, usually offering some approximation of the behavior of this type of automata. We did not mention results specific to the ITU standard of *message sequence charts*, see [12] for some further references on this topic.

Concerning further work on communicating automata, let us mention some recent development. The extension of communicating automata by local pushdowns received recently attention justified by questions on the analysis of multithreaded programs or distributed software. Of course, the model is highly undecidable but still, methods like context-bounded model-checking or suitable restrictions on the network provide reasonable practical settings where reachability is decidable, [17,15].

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite state systems. In: LICS'96, pp. 313–323 (1996)
2. Abdulla, P.A., Jonsson, B.: Verifying programs with unreliable channels. Information and Computation 127(2), 91–101 (1996)
3. Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using QDDs. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 1–12. Springer, Heidelberg (1996)
4. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs. In: Van Hentenryck, P. (ed.) SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997)
5. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. Theor. Comp. Science 221(1-2), 211–250 (1999)
6. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM 30(2), 323–342 (1983)
7. 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)
8. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific, Singapore (1995)
9. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science 256(1-2), 63–92 (2001)
10. 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)
11. Genest, B., Kuske, D., Muscholl, A.: On communicating automata with bounded channels. Fundam. Inform. 80(1-3), 147–167 (2007)
12. Genest, B., Muscholl, A., Peled, D.: Message sequence charts. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) Lectures on Concurrency and Petri Nets. LNCS, vol. 3098, pp. 537–558. Springer, Heidelberg (2004)

13. Genest, B., Muscholl, A., Seidl, H., Zeitoun, M.: Infinite-state high-level MSCs: Model-checking and realizability. J. Comput. Syst. Sci. 72(4), 617–647 (2006)
14. Henriksen, J.G., Mukund, M., Kumar, K.N., Sohoni, M., Thiagarajan, P.: A theory of regular MSC languages. Information and Computation 202(1), 1–38 (2005)
15. Heußner, A., Leroux, J., Muscholl, A., Sutre, G.: Reachability analysis of communicating pushdown systems. In: Ong, L. (ed.) FoSSaCS 2010. LNCS, vol. 6014, pp. 267–281. Springer, Heidelberg (2010)
16. Kuske, D.: Regular sets of infinite message sequence charts. Information and Computation 187, 80–109 (2003)
17. La Torre, S., Madhusudan, P., Parlato, G.: Context-bounded analysis of concurrent queue systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
18. Manna, Z., Pnueli, A.: Verification of the concurrent programs: the temporal framework. In: Boyer, R., Moore, J. (eds.) The Correctness Problem in Computer Science, pp. 215–273. Academic Press, London (1981)
19. Mayr, R.: Undecidable problems in unreliable computations. Theor. Comp. Science 297(1-3), 337–354 (2003)
20. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus (1977)
21. Post, E.: Formal reductions of the combinatorial decision problem. American Journal of Mathematics 65(2), 197–215 (1943)
22. Schnoebelen, P.: Verifying lossy channel systems has nonprimitive recursive complexity. Information Processing Letters 83(5), 251–261 (2002)
23. Zielonka, W.: Asynchronous automata. In: Rozenberg, G., Diekert, V. (eds.) Book of Traces, pp. 175–217. World Scientific, Singapore (1995)
24. Zielonka, W.: Notes on finite asynchronous automata. R.A.I.R.O. — Informatique Théorique et Applications 21, 99–135 (1987)

Complexity of the Satisfiability Problem for a Class of Propositional Schemata

Vincent Aravantinos, Ricardo Caferra, and Nicolas Peltier

Grenoble University (LIG/CNRS)

Abstract. Iterated schemata allow to define infinite languages of propositional formulae through formulae patterns. Formally, schemata extend propositional logic with new (generalized) connectives like e.g. $\bigwedge_{i=1}^{n}$ and $\bigvee_{i=1}^{n}$ where n is a *parameter*. With these connectives the new logic includes formulae such as $\bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1})$ (atoms are of the form P_1, P_{i+5}, P_n, ...). The satisfiability problem for such a schema S is: "Are all the formulae denoted by S valid (or satisfiable)?" which is undecidable [2]. In this paper we focus on a specific class of schemata for which this problem is decidable: *regular schemata*. We define an automata-based procedure, called SchAUT, solving the satisfiability problem for such schemata. SchAUT has many advantages over procedures in [2,1]: it is more intuitive, more concise, it allows to make use of classical results on finite automata and it is tuned for an efficient treatment of regular schemata. We show that the satisfiability problem for regular schemata is in 2-EXPTIME and that this bound is tight for our decision procedure.

1 Introduction

In applied logics (theorem proving, formal methods in programming, proof assistants, ...) one often encounters sets of structurally similar propositional formulae. For instance, in the field of circuit verification, consider the formalization of a sequential adder circuit i.e. a circuit that takes as input two n-bit vectors and computes their sum. Such a circuit is the composition of n 1-bit adders. For instance an 8-bit adder, a 16-bit adder and a 32-bit adder all share the same structure, and can be specified by structurally similar propositional formulae:

$$Adder \stackrel{\text{def}}{=} \bigwedge_{i=1}^{n} Sum_i \wedge \bigwedge_{i=1}^{n} Carry_i \wedge \neg C_1$$

where n is the number of bits, $Sum_i \stackrel{\text{def}}{=} S_i \Leftrightarrow (A_i \oplus B_i) \oplus C_i$, $Carry_i \stackrel{\text{def}}{=} C_{i+1} \Leftrightarrow (A_i \wedge B_i) \vee (B_i \wedge C_i) \vee (A_i \wedge C_i)$, $\oplus$ denotes the exclusive or, $A_1, \dots, A_n$ (resp. $B_1, \dots, B_n$) the bits of the first (resp. second) operand of the circuit, $S_1, \dots, S_n$ the output (the **S**um), and $C_1, \dots, C_n$ the intermediate **C**arries.

The *Adder* specification is not a propositional formula, it is a more abstract object that actually represents an *infinite* set of propositional formulae (one formula for every positive number assigned to n). The idea of defining formal *languages* to represent such sets then naturally arises. Propositional iterated schemata permit to express those sets through a syntax similar to the one of the above example i.e. connectives $\bigwedge_{i=1}^{n}$ or $\bigvee_{i=1}^{n}$ are allowed. Such constructions are very common and indeed they arise naturally in several problems: e.g. the pigeonhole, the colouring of a graph or the n queens.

The challenge is not only to represent those sets of formulae but also to *reason* with them, e.g. many properties of the above circuit can be easily expressed as an iterated schema T_n (e.g. $(\bigwedge_{i=1}^{n} \neg B_i) \Rightarrow \bigwedge_{i=1}^{n}(A_i \Leftrightarrow S_i)$ states that 0 is a neutral element of *Adder*). Then one wants to prove that every propositional formula in $\{T_1, T_2, \dots\}$ is valid or unsatisfiable ($T_1, T_2, \dots$ are called *instances* of T_n). As usual in automated theorem proving, we focus on unsatisfiability. This problem is thus called the *satisfiability problem* for schemata. It is equivalent to proving $\forall n \in \mathbb{N}. \neg T_n$, and is thus a hard problem which generally requires mathematical induction. Actually it has been shown to be undecidable in [2].

In [2] we defined a tableaux-based procedure called STAB designed to solve the satisfiability problem for schemata. As the problem is undecidable this procedure does not terminate in general, but we proved that it always terminates for a particular class of schemata, called *regular*.

The purpose of the present paper is to turn this termination proof into an effective decision procedure for the class of regular schemata. We describe an automata-based decision procedure which is both more efficient and more convenient than the procedure of [2]. A fine analysis of this procedure allows us to prove that the satisfiability problem for regular schemata is in 2-EXPTIME.

The paper is organized as follows: Section 2 defines regular schemata, their syntax and semantics. Section 3 defines SchAUT, a procedure which, for a schema S, constructs an automaton which is empty iff *all* instances of S are unsatisfiable. In Section 4 we study the complexity of SchAUT. Section 5 concludes the paper and presents a short overview of related and future work.

Due to space restrictions, some of the proofs are omitted.

2 Regular Schemata

We adopt the following conventions: integer variables are denoted by n (for parameters) or i, j (for bound variables), integers by N, M_1, M_2, K, arithmetic expressions by e, f, propositional symbols by P, Q, schemata by S, T and sets of schemata by $\mathcal{S}$.

2.1 Syntax and Semantics

The set of *linear arithmetic expressions* is built as usual on the signature 0, s, $+$, $-$ and a set of integer variables $\mathcal{V}$, modulo the standard properties of the arithmetic symbols (e.g. $s(0) + i + s(s(0))$ is assumed to be the same as $i + s(s(s(0)))$). For $N > 0$, $s^N(0)$ is simply written N.

Propositional formulae are built as usual on a set of atoms and $\bot, \top$ using the propositional connectives $\vee, \wedge, \neg$. $A \Rightarrow B$, $A \Leftrightarrow B$ and $A \oplus B$ are shorthands for $\neg A \vee B$, $(A \Rightarrow B) \wedge (B \Rightarrow A)$ and $\neg(A \Leftrightarrow B)$ respectively.

Definition 1 (Regular Schemata). *Let $\mathcal{P}$ be a set of* propositional symbols*, n an integer variable, called the* parameter*, and $M_1, M_2 \in \mathbb{Z}$. Let $\mathcal{R}_{M_1}^{n-M_2}$ be the set of propositional formulae whose atoms are of one of the following forms:*

- *P_{n+K} or P_K (called* indexed propositions*) where $P \in \mathcal{P}$ and $K \in \mathbb{Z}$.*
- *$\bigwedge_{i=M_1}^{n-M_2} \phi$ or $\bigvee_{i=M_1}^{n-M_2} \phi$ (called* iterations*) where $i \in \mathcal{V}$, $i \neq n$ and ϕ is a propositional formula whose atoms are of the form P_{i+K} or P_K (also called* indexed propositions*) where $P \in \mathcal{P}$ and $K \in \mathbb{Z}$. i is called a* bound variable*.*

An element of $\mathcal{R}_{M_1}^{n-M_2}$ for any $n \in \mathcal{V}$, $M_1, M_2 \in \mathbb{Z}$ is called a regular schema*. A schema which is just an indexed proposition or its negation is called a* literal*.*

Example 1. Consider *Adder* as defined in the Introduction: *Adder* $\in \mathcal{R}_1^{n-0}$, hence *Adder* is a regular schema. $A_i, B_i, C_i, S_i, C_{i+1}, C_1$ are indexed propositions, $\bigwedge_{i=1}^{n} Sum_i$ and $\bigwedge_{i=1}^{n} Carry_i$ are iterations, n is the parameter.

A *substitution* maps integer variables to linear arithmetic expressions, $[e/i]$ denotes the substitution mapping i to e. The application of a substitution σ to an arithmetic expression e, written $e\sigma$, is defined as usual and naturally extended to schemata and sets of schemata. For $S \in \mathcal{R}_{M_1}^{n-M_2}$ $(M_1, M_2 \in \mathbb{Z}, n \in \mathcal{V})$ and $N \in \mathbb{Z}$, $S\,[N/n]$ contains no occurrence of n. In particular the upper bound $n-M_2$ of iterations is now an integer (equal to $N - M_2$) as we considered arithmetic expressions modulo the standard arithmetic properties. Then $S\,[N/n]$ is turned into a propositional formula through the following rewrite system $\mathcal{R}$:

$$\bigwedge_{i=M_1}^{M_2'} S \to \left(\bigwedge_{i=M_1}^{M_2'-1} S\right) \wedge S\,[M_2'/i] \quad \text{if } M_1 \leq M_2' \qquad \bigwedge_{i=M_1}^{M_2'} S \to \top \quad \text{if } M_1 > M_2'$$

$$\bigvee_{i=M_1}^{M_2'} S \to \left(\bigvee_{i=M_1}^{M_2'-1} S\right) \vee S\,[M_2'/i] \quad \text{if } M_1 \leq M_2' \qquad \bigvee_{i=M_1}^{M_2'} S \to \bot \quad \text{if } M_1 > M_2'$$

The length of every iteration strictly decreases at each rewriting step thus $\mathcal{R}$ terminates. Furthermore the rules are orthogonal so $S\,[N/n]$ has a unique normal form by $\mathcal{R}$, written $S\,[N/n]\!\downarrow$. $S\,[N/n]\!\downarrow$ is always a propositional formula, called an *instance* of S, e.g. $\left(\bigwedge_{i=1}^{3} P_i\right)\Big\downarrow = P_1 \wedge P_2 \wedge P_3$ (we always simplify $\top \wedge \phi$ into ϕ). For a *set* of schemata $\mathcal{S}$: $\mathcal{S}\,[N/n]\!\downarrow \overset{\text{def}}{=} \{S\,[N/n] \mid S \in \mathcal{S}\}$.

The semantics for propositional logic are the same as usual. A *model* of a propositional formula ϕ is a mapping of its propositional variables into $\{true, false\}$ making ϕ true. A propositional formula is *unsatisfiable* iff it has no model. A finite set of propositional formulae Φ is interpreted as $\bigwedge_{\phi \in \Phi} \phi$.

Definition 2 (Schemata Semantics). *Consider $n \in \mathcal{V}$, $M_1, M_2 \in \mathbb{Z}$ and $S \in \mathcal{R}_{M_1}^{n-M_2}$. The* language *of S, written $\mathcal{L}(S)$, is defined as: $\mathcal{L}(S) \overset{def}{=} \{\, S\,[N/n]\!\downarrow \mid N \in \mathbb{Z}, N \geq M_1 + M_2 - 1\}$. $\mathcal{L}(S)$ contains only propositional formulae. A schema is* unsatisfiable *iff every formula of its language is (propositionally) unsatisfiable.*[1]

[1] Notice that if S does not contain iterations, then $S \in \mathcal{R}_{M_1}^{n-M_2}$ for any M_1, M_2, hence possibly various semantics. This is not a problem but, for the sake of clarity, we assume that all schemata contain iterations.

Example 2. Let $S = P_1 \wedge \bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1}) \wedge \neg P_{n+1}$. $\mathcal{L}(S) = \{P_1 \wedge \neg P_1,\ P_1 \wedge (P_1 \Rightarrow P_2) \wedge \neg P_2, P_1 \wedge (P_1 \Rightarrow P_2) \wedge (P_2 \Rightarrow P_3) \wedge \neg P_3, \dots\}$. S is clearly unsatisfiable.

For $\mathcal{R}_{M_1}^{n-M_2}$, the linear expression $n - M_2 - M_1 + 1$ is called the *length* of iterations. The semantics given here are a bit different from the ones of our previous works: rather than $N \in \mathbb{Z}$, we impose $N \geq M_1 + M_2 - 1$ i.e. the length of iterations must be positive. Indeed allowing negative lengths does not seem useful in the context of regular schemata where all iterations have the same bounds. More important, it allows to use mathematical induction on the length of iterations.

2.2 Additional Definitions

We will use the two following transformations of schemata: *grounding* amounts to consider that the length of iterations in a schema is null, and *unfolding* extracts the last rank of each iteration.

Definition 3. *Consider $S \in \mathcal{R}_{M_1}^{n-M_2}$ for some $n \in \mathcal{V}$ and $M_1, M_2 \in \mathbb{Z}$.*

The grounding *of S, written $\mathcal{G}(S)$, is defined as $\mathcal{G}(S) \overset{def}{=} S\,[M_2 + M_1 - 1/n]\!\downarrow$*

The unfolding *of S, written $\mathcal{U}(S)$, is obtained by substituting each iteration $\bigoplus_{i=M_1}^{n-M_2} T$ by $T\,[n - M_2/i] \oplus \bigoplus_{i=M_1}^{n-M_2-1} T$ (where $(\oplus, \bigoplus) \in \{(\wedge, \bigwedge), (\vee, \bigvee)\}$) and then substituting all occurrences of n by $n+1$.*

Example 3. Consider S as in Example 2, then $\mathcal{G}(S) = P_1 \wedge \neg P_1$ and $\mathcal{U}(S) = P_1 \wedge \bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1}) \wedge (P_{n+1} \Rightarrow P_{n+2}) \wedge \neg P_{n+2}$.

Let $S \in \mathcal{R}_{M_1}^{n-M_2}$ then: $\mathcal{E}_{\mathbb{Z}}(S) \overset{\text{def}}{=} \{K \mid K \in \mathbb{Z}, P_K$ occurs in $S, P \in \mathcal{P}\}$, $\mathcal{E}_{n+K}(S) \overset{\text{def}}{=} \{K \mid K \in \mathbb{Z}, P_{n+K}$ occurs in $S, P \in \mathcal{P}\}$, $\mathcal{E}_{i+K}(S) \overset{\text{def}}{=} \{K \mid K \in \mathbb{Z}, P_{i+K}$ occurs in $S, P \in \mathcal{P}, i$ bound in $\mathcal{S}\}$. For $K_1, K_2 \in \mathbb{Z}$ we write $n + K_1 \preceq n + K_2$ iff $K_1 \leq K_2$.

Proposition 1. *Consider a literal of index $n+K$ occurring in $\mathcal{U}(S)$ where $K \in \mathbb{Z}$, $S \in \mathcal{R}_{M_1}^{n-M_2}$, then $n + K \preceq n + 1 + \max[\max(\mathcal{E}_{i+K}(S)) - M_2, \max(\mathcal{E}_{n+K}(S))]$.*

Proof. By Definition 1, no literal of the form P_{n+K} can occur in an iteration $\bigoplus_{i=M_1}^{n-M_2-1} T$ and thus neither in $(\bigoplus_{i=M_1}^{n-M_2-1} T)\,[n+1/n]$. So by Definition 3, P_{n+K} is either a literal of $T\,[n - M_2/i]\,[n+1/n]$ for some T, or a literal of $S\,[n+1/n]$ (i.e. P_{n+K-1} occurs in S before the unfolding). In the first case we have $n + K \preceq n + 1 - M_2 + \max(\mathcal{E}_{i+K}(S))$, and in the second case $n + K \preceq n + 1 + \max(\mathcal{E}_{n+K}(S))$. □

Finally we extend the notion of "pure literal", widely used in propositional theorem proving, to schemata. From now on, w.l.o.g. we assume that all schemata are in negation normal form (i.e. all negations appear in front of an indexed proposition, this form is easily obtained using De Morgan's laws).

Definition 4. *A literal P_e (resp. $\neg P_e$) is* pure *in S iff for all $N \in \mathbb{N}$, $\neg P_e\,[N/n]$ (resp. $P_e\,[N/n]$) does not occur positively in $S\,[N/n]\!\downarrow$.*

Example 4. Consider S as in Example 2. Remember that $P_i \Rightarrow P_{i+1}$ denotes $\neg P_i \vee P_{i+1}$. Then P_{n+1} is pure in S, however $\neg P_{n+1}$ is not (take any instance of S). For every $K > 0$, P_{n+1+K} and $\neg P_{n+1+K}$ are pure in S (neither occurs in any instance of S). Similarly, for every $K \in \mathbb{N}$, P_{-K} and $\neg P_{-K}$ are pure in S.

The following proposition gives a way to decide whether a literal is pure or not:

Proposition 2. *Consider $S \in \mathcal{R}_{M_1}^{n-M_2}$. A literal P_e (resp. $\neg P_e$) is pure in S iff:*

- *$e \in \mathbb{Z}$ and for every $\neg P_f$ (resp. P_f) in S:*
  - *if $f \in \mathbb{Z}$ then $f \neq e$.*
  - *if $f = n + K$ where $K \in \mathbb{Z}$ then $M_1 + M_2 - 1 + K > e$.*
  - *if $f = i + K$ where $i \in \mathcal{V}$, $K \in \mathbb{Z}$ then $M_1 + K > e$.*
- *or $e = n + K$ for some $K \in \mathbb{Z}$ and for every P_f (resp. $\neg P_f$) in S:*
  - *if $f \in \mathbb{Z}$ then $f < M_1 + M_2 - 1 + K$.*
  - *if $f = n + K'$ where $K' \in \mathbb{Z}$ then $K' \neq K$.*
  - *if $f = i + K'$ where $i \in \mathcal{V}$, $K' \in \mathbb{Z}$ then $-M_2 + K' < K$.*

Proof. Suppose that P_e is pure in S and consider $N \in \mathbb{Z}$ s.t. $N \geq M_1 + M_2 - 1$. We know that $\neg P_{e[N/n]}$ does not occur in $S[N/n]\downarrow$ i.e. for every $\neg P_F$ occurring in $S[N/n]\downarrow$, $F \neq e[N/n]$. Now F comes from f in a literal $\neg P_f$ of S, by a rewriting of $S[N/n]$ by $\mathcal{R}$. The proof is done by inspection of the different cases for e and f listed above. The case $\neg P_e$ is similar. The converse is trivial. □

Determining whether a literal is pure or not in a regular schema is then decidable: one just has to go through all the indexed propositions of S and check the corresponding inequality. One can also solve this problem by stating its equivalence with the unsatisfiability of a Presburger arithmetic formula [2]. But Proposition 2 shows that, in the case of regular schemata, this problem is linear w.r.t. the size of the schema (this is not the case of Presburger arithmetic which is doubly exponential [9]). This is essential for the complexity analysis in Section 4.

3 Deciding the Unsatisfiability of Regular Schemata

In contrast to the tableaux-based approach of [2] we use in this paper an automata based method. Although the new procedure is actually very similar, it is more elegant and more convenient from a theoretical point of view, because we can rely on well-known properties of finite automata. More precisely we show that for every schema S one can construct an automaton accepting exactly the set of integers N s.t. $S[N/n]\downarrow$ is satisfiable (in full rigor the set of integers the automaton accepts is the one corresponding to the lengths of iterations, i.e. $N - M_2 - M_1 + 1$). Then the desired result follows from the decidability of the emptiness problem in finite automata. The automata approach has the advantage that it gives an explicit presentation of the possible set of values of n (rather than just a 'yes' or 'no' answer). Furthermore, the states corresponding to the same formula are implicitly shared which makes the handling of *looping* (see [2]) very natural. In addition this improves the efficiency of the procedure without having to introduce any additional optimisation machinery.

We recall the well-known notion of non-deterministic finite automaton [12]:

Definition 5. *Let Σ be a finite alphabet. A* non-deterministic finite automaton *is a tuple $\langle \mathcal{Q}, \mathcal{I}, \mathcal{F}, \mathcal{T} \rangle$ where $\mathcal{Q}$ is a finite set of* states*, $\mathcal{I} \subseteq \mathcal{Q}$ is a set of* initial states*, $\mathcal{F} \subseteq \mathcal{Q}$ is a set of* final states*, $\mathcal{T} \subseteq \mathcal{Q} \times \Sigma \cup \{\epsilon\} \times \mathcal{Q}$ is a set of* transitions*.*

Given a word w on Σ, a run *of $\mathcal{A}$ on w from q_0 is a finite sequence of states $r = (q_0, \dots, q_K)$ s.t. for every $i \in 0..K$, there is $\alpha_i \in \Sigma \cup \{\epsilon\}$ s.t. (q_i, α_i, q_{i+1}) is a transition of $\mathcal{A}$ and $w = \alpha_0 \dots \alpha_K$. r is* accepting *iff $q_K \in \mathcal{F}$. w is* accepted *by $\mathcal{A}$ iff it has an accepting run on $\mathcal{A}$ from an initial state.*

We set $\Sigma = \{s, 0\}$: $N \in \mathbb{N}$ is represented by $s \dots s0$, where s occurs N times. Consider $S \in \mathcal{R}_{M_1}^{n-M_2}$, we describe the construction of an automaton $\mathcal{A}_S$ s.t. it accepts K iff $S\left[M_2 + M_1 + K - 1/n\right]\downarrow$ is satisfiable. This instantiation might look surprising as it would be more natural to take directly a value for n. It is better understood if we think of it as $S\left[K/\operatorname{length}(S)\right]\downarrow$ (which is of course a notation abuse), indeed $n = \operatorname{length}(S) + M_2 + M_1 - 1$. This allows to get rid of arithmetic constraints on n, which were needed in [2]. Finally the states of $\mathcal{A}_S$ are sets of regular schemata: those are the states *themselves* and not just labels, else we would have to explicitly identify two states sharing the same labels.

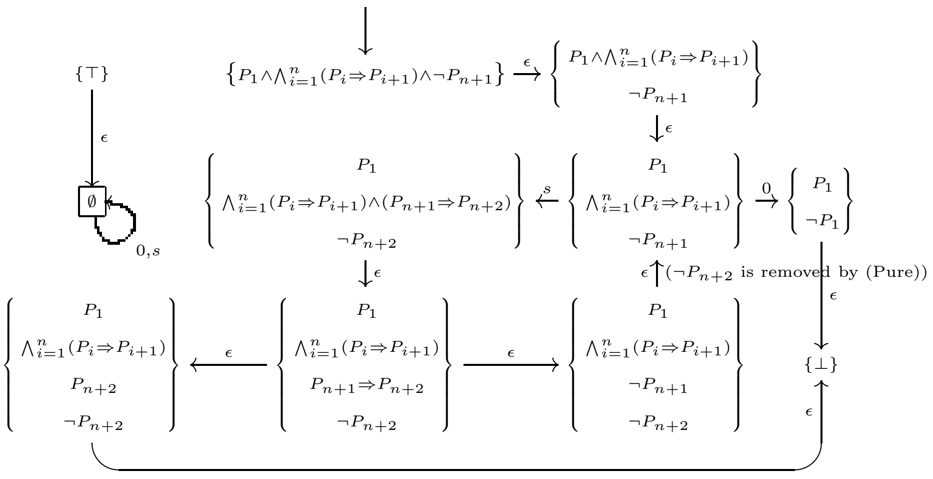
Definition 6 (SchAUT). *Consider $S \in \mathcal{R}_{M_1}^{n-M_2}$. The construction of $\mathcal{A}_S$ is described by a rewrite system: $\mathcal{S} \overset{x}{\Longrightarrow} \mathcal{S}_1$ & $\mathcal{S} \overset{y}{\Longrightarrow} \mathcal{S}_2$ denotes the rewrite rule $\langle \mathcal{Q} \cup \{\mathcal{S}\}, \mathcal{I}, \mathcal{F}, \mathcal{T} \rangle \rightarrow \langle \mathcal{Q} \cup \{\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2\}, \mathcal{I}, \mathcal{F}, \mathcal{T} \cup \{(\mathcal{S}, x, \mathcal{S}_1), (\mathcal{S}, y, \mathcal{S}_2)\} \rangle$ i.e. if $\mathcal{S}$ is a state of the rewritten automaton $\mathcal{A}$ then we add the states $\mathcal{S}_1, \mathcal{S}_2$ and the transitions $(\mathcal{S}, x, \mathcal{S}_1)$ and $(\mathcal{S}, y, \mathcal{S}_2)$ to $\mathcal{A}$. Similarly $\mathcal{S} \overset{x}{\Longrightarrow} \mathcal{S}'$ is a shorthand for $\langle \mathcal{Q} \cup \{\mathcal{S}\}, \mathcal{I}, \mathcal{F}, \mathcal{T} \rangle \rightarrow \langle \mathcal{Q} \cup \{\mathcal{S}, \mathcal{S}'\}, \mathcal{I}, \mathcal{F}, \mathcal{T} \cup \{(\mathcal{S}, x, \mathcal{S}')\} \rangle$. Let $\mathcal{A}_S$ be an (arbitrary) normal form of $\langle \{\{S\}\}, \{\{S\}\}, \{\emptyset\}, \{(\emptyset, 0, \emptyset), (\emptyset, s, \emptyset), (\{\top\}, \epsilon, \emptyset)\} \rangle$ by the following rewrite system, called* SchAUT *(for* ***Sch****ema* ***AUT****omaton):*

$$
\begin{array}{ll}
\mathcal{S} \cup \{S_1 \wedge S_2\} \overset{\epsilon}{\Longrightarrow} \mathcal{S} \cup \{S_1, S_2\} & (\wedge) \\
\mathcal{S} \cup \{S_1 \vee S_2\} \overset{\epsilon}{\Longrightarrow} \mathcal{S} \cup \{S_1\} \text{ \& } \mathcal{S} \cup \{S_1 \vee S_2\} \overset{\epsilon}{\Longrightarrow} \mathcal{S} \cup \{S_2\} & (\vee) \\
\mathcal{S} \cup \{P_e, \neg P_e\} \overset{\epsilon}{\Longrightarrow} \{\bot\} & (Closure) \\
\mathcal{S} \cup \{P_e\} \overset{\epsilon}{\Longrightarrow} \mathcal{S} \text{ if } P_e \text{ is pure in } \mathcal{S} & (Pure) \\
\mathcal{S} \overset{0}{\Longrightarrow} \{\mathcal{G}(S) \mid S \in \mathcal{S}\} \text{ \& } \mathcal{S} \overset{s}{\Longrightarrow} \{\mathcal{U}(S) \mid S \in \mathcal{S}\} \text{ if } \mathcal{S} \notin \{\emptyset, \{\top\}, \{\bot\}\} & (Decrease)
\end{array}
$$

$\mathcal{G}(S)$ and $\mathcal{U}(S)$ are defined in Definition 3. All schemata of $\mathcal{A}_S$ trivially belong to $\mathcal{R}_{M_1}^{n-M_2}$. We assume that at most one rule is applied on a given state and that the (Decrease) rule is applied only if no other rule can be applied.

Example 5. Consider S as in Example 2, then a possible $\mathcal{A}_S$ is given in Figure 1. $\emptyset$ is not reachable hence $\mathcal{A}_S$ is empty (and, as we shall see, S is unsatisfiable).

Notice that SchAUT terminates iff the set of states and transitions added by the rules is finite. When SchAUT terminates there may be various normal forms, however it is easily seen that those mainly differ by the order of the ϵ-transitions and the order in which the formulae are decomposed. A unique normal form can be obtained by imposing a priority among ϵ-rules and among formulae. This is not done for the sake of simplicity and generality. Properties of $\mathcal{A}_S$ will now be proved: they do not need termination which will be proved in Section 4.



$\{\top\}$
$\{P_1 \wedge \bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1}) \wedge \neg P_{n+1}\}$
ϵ
$\emptyset$
$0, s$
s
0
$\{P_1, \neg P_1\}$
$(\neg P_{n+2}$ is removed by (Pure))
$\{\bot\}$
P_1
$\bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1}) \wedge (P_{n+1} \Rightarrow P_{n+2})$
$\bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1})$
P_{n+2}
$P_{n+1} \Rightarrow P_{n+2}$
$\neg P_{n+1}$
$\neg P_{n+2}$

Fig. 1. Example 5 – $\mathcal{A}_{P_1 \wedge \bigwedge_{i=1}^{n}(P_i \Rightarrow P_{i+1}) \wedge \neg P_{n+1}}$ (the final state is framed)

Proposition 3. *Let S be a schema. A state of $\mathcal{A}_S$ either has been applied one of the rules, or it belongs to $\{\emptyset, \{\top\}, \{\bot\}\}$.*

A state of $\mathcal{A}_S$ is a $\wedge$-*state* (resp. $\vee$-*state*, *closure state*, *pure state*, *decrease state*) iff the rule ($\wedge$) (resp. ($\vee$), (Closure), (Pure), (Decrease)) applied to it. An $\vee$, $\wedge$ or pure state is called a *propositional* state. From now on for a decrease state $\mathcal{S}$, $\mathcal{S}^0$ and $\mathcal{S}^s$ denote the states s.t. $(\mathcal{S}, 0, \mathcal{S}^0)$ and $(\mathcal{S}, s, \mathcal{S}^s)$ are transitions of $\mathcal{A}_S$.

Theorem 1 (Main Property). *Let $S \in \mathcal{R}_{M_1}^{n-M_2}$. $S[M_1 + M_2 + K - 1/n]$ is satisfiable iff $\mathcal{A}_S$ accepts K.*

Notice that Theorem 1 is more general than the soundness and completeness result in [2] because the constructed automaton provides a description of the set of natural numbers N s.t. $S\,[N/n]\!\downarrow$ is satisfiable, whereas a non closed tableau only guarantees the existence of such an N. Furthermore, we do not rely on an external procedure for deciding the satisfiability of arithmetic formulae.

This result gives some clues for constructing a more general automaton that not only takes integers as input but also *interpretations*. This gives a way to provide explicitly a model for a particular instance of a schema. The method is not presented here due to space restrictions.

4 Termination and Complexity

The (Decrease) rule, through unfolding, replaces n by $n + 1$, thus an infinite number of distinct literals can be obtained. For instance from $\bigwedge_{i=1}^{n} P_i$ we can get $P_{n+1} \wedge \bigwedge_{i=1}^{n} P_i$, $P_{n+2} \wedge P_{n+1} \wedge \bigwedge_{i=1}^{n} P_i$, etc. In this section we show that all these literals can eventually be removed by the rule (Pure) which ensures termination. Furthermore we provide a bound on the number of distinct states in $\mathcal{A}_S$, which allows us to obtain a complexity bound for SchAut.

4.1 Maximal Number of States

Consider $n \in \mathcal{V}$, $M_1, M_2 \in \mathbb{Z}$. In the following, we assume that $S \in \mathcal{R}_{M_1}^{n-M_2}$. The *size* of S, written $\#S$, is the number of symbols in S (numbers occurring in schemata are encoded as bit strings) and the size of a state $\mathcal{S}$ (i.e. a set of schemata) is $\#\mathcal{S} \overset{\text{def}}{=} \sum_{T \in \mathcal{S}} \#T$.

Proposition 4. *Consider a literal of index $K \in \mathbb{Z}$ occurring in some decrease state. Then* $\min(\mathcal{E}_{\mathbb{Z}}(S)) \leq K \leq \max(\mathcal{E}_{\mathbb{Z}}(S))$.

Proof. The only literals whose index belong to $\mathbb{Z}$ are those of S and those introduced by SchAUT through a grounding (in the (Decrease) rule). As we consider a decrease state, no grounding has already occurred (otherwise it is easily seen that all other rules would have applied until we get $\emptyset$, $\{\top\}$ or $\{\bot\}$). So it only remains the literals of S and the result follows from the definition of $\mathcal{E}_{\mathbb{Z}}(S)$. $\square$

For $N \in \mathbb{N}$, an *N-unfolded state* is a decrease state $\mathcal{S}$ of $\mathcal{A}_S$ s.t. there is a run $r = (\{S\}, \ldots, \mathcal{S})$ where there are exactly N decrease states before $\mathcal{S}$ in r. Intuitively there is a run with N unfoldings before $\mathcal{S}$ is reached. N is not unique in general: if S occurs in a cycle then there exist several N s.t. S is an N-unfolded state. Notice that every decrease state is an N-unfolded state for some N as there must be a run from $\{S\}$ to this state by construction of $\mathcal{A}_S$.

Proposition 5. *Consider a literal of index $n+K$ occurring in some N-unfolded state for some $K \in \mathbb{Z}$ and $N \geq 1$. Then* $n + K \preceq \max[n - M_2 + N + \max(\mathcal{E}_{i+K}(S)), n + N + \max(\mathcal{E}_{n+K}(S)))]$.

Proof. The (Decrease) rule is the only rule that introduces new literals (by grounding and unfolding). Besides, only an unfolding may have introduced a literal of the form P_{n+K} (there is no occurrence of n after a grounding). So we get the result by induction on N and by Proposition 1 which shows that an unfolding increases the index of every P_{n+K} by 1 (as n is replaced by $n+1$). $\square$

We set $N_0 \overset{\text{def}}{=} \max[0,\ \max(\mathcal{E}_{\mathbb{Z}}(S)) - \max(\mathcal{E}_{i+K}(S)) - M_1 + 1,\ \max(\mathcal{E}_{\mathbb{Z}}(S)) - \max(\mathcal{E}_{n+K}(S)) - M_1 - M_2 + 1,\ \max(\mathcal{E}_{i+K}(S)) - M_2 - \max(\mathcal{E}_{n+K}(S))]$.

Lemma 1. *Consider a literal of index $n + K$ occurring in some N-unfolded state $\mathcal{S}$ for some $K \in \mathbb{Z}$ and $N \geq 1$. Then* $n + K \preceq \max[n - M_2 + N_0 + \max(\mathcal{E}_{i+K}(S)), n + N_0 + \max(\mathcal{E}_{n+K}(S))]$.

Proof. Let $M \overset{\text{def}}{=} \max[n - M_2 + N_0 + \max(\mathcal{E}_{i+K}(S)), n + N_0 + \max(\mathcal{E}_{n+K}(S)))]$. We show that all literals of index above M are pure. As the (Decrease) rule is applied only if no other rule applies, a decrease node cannot contain any pure literal. As an N-unfolded state is a decrease node by definition, there cannot be in $\mathcal{S}$ a node of index above M and we get the result with Proposition 5.

Consider $K > N_0$ and a literal of the form P_e (resp. $\neg P_e$) with $e = n - M_2 + K + \max(\mathcal{E}_{i+K}(S))$ or $e = n + K + \max(\mathcal{E}_{n+K}(S))$. We use Proposition 2 to show that P_e (resp. $\neg P_e$) must be pure. Consider a literal $\neg P_f$ (resp. P_f) occurring in $\mathcal{S}$. Depending on the form of f we show that the corresponding condition in Proposition 2 holds. First, e contains n so we only consider the three cases of the second item. If f matches the first or third case then the conditions are easy consequences of the definition of N_0. If f matches the second case, as the (Decrease) rule is applied only if no other rule applies, a decrease state (and *a fortiori* an N-unfolded state) cannot contain a literal and its negation: else the (Closure) rule would apply. So $\mathcal{S}$ cannot be a decrease state, contradiction. □

Lemma 2. *Consider a literal of index $n+K$ occurring in some N-unfolded state for some $K \in \mathbb{Z}$ and $N \geq 1$. Then* $\min[n - M_2 + \min(\mathcal{E}_{i+K}(S)), n + 1 + \min(\mathcal{E}_{n+K}(S))] \preceq n + K$.

Proof. (Sketch) As in Propositions 1 and 5, we easily obtain: $\min[n - M_2 + \min(\mathcal{E}_{i+K}(S)), n + N + \min(\mathcal{E}_{n+K}(S))] \preceq n + K$. $N \geq 1$ entails the result. □

Lemma 3. *There is a* finite *set containing all literals occurring in any N-unfolded state for any $N \in \mathbb{Z}$. $\mathcal{L}$ has $O(2^{\#S})$ elements.*

Proof. A literal is either negated or not, it contains a propositional symbol and an index so there are $2.n_{symb}.n_{idx}$ possible literals where n_{symb} (resp. n_{idx}) is the number of propositional symbols (resp. indices). There are finitely many possible propositional symbols in S and $n_{symb} < \#S$. There are two kinds of indices: those of the form $n + K$ for $K \in \mathbb{Z}$ and integers. Lemmata 1 and 2 in the first case and Proposition 4 in the second case enable to conclude.

$\min(\mathcal{E}_{i+K}(S))$, $\max(\mathcal{E}_{i+K}(S))$, $\min(\mathcal{E}_{\mathbb{Z}}(S))$, $\max(\mathcal{E}_{\mathbb{Z}}(S))$, $\min(\mathcal{E}_{n+K}(S))$ and $\max(\mathcal{E}_{n+K}(S))$ are lower than $2^{\#S}$ (numbers are encoded as bit strings and all those numbers occur in S). So the intervals of Lemmata 1 and 2 and Proposition 4 clearly have a size proportional to $2^{\#S}$, hence the exponential bound. □

Theorem 2. *(i) The set of states in $\mathcal{A}_S$ is finite, it has $O(2^{2^{\#S}})$ elements. (ii) Each state contains at most $O(2^{\#S})$ formulae.*

Proof. We first focus on decrease states which are also N-unfolded states for some $N \in \mathbb{N}$. As the (Decrease) rule is applied only if no other rule can apply such a state contains only iterations and literals. It is easily seen that every iteration occurring in an N-unfolded state must also occur in S. Thus there are finitely many possible iterations: at worst $\#S$. Furthermore by Lemma 3 there are $O(2^{\#S})$ possible literals. So the set of all possible iterations and literals has size $O(2^{\#S})$. As a decrease state is a subset of this set, there are $O(2^{2^{\#S}})$ of them. This gives (i) and (ii) for decrease states.

Consider a sequence of propositional states between two decrease states. Such a sequence is finite and have a length which is linear w.r.t. the size of the first decrease state: indeed all transitions just decompose the formulae of the first decrease state. As we just saw, the size of a decrease state is at worst $O(2^{\#S})$, so the intermediate states do not change the overall number of states. The same result holds between the initial state and a 1-unfolded state, and between a decrease or initial state and a final state or a state without transition. This covers all the possible cases for a given state so the whole set of states is finite and has the announced size. □

Corollary 1. schAUT *terminates.*

4.2 Consequences

As emptiness of finite automata is decidable we get:

Proposition 6. *Unsatisfiability of regular schemata is decidable.*

Proposition 7. *Let $S \in \mathcal{R}_{M_1}^{n-M_2}$. There is $N \in \mathbb{Z}$ s.t. S is unsatisfiable iff for all $K \in [M_1+M_2-1..N]$, $S\,[K/n]\!\downarrow$ is unsatisfiable. Furthermore $N = O(2^{2^{\#S}})$.*

Proof. The first implication is a trivial consequence of Definition 2. Then there are finitely many runs to go through to check that $\mathcal{A}_S$ is empty i.e. that S is unsatisfiable. To each of those runs corresponds a word, i.e. a number. Take N to be the maximum of those numbers. It is easily seen that this number is at most the number of decrease nodes, that we have already seen to be $O(2^{2^{\#S}})$. □

Finding a way to compute N would give another decision procedure: we compute the set of instances "lower than" N and feed a SAT-Solver with each of these formulae. However this would probably be highly inefficient.

The bound on the number of states gives the basis for a complexity study. Notice that first of all we have the easy result that the satisfiability problem for regular schemata is NP-hard. Indeed propositional logic is trivially embedded into schemata, hence the satisfiability problem contains SAT. But with SchAUT we can have a more precise result:

Theorem 3. *Consider $S \in \mathcal{R}_{M_1}^{n-M_2}$.* SchAUT *terminates on S in $O(2^{2^{\#S}})$ steps.*

Proof. (Sketch) Theorem 2 (ii) provides the most important fact: the number of states of $\mathcal{A}_S$ is double exponential. Then we analyse the complexity of the rewriting itself. Proposition 2 plays an essential role. □

Corollary 2. *Unsatisfiability of regular schemata is in 2-EXPTIME.*

Theorem 4. *The bound on* SchAUT *is tight, i.e. there is a class of regular schemata $(S_K)_{K\in\mathbb{N}}$ s.t.* SchAUT *terminates in time $O(2^{2^{\#S_K}})$.*

Proof. Consider $K \in \mathbb{N}$, we construct a schema S_K s.t. $\mathcal{A}_{S_K}$ contains necessarily $2^{2^{\#S_K}}$ states, hence the result. More precisely we consider all numbers encoded on K bits $(a_0, a_1, \ldots, a_{2^K-1})$ and we manage to have one state per number. This encoding requires K to occur in S_K, so $K \leq 2^{\#S}$, hence the double exponential.

We represent each of $a_0, a_1, \ldots$ by K indexed propositions representing their bits: a_0 by $A_1, \ldots, A_K$, a_1 by $A_{K+1}, \ldots, A_{K+K}$, etc., a_I by $A_{I.K+1}, \ldots, A_{I.K+K}$, etc. F_i indicates if i is the Final bit of one of those numbers: $Final \overset{\text{def}}{=} F_0 \wedge \bigwedge_{i=0}^{n} F_i \Rightarrow F_{i+K}$. For all $I \leq 2^K$, we impose $a_{I+1} = a_I + 1$. A carry C_i is used for this: $Incr \overset{\text{def}}{=} \bigwedge_{i=0}^{n}(F_i \Rightarrow C_{i+1}) \wedge (A_{i+K} \Leftrightarrow C_i \oplus A_i) \wedge (\neg F_{i+1} \Rightarrow (C_{i+1} \Leftrightarrow (C_i \wedge A_i))$. Finally we impose $a_0 = 0$. This is done by B_i (which Begins the sequence): $Begin \overset{\text{def}}{=} B_1 \wedge \bigwedge_{i=1}^{n}(B_i \wedge \neg F_i) \Rightarrow (B_{i+1} \wedge \neg A_i)$. We set $S_K \overset{\text{def}}{=} Begin \wedge Incr \wedge Final$. Clearly, $\mathcal{A}_{S_K}$ must have $2^{2^{\#S_K}}$ states. □

Finally notice that if we keep $K_1, K_2 \in \mathbb{Z}$ constant and consider only schemata S s.t. $[\min(\mathcal{E}_{i+K}(S)); \max(\mathcal{E}_{i+K}(S))] \subseteq [K_1; K_2]$, then the complexity of SchAUT is simply exponential. This is interesting because one can increase the size of a schema without necessarily increasing the size of $[\min(\mathcal{E}_{i+K}(S)); \max(\mathcal{E}_{i+K}(S))]$.

5 Conclusion

We presented a new decision procedure for regular propositional iterated schemata called SchAUT. SchAUT has several pros w.r.t. procedures given in [2,1]: it is more readable, more convenient technically and more efficient. We proved that time and space complexity of SchAUT is double exponential and consequently that the satisfiability problem for regular schemata is in 2-EXPTIME. We presented an example showing that the bound for SchAUT is tight.

We are not aware of similar approaches in automated deduction, but related works have been carried on in pure logic. Indeed from a logical point of view iterations are just a particular case of fixed points e.g. the schema $\bigwedge_{i=1}^{n} P_i$ might be represented as $\mu X(i)[i \geq 1 \Rightarrow (P(i) \wedge X(i-1))](n)$. But the propositional (modal) μ-calculus (see e.g. [4]) is not powerful enough to embed regular schemata (the satisfiability problem for propositional μ-calculus is in PSPACE, which seems highly improbable to be the case of regular schemata). There are other fixpoint logics such as LFP [13], however they generally do not admit any complete proof procedure and their purposes are essentially theoretical (LFP is mainly studied in finite model theory). The only study we know of a complete subclass is in [3] and regular schemata cannot be reduced to it.

A key point of SchAUT is the ability to deal with cycles in proofs. This is natural in an automata approach but proof procedures rarely incorporate such features. Slightly similar ideas exist in automated deduction e.g. in some tableaux methods for modal logics [10], or in μ-calculi [7]. However our work is much closer to that on *cyclic proofs*, which are studied in proof theory in the context of proofs by induction [5,15], one of their advantages being to avoid the explicit use of an induction principle.

Schemata can also be specified in first-order logic, axiomatizing (first-order) arithmetic. Iterations can be translated into bounded quantifications, e.g. $\bigvee_{i=1}^{n} S$ becomes $\forall n \exists i (1 \leq i \leq n \wedge S)$. Then we would use inductive theorem provers (e.g. [6,8]). However there are very few decidability results with such provers and most systems are designed to prove formulae of the form $\forall \boldsymbol{x} \phi$ where ϕ is *quantifier-free*. (Translated) schemata do not fall in this class as iterated disjunctions are translated into existential quantifications.

We implemented a solver for regular schemata, called RegSTAB, and available at `http://regstab.forge.ocamlcore.org`. Examples are provided with RegSTAB that show how regular schemata can express non-trivial problems: we can specify the commutativity or the associativity of *Adder*, or the inclusion between two automata (the parameter being the length of the run). Many electronic circuits shall be expressible by regular schemata whose parameter is the number of bits. Actually the main limitation of regular schemata is the impossibility to handle *nested iterations*. Allowing such schemata is the aim of the calculus presented in [1].

Notice that all iterations occurring in a regular schema have the same bounds. This is for technical convenience only: iterations that do not have the same bounds can be unfolded to ensure that this property holds, e.g. $(\bigvee_{i=1}^{n-1} P_i) \wedge (\bigvee_{i=2}^{n} Q_i)$ can be written $(P_1 \vee \bigvee_{i=2}^{n-1} P_i) \wedge (\bigvee_{i=2}^{n-1} Q_i \vee Q_n)$. We also forbid arithmetic expressions of the form $n-i$ or $2n+i$. Again, in many cases we can get rid of such expressions e.g. a schema $\bigvee_{i=1}^{n} P_i \Leftrightarrow P_{2n-i}$ can be written $\bigvee_{i=1}^{n} P_i \Leftrightarrow Q_i$, where Q_i denotes the variable P_{2n-i}. So non regular schemata can sometimes be transformed into equivalent regular ones. We are currently investigating this idea in order to get a syntactic characterisation of the class of schemata that can be reduced to regular ones. This work would significantly increase the scope of our decision procedure.

Finally, extending schemata to first order logic and working more precisely on the notion of *proof* schema might have practical applications in the formalisation of mathematical proofs, see e.g. [11] where proof schemata are used in the cut elimination system CERES.

References

1. Aravantinos, V., Caferra, R., Peltier, N.: A DPLL Proof Procedure For Propositional Iterated Schemata. In: Proceedings of the 21st European Summer School in Logic, Language and Information (Worskhop Structures and Deduction) (2009)
2. Aravantinos, V., Caferra, R., Peltier, N.: A Schemata Calculus For Propositional Logic. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 32–46. Springer, Heidelberg (2009)
3. Baelde, D.: On the Proof Theory of Regular Fixed Points. In: Giese, M., Waaler, A. (eds.) TABLEAUX 2009. LNCS, vol. 5607, pp. 93–107. Springer, Heidelberg (2009)
4. Bradfield, J., Stirling, C.: Modal Mu-Calculi. In: Blackburn, P., van Benthem, J., Wolter, F. (eds.) Handbook of Modal Logic, vol. 3, pp. 721–756. Elsevier Science Inc., New York (2007)
5. Brotherston, J.: Cyclic Proofs for First-Order Logic with Inductive Definitions. In: Beckert, B. (ed.) TABLEAUX 2005. LNCS (LNAI), vol. 3702, pp. 78–92. Springer, Heidelberg (2005)
6. Bundy, A.: The Automation of Proof by Mathematical Induction. In: [14], pp. 845–911
7. Cleaveland, R.: Tableau-based Model Checking in the Propositional Mu-calculus. Acta Inf. 27(9), 725–747 (1990)
8. Comon, H.: Inductionless induction. In: [14], ch. 14
9. Fisher, M., Rabin, M.: Super Exponential Complexity of presburger's Arithmetic. SIAM-AMS Proceedings 7, 27–41 (1974)
10. Goré, R.: Tableau Methods for Modal and Temporal Logics. In: D'Agostino, M., Gabbay, D., Hähnle, R., Posegga, J. (eds.) Handbook of Tableau Methods, ch. 6, pp. 297–396. Kluwer Academic Publishers, Dordrecht (1999)
11. Hetzl, S., Leitsch, A., Weller, D., Paleo, B.W.: Proof Analysis with HLK, CERES and ProofTool: Current Status and Future Directions. In: Sutcliffe, G., Colton, S., Schulz, S. (eds.) Workshop on Empirically Successful Automated Reasoning for Mathematics (ESARM), July 2008, pp. 21–41 (2008)
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Pu. Co., Reading (1979)
13. Immerman, N.: Relational Queries Computable in Polynomial Time (Extended Abstract). In: STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing, pp. 147–152. ACM, New York (1982)
14. Robinson, J.A., Voronkov, A. (eds.): Handbook of Automated Reasoning, vol. 2. Elsevier/MIT Press (2001)
15. Sprenger, C., Dam, M.: On the Structure of Inductive Reasoning: Circular and Tree-shaped Proofs in the mu-Calculus. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 425–440. Springer, Heidelberg (2003)

A Simple n-Dimensional Intrinsically Universal Quantum Cellular Automaton

Pablo Arrighi and Jonathan Grattage

[1] University of Grenoble, LIG, 220 rue de la Chimie, 38400 SMH, France
[2] ENS-Lyon, LIP, 46 allée d'Italie, 69364 Lyon cedex 07, France

Abstract. We describe a simple n-dimensional quantum cellular automaton (QCA) capable of simulating all others, in that the initial configuration and the forward evolution of any n-dimensional QCA can be encoded within the initial configuration of the intrinsically universal QCA. Several steps of the intrinsically universal QCA then correspond to one step of the simulated QCA. The simulation preserves the topology in the sense that each cell of the simulated QCA is encoded as a group of adjacent cells in the universal QCA.

1 Introduction

Cellular automata (CA), first introduced by Von Neumann [38], consist of an array of identical cells, each of which may take one of a finite number of possible states. The whole array evolves in discrete time steps by iterating a function G. This global evolution G is shift-invariant (it acts everywhere the same) and local (information cannot be transmitted faster than some fixed number of cells per time step). Because this is a physics-like model of computation [19], Feynman [17], and later Margolus [20], suggested that quantising this model was important, for two reasons: firstly, because in CA computation occurs without extraneous (unnecessary) control, hence eliminating a source of decoherence; and secondly because they are a good framework in which to study the quantum simulation of a quantum system. From a computation perspective there are other reasons to study QCA, such as studying space-sensitive problems in computer science, *e.g.* 'machine self-reproduction' [38] or 'Firing Squad Synchronisation', which QCA allow in the quantum setting. There is also a theoretical physics perspective, where CA are used as toy models of quantum space-time [18]. The first approach to defining QCA [2,16,39] was later superseded by a more axiomatic approach [8,9,32] together with the more operational approaches [39,11,30,25,31,36].

The most well known CA is Conway's 'Game of Life', a two-dimensional CA which has been shown to be universal for computation, in the sense that any Turing Machine (TM) can be encoded within its initial state and then executed by evolution of the CA. Because TM have long been regarded as the best definition of 'what an algorithm is' in classical computer science, this result could have been perceived as providing a conclusion to the topic of CA universality.

This was not the case, because CA do more than just running any algorithm. They run distributed algorithms in a distributed manner, model phenomena together with their spatial structure, and allow the use of the spatial parallelism inherent to the model. These features, modelled by CA and not by TM, are all interesting, and so the concept of universality must be revisited in this context to account for space. This is achieved by returning to the original meaning of the word *universality* [1,10,13], namely the ability for one instance of a computational model to be able to simulate other instances of the same computational model. Intrinsic universality formalises the ability of a CA to simulate another in a space-preserving manner [21,27,34], and was extended to the quantum setting in [3,5,6].

There are several related results in the CA literature. For example, refs. [19,23,24] provide computation universal Reversible Partitioned CA constructions, whereas ref. [22] deals with their ability to simulate any CA in the one-dimensional case. The problem of minimal intrinsically universal CA was addressed, *cf.* [28], and for Reversible CA (RCA) the issue was tackled by Durand-Löse [14,15]. The difficulty is in having an n-dimensional RCA simulate all other n-dimensional RCA and not, say, the $(n-1)$-dimensional RCA, otherwise a history-keeping dimension could be used, as by Toffoli [35]. There are also several other QCA related results. Watrous [40] has proved that QCA are universal in the sense of QTM. Shepherd, Franz and Werner [33] defined a class of QCA where the scattering unitary U_i changes at each step i (CCQCA). Universality in the circuit-sense has already been achieved by Van Dam [36], Cirac and Vollbrecht [37], Nagaj and Wocjan [25] and Raussendorf [31]. In the bounded-size configurations case, circuit universality coincides with intrinsic universality, as noted by Van Dam [36]. QCA intrinsic universality in the one-dimensional case is resolved in ref. [4]. Given the crucial role of this in classical CA theory, the issue of intrinsic universality in the n-dimensional case began to be addressed in refs. [5,6], where it was shown that a simple subclass of QCA, namely Partitioned QCA (PQCA), are intrinsically universal. Having shown that PQCA are intrinsically universal, it remains to be shown that there exists a n-dimensional PQCA capable of simulating all other n-dimensional PQCA for $n > 1$, which is presented here.

PQCA are QCA of a particular form, where incoming information is scattered by a fixed unitary U before being redistributed. Hence the problem of finding an intrinsically universal PQCA reduces to finding some universal scattering unitary U (this is made formal in section 2, see Fig.2). Clearly the universality requirement on U is much more difficult than just quantum circuit universality. This is because the simulation of a QCA H has to be done in a parallel, space-preserving manner. Moreover we must simulate not only an iteration of H but several (H^2, ...), so after every simulation the universal PQCA must be ready for a further iteration.

From a computer architecture point of view, this problem can be recast in terms of finding some fundamental quantum processing unit which is capable of simulating any other network of quantum processing units, in a space-preserving manner. From a theoretical physics perspective, this amounts to specifying a scattering phenomenon that is capable of simulating any other, again in a space-preserving manner.

2 An Intrinsically Universal QCA

The aim is to find a particular U-defined PQCA which is capable of intrinsically simulating any V-defined PQCA, for any V. In order to describe such a U-defined PQCA in detail, two things must be given: the dimensionality of the cells (including the meaning attached to each of the states they may take), and the way the scattering unitary U acts upon these cells. The necessary definitions for n-dimensional QCA are given in refs. [5,6].

Circuit Universality versus Intrinsic Universality in Higher Dimensions
As already discussed, intrinsic universality refers to the ability for one CA to simulate any other CA, whereas computation universality is about simulating a TM. Additionally, circuit universality is the ability of one CA to simulate any circuit. Informally, in a quantum setting, circuit universality is the ability of a PQCA to simulate any finitary combination of a universal set of quantum gates, such as the standard gate set: Cnot, R($\frac{\pi}{4}$) (also known as the $\frac{\pi}{8}$ gate), and the Hadamard gate.

In n-dimensions, it is often assumed in the classical CA literature that circuit universality implies intrinsic universality, and that both are equivalent to computation universality [27], without provision of an explicit construction. Strictly speaking this is not true. Consider a two-dimensional CA which runs one-dimensional CA in parallel. If the one-dimensional CA is circuit/computation universal, but not computation/intrinsically universal, then this is also true for the two-dimensional CA. Similarly, in the PQCA setting, the two-dimensional constructions in [30] and [31] are circuit universal but not intrinsically universal.

However, this remains a useful intuition: Indeed, CA admit a block representation, where these blocks are permutations for reversible CA, while for PQCA the blocks are unitary matrices. Thus the evolution of any (reversible/quantum) CA can be expressed as an infinite (reversible/quantum) circuit of (reversible/quantum) gates repeating across space. If a CA is circuit universal, and if it is possible to wire together different circuit components in different regions of space, then the CA can simulate the block representation of any CA, and hence can simulate any CA in a way which preserves its spatial structure. It is intrinsically universal.

Flattening a PQCA into Space
Any CA can be encoded into a 'wire and gates' arrangement following the above argument, but this has never been made explicit in the literature. This section makes more precise how to flatten any PQCA in space, so that it is simulated by a PQCA which implements quantum wires and universal quantum gates. Flattening a PQCA means that the infinitely repeating, two-layered circuit is arranged in space so that at the beginning all the signals carrying qubits find themselves in circuit-pieces which implement a scattering unitary of the first

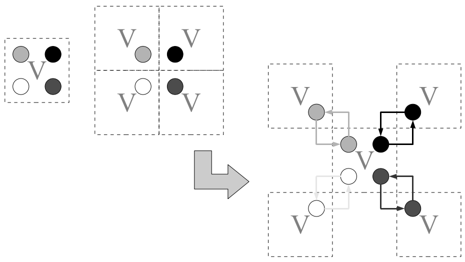


Fig. 1. Flattening a PQCA into a simulating PQCA. *Left*: Consider four cells (white, light grey, dark grey, black) of a PQCA having scattering unitary V. The first layer PQCA applies V to these four cells, then the second layer applies V at the four corners. *Right*: We need to flatten this so that the two-layers become non-overlapping. The first layer corresponds to the centre square, and the second layer to the four corner squares. At the beginning the signals (white, light grey, dark grey, black) coding for the simulated cells are in the centre square.

layer, and then all synchronously exit and travel to circuit-pieces implementing the scattering unitary of the second layer, etc. An algorithm for performing this flattening can be provided, however the process will not be described in detail, for clarity and following the classical literature, which largely ignores this process.

The flattening process can be expressed in three steps: Firstly, the V-defined PQCA is expanded in space by coding each cell into a hypercube of 2^n cells. This allows enough space for the scattering unitary V to be applied on non-overlapping hypercubes of cells, illustrated in the two-dimensional case in Fig. 1. Secondly, the hypercubes where V is applied must be connected with wires, as shown in Fig. 1 (*right*). Within these hypercubes wiring is required so that

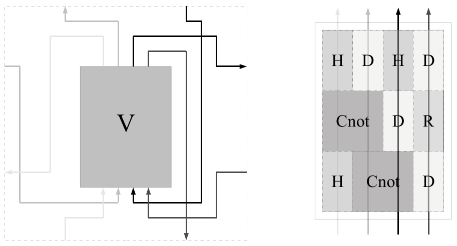


Fig. 2. Flattening a PQCA into a simulating PQCA (cont'd). *Left*: Within the central square the incoming signals are bunched together so as to undergo a circuit which implements V, and are then dispatched towards the four corners. This diagram does not make explicit a number of signal delays, which may be needed to ensure that they arrive synchronously at the beginning of the circuit implementing V. *Right*: Within the central rectangle, the circuit which implements V is itself a combination of smaller circuits for implementing a universal set of quantum gates such as Cnot, Hadamard and the $\mathrm{R}(\frac{\pi}{4})$, together with delays.

incoming signals are bunched together to undergo a circuit implementation of V, and are then dispatched appropriately, as shown in Fig. 2 (*left*). This requires both time and space expansions, with factors that depend non-trivially (but uninterestingly) upon the size of the circuit implementation of V and the way the wiring and gates work in the simulating PQCA. Next, an encoding of the circuit description of the scattering unitary V is implemented in the simulating PQCA upon these incoming bunched wires, as shown in Fig. 2 (*right*). This completes the description of the overall scheme according to which a PQCA that is capable of implementing wires and gates is also capable of intrinsically simulating any PQCA, and hence any QCA. A particular PQCA that supports these wires and gates can now be constructed.

Barriers and Signals Carrying Qubits
Classical CA studies often refer to 'signals' without an explicit definition. In this context, a signal refers to the state of a cell which may move to a neighbouring cell consistently, from one step to another, by the evolution of the CA. Therefore a signal would appear as a line in the space-time diagram of the CA. These lines need to be implemented as signal redirections. A 2D solution is presented here, but this scheme can easily be extended to higher dimensions. Each cell has four possible basis states: empty (ϵ), holding a qubit signal (0 or 1), or a barrier ($\blacksquare$). The scattering unitary U of the universal PQCA acts on 2×2 cell neighbourhoods.

Signals encode qubits which can travel diagonally across the 2D space (NE, SE, SW, or NW). Barriers do not move, while signals move in the obvious way if unobstructed, as there is only one choice for any signal in any square of four cells. Hence the basic movements of signals are given by the following four rules:

$$\left|\begin{array}{|c|c|}\hline & \\ \hline s & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline & s \\ \hline & \\ \hline\end{array}\right\rangle, \qquad \left|\begin{array}{|c|c|}\hline s & \\ \hline & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline & \\ \hline & s \\ \hline\end{array}\right\rangle,$$

$$\left|\begin{array}{|c|c|}\hline & s \\ \hline & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline & \\ \hline s & \\ \hline\end{array}\right\rangle, \qquad \left|\begin{array}{|c|c|}\hline & \\ \hline & s \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline s & \\ \hline & \\ \hline\end{array}\right\rangle.$$

where $s \in \{0, 1\}$ denotes a signal, and blank cells are empty.

The way to interpret the four above rules in terms of the scattering unitary U is just case-by-case definition, *i.e.* they show that $U\left|\begin{array}{|c|c|}\hline & \\ \hline s & \\ \hline\end{array}\right\rangle = \left|\begin{array}{|c|c|}\hline & s \\ \hline & \\ \hline\end{array}\right\rangle$. Moreover, each rule can be obtained as a rotation of another, hence by stating that the U-defined PQCA is isotropic the first rule above suffices. This convention will be used throughout.

The ability to redirect signals is achieved by 'bouncing' them off walls constructed from two barriers arranged either horizontally or vertically:

$$\left|\begin{array}{|c|c|}\hline \blacksquare & s \\ \hline \blacksquare & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline \blacksquare & \\ \hline \blacksquare & s \\ \hline\end{array}\right\rangle.$$

where s again denotes the signal and the shaded cells denote the barriers which causes the signal to change direction. If there is only one barrier present in the four cell square being operated on then the signal simply propagates as normal and is not deflected:

$$\left|\begin{array}{|c|c|}\hline \blacksquare & \\ \hline s & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline \blacksquare & s \\ \hline & \\ \hline\end{array}\right\rangle .$$

Using only these basic rules of signal propagation and signal reflection from barrier walls, signal delay (Fig. 3) and signal swapping (Fig. 4) tiles can be constructed. All of the rules presented so far are permutations of some of the base elements of the vector space generated by

$$\left\{\left|\begin{array}{|c|c|}\hline w & x \\ \hline y & z \\ \hline\end{array}\right\rangle\right\}_{w,x,y,z\in\{\epsilon,0,1,\blacksquare\}}$$

therefore U is indeed unitary on the subspace upon which its action has so far been described.

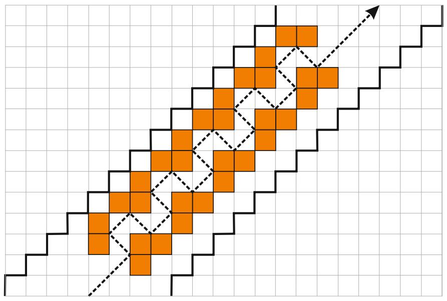


Fig. 3. The 'identity circuit' tile, an 8×14 tile taking 24 time-steps, made by repeatedly bouncing the signal from walls to slow its movement through the tile. The dotted line gives the signal trajectory, with the arrow showing the exit point and direction of signal propagation. The bold lines show the tile boundary.

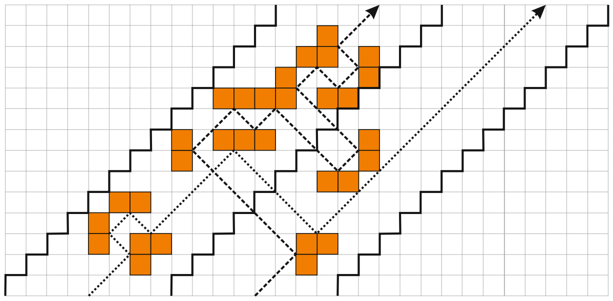


Fig. 4. The 'swap circuit' tile, a 16×14 tile, where both input signals are permuted and exit synchronously after 24 time-steps. As the first signal (*bottom left*) is initially delayed, there is no interaction.

Gates
To allow a universal set of gates to be implemented by the PQCA, certain combinations of signals and barriers can be assigned special importance. The Hadamard operation on a single qubit-carrying signal can be implemented by interpreting a signal passing through a diagonally oriented wall, analogous to a semitransparent barrier in physics. This has the action defined by the following rule:

$$\left|\begin{array}{|c|c|}\hline \blacksquare & \\ \hline 0 & \blacksquare \\ \hline\end{array}\right\rangle \mapsto \frac{1}{\sqrt{2}}\left|\begin{array}{|c|c|}\hline \blacksquare & 0 \\ \hline & \blacksquare \\ \hline\end{array}\right\rangle + \frac{1}{\sqrt{2}}\left|\begin{array}{|c|c|}\hline \blacksquare & 1 \\ \hline & \blacksquare \\ \hline\end{array}\right\rangle$$

$$\left|\begin{array}{|c|c|}\hline \blacksquare & \\ \hline 1 & \blacksquare \\ \hline\end{array}\right\rangle \mapsto \frac{1}{\sqrt{2}}\left|\begin{array}{|c|c|}\hline \blacksquare & 0 \\ \hline & \blacksquare \\ \hline\end{array}\right\rangle - \frac{1}{\sqrt{2}}\left|\begin{array}{|c|c|}\hline \blacksquare & 1 \\ \hline & \blacksquare \\ \hline\end{array}\right\rangle$$

This implements the Hadamard operation, creating a superposition of configurations with appropriate phases. Using this construction a Hadamard tile can be constructed (Fig. 5) by simply adding a semitransparent barrier to the end of the previously defined delay (identity) tile (Fig. 3). A way of encoding two

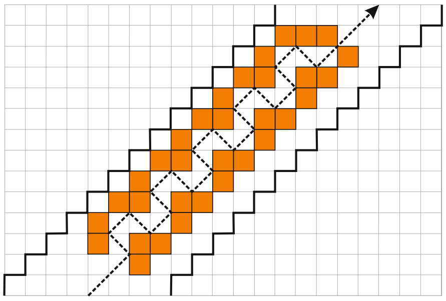


Fig. 5. The 'Hadamard gate' tile applies the Hadamard operation to the input signal. It is a modification of the identity circuit tile, with a diagonal (semitransparent) barrier added at the end which performs the Hadamard operation.

qubit gates in this system is to consider that two signals which cross paths interact with one another. The controlled-R$(\frac{\pi}{4})$ operation can be implemented by considering signals that cross each other as interacting only if they are both 1, in which case a global phase of $e^{\frac{i\pi}{4}}$ is applied. Otherwise the signals continue as normal. This behaviour is defined by the following rule:

$$\left|\begin{array}{|c|c|}\hline 1 & \\ \hline 1 & \\ \hline\end{array}\right\rangle \mapsto e^{\frac{i\pi}{4}}\left|\begin{array}{|c|c|}\hline & 1 \\ \hline & 1 \\ \hline\end{array}\right\rangle, \qquad \left|\begin{array}{|c|c|}\hline x & \\ \hline y & \\ \hline\end{array}\right\rangle \mapsto \left|\begin{array}{|c|c|}\hline & y \\ \hline & x \\ \hline\end{array}\right\rangle \textit{ otherwise}$$

where $x, y \in \{0, 1\}$. This signal interaction which induces a global phase change allows the definition of both a two signal controlled-R$(\frac{\pi}{4})$ tile (Fig. 6) and a single signal R$(\frac{\pi}{4})$ operation tile (Fig. 7). These rules are simply a permutation and phase change of base elements of the form:

$$\left\{\left|\begin{array}{|c|c|}\hline x & \\ \hline y & \\ \hline\end{array}\right\rangle\right\}_{x,y\in\{0,1\}}$$

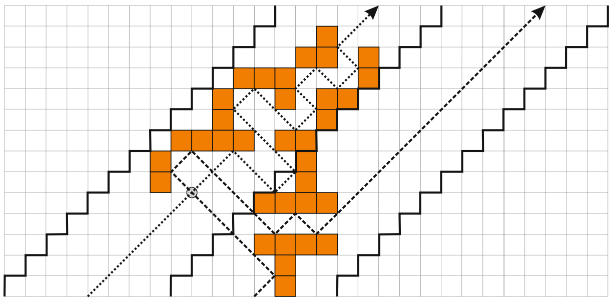


Fig. 6. The 'controlled-R($\frac{\pi}{4}$) gate' tile, with a signal interaction at the highlighted cell

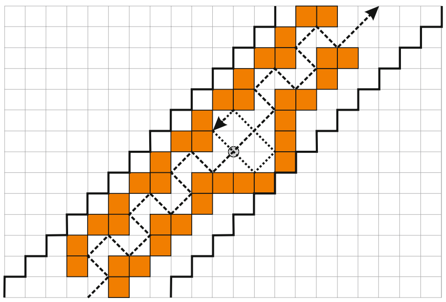


Fig. 7. The 'R($\frac{\pi}{4}$) gate' tile. This tile makes use of a signal, set to $|1\rangle$, which loops inside the grid every six time-steps, ensuring that it will interact with the signal that enters the tile, and causing it to act as the control qubit to a controlled-R($\frac{\pi}{4}$) operation. It therefore acts as a phase rotation on the input qubit, which passes directly through.

(and their rotations), therefore U is a unitary operation on the subspace upon which its action has so far been described. Wherever U has not yet been defined, it is the identity. Hence U is unitary.

Circuits: Combining Gates
A signal is given an 8×14 tile (16×14 for two signal operations) in which the action is encoded. The signals enter each tile at the fifth cell from the left, and propagate diagonally NE. Each time step finds the tile shifted one cell to the right to match this diagonal movement, giving a diagonal tile. The signal exits the tile 14 cells North and East of where it entered. This allows these tiles to be composed in parallel and sequentially with the only other requirement being that the signal exits at the appropriate point, *i.e.* the fifth cell along the tile, after 24 time-steps. This ensures that all signals are synchronised as in Fig. 2 (*right*), allowing larger circuits to be built from these elementary tiles by simply plugging them together. Non-contiguous gates can also be wired together using appropriate wall constructions to redirect and delay signals so that they are correctly synchronised.

The implemented set of quantum gates, the identity, Hadamard, swap, $\mathrm{R}(\frac{\pi}{4})$ and controlled-$\mathrm{R}(\frac{\pi}{4})$, gives a universal set. Indeed the standard set of cNot, H, $\mathrm{R}(\frac{\pi}{4})$ can be recovered as follows:

$$\textsc{cNot}\,|\psi\rangle = (\mathbb{I}\otimes H)(\mathrm{cR}(\pi/4))^4(\mathbb{I}\otimes H)\,|\psi\rangle$$

where $\mathrm{cR}(\frac{\pi}{4})^4$ denotes four applications of the controlled-$\mathrm{R}(\frac{\pi}{4})$ gate, giving the controlled-Phase operation.

3 Conclusion

This paper presents a simple PQCA which is capable of simulating all other PQCA, preserving the topology of the simulated PQCA. This means that the initial configuration and the forward evolution of any PQCA can be encoded within the initial configuration of this PQCA, with each simulated cell encoded as a group of adjacent cells in the PQCA, *i.e.* intrinsic simulation. The construction in section 2 is given in two-dimensions, which can be seen to generalise to $n > 1$-dimensions. The main, formal result of this work can therefore be stated as:

Claim 1. *There exists an n-dimensional U-defined PQCA, G, which is an intrinsically universal PQCA. Let H be a n-dimensional V-defined PQCA such that V can be expressed as a quantum circuit C made of gates from the set* Hadamard, Cnot, *and* $\mathrm{R}(\frac{\pi}{4})$. *Then G is able to intrinsically simulate H.*

Any finite-dimensional unitary V can always be approximated by a circuit $C(V)$ with an arbitrary small error $\varepsilon = \max_{|\psi\rangle} ||V\,|\psi\rangle - C\,|\psi\rangle\,||$. Assuming instead that G simulates the $C(V)$-defined PQCA, for a region of s cells over a period t, the error with respect to the V-defined PQCA will be bounded by $st\varepsilon$. This is due to the general statement that errors in quantum circuits increase, at most, proportionally with time and space [26]. Combined with the fact that PQCA are universal [5,6], this means that G is intrinsically universal, up to this unavoidable approximation.

Discussion and Future Work
QC research has so far focused on applications for more secure and efficient computing, with theoretical physics supporting this work in theoretical computer science. The results of this interdisciplinary exchange led to the assumptions underlying computer science being revisited, with information theory and complexity theory, for example, being reconsidered and redeveloped. However, information theory also plays a crucial role in the foundations of theoretical physics (*e.g.* deepening our understanding of entanglement [12] and decoherence [29]). These developments are also of interest in theoretical physics studies where physical aspects such as particles and matter are considered; computer science studies tend to consider only abstract mathematical quantities. Universality, among the many computer science concepts, is a simplifying methodology in this respect. For example, if the problem being studied crucially involves some

idea of interaction, universality makes it possible cast it in terms of information exchanges *together* with some universal information processing. This paper presents an attempt to export universality as a tool for application in theoretical physics, a small step towards the goal of finding and understanding a *universal physical phenomenon*, within some simplified mechanics. Similar to the importance of the idea of the spatial arrangement of interactions in physics, intrinsic universality has broader applicability than computation universality and must be preferred. In short, if only one physical phenomenon is considered, it should be an intrinsically universal physical phenomenon, as it could be used to simulate all others.

The PQCA cell dimension of the simple intrinsically universal construction given here is four (empty, a qubit ($|0\rangle$ or $|1\rangle$), or a barrier). In comparison, the simplest classical Partitioned CA has cell dimension two [20]. Hence, although the intrinsically universal PQCA presented here is the simplest found, it is not minimal. In fact, one can also manage [7] an intrinsically universal PQCA with a cell dimension of three, in two different ways. One way is to encode the spin degree of freedom (0 and 1) into a spatial degree of freedom, so that now the semitransparent barrier either splits or combines signals. The second way is to code barriers as pairs of signals as in the Billiard Ball CA model [20]. These constructions may be minimal, but are not as elegant as the one presented here. In future work we will show that there is a simple, greater than two-dimensional PQCA which is minimal, as it has a cell dimension of two.

Acknowledgements

The authors would like to thank Jérôme Durand-Löse, Jarkko Kari, Jacques Mazoyer, Kenichi Morita, Nicolas Ollinger, Guillaume Theyssier and Philippe Jorrand.

References

1. Albert, J., Culik, K.: A simple universal cellular automaton and its one-way and totalistic version. Complex Systems 1, 1–16 (1987)
2. Arrighi, P.: Algebraic characterizations of unitary linear quantum cellular automata. In: Královič, R., Urzyczyn, P. (eds.) MFCS 2006. LNCS, vol. 4162, pp. 122–133. Springer, Heidelberg (2006)
3. Arrighi, P., Fargetton, R.: Intrinsically universal one-dimensional quantum cellular automata. In: Proceedings of the Development of Computational Models workshop, DCM '07 (2007)
4. Arrighi, P., Fargetton, R., Wang, Z.: Intrinsically universal one-dimensional quantum cellular automata in two flavours. Fundamenta Informaticae 21, 1001–1035 (2009)
5. Arrighi, P., Grattage, J.: Intrinsically universal n-dimensional quantum cellular automata. Extended version of this paper. ArXiv preprint: arXiv:0907.3827 (2009)
6. Arrighi, P., Grattage, J.: Partitioned quantum cellular automata are intrinsically universal (2009) (submitted)

7. Arrighi, P., Grattage, J.: Two minimal n-dimensional intrinsically universal quantum cellular automata (2009) (manuscript)
8. Arrighi, P., Nesme, V., Werner, R.: Unitarity plus causality implies localizability. Quantum Information Processing (QIP) 2010, ArXiv preprint: arXiv:0711.3975 (2007)
9. Arrighi, P., Nesme, V., Werner, R.F.: Quantum cellular automata over finite, unbounded configurations. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 64–75. Springer, Heidelberg (2008)
10. Banks, E.R.: Universality in cellular automata. In: Proceedings of the 11th Annual Symposium on Switching and Automata Theory (SWAT '70), Washington, DC, USA, pp. 194–215. IEEE Computer Society, Los Alamitos (1970)
11. Brennen, G.K., Williams, J.E.: Entanglement dynamics in one-dimensional quantum cellular automata. Phys. Rev. A 68(4), 042311 (2003)
12. Dür, W., Vidal, G., Cirac, J.I.: Three qubits can be entangled in two inequivalent ways. Phys. Rev. A 62, 062314 (2000)
13. Durand, B., Roka, Z.: The Game of Life: universality revisited, Research Report 98-01. Technical report, Ecole Normale Suprieure de Lyon (1998)
14. Durand-Lose, J.O.: Reversible cellular automaton able to simulate any other reversible one using partitioning automata. In: Baeza-Yates, R., Poblete, P.V., Goles, E. (eds.) LATIN 1995. LNCS, vol. 911, pp. 230–244. Springer, Heidelberg (1995)
15. Durand-Lose, J.O.: Intrinsic universality of a 1-dimensional reversible cellular automaton. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, p. 439. Springer, Heidelberg (1997)
16. Durr, C., Le Thanh, H., Santha, M.: A decision procedure for well-formed linear quantum cellular automata. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 281–292. Springer, Heidelberg (1996)
17. Feynman, R.P.: Quantum mechanical computers. Foundations of Physics (Historical Archive) 16(6), 507–531 (1986)
18. Lloyd, S.: A theory of quantum gravity based on quantum computation. ArXiv preprint: quant-ph/0501135 (2005)
19. Margolus, N.: Physics-like models of computation. Physica D: Nonlinear Phenomena 10(1-2) (1984)
20. Margolus, N.: Parallel quantum computation. In: Complexity, Entropy, and the Physics of Information: The Proceedings of the 1988 Workshop on Complexity, Entropy, and the Physics of Information, Santa Fe, New Mexico, Perseus Books, May-June 1989, p. 273 (1990)
21. Mazoyer, J., Rapaport, I.: Inducing an order on cellular automata by a grouping operation. In: Meinel, C., Morvan, M. (eds.) STACS 1998. LNCS, vol. 1373, pp. 116–127. Springer, Heidelberg (1998)
22. Morita, K.: Reversible simulation of one-dimensional irreversible cellular automata. Theoretical Computer Science 148(1), 157–163 (1995)
23. Morita, K., Harao, M.: Computation universality of one-dimensional reversible (injective) cellular automata. IEICE Trans. Inf. & Syst., E 72, 758–762 (1989)
24. Morita, K., Ueno, S.: Computation-universal models of two-dimensional 16-state reversible cellular automata. IEICE Trans. Inf. & Syst., E 75, 141–147 (1992)
25. Nagaj, D., Wocjan, P.: Hamiltonian Quantum Cellular Automata in 1D. ArXiv preprint: arXiv:0802.0886 (2008)
26. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge (October 2000)

27. Ollinger, N.: Universalities in cellular automata a (short) survey. In: Durand, B. (ed.) Proceedings of First Symposium on Cellular Automata Journées Automates Cellulaires (JAC 2008), Uzès, France, April 21-25, pp. 102–118. MCCME Publishing House, Moscow (2008)
28. Ollinger, N., Richard, G.: A Particular Universal Cellular Automaton. In: Neary, T., Woods, D., Seda, A.K., Murphy, N. (eds.) CSP, pp. 267–278. Cork University Press (2008)
29. Paz, J.P., Zurek, W.H.: Environment-induced decoherence and the transition from quantum to classical. Lecture Notes in Physics, pp. 77–140 (2002)
30. Pérez-Delgado, C., Cheung, D.: Local unitary quantum cellular automata. Physical Review A 76(3), 32320 (2007)
31. Raussendorf, R.: Quantum cellular automaton for universal quantum computation. Phys. Rev. A 72(022301) (2005)
32. Schumacher, B., Werner, R.: Reversible quantum cellular automata. ArXiv preprint quant-ph/0405174 (2004)
33. Shepherd, D.J., Franz, T., Werner, R.F.: A universally programmable quantum cellular automata. Phys. Rev. Lett. 97(020502) (2006)
34. Theyssier, G.: Captive cellular automata. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 427–438. Springer, Heidelberg (2004)
35. Toffoli, T.: Computation and construction universality of reversible cellular automata. J. of Computer and System Sciences 15(2) (1977)
36. Van Dam, W.: Quantum cellular automata. Masters thesis, University of Nijmegen, The Netherlands (1996)
37. Vollbrecht, K.G.H., Cirac, J.I.: Reversible universal quantum computation within translation-invariant systems. New J. Phys. Rev. A 73, 012324 (2004)
38. von Neumann, J.: Theory of Self-Reproducing Automata. University of Illinois Press, Champaign (1966)
39. Watrous, J.: On one-dimensional quantum cellular automata. Complex Systems 5(1), 19–30 (1991)
40. Watrous, J.: On one-dimensional quantum cellular automata. In: Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, Washington, DC, USA, pp. 528–537. IEEE Computer Society, Los Alamitos (1995)

A Fast Longest Common Subsequence Algorithm for Similar Strings

Abdullah N. Arslan

Department of Computer Science and Information Systems,
Texas A & M University - Commerce,
TX 75428, USA
Abdullah_Arslan@tamu-commerce.edu

Abstract. The longest common subsequence problem is a very important computational problem for which there are many algorithms. We present a new algorithm for this problem. Let X and Y be any two given strings each of length $O(n)$. We observe that a longest common subsequence can be obtained by using longest common prefixes of suffixes (longest common extensions) of X and Y. The longest common extension problem asks for the longest common prefix of suffixes starting in a given pair of positions in X and Y, respectively. Let e be the number of edit operations, insert, delete, and substitute to change X to Y (i.e. let e be the edit distance between X and Y). Our algorithm visits $O(\min\{en, (1+\sqrt{2})^{2e+1})$ nodes in the edit graph, and for every visited node, performs one longest common extension query. Each of these queries can be answered in constant time if we represent the strings by a suffix tree or a suffix array. These data structures can be created in linear time. We do not assume that the edit distance e is known beforehand, therefore we try values for e starting with $e = 1$ (without loss of generality $X \neq Y$) and double e until our algorithm finds a longest common subsequence. The total time complexity of our algorithm is $O(\min\{en \log n, n + e(1+\sqrt{2})^{2e+1}\})$. This is a better time complexity result compared to those of existing solutions for the problem when e is small. For example, when $e \leq \frac{1}{3}((\log_{(1+\sqrt{2})}\ n) - 1)$ our algorithm finds an optimal solution in time $O(n)$.

Keywords: algorithm, string, edit distance, longest common subsequence, suffix tree, lowest common ancestor, suffix array, longest common extension, dynamic programming.

1 Introduction

Given two strings X and Y, a *longest common subsequence* (*lcs*) is a longest string which is a subsequence of both X and Y. Let both X and Y be of length $O(n)$. The problem of finding a longest common subsequence for given strings (*the lcs problem*) has been studied extensively. There are many algorithms for this problem in the literature. Bergroth et al. [3] give a survey of *lcs* algorithms. The *lcs* problem can be viewed as a special case of the *edit distance* problem, and therefore, it can be solved using the classical edit-distance dynamic programming algorithm presented by Wagner and Fisher [13], which runs in time $O(n^2)$. The theoretically fastest *lcs* algorithm to date is the algorithm of Masek and Paterson [9]. This algorithm runs in time $O(n^2/\log n)$. There are many algorithms that are fast for certain special cases of the problem. For example, the time complexity of the algorithm of Kuo and Cross [8] is $O(|M| + n(r + \log n)$, where $|M|$ denotes the number of all matches between symbols of X and Y, and r denotes the number of matches in an *lcs* of X and Y. This algorithm is fast when strings are very dissimilar. There are also algorithms that are specialized for small alphabets. For example, the time complexity of the algorithm of Apostolico and Guerra [1] is $O(rn + \sigma n + n \log \sigma)$, where σ is the alphabet size. There are other algorithms that are fast for certain other parameters. In this paper, we consider the case when the strings are similar. Motivating example applications for this case include comparison of large texts with minor changes (e.g. similar program source codes). There are several algorithms that are relevant for this case. Each of the algorithms of Nakatsu et al. [11], Miller and Myers [10] and Wu et al. [14] runs in time $\Theta(n(n-r))$ when X and Y are both strings of length $O(n)$. In this paper, we present an algorithm with which we achieve a provably faster time complexity when the *simple edit distance* (the minimum total number of insertions, deletions and substitutions necessary to change one string into the other) e between X and Y is small. The time complexity of our algorithm is $O(\min\{en \log n, n + e(1+\sqrt{2})^{2e+1}\})$. Other existing algorithms in this case of the problem require $\Omega(ne)$ time because $n - r \geq e$. The time complexity of our algorithm is superior when $(1+\sqrt{2})^{2e+1}$ is $o(n)$. For example, when $e \leq \frac{1}{3}((\log_{(1+\sqrt{2})} n) - 1)$ our algorithm runs in time $O(n)$ while other algorithms require $\Omega(n \log n)$ time.

Our algorithm uses the fact that *lcs* length can be computed by modifying the dynamic programming computation of the edit distance. It only examines the nodes in a narrow diagonal band in the edit graph. For every node it examines, it performs one *longest common extension query.* We show that whenever a match is included in an *lcs*, there exists an optimal *lcs* that includes the entire longest common extension that starts with this node, therefore, our search for an optimal *lcs* can directly jump to the end of block of matches ignoring the intermediate matches. This gives rise to a smaller search space for the nodes on optimal *edit paths.* Answering each longest common query takes constant time if we use a suffix tree or a suffix array to represent the strings. The processing time for creating either of these data structures is $O(n)$. In our algorithm, we maintain ordered queues (heaps) for the nodes we examine. We show that the total time (after we create the suffix tree or suffix array) for computing an optimal *lcs* is $O(\min\{en \log n, e(1+\sqrt{2})^{e+1}\})$.

The outline of this paper is as follows: In Section 2, we give basic definitions for edit distance, and the longest common extension problem. In Section 3, we present an *lcs* algorithm that takes a parameter which is assumed to be the edit distance between the input strings. In Section 4, we give our main *lcs* algorithm that uses the parametric algorithm as a subroutine. We conclude in Section 5.

2 Preliminaries

Let Σ be an alphabet with cardinality at least two. For any given string S over alphabet Σ, we denote by S_i the ith symbol of string $S = S_1 S_2 \dots S_{|S|}$, where $|S|$ is the length of string S. Similarly, let $S_{i..j}$ denote the substring $S_i S_{i+1} \dots S_j$ of S. Given two strings X and Y each of length $O(n)$ over alphabet Σ, we use the *edit graph* $G_{X,Y}$ to analyze all possible edit-transformations of X into Y. The edit graph is a directed acyclic graph having $(|X|+1)(|Y|+1)$ lattice points (u,v) as vertices for $0 \le u \le |X|$, and $0 \le v \le |Y|$ (see [5]). An *edit path* for strings X and Y is a directed path from vertex $(0,0)$ to $(|X|,|Y|)$ in $G_{X,Y}$. To each vertex there is an incoming arc from each neighbor if it exists. Horizontal and vertical arcs correspond to insert and delete operations on X, respectively. The diagonal arcs correspond to substitutions which are either matching (if the corresponding symbols are the same), or mismatching (otherwise).

Let $D_{i,j}$ denote the edit distance between $X_{1..i}$ and $Y_{1..j}$. $D_{i,j}$ can be calculated by the following dynamic programming formulation [13,5]: For all i, j, $1 \le i \le |X|$, $1 \le j \le |Y|$,

$$D_{i,j} = \min\{D_{i-1,j} + \mu,\ D_{i,j-1} + \mu,\ D_{i-1,j-1} + \alpha(X_i, Y_j)\} \tag{1}$$

where μ is the cost of an insert or a delete, and $\alpha(X_i, Y_j) = \beta$ if $X_i \ne Y_j$ (β is the substitution cost); 0 (no cost for a match), otherwise. At the boundary $D_{i,0} = i\mu$ for all $i,\ 0 \le i \le |X|$, and $D_{0,j} = j\mu$ for all $j,\ 0 \le j \le |Y|$. The edit distance between X and Y is $D_{|X|,|Y|}$, and we denote it by $e_{\mu,\beta}$. The simple edit distance $e_{1,1}$ is the total number of edit operations (insert, delete and substitute; number of matches is not included) necessary to transform X into Y.

In this paper, we consider edit distance computations with $\mu = 1$ and $\beta = 2$, i.e. in this case, the cost of an insertion or a deletion is 1, the cost of a substitution is 2, and the cost is zero for a match. Consider $D_{i,j}$ computed using Equation (1) using these costs. Let $L_{i,j}$ be the length of *lcs* of $X_{1..i}$ and $Y_{1..j}$. $L_{i,j}$ can be computed using a dynamic programming formulation similar to Equation (1). We can also obtain it from $D_{i,j}$ using the following equation:

$$L_{i,j} = (i + j - D_{i,j})/2 \tag{2}$$

This is because on any optimal edit path for $X_{1..i}$ and $Y_{1..j}$, any substitution can be replaced by a pair of an insertion and a deletion without changing the optimality when $\mu = 1$ and $\beta = 2$. On the resulting substitution-free optimal edit path, the total number of insertions and matches is i, and the total number of deletions and matches is j. An optimal edit path ending at (i,j) has the maximum number of matches on any path from $(0,0)$ to (i,j), therefore, we call this optimal edit path also an *lcs path* between $X_{1..i}$ and $Y_{1..j}$.

For given X and Y, we will compute $e_{1,2}$ to obtain the *lcs* length using Equation 2. We use the terms optimal edit path and *lcs* path interchangeably because every optimal *lcs* path is also optimal for the computation of the edit distance $e_{1,2}$ and vice versa.

Let suf_i denote the suffix $S_{i..n}$ of string S. The *longest common extension* of X and Y for (i,j) is the longest prefix of suffixes suf_i and suf_j. We can compute its length $LCE_{X,Y}(i,j)$ as follows: let S be the string $S = X\#Y$ where $\#$ is a special character that is not included in alphabet Σ. Then, $LCE_{X,Y}(i,j) = LCE_S(i, |X|+1+j) =$

$$\begin{cases} 0, & \text{if } S_i \neq S_{|X|+1+j}\ ; \\ \max\{k \mid S_{i..(i+k-1)} = S_{(|X|+1+j)..(|X|+k)}\}, & \text{otherwise.} \end{cases} \tag{3}$$

The longest common extension problem for string S can be answered by finding the lowest common ancestor in the suffix tree that represents S. There exist algorithms (see for example [2] and [4]) that construct a suffix tree in $O(n)$ time, so that subsequent lowest common ancestor queries can be answered in constant time per query. Similarly the *lce* problem can also be solved in constant time per query using a suffix array and additional auxiliary arrays, all of which can be created in $O(n)$ time [7]. Theoretical and practical improvements over the *lce* problem can be found in [4], and a study and comparison of practical *lce* algorithms can be found in [6].

Proposition 1. *Given two strings X and Y each of length $O(n)$ over alphabet Σ, we create the string $S = X\#Y$, where $\# \notin \Sigma$. Let D be a suffix tree, or a suffix array (depending on our choice) of the string S. We create D in time $O(n)$. Subsequently all longest common extension queries $LCE_S(i, |X|+1+j)$ for all possible i, j can be answered in constant time using available constant-time lce algorithms on D.*

3 An Algorithm for the Parametric *LCS* Problem

We first present an algorithm for the following *parametric lcs problem*: given that the edit distance $e_{1,2}$ between X and Y is at most $\hat{e}$, find an *lcs* of X and Y. We start with a definition. For our algorithm for this parametric problem, we describe it first in details, and address its correctness second.

Definition 1. *An edit path $\tilde{p}$ is a* **maximally match extended** *edit path if the following is true: on $\tilde{p}$ for every match arc $((i-1, j-1), (i,j))$ (i.e. $X_i = Y_j$), the diagonal path from $(i-1, j-1)$ to $(i+k-1, j+k-1)$ is part of $\tilde{p}$, where k is the largest number such that $X_{i..(i+k-1)} = Y_{j..(j+k-1)}$ (i.e. $k = LCE_S(i, |X|+1+j)$).*

We can see that on a maximally match extended edit path each match block is a longest common extension, i.e. every match is followed by longest consecutive matches.

Lemma 1. *For given X and Y, there exists a maximally match extended edit path that is also an lcs path.*

Proof. Let $\tilde{a}$ be an *lcs* path for X and Y. If $\tilde{a}$ is not a maximally match extended edit path then let $\tilde{a}$ include a match arc $((i-1, j-1), (i,j))$ (i.e. $X_i = Y_j$) on the edit graph, and let $\tilde{a}$ not include completely the longest common extension

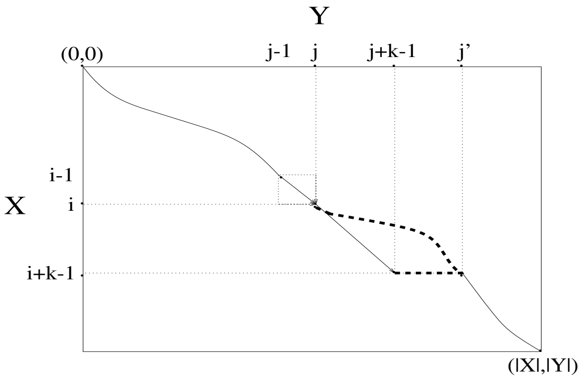


Fig. 1. If there exists an *lcs* path using arc $((i-1, j-1), (i, j))$ then there exists an *lcs* path that uses the longest common extension starting at position (i, j) (i.e. the maximal match block from $(i-1, j-1)$ to $(i+k-1, j+k-1)$, where $k = LCE_S(i, |X|+1+j)$)

starting with this match. Let the length of the longest common extension be k. Let $\tilde{a}$ pass through some node $(i+k-1, j')$, where $j' > j+k-1$ ($j' < j+k-1$ is another possible but symmetric case, therefore, we do not consider this case separately). We schematically summarize these in Figure 1. Since the diagonal path from $(i-1, j-1)$ to $(i+k-1, j+k-1)$ is completely composed of match arcs, the number of match arcs on it is larger than or equal to the number of matches on any edit path from (i, j) to $(i+k-1, j')$ for $j' > j+k-1$. This means that we can obtain another *lcs* path from $\tilde{a}$ by replacing the part of the path from $(i-1, j-1)$ to $(i+k-1, j')$ by the longest common extension corresponding to the diagonal path from $(i-1, j-1)$ to $(i+k-1, j+k-1)$ followed by the horizontal path that ends at $(i+k-1, j')$. We can continue this replacement process in other parts of the new path until the resulting *lcs* path becomes a maximally match extended edit path.

Proposition 2. *An lcs of given strings X and Y can be computed by modifying the dynamic programming computation of the edit distance $e_{1,2}$ such that whenever a new match is encountered only the longest common extension that starts with this match is considered in its entirety as part of a possible lcs path. That is, intermediate nodes on this longest common extensions are all ignored, the optimum lengths are computed for the end points of the corresponding longest common extensions only (We note, however, that an intermediate match in one longest common extension may be considered as the first match for another longest common extension when multiple paths being explored intersect in this match).*

This proposition suggests an algorithm for the parametric problem we defined at the beginning of this section: compute an *lcs* of X and Y given that the edit distance $e_{1,2}$ between them is $\leq \hat{e}$. For this problem, we can narrow the computations to a diagonal band in the edit graph. This is a strategy that accelerates computing the edit distance which was used first by Ukkonen [12].

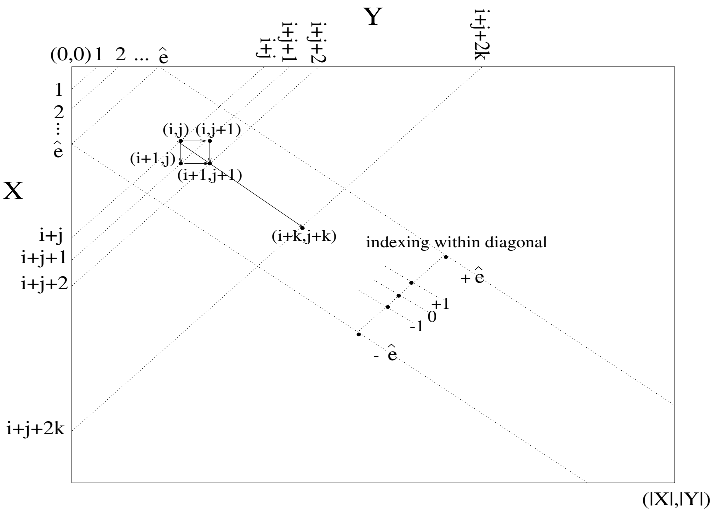


Fig. 2. Diagonals and schematic description of indexing and processing order of nodes

Consider on the edit graph of X and Y the longest diagonal path starting with arc $((0,0),(1,1))$. We only need to examine nodes whose distance diagonally are within $\hat{e}$ from this diagonal path. This defines a diagonal band that contains $O(n\hat{e})$ nodes as can be seen in Figure 2. Every optimal path is completely included in this band. We will show later that it is enough to explore only a small subset of the nodes in this band when $\hat{e}$ is sufficiently small. We can compute the lcs length as follows: Starting from node $(0,0)$, calculate optimum total edit path weights to nodes using the dynamic programming formulation for edit distance $e_{1,2}$. Explore and maintain only those nodes whose weights calculated in this way are at most $\hat{e}$. Additionally whenever a visited node (i,j) is a match enter its end-point into the set of nodes to be explored, and set its weight to the same as that of (i,j). Since we are not examining all nodes within the band, we decide to maintain them in an ordered queue (or queues). On the edit graph, we index back diagonals such that all nodes (i,j) appear in back diagonal $i+j$ as we show in Figure 2. We define a key for each (i,j) as follows:

$$key(i,j) = (i+j)(2\hat{e}+1) + j - i \qquad (4)$$

The key for (i,j) is the sum of two parts: the base $(i+j)(2\hat{e}+1)$, and the offset $j-i$. Since each back diagonal contains at most $2\hat{e}+1$ nodes that are within diagonal distance $\hat{e}$ from the main diagonal, the difference in bases for $i+j+1$ and for $i+j$ is large enough to accommodate all nodes on back diagonal $i+j$ without causing any collision in keys. The offset is $j-i$ and it runs from $-\hat{e}$ to $+\hat{e}$ on a given back diagonal within the band as we show in Figure 2. For the nodes on the main forward diagonal, where $j=i$, the offset is 0. For each node (i,j) we assign a weight $w(i,j)$ whose final value after the computations will be equal to the edit distance $e_{1,2}$ between prefixes $X_{1..i}$ and $Y_{1..j}$. During the computations we enter new weights for nodes in Q to examine. Q is a queue ordered on weights (min heap). We use another data structure R, which is a queue ordered on key values of nodes as described in Equation (4). As nodes are processed in Q, the weights of reachable nodes are entered or updated in R. New reachable nodes are entered into Q if they are not already in R, or if they have not already been processed with a smaller weight and placed in R. New minimal values are updated in R. We show in Figure 3 function *Update-QR* that implements these. We propose Algorithm *LCSP* shown in Figure 4 for the parametric *lcs*

```
Function $Update\text{-}QR(node\ (u,v), weight\ w)$
    set $k = key(u,v)$ as defined in Equation (4)
    if a node with key $k$ does not exist in $R$ then
      { enter this node $(u,v)$ with weight $w$ as its key in $Q$
        enter this node $(u,v)$ with weight $w$, and $k$ as its key in $R$ }
    else { let $w'$ be the weight of this node $(u,v)$ in $R$
              if $w < w'$ then
                      { enter this node $(u,v)$ with weight $w$ as its key in $Q$
                        update the weight of this node $(u,v)$ to $w$ in $R$ }
}
```

Fig. 3. Function *Update-QR* enters/updates queues Q and R for a given node (u,v) with weight w

problem. The algorithm processes nodes (i,j) in Q in their non-decreasing order of weights. It starts with node $(0,0)$ with weight 0. When it processes node (i,j) it enters in Q, or enters and updates the weights of the nodes in R reachable by a longest common extension whose starting match is at $(i+1,j+1)$, or by a single arc if the resulting weights are $\leq \hat{e}$. If a node reachable from node (i,j) via a horizontal arc (node $(i,j+1)$) or a vertical arc (node $(i+1,j)$) is not present in R or present but with a larger weight then it is inserted into Q by invoking the function *Update-QR* which calculates the key and enters weight $w(i,j)+1$; and in R its weight is updated to $w(i,j)+1$ if this value is smaller than its current weight in R. We note that $w(i,j)+1$ is the edit distance that can be obtained at nodes $(i,j+1)$ and $(i+1,j)$ using the corresponding arc from node (i,j). If X_{i+1} and Y_{j+1} exist ($i+1$ and $j+1$ are within the bounds) then we consider them, too. If $X_{i+1} \neq Y_{j+1}$ then node $(i+1,j+1)$ is entered or its weight is updated in Q by invoking the function *Update-QR* which calculates the key and uses the weight $w(i,j)+2$ to enter or update Q and R (if this gives a smaller value in this case) the weight of this node. We note that in computing $e_{1,2}$ the cost of a mismatch is 2. If $X_{i+1} = Y_{j+1}$ then the longest common extension length k is computed by invoking $LCE_S(i+1, |X|+1+j+1)$ as described in Proposition 1. The information about node $(i+k, j+k)$ is entered/updated by invoking *Update-QR* which calculates the key and uses the weight $w(i,j)$ to enter or update Q and R (if this gives a smaller value in this case) the weight of this node. We note that the total edit cost of a longest common extension itself is 0. After finishing processing all reachable nodes from (i,j) this way, the algorithm removes it from

```
Algorithm $LCSP(X, Y, D, \hat{e})$
   Enter $(0, 0)$ with key $0$ and weight $w(0, 0) = 0$ into $Q$
   repeat
      let $(i, j)$ be the head node with the minimum weight $w$ in $Q$
      if $(i, j) = (|X|, |Y|)$ then
        break (go to the next statement after repeat-until)
      if $i + 1 \leq |X|$ and $w + 1 \leq \hat{e}$ then $UpdateQ((i + 1, j), w + 1)$ }
      if $j + 1 \leq |Y|$ and $w + 1 \leq \hat{e}$ then $UpdateQ((i, j + 1), w + 1)$ }
      if $(i + 1 \leq |X|$ and $j + 1 \leq |Y|\}$
      {
        if $X[i + 1] \neq Y[j + 1])$ then
          { if $w + 2 \leq \hat{e}$ then $UpdateQ((i + 1, j + 1), w + 2)$ }
        else { ** find the largest $k$
                    such that $X[(i + 1)..(i + k)] = Y[(j + 1)..(j + k)]$ **
                 set $k = LCE_S(i + 1, |X| + 1 + j + 1)$, where $S = X\#Y$
                 (use the given data structure $D$ as described in Prop. 1)
                 $UpdateQ((i + k, j + k), w)$   }
      }
      remove the head node $(i, j)$ from $Q$
   until ($Q$ is empty)
   if $R$ contains $(|X|, |Y|)$ then return $(|X| + |Y| - w(|X|, |Y|))/2$
    (where $w(|X|, |Y|)$ is the weight of node $(|X|, |Y|)$ in $R$)
   else return $0$
```

Fig. 4. Algorithm *LCSP* for finding the *lcs* length if $e_{1,2}$ is at most $\hat{e}$

Q. It is easy to see that $w(i, j)$ is the minimum weight for node (i, j) since all future nodes in Q will have weights at least $w(i, j)$. The algorithm continues with a node whose key is the minimum in the remaining (updated) Q. This process continues for every node in Q until Q is empty. When node $(|X|, |Y|)$ becomes the head of Q (i.e. when node $(|X|, |Y|)$ has minimal weight in Q) then the repeat-until loop breaks, and this node's weight in R is used as the weight $e_{1,2}$ in Equation (2) in calculating and returning the *lcs* length. Occurrence of this is an indication that the weight of $(|X|, |Y|)$ is $e_{1,2} \leq \hat{e}$. If this never happens, Q will eventually become empty, and node $(|X|, |Y|)$ is not a reachable node. In this case, the function returns 0 indicating that the edit distance $e_{1,2}$ between X and Y is larger than $\hat{e}$, and the *lcs* length cannot be computed with this parameter (a larger parameter $\hat{e}$ is needed). In Figure 5, we show results of Function *LCSP* for parameter $\hat{e} = 4$ (this is a large enough edit distance to find the *lcs* length in this case). The numbers in italics are the weights the function computes at nodes (i.e. weights in R). We also show an optimal edit path for this case. The algorithm starts with node $(0, 0)$ in Q. When this node is processed, nodes $(0, 1)$ and $(1, 0)$ are entered into Q, and into R with weight 1. Similarly new nodes are entered into Q when a head node in Q (a node with minimal weight) is processed. In this example, all nodes with weight 1 are processed, then those with weight 2. New minimal weights are recorded in R as described in the function description. We note that the weights computed at the nodes are in general the edit distances

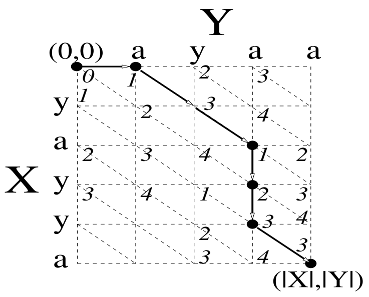


Fig. 5. An example edit distance $e_{1,2}$ (or, equivalently, the *lcs*) computation: explored nodes and their calculated weights, and an optimal path

$e_{1,2}$ for the corresponding nodes. Only exceptions are the nodes that appear only as intermediate nodes in longest common extensions (see the weights for example at nodes $(1,2)$ and $(2,1)$). However, in these cases, the weights are correct at the end nodes of the corresponding longest common extensions (those that include these nodes as intermediate matches). By Lemma 1, this is sufficient to conclude that the function will compute the optimum edit distance. In this particular example, node $(0,1)$ is one of the nodes that is entered into Q and R with weight 1. Later when the algorithm picks $(0,1)$ from the head of Q, it enters (among other nodes), node $(2,3)$ with weight 1 into Q and R. Similarly, in later steps node $(3,3)$ with weight 2, and node $(4,3)$ with weight 3 are entered into Q and R. When node $(4,3)$ is picked from the head of Q, node $(5,4)$ is entered in Q and R. The algorithm explores and deletes all the nodes with weights 1 and 2. We assume without loss of generality that node $(5,4)$ is the first node with weight 3 chosen to be processed. This means that the minimum edit distance $e_{1,2}$ is achieved at node $(5,4)$. The algorithm returns *lcs* length 3 for $e_{1,2} = 3$ by Equation 2.

Lemma 2. *Let $e_{1,2}$ be the edit distance between X and Y (assume $e_{1,2} > 0$ without loss of generality). Let $\delta_{i,j}$ denote the edit distance for $X_{1..i}$ and $Y_{1..j}$. After $LCSP(X, Y, D, \hat{e})$ runs for $\hat{e} \geq e_{1,2}$, for all edit distances d, $0 \leq d < e_{1,2}$, for all nodes (i,j) on the edit graph, if (i,j) is the last match of some longest common extension on some maximally match extended edit path for X and Y, or if $X_i \neq Y_j$, then $\delta_{i,j}$ is the weight of node (i,j) in queue R.*

We note that the lemma excludes the nodes that appear as intermediate match on all possible maximally extended edit paths for X and Y. Those nodes are unnecessary and all are ignored. Also, the edit distance $d = e_{1,2}$ is excluded because when node $(|X|, |Y|)$ has the minimum weight in Q (termination condition), some nodes with weight $e_{1,2}$ may not have been all explored yet. We can prove this lemma by induction on edit distances d. For the base case, for $d = 0$, node $(0,0)$ has weight 0 in R. If there is a longest common extension starting at $(1,1)$, its end-point is entered in R when node $(0,0)$ is processed. These are the only possible nodes with weight 0, and the lemma is correct in this case. Suppose that for all d', $0 \leq d' < d$, for all nodes (i, j), if (i, j) is the last match of some longest common extension on some maximally match extended edit path for X and Y, or if $X_i \neq Y_j$, then $\delta_{i,j}$ is the weight of node (i, j) in queue R. Since these nodes are in R, they must have been processed when they were in Q, and all reachable nodes from them via a single arc, or a longest common extension have been entered into Q. Consider nodes (i, j), where $\delta_{i,j} = d$, and which are the last match of some longest common extension on some maximally match extended edit path for X and Y, or $X_i \neq Y_j$. By the induction hypothesis there exist at least one of the following: node $(i-1, j)$ with weight $d-1$, node $(i, j-1)$ with weight $d-1$, or a node $(i-1, j-1)$ with weight $d-2$ and $X_i \neq Y_j$. In any of these cases, weight d is obtained at (i, j) and updated in R. This concludes the proof of the claim in the lemma.

Consider an optimal edit path ending at node $(|X|, |Y|)$. If this node is not the end point of a longest common extension, it is reached on this path from one of its neighbors at $(i-1, j)$, $(i, j-1)$, or $(i-1, j-1)$. Lemma 2 implies in this case that the weight of node $(|X|, |Y|)$ in R is $e_{1,2}$. If node $(|X|, |Y|)$ is the end point of some longest common extension then let (i', j') be the node where the first match of this longest common extension is located. This implies that at least one of the nodes at $(i'-1, j')$, $(i', j'-1)$, or $(i'-1, j'-1)$ has weight less than $e_{1,2}$. Lemma 2 implies that one or more of these nodes yield weight $e_{1,2}$ first at node (i', j') then at $(|X|, |Y|)$ (via the longest common extension) in R.

Let $f(d)$ be the total number of times our algorithm assigns the weight d during the course of its execution to nodes which are not ends of longest common extensions on maximally extended edit paths that our algorithm explores (what we mean by assignment here is an attempt to assign a new minimum weight to a node, i.e. a read/insert/update of a node's weight in Q and subsequently in R). The total number of times our algorithm assigns the weight d to nodes which are ends of maximal match blocks on edit paths explored cannot be larger than d because we enter or update these nodes in R after, for each such node, we find a node with weight d that is the start of a longest common extension. Therefore, the true count for the total assignment of the weight d is at most $2f(d)$ (in the worst each assignment causes another assignment: one at the end point of the corresponding longest common extension). If a given node (i, j) is not the end of a longest common extension, there are three reasons our algorithm inserts or updates the node's weight to d: 1) node $(i-1, j)$ has weight $d-1$, or 2) node $(i, j-1)$ has weight $d-1$, or 3) node $(i-1, j-1)$ has weight $d-2$ (note that the weight of a mismatch is 2 in computing $e_{1,2}$). We also note that we do not consider in counting $f(d)$, a match extended from $(i-1, j-1)$ (i.e. a possible block of matches ending at node $(i-1, j-1)$) because our algorithm considers the end point of the longest common extension starting at node (i', j') for every explored node (i', j'). We note that if multiple nodes are visited on some longest common extension, all yield the same end point (the end point of this longest common extension). Hence, we write the following recurrence relation for $f(d)$: $f(d) \leq 2f(d-1) + f(d-2)$ with $f(0) = 1$ (the weight 0 is assigned only once: the weight of node $(0, 0)$ is initialized to 0) and $f(1) = 2$

(there are two weight assignments of weight 1 in total: at nodes $(0, 1)$ and $(1, 0)$ once in each). From this recurrence, we see and verify that $f(d) \leq (1+\sqrt{2})^d$. Therefore, the total number of entries and updates our algorithm performs in Q and R is $\sum_{d=0}^{\hat{e}} f(d) = O((1+\sqrt{2})^{\hat{e}+1})$. Also, since the total number of nodes in the diagonal band is $\leq (2\hat{e}+1)n$, or $O(\hat{e}n)$, and since every time a new node is entered in Q there are at most 3 entries/updates for its neighboring nodes in Q and R, we can see that the number of entries/updates ever attempted in Q and R is $O(\hat{e}n)$. Combining these two bounds, we obtain that the total number of entries/updates in Q and R is $\min\{\hat{e}n, (1+\sqrt{2})^{\hat{e}+1}\}$. Clearly, this is also the upper bound on the size of Q, $|Q|$, and size of R, $|R|$. Since each read/insert/update or delete can be done in time $O(\log |Q|)$ in Q, and $O(\log |R|)$ in R, we can see that the total time our algorithm spends on operations on Q and R is $O(\min\{\hat{e}n \log n, \hat{e}(1+\sqrt{2})^{\hat{e}+1}\})$. We can easily verify that this is the time complexity of $LCSP(X, Y, D, \hat{e})$ since its time complexity is dominated by the total time spent on Q and R.

4 Main LCS Algorithm

We give our main algorithm in Figure 6. We create from X and Y first the data structure D in Proposition 1. The total time we spend on this step is $O(n)$. We do not assume prior knowledge of the exact edit distance $e_{1,2}$. We initialize $\hat{e}$ to 1 and call $LCS(X, Y, D, \hat{e})$. If 0 is returned we double $\hat{e}$ and repeat calling $LCS(X, Y, D, \hat{e})$ until a nonzero value is obtained. When $\hat{e} \geq e_{1,2}$, a nonzero value returned is the *lcs* length. We note that when this happens, $\hat{e} \leq 2e$, where e is the simple edit distance $e = e_{1,1}$. This is because on any edit path every substitution can be replaced by a pair of an insertion and a deletion, and therefore $e_{1,2} \leq 2e$. Therefore, the time complexity of Algorithm LCS is $\sum_{i=1}^{\lceil \log \hat{e} \rceil} \min\{2^i n \log n, 2^i(1+\sqrt{2})^{2^i+1}\} \leq \left(\sum_{i=1}^{\lceil \log \hat{e} \rceil} 2^i\right) \min\{n \log n, (1+\sqrt{2})^{\hat{e}+1}\} = O(\min\{\hat{e}n \log n, \hat{e}(1+\sqrt{2})^{\hat{e}+1}\})$. Including the linear time construction of data structures, the total time complexity of our algorithm is $O(\min\{en \log n, n + e(1+\sqrt{2})^{2e+1}\})$. If an optimal *lcs* (along with the length) is desired then we can compute it in the following way: with each node we store in R, we keep the node id (position pair) from which maximum scores are obtained. We can trace an *lcs* path by following these nodes backwards starting with node $(|X|, |Y|)$ to locate the node from which there is an arc to $(|X|, |Y|)$ on this *lcs* path. We should continue this tracing until reaching node $(0, 0)$.

```
Algorithm $LCS(X,Y)$
    if X=Y then return $|X|$
    Create from $X$ and $Y$ the data structure $D$ in Prop. 1
    set $\hat{e} = 1$
    while (TRUE) { $k = LCSP(X,Y,D,\hat{e})$; if $k > 0$ then return $k$; $\hat{e} = 2 * \hat{e}$ }
```

Fig. 6. Algorithm $LCS(X, Y)$ for finding the *lcs* length of X and Y

5 Conclusion

We present a new algorithm for the longest common subsequence problem. Our algorithm achieves provably superior time complexity compared to existing solutions when the edit distance between input strings is a sufficiently small function of the length of the strings.

References

1. Apostolico, A., Guerra, C.: The longest common subsequence problem revisited. Algorithmica (2), 315–336 (1987)
2. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Gonnet, G.H., Viola, A. (eds.) LATIN 2000. LNCS, vol. 1776, pp. 88–94. Springer, Heidelberg (2000)
3. Bergroth, L., Hakonen, H., Ratia, T.: A survey of longest common subsequence algorithms. In: SPIRE, pp. 39–48 (2000)
4. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
5. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, Cambridge (1997)
6. 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)
7. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest common prefix computation in suffix arrays and its applications. In: Amir, A., Landau, G.M. (eds.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001)
8. Kuo, S., Cross, G.R.: An algorithm to find the length of the longest common subsequence of two strings. ACM SIGIR Forum 23(3-4), 89–99 (1989)
9. Masek, W.J., Paterson, M.S.: A faster algorithm for computing string-edit distances. Journal of Computer and System Sciences 20(1), 18–31 (1980)
10. Miller, W., Myers, E.W.: A file comparison program. Softw. Pract. Exp. 15(11), 1025–1040 (1985)
11. Nakatsu, N., Kambayashi, Y., Yajima, S.: A longest common subsequence algorithm suitable for similar texts. Acta Informatica 18, 171–179 (1982)
12. Ukkonen, E.: Algorithms for approximate string matching. Information and Control 64, 100–118 (1985)
13. Wagner, R.A., Fisher, M.J.: The string-to-string correction problem. Journal of the ACM 21(1), 168–173 (1975)
14. Wu, S., Manber, U., Myers, G., Miller, W.: An O(NP) sequence comparison algorithm. Inf. Proc. Lett. 35, 317–323 (1990)

Abelian Square-Free Partial Words*

Francine Blanchet-Sadri[1], Jane I. Kim[2], Robert Mercaş[3], William Severa[4], and Sean Simmons[5]

[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, Columbia University, 2960 Broadway, New York, NY 10027-6902, USA
[3] Harriet L. Wilkes Honors College, Florida Atlantic University, 5353 Parkside Dr., Jupiter, FL 33458, USA
[4] GRLMC, Universitat Rovira i Virgili, Departament de Filologies Romàniques, Av. Catalunya 35, Tarragona, 43002, Spain
robertmercas@gmail.com
[5] Department of Mathematics, University of Texas at Austin, 1 University Station C1200, Austin, TX 78712-0233, USA

Abstract. Erdös raised the question whether there exist infinite abelian square-free words over a given alphabet (words in which no two adjacent subwords are permutations of each other). Infinite abelian square-free words have been constructed over alphabets of sizes as small as four. In this paper, we investigate the problem of avoiding abelian squares in partial words (sequences that may contain some holes). In particular, we give lower and upper bounds for the number of letters needed to construct infinite abelian square-free partial words with finitely or infinitely many holes. In the case of one hole, we prove that the minimal alphabet size is four, while in the case of more than one hole, we prove that it is five.

1 Introduction

Words or strings belong to the very basic objects in theoretical computer science. The systematic study of word structures (combinatorics on words) was started by a Norwegian mathematician Axel Thue [21,22,1] at the beginning of the last century. One of the remarkable discoveries made by Thue is that the consecutive repetitions of non-empty factors (squares) can be avoided in infinite words over a three-letter alphabet. Recall that an infinite word w over an alphabet is said to be *k-free* if there exists no word x such that x^k is a factor of w. For simplicity, a word that is 2-free is said to be *square-free*.

Erdös [9] raised the question whether *abelian squares* can be avoided in infinitely long words, i.e., whether there exist infinite abelian square-free words

* This material is based upon work supported by the National Science Foundation under Grant No. DMS–0754154. The Department of Defense is also gratefully acknowledged. A World Wide Web server interface has been established at www.uncg.edu/cmp/research/abelianrepetitions for automated use of programs.

over a given alphabet. An abelian square is a non-empty word uv, where u and v are permutations of each other. For example, *abcacb* is an abelian square. A word is called *abelian square-free*, if it does not contain any abelian square as a factor. For example, the word *abacaba* is abelian square-free, while *abcdadcada* is not (it contains the subword *cdadca*). It is easily seen that abelian squares cannot be avoided over a three-letter alphabet. Indeed, each word of length eight over three letters contains an abelian square. A first step in solving special cases of Erdös' problem was taken in [10], where it was shown that the 25th abelian powers were avoidable in the binary case. Later on, Pleasants [20] showed that there exists an infinite abelian square-free word over five letters, using a uniform iterated morphism of size fifteen. This result was improved in [14] using uniform morphisms of size five.

Dekking [8] proved that over a binary alphabet there exists a word that is abelian 4-free. Moreover, using $\mathbb{Z}_7$ instead of $\mathbb{Z}_5$, in the proof of this result, we get that over a ternary alphabet an abelian 3-free infinite word is constructible. The problem of whether abelian squares can be avoided over a four-letter alphabet was open for a long time. In [15], using an interesting combination of computer checking and mathematical reasoning, Keränen proves that abelian squares are avoidable on four letters. To do this, he presents an abelian square-free morphism $g : \{a, b, c, d\}^* \to \{a, b, c, d\}^*$ whose size is $|g(abcd)| = 4 \times 85$:

$$\begin{aligned} g(a) = &\mathtt{abcacdcbcdcadcdbdabacabadbabcbdbcbacbcdcacb} \\ &\mathtt{abdabacadcbcdcacdbcbacbcdcacdcbdcdadbdcbca} \end{aligned}$$

and the image of the letters b, c, d are obtained by cyclic permutation of letters in the preceding words.

Most of the currently known methods, [6], for constructing arbitrarily long abelian square-free words over a four-letter alphabet are based on the structure of this endomorphism g. Moreover, it is shown that no smaller uniform morphism works here! In [16] a completely new morphism of length 4×98, possessing similar properties for iterations, is given.

Now let us move to partial words. Being motivated by a practical problem on gene comparison, Berstel and Boasson introduced the notion of *partial words*, sequences over a finite alphabet that may have some undefined positions or holes (the $\diamond$ symbol represents a hole and matches every letter of the alphabet) [2]. For instance, $a\diamond bca\diamond b$ is a partial word with two holes over the three-letter alphabet $\{a, b, c\}$. Several interesting combinatorial properties of partial words have been investigated, and connections have been made with problems concerning primitive sets of integers, lattices, vertex connectivity, etc [3].

In [19], the question was raised as to whether there exist cube-free infinite partial words, and an optimal construction over a binary alphabet was given (a partial word w is called *k-free*, if for every factor $x_0x_1\cdots x_{k-1}$ of w there does not exist a word u, such that for each i, the defined positions of x_i match the corresponding positions of u). In [5], the authors settled the question of overlap-freeness by showing that over a two-letter alphabet there exist overlap-free infinite partial words with at most one hole, and that a three-letter alphabet is enough for an infinity of holes. An overlap represents a word consisting of two overlapping occurrences of the same factor. The problem of square-freeness in partial words is settled in [5] and [12] where it is shown that a three-letter alphabet is enough for constructing such words. Quite naturally, all the constructions of these words are done by iterating morphisms, most of them uniform, similarly or directly implied by the original result of Thue. Moreover, in [19,5,4], the concept of repetitions is also solved in more general terms. The authors show that, for given alphabets, replacing arbitrary positions of some infinite words by holes, does not change the repetition degree of the word. Furthermore in [13], the authors show that there exist binary words that are 2-overlap-free.

This paper focuses on the problem of avoiding abelian squares in partial words. In Section 2, we give some preliminaries on partial words. In Section 3, we explore the minimal size of alphabet needed for the construction of (two-sided) infinite abelian square-free partial words with a given finite number of holes. In particular, we construct an abelian square-free infinite partial word with one hole without expanding beyond the minimal four-letter alphabet. For more than one hole, the minimal number of letters is at least five, when such words exist. In Section 4, we prove by explicit construction the existence of abelian square-free partial words with infinitely many holes. The minimal alphabet size turns out to be five for such words. In Section 5, we discuss some constructions for the finite case. Finally in Section 6, we conclude with some directions for future work.

2 Preliminaries

Let A be a non-empty finite set of symbols called an *alphabet*. Each element $a \in A$ is called a *letter*. A *full word* over A is a sequence of letters from A. A *partial word* over A is a sequence of symbols from $A_\diamond = A \cup \{\diamond\}$, the alphabet A being augmented with the "hole" symbol $\diamond$ (a full word is a partial word that does not contain the $\diamond$ symbol).

The *length* of a partial word w is denoted by $|w|$ and represents the number of symbols in w, while $w(i)$ represents the ith symbol of w, where $0 \leq i < |w|$. The *empty word* is the sequence of length zero and is denoted by ε. The set of all words over A is denoted by A^*, while the set of all partial words over A is denoted by $A_\diamond^*$. A (right) (resp., two-sided) infinite partial word is a function $w : \mathbb{N} \to A_\diamond$ (resp., $w : \mathbb{Z} \to A_\diamond$).

Let u and v be partial words of equal length. Then u is said to be *contained* in v, denoted $u \subset v$, if $u(i) = v(i)$, for all i such that $u(i) \in A$. Partial words u and v are *compatible*, denoted $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$. If u and v are non-empty, then uv is called a *square*. Whenever we refer to a square uv, it implies that $u \uparrow v$.

A partial word u is a *factor* or *subword* of a partial word v if there exist x, y such that $v = xuy$. We say that u is a *prefix* of v if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$. If $w = a_0a_1 \cdots a_{n-1}$, then $w[i..j) = a_i \cdots a_{j-1}$ and $w[i..j] = a_i \cdots a_j$. The *reversal* of a partial word $w = a_0a_1 \cdots a_{n-1}$, where each $a_i \in A_\diamond$, is simply the word written backwards $a_{n-1} \cdots a_1 a_0$, and is denoted $\text{rev}(w)$. For partial words u and v, $|u|_v$ denotes the number of occurrences of v found in u. The *Parikh vector* of a word $w \in A^*$, denoted by $P(w)$, is defined as $P(w) = \langle |w|_{a_0}, |w|_{a_1}, \ldots, |w|_{a_{\|A\|-1}} \rangle$, where $A = \{a_0, a_1, \ldots, a_{\|A\|-1}\}$ (here $\|A\|$ denotes the cardinality of A).

A word $uv \in A^+$ is called an *abelian square* if $P(u) = P(v)$. A word w is *abelian square-free* if no factor of w is an abelian square.

Definition 1. *A partial word $w \in A_\diamond^+$ is an abelian square if it is possible to substitute letters from A for each hole in such a way that w becomes an abelian square full word. The partial word w is abelian square-free if it does not have any full or partial abelian square, except those of the form $\diamond a$ or $a\diamond$, where $a \in A$.*

A morphism $\phi : A^* \to B^*$ is called *abelian square-free* if $\phi(w)$ is abelian square-free whenever w is abelian square-free.

3 The Infinite Case with a Finite Number of Holes

It is not hard to check that every abelian square-free full word over a three-letter alphabet has length less than eight. Using a computer it can be checked that the maximum length of an abelian square-free partial word, over such an alphabet, is six. So to construct infinite partial words with a finite number of holes, we need at least four letters. Let us first state some remarks.

Remark 1. Let $w \in A^*$ be an abelian square-free word. Inserting a new letter a, $a \notin A$, between arbitrary positions of w (so that aa does not occur) yields a word $w' \in (A \cup \{a\})^*$ that is abelian square-free.

Consider *abacba* which is abelian square-free. Inserting letter d between positions 0 and 1, 3 and 4, and 5 and 6, yields *adbacdbda* which is abelian square-free.

Remark 2. Let $uv \in A^*$ with $|u| = |v|$, $a \in A$ and $b \notin A$. Replace a number of a's in u and the same number of a's in v with b's, yielding a new word $u'v'$. If uv is an abelian square, then $u'v'$ is an abelian square. Similarly, if uv is abelian square-free, then $u'v'$ is abelian square-free.

The question whether there exist infinite abelian square-free full words over a given alphabet was originally raised by Erdös in [9]. As mentioned above, no such word exists over a three-letter alphabet. However, infinite abelian square-free full words are readily available over a four-letter [15,16,17], five-letter [20], and larger alphabets [11]. These infinite words are created using repeated application of morphisms, where most of these morphisms are abelian square-free.

We now investigate the minimum alphabet size needed to construct infinite abelian square-free partial words with a given finite number of holes.

Remark 3. Let u, v be partial words of equal length. If uv is an abelian square, then so is any permutation of u concatenated with any permutation of v.

Theorem 1. *There exists an infinite abelian square-free partial word with one hole over a four-letter alphabet.*

Proof. We use an abelian square-free morphism $\phi : A^* \to A^*$, where $A = \{a, b, c, d\}$, provided by Keränen [17] that is defined by

$$\begin{aligned}\phi(a) =& \texttt{abcacdcbcdcadbdcadabacadcdbcbabcbdbadbdcbabcbdcdacd}\\ & \texttt{cbcacbcdbcbabdbabcabadcbcdcbadbabcbabdbcdbdadbdcbca}\\ \phi(b) =& \texttt{bcdbdadcdadbacadbabcbdbadacdcbcdcacbacadcbcdcadabda}\\ & \texttt{dcdbdcdacdcbcacbcdbcbadcdadcbacbcdcbcacdacabacadcdb}\\ \phi(c) =& \texttt{cdacabadabacbdbacbcdcacbabdadcdadbdcbdbadcdadbabcab}\\ & \texttt{adacadabdadcdbdcdacdcbadabadcbdcdadcdbdabdbcbdbadac}\\ \phi(d) =& \texttt{dabdbcbabcbdcacbdcdadbdcbcabadabacadcacbadabacbcdbc}\\ & \texttt{babdbabcabadacadabdadcbabcbadcadabadacabcacdcacbabd}\end{aligned}$$

The length of each image is 102 and the Parikh vector of each is a permutation of $P(\phi(a)) = \langle 21, 31, 27, 23\rangle$. We show that the word $\diamond\phi^n(a)$ is abelian square-free for all integers $n \geq 0$. Since ϕ is abelian square-free, it is sufficient to check if we have abelian squares uv that start with the hole, for $|u| = |v|$.

We refer to the factors created by the images of ϕ as blocks. Now, assume that some prefix uv of $w = \diamond\phi^n(a)$ is an abelian square. We can write $uv = \diamond\phi(w_0)\phi(e)\phi(w_1)x$, where $e \in A$, $w_0, w_1, x \in A^*$ are such that $\diamond\phi(w_0)$ is a prefix of u, u is a proper prefix of $\diamond\phi(w_0 e)$, and $|x| < 102$. If we delete the same number of occurrences of any given block present in both $\phi(w_0)$ and $\phi(w_1)$, we claim that we only need to consider the case where $0 \leq |w_1| \leq |w_0| < 2$ (the case $|w_1| > |w_0|$ obviously leads to $|u| < |v|$, which is a contradiction). If this were not the case, then the reduced w_0 and w_1 would have no letter in common. Denoting by u' the word obtained from u after replacing the hole by a letter in A so that $P(u') = P(v)$, we can build a system of equations for each letter in A.

For example, the system for letter a is determined by

$$|u'|_{\phi(a)} + |u'|_{\phi(b)} + |u'|_{\phi(c)} + |u'|_{\phi(d)} = |v|_{\phi(b)} + |v|_{\phi(c)} + |v|_{\phi(d)} + \Lambda$$

The number of occurrences of a (resp., b, c, d) in u' and v must be equal, so we get the system of equations:

$$\begin{aligned}21|u'|_{\phi(a)} + 23|u'|_{\phi(b)} + 27|u'|_{\phi(c)} + 31|u'|_{\phi(d)} &= 23|v|_{\phi(b)} + 27|v|_{\phi(c)} + 31|v|_{\phi(d)} + \lambda_a\\ 31|u'|_{\phi(a)} + 21|u'|_{\phi(b)} + 23|u'|_{\phi(c)} + 27|u'|_{\phi(d)} &= 21|v|_{\phi(b)} + 23|v|_{\phi(c)} + 27|v|_{\phi(d)} + \lambda_b\\ 27|u'|_{\phi(a)} + 31|u'|_{\phi(b)} + 21|u'|_{\phi(c)} + 23|u'|_{\phi(d)} &= 31|v|_{\phi(b)} + 21|v|_{\phi(c)} + 23|v|_{\phi(d)} + \lambda_c\\ 23|u'|_{\phi(a)} + 27|u'|_{\phi(b)} + 31|u'|_{\phi(c)} + 21|u'|_{\phi(d)} &= 27|v|_{\phi(b)} + 31|v|_{\phi(c)} + 21|v|_{\phi(d)} + \lambda_d\end{aligned}$$

The parameter Λ is an error term taking values in $\{-1, 0, 1\}$, but can only be non-zero for one system of equations (this is because it obviously replaces only one of the images). Each λ_i represents the error caused by $\diamond$, $\phi(e)$ or x. Using Gaussian elimination, it is easy to see that this system is inconsistent provided that some λ_i is distinct from 0. However, the hole at the beginning ensures at least one non-zero λ_i. Thus, $w_0 = w_1 = \varepsilon$ yielding $uv = \diamond\phi(e)x$, or $w_0 = f \in A$ and $w_1 = \varepsilon$ yielding $uv = \diamond\phi(f)\phi(e)x$. It is easy to verify that all such partial words are abelian square-free. □

Corollary 1. *There exists a two-sided infinite abelian square-free partial word with one hole over a five-letter alphabet.*

Proof. For a word w, let $\phi'(w) = \text{rev}(\phi(w))$ with $\phi : A^* \to A^*$ being the morphism from the proof of Theorem 1. Hence, $\phi'(w)$ is abelian square-free for all abelian square-free words w and $\phi'^n(a)\diamond$ is abelian square-free for all integers $n \geq 0$. Also, let $\chi : B^* \to B^*$, where $B = \{b, c, d, e\}$, be the morphism that is constructed by replacing each a in the definition of ϕ with a new letter e. By construction, χ is an abelian square-free morphism and $\diamond\chi^n(e)$ is abelian square-free for all integers $n \geq 0$.

We show that $\phi'^n(a)\diamond\chi^n(e) \in \{a, b, c, d, e\}^*$, is abelian square-free for all integers $n \geq 0$. Suppose to the contrary that there exists an abelian square w, which is a subword of $\phi'^n(a)\diamond\chi^n(e)$, for some integer $n \geq 0$. Then, the word w must contain parts of both $\phi'(a)$ and $\chi(e)$. Therefore, at least one half, called u, is a subword of either $\phi'^n(a)$ or $\chi^n(e)$ meaning it contains either a or e but not both and it does not contain the hole. Whereas the other half of w, called v, necessarily contains the other letter and the hole. Since v contains a letter that u does not, and u has no holes, w is not an abelian square. □

Corollary 2. *The word $\phi'^n(a)\diamond efg\diamond\phi^n(a) \in \{a, b, c, d.e, f, g\}_\diamond^*$ is a two-sided infinite abelian square-free partial word with two holes over a seven-letter alphabet.*

Using a computer program, we have checked that over a four-letter alphabet all words of the form $u\diamond v$, where $|u| = |v| = 12$, contain an abelian square. It follows that, over a four-letter alphabet, an infinite abelian square-free partial word containing more than one hole, must have all holes within the first 12 positions. We have also checked that all partial words $\diamond u\diamond v$ with $|u| \leq 10, |v| \leq 10$ or with $|u| = 11, |v| = 5$ contain abelian squares (and consequently so do the words with $|u| = 11$ and $|v| \geq 5$).

Proposition 1. *Over a four-letter alphabet, there exists no two-sided infinite abelian square-free partial word with one hole, and all right infinite partial words contain at most one hole.*

In addition, over a four-letter alphabet, for all words u and v, $|u|, |v| \leq 12$, the partial word $\diamond u\diamond v\diamond$ contains an abelian square.

Proposition 2. *If a finite partial word over a four-letter alphabet contains at least three holes, then it has an abelian square.*

4 The Case with Infinitely Many Holes

The next question is how large should the alphabet be so that an abelian square-free partial word with infinitely many holes can be constructed. In this section, we construct such words over a minimal alphabet size of five.

Theorem 2. *There exists an abelian square-free partial word with infinitely many holes over a seven-letter alphabet.*

Proof. According to [15], there exists an infinite abelian square-free word w over a four-letter alphabet $A = \{a, b, c, d\}$. Furthermore, there exist some distinct letters $x, y, z \in A$ so that for infinitely many j's we have $w(j-1) = z$, $w(j) = x$, and $w(j+1) = y$. Let k_0 be the smallest integer such that $w(k_0 - 1) = z$, $w(k_0) = x$ and $w(k_0+1) = y$. Then, define k_j recursively, where k_j is the smallest integer such that $k_j > 5k_{j-1}$, $w(k_j - 1) = z$, $w(k_j) = x$ and $w(k_j + 1) = y$. Moreover, define $A' = A \cup \{e, f, g\}$, where $e, f, g \notin A$.

Note that in order to avoid abelian squares, the holes must be somehow sparse. We now define an infinite partial word w'. For any integer $i \geq 0$, there exists some integer $j \geq 0$ so that $k_j - 1 \leq i < k_{j+1} - 1$. If $j \equiv 0 \bmod 5$, then for $i = k_j - 1$, let $w'(i) = e$; for $i = k_j$, let $w'(i) = \diamond$; for $i = k_j + 1$, let $w'(i) = f$. If $j \not\equiv 0 \bmod 5$ and $i = k_j$, then let $w'(i) = g$. For all other i's, let $w'(i) = w(i)$. Clearly, w' has infinitely many holes. The modulo 5 here helps prevent the creation of squares, by assuring that the occurrences of a letter grow faster than the ones of the hole.

In order to prove that w' has no abelian squares, we assume that it has one and get a contradiction. Let uv be an occurrence of an abelian square, where $u = w'[i..i+l]$ and $v = w'[i+l+1..i+2l+1]$ for some i, l. Let $J_1 = \{j \mid i \leq k_j \leq i+l\}$ and $J_2 = \{j \mid i+l+1 \leq k_j \leq i+2l+1\}$. Then $|J_1| < 3$ and $|J_2| < 2$, which implies $|J_1 \cup J_2| < 4$. To see this, first assume that $|J_2| > 1$. Note that there exists $j \in J_2$ so that $j+1 \in J_2$. However, this implies that $l = i+2l+1-(i+l+1) \geq k_{j+1} - k_j > k_j > i+l \geq i+l-i = l$, a contradiction. Now assume that $|J_1| > 2$. Then there are at least two occurrences of the letter g in u, and for each occurrence of g there must also be a g or a hole in v. However, g's and holes only occur when $i = k_j$ for some j, so this implies $|J_2| \geq 2$, which violates the claim that $|J_2| < 2$.

Next, we want to show that no holes occur in the abelian square uv. We prove none occurs in u, the case when the hole is in v being similar. The occurrence of a hole in u implies that there exists j so that $i \leq k_j \leq i+l$. Note that this implies $l > 0$, since otherwise $uv = \diamond w(i+1)$ would be a trivial square. Therefore either e or f occurs in u, since u must contain either $w'(k_j - 1)$ or $w'(k_j + 1)$. Assume that e occurs, the f case being similar. Then v must contain either e or a hole, but that implies $i+l \leq k_{j+5} - 1 \leq i+2l+1$, since $w'(k_{j+5} - 1)$ is the next occurrence in w' of either e or a hole. Thus $j, j+1, \ldots, j+4 \in J_1 \cup J_2$ which implies that $|J_1 \cup J_2| \geq 5 > 3$, a contradiction, so no such hole can exist. Therefore, all symbols in uv are letters in A'. By Remark 2, since w' contains an abelian square, w must also contain an abelian square. □

Using a similar construction we can reduce the alphabet size to six.

Theorem 3. *There exists an abelian square-free partial word with infinitely many holes over a six-letter alphabet.*

The next question is whether or not it is possible to construct such partial words over a five-letter alphabet. Although somehow superfluous, the previous two theorems give both the method and history that were used to prove our main result. First let us state two lemmas that help us achieve our goal.

Lemma 1. *Let z be a word which is not an abelian square, x (resp., y) be a prefix (resp., suffix) of $\phi(e)$, where ϕ is defined as in Theorem 1 and $e \in \{b, c, d\}$. No* *word of the form $\phi(z)y$, $a\phi(z)y$ or $x\phi(z)$, preceded or followed by a hole, is an abelian square, unless either ez or ze is an abelian square.*

Lemma 2. *Let z be a word which is not an abelian square, x (resp., y) be a prefix (resp., suffix) of $\phi(a)$, where ϕ is defined as in Theorem 1. Then, no word of the form $\diamond x\phi(z)y$ is an abelian square, unless az or za is an abelian square.*

Theorem 4. *There exists an abelian square-free partial word with infinitely many holes over a five-letter alphabet.*

Proof. Let us denote by w the infinite abelian square-free full word over $A = \{a,b,c,d\}$ from the proof of Theorem 1. There exist infinitely many j's such that $w[j-101..j] = \phi(a)$. Let k_0 be the smallest integer so that $w[k_0-101..k_0] = \phi(a)$. Then define k_j recursively, where k_j is the smallest integer such that $k_j > 5k_{j-1}$ and $w[k_j - 101..k_j] = \phi(a)$.

Construct an infinite partial word w' over $A \cup \{e\}$ by introducing factors in w as follows. Let $j \geq 0$. If $i = k_j$ and $j \equiv 0 \bmod 5$, then introduce $\diamond e$ between positions i and $i+1$ of w. If $i = k_j$ and $j \not\equiv 0 \bmod 5$, then introduce four e's in the image of $\phi(a)$ that ends at position i, in the following way: setting $w[k_j - 101..k_j] = \phi(a) = abXca$, where $X \in A^*$, the word $abXca$ is replaced with $X' = eaebXceae$. Clearly, w' has infinitely many holes. Moreover, if the holes are not taken into consideration, since no two e's are next to each other, by Remark 1, the word is still abelian square-free.

In order to prove that w' has no abelian squares, we assume that it has one and get a contradiction. Let uv be an occurrence of an abelian square, where $u = w'([i..i+l]$ and $v = w'[i+l+1..i+2l+1]$ for some i, l. Let $J_1 = \{j \mid i \leq k_j \leq i+l\}$ and $J_2 = \{j \mid i+l+1 \leq k_j \leq i+2l+1\}$. Then $|J_1| < 4$ and $|J_2| < 2$, which implies $|J_1 \cup J_2| < 5$. As in Theorem 2, it is trivial to show that $|J_2| < 2$. Now assume that $|J_1| > 3$. Then there are at least seven occurrences of the letter e in u, and for each occurrence of e there must also be an e or a hole in v. However, this implies $|J_2| \geq 2$, which violates the fact that $|J_2| < 2$.

Next, we want to show that no holes occur in the abelian square uv. First observe that v cannot contain more than four e's, since otherwise, $|J_1| > 6$, a contradiction. If the last position of u is a hole, then v contains an e. If no e occurs in u, then the hole in u and the e in v are cancelling each other, giving us a factor of the original word, which is abelian square-free. So there must exist an e in u. But, this implies that there exists an abelian square of one of the forms $e\phi(z_0)\diamond e\phi(z_1)y$ or $ae\phi(z_0)\diamond e\phi(z_1)y$, for some words $z_0, z_1, y \in A^*$, with $|z_0| = |z_1|$ and $|y| \in \{1,2\}$. After cancelling the e's and any common letters from z_0 and z_1, we get that $P(a\phi(z_0))$ and $P(\phi(z_1)y)$ differ in only one component (by only one), and z_0, z_1 have different letters (otherwise the letters would cancel each other). It is easy to see that this is impossible.

Let us now assume that the last position of v is a hole. If v would contain any e's, then we would get a contradiction with the fact that $|J_2| < 2$. Moreover, u does not contain any e's, since otherwise the hole and the e would cancel each other and we would get that the original word is not abelian square-free. Hence, there exist words $x, z \in A^*$ with $|x| < 102$, such that $x\phi(z)\diamond$ is an abelian square.

By Lemma 1, this is impossible. If v has a hole in any other position, then v also contains an e. Again we get that u contains an e, and so, if an abelian square exists, it would be of one of the forms $e\phi(z)y\diamond e$ or $ea\phi(z)y\diamond e$, for some words y, z with $|y| < 102$. After cancelling the e's, this is also impossible by Lemma 1.

Now let us consider the case when a position in u, other than the last one, is a hole. If $|J_1| = 1$, since u contains the $\diamond$ and an e, then v also contains an e. Hence, we have that either $\diamond ex\phi(z)e$ or $\diamond ex\phi(z)ea$, for some words x, z with $|x| < 102$, are abelian squares, which is a contradiction by Lemma 1. If $|J_1| = 3$, then the only possibilities are that either $ae\phi(z_0)\diamond e\phi(z_1)eaebXeceae\phi(z_2)y$ or $e\phi(z_0)\diamond e\phi(z_1)eaebXeceae\phi(z_2)y$ are abelian squares. Since, the hole can be taken to one end and the e's and the common images of ϕ cancel, we get that either $\diamond a\phi(z)y$ or $\diamond\phi(z)y$ are abelian squares, for some y with $|y| < 102$. According to Lemma 1 no such factors preceded by the hole would create an abelian square. If $|J_1| = 2$, then the case when X' comes after the hole in u is impossible, since then the length of v would be greater than that of u. If X' comes before the hole in u, then u contains one more e than the suffix of X' from u.

We reach a contradiction with Lemma 2, hence, all symbols in uv correspond to letters in $A \cup \{e\}$. The conclusion follows as in the proof of Theorem 2. □

5 The Distinct Finite Case

Finite abelian square-free words are difficult to characterize and to build without the aid of a computer. This is due to the fact that they have very little structure. However, there are a few special constructions, such as Zimin words, that have been investigated. In this section, we show that the replacement of letters with holes in these words result in partial words that are not abelian square-free.

Zimin words were introduced in [23] in the context of blocking sets. Due to their construction, Zimin words are not only abelian square-free, but also maximal abelian square-free in the sense that any addition of letters, from the alphabet they are defined on, to their left or right introduces an abelian square.

Definition 2. [23] *Let $\{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet. The Zimin words z_i are defined by $z_0 = a_0$ for $i = 0$, and $z_i = z_{i-1}a_iz_{i-1}$ for $1 \leq i < k$.*

Note that $|z_i| = 2^{i+1} - 1$ and $P(z_i) = \langle 2^i, 2^{i-1}, \ldots, 2, 1\rangle$ for all $i = 0, \ldots, k-1$.

Proposition 3. *Let $\{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet. For $1 < i < k$, the replacement of any letter in z_i with a hole yields a word with an abelian square.*

Proof. The replacement of any letter in an odd position yields an abelian square factor compatible with $abab$ for some letters a, b. For an even position, the factor is of one of the forms $\diamond bacab$, $ab\diamond cab$, $bac\diamond ba$, $bacab\diamond$. □

In [7], Cummings and Mays introduced a modified construction, which they named a one-sided Zimin construction. The resulting words are much shorter than Zimin words.

Definition 3. [7] *Let $\{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet. Left Zimin words y_i are defined recursively as follows: For $i = 0$, $y_0 = a_0$. For $i = 1, \ldots, k-1$, $y_i = y_{i-1}a_i z_{\lfloor \frac{i-1}{2} \rfloor}$, where $z_{\lfloor \frac{i-1}{2} \rfloor}$ is a Zimin word over $\{a_0, a_2, \ldots, a_{i-1}\}$ whenever i is odd and $\{a_1, a_3, \ldots, a_{i-1}\}$ whenever i is even. Right Zimin words can be defined similarly.*

For example, $y_4 = abacbdacaebdb$ and $y_5 = abacbdacaebdbfacaeaca$.

Note that left and right Zimin words are symmetric, and both one-sided constructions have Parikh vector $P(y_i) = \langle 2^{\lfloor \frac{i+1}{2} \rfloor}, 2^{\lfloor \frac{i}{2} \rfloor}, \ldots, 4, 2, 2, 1 \rangle$. Furthermore, y_i is a left maximal abelian square-free word over the alphabet $\{a_0, a_1, \ldots, a_i\}$, for each $i = 0, \ldots, k-1$.

Proposition 4. *Let $\{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet. For each $5 \leq i < k$, the replacement of any letter in y_i with a hole results in a word containing an abelian square.*

Proof. We prove the result by induction on k. For $k = 6$, we find by exhaustive search that no hole can replace any letter of y_5 without creating an abelian square. Assuming that the result is true for $y_5, \ldots, y_{k-1}$, consider $y_k = y_{k-1}a_k z_{\lfloor \frac{k-1}{2} \rfloor}$, where $z_{\lfloor \frac{k-1}{2} \rfloor}$ is a Zimin word. By Proposition 3, it is not possible to place holes in $z_{\lfloor \frac{k-1}{2} \rfloor}$ while remaining abelian square-free. Replacing a_k with a hole yields $\diamond z_{\lfloor \frac{k-1}{2} \rfloor}$, which is an abelian square since $z_{\lfloor \frac{k-1}{2} \rfloor}$ is a maximal abelian square-free word. And by the inductive hypothesis, no hole can replace a letter in y_{k-1} without the resulting word having abelian square factors. □

In [18], Korn gives a construction that provides shorter maximal abelian square-free words. The words' construction is very different from the variations on Zimin words.

Definition 4. [18] *Let $\{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet, where $k \geq 4$. The words v_i are defined recursively by $v_0 = a_2a_1$ for $i = 0$, and $v_i = v_{i-1}a_{i+2}a_{i+1}$ for $1 \leq i \leq k-3$. Then $w_{k-1} = a_0ua_1ua_0v_{k-3}a_0ua_{k-1}ua_0$, where $u = a_2 \cdots a_{k-2}$.*

For example, $w_4 = acdbcdacbdcedacdecda$.

Proposition 5. *Let $A = \{a_0, \ldots, a_{k-1}\}$ be a k-letter alphabet, where $k \geq 4$, and $w_{k-1} \in A^*$ be constructed according to Definition 4. The replacement of any letter in w_{k-1} with a hole results in a word containing an abelian square.*

Proof. After replacing the first or last letter with a hole, v_{k-3} remains abelian square-free. Note that every letter in v_{k-3}, with the exception of a_1 and a_{k-1}, occurs exactly twice. Moreover, if a hole replaces any letter in v_{k-3}, at a position other than the first or the last one, then we would get a factor of either the form $a_la_{l-1}\diamond a_l$ or $a_l\diamond a_{l+1}a_l$ for some l. Note that both these partial words represent abelian squares.

It is not possible to replace letters with holes in the subword $v_{k-3}[1..|v_{k-3}|-1)$ of w_{k-1} while keeping abelian square-freeness. Replacing the first (last) letter of v_{k-3} with a hole yields the abelian square $a_0ua_1ua_0\diamond$ ($\diamond a_0ua_{k-1}ua_0$). Consider now the subword $a_0ua_1ua_0$ of w_{k-1} (the proof is similar for the subword

$a_0ua_{k-1}ua_0$). Clearly, replacing a_0 or a_1 with a hole yields an abelian square. Note that the equality $2|u| + 2 = |v_{k-3}|$ holds. When a hole replaces the letter at position j in any of the u's, consider the factor $a_0ua_1ua_0v_{k-3}[0..2j+4)$. Since $u[0..j) = a_2 \cdots a_{j+1}$ and $v_{k-3}[0..2j] = a_2a_1a_3a_2 \cdots a_{j-2}a_ja_{j-1}a_{j+1}a_ja_{j+2}$, we have that $a_0u[0..j)u[j..|u|)a_1u[0..j)$ has an extra occurrence of a_{j+1} compared to $u[j..|u|)a_0v_{k-3}[0..2j]$, while the second factor has an extra occurrence of a_{j+2} compared to the first factor. Since an a_{j+1} from the first factor is replaced by a $\diamond$, this yields an abelian square, with the $\diamond$ corresponding to a_{j+2} in the prefix of v_{k-3}. $\square$

6 Conclusion

As a possible topic for future work, we propose the study of avoidance of abelian powers greater than two. From [8], we know that over a binary alphabet, we can construct an infinite word that avoids 4-powers. In the context of partial words, which abelian powers can be avoided over a binary alphabet? In this case, certain repetitions are created, since, if w is an abelian square-free partial word of length n and a is the most common letter of w, then it is easy to see that $|w|_a + |w|_\diamond \leq \lceil \frac{n}{2} \rceil$.

Another interesting topic is to replace letters with holes in arbitrary positions of a word, without changing the word's properties of abelian repetition-freeness. Note that introducing holes two positions apart would create an abelian square, $\diamond ab \diamond$. Moreover, note that even if the word is defined in such a way that each two occurrences of a letter are four symbols apart, the resulting word could contain an abelian square. For example, if we consider

$$a_1a_2a_3a_4\underline{a_0}a_5a_6a_7a_8\underline{a_0}a_9a_{10}a_{11}a_{12}$$

in order to avoid abelian squares, no two a_i's are a_j's, no two a_j's are a_k's, and all a_i's, a_j's, and a_k's are different, for $0 < i \leq 4 < j \leq 8 < k \leq 12$. Also, if we allow for holes to be three positions apart, only one of the a_k's equals an a_j.

Also, investigating the number of abelian square-free partial words over an alphabet of size five would be interesting. This number has been studied in [6] and [17] for full words over an alphabet of size four.

References

1. Berstel, J.: Axel Thue's work on repetitions in words. In: Leroux, P., Reutenauer, C. (eds.) Invited Lecture at the 4th Conference on Formal Power Series and Algebraic Combinatorics, pp. 65–80 (1992)
2. Berstel, J., Boasson, L.: Partial words and a theorem of Fine and Wilf. Theoretical Computer Science 218, 135–141 (1999)
3. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press (2008)

4. Blanchet-Sadri, F., Mercaş, R., Rashin, A., Willett, E.: An answer to a conjecture on overlaps in partial words using periodicity algorithms. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 188–199. Springer, Heidelberg (2009)
5. Blanchet-Sadri, F., Mercaş, R., Scott, G.: A generalization of Thue freeness for partial words. Theoretical Computer Science 410(8-10), 793–800 (2009)
6. Carpi, A.: On the number of abelian square-free words on four letters. Discrete Applied Mathematics 81(1-3), 155–167 (1998)
7. Cummings, L.J., Mays, M.: A one-sided Zimin construction. The Electronic Journal of Combinatorics 8 (2001)
8. Dekking, F.: Strongly non-repetitive sequences and progression-free sets. Journal of Combinatorial Theory 27(2), 181–185 (1979)
9. Erdös, P.: Some unsolved problems. Magyar Tudományos Akadémia Matematikai Kutató Intézete 6, 221–254 (1961)
10. Evdokimov, A.: Strongly asymmetric sequences generated by a finite number of symbols. Doklady Akademii Nauk SSSR 179, 1268–1271 (1968) (in Russian); English translation in Soviet mathematics - Doklady 9, 536–539 (1968)
11. Evdokimov, A.: The existence of a basis that generates 7-valued iteration-free sequences. Diskretnyĭ Analiz 18, 25–30 (1971)
12. Halava, V., Harju, T., Kärki, T.: Square-free partial words. Information Processing Letters 108(5), 290–292 (2008)
13. Halava, V., Harju, T., Kärki, T., Séébold, P.: Overlap-freeness in infinite partial words. Theoretical Computer Science 410(8-10), 943–948 (2009)
14. Justin, J.: Characterization of the repetitive commutative semigroups. Journal of Algebra 21, 87–90 (1972)
15. Keränen, V.: Abelian squares are avoidable on 4 letters. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 41–52. Springer, Heidelberg (1992)
16. Keränen, V.: New abelian square-free DT0L-languages over 4 letters. In: Proceedings of the Fifth International Arctic Seminar, Murmansk, Russia. Murmansk State Pedagogical Institute (2002)
17. Keränen, V.: A powerful abelian square-free substitution over 4 letters. Theoretical Computer Science 410(38-40), 3893–3900 (2009)
18. Korn, M.: Maximal abelian square-free words of short length. Journal of Combinatorial Theory, Series A 102, 207–211 (2003)
19. Manea, F., Mercaş, R.: Freeness of partial words. Theoretical Computer Science 389(1-2), 265–277 (2007)
20. Pleasants, P.: Non repetitive sequences. Proceedings of the Cambridge Philosophical Society 68, 267–274 (1970)
21. Thue, A.: Über unendliche Zeichenreihen. Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7, 1–22 (1906); Nagell, T. (ed.), Reprinted in Selected Mathematical Papers of Axel Thue, Universitetsforlaget, Oslo, Norway, pp. 139–158 (1977)
22. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 1, 1–67 (1912); Nagell, T. (ed.) Reprinted in Selected Mathematical Papers of Axel Thue, Universitetsforlaget, Oslo, Norway, pp. 413–478 (1977)
23. Zimin, A.I.: Blocking sets of terms. Mathematics of the USSR Sbornik 47, 353–364 (1984)

Avoidable Binary Patterns in Partial Words[⋆]

Francine Blanchet-Sadri[1], Robert Mercaş[2,⋆⋆], Sean Simmons[3], and Eric Weissenstein[4]

[1] Department of Computer Science, University of North Carolina, P.O. Box 26170, Greensboro, NC 27402–6170, USA
blanchet@uncg.edu

[2] GRLMC, Departament de Filologies Romàniques, Universitat Rovira i Virgili, Av. Catalunya 35, Tarragona, 43002, Spain
robertmercas@gmail.com

[3] Department of Mathematics, The University of Texas at Austin, 2515 Speedway Rm 8, Austin, TX 78712–0233, USA

[4] Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Amos Eaton 301, 110 8th Street, Troy, NY 12180, USA

Abstract. The problem of classifying all the avoidable binary patterns in words has been completely solved (see Chapter 3 of M. Lothaire, *Algebraic Combinatorics on Words*, Cambridge University Press, 2002). Partial words represent sequences that may have some undefined positions called holes. In this paper, we show that, if we do not substitute any variable of the pattern by a trivial partial word consisting of only one hole, the avoidability index of the pattern remains the same as in the full word case.

1 Introduction

A *pattern* p is a word over an alphabet E of *variables*, denoted by $\alpha, \beta, \gamma, \ldots$, and the associated set is built by replacing p's variables with non-empty words over a finite alphabet A so that the occurrences of the same variable be replaced with the same word.

The concept of *unavoidable pattern*, see Section 2, was introduced, in the context of full words, by Bean, Ehrenfeucht and McNulty [1] (and by Zimin who used the terminology "blocking sets of terms" [14]). Although they characterized such patterns (in fact, avoidability can be decided using the Zimin algorithm by reduction of patterns), there is no known characterization of the patterns unavoidable over a k-letter alphabet (also called k-unavoidable). An alternative is to find all unavoidable patterns for a fixed alphabet size. The unary patterns, or

[⋆] This material is based upon work supported by the National Science Foundation under Grant No. DMS–0754154. The Department of Defense is also gratefully acknowledged. A World Wide Web server interface has been established at www.uncg.edu/cmp/research/unavoidablesets4 for automated use of the program.

[⋆⋆] Author's work was partially supported by the Research Grant No. 1323 U07 E30 N-2008/InvAct/Bel, G./BJ01 of the University Rovira i Virgili.

powers of a single variable α, were investigated by Thue [12,13]: α is unavoidable, $\alpha\alpha$ is 2-unavoidable but 3-avoidable, and α^m with $m \geq 3$ is 2-avoidable. Schmidt proved that there are only finitely many binary patterns, or patterns over $E = \{\alpha, \beta\}$, that are 2-unavoidable [10,11]. Later on, Roth showed that there are no binary patterns of length six or more that are 2-unavoidable [9]. The classification of unavoidable binary patterns was completed by Cassaigne [4], who showed that $\alpha\alpha\beta\beta\alpha$ is 2-avoidable.

In this paper, our goal is to classify binary patterns with respect to partial word *non-trivial avoidability.* A partial word is a sequence of symbols from a finite alphabet that may have some undefined positions, called holes, and denoted by $\diamond$'s, and a pattern is called non-trivial if none of its variables is substituted by only one hole. Here $\diamond$ is *compatible* with, or matches, every letter of the alphabet. In this context, in order for a pattern p to occur in a partial word, it must be the case that for each variable α of p, all its substituted partial words be pairwise compatible.

The contents of our paper is as follows: In Section 2, we start our investigation of avoidability of binary patterns in partial words. There, we explain that in order to classify all binary patterns with respect to our concept of non-trivial avoidability, we are left with studying five patterns. In Section 4 using iterated morphisms, we construct infinite binary partial words with infinitely many holes that avoid the patterns $\alpha\beta\alpha\beta\alpha$, $\alpha\beta\alpha\beta\beta\alpha$ and $\alpha\alpha\beta\alpha\beta\beta$. In Section 5 using non-iterated morphisms, we construct such words that avoid the patterns $\alpha\alpha\beta\beta\alpha$ and $\alpha\beta\alpha\alpha\beta$. This concludes the fact that all binary patterns 2-avoidable in full words are also non-trivially 2-avoidable in partial words.

We end this section with some preliminaries. For more information regarding concepts on partial words, the reader is referred to [2].

Let A be a non-empty finite set of symbols called an *alphabet.* Each element $a \in A$ is called a *letter.* A *(full) word* over A is a sequence of letters from A. A *partial word* over A is a sequence of symbols from $A_\diamond = A \cup \{\diamond\}$, the alphabet A being augmented with the "hole" symbol $\diamond$ (a full word is a partial word without holes). We denote by $u(i)$ the symbol at position i of a partial word u. The *length* of u is denoted by $|u|$ and represents the number of symbols in u. The *empty word* is the sequence of length zero and is denoted by ε. The set containing all full words (respectively, non-empty full words) over A is denoted by A^* (respectively, A^+), while the set of all partial words (respectively, non-empty partial words) over A is denoted by $A_\diamond^*$ (respectively, $A_\diamond^+$).

If u and v are two partial words of equal length, then u is said to be *contained in* v, denoted $u \subset v$, if $u(i) = v(i)$ for all i such that $u(i) \in A$. Partial words u and v are *compatible*, denoted $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$. If u and v are non-empty compatible partial words, then uv is called a *square.*

A partial word u is a *factor* of a partial word v if there exist x, y such that $v = xuy$ (the factor u is *proper* if $u \neq \varepsilon$ and $u \neq v$). We say that u is a *prefix* of v if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$.

2 Avoidability on Partial Words

Let E be a non-empty finite set of symbols, distinct from A, whose elements are denoted by α, β, γ, etc. Symbols in E are called *variables*, and words in E^* are called *patterns*. For the remaining of this paper, we only consider binary alphabets and patterns, hence we can fix $A = \{a, b\}$ and $E = \{\alpha, \beta\}$. Moreover, we define $\overline{a} = b$ and $\overline{b} = a$, and similarly $\overline{\alpha} = \beta$ and $\overline{\beta} = \alpha$.

The *pattern language*, over A, associated with a pattern $p \in E^*$, denoted by $p(A_\diamond^+)$, is the subset of $A_\diamond^*$ containing all partial words compatible with $\varphi(p)$, where φ is any non-erasing morphism from E^* to A^*. A partial word $w \in A_\diamond^*$ *meets* the pattern p (or p *occurs* in w) if for some factorization $w = xuy$, we have $u \in p(A_\diamond^+)$. Otherwise, w *avoids* p.

To be more precise, let $p = \alpha_0 \cdots \alpha_m$, where $\alpha_i \in E$ for $i = 0, \ldots, m$. Define an *occurrence* of p in a partial word w as a factor $u_0 \cdots u_m$ of w, where for all $i, j \in \{0, \ldots, m\}$, if $\alpha_i = \alpha_j$, then $u_i \uparrow u_j$. We call such an occurrence *non-trivial* if $u_i \neq \diamond$, for all $i \in \{0, \ldots, m\}$. We call a word *non-trivially p-free* if it contains no non-trivial occurrences of p. Note that these definitions also apply to (one-sided) infinite partial words w over A which are functions from $\mathbb{N}$ to $A_\diamond$.

Considering the pattern $p = \alpha\beta\beta\alpha$, the language associated with p over the alphabet $\{a, b\}$ is $p(\{a, b, \diamond\}^+) = \{u_1 v_1 v_2 u_2 \mid u_1, u_2, v_1, v_2 \in \{a, b, \diamond\}^+$ such that $u_1 \uparrow u_2$ and $v_1 \uparrow v_2\}$. The partial word $ab\diamond ba\diamond bba$ meets p (take $\varphi(\alpha) = bb$ and $\varphi(\beta) = a$), while the word $\diamond babbbaaab\diamond$ avoids p.

Let p and p' be two patterns. If p' meets p, then p *divides* p', which we denote by $p \mid p'$. For example, $\alpha\alpha \nmid \alpha\beta\alpha$ but $\alpha\alpha \mid \alpha\beta\alpha\beta$.

A pattern $p \in E^*$ is *k-avoidable* if there are infinitely many partial words in $A_\diamond^*$ that avoid p, where A is any alphabet of size k. On the other hand, if every long enough partial word in $A_\diamond^*$ meets p, then p is *k-unavoidable* (it is also called unavoidable over A). The *avoidability index* of p is the smallest integer k such that p is k-avoidable, or is ∞ if p is unavoidable.

Remark 1. Let $p, p' \in E$ be such that p divides p'. If an infinite partial word avoids p, then it also avoids p'.

In the context of full words all binary patterns' avoidability index have been characterized [7]. Since a full word is a partial word without holes, the avoidability index of a binary pattern in full words is not greater than the avoidability index of that pattern in partial words. Thus, all unavoidable binary patterns in full words have avoidability index ∞ in partial words as well.

In [3], it was shown that there exist infinitely many partial words with infinitely many holes over a 3-letter alphabet that non-trivially avoid $\alpha\alpha$, and so the avoidability index of $\alpha\alpha$ in partial words is 3. Since in full words all binary patterns with avoidability index 3 are divisible by $\alpha\alpha$, using Remark 1 we conclude that all 3-avoidable binary patterns in full words also have avoidability index 3 in the context of partial words.

Thus, according to [7, Theorem 3.3.3], if we can find the avoidability index of $\alpha\alpha\alpha$, $\alpha\beta\alpha\beta\alpha$, $\alpha\beta\alpha\beta\beta\alpha$, $\alpha\alpha\beta\alpha\beta\beta$, $\alpha\beta\alpha\alpha\beta$ and $\alpha\alpha\beta\beta\alpha$, then we will have completed the classification of the binary patterns in terms of avoidability in partial words.

First let us recall that in [8], the case of patterns of the form α^m, $m \geq 3$, was considered, the avoidability index in partial words being 2. Furthermore, in [3,5] it was shown that the pattern $\alpha\beta\alpha\beta\alpha$ is trivially 2-unavoidable, but it is 3-avoidable in partial words.

In this paper, our main result is the following.

Theorem 1. *With respect to non-trivial avoidability in partial words, the avoidability index of a binary pattern is the same as in the full word case.*

3 Binary Patterns 2-Avoidable by Iterated Morphisms

Let us recall the iterative Thue-Morse morphism ϕ such that $\phi(a) = ab$ and $\phi(b) = ba$. It is well known that $\phi^\omega(a)$ avoids $\alpha\beta\alpha\beta\alpha$ [6].

Proposition 1. *Over a binary alphabet there exist infinitely many infinite partial words, containing exactly one hole, that non-trivially avoid $\alpha\beta\alpha\beta\alpha$.*

Proof. Let $p = \alpha\beta\alpha\beta\alpha$, and t be the fixed point of the Thue-Morse morphism. We show that there exist infinitely many positions in t in which one can replace the letter at that position with a hole and obtain a new word t' that is still non-trivially p-free. Also, since all factors of the infinite Thue-Morse word t (powers of ϕ) avoid p, it follows that any occurrence of p in t' must contain the hole.

Let $x_1, x_2, x_3 \subset x$ and $y_1, y_2 \subset y$, for some partial words $x_1, x_2, x_3, x, y_1, y_2, y$ such that $|x|, |y| \geq 1$. We start by proving that there does not exist a non-trivial occurrence of p, $x_1y_1x_2y_2x_3$, in t' such that $|x| \geq 8$ or $|y| \geq 8$. We proceed by contradiction. We analyze several cases based on the possible positions of the hole.

Assume that the hole is in x_1. Note that this case is symmetrical to when the hole is in x_3 (we are implicitly using the fact that if w is a factor of the Thue-Morse word t, then so is $\text{rev}(w)$). Since t is overlap-free, it follows that the only possibility is to have in t a factor of the form $x'cx''yx'\overline{c}x''yx'\overline{c}x''$, with $c \in \{a, b\}$, and $x_1 = x'\diamond x''$, $x_2 = x_3 = x'\overline{c}x'' = x$ and $y_1, y_2 = y$, for some words $x', x'' \in \{a, b\}^*$ with $|x'x''| \geq 7$ or $|y| > 7$ (moreover, x' is non-empty since, otherwise, t would contain the factor $x''y\overline{c}x''y\overline{c}x''$ which is impossible since t is p-free). Looking at the symbols that precede and follow c in x_1 and $\overline{c}$ in x_2, we get that if $|x'x''| \geq 7$ either $\overline{c}c\overline{c}c\overline{c}$ is a factor of $x_2 = x\overline{c}y$ when c is preceded by c in xcy, or $c\overline{c}c\overline{c}c$ is a factor of $x_1 = xcy$ when c is preceded by $\overline{c}$ in xcy, and if $|y| > 7$ either $c\overline{c}c\overline{c}c$ is a factor of $x_1 = xcy$ when c is preceded by c in x_1, or $\overline{c}c\overline{c}c\overline{c}$ is a factor of $x_2 = x\overline{c}y$ when c is preceded by $\overline{c}$ in x_1. All cases lead to contradiction with the fact that t is overlap-free.

Let us illustrate by an example how this works. Let us consider the case when c is preceded by a c, $|x'| = 1$ and $|y| > 7$. We look at the factors x_1y_1 and x_2y_2 that differ at only one position. We have that y_1 starts with $\overline{c}$, such that ccc is not a prefix of our factor. It follows that $x''y_2$ starts with $\overline{c}c$ such that we do not get the cube $\overline{ccc}$ in t. But, in x_1y_1 we have the factor $cc\overline{c}c$, which must be followed by $\overline{c}$. Again looking at the prefix of $x''y_2$, we get $\overline{c}c\overline{c}c$. It follows that $ccx''y$ has as prefix $cc\overline{c}c\overline{c}c$ which contains an overlap, a contradiction with the fact that t is overlap-free.

Assume that the hole is in y_1. This case is symmetrical to when the hole is in y_2. In this case, since t is overlap-free, the only possibility is to have in t a factor of the form $xy'cy''xy'\overline{c}y''x$, with $c \in \{a, b\}$, and $x_1 = x_2 = x_2 = x$, $y_1 = y'\diamond y''$, $y_2 = y'\overline{c}y''$, for some words $y', y'' \in \{a, b\}^*$ with $|y'| + |y''| \geq 1$. Since at least one of y' and y'' is non-empty, and either $|x| > 7$ or $|y'y''| \geq 7$, the proof follows from the previous case.

Finally, assume that the hole is in x_2. Since t is overlap-free, the only possibility is to have in t a factor of the form $x'cx''yx'\overline{c}x''yx'cx''$, with $c \in \{a, b\}$, and $x_1 = x_3 = x'cx''$, $x_2 = x'\overline{c}x''$ and $y_1 = y_2 = y$, for some words $x', x'' \in \{a, b\}^*$ with $|x'| + |x''| \geq 1$. Again, since at least one of x' and x'' is non-empty, and either $|y| > 7$ or $|x'x''| \geq 7$, the proof follows from the first case.

We have thus shown that if $|x| \geq 8$ or $|y| \geq 8$, there are no non-trivial occurrences $x_1y_1x_2y_2x_3$, where $x_1, x_2, x_3 \subset x$ and $y_1, y_2 \subset y$, of p in t', therefore any such occurrence in t' must satisfy $|x_1y_1x_2y_2x_3| < 40$. We now put a hole at Position 47 of $t_7 = \phi^7(a)$:

$$t'_7 = abbabaabbaababbabaababbaabbabaabbaababbaabbabaa\underline{b}abbabaabbaababbab$$
$$aababbaabbabaababbabaabbaababbaabbabaabbaababbabaababbaabbabaab$$

It is not hard to verify (using a computer program) that there are no occurrences of p in t'_7. Moreover we have placed the hole more than 40 positions from either end of t_7. Note that t_7 occurs as a factor of t infinitely often. We can choose any arbitrary occurrence of the factor t_7 in t, and replace it with t'_7 in order to obtain an infinite word with one hole that non-trivially avoids p. □

Theorem 2. *Over a binary alphabet there exist infinitely many partial words, containing infinitely many holes, that non-trivially avoid the pattern $\alpha\beta\alpha\beta\alpha$.*

Proof. Let us give a sketch of the proof. We can obtain such words t', by replacing arbitrarily many occurrences of t_7 in t with the factor t'_7 obtained in the manner discussed above.

In order to prove the theorem we assume there exists some occurrence of p in t' having the form $x_1y_1x_2y_2x_3$, such that $x_1, x_2, x_3 \subset x$ and $y_1, y_2 \subset y$ for some words $x, y \in \{a, b\}^*$, and having the minimum number of holes possible. Because of the way we insert holes, there are at least 127 letters in between any two holes, hence, $|xy| \geq 43$. Moreover, since the number of holes is minimum, one of the inequalities $x_1 = x_2 = x_3$ or $y_1 = y_2$ must fail. This implies that the position corresponding to the hole in one of the other factors must be an a. By looking at the factors preceding and following the hole, respectively the ones preceding and following the a in the corresponding factor, we get into contradictions either regarding valid factors of w or the minimality of the number of holes. □

Let us move on and take ν to be the morphism that maps a to aab and b to bba. Define the sequence produced by ν as $t_0 = a$, and $t_n = \nu(t_{n-1})$. Recall that ν avoids $\alpha\beta\alpha\beta\beta\alpha$ and $\alpha\alpha\beta\alpha\beta\beta$ [7].

Proposition 2. *For any $n \geq 0$, $t_{n+1} = t_n t_n \overline{t_n}$, where t_i is the ith iteration of the sequence produced by ν.*

Proof. We proceed by induction. Note that $t_1 = aab = t_0 t_0 \overline{t_0}$. Assume $t_n = t_{n-1}t_{n-1}\overline{t_{n-1}}$, for some integer $n > 0$. Thus, $t_{n+1} = \nu(t_n) = \nu(t_{n-1}t_{n-1}\overline{t_{n-1}})$, and so $t_{n+1} = \nu(t_{n-1})\nu(t_{n-1})\nu(\overline{t_{n-1}}) = t_n t_n \overline{t_n}$. □

Proposition 3. *Over a binary alphabet there exist infinitely many infinite partial words, containing exactly one hole, that avoid the pattern $\alpha\beta\alpha\beta\beta\alpha$.*

Proof. Let $p = \alpha\beta\alpha\beta\beta\alpha$, and let t be the fixed point of the morphism ν. We show that there exist infinitely many positions in t in which one can replace the letter at that position with a hole and obtain a new word t' that is still non-trivially p-free. Note that since t avoids p, it follows that for p to occur in t' it must contain the hole.

First let us replace Position 58 of t_5 by a hole:

$$t_5 = aabaabbbaaabaabbbabbabbaaabaabaabbbaaabaabbbabbabbaaabbbab\underline{b}aaabbbab$$
$$baaabaabaabbbaaabaabbbaaabaabbbabbabbaaabaabaabbbaaabaabbbabbabbaa$$
$$abbbabbaaabbbabbaaabaabaabbbabbabbaaabbbabbaaabaabaabbbabbabbaaabbb$$
$$abbaaabaabaabbbaaabaabbbaaabaabbbabbabbaaab$$

It is easy to verify with a computer program that the resulting word has no occurrences of p with $|\alpha|, |\beta| < 9$, since the hole is more than 58 positions from either end of t_5.

Let $x_1, x_2, x_3 \subset u$ and $y_1, y_2, y_3 \subset z$ for some partial words x, y such that $|x|, |y| \geq 1$. We prove that there does not exist an occurrence of the factor $x_1y_1x_2y_2y_3x_3$ in t' such that $|x| \geq 9$ or $|y| \geq 9$.

Assume that the hole is in x_1 (the cases when the hole is in y_1, x_2 or y_2 are similar). Since $|x_1y_1| > 9$, it follows that the hole is either preceded by *bab* or followed by *aaa*. Because of this, it must be the case that the a from x_2y_2 corresponding to the hole (if there were an occurrence of b corresponding to the hole then t would contain p which is a contradiction), it is either preceded by *bab* or followed by *aaa*. We get that t either contains the factor *baba* or the factor *aaaa*, a contradiction with the construction of t.

If the hole is in x_3 (the case when the hole is in y_3 is similar), since $|y_3x_3| > 9$, it follows that the hole is either preceded by *bab* or followed by *aaa*. Comparing it to the a from y_1x_2 corresponding to the hole, we have that the a is either preceded by *bab* or followed by *aaa*. We get once more that t either contains the factor *baba* or the factor *aaaa*, a contradiction with the construction of t.

Note that t_5 occurs as a factor of t infinitely often. We can choose any arbitrary occurrence of the factor t_5 in t, and place a hole at Position 58 to obtain an infinite word with one hole that avoids p. □

Remark 2. If uu is a factor of t, the fixed point of the morphism ν, for some word u of length $|u| > 3$, it must be the case that $|u| = 0 \bmod 3$. Moreover, for all different occurrences of the same factor v of length $|v| > 3$, there exist unique words x, y, z such that $v = x\nu(y)z$, with $|x|, |z| < 3$. In other words, all occurrences of the same factor start at the same position of an iteration of ν.

Theorem 3. *Over a binary alphabet there exist infinitely many partial words, containing infinitely many holes, that avoid the pattern $\alpha\beta\alpha\beta\beta\alpha$.*

Proof. Let us denote by t'_5 the word obtained by replacing the letter a at Position 58 by a $\diamond$ in t_5, and by t' the word where infinitely many occurrences of the factor t_5, that start at an even position in t, have been replaced by t'_5.

Assume, to get a contradiction, that the pattern occurs somewhere in t'. It must be the case that there exists a factor $x_1y_1x_2y_2y_3x_3$ that contains h holes, (the case $h = 1$ is proved in Proposition 3), with all x_i's and all y_i's pairwise compatible for all $i \in \{1, 2, 3\}$, and no occurrence of the pattern with less than h holes exists.

It is obvious that $h > 1$ according to the previous proposition. If there exists a hole in x_i and $|x_i| > 4$, for $0 < i \leq 3$, then there exists x_j, with $j \neq i$, that has a factor that is compatible with a word from $\{\underline{a}aaab, b\underline{a}aaa, ab\underline{a}aa, bab\underline{a}a, bbab\underline{a}\}$ (note that the underlined letter is the one that corresponds to the hole in x_i), and if x_j has a hole, then the hole does not correspond to the hole in x_i. Note that it is impossible to have a hole at another position than the underlined one, in any of the previously mentioned factors. We conclude that x_j has no holes. But, in this case we would have that t contains one of the factors $aaaa$, $abaaa$ or $baba$, which is a contradiction. The same proof works for y_i, where $0 < i \leq 3$.

Thus, either $|\alpha| \leq 4$ and y_i contains no holes for $0 < i \leq 3$, or $|\beta| \leq 4$ and x_i contains no holes for $0 < i \leq 3$ (otherwise we would have that $|x_1y_1x_2y_2y_3x_3| \leq 24$ contains more than two holes, which is a contradiction since between each two holes there are at least 72 symbols according to our construction).

Let us first assume that the hole is in α. If x_1 contains the hole then, since $|x_1| \leq 4$ and y_1 contains no hole, looking at the factor following $\diamond$ we conclude that the corresponding position in x_2 must also contain a hole. Now, if x_2 contains the hole then, it follows from the previous observation that x_1 has to contain a hole, and moreover, since y_1 and y_3 contain no holes, looking at the factor preceding the hole, we get that x_3 has a hole at the corresponding position. In the case x_3 has a hole, according to the previous observation, it must be the case that x_2 has a hole. We conclude that if x_i has a hole, then $x_i = x_j$, for all $i, j \in \{1, 2, 3\}$. Hence, there exists an occurrence of the pattern having no holes, a contradiction.

Since the case when the hole is in β is similar, we conclude that t' does not contain any occurrence of the pattern $\alpha\beta\alpha\beta\beta\alpha$. $\square$

Proposition 4. *Over a binary alphabet there exist infinitely many infinite partial words, containing exactly one hole, that avoid the pattern $\alpha\alpha\beta\alpha\beta\beta$.*

The word is obtained by placing a hole at Position 57 of t_5:

$$
\begin{aligned}
t_5 =\, & aabaabbbaaabaabbbabbabbaaabaabaabbbaaabaabbbabbabbaaabbba\underline{b}baaabbbab \\
& baaabaabaabbbaaabaabbbaaabaabbbabbabbaaabaabaabbbaaabaabbbabbabbaa
\end{aligned}
$$

$$abbbabbaaabbbabbaaabaabaabbbabbabbaaabbbabbaaabaabaabbbabbabbaaabbb$$
$$abbaaabaabaabbbaaabaabbbaaabaabbbabbabbaaab$$

The proof is similar to the one for the pattern $\alpha\beta\alpha\beta\beta\alpha$.

Theorem 4. *Over a binary alphabet there exist infinitely many partial words, containing infinitely many holes, that avoid the pattern $\alpha\alpha\beta\alpha\beta\beta$.*

The proof of this result is similar to the one for the pattern $\alpha\beta\alpha\beta\beta\alpha$.

4 Binary Patterns 2-Avoidable by Non-iterated Morphisms

Let us now look at the pattern $\alpha\alpha\beta\beta\alpha$. Let $A = \{a, b\}$ and $A' = A \cup \{c\}$, and let $\psi : A'^* \to A'^*$ be the morphism defined by $\psi(a) = abc$, $\psi(b) = ac$ and $\psi(c) = b$. We know that $\psi^\omega(a)$ avoids $\alpha\alpha$, in other words it is square-free [7].

Furthermore, define the morphism $\chi : A'^* \to A^*$ such that $\chi(a) = aa$, $\chi(b) = aba$, and $\chi(c) = abbb$. If $w = \chi(\psi^\omega(a))$, then we know from [4] that w does not contain any occurrence of $\alpha\alpha\beta\beta\alpha$. Moreover, denote by χ_4 the application of χ to the fourth iteration of ψ. Since $\psi^4(a)$ occurs infinitely often as a factor of $\psi^\omega(a)$, it follows that $\chi_4(a) = \chi(\psi^4(a))$ occurs infinitely often as a factor of w. Hence, we can write $w = w_0\chi_4 w_1\chi_4 w_2\chi_4 \cdots$, for some words w_i with $|w_i| > 1$, for all i. Now, let us replace Position 23 of χ_4 by a $\diamond$ and denote the new partial word by χ_4':

$$aaabaabbbaaabbbabaaaaba\underline{a}bbbabaaaabbbaaabaabbbaaabbbabaaaabbbaaabaabbbaba$$

Lemma 1. *Let u_1u_2 denote a factor of $w' = w_0\chi_4' w_1\chi_4' w_2\chi_4' \cdots$ that was obtained by inserting holes in v_1v_2, a factor of w with $u_1 \subset v_1$ and $u_2 \subset v_2$. If $u_1 \uparrow u_2$, but $v_1 \neq v_2$, then $|u_1| \leq 4$, more specifically, either u_1 or u_2 is in $\{\diamond, \diamond b, \diamond bb, a\diamond, a\diamond b, ba\diamond, a\diamond bb\}$.*

Proof. Obviously, if $u_1 \uparrow u_2$ but $v_1 \neq v_2$, then a hole appears in u_1 and there is no hole at the corresponding position in u_2, or vice versa. Without loss of generality we can assume that the hole appears in u_1. Assume that $u_1 \notin \{\diamond, \diamond b, \diamond bb, a\diamond, a\diamond b, a\diamond bb, ba\diamond\}$. It follows that u_1 has as a factor $\diamond bbb$, $aba\diamond$ or $ba\diamond b$. Moreover, note that the only time bbb appears in w' is in the factors $abbba$ and $\diamond bbba$. Similarly, the only time aba appears as a factor of w' is as a factor of $abaa$ and $aba\diamond$, and the only time a word x compatible with $ba\diamond b$ appears in w' is when $x = ba\diamond b$ or $x = baab$. We see that in all of these cases the corresponding factor in u_2 must be $abbba$, $abaa$ or $baab$, a contradiction since we always have $v_1 = v_2$. □

Lemma 2. *There exists no factor uu of $w = \chi(\psi^\omega(a))$, such that either $aaab$ or aba is a prefix of u.*

Proof. Assume there exists an u with prefix $aaab$ so that $uu = w(i) \cdots w(i + 2l - 1)$, for some integers i, l with $l > 3$. Note that $aaab$ only appears as a prefix of $\chi(x)$, for some word $x \in \{a, b, c\}^+$. Moreover, since the second u also starts with $aaab$, we have that $u = \chi(x)$. Hence, uu is actually $\chi(xx)$, for some word $x \in \{a, b, c\}^+$. It follows that $\phi^\omega(a)$ contains the square xx, which is a contradiction with the nature of the ψ morphism. Similarly, aba only appears as an image of $\chi(b)$. □

Theorem 5. *Over a binary alphabet there exist infinitely many partial words, containing infinitely many holes, that avoid the pattern $\alpha\alpha\beta\beta\alpha$.*

Proof. Let $p = \alpha\alpha\beta\beta\alpha$. To prove this claim, we assume that w' is not p-free and get a contradiction. Let $x_1' x_2' y_1' y_2' x_3'$ be an occurrence of p in w', and denote by $x_1 x_2 y_1 y_2 x_3$ the factor of w in which holes were inserted to get p. Note that if $x_1 = x_2 = x_3$ and $y_1 = y_2$, then we have an occurrence of p in w, which would be a contradiction. Therefore one of the inequalities fails. Also, note that if $x_i \neq x_j$ then either x_i' or x_j' contains a hole, where $i, j \in \{1, 2, 3\}$, while if $y_1 \neq y_2$ then either y_1' or y_2' contains a hole. Moreover, if $x_1 \neq x_2$ or $y_1 \neq y_2$, according to Lemma 1, it must be the case that x_1 or x_2 or, y_1 or y_2 are in $\{\diamond, \diamond b, \diamond bb, a\diamond, a\diamond b, a\diamond bb, ba\diamond\}$. By looking at the factor χ_4', it is easy to check that the only possibilities are for x_1' and y_1' to be in $\{\diamond, \diamond b\}$ and for x_2' and y_2' to be in $\{\diamond, a\diamond, a\diamond b\}$.

If $y_1' = \diamond b$, it is easy to check that x_3' must start with aab. According to Lemma 2, we cannot have that $x_1 = x_2$. It follows, according to Lemma 1, that $|x_1'| \leq 4$, a contradiction with the factor preceding the hole. The proof is identical for the case when $x_1' = \diamond b$. If $y_1' = \diamond$, then x_3' starts with bba. Thus, x_2' starts with bba and x_1' ends in ab or $\diamond b$. It follows that x_2' ends in ab or $\diamond b$, thus, y_1' is preceded by ab, which is a contradiction. If $x_1' = \diamond$, then y_1' and y_2' start with bba and x_3' is a b. From Lemma 1 we get that $y_1 = y_2$. Thus, $y_1 y_2 bb$ is a factor of w. It follows that for some word $x \in \{a, b\}^+$, $y_1 y_2 bb = bb\chi(xc)\chi(xc)$ is a factor of w. We get a contradiction with the fact that $\phi^\omega(a)$ is square-free.

If x_2' or y_2' are in $\{\diamond, a\diamond, a\diamond b\}$ then we get that either y_1' or x_3' are in $\{bbba, bba\}$. A contradiction is reached again with the help of Lemma 1 and the fact that $\phi^\omega(a)$ is square-free.

The final case that needs to be analyzed is when $x_1 = x_2$ and $y_1 = y_2$, and $x_3' \uparrow x_1$ and $x_3 \neq x_1$. Let us denote $x = x_1 = x_2$ and $y = y_1 = y_2$. We get that w' has $xxyyx_3'$ as a factor and there exists at least one hole in x_3' that corresponds to b's in x_1' and x_2'.

If $|x| > 4$ it follows that x has $aba\underline{b}$, $ba\underline{b}b$, $\underline{b}bbb$ as a factor (the underlined b represents the letter corresponding to the hole in x_3'). Since none of these are possible factors of w, we conclude that it is impossible. Hence, it must be the case that $x_3' \in \{\diamond, \diamond b, \diamond bb, a\diamond, a\diamond b, a\diamond bb, ba\diamond\}$. It follows that $xx \in \{b^2, (bb)^2, (bbb)^2, (ab)^2, (abb)2, (abbb)^2, (bab)^2\}$. But, only bb is a possible factor of w. It is easy to check that in this case $|y| > 6$ and we conclude that y has either aaa, aba, $baaa$ or $baba$ as a prefix. In all of these cases, using Lemma 2 we reach a contradiction.

Since all cases lead to contradictions we conclude that $\alpha\alpha\beta\beta\alpha$ is trivially-avoidable over a binary alphabet. □

Finally, let us look at the pattern $\alpha\beta\alpha\alpha\beta$. According to [7, Lemma 3.3.2], $\gamma(\psi^\omega(a))$ avoids $\alpha\beta\alpha\alpha\beta$, where $\gamma : \{a, b, c\}^* \to \{a, b\}^*$ with $\chi(a) = aaa$, $\chi(b) = bbb$, and $\chi(c) = ababab$. Moreover, the only squares that occur in $w = \gamma(\psi^\omega(a))$ are $a^2, b^2, (aa)^2, (ab)^2, (ba)^2, (bb)^2$ and $(baba)^2$. As a last thing, note that $\psi^\omega(a)$ does not contain any of the factors aba or cbc.

Now let us replace Position 84 of $\gamma_5 = \gamma(\psi^5(a))$ by a $\diamond$ and denote the new partial word by γ_5':

$$\begin{aligned}\gamma_5' =\, & aaabbbabababaaaabababbbbaaabbbabababbbbaaaabababaaabbbabababaaaabab\\ & abbbbaaaabababaa\underline{a}bbbabababbbbaaabbbabababaaaabababbbbaaabbbabababbbb\\ & aaaabababaaabbbabababbbbaaabbbabababaaaabababbbbaaaababab\end{aligned}$$

Moreover, let us denote by w' the word obtained from w after the insertion of a hole at Position 84 of an occurrence of γ_5.

Proposition 5. *Over a binary alphabet there exist infinitely many infinite partial words, containing exactly one hole, that non-trivially avoid $\alpha\beta\alpha\alpha\beta$.*

Proof. Let us assume, to get a contradiction, that there exists an occurrence of $p = \alpha\beta\alpha\alpha\beta$ in w', and denote this occurrence by $x_1y_1x_2x_3y_2$, for $x_i, y_j \in \{a, b\}^+$, $0 < i \leq 3$ and $0 < j < 3$. It must be the case that either $x_1 = x_2 = x_3 = x$ or $y_1 = y_2 = y$, for some words x and y, but not both. Note that if the variable containing the hole has length greater than 5 then the corresponding variable has as a factor $baa\underline{b}$, $aa\underline{b}bbb$ or $a\underline{b}bbba$ (the underlined b stands for the hole position in the first variable). Only the last of these words is a valid factor of w. Moreover, note that actually this represents the prefix of the variable since, having an extra symbol in front would give us the factor $aabbbb$ which is not a valid one for w.

If the hole is in one of the α's, note that, since $x_2 \uparrow x_3$ and w contains no squares of length greater than four, it must be the case that the hole is either in x_2 or x_3. That implies that w' has either the factor $abbbbax'ya\diamond bbbax'abbbbax'y$, or $abbbbax'yabbbbax'a\diamond bbbax'y$, where $x = abbbbax'$, for some non-empty word x'. In the first case, this implies that y starts with ab, giving us in w the square $bbbbax'ab\ bbbax'ab$, a contradiction. In the second case, since x' is non-empty and it is preceded by $\diamond bbba$, we conclude that x' starts with $abab$. Looking at the prefix of x_2, we get the factor $abbbbabab$, which is a contradiction with the fact that $\psi^\omega(a)$ does not contain cbc as a factor. If the hole is in β, then $x \in \{a, b, aa, ab, ba, bb, baba\}$. Thus, we get one of the factors $xa\diamond bbbay'xxabbbbay'$ or $xabbbbay'xxa\diamond bbbay'$, where $y = abbbbay'$ for some word y'. Replacing x with any of the possible values gives us a contradiction with either the factor preceding the hole or the construction of w. We conclude that none of the cases are possible.

Hence, it must be that the variable containing the hole has length at most five. If we denote by z the variable containing the hole and by z' one of the variables corresponding to z, $z \uparrow z'$, we get that $z \in \{a\diamond$, $\diamond b$, $aa\diamond$, $a\diamond b$, $\diamond bb$, $aa\diamond b$, $a\diamond bb$, $\diamond bbb$, $aa\diamond bb$, $a\diamond bbb$, $\diamond bbba\}$ and $z' \in \{ab$, bb, aab, abb, bbb, $aabb$, $abbb$,

$bbbb$, $aabbb$, $abbbb$, $bbbba\}$ (z' cannot contain the factor $baab$). By looking at the possible factors preceding and following the holes, and the squares that can be found in w, we conclude that if the hole is in x_1, then $x_1 \in \{a\diamond, \diamond b\}$, if the hole is in x_2, then $x_2 = \diamond b$, and it is impossible to have the hole in x_3.

If $x_1 = \diamond b$, then it follows that y has bb as a prefix, and a contradiction is reached with the fact that $bbbbb$ is a factor of w. If $x_1 = a\diamond$, it follows that y_1 starts with $bbbab$, and so, we get that $ababy_2$ determines in $\psi^\omega(a)$ the factor cbc, which is a contradiction.

If $x_2 = \diamond b$, then w contains the factor $bbababab y' \underline{a} bbbabababy'$ (the underlined letter is the one that we changed into a hole), where $y = abababy'$ for some word y'. It can be checked that y' ends in $aabababaa$, and in order to avoid having the square $acac$ in $\psi^\omega(a)$, it must be the case that y' is always followed by ab. Hence, w has the factor $xyxxy$, where $x = b$ and $y = abababy'ab$, a contradiction.

If the hole is in β, since p has $\alpha\alpha$ as a factor, and the only possible squares in w are $a^2, b^2, (aa)^2, (ab)^2, (ba)^2, (bb)^2$ and $(baba)^2$, by looking at the possible factor preceding and following the hole, we conclude that y_1 cannot contain the hole and the only possibility for y_2 to contain the hole is when $y_2 = \diamond$. But, in this case the occurrence of the pattern is a trivial one, hence, we get a contradiction.

Since all cases lead to contradiction, the conclusion follows. □

Theorem 6. *Over a binary alphabet there exist infinitely many partial words, containing infinitely many holes, that non-trivially avoid the pattern $\alpha\beta\alpha\alpha\beta$.*

Proof. Let us denote by w' the word obtained from w after an infinite number of non-overlapping occurrences of γ_5 starting at an even position have been replaced by γ_5'. Furthermore, let us assume, to get a contradiction, that the pattern $p = \alpha\beta\alpha\alpha\beta$ is unavoidable and denote by $x_1y_1x_2x_3y_2$ an occurrence of p containing $h > 1$ holes, such that no occurrence of the pattern p having less than h holes appears in w'.

Since, according to Proposition 5, $h > 1$ and the distance between every two holes is at least 170, it follows that $|\alpha\beta| > 85$. Thus, there exist $z \in \{x_1, x_2, x_3, y_1, y_2\}$ and a variable $z' \in \{x_1, x_2, x_3, y_1, y_2\}$ distinct from z, with $z \uparrow z'$ and, $z = z_1\diamond z_2$ and $z' = z_1'bz_2'$, for some words z_i, z_i' with $z_i \uparrow z_i'$, for $0 < i < 3$. If $|z_1| > 2$, it follows that z_1' has a suffix compatible with baa. Since the only factor in w' compatible with baa is baa we conclude that $z_1'b$ has $baab$ as a suffix, which is a contradiction with the fact that $baab$ is a valid factor of w. It follows that $|z_1| < 3$. Moreover, if $z \in \{y_1, y_2\}$, since y_1 is preceded by x_1 and y_2 is preceded by x_3, we get $x_1y_1 \uparrow x_3y_3$. If $|\alpha| > 2$ a conclusion similar to the previous one is reached. It follows that $0 < |x_1z_1| < 3$. In this case $|z_2| > 82$ and we get that the hole is followed by $bbbab$. So the prefix of length five of bz_2', $bbbbab$, represents a factor of the image of $\gamma(cbc)$. This is a contradiction since, cbc is not a factor of $\psi^\omega(a)$. Thus, $z \in \{x_1, x_2, x_3\}$.

Note that, if for all $\diamond$'s in x_2 the corresponding position in x_3 is an a or a $\diamond$, or vice versa, then x_2, x_3 are compatible with an element of $\{aa, ab, ba, bb, baba\}$, since these are the only possible squares of length greater than one in w. Since none of these creates a valid factor, we conclude that there exists a $\diamond$ in x_2 such that the corresponding position in x_3 is a b, or vice versa.

Let us assume that $x_2 = z$ and $x_3 = z'$. The other case is similar. It follows that w' has $z_1 \diamond z_2 z'_1 b z'_2$ as a factor, where $|z_1| < 3$. It can be checked that unless $x_2 = \diamond b$ and $x_3 = bb$, then $|z_2| > 5$. If z_2 has $bbbab$ as a factor, since the only factor of w' compatible with it is $bbbab$, we conclude that z'_2 has $bbbab$ as a factor. This implies that the prefix of length six of bz'_2, a factor of w', was determined by $\gamma(cbc)$, with cbc factor of $\psi^{\omega}(a)$, which is a contradiction. It must be the case that $x_2 = \diamond b$, $x_3 = bb$ and $x_1 \in \{\diamond b, bb\}$. If $x_1 = \diamond b$, it follows that y_1 has bb as a prefix. but, since $x_2 x_3 = \diamond bbb$, it follows that y_2 has ab as a prefix, a contradiction. Hence, it must be that $x_1 y_1 x_2 x_3 y_2 = bby_1 \diamond bbby_2$. But since all the holes in y_1 correspond to a's or $\diamond$'s in y_2, and vice versa, it follows that, replacing all holes but the one in x_2 we get an occurrence of the pattern having only one hole. This is a contradiction with Proposition 5. The conclusion follows. $\square$

Since all these patterns prove to have a non-trivial avoidability index 2, the result of Theorem 1 follows.

References

1. Bean, D.R., Ehrenfeucht, A., McNulty, G.: Avoidable patterns in strings of symbols. Pacific Journal of Mathematics 85, 261–294 (1979)
2. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press (2008)
3. Blanchet-Sadri, F., Mercaş, R., Scott, G.: A generalization of Thue freeness for partial words. Theoretical Computer Science 410(8-10), 793–800 (2009)
4. Cassaigne, J.: Unavoidable binary patterns. Acta Informatica 30, 385–395 (1993)
5. Halava, V., Harju, T., Kärki, T., Séébold, P.: Overlap-freeness in infinite partial words. Theoretical Computer Science 410(8-10), 943–948 (2009)
6. Lothaire, M.: Combinatorics on Words. Cambridge University Press, Cambridge (1997)
7. Lothaire, M.: Algebraic Combinatorics on Words. Cambridge University Press, Cambridge (2002)
8. Manea, F., Mercaş, R.: Freeness of partial words. Theoretical Computer Science 389(1-2), 265–277 (2007)
9. Roth, P.: Every binary pattern of length six is avoidable on the two-letter alphabet. Acta Informatica 29(1), 95–107 (1992)
10. Schmidt, U.: Motifs inévitables dans les mots. Rapport LITP, pp. 86–63, Paris VI (1986)
11. Schmidt, U.: Avoidable patterns on two letters. Theoretical Computer Science 63(1), 1–17 (1989)
12. Thue, A.: Über unendliche Zeichenreihen. Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 7, 1–22 (1906); Nagell, T. (ed.) Reprinted in Selected Mathematical Papers of Axel Thue, Universitetsforlaget, Oslo, Norway, pp. 139–158 (1977)
13. Thue, A.: Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen. Norske Vid. Selsk. Skr. I, Mat. Nat. Kl. Christiana 1, 1–67 (1912); Nagell, T. (ed.) Reprinted in Selected Mathematical Papers of Axel Thue, Universitetsforlaget, Oslo, Norway, pp. 413–478 (1977)
14. Zimin, A.I.: Blocking sets of terms. Mathematics of the USSR Sbornik 47, 353–364 (1984)

Equivalence and Inclusion Problem for Strongly Unambiguous Büchi Automata

Nicolas Bousquet[1] and Christof Löding[2]

[1] ENS Chachan, France
nbousque@dptinfo.ens-cachan.fr
[2] RWTH Aachen, Informatik 7, 52056 Aachen, Germany
loeding@cs.rwth-aachen.de

Abstract. We consider the inclusion and equivalence problem for unambiguous Büchi automata. We show that for a strong version of unambiguity introduced by Carton and Michel these two problems are solvable in polynomial time. We generalize this to Büchi automata with a fixed finite degree of ambiguity in the strong sense. We also discuss the problems that arise when considering the decision problems for the standard notion of ambiguity for Büchi automata.

1 Introduction

The model of unambiguous automata is located between deterministic and nondeterministic automata. An unambiguous automaton is a nondeterministic automaton such that each input that is accepted has a unique accepting run. The concept of unambiguity also occurs in other areas of theoretical computer science, for example in complexity theory. The problems solvable in polynomial time by unambiguous (nondeterministic) Turing machines are collected in the subclass UP (Unambiguous Polynomial time) of NP [17].

There are two aspects in the study of unambiguous automata: expressiveness and computational complexity. Concerning expressiveness, because of the well-known equivalence between deterministic and nondeterministic finite automata over finite words and trees, unambiguous automata can recognize all the regular languages over these two domains. For automata over ω-words it is known that deterministic Büchi automata are strictly less expressive than nondeterministic ones [10]. However, Arnold showed that all the ω-regular languages can be recognized by an unambiguous Büchi automaton [2]. For automata over ω-trees, the class of unambiguous automata is not as expressive as the class of full nondeterministic tree automata (with standard acceptance conditions like parity, Rabin or Muller) [12,6].

An interesting subclass of the class of unambiguous Büchi automata is considered by Carton and Michel [7]: Their definition requires that for each input (accepted or not) there is a unique run passing infinitely often through a final state (whether from an initial state or not). Thus, an infinite word is accepted if the initial state is the first state of the path passing infinitely often through a final state. Non-acceptance means that the unique path of this form does not start in an initial state. Carton and Michel show [7] that this restricted class of Büchi automata suffices to capture the class of ω-regular languages. We consider in this paper a slight modification of their definition, and refer to these automata as strongly unambiguous.

The second interesting aspect for unambiguous models is the computational complexity of algorithmic problems. We consider here the equivalence problem (as well as the inclusion problem which turns out to have the same complexity). It is well known that there is a gap in complexity for the equivalence problem between deterministic and nondeterministic automata. The problem can be decided in polynomial time over finite words [15] and finite trees for deterministic automata, whereas the problem is PSPACE-complete over finite words (see Section 10.6 of [1][1]) and EXPTIME-complete over finite trees [8] for nondeterministic automata.

As shown by Stearns and Hunt, the equivalence problem for unambiguous finite automata over finite words is still polynomial [14] and Seidl showed the same over finite trees [13]. In the present paper, we show that this result also holds for strongly unambiguous Büchi automata. To our knowledge, this identifies the first subclass of Büchi automata that is expressively complete for the ω-regular languages and at the same time allows a polynomial time equivalence test.

The polynomial time equivalence test over finite words from [14] uses a counting argument: The main idea is that, for unambiguous automata, the number of accepting runs is equal to the number of accepted inputs. Stearns and Hunt proved that it is sufficient to count the number of accepting paths of the given unambiguous automata only up to a certain length and that this can be done in polynomial time. The problem when trying to adapt such an approach to Büchi automata is that runs of Büchi automata are infinite and one cannot simply count the number of accepted words up to a certain length. However, it is possible to restrict the problem of equivalence of regular languages of infinite words to ultimately periodic words [4] (see also [5]). A word is ultimately periodic if it is of the form $u \cdot v^\omega$ where u and v are finite. It turns out that this restriction to ultimately periodic words allows to adapt the counting argument to the case of strongly unambiguous Büchi automata. Instead of presenting a direct adaption of the proof for finite words we show that the equivalence problem for strongly unambiguous automata can be reduced in polynomial time to the equivalence problem for unambiguous automata on finite words.

This kind of reduction does not seem to work for unambiguous Büchi automata. We show that deciding whether an unambiguous Büchi automaton (even a deterministic one) accepts some periodic word v^ω, where v is of a given length n, is NP-complete. Although this proof does not show that the equivalence problem for unambiguous Büchi automata is difficult, it shows that different methods are required.

[1] In [1] the PSPACE-hardness of the non-universality problem for regular expressions is shown. This can easily be turned into a PSPACE-hardness proof for the equivalence problem for nondeterministic finite automata.

The remainder of the paper is structured as follows. In the second section we give some definitions and simple properties of Büchi automata. In Section 3 we show how to reduce the equivalence problem for strongly unambiguous Büchi automata to the case of unambiguous automata on finite words. In Section 4 we extend these results to strongly k-ambiguous automata, a relaxed notion of strong unambiguity, where each word can have at most k final paths. In Section 5 we show that deciding if a deterministic Büchi automaton accepts periodic words of a given length is NP-complete. We conclude in the last section.

2 Definitions and Background

For an alphabet Σ we denote as usual the set of finite words over Σ by Σ^*, the set of nonempty finite words by Σ^+, and the set of infinite words by Σ^ω. The length of a finite word $u \in \Sigma^*$ is denoted by $|u|$. For an infinite word $\alpha \in \Sigma^\omega$ we denote the jth letter by $\alpha(j)$, i.e., $\alpha = \alpha(0)\alpha(1)\cdots$.

An infinite word of the form $uv^\omega = uvvv\cdots$ for finite words u, v is called ultimately periodic.

We consider nondeterministic finite automata (NFA) on finite words of the form $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$, where Q is a finite set of states, Σ is the input alphabet, $Q_{\text{in}} \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $F \subseteq Q$ is the set of final states. We use the standard terminology for NFAs (see e.g. [9]) and denote the language of words accepted by A by $L(A)$.

A Büchi automaton $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$ is of the same form as an NFA. In contrast to NFAs, a Büchi automaton defines a language of infinite words. A *path* for the infinite word $\alpha \in \Sigma^\omega$ is an infinite sequence of states $q_0 q_1 ...$ such that for all $j \in \mathbb{N}$, $(q_{j-1}, \alpha(j), q_j) \in \Delta$. A *final path* is a path that passes infinitely often through a final state. A path begins in q if $q_0 = q$. If a final path for α begins in some $q_0 \in Q_{\text{in}}$, then the word α is *accepted* by A. So a final path is *accepting* if it starts with an initial state. If an accepting path for α exists, then A *accepts* α. The *language* $L(A)$ is the set of infinite words α accepted by A. For an automaton $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$ we denote by $L_A(q)$ the language accepted by $(Q, \Sigma, \{q\}, \Delta, F)$. The class of languages that can be accepted by Büchi automata is called the class of *ω-regular languages*.

For a finite word u and two states q, q' of A we write $A : q \xrightarrow{u} q'$ if one can reach q' from q on reading u, and we write $A : q \xrightarrow{u}_F q'$ if one can reach q' from q on reading u and by passing through a final state on the way.

It is well known that the equivalence problem for NFAs, i.e., the question whether two given NFAs accept the same language, is PSPACE-complete (see [1]). The same holds for the inclusion problem because equivalence can easily be tested by checking for both inclusions. Furthermore, the lower bound on the complexity easily extends to Büchi automata.

In [14] unambiguous NFAs are considered and it is shown that the equivalence and inclusion problem for these automata are solvable in polynomial time. Unambiguous automata are nondeterministic automata in which for each word there is at most one accepting path.

Our aim is to see to what extent these results can be lifted to Büchi automata. We introduce two notions of unambiguity, the standard one and a stronger notion introduced in [7].

Definition 1. *A Büchi automaton A is called* unambiguous *if every infinite word has at most one accepting path in A, and it is called* strongly unambiguous *if every infinite word has at most one final path.*

Clearly, if A is strongly unambiguous, then for each infinite word α there is at most one state q such that $\alpha \in L_A(q)$. We state this observation as a remark for later reference.

Remark 1. Let A be a strongly unambiguous Büchi automaton and α be an infinite word. If $\alpha \in L_A(q_1)$ and $\alpha \in L_A(q_2)$ then $q_1 = q_2$.

Consider, for example, the automata shown in Figure 1. Both automata accept the language over $\{a, b\}$ consisting of all words that contain infinitely many b. The automaton on the left-hand side is deterministic (where deterministic automata as usual only have a single initial state and for each state and letter at most one outgoing transition) and therefore unambiguous, but it is not strongly unambiguous: the word b^ω is accepted from both states q_0 and q_1. The automaton on the right-hand side is strongly unambiguous. It accepts the same language as the deterministic automaton, but from state p_0 all accepted words start with a, and from p_1 all accepted words start with b.

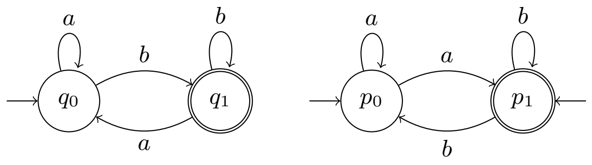


Fig. 1. Example for a deterministic Büchi automaton (left-hand side) and a strongly unambiguous Büchi automaton (right-hand side) for the same language

Note that each strongly unambiguous automaton is unambiguous because each accepting path is also a final path. It has been shown in [2] that each ω-regular language can be accepted by an unambiguous Büchi automaton. The class of strongly unambiguous automata has been introduced in [7][2] and it has been shown that this class is expressively complete for the ω-regular languages.

Theorem 1 ([7]). *Every ω-regular language can be recognized by a strongly unambiguous Büchi automaton.*

[2] The definition in [7] is even more restrictive: It is required that each word has exactly one final path. This allows an easy complementation by complementing the set of initial states. We have chosen the more relaxed notion because the polynomial time equivalence test also works in this setting. Further note that in [7] theses automata are simply called unambiguous and not strongly unambiguous.

This expressive completeness makes strongly unambiguous Büchi automata an interesting class. It is also worth noting that strongly unambiguous Büchi automata naturally occur in the translation from linear temporal logic formulas into Büchi automata. In the standard approach for this translation the Büchi automaton guesses valuations of all subformulas of the given formula and verifies that the guesses are correct (see [3]). An input word is accepted from the unique state that evaluates all subformulas correctly. Hence the automaton that is constructed in this standard way is strongly unambiguous.

Before we turn to the decision problems for strongly unambiguous Büchi automata, we compare them to deterministic automata. Note that deterministic Büchi automata do not capture the full class of ω-regular languages, but using extended acceptance conditions like the Muller acceptance condition, deterministic automata become expressively complete [11] (see also [16]).

The example from Figure 1 already shows that deterministic Büchi automata need not to be strongly unambiguous. In fact, there is no deterministic Büchi automaton that is strongly unambiguous and equivalent to the one from Figure 1: Assume A is a deterministic and strongly unambiguous Büchi automaton accepting all words containing infinitely many b. Then the word b^ω is accepted from the unique initial state. Then b^ω is not accepted from any other state by Remark 1. Thus, the initial state is final and has a b-loop. The only way to accept ab^ω would be to also have an a-loop on the initial state. This would mean that A accepts all ω-words over $\{a, b\}$. Note that we only used the fact that a deterministic automaton has only a single initial state. So this example shows that a set of initial states is necessary for strongly unambiguous Büchi automata, in general.

The next example shows that strongly unambiguous automata can be exponentially more succinct than deterministic ones. We formulate the following remark for deterministic Muller automata, because the Muller condition is the most general one of the standard acceptance conditions that are usually considered: A Muller condition is specified by a family $\mathcal{F}$ of state sets. A run is accepting if the set of states that appear infinitely often in this run is a member of $\mathcal{F}$.

Remark 2. There is a family $(L_n)_{n\geq 1}$ of ω-languages over the alphabet $\{a, b\}$ such that each L_n can be accepted by a strongly unambiguous Büchi automaton with $n+2$ states, and each deterministic Muller automaton for L_n needs at least 2^n states.

Proof. We use the standard syntactic right-congruence for ω-languages L, defined by $u \sim_L v$ iff $u\alpha \in L \Leftrightarrow v\alpha \in L$ for all $\alpha \in \Sigma^\omega$. As for automata on finite words (see [9]), one can show that each Muller automaton for L needs at least as many states as there are classes of $\sim_L$.

The language $L_n = \Sigma^* a \Sigma^{n-1} ab^\omega$ can be recognized by a strongly unambiguous Büchi automaton of size $n + 2$ as shown in Figure 2. The number of $\sim_{L_n}$ classes is at least 2^n, so a deterministic Muller automaton which recognizes L_n has at least 2^n states. $\square$

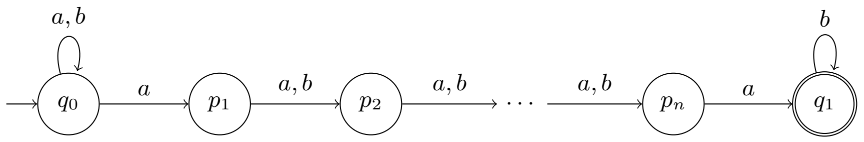


Fig. 2. A strongly unambiguous Büchi automaton for $L = \Sigma^* a \Sigma^{n-1} ab^\omega$

Finally, we would like to mention that there are also deterministic Büchi automata exponentially smaller than strongly unambiguous ones. This is in contrast to unambiguous automata, since each deterministic automaton is unambiguous.

Remark 3. There is a family $(L_n)_{n\geq 1}$ of ω-languages over the alphabet $\{a, b\}$ such that each L_n can be accepted by a deterministic Büchi automaton with $n + 1$ states, and each strongly unambiguous Büchi automaton for L_n needs at least 2^{n-1} states.

Proof. Let L_n be the language of all words in $\{a, b\}^n$ in which the nth letter is a. Using $n + 1$ states a deterministic automaton can check this property.

Let A_n be a strongly unambiguous automaton for L_n. Assume that A_n has less than 2^{n-1} states. Then there are two different words $w_1, w_2 \in \{a, b\}^{n-1}$ of length $n - 1$ such that $w_1 ab^\omega$ and $w_2 ab^\omega$ are accepted by A_n from the same state q. Since w_1 and w_2 are different, we can assume w.l.o.g. that $w_1 = waw_1'$ and $w_2 = wbw_2'$ for some words w, w_1', w_2'. Let $m = |w|$ and v be a word of length $n-m-1$. Then $vwaw_1'b^\omega$ is in L_n. The corresponding accepting run must reach q after having read v because otherwise there would be two different final paths for $waw_1'b^\omega$. Hence, there is also an accepting run for $vwbw_2'b^\omega$: the one that moves to q on reading v, and then accepting $wbw_2'b^\omega$ from q. Since $wbw_2'b^\omega \notin L_n$ we get a contradiction and thus A_n has at least 2^{n-1} states. □

3 Equivalence for Strongly Unambiguous Büchi Automata

In this section we prove that the equivalence problem for strongly unambiguous Büchi automata can be solved in polynomial time. For the case of finite words, the proof of Stearns and Hunt [14] uses a counting argument: For an unambiguous automaton over finite words the number of accepted words of a certain length is the same as the number of accepting paths of this length. The idea is to decide if $L(A_1) \cap L(A_2) = L(A_i)$ for $i = 1, 2$ by simply counting the number of accepting paths. Since $L(A_1) \cap L(A_2) \subseteq L(A_i)$, the equality holds if for each n there is the same number of accepting paths in the automaton for $L(A_1) \cap L(A_2)$ and in A_i. The key argument is then that a comparison of the number of accepting paths is sufficient up to a certain bound of the length.

For general infinite words it is impossible to count final paths in a reasonable way. However, there are infinite words that can be represented in a finite way: the ultimately periodic words introduced in the previous section. This class of words is particularly interesting because it can be used to characterize equivalence and inclusion of ω-regular languages. This easily follows from the closure properties of the class of ω-regular languages and the fact that each non-empty ω-regular language contains an ultimately periodic word.

Theorem 2 ([4]). *Let A_1 and A_2 be two Büchi automata.*

1. *A_1 and A_2 accept the same language if and only if they accept the same ultimately periodic words.*
2. *$L(A_1) \subseteq L(A_2)$ if and only if the ultimately periodic words recognized by A_1 are in $L(A_2)$.*

Using this fact one can indeed adapt the counting argument from [14] to count ultimately periodic words accepted by Büchi automata. However, instead of adapting the proof of [14] we can also give a reduction of the equivalence problem for strongly unambiguous Büchi automata to the equivalence problem for unambiguous automata over finite words. The idea of reducing decision problems for Büchi automata to automata over finite words has already been used in [5]. The main difference here is that the reduction is computable in polynomial time when starting from a strongly unambiguous automaton. Before we present the reduction we state a simple but important property of strongly unambiguous Büchi automata that makes our reduction work.

Lemma 1. *Let $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$ be a strongly unambiguous Büchi automaton. An ultimately periodic word uv^ω is accepted by A iff there are states $q_0 \in Q_{\text{in}}$ and $q \in Q$ such that $A : q_0 \xrightarrow{u} q \xrightarrow{v}_F q$.*

Proof. Obviously, if $A : q_0 \xrightarrow{u} q \xrightarrow{v}_F q$ for states $q_0 \in Q_{\text{in}}$ and $q \in Q$, then uv^ω is accepted. Now suppose that uv^ω is accepted by A. Then there is an accepting path $\rho = q_0 q_1 \ldots.$ on uv^ω that starts in some $q_0 \in Q_{\text{in}}$. Let $q = q_{|u|}$ be the state reached in ρ after reading u, and let $q' = q_{|u|+|v|}$ be the state reached after reading uv. Then v^ω is in $L_A(q)$ and in $L_A(q')$ and thus, by Remark 1 we have $q = q'$. Furthermore, that path $A : q \xrightarrow{v} q$ must pass through a final state because otherwise there would be another accepting path for v^ω that starts from q. By prefixing this accepting path with the v-loop from q to q we would obtain more than one accepting path for v^ω starting in q, contradicting the strong unambiguity of A. □

For the reduction to unambiguous automata on finite words we now build from a strongly unambiguous Büchi automaton an unambiguous automaton on finite words that accepts precisely the words of the form $u\#v$ such that uv^ω is accepted by A. By Theorem 2 two strongly unambiguous Büchi automata are equivalent iff the resulting finite automata are.

Let $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$ be a strongly unambiguous Büchi automaton. The finite automaton we are constructing simulates A on the first part u of the input $u\#v$. When reading $\#$ it stores the current state q and continues reading the input. It accepts if it reaches q again after having read v, and if it has passed through a final state on the way. Since the automaton accepts codings of ultimately periodic words, we call it A_{up}, where the subscript abbreviates ultimately periodic.

Formally, $A_{\mathrm{up}} = (Q', \Sigma \cup \{\#\}, Q_{\mathrm{in}}, \Delta', F')$ is defined as follows:

- $Q' = Q \cup (Q \times Q \times \{0, 1\})$.
- Δ' contains
  - all transitions from Δ,
  - all transitions of the form $(q, \#, (q, q, 0))$ with $q \in Q$, and
  - all transitions of the form $((q, p, i), a, (q, p', i'))$, where $(p, a, p') \in \Delta$, and $i' = \begin{cases} 1 \text{ if } p' \in F, \\ i \text{ if } p' \notin F. \end{cases}$
- $F' = \{(q, q, 1) \mid q \in Q\}$.

Lemma 2. *The automaton A_{up} over finite words is unambiguous and accepts the language $L(A_{\mathrm{up}}) = \{u\#v \in \Sigma^*\#\Sigma^+ \mid uv^\omega \in L(A)\}$.*

Proof. It is clear from the construction that A_{up} only accepts words of the form $u\#v \in \Sigma^*\#\Sigma^+$.

Furthermore, one easily sees that A_{up} accepts precisely those $u\#v$ such that there are states $q_0 \in Q_{\mathrm{in}}$ and $q \in Q$ with $A : q_0 \xrightarrow{u} q$ and $A : q \xrightarrow{v}_F q$. Lemma 1 allows us to conclude that $L(A_{\mathrm{up}}) = \{u\#v \in \Sigma^*\#\Sigma^+ \mid uv^\omega \in L(A)\}$.

The automaton A_{up} simulates the automaton A when reading its input. In particular, if there are two different paths for accepting $u\#v$, then there are also two different paths in A accepting uv^ω. Thus, since A is strongly unambiguous, A_{up} is unambiguous. □

Since inclusion and equivalence for unambiguous automata on finite words are decidable in polynomial time [14], we obtain the following result for strongly unambiguous Büchi automata.

Theorem 3. *The inclusion and equivalence problem for strongly unambiguous Büchi automata are decidable in polynomial time.*

Proof. Given two strongly unambiguous Büchi automata A and B we transform them into the unambiguous automata A_{up} and B_{up}. For these we can test inclusion or equivalence in polynomial time. By Lemma 2 and Theorem 2 the corresponding result is correct for A and B. □

4 Extension to Strongly k-Ambiguous Automata

An automaton over finite words is called k-ambiguous if each accepted word has at most k accepting runs. In [14] it is shown that equivalence and inclusion of k-ambiguous automata can be solved in polynomial time (for a fixed k). Following this idea, we extend the result from the previous section to strongly k-ambiguous Büchi automata, as defined below.

Definition 2. *A Büchi automaton A is called* k-ambiguous *if each infinite word has at most k accepting paths in A, and is called* strongly k-ambiguous *if each infinite word has at most k final paths in A.*

The accepting paths for ultimately periodic words are not as constrained in strongly k-ambiguous automata as they are in strongly unambiguous automata, but there is a similar characterization. The difference is that the loop on the periodic part can be longer, but it can consist of at most k repetitions of the periodic pattern.

Lemma 3. *Let $A = (Q, \Sigma, Q_{\text{in}}, \Delta, F)$ be a strongly k-ambiguous Büchi automaton. An ultimately periodic word uv^ω is accepted by A iff there are states $q_0 \in Q_{\text{in}}$ and $q_1, \dots, q_{k+1} \in Q$ such that*

$$A : q_0 \xrightarrow{u} q_1 \xrightarrow{v} q_2 \xrightarrow{v} \cdots \xrightarrow{v} q_k \xrightarrow{v} q_{k+1} \ ,$$

$q_{k+1} = q_s$ for some $1 \leq s \leq k$, and $q_i \xrightarrow{v}_F q_{i+1}$ for some $s \leq i \leq k$.

Proof. The proof is similar to the proof of Lemma 1. Assume that uv^ω is accepted by A, let ρ be an accepting path, and let $q_0, \dots, q_{k+1}$ be such that ρ starts as follows:

$$q_0 \xrightarrow{u} q_1 \xrightarrow{v} q_2 \xrightarrow{v} \cdots \xrightarrow{v} q_{k+1}.$$

Then v^ω is in $L_A(q_i)$ for all $i \in \{1, \dots, k+1\}$. Since A is strongly k-ambiguous, there must be some $i \neq j$ such that $q_i = q_j$. Now assume that $q_{k+1} \neq q_s$ for all $s \in \{1, \dots, k\}$. Then we get arbitrarily many different accepting paths of A on uv^ω by repeating the loop between q_i and q_j an arbitrary number of times before continuing the path towards q_{k+1}. Hence, there must exist some $1 \leq s \leq k$ with $q_{k+1} = q_s$.

It remains to show that there is some $i \in \{s, \dots, k\}$ such that $q_i \xrightarrow{v}_F q_{i+1}$. Assume the contrary. Then we can again produce an arbitrary number of accepting paths for uv^ω by repeating the loop $q_s \xrightarrow{v} \cdots q_k \xrightarrow{v} q_s$ (that does not contain a final state) before continuing the path with the accepting part after q_{k+1} in ρ. All these paths are different because we always increase the part that does not contain a final state before continuing with the part of ρ after q_{k+1} that contains infinitely many accepting states. This proves one direction of the claim. The other direction is obvious. □

We now construct an automaton A_{up}^k that has the same property as A_{up} from the previous section. This automaton, after having read u and when reading #, guesses the states $q_2, \dots, q_k$ and then verifies that the properties from Lemma 3 are satisfied.

Formally $A_{\text{up}}^k = (Q', \Sigma \cup \{\#\}, Q_{\text{in}}, \Delta', F')$ is constructed as follows:

- $Q' = Q \cup ((Q \times Q)^k \times \{0, \dots, k\})$.
- Δ' contains all transitions from Δ, all transitions of the form

$$(q_1, \#, [(q_1, q_1), \dots, (q_k, q_k), 0])$$

  and all transitions of the form (written with an arrow for better readability)

$$[(q_1, p_1), \dots, (q_k, p_k), i] \xrightarrow{a} [(q_1, p_1'), \dots, (q_k, p_k'), i']$$

where $(p_j, a, p'_j) \in \Delta$ for all j, and

$$i' = \max(\{i\} \cup \{j \mid p'_j \in F\}).$$

– $F' = \{[(q_1, q_2), (q_2, q_3) \ldots, (q_k, q_s), i] \mid 1 \leq s \leq k \text{ and } s \leq i \leq k\}$.

Lemma 4. *The automaton A^k_{up} over finite words is k-ambiguous and accepts the language $L(A^k_{\text{up}}) = \{u\#v \in \Sigma^*\#\Sigma^+ \mid uv^\omega \in L(A)\}$.*

Proof. The proof is based on Lemma 3 and is similar to the proof of Lemma 2. □

As in the previous section we conclude:

Theorem 4. *For a fixed k, the inclusion and equivalence problem for strongly k-ambiguous Büchi automata can be decided in polynomial time.*

5 Periodic Words in Deterministic Automata

In this section we show that deciding for a given deterministic Büchi automaton whether it accepts a periodic word with a period of a given length is NP-complete. Since deterministic automata are special cases of unambiguous automata, this shows that it is unlikely that the counting techniques that work for unambiguous automata on finite words and for strongly unambiguous Büchi automata can be transferred to unambiguous Büchi automata.

More formally, consider for a class $\mathcal{C}$ of Büchi automata the following decision problem Periodic($\mathcal{C}$):

Given: A Büchi automaton A from the class $\mathcal{C}$ and a natural number n.
Question: Does there exist a word v with $|v| \leq n$ such that A accepts v^ω?

Proposition 1. *The problem* Periodic($\mathcal{C}$) *is NP-complete for the class $\mathcal{C}$ of deterministic Büchi automata.*

Proof. Membership in NP can easily be verified: If n is bigger than the number of states of A, then A accepts a periodic word of length n iff the initial state of A is contained in a loop with a final state. This can be checked in polynomial time by standard graph algorithms. If n is smaller than the number of states of A, then we can guess a word v of length at most n and verify in polynomial time whether v^ω is accepted by A.

For the NP-hardness we give a reduction from the satisfiability problem of Boolean formulas in conjunctive normal form (CNF). Let $\varphi = C_1 \wedge \cdots \wedge C_k$ be such a formula over n variables $x_1, \ldots, x_n$ consisting of clauses $C_1, \ldots, C_k$. A truth assignment of the variables can naturally be encoded by a word of length n over the alphabet $\{0, 1\}$: Position i of the word is 0 if x_i is false, and 1 if x_i is true.

The deterministic Büchi automaton that we construct works over the alphabet $\{0, 1, \#\}$ and only accepts words from $(\{0, 1\}^n\#)^\omega$: It basically consists of a loop of length $(n+1) \cdot k$ that reads words $v_1\#v_2\#\cdots v_k\#$ where each v_i is from $\{0,1\}^n$. Each v_i encodes an assignment of the n variables as described above. The automaton A checks whether the assignment coded by v_i satisfies the clause C_i: Assume that A reads letter j of v_i. If this letter is 0 and $\neg x_j$ is contained in C_i or the letter is 1 and x_j is contained in C_i, then A sets a bit indicating that the current clause is satisfied. If it reaches the end of v_i without the bit being set it rejects by moving to a sink state. Otherwise it reads # and proceeds to v_{i+1}. After having processed k such words, A loops back to the initial state. All states except the sink state are accepting.

If this automaton accepts a periodic word with period of length at most $n+1$, then it must be of the form $(v\#)^\omega$, where $v \in \{0,1\}^n$ satisfies all the clauses of φ. This shows that φ is satisfiable iff the constructed automaton accepts a periodic word with period of length at most $n+1$. □

This is of course not a proof for the hardness of equivalence for unambiguous Büchi automata. It only shows that the techniques that have been used so far for obtaining polynomial time equivalence tests are unlikely to work for the case of unambiguous Büchi automata.

6 Conclusion

The class of strongly unambiguous Büchi automata is the first known class of Büchi automata as expressive as nondeterministic Büchi automata for which the inclusion and equivalence problem can be decided in polynomial time. However, this class is quite difficult to understand because strongly unambiguous Büchi automata are co-deterministic [7] and we usually think in a deterministic way. In addition, there are deterministic Büchi automata exponentially smaller than strongly unambiguous ones, which is impossible for unambiguous Büchi automata, because every deterministic Büchi automaton is unambiguous. Therefore, it would be interesting to settle the complexity of the equivalence problem for unambiguous Büchi automata.

References

1. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley, New York (1974)
2. Arnold, A.: Rational ω-languages are non-ambiguous. Theoretical Computer Science 26, 221–223 (1983)
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: International Congress on Logic, Methodology and Philosophy of Science, pp. 1–11. Stanford University Press, Stanford (1962)
5. Calbrix, H., Nivat, M., Podelski, A.: Ultimately periodic words of rational ω-languages. In: Main, M.G., Melton, A.C., Mislove, M.W., Schmidt, D., Brookes, S.D. (eds.) MFPS 1993. LNCS, vol. 802, pp. 554–566. Springer, Heidelberg (1994)

6. Carayol, A., Löding, C.: MSO on the infinite binary tree: Choice and order. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 161–176. Springer, Heidelberg (2007)
7. Carton, O., Michel, M.: Unambiguous Büchi automata. Theor. Comput. Sci. 297(1-3), 37–81 (2003)
8. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Löding, C., Lugiez, D., Tison, S., Tommasi, M.: Tree Automata Techniques and Applications, http://tata.gforge.inria.fr/ (last release: October 12, 2007)
9. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
10. Landweber, L.H.: Decision problems for ω-automata. Mathematical Systems Theory 3, 376–384 (1969)
11. McNaughton, R.: Testing and generating infinite sequences by a finite automaton. Information and Control 9(5), 521–530 (1966)
12. Niwiński, D., Walukiewicz, I.: Ambiguity problem for automata on infinite trees (unpublished note)
13. Seidl, H.: Deciding equivalence of finite tree automata. SIAM J. Comput. 19(3), 424–437 (1990)
14. Stearns, R.E., Hunt III, H.B.: On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata. SIAM Journal on Computing 14(3), 598–611 (1985)
15. Stockmeyer, L.J.: The Complexity of Decision Problems in Automata Theory and Logic. PhD thesis, Dept. of Electrical Engineering, MIT, Boston, Mass. (1974)
16. Thomas, W.: Languages, automata, and logic. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Language Theory, vol. III, pp. 389–455. Springer, Heidelberg (1997)
17. Valiant, L.G.: Relative complexity of checking and evaluating. Inf. Process. Lett. 5(1), 20–23 (1976)

Pregroup Grammars with Letter Promotions

Wojciech Buszkowski[1,2] and Zhe Lin[1,3]

[1] Adam Mickiewicz University in Poznań, Poland
[2] University of Warmia and Mazury in Olsztyn, Poland
[3] Sun Yat-sen University in Guangzhou, China
buszko@amu.edu.pl, pennyshaq@gmail.com

Abstract. We study pregroup grammars with letter promotions $p^{(m)} \Rightarrow q^{(n)}$. We show that the Letter Promotion Problem for pregroups is solvable in polynomial time, if the size of $p^{(n)}$ is counted as $|n|+1$. In Mater and Fix [11], the problem is shown to be NP-hard, but their proof assumes the binary (or decimal, etc.) representation of n in $p^{(n)}$, which seems less natural for applications. We reduce the problem to a graph-theoretic problem, which is subsequently reduced to the emptiness problem for context-free languages. As a consequence, the following problems are in P: the word problem for pregroups with letter promotions and the membership problem for pregroup grammars with letter promotions.

1 Introduction and Preliminaries

Pregroups, introduced in Lambek [8], are ordered algebras $(M, \leq, \cdot, l, r, 1)$ such that $(M, \leq, \cdot, 1)$ is a partially ordered monoid (hence $\cdot$ is monotone in both arguments), and l, r are unary operations on M, fulfilling the following conditions:

$$a^l a \leq 1 \leq a a^l \,,\; a a^r \leq 1 \leq a^r a \,, \tag{1}$$

for all $a \in M$. The operation $\cdot$ is referred to as *product*. The element a^l (resp. a^r) is called the *left* (resp. *right*) *adjoint* of a.

The following laws are valid in pregroups:

$$1^l = 1 = 1^r \,, \tag{2}$$

$$(a^l)^r = a = (a^r)^l \,, \tag{3}$$

$$(ab)^l = b^l a^l \,,\; (ab)^r = b^r a^r \,, \tag{4}$$

$$a \leq b \text{ iff } b^l \leq a^l \text{ iff } b^r \leq a^l \,. \tag{5}$$

In any pregroup, one defines $a \backslash b = a^r b$, $a/b = ab^l$, and proves that $\cdot, \backslash, /$ satisfy the residuation law:

$$ab \leq c \text{ iff } b \leq a \backslash c \text{ iff } a \leq c/b, \tag{6}$$

for all elements a, b, c. Consequently, pregroups are a special class of residuated monoids, i.e. models of the Lambek calculus **L*** [3,2].

Lambek [8] (also see [9,10]) offers (free) pregroups as a computational machinery for lexical grammars, alternative to the Lambek calculus. The latter is widely recognized as a basic logic of categorial grammars [17,2]; linguists usually employ the system **L** of the Lambek calculus, which is complete with respect to residuated semigroups (it is weaker than **L***).

The logic of pregroups is called Compact Bilinear Logic (**CBL**). It arises from Bilinear Logic (Noncommutative **MLL**) by collapsing 'times' and 'par', whence also 0 and 1. **CBL** is stronger than **L***; for instance, $(p/((q/q)/p))/p \leq p$ is valid in pregroups but not in residuated monoids [3], whence it is provable in **CBL**, but not in **L***. By the same example, **CBL** is stronger than Bilinear Logic, since the latter is a conservative extension of **L***.

Let M be a pregroup. For $a \in M$, one defines $a^{(n)}$ as follows: $a^{(0)} = a$; if n is negative, then $a^{(n)} = a^{l \dots l}$ (l is iterated $|n|$ times); if n is positive, then $a^{(n)} = a^{r \dots r}$ (r is iterated n times). The following laws can easily be proved:

$$(a^{(n)})^l = a^{(n-1)}\,,\ (a^{(n)})^r = a^{(n+1)}\,,\ \text{for all } n \in \mathbf{Z}\,, \tag{7}$$

$$a^{(n)}a^{(n+1)} \leq 1 \leq a^{(n+1)}a^{(n)}\,,\ \text{for all } n \in \mathbf{Z}\,, \tag{8}$$

$$(a^{(m)})^{(n)} = a^{(m+n)}\,,\ \text{for all } m, n \in \mathbf{Z}\,, \tag{9}$$

$$a \leq b \text{ iff } a^{(n)} \leq b^{(n)}\,,\ \text{for all even } n \in \mathbf{Z}\,, \tag{10}$$

$$a \leq b \text{ iff } b^{(n)} \leq a^{(n)}\,,\ \text{for all odd } n \in \mathbf{Z}\,, \tag{11}$$

where **Z** denotes the set of integers.

CBL can be formalized as follows. Let $(P, \leq)$ be a nonempty finite poset. Elements of P are called *atoms*. *Terms* are expressions of the form $p^{(n)}$ such that $p \in P$ and n is an integer. One writes p for $p^{(0)}$. *Types* are finite strings of terms. Terms are denoted by t, u and types by X, Y, Z. The relation $\Rightarrow$ on the set of types is defined by the following rules:

(CON) $X, p^{(n)}, p^{(n+1)}, Y \Rightarrow X, Y$,
(EXP) $X, Y \Rightarrow X, p^{(n+1)}, p^{(n)}, Y$,
(POS) $X, p^{(n)}, Y \Rightarrow X, q^{(n)}, Y$, if $p \leq q$, for even n, and $q \leq p$, for odd n,

called Contraction, Expansion, and Poset rules, respectively (the latter are called Induced Steps in Lambek [8]). To be precise, $\Rightarrow$ is the reflexive and transitive closure of the relation defined by these rules. The pure **CBL** is based on a trivial poset $(P, =)$.

An *assignment* in a pregroup M is a mapping $\mu : P \mapsto M$ such that $\mu(p) \leq \mu(q)$ in M whenever $p \leq q$ in $(P, \leq)$. Clearly any assignment μ is uniquely extendible to a homomorphism of the set of types into M; one sets $\mu(\epsilon) = 1$, $\mu(p^{(n)}) = (\mu(p))^{(n)}$, $\mu(XY) = \mu(X)\mu(Y)$. The following completeness theorem is true: $X \Rightarrow Y$ holds in **CBL** if and only if, for any pregroup M and any assignment μ of P in M, $\mu(X) \leq \mu(Y)$ [3].

A *pregroup grammar* assigns a finite set of types to each word from a finite lexicon Σ. Then, a nonempty string $v_1 \dots v_n$ ($v_i \in \Sigma$) is assigned type X, if there exist types $X_1, \dots, X_n$ initially assigned to words $v_1, \dots, v_n$, respectively, such that $X_1, \ldots, X_n \Rightarrow X$ in **CBL**. For instance, if 'goes' is assigned type $\pi_3^{(1)} s_1$ and 'he' type π_3, then 'he goes' is assigned type s_1 (statement in the present tense). π_k represents the $k-$th person pronoun. For the past tense, the person is irrelevant; so, π represents pronoun (any person), and one assumes $\pi_k \leq \pi$, for $k = 1, 2, 3$. Now, if 'went' is assigned type $\pi^{(1)} s_2$, then 'he went' is assigned type s_2 (statement in the past tense), and similarly for 'I went', 'you went'. Assuming $s_i \leq s$, for $i = 1, 2$, one can assign type s (statement) to all sentences listed above. These examples come from [8].

A pregroup grammar is formally defined as a quintuple $G = (\Sigma, P, I, s, R)$ such that Σ is a finite alphabet (lexicon), P is a finite set (of atoms), s is a designated atom (the principal type), I is a finite relation between elements of Σ and types on P, and R is a partial ordering on P. One writes $p \leq q$ for pRq, if R is fixed. *The language* of G, denoted $L(G)$, consists of all strings $x \in \Sigma^+$ such that G assigns type s to x (see the above paragraph). Pregroup grammars are weakly equivalent to $\epsilon-$free context-free grammars [3]; hence, the former provide a lexicalization of the latter.

As shown in [1], every pregroup grammar can be fully lexicalized; there exists a polynomial time transformation which sends any pregroup grammar to an equivalent pregroup grammar on a trivial poset $(P, =)$. Actually, an exponential time procedure is quite obvious: it suffices to apply all possible (POS)-transitions to the lexical types in I [5].

Lambek [8] proves a normalization theorem for **CBL** (also called: Lambek Switching Lemma). One introduces new rules:

(GCON) $X, p^{(n)}, q^{(n+1)}, Y \Rightarrow X, Y$,
(GEXP) $X, Y \Rightarrow X, p^{(n+1)}, q^{(n)}, Y$,

if either n is even and $p \leq q$, or n is odd and $q \leq p$. These rules are called Generalized Contraction and Generalized Expansion, respectively. Clearly they are derivable in **CBL**: (GCON) amounts to (POS) followed by (CON), and (GEXP) amounts to (EXP) followed by (POS). Lambek's normalization theorem states: if $X \Rightarrow Y$ in **CBL**, then there exist types Z, U such that $X \Rightarrow Z$, by a finite number of instances of (GCON), $Z \Rightarrow U$, by a finite number of instances of (POS), and $U \Rightarrow Y$, by a finite number of instances of (GEXP). Consequently, if Y is a term or $Y = \epsilon$, then $X \Rightarrow Y$ in **CBL** if and only if X can be reduced to Y without (GEXP) (hence, by (CON) and (POS) only). The normalization theorem is equivalent to the cut-elimination theorem for a sequent system of **CBL** [4].

This yields the polynomial time complexity of the provability problem for **CBL** [3,4]. For any type X, define X^l and X^r as follows:

$$\epsilon^l = \epsilon^r = \epsilon\,,\ (t_1 t_2 \cdots t_k)^\alpha = (t_k)^\alpha \cdots (t_2)^\alpha (t_1)^\alpha\,, \tag{12}$$

for $\alpha \in \{l, r\}$, where t^α is defined according to (7): $(p^{(n)})^l = p^{(n-1)}$, $(p^{(n)})^r = p^{(n+1)}$. In **CBL** the following equivalences hold:

$$X \Rightarrow Y \text{ iff } X, Y^r \Rightarrow \epsilon \text{ iff } Y^l, X \Rightarrow \epsilon\,, \tag{13}$$

for all types X, Y. We prove the first equivalence. Assume $X \Rightarrow Y$. Then, $X, Y^r \Rightarrow Y, Y^r \Rightarrow \epsilon$, by an obvious congruence property of $\Rightarrow$ and a finite number of (CON). Assume $X, Y^r \Rightarrow \epsilon$. Then, $X \Rightarrow X, Y^r, Y \Rightarrow Y$, by a finite number of (EXP) and a congruence property of $\Rightarrow$. In a similar way, one proves: $X \Rightarrow Y$ iff $Y^l, X \Rightarrow \epsilon$.

In order to verify whether $X \Rightarrow Y$ in **CBL** one verifies whether $X, Y^r \Rightarrow \epsilon$; the latter holds if and only if XY^r can be reduced to ϵ by a finite number of instances of (GCON). An easy modification of the CYK-algorithm for context-free grammars yields a polynomial time algorithm, solving this problem (also see [12]). Furthermore, every pregroup grammar can be transformed into an equivalent context-free grammar in polynomial time [3,5]. Francez and Kaminsky [6] show that pregroup grammars augmented with partial commutation can generate some non-context-free languages.

We have formalized **CBL** with special assumptions. Assumptions $p \leq q$ in nontrivial posets express different forms of subtyping, as shown in the above examples.

It is interesting to consider **CBL** enriched with more general assumptions. Mater and Fix [11] show that **CBL** enriched with finitely many assumptions of the general form $X \Rightarrow Y$ can be undecidable (the word problem for groups is reducible to systems of that kind). For assumptions of the form $t \Rightarrow u$ (called *letter promotions*) they prove a weaker form of Lambek's normalization theorem for the resulting calculus (for sequents $X \Rightarrow \epsilon$ only).

A complete system of **CBL** with letter promotions is obtained by modifying (POS) to the following Promotion Rules:

(PRO) $X, p^{(m+k)}, Y \Rightarrow X, q^{(n+k)}, Y$, if either k is even and $p^{(m)} \Rightarrow q^{(n)}$ is an assumption, or k is odd and $q^{(n)} \Rightarrow p^{(m)}$ is an assumption.

The Letter Promotion Problem for pregroups (LPPP) is the following: given a finite set R, of letter promotions, and terms t, u, verify whether $t \Rightarrow u$ in **CBL** enriched with all promotions from R as assumptions.

To formulate the problem quite precisely, we need some formal notions. Let R denote a finite set of letter promotions. We write $R \vdash_{\mathbf{CBL}} X \Rightarrow Y$, if X can be transformed into Y, using finitely many instances of (CON), (EXP) and (PRO), restricted to the assumptions from R. Now, the problem under consideration amounts to verifying whether $R \vdash_{\mathbf{CBL}} t \Rightarrow u$, for given R, t, u.

Since the formalism is based on no fixed poset, we have to explain what are atoms (atomic types). We fix a denumerable set P of atoms. Terms and types are defined as above. $P(R)$ denotes the set of atoms appearing in assumptions from R. By an assignment in M we mean now a mapping $\mu : P \mapsto M$. We prove a standard completeness theorem.

Theorem 1. *$R \vdash_{\mathbf{CBL}} X \Rightarrow Y$ if, and only if, for any pregroup M and any assignment μ in M, if all assumptions from R are true in (M, μ), then $X \Rightarrow Y$ is true in (M, μ).*

Proof. The 'only if' part is easy. For the 'if' part one constructs a special pregroup M whose elements are equivalence classes of the relation: $X \sim Y$ iff

$R \vdash_{CBL} X \Rightarrow Y$ and $R \vdash_{CBL} Y \Rightarrow X$. One defines: $[X] \cdot [Y] = [XY]$, $[X]^\alpha = [X^\alpha]$, for $\alpha \in \{l, r\}$, $[X] \leq [Y]$ iff $R \vdash_{CBL} X \Rightarrow Y$. For $\mu(p) = [p]$, $p \in P$, one proves: $X \Rightarrow Y$ is true in (M, μ) iff $R \vdash_{CBL} X \Rightarrow Y$. □

Mater and Fix [11] claim that LPPP is NP-complete. Actually, their paper only provides a proof of NP-hardness; even the decidability of LPPP does not follow from their results.

The NP-hardness is proved by a reduction of the following Subset Sum Problem to LPPP: given a nonempty finite set of integers $S = \{k_1, \ldots, k_m\}$ and an integer k, verify whether there exists a subset $X \subseteq S$ such that the sum of all integers from X equals k. The latter problem is NP-complete, if integers are represented in a binary (or decimal, etc.) code; see [7]. For the reduction, one considers $m+1$ atoms $p_0, \ldots, p_m$ and the promotions R: $p_{i-1} \Rightarrow p_i$, for all $i = 1, \ldots, m$, and $p_{i-1} \Rightarrow (p_i)^{(2k_i)}$, for all $i = 1, \ldots, m$. Then, the Subset Sum Problem has a solution if and only if $p_0 \Rightarrow (p_m)^{(2k)}$ is derivable from R. Clearly the reduction assumes the binary representation of n in $p^{(n)}$.

In linguistic applications, it is more likely that R contains many promotions $p^{(m)} \Rightarrow q^{(n)}$, but all integers in them are relatively small. In Lambek's original setting, these integers are equal to 0. It is known that in pregroups: $a \leq a^{ll}$ iff a is surjective (i.e. $ax = b$ has a solution, for any b), and $a^{ll} \leq a$ iff a is injective (i.e. $ax = ay$ implies $x = y$) [3]. One can postulate these properties by promotions: $p \Rightarrow p^{(-2)}$, $p^{(-2)} \Rightarrow p$. Let n be the atomic type of negation 'not', then $nn \Rightarrow \epsilon$ expresses the double negation law on the syntactic level, and this promotion is equivalent to $n \Rightarrow n^{(-1)}$. All linguistic examples in [8,10] use at most three (usually, one or two) iterated left or right adjoints. Accordingly, binary encoding is not very useful for such applications.

It seems more natural to look at $p^{(n)}$ as an abbreviated notation for $p^{l \cdots l}$ or $p^{r \cdots r}$, where adjoints are iterated $|n|$ times, and take $|n|+1$ as the proper complexity measure of this term. Under this proviso, we prove below that LPPP is polynomial time decidable. As a consequence, the provability problem for **CBL** with letter promotions has the same complexity. Accordingly, we prove the decidability of both problems, and the polynomial time complexity of them (under the proviso). (The final comments of [11] suggest that a practically useful version of LPPP may have a lower complexity.)

Oehrle [15] and Moroz [14] provide some cubic parsing algorithms for pregroup grammars (the former uses some graph-theoretic ideas; the latter modifies Savateev's algorithm for the unidirectional Lambek grammars [16]). These algorithms can be adjusted for pregroup grammars with letter promotions. Pregroup grammars with (finitely many) letter promotions are weakly equivalent to $\epsilon-$free context-free grammars. We do not elaborate these matters here, since they are rather routine variants of results obtained elsewhere; also see [3,5,14].

2 The Normalization Theorem

We provide a full proof of the Lambek-style normalization theorem for **CBL** with letter promotions, which yields a simpler formulation of LPPP.

We write $t \Rightarrow_R u$, if $t \Rightarrow u$ is an instance of (PRO), restricted to the assumptions from R (X, Y are empty). We write $t \Rightarrow_R^* u$, if there exist terms $t_0, \ldots, t_k$ such that $k \geq 0$, $t_0 = t$, $t_k = u$, and $t_{i-1} \Rightarrow_R t_i$, for all $i = 1, \ldots, k$. Hence $\Rightarrow_R^*$ is the reflexive and transitive closure of $\Rightarrow_R$.

It is expedient to introduce derivable rules of Generalized Contraction and Generalized Expansion for **CBL** with letter promotions.

(GCON$-R$) $X, p^{(m)}, q^{(n+1)}, Y \Rightarrow X, Y$, if $p^{(m)} \Rightarrow_R^* q^{(n)}$,
(GEXP$-R$) $X, Y \Rightarrow X, p^{(n+1)}, q^{(m)}, Y$, if $p^{(n)} \Rightarrow_R^* q^{(m)}$.

These rules are derivable in **CBL** with assumptions from R, and (CON), (EXP) are special instances of them. We also treat any iteration of (PRO)-steps as a single step:

(PRO$-R$) $X, t, Y \Rightarrow X, u, Y$, if $t \Rightarrow_R^* u$.

The following normalization theorem has been proved in [11], for the particular case $Y = \epsilon$: if $X \Rightarrow \epsilon$ is provable, then X reduces to ϵ by (GCON$-R$) only. This easily follows from Theorem 2 and does not directly imply Lemma 1. Here we prove the full version (this result is essential for further considerations).

Theorem 2. *If $R \vdash_{\mathbf{CBL}} X \Rightarrow Y$, then there exist Z, U such that $X \Rightarrow Z$ by a finite number of instances of (GCON$-R$), $Z \Rightarrow U$ by a finite number of instances of (PRO$-R$), and $U \Rightarrow Y$ by a finite number of instances of (GEXP$-R$).*

Proof. By *a derivation* of $X \Rightarrow Y$ in **CBL** from the set of assumptions R, we mean a sequence $X_0, \ldots, X_k$ such that $X = X_0$, $Y = X_k$ and, for any $i = 1, \ldots, k$, $X_{i-1} \Rightarrow X_i$ is an instance of (GCON$-R$), (GEXP$-R$) or (PRO$-R$); k is *the length* of this derivation. We show that every derivation $X_0, \ldots, X_k$ of $X \Rightarrow Y$ in **CBL** from R can be transformed into a derivation of the required form (a normal derivation) whose length is at most k. We proceed by induction on k.

For $k = 0$ and $k = 1$ the initial derivation is normal; for $k = 0$, one takes $X = Z = U = Y$, and for $k = 1$, if $X \Rightarrow Y$ is an instance of (GCON$-R$), one takes $Z = U = Y$, if $X \Rightarrow Y$ is an instance of (GEXP$-R$), one takes $X = Z = U$, and if $X \Rightarrow Y$ is an instance of (PRO$-R$), one takes $X = Z$ and $U = Y$.

Assume $k > 1$. The derivation $X_1, \ldots, X_k$ is shorter, whence it can be transformed into a normal derivation $Y_1, \ldots, Y_l$ such that $X_1 = Y_1$, $X_k = Y_l$ and $l \leq k$. If $l < k$, then $X_0, Y_1, \ldots, Y_l$ is a derivation of $X \Rightarrow Y$ of length less than k, whence it can be transformed into a normal derivation, by the induction hypothesis. So assume $l = k$.

Case 1. $X_0 \Rightarrow X_1$ is an instance of (GCON$-R$). Then $X_0, Y_1, \ldots, Y_l$ is a normal derivation of $X \Rightarrow Y$ from R.

Case 2. $X_0 \Rightarrow X_1$ is an instance of (GEXP$-R$), say $X_0 = UV$, $X_1 = Up^{(n+1)} q^{(m)}V$, and $p^{(n)} \Rightarrow_R^* q^{(m)}$. We consider two subcases.

Case 2.1 No (GCON$-R$)-step of $Y_1, \ldots, Y_l$ acts on the designated occurrences of $p^{(n+1)}, q^{(m)}$. If also no (PRO$-R$)-step of $Y_1, \ldots, Y_l$ acts on these designated terms, then we drop $p^{(n+1)}q^{(m)}$ from all types appearing in (GCON$-R$)- steps and (PRO$-R$)-steps of $Y_1,\ldots,Y_l$, then introduce them by a single instance of (GEXP$-R$), and continue the (GEXP$-R$)-steps of $Y_1,\ldots,Y_l$; this yields a normal derivation of $X \Rightarrow Y$ of length k. Otherwise, let $Y_{i-1} \Rightarrow Y_i$ be the first (PRO$-R$)-step of $Y_1,\ldots,Y_l$ which acts on $p^{(n+1)}$ or $q^{(m)}$. If it acts on $p^{(n+1)}$, then there exist a term $r^{(m')}$ and types T, W such that $Y_{i-1} = Tp^{(n+1)}W$, $Y_i = Tr^{(m')}W$ and $p^{(n+1)} \Rightarrow^*_R r^{(m')}$. Then, $r^{(m'-1)} \Rightarrow^*_R p^{(n)}$, whence $r^{(m'-1)} \Rightarrow^*_R q^{(m)}$, and we can replace the derivation $X_0, Y_1,\ldots,Y_l$ by a shorter derivation: first apply (GEXP$-R$) of the form $U, V \Rightarrow U, r^{(m')}, q^{(m)}, V$, then derive $Y_1,\ldots,Y_{i-1}$ in which $p^{(n+1)}$ is replaced by $r^{(m')}$, drop Y_i, and continue $Y_{i+1},\ldots,Y_l$. By the induction hypothesis, this derivation can be transformed into a normal derivation of length less than k. If $Y_{i-1} \Rightarrow Y_i$ acts on $q^{(m)}$, then there exist a term $r^{(m')}$ and types T, W such that $Y_{i-1} = Tq^{(m)}W$, $Y_i = Tr^{(m')}W$ and $q^{(m)} \Rightarrow^*_R r^{(m')}$. Then, $p^{(n)} \Rightarrow^*_R r^{(m')}$, and we can replace the derivation $X_0, Y_1,\ldots,Y_l$ by a shorter derivation: first apply (GEXP$-R$) of the form $U, V \Rightarrow U, p^{(n+1)}, r^{(m')}, V$, then derive $Y_1,\ldots,Y_{i-1}$ in which $q^{(m)}$ is replaced by $r^{(m')}$, drop Y_i, and continue $Y_{i+1},\ldots,Y_l$. Again we apply the induction hypothesis.

Case 2.2. Some (GCON$-R$)-step of $Y_1,\ldots,Y_l$ acts on (some of) the designated occurrences of $p^{(n+1)}, q^{(m)}$. Let $Y_{i-1} \Rightarrow Y_i$ be the first step of that kind. There are three possibilities. (I) This step acts on both $p^{(n+1)}$ and $q^{(m)}$. Then, the derivation $X_0, Y_1,\ldots,Y_l$ can be replaced by a shorter derivation: drop the first application of (GEXP$-R$), then derive $Y_1,\ldots,Y_{i-1}$ in which $p^{(n+1)}q^{(m)}$ is omitted, drop Y_i, and continue $Y_{i+1},\ldots,Y_l$. We apply the induction hypothesis. (II) This step acts on $p^{(n+1)}$ only. Then, $Y_{i-1} = Tr^{(m')}p^{(n+1)}q^{(m)}W$, $Y_i = T, q^{(m)}, W$ and $r^{(m')} \Rightarrow^*_R p^{(n)}$. The derivation $X_0, Y_1,\ldots,Y_l$ can be replaced by a shorter derivation: drop the first application of (GEXP$-R$), then derive $Y_1,\ldots,Y_{i-1}$ in which $p^{(n+1)}q^{(m)}$ is omitted, derive Y_i by a (PRO$-R$)-step (notice $r^{(m')} \Rightarrow^*_R q^{(m)}$), and continue $Y_{i+1},\ldots,Y_l$. We apply the induction hypothesis. (III) This step acts on $q^{(m)}$ only. Then, $Y_{i-1} = Tp^{(n+1)}q^{(m)}r^{(m'+1)}W$, $Y_i = Tp^{(n+1)}W$ and $q^{(m)} \Rightarrow^*_R r^{(m')}$. The derivation $X_0, Y_1,\ldots,Y_l$ can be replaced by a shorter derivation: drop the first application of (GEXP$-R$), then derive $Y_1,\ldots,Y_{i-1}$ in which $p^{(n+1)}q^{(m)}$ is dropped, derive Y_i by a (PRO$-R$)-step (notice $r^{(m'+1)} \Rightarrow^*_R p^{(n+1)}$), and continue $Y_{i+1},\ldots,Y_l$. We apply the induction hypothesis.

Case 3. $X_0 \Rightarrow X_1$ is an instance of (PRO$-R$), say $X_0 = UtV$, $X_1 = UuV$ and $t \Rightarrow^*_R u$. We consider two subcases.

Case 3.1. No (GCON$-R$)-step of $Y_1,\ldots,Y_l$ acts on the designated occurrence of u. Then $X_0, Y_1,\ldots,Y_l$ can be transformed into a normal derivation of length k: drop the first application of (PRO$-R$), apply all (GCON$-R$)-steps of $Y_1,\ldots,Y_l$ in which the designated occurrences of u are replaced by t, apply a (PRO$-R$)-step which changes t into u, and continue the remaining steps of $Y_1,\ldots,Y_l$.

Case 3.2. Some (GCON$-R$)-step of $Y_1,\ldots,Y_l$ acts on the designated occurrence of u. Let $Y_{i-1} \Rightarrow Y_i$ be the first step of that kind. There are two possibilities. (I) $Y_{i-1} = Tuq^{(n+1)}W$, $Y_i = TW$ and $u \Rightarrow^*_R q^{(n)}$. Since $t \Rightarrow^*_R q^{(n)}$, then $X, Y_1,\ldots,Y_l$ can be transformed into a shorter derivation: drop the first application of (PRO$-R$), derive $Y_1, \ldots, Y_{i-1}$ in which the designated occurrences of u are replaced by t, derive Y_i by a (GCON$-R$)-step of the form $T, t, q^{(n+1)}, W \Rightarrow T, W$, and continue $Y_{i+1}, \ldots, Y_l$. We apply the induction hypothesis. (II) $u = q^{(n+1)}$, $Y_{i-1} = Tp^{(m)}uW$, $Y_i = TW$ and $p^{(m)} \Rightarrow_R^* q^{(n)}$. Let $t = r^{(n')}$. We have $q^{(n)} \Rightarrow_R^* r^{(n'-1)}$, whence $p^{(m)} \Rightarrow_R^* r^{(n'-1)}$. The derivation $X_0, Y_1, \ldots, Y_l$ can be transformed into a shorter derivation: drop the first aplication of (PRO$-R$), derive $Y_1, \ldots, Y_{i-1}$ in which the designated occurrences of u are replaced by t, derive Y_i by a (GCON$-R$)-step of the form $T, p^{(m)}, r^{(n')}, W \Rightarrow T, W$, and continue $Y_{i+1}, \ldots, Y_l$. We apply the induction hypothesis. □

As a consequence, we obtain:

Lemma 1. *$R \vdash_{\mathbf{CBL}} t \Rightarrow u$ if, and only if, $t \Rightarrow_R^* u$.*

Proof. The 'if' part is obvious. The 'only if' part employs Theorem 2. Assume $R \vdash_{\mathbf{CBL}} t \Rightarrow u$. There exists a normal derivation of $t \Rightarrow u$ from R. The first step of this derivation cannot be (GCON$-R$), whence (GCON$-R$) is not applied at all; the last step cannot be (GEXP$-R$), whence (GEXP$-R$) cannot be applied at all. Consequently, each step of the derivation is a (PRO$-R$)-step (with X, Y empty). whence the derivation reduces to a single (PRO$-R$)-step. This yields $t \Rightarrow_R^* u$. □

Accordingly, LPPP amounts to verifying whether $t \Rightarrow_R^* u$, for any given R, t, u.

3 LPPP and Weighted Graphs

We reduce LPPP to a graph-theoretic problem. In the next section, the second problem is reduced to the emptiness problem for context-free languages. Both reductions are polynomial, and the third problem is solvable in polynomial time. This yields the polynomial time complexity of LPPP.

We define a finite weighted directed graph $G(R)$. $P(R)$ denotes the set of atoms occurring in promotions from R. The vertices of $G(R)$ are elements p_0, p_1, for all $p \in P(R)$. For any integer n, we set $\pi(n) = 0$, if n is even, and $\pi(n) = 1$, if n is odd. We also set $\pi^*(n) = 1 - \pi(n)$. For any promotion $p^{(m)} \Rightarrow q^{(n)}$ from R, $G(R)$ contains an arc from $p_{\pi(m)}$ to $q_{\pi(n)}$ with weight $n - m$ and an arc from $q_{\pi^*(n)}$ to $p_{\pi^*(m)}$ with weight $m - n$. Thus, each promotion from R gives rise to two weighted arcs in $G(R)$.

An arc from v to w of weight k is represented as the triple (v, k, w). As usual, a *route* from a vertex v to a vertex w in $G(R)$ is defined as a sequence of arcs $(v_0, k_1, v_1), \ldots, (v_{r-1}, k_r, v_r)$ such that $v_0 = v$, $v_r = w$, and the target of each but the last arc equals the source of the next arc. The *length* of this route is r, and its *weight* is $k_1 + \cdots + k_r$. We admit a trivial route from v to v of length 0 and weight 0.

Lemma 2. *If $p^{(m)} \Rightarrow_R q^{(n)}$, then $(p_{\pi(m)}, n - m, q_{\pi(n)})$ is an arc in $G(R)$.*

Proof. Assume $p^{(m)} \Rightarrow_R q^{(n)}$. We consider two cases.

(I) $m = m' + k$, $n = n' + k$, k is even, and $p^{(m')} \Rightarrow q^{(n')}$ belongs to R. Then $(p_{\pi(m')}, n' - m', q_{\pi(n')})$ is an arc in $G(R)$. We have $\pi(m) = \pi(m')$, $\pi(n) = \pi(n')$ and $n - m = n' - m'$, which yields the thesis.

(II) $m = m' + k$, $n = n' + k$, k is odd, and $q^{(n')} \Rightarrow p^{(m')}$ belongs to R. Then $(p_{\pi^*(m')}, n' - m', q_{\pi^*(n')})$ is an arc in $G(R)$. We have $\pi^*(m') = \pi(m)$, $\pi^*(n') = \pi(n)$ and $n - m = n' - m'$, which yields the thesis. □

Lemma 3. *Let $(v, r, q_{\pi(n)})$ be an arc in $G(R)$. Then, there is some $p \in P(R)$ such that $v = p_{\pi(n-r)}$ and $p^{(n-r)} \Rightarrow_R q^{(n)}$.*

Proof. We consider two cases.

(I) $(v, r, q_{\pi(n)})$ equals the arc $(p_{\pi(m')}, n'-m', q_{\pi(n')})$, and $p^{(m')} \Rightarrow q^{(n')}$ belongs to R. Then $r = n'-m'$ and $\pi(n) = \pi(n')$. We have $n = n'+k$, for an even integer k, whence $n - r = m' + k$. This yields $\pi(n - r) = \pi(m')$ and $p^{(n-r)} \Rightarrow_R q^{(n)}$.

(II) $(v, r, q_{\pi(n)})$ equals $(p_{\pi^*(m')}, n' - m', q_{\pi^*(n')})$, and $q^{(n')} \Rightarrow p^{(m')}$ belongs to R. Then $r = n' - m'$ and $\pi(n) = \pi^*(n')$. We have $n = n' + k$, for an odd integer k, whence $n-r = m'+k$. This yields $\pi(n-r) = \pi^*(m')$ and $p^{(n-r)} \Rightarrow_R q^{(n)}$. □

Theorem 3. *Let $p, q \in P(R)$. Then, $p^{(m)} \Rightarrow_R^* q^{(n)}$ if and only if there exists a route from $p_{\pi(m)}$ to $q_{\pi(n)}$ of weight $n - m$ in $G(R)$.*

Proof. The 'only if' part easily follows from Lemma 2. The 'if' part is proved by induction on the length of a route from $p_{\pi(m)}$ to $q_{\pi(n)}$ in $G(R)$, using Lemma 3. For the trivial route, we have $p = q$ and $n - m = 0$, whence $n = m$; so, the trivial derivation yields $p_{\pi(m)}^{(m)} \Rightarrow_R^* p_{\pi(m)}^{(m)}$. Assume that $(p_{\pi(m)}, r_1, v_1)$, (v_1, r_2, v_2), $\ldots, (v_k, r_{k+1}, q_{\pi(n)})$ is a route of length $k + 1$ and weight $n - m$ in $G(R)$. By Lemma 3, there exists $s \in P$ such that $v_k = s_{\pi(n-r_{k+1})}$ and $s^{(n-r_{k+1})} \Rightarrow_R q^{(n)}$. The weight of the initial subroute of length k is $n - m - r_{k+1}$, which equals $n - r_{k+1} - m$. By the induction hypothesis $p^{(m)} \Rightarrow_R^* s^{(n-r_{k+1})}$, which yields $p^{(m)} \Rightarrow_R^* q^{(n)}$. □

We return to LPPP. To verify whether $R \vdash p^{(m)} \Rightarrow q^{(n)}$ we consider two cases. If $p, q \in P(R)$, then, by Lemma 1 and Theorem 3, the answer is YES iff there exists a route in $G(R)$, as in Theorem 3. Otherwise, $R \vdash p^{(m)} \Rightarrow q^{(n)}$ iff $p = q$ and $m = n$.

4 Main Results

We have reduced LPPP to the following problem: given a finite weighted directed graph G with integer weights, two vertices v, w and an integer k, verify whether there exists a route from v to w of weight k in G. Caution: integers are represented in unary notation, e.g. 5 is the string of five digits.

We present a polynomial time reduction of this problem to the emptiness problem for context-free languages. Since a trivial route exists if and only if $v = w$ and $k = 0$, then we may restrict the problem to nontrivial routes.

First, the graph G is transformed into a non-deterministic FSA $M(G)$ in the following way. The alphabet of $M(G)$ is $\{+, -\}$. We describe the graph of $M(G)$. The states of $M(G)$ are vertices of G and some auxiliary states. If (v', n, w') is an arc in G, $n > 0$, then we link v' with w' by n transitions $v' \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n = w'$, all labeled by $+$, where $s_1, \ldots, s_{n-1}$ are new states; similarly for $n < 0$ except that now the transitions are labeled by $-$. For $n = 0$, we link v' with w' by two transitions $v' \rightarrow s \rightarrow w'$, the first one labeled by $+$, and the second one by $-$, where s is a new state. The final state is w. If $k = 0$, then v is the start state. If $k \neq 0$, then we add new states $i_1, \ldots, i_k$ with transitions $i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_k$ and $i_k \rightarrow v$, all labeled by $-$, if $k > 0$, and by $+$, if $k < 0$; the start state is i_1. The following equivalence is obvious: there exists a nontrivial route from v to w of weight k in G iff there exists a nontrivial route from the start state to the final state in $M(G)$ which visits as many pluses as minuses.

Let L be the context-free language, consisting of all nonempty strings on $\{+, -\}$ which contain as many pluses as minuses. The right-hand side of the above equivalence is equivalent to $L(M(G)) \cap L \neq \emptyset$.

A CFG for L consists of the following production rules: $S \mapsto SS$, $S \mapsto +S-$, $S \mapsto -S+$, $S \mapsto +-$, $S \mapsto -+$. We transform it to a CFG in the Chomsky Normal Form (i.e. all rules are of the form $A \mapsto BC$ or $A \mapsto a$) in a constant time. The latter is modified to a CFG for $L(M(G)) \cap L$ in a routine way. The new variables are of the form (q, A, q'), where q, q' are arbitrary states of $M(G)$, A is a variable of the former grammar. The initial symbol is (q_0, S, q_f), where q_0 is the start state and q_f the final state of $M(G)$. The new production rules are:

(1) $(q_1, A, q_3) \mapsto (q_1, B, q_2)(q_2, C, q_3)$ for any rule $A \mapsto BC$ of the former grammar,
(2) $(q_1, A, q_2) \mapsto a$, whenever $A \mapsto a$ is a rule of the former grammar, and $M(G)$ admits the transition from q_1 to q_2, labeled by $a \in \{+, -\}$.

The size of a graph G is defined as the sum of the following numbers: the number of vertices, the number of arcs, and the sum of absolute values of weights of arcs. The time of the construction of $M(G)$ is $O(n^2)$, where n is the size of G. A CFG for $L(M(G)) \cap L$ can be constructed in time $O(n^3)$, where n is the size of $M(G)$, defined as the number of transitions. The emptiness problem for a context-free language can be solved in time $O(n^2)$, where n is the size of the given CFG for the language, defined as the sum of the number of variables and the number of rules. Since the construction of $G(R)$ can be performed in linear time, we have proved the following theorem.

Theorem 4. *LPPP is solvable in polynomial time.*

As a consequence, the provability problem for **CBL** enriched with letter promotions (the word problem for pregroups with letter promotions) is solvable in polynomial time. First, $X \Rightarrow Y$ is derivable iff $X, Y^{(1)} \Rightarrow \epsilon$ is so. By Theorem 2, $X \Rightarrow \epsilon$ is derivable iff X can be reduced to ϵ by generalized contractions $Y, t, u, Z \Rightarrow Y, Z$ such that t, u appear in X and $t, u \Rightarrow \epsilon$ is derivable. The latter is equivalent to $t \Rightarrow_R^* u^{(-1)}$. By Theorem 4, the required instances of generalized contractions can be determined in polynomial time on the basis of R and X.

Corollary 1. *The word problem for pregroups with letter promotions is solvable in polynomial time.*

A pregroup grammar with letter promotions can be defined as a pregroup grammar in section 1 except that R is a finite set of letter promotions such that $P(R) \subseteq P$. $T^+(G)$ denotes the set of types appearing in I (of G) and $T(G)$ the set of terms occurring in the types from $T^+(G)$. One can compute all generalized contractions $t, u \Rightarrow \epsilon$, derivable from R in **CBL**, for arbitrary terms $t, u \in T(G)$. As shown in the above paragraph, this procedure is polynomial.

As in [3] for pregroup grammars, one can prove that pregroup grammars with letter promotions are equivalent to $\epsilon-$free context-free grammars. For $G = (\Sigma, P, I, s, R)$, one constructs a CFG G' (in an extended sense) in which the terminals are the terms from $T(G)$ and the nonterminals are the terminals and 1, the start symbol equals the principal type of G and the production rules are:

(P1) $u \mapsto t$, if $R \vdash_{CBL} t \Rightarrow u$,
(P2) $1 \mapsto t, u$, if $R \vdash_{CBL} t, u \Rightarrow \epsilon$,
(P3) $t \mapsto 1, t$ and $t \mapsto t, 1$, for any $t \in T(G)$.

By Theorem 2, G' generates precisely all strings $X \in (T(G))^+$ such that $R \vdash_{CBL} X \Rightarrow s$. $L(G) = f[g^{-1}[L(G')]]$, where $g : \Sigma \times T^+(G) \mapsto T^+(G)$ is a partial mapping, defined by $g((v, X)) = X$ whenever $(v, X) \in I$, and $f : \Sigma \times T^+(G) \mapsto \Sigma$ is a mapping, defined by $f((v, X)) = v$ (we extend f, g to homomorphisms of free monoids). Consequently, $L(G)$ is context-free, since the context-free languages are closed under homomorphisms and inverse homomorphisms.

Pregroup grammars with letter promotions can be transformed into equivalent context-free grammars in polynomial time (as in [5] for pregroup grammars), and the membership problem for the former is solvable in polynomial time. A parsing algorithm of complexity $O(n^3)$ can be designed, following the ideas of Oehrle [15] or Moroz [14]; see [13].

Acknowledgements. The polynomial time complexity of LPPP has been announced by the authors at the conference "Topology, Algebra and Categories in Logic", Amsterdam, 2009. The conference did not publish any proceedings.

References

1. Béchet, D., Foret, A.: Fully lexicalized pregroup grammars. In: Leivant, D., de Queiroz, R. (eds.) WoLLIC 2007. LNCS, vol. 4576, pp. 12–25. Springer, Heidelberg (2007)
2. Buszkowski, W.: Mathematical linguistics and proof theory. In: van Benthem, J., ter Meulen, A. (eds.) Handbook of Logic and Language, pp. 683–736. Elsevier Science B. V, Amsterdam (1997)
3. Buszkowski, W.: Lambek grammars based on pregroups. In: de Groote, P., Morrill, G., Retoré, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, pp. 95–109. Springer, Heidelberg (2001)
4. Buszkowski, W.: Sequent systems for compact bilinear logic. Mathematical Logic Quarterly 49(5), 467–474 (2003)

5. Buszkowski, W., Moroz, K.: Pregroup grammars and context-free grammars. In: Casadio, C., Lambek, J. (eds.) Computational Algebraic Approaches to Natural Language, Polimetrica, pp. 1–21 (2008)
6. Francez, N., Kaminski, M.: Commutation-augmented pregroup grammars and mildly context-sensitive languages. Studia Logica 87(2-3), 297–321 (2007)
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
8. Lambek, J.: Type grammars revisited. In: Lecomte, A., Perrier, G., Lamarche, F. (eds.) LACL 1997. LNCS (LNAI), vol. 1582, pp. 1–27. Springer, Heidelberg (1999)
9. Lambek, J.: Type grammars as pregroups. Grammars 4, 21–39 (2001)
10. Lambek, J.: From Word to Sentence: a computational algebraic approach to grammar. Polimetrica (2008)
11. Mater, A.H., Fix, J.D.: Finite presentations of pregroups and the identity problem. In: Proc. of Formal Grammar - Mathematics of Language, pp. 63–72. CSLI Publications, Stanford (2005)
12. Moortgat, M., Oehrle, R.T.: Pregroups and type-logical grammar: Searching for convergence. In: Casadio, C., Scott, P.J., Seely, R.A. (eds.) Language and Grammar. Studies in Mathematical Linguistics and Natural Language. CSLI Lecture Notes, vol. 168, pp. 141–160. CSLI Publications, Stanford (2005)
13. Moroz, K.: Algorithmic problems for pregroup grammars. PhD thesis, Adam Mickiewicz University, Poznań (2010)
14. Moroz, K.: A Savateev-style parsing algorithm for pregroup grammars. LNCS, vol. 5591. Springer, Heidelberg (to appear, 2010)
15. Oehrle, R.T.: A parsing algorithm for pregroup grammars. In: Categorial Grammars: an Efficient Tool for Natural Language Processing, pp. 59–75. University of Montpellier (2004)
16. Savateev, Y.: Unidirectional Lambek grammars in polynomial time. Theory of Computing Systems (to appear, 2010)
17. van Benthem, J.: Language in Action. Categories, Lambdas and Dynamic Logic. Studies in Logic and The Foundations of Mathematics. North-Holland, Amsterdam (1991)

A Hierarchical Classification of First-Order Recurrent Neural Networks

Jérémie Cabessa[1] and Alessandro E.P. Villa[1,2]

[1] GIN Inserm UMRS 836, University Joseph Fourier, FR-38041 Grenoble
[2] Faculty of Business and Economics, University of Lausanne, CH-1015 Lausanne
{jcabessa,avilla}@nhrg.org

Abstract. We provide a refined hierarchical classification of first-order recurrent neural networks made up of McCulloch and Pitts cells. The classification is achieved by first proving the equivalence between the expressive powers of such neural networks and Muller automata, and then translating the Wadge classification theory from the automata-theoretic to the neural network context. The obtained hierarchical classification of neural networks consists of a decidable pre-well ordering of width 2 and height ω^ω, and a decidability procedure of this hierarchy is provided. Notably, this classification is shown to be intimately related to the attractive properties of the networks, and hence provides a new refined measurement of the computational power of these networks in terms of their attractive behaviours.

1 Introduction

In neural computability, the issue of the computational power of neural networks has often been approached from the automata-theoretic perspective. In this context, McCulloch and Pitts, Kleene, and Minsky already early proved that the class of first-order recurrent neural networks discloses equivalent computational capabilities as classical finite state automata [5,7,8]. Later, Kremer extended this result to the class of Elman-style recurrent neural nets, and Sperduti discussed the computational power of different other architecturally constrained classes of networks [6,15].

Besides, the computational power of first-order recurrent neural networks was also proved to intimately depend on both the choice of the activation function of the neurons as well as the nature of the synaptic weights under consideration. Indeed, Siegelmann and Sontag showed that, assuming rational synaptic weights, but considering a saturated-linear sigmoidal instead of a hard-threshold activation function drastically increases the computational power of the networks from finite state automata up to Turing capabilities [12,14]. In addition, Siegelmann and Sontag also nicely proved that real-weighted networks provided with a saturated-linear sigmoidal activation function reveal computational capabilities beyond the Turing limits [10,11,13].

This paper concerns a more refined characterization of the computational power of neural nets. More precisely, we restrict our attention to the simple class of rational-weighted first-order recurrent neural networks made up of McCulloch and Pitts cells, and provide a refined classification of the networks of this class. The classification is achieved by first proving the equivalence between the expressive powers of such neural networks and Muller automata, and then translating the Wadge classification theory from the automata-theoretic to the neural network context [1,2,9,19]. The obtained hierarchical classification of neural networks consists of a decidable pre-well ordering of width 2 and height ω^ω, and a decidability procedure of this hierarchy is provided. Notably, this classification is shown to be intimately related to the attractive properties of the considered networks, and hence provides a new refined measurement of the computational capabilities of these networks in terms of their attractive behaviours.

2 The Model

In this work, we focus on synchronous discrete-time first-order recurrent neural networks made up of classical McCulloch and Pitts cells.

Definition 1. *A first-order recurrent neural network consists of a tuple $\mathcal{N} = (X, U, a, b, c)$, where $X = \{x_i : 1 \leq i \leq N\}$ is a finite set of N activation cells, $U = \{u_i : 1 \leq i \leq M\}$ is a finite set of M external input cells, and $a \in \mathbb{Q}^{N \times N}$, $b \in \mathbb{Q}^{N \times M}$, and $c \in \mathbb{Q}^{N \times 1}$ are rational matrices describing the weights of the synaptic connections between cells as well as the incoming background activity.*

The activation value of cells x_j and u_j at time t, respectively denoted by $x_j(t)$ and $u_j(t)$, is a boolean value equal to 1 if the corresponding cell is firing at time t and to 0 otherwise. Given the activation values $x_j(t)$ and $u_j(t)$, the value $x_i(t+1)$ is then updated by the following equation

$$x_i(t+1) = \sigma\left(\sum_{j=1}^{N} a_{i,j} \cdot x_j(t) + \sum_{j=1}^{M} b_{i,j} \cdot u_j(t) + c_i\right), \quad i = 1, \ldots, N \tag{1}$$

where σ is the classical hard-threshold activation function defined by $\sigma(\alpha) = 1$ if $\alpha \geq 1$ and $\sigma(\alpha) = 0$ otherwise.

Note that Equation (1) ensures that the whole dynamics of network $\mathcal{N}$ is described by the following governing equation

$$\boldsymbol{x(t+1)} = \sigma\left(a \cdot \boldsymbol{x(t)} + b \cdot \boldsymbol{u(t)} + c\right), \tag{2}$$

where $\boldsymbol{x(t)} = (x_1(t), \ldots, x_N(t))$ and $\boldsymbol{u(t)} = (u_1(t), \ldots, u_M(t))$ are boolean vectors describing the spiking configuration of the activation and input cells, and σ denotes the classical hard threshold activation function applied component by component. An example of such a network is given below.

Example 1. Consider the network $\mathcal{N}$ depicted in Figure 1. The dynamics of this network is then governed by the following equation:

$$\begin{pmatrix} x_1(t+1) \\ x_2(t+1) \\ x_3(t+1) \end{pmatrix} = \sigma\left[\begin{pmatrix} 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1(t) \\ x_2(t) \\ x_3(t) \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \cdot \begin{pmatrix} u_1(t) \\ u_2(t) \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \\ 0 \end{pmatrix}\right]$$

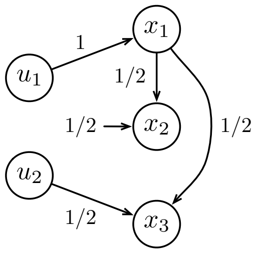


Fig. 1. A simple neural network

3 Attractors

The dynamics of recurrent neural networks made of neurons with two states of activity can implement an associative memory that is rather biological in its details [3]. In the Hopfield framework, stable equilibrium reached by the network that do not represent any valid configuration of the optimization problem are referred to as *spurious attractors*. According to Hopfield et al., spurious modes can disappear by "unlearning" [3], but Tsuda et al. have shown that rational successive memory recall can actually be implemented by triggering spurious modes [17]. Here, the notions of attractors, meaningful attractors, and spurious attractors are reformulated in our precise context. Networks will then be classified according to their ability to switch between different types of attractive behaviours. For this purpose, the following definitions need to be introduced.

As preliminary notations, for any $k > 0$, we let the space of k-dimensional boolean vectors be denoted by $\mathbb{B}^k$, and we let the space of all infinite sequences of k-dimensional boolean vectors be denoted by $[\mathbb{B}^k]^\omega$. Moreover, for any finite sequence of boolean vectors v, we let the expression $v^\omega = vvvv\cdots$ denote the infinite sequence obtained by infinitely many concatenations of v.

Now, let $\mathcal{N}$ be some network with N activation cells and M input cells. For each time step $t \geq 0$, the boolean vectors $\boldsymbol{x(t)} = (x_1(t), \ldots, x_N(t)) \in \mathbb{B}^N$ and $\boldsymbol{u(t)} = (u_1(t), \ldots, u_M(t)) \in \mathbb{B}^M$ describing the spiking configurations of both the activation and input cells of $\mathcal{N}$ at time t are respectively called the *state* of $\mathcal{N}$ at time t and the *input* submitted to $\mathcal{N}$ at time t. An *input stream* of $\mathcal{N}$ is then defined as an infinite sequence of consecutive inputs $s = (\boldsymbol{u(i)})_{i\in\mathbb{N}} = \boldsymbol{u(0)u(1)u(2)}\cdots \in [\mathbb{B}^M]^\omega$. Moreover, assuming the initial state of the network to be $\boldsymbol{x(0)} = \boldsymbol{0}$, any input stream $s = (\boldsymbol{u(i)})_{i\in\mathbb{N}} = \boldsymbol{u(0)u(1)u(2)}\cdots \in [\mathbb{B}^M]^\omega$ induces via Equation (2) an infinite sequence of consecutive states $e_s = (\boldsymbol{x(i)})_{i\in\mathbb{N}} = \boldsymbol{x(0)x(1)x(2)}\cdots \in [\mathbb{B}^N]^\omega$ that is called the *evolution* of $\mathcal{N}$ induced by the input stream s.

Along some evolution $e_s = \boldsymbol{x(0)x(1)x(2)}\cdots$, irrespective of the fact that this sequence is periodic or not, some state will repeat finitely often whereas other will repeat infinitely often. The (finite) set of states occurring infinitely often in the sequence e_s is denoted by $\inf(e_s)$. It can be observed that, for any evolution e_s, there exists a time step k after which the evolution e_s will necessarily remain confined in the set of states $\inf(e_s)$, or in other words, there exists an index k

such that $\boldsymbol{x(i)} \in \inf(e_s)$ for all $i \geq k$. However, along evolution e_s, the recurrent visiting of states in $\inf(e_s)$ after time step k does not necessarily occur in a periodic manner.

Now, given some network $\mathcal{N}$ with N activation cells, a set $A = \{\boldsymbol{y_0}, \ldots, \boldsymbol{y_k}\} \subseteq \mathbb{B}^N$ is called an *attractor* for $\mathcal{N}$ if there exists an input stream s such that the corresponding evolution e_s satisfies $\inf(e_s) = A$. Intuitively, an attractor can be seen a trap of states into which some network's evolution could become forever confined. We further assume that attractors can be of two distinct types, namely *meaningful* or *optimal* vs. *spurious* or *non-optimal*. In this study we do not extend the discussion about the attribution of the attractors to either type. From this point onwards, we assume any given network to be provided with the corresponding classification of its attractors into meaningful and spurious types.

Now, let $\mathcal{N}$ be some network provided with an additional type specification of each of its attractors. The complementary network $\mathcal{N}^{\complement}$ is then defined to be the same network as $\mathcal{N}$ but with an opposite type specification of its attractors.[1] In addition, an input stream s of $\mathcal{N}$ is called *meaningful* if $\inf(e_s)$ is a meaningful attractor, and it is called *spurious* if $\inf(e_s)$ is a spurious attractor. The set of all meaningful input streams of $\mathcal{N}$ is called the *neural language* of $\mathcal{N}$ and is denoted by $L(\mathcal{N})$. Note that the definition of the complementary network implies that $L(\mathcal{N}^{\complement}) = L(\mathcal{N})^{\complement}$. Finally, an arbitrary set of input streams $L \subseteq [\mathbb{B}^M]^\omega$ is defined as *recognizable* by some neural network if there exists a network $\mathcal{N}$ such that $L(\mathcal{N}) = L$. All preceding definitions are now illustrated in the next example.

Example 2. Consider again the network $\mathcal{N}$ described in Example 1, and suppose that an attractor is meaningful for $\mathcal{N}$ if and only if it contains the state $(1,1,1)^T$ (i.e. where the three activation cells simultaneously fire). The periodic input stream $s = [\binom{1}{1}\binom{1}{1}\binom{1}{1}\binom{0}{0}]^\omega$ induces the corresponding periodic evolution

$$e_s = \begin{pmatrix}0\\0\\0\end{pmatrix}\begin{pmatrix}1\\0\\0\end{pmatrix}\left[\begin{pmatrix}1\\1\\1\end{pmatrix}\begin{pmatrix}1\\1\\1\end{pmatrix}\begin{pmatrix}0\\1\\0\end{pmatrix}\begin{pmatrix}1\\0\\0\end{pmatrix}\right]^\omega .$$

Hence, $\inf(e_s) = \{(1,1,1)^T, (0,1,0)^T, (1,0,0)^T\}$, and the evolution e_s of $\mathcal{N}$ remains confined in a cyclic visiting of the states of $\inf(e_s)$ already from time step $t = 2$. Thence, the set $\{(1,1,1)^T, (0,1,0)^T, (1,0,0)^T\}$ is an attractor of $\mathcal{N}$. Moreover, this attractor is meaningful since it contains the state $(1,1,1)^T$.

4 Recurrent Neural Networks and Muller Automata

In this section, we provide an extension of the classical result stating the equivalence of the computational capabilities of first-order recurrent neural networks and finite state machines [5,7,8]. More precisely, here, the issue of the expressive power of neural networks is approached from the point of view of the theory of automata on infinite words, and it is proved that first-order recurrent neural

[1] More precisely, A is a meaningful attractor for $\mathcal{N}^{\complement}$ if and only if A is a spurious attractor for $\mathcal{N}$.

networks actually disclose the very same expressive power as finite Muller automata. Towards this purpose, the following definitions first need to be recalled.

A *finite Muller automaton* is a 5-tuple $\mathcal{A} = (Q, A, i, \delta, \mathcal{T})$, where Q is a finite set called the set of states, A is a finite alphabet, i is an element of Q called the initial state, δ is a partial function from $Q \times A$ into Q called the transition function, and $\mathcal{T} \subseteq \mathcal{P}(Q)$ is a set of set of states called the table of the automaton. A finite Muller automaton is generally represented by a directed labelled graph whose nodes and labelled edges respectively represent the states and transitions of the automaton.

Given a finite Muller automaton $\mathcal{A} = (Q, A, i, \delta, \mathcal{T})$, every triple (q, a, q') such that $\delta(q, a) = q'$ is called a *transition* of $\mathcal{A}$. A *path* in $\mathcal{A}$ is then a sequence of consecutive transitions $\rho = ((q_0, a_1, q_1), (q_1, a_2, q_2), (q_2, a_3, q_3), \ldots)$, also denoted by $\rho : q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} q_3 \cdots$. The path ρ is said to successively *visit* the states $q_0, q_1, \ldots$. The state q_0 is called the *origin* of ρ, the word $a_1 a_2 a_3 \cdots$ is the *label* of ρ, and the path ρ is said to be *initial* if $q_0 = i$. If ρ is an infinite path, the set of states visited infinitely often by ρ is denoted by $\inf(\rho)$. Besides, a *cycle* in $\mathcal{A}$ consists of a finite set of states c such that there exists a finite path in $\mathcal{A}$ with same origin and ending state that visits precisely all the sates of c. A cycle is called *successful* if it belongs to $\mathcal{T}$, and *non-succesful* otherwise. Moreover, an infinite initial path ρ of $\mathcal{A}$ is called *successful* if $\inf(\rho) \in \mathcal{T}$. An infinite word is then said to be *recognized* by $\mathcal{A}$ if it is the label of a successful infinite path in $\mathcal{A}$, and the *ω-language recognized by* $\mathcal{A}$, denoted by $L(\mathcal{A})$, is defined as the set of all infinite words recognized by $\mathcal{A}$. The class of all ω-languages recognizable by some Muller automata is precisely the class of *ω-rational languages*.

Now, for each ordinal $\alpha < \omega^\omega$, we introduce the concept of an α-alternating tree in a Muller automaton $\mathcal{A}$, which consists of a tree-like disposition of the successful and non-successful cycles of $\mathcal{A}$ induced by the ordinal α (see Figure 2). We first recall that any ordinal $0 < \alpha < \omega^\omega$ can uniquely be written of the form $\alpha = \omega^{n_p} \cdot m_p + \omega^{n_{p-1}} \cdot m_{p-1} + \ldots + \omega^{n_0} \cdot m_0$, for some $p \geq 0$, $n_p > n_{p-1} > \ldots > n_0 \geq 0$, and $m_i > 0$. Then, given some Muller automata $\mathcal{A}$ and some ordinal $\alpha = \omega^{n_p} \cdot m_p + \omega^{n_{p-1}} \cdot m_{p-1} + \ldots + \omega^{n_0} \cdot m_0 < \omega^\omega$, an α-alternating tree (resp. α-co-alternating tree) is a sequence of cycles of $\mathcal{A}$ $(C^{i,j}_{k,l})_{i \leq p, j < 2^i, k < m_i, l \leq n_i}$ such that: firstly, $C^{0,0}_{0,0}$ is successful (resp. not successful); secondly, $C^{i,j}_{k,l} \subsetneq C^{i,j}_{k,l+1}$, and $C^{i,j}_{k,l+1}$ is successful iff $C^{i,j}_{k,l}$ is not successful; thirdly, $C^{i,j}_{k+1,0}$ is strictly accessible from $C^{i,j}_{k,0}$, and $C^{i,j}_{k+1,0}$ is successful iff $C^{i,j}_{k,0}$ is not successful; fourthly, $C^{i+1,2j}_{0,0}$ and $C^{i+1,2j+1}_{0,0}$ are both strictly accessible from $C^{i,j}_{m_i-1,0}$, and each $C^{i+1,2j}_{0,0}$ is successful whereas each $C^{i+1,2j+1}_{0,0}$ is not successful. An α-alternating tree is said to be maximal in $\mathcal{A}$ if there is no β-alternating tree in $\mathcal{A}$ such that $\beta > \alpha$.

We now come up to the equivalence of the expressive power of recurrent neural networks and Muller automaton. First of all, we prove that any first-order recurrent neural network can be simulated by some Muller automaton.

Proposition 1. *Let $\mathcal{N}$ be a network provided with a type specification of its attractors. Then there exists a Muller automaton $\mathcal{A}_\mathcal{N}$ such that $L(\mathcal{N}) = L(\mathcal{A}_\mathcal{N})$.*

$$\begin{tikzcd}
 & & & & & C^{1,0}_{0,n_1} & C^{1,0}_{1,n_1} & & C^{1,0}_{m_1-1,n_1} & & \\
 & & & & & \vdots & \vdots & & \vdots & & \\
 & & & & & \rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & \\
 & & & & & C^{1,0}_{0,1} & C^{1,0}_{1,1} & & C^{1,0}_{m_1-1,1} & & \cdots \\
C^{0,0}_{0,n_0} & C^{0,0}_{1,n_0} & & C^{0,0}_{m_0-1,n_0} & & \rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & \\
\vdots & \vdots & & \vdots & & C^{1,0}_{0,0} \arrow[r] & C^{1,0}_{1,0} \arrow[r] & \cdots \arrow[r] & C^{1,0}_{m_1-1,0} \arrow[uurr] \arrow[ddrr] & & \\
\rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & & & & & & \\
C^{0,0}_{0,1} & C^{0,0}_{1,1} & & C^{0,0}_{m_0-1,1} & & & & & & & \cdots \\
\rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & & & & & & \\
C^{0,0}_{0,0} \arrow[r] & C^{0,0}_{1,0} \arrow[r] & \cdots \arrow[r] & C^{0,0}_{m_0-1,0} \arrow[uuuurr] \arrow[ddddddrr] & & & & & & & \\
 & & & & & C^{1,1}_{0,n_1} & C^{1,1}_{1,n_1} & & C^{1,1}_{m_1-1,n_1} & & \\
 & & & & & \vdots & \vdots & & \vdots & & \\
 & & & & & \rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & \\
 & & & & & C^{1,1}_{0,1} & C^{1,1}_{1,1} & & C^{1,1}_{m_1-1,1} & & \cdots \\
 & & & & & \rotatebox[origin=c]{90}{$\subsetneq$} & \rotatebox[origin=c]{90}{$\subsetneq$} & & \rotatebox[origin=c]{90}{$\subsetneq$} & & \\
 & & & & & C^{1,1}_{0,0} \arrow[r] & C^{1,1}_{1,0} \arrow[r] & \cdots \arrow[r] & C^{1,1}_{m_1-1,0} \arrow[uurr] \arrow[ddrr] & & \\
 & & & & & & & & & & \\
 & & & & & & & & & & \cdots
\end{tikzcd}$$

Fig. 2. The inclusion and accessibility relations between cycles in an α-alternating tree

Proof. Let $\mathcal{N}$ be given by the tuple (X, U, a, b, c), with $card(X) = N$, $card(U) = M$, and let the meaningful attractors of $\mathcal{N}$ be given by $A_1, \ldots, A_K$. Now, consider the Muller automaton $\mathcal{A}_{\mathcal{N}} = (Q, A, i, \delta, \mathcal{T})$, where $Q = \mathbb{B}^N$, $A = \mathbb{B}^M$, i is the N-dimensional zero vector, $\delta : Q \times A \to Q$ is defined by $\delta(\boldsymbol{x}, \boldsymbol{u}) = \boldsymbol{x'}$ if and only if $\boldsymbol{x'} = \sigma\left(a \cdot \boldsymbol{x} + b \cdot \boldsymbol{u} + c\right)$, and $\mathcal{T} = \{A_1, \ldots, A_K\}$. According to this construction, any input stream s of $\mathcal{N}$ is meaningful for $\mathcal{N}$ if and only if s is recognized by $\mathcal{A}_{\mathcal{N}}$. In other words, $s \in L(\mathcal{N})$ if and only if $s \in L(\mathcal{A}_{\mathcal{N}})$, and therefore $L(\mathcal{N}) = L(\mathcal{A}_{\mathcal{N}})$. □

According to the construction given in the proof of Proposition 1, any evolution of the network $\mathcal{N}$ naturally induces a corresponding infinite initial path in the Muller automaton $\mathcal{A}_{\mathcal{N}}$, and conversely, any infinite initial path in $\mathcal{A}_{\mathcal{N}}$ corresponds to some possible evolution of $\mathcal{N}$. This observation ensures the existence of a biunivocal correspondence between *the attractors* of the network $\mathcal{N}$ and *the cycles* in the graph of the corresponding Muller automaton $\mathcal{A}_{\mathcal{N}}$. Consequently, a procedure to compute all possible attractors of a given network $\mathcal{N}$ is simply obtained by first constructing the corresponding Muller automaton $\mathcal{A}_{\mathcal{N}}$ and then listing all cycles in the graph of $\mathcal{A}_{\mathcal{N}}$.

Conversely, we now prove that any Muller automaton can be simulated by some first-order recurrent neural network. For the sake of convenience, we choose to restrict our attention to Muller automata over the binary alphabet $\mathbb{B}^1$.

Proposition 2. *Let $\mathcal{A}$ be some Muller automaton over the alphabet $\mathbb{B}^1$. Then there exists a network $\mathcal{N}_{\mathcal{A}}$ such that $L(\mathcal{A}) = L(\mathcal{N}_{\mathcal{A}})$.*

Proof. Let $\mathcal{A}$ be given by the tuple $(Q, A, q_1, \delta, \mathcal{T})$, with $Q = \{q_1, \ldots, q_N\}$ and $\mathcal{T} \subseteq \mathcal{P}(Q)$. Now, consider the network $\mathcal{N}_{\mathcal{A}} = (X, U, a, b, c)$ defined as follows: First of all, $X = \{x_i : 1 \leq i \leq 2N\} \cup \{x'_1, x'_2, x'_3, x'_4\}$, $U = \{u_1\}$, and each state q_i in the automaton $\mathcal{A}$ gives rise to a two cell layer $\{x_i, x_{N+i}\}$ in the network $\mathcal{N}_{\mathcal{A}}$ as illustrated in Figure 3. Moreover, the synaptic weights between

u_1 and all activation cells, between all cells in $\{x'_1, x'_2, x'_3, x'_4\}$, as well as the background activity are precisely as depicted in Figure 3. Furthermore, for each $1 \leq i \leq N$, both cells x_i and x_{N+i} receive a weighted connection of intensity $\frac{1}{2}$ from cell x'_4 (resp. x'_2) if and only if $\delta(q_1, (0)) = q_i$ (resp. $\delta(q_1, (1)) = q_i$), as also shown in Figure 3. Farther, for each $1 \leq i, j \leq N$, there exist two weighted connection of intensity $\frac{1}{2}$ from cell x_i (resp. from cell x_{N+i}) to both cell x_j and x_{N+j} if and only if $\delta(q_i, (1)) = q_j$ (resp. $\delta(q_i, (0)) = q_j$), as partially illustrated in Figure 3 only for the k-th layer. This description of the network $\mathcal{N}_\mathcal{A}$ ensures that, for any possible evolution of $\mathcal{N}_\mathcal{A}$, the two cells x'_1 and x'_3 are firing at each time step $t \geq 1$, and furthermore, one and only one cell of $\{x_i : 1 \leq i \leq 2N\}$ are firing at each time step $t \geq 2$. According to this observation, for any $1 \leq j \leq N$, let $\mathbf{1}_j \in \mathbb{B}^{2N+4}$ (resp. $\mathbf{1}_{N+j} \in \mathbb{B}^{2N+4}$) denote the boolean vector describing the spiking configuration where only the cells x'_1, x'_3, and x_j (resp. x'_1, x'_3, and x_{N+j}) are firing. Hence, any evolution $\boldsymbol{x(0)x(1)x(2)}\cdots$ of $\mathcal{N}_\mathcal{A}$ satisfies $\boldsymbol{x(t)} \in \{\mathbf{1}_k : 1 \leq k \leq N\} \cup \{\mathbf{1}_{N+l} : 1 \leq l \leq N\}$ for all $t \geq 2$, and thus any attractor A of $\mathcal{N}$ can necessarily be written of the form $A = \{\mathbf{1}_k : k \in K\} \cup \{\mathbf{1}_{N+l} : l \in L\}$, for some $K, L \subseteq \{1, 2, \ldots, N\}$. Now, any infinite sequence $s = \boldsymbol{u(0)u(1)u(2)}\cdots \in [\mathbb{B}^1]^\omega$ induces both a corresponding infinite path $\rho_s : q_1 \xrightarrow{\boldsymbol{u(0)}} q_{j_1} \xrightarrow{\boldsymbol{u(1)}} q_{j_2} \xrightarrow{\boldsymbol{u(2)}} q_{j_3} \cdots$ in $\mathcal{A}$ as well as a corresponding evolution $e_s = \boldsymbol{x(0)x(1)x(2)}\cdots$ in $\mathcal{N}_\mathcal{A}$. The network $\mathcal{N}_\mathcal{A}$ is then related to the automaton $\mathcal{A}$ via the following important property: for each time step $t \geq 1$, if $\boldsymbol{u(t)} = (1)$, then $\boldsymbol{x(t+1)} = \mathbf{1}_{j_t}$, and if $\boldsymbol{u(t)} = (0)$, then $\boldsymbol{x(t+1)} = \mathbf{1}_{N+j_t}$. In other words, the infinite path ρ_s and the evolution e_s evolve in parallel and satisfy the property that the cell x_j is spiking in $\mathcal{N}_\mathcal{A}$ if and only if the automaton $\mathcal{A}$ is in state q_j and reads letter (1), and the cell x_{N+j} is spiking in $\mathcal{N}_\mathcal{A}$ if and only if the automaton $\mathcal{A}$ is in state q_j and reads letter (0). Finally, an attractor $A = \{\mathbf{1}_k : k \in K\} \cup \{\mathbf{1}_{N+l} : l \in L\}$ with $K, L \subseteq \{1, 2, \ldots, N\}$ is set to be meaningful if and only if $\{q_k : k \in K\} \cup \{q_l : l \in L\} \in \mathcal{T}$. Consequently, for any infinite infinite sequence $s \in [\mathbb{B}^1]^\omega$, the infinite path ρ_s in $\mathcal{A}$ satisfies $\inf(\rho_s) \in \mathcal{T}$

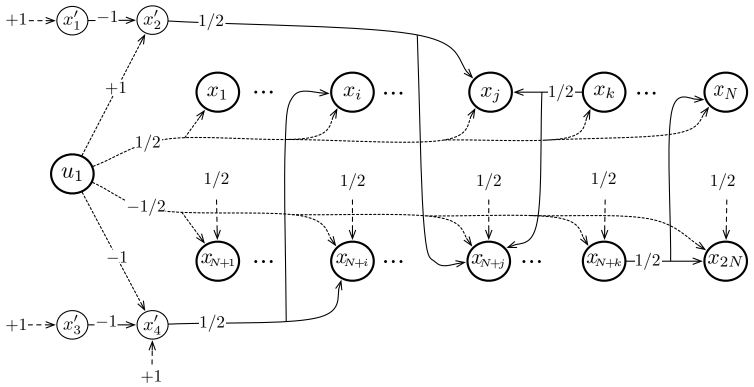


Fig. 3. The network $\mathcal{N}_\mathcal{A}$

if and only if the evolution e_s in $\mathcal{N}_{\mathcal{A}}$ is such that $\inf(e_s)$ is a meaningful attractor. Therefore, $L(\mathcal{A}) = L(\mathcal{N}_{\mathcal{A}})$. □

Finally, the following example provides an illustration of the two translating procedures described in the proofs of propositions 1 and 2.

Example 3. The translation from the network $\mathcal{N}$ described in Example 2 to its corresponding Muller automaton $\mathcal{A}_{\mathcal{N}}$ is illustrated in Figure 4. Proposition 1 ensures that $L(\mathcal{N}) = L(\mathcal{A}_{\mathcal{N}})$. Conversely, the translation from some given Muller automaton $\mathcal{A}$ over the alphabet $\mathbb{B}^1$ to its corresponding network $\mathcal{N}_{\mathcal{A}}$ is illustrated in Figure 5. Proposition 2 ensures that $L(\mathcal{A}) = L(\mathcal{N}_{\mathcal{A}})$.

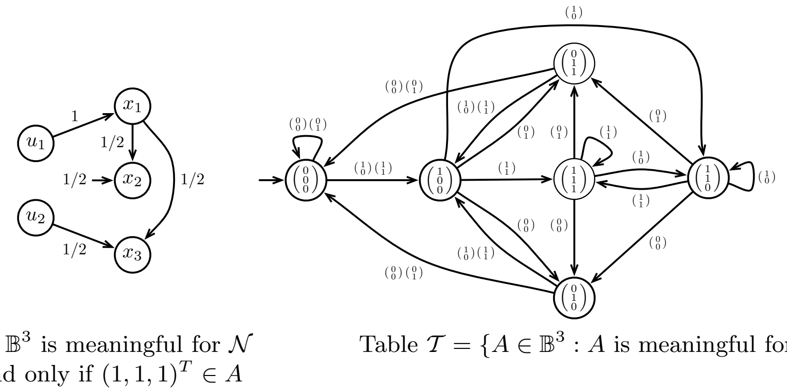

$A \subseteq \mathbb{B}^3$ is meaningful for $\mathcal{N}$ if and only if $(1,1,1)^T \in A$
Table $\mathcal{T} = \{A \in \mathbb{B}^3 : A$ is meaningful for $\mathcal{N}\}$

Fig. 4. Translation from a given network $\mathcal{N}$ provided with a type specification of its attractors to a corresponding Muller automaton $\mathcal{A}_{\mathcal{N}}$

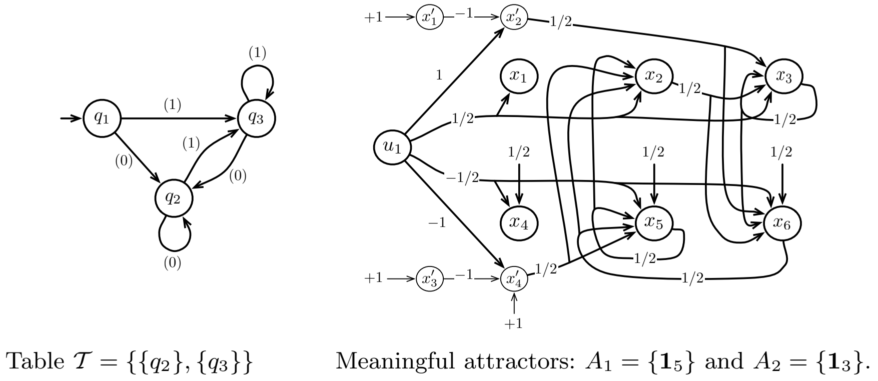


Table $\mathcal{T} = \{\{q_2\}, \{q_3\}\}$
Meaningful attractors: $A_1 = \{\mathbf{1}_5\}$ and $A_2 = \{\mathbf{1}_3\}$.

Fig. 5. Translation from a given Muller automaton $\mathcal{A}$ to a corresponding network $\mathcal{N}_{\mathcal{A}}$ provided with a type specification of its attractors

5 The RNN Hierarchy

In the theory of automata on infinite words, abstract machines are commonly classified according the topological complexity of their underlying ω-language, as for instance in [1,2,9,19]. Here, this approach is translated from the automata to the neural networks context, in order to obtain a refined classification of first-order recurrent neural networks. Notably, the obtained classification actually refers to the ability of the networks to switch between meaningful and spurious attractive behaviours.

For this purpose, the following facts and definitions need to be introduced. To begin with, for any $k > 0$, the space $[\mathbb{B}^k]^\omega$ can naturally be equipped with the product topology of the discrete topology over $\mathbb{B}^k$. Thence, a function $f : [\mathbb{B}^k]^\omega \to [\mathbb{B}^l]^\omega$ is said to be continuous if and only if the inverse image by f of every open set of $[\mathbb{B}^l]^\omega$ is an open set of $[\mathbb{B}^k]^\omega$. Now, given two first-order recurrent neural networks $\mathcal{N}_1$ and $\mathcal{N}_2$ with M_1 and M_2 input cells respectively, we say that $\mathcal{N}_1$ *Wadge reduces* [18] (or *continuously reduces* or simply *reduces*) to $\mathcal{N}_2$, denoted by $\mathcal{N}_1 \leq_W \mathcal{N}_2$, if any only if there exists a continuous function $f : [\mathbb{B}^{M_1}]^\omega \to [\mathbb{B}^{M_2}]^\omega$ such that any input stream s of $\mathcal{N}_1$ satisfies $s \in L(\mathcal{N}_1) \Leftrightarrow f(s) \in L(\mathcal{N}_2)$. The corresponding strict reduction, equivalence relation, and incomparability relation are then naturally defined by $\mathcal{N}_1 <_W \mathcal{N}_2$ iff $\mathcal{N}_1 \leq_W \mathcal{N}_2 \not\leq_W \mathcal{N}_1$, as well as $\mathcal{N}_1 \equiv_W \mathcal{N}_2$ iff $\mathcal{N}_1 \leq_W \mathcal{N}_2 \leq_W \mathcal{N}_1$, and $\mathcal{N}_1 \perp_W \mathcal{N}_2$ iff $\mathcal{N}_1 \not\leq_W \mathcal{N}_2 \not\leq_W \mathcal{N}_1$. Moreover, a network $\mathcal{N}$ is called *self-dual* if $\mathcal{N} \equiv_W \mathcal{N}^\complement$; it is *non-self-dual* if $\mathcal{N} \not\equiv_W \mathcal{N}^\complement$, which can be proved to be equivalent to saying that $\mathcal{N} \perp_W \mathcal{N}^\complement$. By extension, an $\equiv_W$-equivalence class of networks is called *self-dual* if all its elements are self-dual, and *non-self-dual* if all its elements are non-self-dual.

Now, the Wadge reduction over the class of neural networks naturally induces a hierarchical classification of networks. Formally, the collection of all first-order recurrent neural networks ordered by the Wadge reduction "$\leq_W$" is called the *RNN hierarchy*.

Propositions 1 and 2 ensure that the RNN hierarchy and the Wagner hierarchy – the collection of all ω-rational languages ordered by the Wadge reduction [19] – coincide up to Wadge equivalence. Accordingly, a precise description of the RNN hierarchy can therefore be given as follows. First of all, the RNN hierarchy is well founded, i.e. there is no infinite strictly descending sequence of networks $\mathcal{N}_0 >_W \mathcal{N}_1 >_W \mathcal{N}_2 >_W \ldots$. Moreover, the maximal strict chains in the RNN hierarchy have length ω^ω, meaning that the RNN hierarchy has a height of ω^ω. Furthermore, the maximal antichains of the RNN hierarchy have length 2, meaning that the RNN hierarchy has a width of 2.[2] More precisely, any two networks $\mathcal{N}_1$ and $\mathcal{N}_2$ satisfy the incomparability relation $\mathcal{N}_1 \perp_W \mathcal{N}_2$ if and only if $\mathcal{N}_1$ and $\mathcal{N}_2$ are non-self-dual networks such that $\mathcal{N}_1 \equiv_W \mathcal{N}_2^\complement$. These properties imply that, up to Wadge equivalence and complementation, the RNN

[2] A strict chain (resp. an antichain) in the RNN hierarchy is a sequence of neural networks $(\mathcal{N}_k)_{k\in\alpha}$ such that $\mathcal{N}_i <_W \mathcal{N}_j$ iff $i < j$ (resp. such that $\mathcal{N}_i \perp_W \mathcal{N}_j$ for all $i, j \in \alpha$ with $i \neq j$). A strict chain (resp. an antichain) is said to be maximal if its length is at least as large as the length of every other strict chain (resp. antichain).

hierarchy is actually a well-ordering. In fact, the RNN hierarchy consists of an alternating succession of non-self-dual and self-dual classes with pairs of non-self-dual classes at each limit level, as illustrated in Figure 6, where circle represent the Wadge equivalence classes of networks and arrows between circles represent the strict Wadge reduction between all elements of the corresponding classes. For convenience reasons, the degree of a network $\mathcal{N}$ in the RNN hierarchy is now defined in order to make the non-self-dual (n.s.d.) networks and the self-dual ones located just one level above share the same degree, as illustrated in Figure 6:

$$d(\mathcal{N}) = \begin{cases} 1 & \text{if } L(\mathcal{N}) = \emptyset \text{ or } \emptyset^{\complement}, \\ \sup\{d(\mathcal{M}) + 1 : \mathcal{M} \text{ n.s.d. and } \mathcal{M} <_W \mathcal{N}\} & \text{if } \mathcal{N} \text{ is non-self-dual}, \\ \sup\{d(\mathcal{M}) : \mathcal{M} \text{ n.s.d. and } \mathcal{M} <_W \mathcal{N}\} & \text{if } \mathcal{N} \text{ is self-dual}. \end{cases}$$

Also, the equivalence between the Wagner and RNN hierarchies ensure that the RNN hierarchy is actually decidable, in the sense that there exists a algorithmic procedure computing the degree of any network in the RNN hierarchy. All the above properties of the RNN hierarchy are summarized in the following result.

Theorem 1. *The RNN hierarchy is a decidable pre-well-ordering of width* 2 *and height* ω^ω.

Proof. The Wagner hierarchy consists of a decidable pre-well-ordering of width 2 and height ω^ω [19]. Propositions 1 and 2 ensure that the RNN hierarchy and Wagner hierarchy coincide up to Wadge equivalence. □

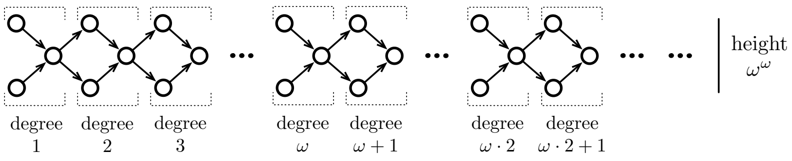


Fig. 6. The RNN hierarchy

The following result provides a detailed description of the decidability procedure of the RNN hierarchy. More precisely, it is shown that the degree of a network $\mathcal{N}$ in the RNN hierarchy corresponds precisely to the largest ordinal α such that there exists an α-alternating tree or an α-co-alternating tree in the Muller automaton $\mathcal{A}_{\mathcal{N}}$.

Theorem 2. *Let $\mathcal{N}$ be a network provided with a type specification of its attractors, $\mathcal{A}_{\mathcal{N}}$ be the corresponding Muller automaton of $\mathcal{N}$, and α be an ordinal such that* $0 < \alpha < \omega^\omega$.

- *If there exists in $\mathcal{A}_{\mathcal{N}}$ a maximal α-alternating tree and no maximal α-co-alternating tree, then $d(\mathcal{N}) = \alpha$ and $\mathcal{N}$ is non-self-dual.*

- *If there exists in $\mathcal{A}_{\mathcal{N}}$ a maximal α-co-alternating tree and no maximal α-alternating tree, then $d(\mathcal{N}) = \alpha$ and $\mathcal{N}$ is non-self-dual.*
- *If there exist in $\mathcal{A}_{\mathcal{N}}$ both a maximal α-alternating tree as well as a maximal α-co-alternating tree, then $d(\mathcal{N}) = \alpha$ and $\mathcal{N}$ is self-dual.*

Proof. For any ω-rational language L, let $d_W(L)$ denote the degree of L in the Wagner hierarchy. On the one hand, propositions 1 and 2 ensure that $d(\mathcal{N}) = d_W(L(\mathcal{A}_{\mathcal{N}}))$. On the other hand, the decidability procedure of the Wagner hierarchy states that $d_W(L(\mathcal{A}_{\mathcal{N}}))$ corresponds precisely to the largest ordinal α such that there exists a maximal α-(co)-alternating tree in $\mathcal{A}_{\mathcal{N}}$ [19]. $\square$

The decidability procedure of the degree of a network $\mathcal{N}$ in the the RNN hierarchy thus consists in first translating the network $\mathcal{N}$ into its corresponding Muller automaton $\mathcal{A}_{\mathcal{N}}$ (as described in Proposition 1), and then returning the ordinal α associated to the maximal α-(co)-alternating tree(s) in contained in $\mathcal{A}_{\mathcal{N}}$ (which can be achieved by some graph analysis of the automaton $\mathcal{A}_{\mathcal{N}}$). In other words, the complexity of a network $\mathcal{N}$ is directly related to the relative disposition of the successful and non-successful cycles in the Muller automaton $\mathcal{A}_{\mathcal{N}}$, or in other words, to how some infinite path in $\mathcal{A}_{\mathcal{N}}$ could maximally alternate between successful and non-successful cycles along its evolution. Therefore, according to the biunivocal correspondence between cycles in $\mathcal{A}_{\mathcal{N}}$ and attractors of $\mathcal{N}$, as well as between infinite paths in $\mathcal{A}_{\mathcal{N}}$ and evolutions of the network $\mathcal{N}$, it follows that the complexity of a network $\mathcal{N}$ in the RNN hierarchy actually refers to the capacity of this network to maximally alternate between punctual visitings of meaningful and spurious attractors along some possible evolution – a concept close to chaotic itinerancy [16,4].

Example 4. Let $\mathcal{N}$ be the network of Example 2. Then $d(\mathcal{N}) = \omega$ and $\mathcal{N}$ is non-self-dual. Indeed, $\{(0,0,0)^T\} \subsetneq \{(0,0,0)^T, (1,0,0)^T, (1,1,1)^T, (0,1,1)^T\}$ is a maximal ω^1-co-alternating tree in the Muller automaton $\mathcal{A}_{\mathcal{N}}$ of Figure 4.

6 Conclusion

The present work proposes a new approach of neural computability from the point of view infinite word reading automata theory. More precisely, the Wadge classification of infinite word languages is translated from the automata-theoretic to the neural network context, and a transfinite decidable hierarchical classification of first-order recurrent neural network is obtained. This classification provides a better understanding of this simple class of neural networks that could be relevant for implementation issues. Moreover, the Wadge hierarchies of deterministic pushdown automata or deterministic Turing Machines both with Muller conditions [1,9] ensure that such Wadge-like classifications of strictly more powerful models of neural networks could also be described; however, in these cases, the decidability procedures of the obtained hierarchies remain hard open problems.

Besides, this work is envisioned to be extended in several directions. First of all, it could be of interest to study the same kind of hierarchical classification applied to more biologically oriented models, like neural networks provided with some additional simple STDP rule. In addition, neural networks' computational capabilities should also rather be approached from the point of view of finite word instead of infinite word reading automata, as for instance in [6,10,11,12,13,14,15]. Unfortunately, as opposed to the case of infinite words, the classification theory of finite words reading machines is still a widely undeveloped, yet promising issue. Finally, the study of hierarchical classifications of neural networks induced by more biologically oriented reduction relations than the Wadge reduction would be of specific interest.

References

1. Duparc, J.: A hierarchy of deterministic context-free ω-languages. Theor. Comput. Sci. 290(3), 1253–1300 (2003)
2. Finkel, O.: An effective extension of the Wagner hierarchy to blind counter automata. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 369–383. Springer, Heidelberg (2001)
3. Hopfield, J.J., Feinstein, D.I., Palmer, R.G.: 'unlearning' has a stabilizing effect in collective memories. Nature 304, 158–159 (1983)
4. Kaneko, K., Tsuda, I.: Chaotic itinerancy. Chaos 13(3), 926–936 (2003)
5. Kleene, S.C.: Representation of events in nerve nets and finite automata. In: Automata Studies. Annals of Mathematics Studies, vol. 34, pp. 3–42. Princeton University Press, Princeton (1956)
6. Kremer, S.C.: On the computational power of elman-style recurrent networks. IEEE Transactions on Neural Networks 6(4), 1000–1004 (1995)
7. McCulloch, W.S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bulletin of Mathematical Biophysic 5, 115–133 (1943)
8. Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River (1967)
9. Selivanov, V.: Wadge degrees of ω-languages of deterministic Turing machines. Theor. Inform. Appl. 37(1), 67–83 (2003)
10. Siegelmann, H.T.: Computation beyond the Turing limit. Science 268(5210), 545–548 (1995)
11. Siegelmann, H.T.: Neural and super-Turing computing. Minds Mach. 13(1), 103–114 (2003)
12. Siegelmann, H.T., Sontag, E.D.: Turing computability with neural nets. Applied Mathematics Letters 4(6), 77–80 (1991)
13. Siegelmann, H.T., Sontag, E.D.: Analog computation via neural networks. Theor. Comput. Sci. 131(2), 331–360 (1994)
14. Siegelmann, H.T., Sontag, E.D.: On the computational power of neural nets. J. Comput. Syst. Sci. 50(1), 132–150 (1995)
15. Sperduti, A.: On the computational power of recurrent neural networks for structures. Neural Netw. 10(3), 395–400 (1997)
16. Tsuda, I.: Chaotic itinerancy as a dynamical basis of hermeneutics of brain and mind. World Futures 32, 167–184 (1991)
17. Tsuda, I., Koerner, E., Shimizu, H.: Memory dynamics in asynchronous neural networks. Prog. Th. Phys. 78(1), 51–71 (1987)
18. Wadge, W.W.: Reducibility and determinateness on the Baire space. PhD thesis, University of California, Berkeley (1983)
19. Wagner, K.: On ω-regular sets. Inform. and Control 43(2), 123–177 (1979)

Choosing Word Occurrences for the Smallest Grammar Problem*

Rafael Carrascosa[1], François Coste[2], Matthias Gallé[2], and Gabriel Infante-Lopez[1,3]

[1] Grupo de Procesamiento de Lenguaje Natural, Universidad Nacional de Córdoba, Argentina
[2] Symbiose Project, IRISA/INRIA Rennes-Bretagne Atlantique, France
[3] Consejo Nacional de Investigaciones Científicas, Argentina

Abstract. The smallest grammar problem - namely, finding a smallest context-free grammar that generates exactly one sequence - is of practical and theoretical importance in fields such as Kolmogorov complexity, data compression and pattern discovery. We propose to focus on the choice of the occurrences to be rewritten by non-terminals. We extend classical offline algorithms by introducing a global optimization of this choice at each step of the algorithm. This approach allows us to define the search space of a smallest grammar by separating the choice of the non-terminals and the choice of their occurrences. We propose a second algorithm that performs a broader exploration by allowing the removal of useless words that were chosen previously. Experiments on a classical benchmark show that our algorithms consistently find smaller grammars then state-of-the-art algorithms.

1 Introduction

The smallest grammar problem - namely, finding a smallest context-free grammar that generates exactly one sequence - is of practical and theoretical importance in fields such as Kolmogorov complexity, data compression and pattern discovery.

The size of a smallest grammar can be considered a computable variant of Kolmogorov complexity, in which the Turing machine description of the sequence is restricted to context-free grammars. The problem is then decidable, but still hard: the problem of finding a smallest grammar with an approximation ratio smaller then $\frac{8569}{8568}$ is NP-HARD [4]. Nevertheless, an $O(\log^3 n)$ approximation ratio - with n the length of the sequence - can be achieved by a simple algorithmic scheme based on approximation to the shortest superstring problem [4] and a smaller $O(\log n/g)$ (where g is the size of a smallest grammar) approximation ratio is possible by more complex mappings from the LZ77-factorization of the sequence to a context-free grammar with a balanced parsing tree [4,15].

* The work described in this paper is partially supported by the Program of International Scientific Cooperation MINCyT - INRIA/CNRS.

If the grammar is small, storing the grammar instead of the sequence can be interesting from a data compression perspective. Kieffer and Yang developed the formal framework of compression by *Grammar Based Codes* from the viewpoint of information theory, defining irreducible grammars and demonstrating their universality [6]. Before this formalization, several algorithms allowing to compress a sequence by context-free grammars had already been proposed. The LZ78-factorization introduced by Ziv and Lempel in [21] can be interpreted as a context-free grammar. Let us remark that this is not true for LZ77, published one year before [20]. Moreover, it is a commonly used result that the size of a LZ77-factorization is a lower bound on the size of a smallest grammar [15,4]. The first approach that generated explicitly a context-free grammar with compression ability is *Sequitur* [13]. Like LZ77 and LZ78, Sequitur is an on-line algorithm that processes the sequence from left to right. It maintains incrementally a grammar generating the part of the sequence read, introducing and deleting rewriting rules to ensure that no digram (pair of adjacent symbols) occurs more than once and that each rule is used at least twice. Other algorithms consider the entire sequence before choosing which repeat will be rewritten by the introduction of a new rule. Most of these offline algorithms proceed in a greedy manner. First the grammar is initialized by a unique initial rule $S \rightarrow s$ where s is the input sequence. Then they proceed iteratively, selecting in each iteration one repeated word w according to a score function and replacing all the (non-overlapping) occurrences of the repeat w in the whole grammar by a new terminal N and adding the new rewriting rule $N \rightarrow w$ to the grammar. Different heuristics have been used to choose the repeat: Frequent [19] chooses the most frequently-occurring digram, Long [3] chooses the longest word while Compressive [12] chooses the word that reduce at most the size of the resulting grammar. Greedy [1] belongs also to this family but the score used for choosing the words is oriented toward directly optimizing the number of bits needed to encode the grammar rather than minimizing its size. The running time of Sequitur is linear and linear-time versions of Frequent and Long have been introduced in [9] and [11] respectively, while the existence of a linear-time algorithm for Compressive and Greedy remains an open question.

In pattern discovery, a smallest grammar is a good candidate for being the one that generates the data according to Occam's razor principle. In that case, the grammar may not only be used for compressing the sequence but also to unveil its structure. Inference of the hierarchical structure of sequences was the initial motivation of Sequitur and has been the subject of several papers applying this scheme to DNA sequences [13,8,5], musical scores [14] or natural language [19,10]. It can also be a first step to learn more general grammars along the lines of [16]. In all the latter cases, a slight difference in the size of the grammar, which would not matter for data compression, can dramatically change the results with respect to the structure and more sophisticated algorithms than those for data compression are needed. In this article, we focus on how to choose occurrences that are going to be rewritten. This mechanism is generally handled straightforwardly in these papers and consists of selecting *all* the non-overlapping occurrences in a left to right order. Moreover, once an occurrence has been chosen for being rewritten, the result is definitive and is not altered by the words that will be chosen in the following iterations. In order to remedy these flaws, we show how to globally optimize the choice of the occurrences to be replaced by non-terminals. We are then able to improve classical greedy algorithms by introducing this optimization step at each iteration of the algorithm. This optimization allows us to separate the choice of the non-terminals and the choice of their occurrences. We redefine the search space and we propose a new procedure performing a wider search by adding the possibility to discard non-terminals previously included in the grammar.

The outline of this paper is the following: in Sect. 2 we introduce formally the definitions and the classical offline algorithms. Section 3 contains our contributions: in Sect. 3.1 we show how to optimize the choice of occurrences to be replaced by non-terminals for a set of words and extend offline algorithms by optimizing the choice of the occurrences at each step. We show that this optimization can also be used directly to guide the search in a new algorithm in Sect. 3.3. We present experiments on a classical benchmark in Sect. 4 showing that the occurrence optimization consistently allows to find smaller grammars and Sect. 5 concludes the paper.

2 Iterative Repeat Replacement Algorithms

2.1 Definitions and Notation

We start by giving a few definitions and setting up the nomenclature that we use along the paper. A string s is a sequence of characters $s_1 \dots s_n$, its length, $|s| = n$. ϵ denotes the empty word, and $s[i:j] = s_i \dots s_j$, $s[i:j] = \epsilon$ if $j < i$. A context-free grammar is a tuple $\langle \Sigma, \mathcal{N}, \mathcal{P}, S \rangle$, where Σ is the set of *terminals*, $\mathcal{N}$ is the set of *non-terminals* and $\mathcal{N}$ and Σ are disjoint. $S \in \mathcal{N}$ is called the *start symbol* and $\mathcal{P}$ is a set of *productions*. Each production is of the form $A \rightarrow \alpha$ where its *left-hand side* A is a non-terminal and its *right-hand side* α belongs to $(\Sigma \cup \mathcal{N})^*$. β can be derived from α, denoted by $\alpha \Rightarrow \beta$, if there exists a set of production rules allowing to produce β starting from α. The language defined by a grammar is the set of words $\{w \in \Sigma^* : S \Rightarrow w\}$.

Several definitions of the grammar size exist. Following [12], we define the *size of the grammar* G, denoted by $|G|$, to be the length of its encoding by concatenation of its right-hand sides separated by end-of-rule markers: $|G| = \sum_{A \rightarrow \alpha \in \mathcal{P}} (|\alpha| + 1)$.

2.2 General Scheme

Most offline algorithms follow the same general scheme. First, the grammar is initialized with a unique initial rule $S \rightarrow s$ where s is the input sequence and then they proceed iteratively. At each iteration, a word ω occurring more than once in s is chosen according to a score function f, all the (non-overlapping) occurrences of ω in the grammar are replaced by a new non-terminal N_ω and a new rewriting rule $N_\omega \to \omega$ is added to the grammar. We give pseudo-code for this general scheme that we name *Iterative Repeat Replacement* (IRR) in Algorithm 1. There, $\mathcal{P}$ is the set of production rules being built: this defines a unique grammar $G(\mathcal{P})$ and therefore we define $|\mathcal{P}| = |G(\mathcal{P})|$. The set of repeated words in the right-hand side of $\mathcal{P}$ is denoted by repeats($\mathcal{P}$) and $\mathcal{P}_{\omega \mapsto N}$ is the result of the substitution of ω by the new symbol N in the right-hand sides of $\mathcal{P}$ as detailed in the next paragraph.

When occurrences overlap, one has to specify which occurrences have to be replaced. One solution is to choose all the elements in the *canonical list of non-overlapping occurrences* of ω in s, which we define to be the list of non-overlapping occurrences of ω in a greedy left to right way (all occurrences overlapping with the first selected occurrence are not considered, then the same thing with the second non-eliminated occurrence, etc). This ensures that a maximal number of occurrences will be replaced. When searching for the smallest grammar, one has to consider not only the occurrences of a word in s but also their occurrence in right-hand sides of rules that are currently part of the grammar. A canonical list of non-overlapping occurrences of ω can be defined for each right-hand side appearing in the set of production rules $\mathcal{P}$. This provides a straightforward list of occurrences used in the scoring function or the replacement step by our pseudo-code defining IRR: $\mathcal{P}_{\omega \mapsto N}$ denotes the result of substituting by N these right-hand side occurrences of ω in $\mathcal{P}$.

Algorithm 1. Iterative Repeat Replacement

```
IRR($s$, $f$)
Require: $s$ is a sequence, and $f$ is a score function on words
1: $\mathcal{N} \leftarrow \{N_s\}$
2: $\mathcal{P} \leftarrow \{N_s \to s\}$
3: while $\exists\omega \; : \; \omega \leftarrow \arg\max_{\alpha \in \text{repeats}(\mathcal{P})} f(\alpha, \mathcal{P}) \wedge |\mathcal{P}_{\omega \mapsto N_\omega}| < |\mathcal{P}|$ do
4:     $\mathcal{N} \leftarrow \mathcal{N} \cup \{N_\omega\}$
5:     $\mathcal{P} \leftarrow \mathcal{P}_{\omega \mapsto N_\omega} \cup \{N_\omega \to \omega\}$
6: end while
7: return $G(\mathcal{P})$
```

The IRR scheme enables us to compare in a uniform framework the behavior of different score functions f that are used in the classical algorithms for choosing the words to replace. We implemented the scores of the most popular offline algorithms (or their extension to include right-hand sides, if that was not considered originally). IRR-MO maximizes the number of non-overlapping occurrences, it uses $f(\alpha, \mathcal{P}) = o$, where o is the size of the canonical non-overlapping list of α in the right-hand sides of rules in $\mathcal{P}$ (Frequent [19] or Re-Pair [9]); IRR-ML selects the largest repeated word: $f(\alpha, \mathcal{P}) = |\alpha|$ (Long [3] LFS, or LFS2 [11]); and IRR-MC minimizes the size of the grammar by maximizing $f(\alpha, \mathcal{P}) = o * |\alpha| - o - |\alpha| - 1$ (Compressive [12]). The complexity of IRR when it uses one of these scores is $O(n^3)$ since for a sequence of size n, the computation of the scores involving only o and $|\alpha|$ of the $O(n^2)$ possible repeats can be done in $O(n^2)$ using a suffix tree structure and the number of iterations is bounded by n since the size of the grammar decreases at each step.

The grammars found by these algorithms, Sequitur and LZ78 on a small example are shown in Fig. 1 while a comparison of the size of the grammars returned by these algorithms over a standard data compression corpus are presented in Sect. 4. These experiments confirm that IRR-MC is the best of these practical heuristics for finding smaller grammars as was suggested in [12]. Even when theoretical algorithms were designed to achieve a low approximation ratio [4,15,17], until now it could not be proven (theoretically or empirically) that they perform better than IRR-MC.

$$\begin{array}{ll} S & \rightarrow N_1\text{dabge}N_1\text{e}N_1\text{d\$} \\ N_1 & \rightarrow \text{a b c} \end{array}$$

IRR-MC

$$\begin{array}{ll} S & \rightarrow N_2\text{d}N_1\text{ge}N_2\text{e}N_2\text{d\$} \\ N_1 & \rightarrow \text{a b} \\ N_2 & \rightarrow N_1 \text{ c} \end{array}$$

IRR-MO

$$\begin{array}{ll} S & \rightarrow N_1\text{abge}N_2\text{e}N_1\text{\$} \\ N_1 & \rightarrow N_2 \text{ d} \\ N_2 & \rightarrow \text{a b c} \end{array}$$

IRR-ML

$$\begin{array}{ll} S & \rightarrow N_1\text{d}N_2\text{g}N_3N_3\text{d\$} \\ N_1 & \rightarrow N_2 \text{ c} \\ N_2 & \rightarrow \text{a b} \\ N_3 & \rightarrow \text{e } N_1 \end{array}$$

Sequitur

$$\begin{array}{llll} \multicolumn{4}{l}{S \rightarrow N_1N_2N_3N_4N_5N_6N_7N_8N_9N_{10}N_{11}} \\ N_1 & \rightarrow \text{a} & N_7 & \rightarrow \text{e} \\ N_2 & \rightarrow \text{b} & N_8 & \rightarrow N_5 \text{ c} \\ N_3 & \rightarrow \text{c} & N_9 & \rightarrow N_7 \text{ a} \\ N_4 & \rightarrow \text{d} & N_{10} & \rightarrow N_2 \text{ c} \\ N_5 & \rightarrow N_1 \text{ b} & N_{11} & \rightarrow \text{d \$} \\ N_6 & \rightarrow \text{g} & & \end{array}$$

LZ78

Fig. 1. Grammars returned by classical algorithms on sequence *abcdabgeabceabcd*\$

2.3 Limitations of IRR

Even though IRR algorithms are the best known practical algorithms for obtaining small grammars, they present some weaknesses. In the first place, their greedy strategy does not guarantee that the compression gain introduced by a selected word ω will still be interesting in the final grammar. Each time a future iteration selects a substring of ω, the length of the production rule is reduced; and each time a superstring is selected, its number of occurrences is reduced. Moreover, the first choices mark some breaking points and future words may appear inside them or in another parts of the grammar, but never cross them.

One could argue that it could be possible to find a score function that considers probable choices in the future. Nevertheless, a third weakness is intrinsic to IRR and does not depend on the score function: consider the sequence $xaxbxcx\#_1xbxcxax\#_2xcxaxbx\#_3xaxcxbx\#_4xbxaxcx\#_5xcxbxax\#_6xax\#_7xbx\#_8xcx$, where each $\#_i$ acts as a separator over which no repeat spans. This sequence exploits the fact that IRR algorithms replace all possible occurrences of the selected word. Let us define G^* as the following grammar:

$$\begin{array}{l} S \rightarrow AbC\#_1BcA\#_2CaB\#_3AcB\#_4BaC\#_5CbA\#_6A\#_7B\#_8C \\ A \rightarrow xax \quad B \rightarrow xbx \quad C \rightarrow xcx \end{array}$$

$|G^*| = 42$. Note that no IRR algorithm could generate G^* and, moreover, the smallest possible grammar that can be obtained with an IRR algorithm has size 46, resulting in an approximation ratio of 1.095. This is a general lower bound for *any* IRR algorithm.

In order to find G^*, the choice of occurrences that will be rewritten should be flexible when considering repeats introduced in future iterations.

3 Optimization of the Occurrences Choice

3.1 Global Optimization of Occurrences Replacement

Once an IRR algorithm has chosen a repeated word ω, it replaces all non-overlapping occurrences of that word in the current grammar by a new non-terminal N and then adds $N \to \omega$ to the set of production rules. In this section, we propose to perform a global optimization of the replacement of occurrences, considering not only the last non-terminal but also all the previously introduced non-terminals. The idea is to allow occurrences of words to be kept (instead of being replaced by non-terminals) if replacing other occurrences of words overlapping them results in a better compression.

Let $\mathcal{N}$ denote the set of non-terminals that can be used for replacing occurrences. Let us remark that each non-terminal N introduced to replace a word ω is not limited to replace ω but can replace any word with the same yield where we define the *yield* of a word $\alpha \in (\Sigma \cup N)^*$ by: $yield(\alpha) = \{w \in \Sigma^* / \alpha \Rightarrow w\}$. For instance, given the set of non-terminals $\mathcal{N}$ and their respective yields (defined at the moment of the introduction of the rule $N \to \alpha$ by $yield_N \leftarrow yield(\alpha)$), we can search for the best replacement in the sequence s by non-terminals of $\mathcal{N} \setminus \{S\}$ such that the replacement results in a minimal sequence s' and the yield of s' is s. This result would provide us with a minimal rule $N_s \to s'$ producing s and assuming the production of their yield by other non-terminals, which can also in turn be minimized by the same kind of optimization. This problem is related to the problem of static dictionary parsing [18] with the difference that the dictionary also has to be parsed. It can be formalized here as searching for the smallest grammar with a set of production rules of the form $\{N \to \alpha / N \in \mathcal{N}, \alpha \in (\mathcal{N} \cup \Sigma)^*, yield(\alpha) = yield_N\}$, the set of non-terminals $\mathcal{N}$ and their respective yields being given.

This problem can be solved in a classical way by searching for the shortest path in $|\mathcal{N}|$ graphs as follows. For each non-terminal $N \in \mathcal{N}$, we introduce a directed labeled acyclic graph Γ_N. To lighten the notation, we assume that the yield of N can be written as $yield_N = y_1 \ldots y_k$. Then, we define the graph Γ_N to have $k+1$ nodes, namely $\{1 \ldots k+1\}$, and edge from node i to node $i+1$ labeled with y_i for each i, and an edge from node i to $j+1$ labeled by M if there exists a non-terminal M different from N such that $y[i:j] = yield_M$. Intuitively, an edge from node i to node $j+1$ with label M represents a possible replacement of the occurrence $y[i:j]$ by M. Searching for the smallest path from state 1 to state $k+1$ with a classical dynamic programming algorithm allows us to find the smallest α such that $\alpha \Rightarrow yield_N$. This procedure is done for each non-terminal. We denote hereafter $\mathcal{P}_{min}(\mathcal{N})$ the set of rules obtained by this optimization.

3.2 IRR with Occurrence Optimization

We can now define the variant of IRR, called *Iterative Repeat Choice with Occurrence Optimization* (IRCOO) with the pseudo-code given in Algorithm 2. The smallest path algorithm has complexity $O(k \times m)$ for a graph Γ_N, where k is the number of nodes of Γ_N $(= |yield(N)|)$ and $m = |\mathcal{N}|$. k is bounded by $|s| = n$, so the complexity of computing $\mathcal{P}_{min}$ is $O(n \times m^2)$. The computation of the arg max depends only on the number of repeats, assuming that f is constant, so that its complexity lies in $O(n^2)$. Like for IRR, the total number of times the while loop is executed is bounded by n. The complexity of this generic scheme is thus $O(n \times (n^2 + n \times m^2))$.

Algorithm 2. Iterative Repeat Choice with Occurrences Optimization

```
IRCOO($s$, $f$)
Require: $s$ is a sequence, and $f$ is a score function on words
 1: $\mathcal{N} \leftarrow \{N_s\}$
 2: $\mathcal{P} \leftarrow \{N_s \rightarrow s\}$
 3: while $(\exists\omega : \omega \leftarrow \text{argmax}_{\alpha \in \text{repeats}(\mathcal{P})} f(\alpha, \mathcal{P})) \wedge |\mathcal{P}_{min}(\mathcal{N} \cup \{N_{yield(\omega)}\})| < |\mathcal{P}|$ do
 4:     $\mathcal{N} \leftarrow \mathcal{N} \cup \{N_{yield(\omega)}\}$
 5:     $\mathcal{P} \leftarrow \mathcal{P}_{min}(\mathcal{N})$
 6: end while
 7: return $G(\mathcal{P})$
```

As an example, consider again the sequence from Sect. 2.3. After three iterations of IRCOO-MC the words xax, xbx and xcx are chosen, and the $\mathcal{P}_{min}$ of these non-terminals and the original sequence results in G^*.

IRRCOO extends IRR by performing a global optimization at each step of the replaced occurrences but still relies on the classical score functions of IRR to choose the words to introduce. But the result of the optimization can be used directly to guide the search in a hill-climbing like approach that we introduce in the next subsection.

3.3 Widening the Explored Space: The ZZ Algorithm

In this section we divert from IRR algorithms by taking the idea presented in IRCOO a step forward. In the optimization of the occurrences replacement performed in Sect. 3.1, the choice of the non-terminals implicitly implied their yields: there was a direct relation between non-terminals and terminals, but the focus was on non-terminals. Instead, we can focus on yields and then, the optimization algorithm can be seen as a procedure that takes a string s and a set of its substrings as input and that returns the smallest grammar that can be built from it, provided that the grammar produces s and that, for each substring in the input set, there exists a non-terminal in the grammar whose yield is the substring itself. The optimization procedure works as a scoring function for sets of substrings: the size of the grammar produced by the optimization procedure is the score of the given set of substrings.

In this section, we take advantage of this idea and present an algorithm, called *Zig Zag* (ZZ), that traverses in a hill-climbing way the lattice of the subsets of repeated substrings of the string s. The search space is a lattice that has one node for each possible set of repeated substrings of s, and has an edge from node a to node b if exactly one substring has to be added to the set that corresponds to a in order to obtain the set that corresponds to b. The bottom node corresponds to the empty set while the top corresponds to the set of all repeated substrings of s. The score of a node is defined as the size of a smallest grammar obtained using the optimization of the occurrences replacement with s and the substrings in the node. As an example, the score of the bottom node is the size of grammar $S \to s$ and the score of the top node corresponds to the size of a smallest grammar that has one non-terminal for s and for each repeated substring of s.

There exists a node in the lattice whose score is the size of a smallest grammar. But, this optimal set of substrings cannot be efficiently computed because the lattice that has to be explored is exponentially big. ZZ explores it by an alternation of two different phases: *bottom-up* and *top-down*. The bottom-up can be started at any node, it moves upwards in the lattice and at each step it looks among its immediate descendants for the one with the lowest score. In order to determine which is the one with the lowest score, it inspects them all. It stops when no descendant has a better score than the current one. As in bottom-up, top-down starts at any given node but it moves downwards looking for the node with the smallest score among its immediate ancestors. Going up or going down from the current node is equivalent to adding or removing a substring to or from the set of substrings in the current node respectively.

ZZ starts at the bottom node, that is, the node that corresponds to the grammar $S \to s$ and it finishes when no improvements are made in the score between two bottom-up–top-down iterations.

For example, suppose that there are 5 substrings that occur more than once in α and that they all have length greater than two. Let these strings be numbered from 0 to 4. We start the ZZ algorithm at the bottom node. It inspects nodes $\{0\}$, $\{1\}$, $\{2\}$, $\{3\}$, and $\{4\}$. Suppose that $\{1\}$ produces the best grammar, then ZZ moves to that node and starts over exploring the nodes above it. Figure 2 shows a part of the lattice being explored. Dotted arrows point to nodes that are explored while full arrows points to nodes having the lowest score. Suppose that the algorithm then continues up until it reaches node $\{1, 2, 3\}$ where it can not go up any further. Then ZZ starts the top-down phase, going down to node $\{2, 3\}$ where it can no go any lower. At this point a bottom-up–top-down iteration is done and the algorithm starts over again. It goes up, suppose that it reaches node $\{2, 3, 4\}$ where it stops. Bold circled nodes correspond to nodes were the algorithm switches phases, grey nodes corresponds to nodes with the best score among its siblings.

Computational Complexity. In the previous section we showed that the computational complexity of computing the score function for each node is $O(n \times m^2)$, where n is the length of the target string and m is the number of substrings in the node. Every time ZZ looks for a substring to add or remove it has to inspect all possible candidates with the aim of finding the one that minimizes the score. Depending on the number of substrings that are already in the node, there might be at most $O(n^2)$ candidate strings. As a consequence, each step upwards or downwards is made in $O(n^2 \times n \times m^2)$. Next, we need to give an upper bound for the length of the path that is potentially traversed by the algorithm. In order to define it, we first note two important properties: the score of the bottom node is equal to n and the score of any node containing more than $n/2$ substrings is at least n. The first one is trivially true, while the second is true because, since every rule body contains at least two symbols, if there were $n/2$ rules, then the grammar size would be at least n. The bottom-up phase visits at most $n/2$ nodes, and consequently, the top-down can only go down at most $n/2$ steps. Adding them together, it turns out that a bottom-up, top-down iteration traverses at most n nodes. Now, each of these iteration decreases the score bt at least 1, otherwise the algorithm stops. Since the initial score is n plus the fact that the score is always positive, it is true that there can be at most n bottom-up–top-down iterations. This results in a complexity for the ZZ algorithm of $O(n^5 \times m^2)$.

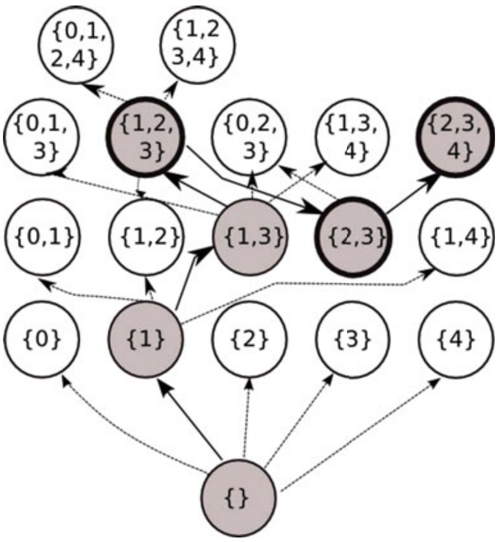


Fig. 2. The fraction of the lattice that is explored by the ZZ algorithm

4 Experiments

In this section we experimentally compare our algorithms with the classical ones from the literature. For this purpose we use the Canterbury Corpus [2] which is a standard corpus for comparing lossless data compression algorithms. Table 1 lists the sequences of the corpus together with their length and number of repeats of length greater then one.

Not all existing algorithms are publicly available, they resolve in different way the case when there are more then two words with the best score, they do not report results on a standard corpus or they use different definitions of size of a grammar. In order to standardize the experiments and score, we implemented

Table 1. Corpus statistics

sequence	length	# of repeats
alice29.txt	152,089	220,204
asyoulik.txt	125,179	152,695
cp.html	24,603	106,235
fields.c	11,150	56,132
grammar.lsp	3,721	12,780
kennedy.xls	1,029,744	87,427
lcet10.txt	426,754	853,083
plrabn12.txt	481,861	491,533
ptt5	513,216	99,944,933
sum	38,240	666,934
xargs.1	4,227	7,502

all the offline algorithms presented in this paper in the IRR framework. For the sake of completeness, we also add to the comparison LZ78 and Sequitur. Note that we have post-processed the output of the LZ78 factorizations to transform them into context-free grammars. The first series of experiments aims at comparing these classical algorithms and are shown in the middle part of Table 2. On this benchmark, we can see that IRR-MC outputs always the smallest grammar, which are in average 4.22% smaller then those of the second best (IRR-MO), confirming the partial results of [12] and showing that IRR-MC is the current state-of-the-art practical algorithm for this problem.

Then we evaluate how the optimization of occurrences improves IRR algorithms. As shown in the IRRCOO column of Table 2, each strategy for choosing the word is improved by introducing the occurrence optimization. The sole exceptions are for the MO strategy on grammar.lsp and xargs.1, but the difference in these cases is very small and the sequences are rather short. More important, we can see that IRCOO-MC is becoming the new state-of-the-art algorithm, proposing for each test a smaller grammar than IRR-MC, and being outperformed only on `plrabn12.txt` by IRCOO-MO.

If we are given more time, these results can still be improved by using ZZ. As shown in column ZZ of Table 2, it obtains in average 3.12% smaller grammars than IRR-MC, a percentage that increases for the sequences containing natural language (for instance, for `alice29.txt` the gain is 8.04%), while it is lower for other sequences (only 0.1% for `kennedy.xls` for example). For the latter case, one can remark that the compression ratio is already very high with IRR-MC and that it may be difficult or impossible to improve it, the last few points of the percentage gain being always the hardest to achieve. As expected, ZZ improves over previous approaches mainly because it explores a much wider fraction of search space. Interestingly enough, the family of IRRCO algorithms also improves state of the art algorithm but still keeps the greedy flavour, and more importantly, it does so with a complexity cost similar to pure greedy approaches. The price to be paid for computing grammars with ZZ is its computational complexity. This problem already showed up with `plrabn12.txt`, `lcet10.txt` (where each repeat individually does not compress much the sequence, so lots of iterations are necessary) and `ptt5` (which contains about 99 millions of repeats).

It is interesting to know whether the structure of the grammars found by occurrence optimization are simply some refinements of the ones found by classical algorithms or whether they differ completely. For the inference of the structure of the sequence, this is even crucial. This subject deserves a more complete study. We present here the result of comparing the structures returned by IRR-MC and ZZ on the typical test case `asyoulik.txt`. In that case the ZZ grammar is 6.6% smaller than that of IRR-MC, but using the standard unlabeled precision and recall metric [7], gives an F-measure – which is roughly speaking the measure of how similar both structures are – is only 34.6%. Moreover, the F-measure of non-crossing brackets – a measure of how compatible the structures are – is also very low: 35.8%. We can see that the size improvement achieved by the algorithm optimizing the choice of the occurrences has a dramatic effect on the structure found. The same kind of behavior can be already seen between IRR-MC and IRCOO-MC, the F-measure and the F-measure of non-crossing brackets being in that case 55.1% and 56.9% respectively, for a size improvement of 2.9%.

Table 2. Grammar sizes on the Canterbury corpus. The files over which ZZ did not finished are marked with a dash.

	Algorithms from the literature					Optimizing occurrences			
			IRR			IRCOO			
Sequences	Sequitur	LZ78	MO	ML	MC	MO	ML	MC	ZZ
alice29.txt	49,147	116,296	42,453	56,056	**41,000**	39,794	5,235	**39,251**	**37,701**
asyoulik.txt	44,123	102,296	38,507	51,470	**37,474**	36,822	48,133	**36,384**	**35,000**
cp.html	9,835	22,658	8,479	9,612	**8,048**	8,369	9,313	**7,941**	**7,767**
fields.c	4,108	11,056	3,765	3,980	**3,416**	3,713	3,892	**3,373**	**3,311**
grammar.lsp	1,770	4,225	1,615	1,730	**1,473**	1,621	1,704	**1,471**	**1,465**
kennedy.xls	174,585	365,466	167,076	179,753	**166,924**	166,817	179,281	**166,760**	**166,704**
lcet10.txt	112,205	288,250	92,913	130,409	**90,099**	90,493	164,728	**88,561**	–
plrabn12.txt	142,656	338,762	125,366	180,203	**124,198**	**114,959**	164,728	117,326	–
ptt5	55,692	106,456	45,639	56,452	**45,135**	44,192	53,738	**43,958**	–
sum	15,329	35,056	12,965	13,866	**12,207**	12,878	13,695	**12,114**	**12,027**
xargs.1	2,329	5,309	2,137	2,254	**2,006**	2,142	2,237	**1,989**	**1,972**

5 Conclusions

We propose to separate the choice of the words from the choice of the occurrences where they are going to be rewritten in algorithms searching for a smallest grammar. First we improve classical offline algorithms by optimizing at each step the choice of the occurrences. The separation allowing to define the search space as a lattice over sets of repeats, we propose then a new algorithm that explores this search space by adding, but also removing, repeats to the current set of words.Our experiments show that both approaches outperform state-of-the-art algorithms.

The optimization of the choice of occurrences opens new perspectives when searching for the smallest grammars, especially for the inference of the structure of sequences. In future work, we want to study how this scheme helps actually to find better structure on real applications.

References

1. Apostolico, A., Lonardi, S.: Off-line compression by greedy textual substitution. Proceedings of the IEEE (January 2000)
2. Arnold, R., Bell, T.: A corpus for the evaluation of lossless compression algorithms. In: Data Compression Conference, Washington, DC, USA, p. 201. IEEE Computer Society, Los Alamitos (1997)

3. Bentley, J., McIlroy, D.: Data compression using long common strings. In: Data Compression Conference, pp. 287–295 (March 1999)
4. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., Shelat, A.: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
5. Evans, S.C., Kourtidis, A., Markham, T., Miller, J.: MicroRNA target detection and analysis for genes related to breast cancer using MDLcompress. EURASIP Journal on Bioinformatics and Systems Biology (3) (2007)
6. Kieffer, J., Yang, E.H.: Grammar-based codes: a new class of universal lossless source codes. IEEE Transactions on Information Theory 46 (2000)
7. Klein, D.: The Unsupervised Learning of Natural Language Structure. PhD thesis, University of Stanford (2005)
8. Lanctot, J.K., Li, M., Yang, E.H.: Estimating DNA sequence entropy. In: ACM-SIAM Symposium on Discrete Algorithms, pp. 409–418 (January 2000)
9. Larsson, N., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)
10. Marcken, C.D.: Unsupervised language acquisition. PhD thesis, Massachusetts Institute of Technology (January 1996)
11. Nakamura, R., Inenaga, S., Bannai, H., Funamoto, T., Takeda, M., Shinohara, A.: Linear-time text compression by longest-first substitution. Algorithms 2(4), 1429–1448 (2009)
12. Nevill-Manning, C., Witten, I.: On-line and off-line heuristics for inferring hierarchies of repetitions in sequences. In: Data Compression Conference, pp. 1745–1755. IEEE, Los Alamitos (2000)
13. Nevill-Manning, C.G.: Inferring Sequential Structure. PhD thesis, University of Waikato (1996)
14. Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical structure in sequences: A linear-time algorithm. Journal of Artificial Intelligence Research 7 (January 1997)
15. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science 302(1-3), 211–222 (2003)
16. Sakakibara, Y.: Efficient learning of context-free grammars from positive structural examples. Inf. Comput. 97(1), 23–60 (1992)
17. Sakamoto, H., Maruyama, S., Kida, T., Shimozono, S.: A space-saving approximation algorithm for grammar-based compression. IEICE Transactions 92-D(2), 158–165 (2009)
18. Schuegraf, E.J., Heaps, H.S.: A comparison of algorithms for data base compression by use of fragments as language elements. Information Storage and Retrieval 10, 309–319 (1974)
19. Wolff, J.: An algorithm for the segmentation of an artificial language analogue. British Journal of Psychology 66 (1975)
20. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)
21. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Agreement and Cliticization in Italian: A Pregroup Analysis

Claudia Casadio

University G. D'Annunzio - Chieti
casadio@unich.it

Abstract. This paper presents an analysis of features specification and agreement based on the parallel computations of a type calculus involving two pregroups grammars: the free pregroup of syntactic types that takes care of the syntactic calculations, and a second free pregroup computing feature operations. As recently argued in [19] , working with two free pregroups in parallel has the advantage of treating featural information more carefully and independently from the type assignments; the calculus is therefore particularly appropriate for the analysis of agreement properties in Romance languages. In the paper we focus on the Italian language, introducing a type syntax of the verbal constructions in which clitic pronouns occur and offering an explanation of the interaction between agreement features and clitic pronouns.

1 The Pregroup Calculus in Linguistic Analysis

The calculus of pregroups is introduced by Lambek in [15] as an alternative to his Syntactic Calculus [14], a well known model of categorial grammar (see [21]). The calculus of pregroups is a particular kind of substructural logic that is compact and non-commutative ([3], [4]; [2]; [15], [17]). Pregroups in fact are non-conservative extensions of classical non-commutative linear logic in which *left* and *right* iterated negations, equivalently *left* and *right* iterated adjoints, do not cancel ([1]; [6]; [9]; [5]). The calculus of pregroups has been applied to a variety of natural languages and many relevant linguistic dimensions have been successfully addressed (see e.g. [16] [17] [18]; for a survey, see [9]).

1.1 Basic Properties of Pregroups

A *pregroup* $\{G, \centerdot\, , 1, ^\ell, ^r, \rightarrow\}$ is a partially ordered monoid in which each element a has a *left adjoint* a^ℓ, and a *right adjoint* a^r such that

$$a^\ell a \rightarrow 1 \rightarrow a\, a^\ell$$
$$a\, a^r \rightarrow 1 \rightarrow a^r a$$

where the dot "**.**", that is usually omitted, stands for multiplication with unit 1, the arrow denotes the partial order, the rules $a^\ell a \rightarrow 1$, $a\, a^r \rightarrow 1$ are called contractions, and the opposite rules $1 \rightarrow a\, a^\ell$, $1 \rightarrow a^r a$ are called expansions.

When considering linguistic applications, the constant 1 stands for the empty string of types, and multiplication is interpreted as concatenation. The following principles are proven showing that the constant 1 is self-dual, adjoints are unique and contravariant, and iterated adjoints can be obtained (in fact double adjoints play a relevant role in linguistic applications [6], [9]).

$$1^\ell = 1 = 1^r ,$$
$$(a \cdot b)^\ell \;=\; b^\ell \cdot a^\ell \quad , \quad (a \cdot b)^r \;=\; b^r \cdot a^r ,$$
$$\frac{a \to b}{b^\ell \to a^\ell} \; , \quad \frac{a \to b}{b^r \to a^r} \; , \quad \frac{b^\ell \to a^\ell}{a^{\ell\ell} \to b^{\ell\ell}} \; , \quad \frac{b^r \to a^r}{a^{rr} \to b^{rr}} \; .$$

In the pregroup calculus one can prove the following equation

$$a^{r\ell} = a = a^{\ell r} ,$$

allowing the cancellation of double opposite adjoints, and the rules,

$$a^{\ell\ell}\; a^\ell \to 1 \to a^\ell\; a^{\ell\ell} \; , \; a^r\; a^{rr} \to 1 \to a^{rr}\; a^r \; ,$$

contracting and expanding identical *left* and *right* double adjoints respectively. Just the *contractions* $a^\ell\, a \to 1$ and $a\, a^r \to 1$ are needed to determine the constituent analysis of a linguistic expression, and to show that a string of words is a sentence; on the other hand, the *expansions* $1 \to a\, a^\ell$, $1 \to a^r a$ are useful for representing general structural properties (see e.g. [22]).

At the syntactic level, a pregroup is *freely generated* by a partially ordered set of *basic* types. From each basic type a we form *simple* types by taking single or repeated adjoints: $\ldots\, a^{\ell\ell}, a^\ell, a, a^r, a^{rr} \ldots$. A *c*ompound type or just a *t*ype is a string of simple types: $a_1\; a_2\; \ldots\; a_n$. A basic type is a type (for n = 1).

Developing a pregroup grammar for a language such as Italian, consists in two main steps: (i) assign one or more (basic or compound) types to each word in the dictionary; (ii) check the grammaticality and sentencehood of a string of words by a calculation on the corresponding types, where the only rules involved are *contractions* and *ordering postulates* such as $\alpha \to \beta$ (α, β basic types). Language specific conditions on strings of types called *metarules* can also be added to the lexicon of the grammar, in order to simplify lexical assignments and make syntactic calculations quicker, but they will not be used in the present context.

1.2 Extension to Multiple Pregroups

In many languages, particularly in Romance languages, features specification and agreement play a relevant role; for example in Italian you say: *tu sei arrivato* or *tu sei arrivata* "you have arrived", depending on the subject being *masculine* or *feminine*; you also have more complex cases involving cliticization: *lui vuole averla abbracciata/*abbracciato* "he wants to have embraced her", where the clitic pronoun *la* agrees in gender (feminine) and number (singular) with the past participle of the verb *abbracciare* "to embrace"; *i libri, li volevo avere letti tutti* "the books, I would have read them all", where the clitic pronoun *li* agrees in gender (masculine) and number (plural) with the noun *i libri* (its antecedent), the past participle of the verb *leggere* "to read" and the adjective *tutti* "all".

To carry out correct computations on features such as *gender*, *number* and *person* while processing the syntactic types assigned to the words of the language under investigation, we introduce a second free pregroup: the *feature* pregroup, working in parallel with the pregroup of *syntactic types*. Feature types will be defined separately from the syntactic types and they will be written below the latter, while the string of types is processed, in the same style of [19]. Similar strategies of processing types and features in parallel have been proposed by Ed Stabler e.g. [24], and by some students of Brendan Gillon, e.g. [13]; in turn Anne Preller and Violaine Prince in [23] prefer to assign featural informations to types (see [10] for details).

2 A Pregroup Grammar for Italian

In this section we introduce the pregroup grammar for the Italian language ([10], [8], [7]). The syntactic free pregroup for Italian is generated by the following *partially ordered* set of basic types:

s, $\bar{\mathrm{s}}$	declarative sentences
i, $\tilde{\imath}$, $\bar{\imath}$, i^*, j, $\bar{\jmath}$, $\bar{\bar{\imath}}$	infinitive clauses
π	subject
o	direct object
ω	indirect object
λ	locative phrase

The type π is assigned to the subject, in *nominative* case, the types o, ω, λ, to the arguments of the verb, in *accusative*, *dative* and *locative* case respectively; the type $\bar{\mathrm{s}}$ is assigned to the expansions of declarative sentences of type s; the type $\bar{\bar{\imath}}$ is the maximal projection of a variety of infinitival complements i, $\tilde{\imath}$, $\bar{\imath}$, i^*, j, $\bar{\jmath}$, where the bar notation is inspired to Chomsky's X-bar theory [11]. The following types are assigned to a few representative verbs: intransitive verbs like *correre* and verbs taking different kinds of complements

(1)	(*to see*)	*vedere* :	$i\ ,\ i\,\mathrm{o}^{\ell}$
(2)	(*to obey*)	*obbedire* :	$i\ ,\ i\,\omega^{\ell}$
(3)	(*to give*)	*dare* :	$i\,\omega^{\ell}\mathrm{o}^{\ell}\ ,\ i\,\mathrm{o}^{\ell}\omega^{\ell}$
(4)	(*to put*)	*mettere* :	$i\,\lambda^{\ell}\mathrm{o}^{\ell}\ ,\ i\,\mathrm{o}^{\ell}\lambda^{\ell}$
(5)	(*to arrive*)	*arrivare* :	$i^*\ ,\ i^*\lambda^{\ell}$.
(6)	(*to run*)	*correre* :	$i\ ,\ i\,\lambda^{\ell}$

The star on i^* is a reminder that the perfect tense is to be formed with the auxiliary *essere* rather than *avere*, producing infinitival phrases of type i^*, rather than of type i.

2.1 Verbs and Verb Phrases

The following examples show how the verb types combine via contraction with the proper arguments to give the expected infinitives of type i (simple types are assigned to the verb complements such as the direct object phrases *un quadro*, *un libro*, the indirect object phrase *a Carla*, etc.):

(1) $\textit{vedere}\ \underbrace{\textit{un quadro}}$ "to see a picture"
$(i\ \mathrm{o}^{\ell})\quad \mathrm{o}\ \to\ i$

(2) $\textit{obbedire}\ \underbrace{\textit{a Mario}}$ "to obey to Mario"
$(i\ \omega^{\ell})\quad \omega\ \to\ i$

(3a) $\textit{dare}\ \underbrace{\textit{un libro}}\ \underbrace{\textit{a Carla}}$ "to give a book to Carla"
$(i\ \omega^{\ell}\ \mathrm{o}^{\ell})\quad \mathrm{o}\quad \omega\ \to\ i$

(3b) $\textit{dare}\ \underbrace{\textit{a Carla}}\ \underbrace{\textit{un libro}}$ "to give Carla a book"
$(i\ \mathrm{o}^{\ell}\ \omega^{\ell})\quad \omega\quad \mathrm{o}\ \to\ i$

(4) $\textit{mettere}\ \underbrace{\textit{un libro}}\ \underbrace{\textit{sul tavolo}}$ "to put a book on the table"
$(i\ \lambda^{\ell}\ \mathrm{o}^{\ell})\quad \mathrm{o}\quad \lambda\ \to\ i$

(5) $\textit{arrivare}\ \underbrace{\textit{a Roma}}$ "to arrive to Rome"
$(i^{*}\ \lambda^{\ell})\quad \lambda\ \to\ i^{*}$

(6) $\textit{correre}\ \underbrace{\textit{sul prato}}$ "to run in the meadow"
$(i\ \lambda^{\ell})\quad \lambda\ \to\ i$

Some of the above verbs take more than one type, e.g. *vedere* is used intransitively as a word of type i and transitively as a word of type $i\mathrm{o}^{\ell}$. The verb *dare* takes two types to combine with the direct object o (what is given) and the indirect object ω (the receiver) in both orders: $o\ \omega$ vs. $\omega\ o$; postverbal complements exchange is in fact a regular phenomenon in Italian. When attaching to a clitic, the final letter of the infinitive is dropped and the resulting *short infinitive* changes the type as follows (the purpose of the *bar* will become clear later):

$$\begin{array}{lll}
\textit{veder}: & \bar{\imath}\,\bar{o}^{\ell} & ; \\
\textit{obbedir}: & \bar{\imath}\,\bar{\omega}^{\ell} & ; \\
\textit{dar}: & \bar{\imath}\,\omega^{\ell}\,\bar{o}^{\ell}\ ,\ \ \bar{\imath}\,\mathrm{o}^{\ell}\,\bar{\omega}^{\ell} & ; \\
\textit{metter}: & \bar{\imath}\,\lambda^{\ell}\,\bar{o}^{\ell}\ ,\ \ \bar{\imath}\,\mathrm{o}^{\ell}\,\overline{\lambda}^{\ell} & ; \\
\textit{arrivar}: & \overline{i^{*}}\,\overline{\lambda}^{\ell} & .
\end{array}$$

2.2 The Auxiliary Verbs

Italian has two auxiliary verbs: *avere* "to have" and *essere* "to be"; differently from English, both are selected by active verbs, but *essere* is also required in passive forms. The perfect infinitive is formed from the past participle with the help of the two auxiliary verbs. In these contexts *avere* requires the type ip_2^{ℓ}, where p_2 is the type of the past participle of a verb with infinitive of type i, e.g. the type of *visto* "seen" when *vedere* "to see" is used intransitively. We analyze the past participles as generated in the grammar by means of the following PERF inflection of type $p_2 i^{\ell}$, applying both to transitive and intrasitive verbs

$$\begin{array}{ll}
\text{PERF}\,(\textit{vedere}) = \textit{visto} & \qquad \text{PERF}\,(\textit{vedere}) = \textit{visto} \\
(p_2\ i^{\ell})\ \ i\ \ \to\ \ p_2 & \qquad (p_2\ i^{\ell})\ (i\ \mathrm{o}^{\ell})\ \to\ \ p_2\,\mathrm{o}^{\ell}
\end{array}$$

Here are some examples of past participles infinitives with the auxiliary avere

$$\begin{array}{lllll} & \textit{avere} & \textit{visto} & \textit{un libro} & , \\ = & \textit{avere} & \text{PERF}\,(\textit{vedere}) & (\textit{un libro}) & \\ & (i\ p_2^{\ell}) & (p_2\ i^{\ell})\ (i\ \mathrm{o}^{\ell}) & \mathrm{o} & \rightarrow\ i \end{array}$$

$$\begin{array}{llllll} & \textit{avere} & \textit{dato} & \textit{un libro} & \textit{a Mario} & , \\ = & \textit{avere} & \text{PERF}\,(\textit{dare}) & (\textit{un libro}) & (\textit{a Mario}) & \\ & (i\ p_2^{\ell}) & (p_2\ i^{\ell})\ (i\ \omega^{\ell}\ \mathrm{o}^{\ell}) & \mathrm{o} & \omega & \rightarrow\ i \end{array}$$

A number of intransitive verbs require the auxiliary *essere* for forming the perfect infinitive, e.g. the motion verb *arrivare* "arrive" with the two types i^* or $i^*\lambda^{\ell}$, and past participle *arrivato*. In these contexts *essere* is assigned the type: $i^* p_2^{*\ell}$ while the PERF inflection takes the type: $p_2^*\, i^{*\ell}$. This example illustrates how the perfect infinitive of the starred verbs is formed:

$$\begin{array}{lllll} & \textit{essere} & \textit{arrivato} & , & \\ = & \textit{essere} & \text{PERF}\,(\textit{arrivare}) & & \\ & (i^* p_2^{*\ell}) & (p_2^*\, i^{*\ell})\ \ i^* & \rightarrow & i^* \end{array}$$

In the following, we will assume that the PERF inflection has already been applied and will assign the past partiples the resulting types p_2, p_2^*, $p_2\mathrm{o}^{\ell}$, and so on.

2.3 Finite Verb-Forms and Declarative Sentences

To produce sentences we need finite verb forms that we obtain by associating to each Italian verb V a matrix V_{jk} of $7 \times 6 = 42$ finite verb-forms, where the subscript j = 1, ... ,7 denotes *tenses* (respectively, present, imperfect, past, future in indicative mood; present and past subjunctive; present conditional), and the subscript k = 1, 2, 3 denotes the three *persons singular*, while k = 4, 5, 6 denotes the three *persons plural*. We shall use the types

s_j for declarative sentences in j-th tense ,

π_k for k-th person subject .

and confine our attention to the cases: j = 1 (present tense), j = 2 (past tense), and k = 1 (first person) or k = 3 (third person). For example, in the sentence

$$\begin{array}{lll} \text{“I} & \text{see} & \text{a girl”} \\ (\textit{io}) & \textit{vedo} & (\textit{una ragazza})\ , \\ \pi_1 & (\pi_1^r\ \mathrm{s}_1\ \mathrm{o}^{\ell}) & \mathrm{o}\ \rightarrow\ \mathrm{s}_1 \end{array}$$

the optional pronoun *io* has type π_1, and the first person verb form *vedo* "I see" has type $(\pi_1^r\ \mathrm{s}_1\ \mathrm{o}^{\ell})$. The properties of the inflectional system represented by the conjugation matrix V_{jk} play a crucial role in word order and constituent formation in Italian and, following [8], we assume that V_{jk} is obtained from the infinitive V by an *inflector* C_{jk} such that

$$\mathsf{C}_{jk}(\mathrm{V}) = \mathsf{V}_{jk}\ .$$

3 Clitic Pronouns and Agreement in Italian

Italian, like Spanish and Portuguese, has both preverbal and postverbal clitic pronouns. We will concentrate on the preverbal clitics in the accusative, dative and locative cases and shall ignore other kinds of clitics such as the partitive *ne* or the possessive *si.* We also shall not take into consideration the postverbal occurences of clitic pronous (treated in details in [8]).

3.1 Preverbal Clitics

The following is the list of clitics and their types when used preverbally:

Accusative	*mi, ti, ci, vi* : $\overline{j}\,\mathrm{o}^{\ell\ell}\,i^{\ell}$
	lo, la, li, le : $j\,\mathrm{o}^{\ell\ell}\,i^{\ell}$
Dative	*mi, ti, ci, vi, gli, le* : $\overline{j}\,\omega^{\ell\ell}\,i^{\ell}$, $\overline{j}^{\,*}\,\omega^{\ell\ell}\,i^{*\ell}$
	me, te, ce, ve, se, glie : $\overline{j}\,\omega^{\ell\ell}\,j^{\ell}$
	se : $\overline{j}^{*}\,\omega^{\ell\ell}\,j^{\ell}$
Locative	*ci, vi* : $\overline{j}\,\lambda^{\ell\ell}\,i^{\ell}$, $\overline{j}^{\,*}\,\lambda^{\ell\ell}\,i^{*\ell}$
	ce, ve : $\overline{j}\,\lambda^{\ell\ell}\,j^{\ell}$

We introduce four new basic types for infinitives j , j^*, $\overline{j}$, $\overline{j}^*$ and determine their relations by postulating $j \to \overline{j}$, $j^* \to \overline{j}^*$, but $i \not\to j \not\to \overline{i}$. It follows that infinitives of type $\overline{j}$ cannot be preceded by any clitics and infinitives of type j only by clitics such as *me* and *ce.* We have double clitics such as

$$\begin{matrix} \textit{me} & . & \textit{lo} & , \\ (\overline{j}\;\omega^{\ell\ell}\;j^{\ell}) & & (\,j\;\mathrm{o}^{\ell\ell}\;i^{\ell}) & \to\;\overline{j}\;\omega^{\ell\ell}\;\mathrm{o}^{\ell\ell}\;i^{\ell}\;. \end{matrix}$$

Here are some illustrations of preverbal clitics where the under-links show how contractions apply to produce the computation leading to the required type

$$\begin{matrix} \textit{me}\;.\;\textit{lo} & \textit{dare} & , \\ (\overline{j}\;\omega^{\ell\ell}\;\mathrm{o}^{\ell\ell}\;i^{\ell}) & (i\;\mathrm{o}^{\ell}\omega^{\ell}) & \to\;\overline{j} \end{matrix}$$

$$\begin{matrix} \textit{ce}\;.\;\textit{lo} & \textit{mettere} & , \\ (\overline{j}\;\lambda^{\ell\ell}\;\mathrm{o}^{\ell\ell}\;i^{\ell}) & (i\;\mathrm{o}^{\ell}\lambda^{\ell}) & \to\;\overline{j} \end{matrix}$$

$$\begin{matrix} \textit{lo} & \textit{vedere} & & , & \textit{ci} & \textit{arrivare} & & . \\ (j\;\mathrm{o}^{\ell\ell}\;i^{\ell}) & (i\;\mathrm{o}^{\ell}) & \to\;j & & (\overline{j}^{\,*}\;\lambda^{\ell\ell}\;i^{*\ell}) & (i^{*}\lambda^{\ell}) & \to\;\overline{j}^{*} & \end{matrix}$$

Partial cliticization can be obtained with double complements verbs

mi *dare* *un libro* , *lo* *dare* *a Mario* ,

$$(\overline{j}\;\omega^{\ell\ell}\;i^{\ell})\;(i\;\omega^{\ell}\mathrm{o}^{\ell})\quad \mathrm{o}\;\to\;\overline{j}\qquad\qquad (j\;\mathrm{o}^{\ell\ell}\;i^{\ell})\;(i\;\mathrm{o}^{\ell}\omega^{\ell})\quad \omega\;\to\;j$$

but the following computations are not allowed as shown by the invalid links

**mi* *dar* . *lo* , **mi* *dare* *lo* .

$$(\overline{j}\;\omega^{\ell\ell}\;i^{\ell})\;(\overline{\imath}\;\omega^{\ell}\;\overline{o}^{\ell})\;\widehat{o}\qquad\qquad (\overline{j}\;\omega^{\ell\ell}\;i^{\ell})\;(i\;\omega^{\ell}\mathrm{o}^{\ell})\;\widehat{o}$$

excluding the ungrammatical strings in which preverbal and postverbal clitics occur together by means of the order conditions on the types hierarchy $\overline{\imath}\not\to i$, $\widehat{o}\to\overline{o}$, $\widehat{o}\not\to \mathrm{o}$. (see [8], 115).

3.2 The Modal Verbs

The group of modal verbs *potere*, *volere*, *dovere* requires the following types for their extended infinitive and short infinitive

potere, *volere*, *dovere* : $i\,\overline{\imath}^{\ell}$, $i^{*}\,\overline{\imath}^{*\ell}$; *poter*, *voler*, *dover* : $\overline{\imath}\,\overline{j}^{\ell}$, $\overline{\imath}^{*}\overline{j}^{*\ell}$

and here are a few illustrative examples

"must obey to Dario" "must to him obey"

dovere obbedire (*a Dario*) , *dover* . *gli obbedire* ,

$$(i\,\overline{\imath}^{\ell})\quad(i\;\omega^{\ell})\qquad \omega\;\to\;i\qquad\qquad (\overline{\imath}\,\overline{j}^{\ell})\;(\overline{j}\omega^{\ell\ell}i^{\ell})\;(i\;\omega^{\ell})\;\to\;\overline{\imath}$$

poter . *me* . *lo* *dare* , *me* . *lo* *potere* *dare* ,

$$(\overline{\imath}\,\overline{j}^{\ell})\;(\overline{j}\;\omega^{\ell\ell}\mathrm{o}^{\ell\ell}\;i^{\ell})\;(i\;\mathrm{o}^{\ell}\omega^{\ell})\;\to\;\overline{\imath}\qquad\qquad (\overline{j}\;\omega^{\ell\ell}\mathrm{o}^{\ell\ell}\;i^{\ell})\;(i\,\overline{\imath}^{\ell})\;(i\;\mathrm{o}^{\ell}\omega^{\ell})\;\to\;\overline{j}$$

ci *potere* *arrivare* , *poter* . *ci* *arrivare* ,

$$(\overline{j}^{*}\lambda^{\ell\ell}i^{*\ell})\;(i^{*}\,\overline{\imath}^{*\ell})\;(i^{*}\lambda^{\ell})\;\to\;\overline{j}^{*}\qquad\qquad (\overline{\imath}^{*}\overline{j}^{*\ell})\;(\overline{j}^{*}\lambda^{\ell\ell}i^{*\ell})\;(i^{*}\lambda^{\ell})\;\to\;\overline{\imath}^{*}$$

Modal verbs allow repetition, not only

potere volere , but even *potere potere* , and *potere poter*,

$$(i\,\overline{\imath}^{\ell})(i\,\overline{\imath}^{\ell})\to i\,\overline{\imath}^{\ell}\qquad\qquad (i\,\overline{\imath}^{\ell})(i\,\overline{\imath}^{\ell})\to i\,\overline{\imath}^{\ell}\qquad\qquad (i\,\overline{\imath}^{\ell})(\overline{\imath}\,\overline{j}^{\ell})\to i\,\overline{j}^{\ell}\;.$$

In these examples we can replace i by i^{*} and j by j^{*}. Modal verbs interact with the auxiliary verbs *avere* and *essere* as shown by the following examples

potere *avere* *visto* *un libro* ,

= *potere* *avere* Perf(*vedere*) (*un libro*)

$$(i\,\overline{\imath}^{\ell})\;(ip_2^{\ell})\quad(p_2 i^{\ell})\;(i\,\mathrm{o}^{\ell})\qquad \mathrm{o}\;\to\;i$$

$$\begin{array}{cllllll} & \textit{avere} & \textit{potuto} & & \textit{vedere} & \textit{un libro}\ , & \\ = & \textit{avere} & \text{Perf}\,(\textit{potere}) & & \textit{vedere} & (\textit{un libro}) & \\ & (ip_2^\ell) & (p_2 i^\ell) & (i\,\bar{\imath}^\ell) & (i\,\mathrm{o}^\ell) & \mathrm{o} & \rightarrow\ i \end{array}$$

3.3 Agreement with Past Participles and Auxiliary Verbs

In Italian, like in other Romance languages, past participles are similar to adjectives and when they occurr with the auxiliary verb *essere* they change the final vowel *o* to *a*, *i* or *e*, depending on the gender and number of the preceding subject, e.g. *essere arrivata* (**arrivato*), when the subject is singular and feminine, but *essere arrivati* (**arrivate*), when the subject is plural and masculine. When cliticization occurs, a similar agreement of features is required also with the auxiliary verb *avere*, as shown in these examples

(1) *avere visto una ragazza* (to have seen a girl)
(2) *poter.la avere vista* (**visto*) (may have seen[*fem*] her)
(3) *avere dato una lettera a Dario* (to have given a letter to Dario)
(4) *poter.la avere data* (**dato*) *a Dario* (may have given[*fem*] it[*fem*] to Dario)
(5) *avere messo il libro sul tavolo* (to have put the book on the table)
(6) *poter.lo avere messo* (**messa*) *sul tavolo* (have put[*masc*] it[*masc*] on the table)

Following the strategy of adding featural information to syntactic types, we would obtain derivations like the following where the type of the past participle *arrivata* is specified for the feature f requiring a subject in the feminine gender

$$\begin{array}{llll} \textit{potere} & \textit{essere} & (\textit{arrivata})\ , & \\ (i^*\,\bar{\imath}^{*\ell}) & (i^* p_2^{*\ell}) & p_{2f}^* & \rightarrow\ i \end{array}$$

but we would be in trouble when trying to assign features to compound types like in the following examples

$$\begin{array}{lllll} \text{“to} & \text{have} & \text{seen} & \text{a girl”} & \\ & \textit{avere} & \textit{visto} & (\textit{una ragazza})\ , & \\ & (i\ p_2^\ell) & (p_2\,o^\ell) & o & \rightarrow\ i \end{array} \qquad \begin{array}{llllll} \text{“may} & \text{her} & \text{have} & \text{seen”} & & \\ \textit{poter}\ . & \textit{la} & \textit{avere} & \textit{vista}\ , & & \\ (\bar{\imath}\bar{\jmath}^\ell) & (jo^{\ell\ell} i^\ell)_f & (i\ p_2^\ell) & (p_2\,o^\ell)_f & \rightarrow & \bar{\imath} \end{array}$$

An elegant and efficient solution to this problem is offered by the strategy of carrying out the computation on features in parallel with the computation on syntactic types, as suggested by [19]. We shall then introduce a second pregroup, freely generated from a set of basic feature types π_{ij}, that will be written below the original syntactic types. We introduce the basic types π_{kg}, π_{ng} with the feature specifications: k = 1 to 6 for the six verbal *persons*; n = s (singular) or p (plural) for *number*; g = f (feminine) or m (masculine) for *gender*. The set of basic feature types will include e.g.

feminine	feminine	masculine	masculine
singular	plural	singular	plural
π_{fs}	π_{fp}	π_{ms}	π_{mp}

Here are some example of the multiple type assignments allowed by the parallel computations obtained by working with the syntactic pregroup and the feature pregroup; on the left side, the constant 1 occurs in the positions in which no feature specification is needed, while on the right side it occurs to indicate that features do agree, i.e. they contract to 1.

$$\begin{array}{lllllll}
\text{She} & \text{may} & \text{have-been} & \text{arrived} & & \\
\textit{Lei} & \textit{può} & \textit{essere} & \textit{arrivata} & & \\
\pi_3 & (\pi_3^r\ \mathrm{s}_1\ \bar{\bar{i}}^{\ell}) & (i^* p_2^{*\ell}) & p_2^* & \to & \mathrm{s}_1 \\
\pi_{3f} & 1 & 1 & \pi_{3f}^r & \to & 1
\end{array}$$

$$\begin{array}{llllll}
\text{may} & \text{her} & \text{have} & \text{seen} & & \\
\textit{poter}\ . & \textit{la} & \textit{avere} & \textit{vista}\ , & & \\
(\bar{i}\bar{j}^{\ell}) & (jo^{\ell\ell} i^{\ell}) & (i\ p_2^{\ell}) & (p_2\, o^{\ell}) & \to & \bar{i} \\
1 & \pi_{3f} & 1 & \pi_{3f}^r & \to & 1
\end{array}$$

Following [19], we assume that the feature system for Italian will observe the relations given below, where the irreversible horizontal arrows forget gender, while the vertical arrows forget person.

$$\begin{tikzcd}
\pi_{kg} \arrow[r, "\neq"] \arrow[d, "\neq"'] & \pi_k \arrow[d, "\neq"] \\
\pi_{ng} \arrow[r, "\neq"'] & \pi
\end{tikzcd}$$

On this basis one can approach the analysis of full sentences in which both agreement to the subject and to the object are required, where the former is embedded into the inflected verb type and the latter is conveyed by the clitic pronoun

$$\begin{array}{lllllllll}
\text{He} & & \text{her} & \text{may} & \text{have} & & \text{seen} & & \\
\textit{Lui} & & \textit{la} & \textit{può} & \textit{avere} & & \textit{vista} & & \\
\textit{Lui} & \mathrm{C}_{13}\,(\textit{la} & & \textit{potere}) & \textit{avere} & & \textit{vista} & & \\
\pi_3 & (\pi_3^r\ \mathrm{s}_1\ \bar{\bar{i}}^{\ell}) & & (j\ o^{\ell\ell} i^{\ell}) & (i\ \bar{i}^{\ell}) & (ip_2^{\ell}) & (p_2\, o^{\ell}) & \to & \mathrm{s}_1 \\
\pi_{3m} & \pi_{3m}^r & & \pi_{3f} & 1 & (\pi^r\ \pi_{3f}) & \pi_{3f}^r & \to & 1
\end{array}$$

In the example the following relations hold

$$\pi_{3m}\ \pi_{3m}^r \to \pi_{sg}\ \pi_{sg}^r \to \pi_{ng}\ \pi_{ng}^r \to \pi\ \pi^r \to 1\ ;$$

$$\pi_{3f}\ \pi^r \to \pi_{sg}\ \pi^r \to \pi_{ng}\ \pi^r \to \pi\ \pi^r \to 1\ ;$$

$$\pi_{3f}\ \pi_{3f}^r \to \pi_{sg}\ \pi_{sg}^r \to \pi_{ng}\ \pi_{ng}^r \to \pi\ \pi^r \to 1\ .$$

3.4 Cliticization with Modal and Auxiliary Verbs

Clitics can occurr both after the modal verb, e.g. *poter.lo avere visto* "may-it have seen" and after the auxiliary verb *potere aver.lo visto* "may have-it seen". This phenomen, known as clitic movement or clitic climbing ([26]; [25]; [12]; [20]), will be addressed in the last part of the final version of the paper. Up to now we have introduced eight different basic types

$$i \to \bar{\imath}\,,\; j \to \bar{\jmath}\,,\; i^* \to \overline{\imath^*}\,,\; j^* \to \bar{\jmath}^{\,*}$$

all representing *complete* infinitival phrases; we subsume all of them under a single basic type $\bar{\bar{\imath}}$, by postulating

$$\bar{\imath}\,,\; \bar{\jmath}\,,\; \overline{\imath^*}\,,\; \bar{\jmath}^{\,*} \;\to\; \bar{\bar{\imath}}$$

3.5 Finite Verb-Forms and Declarative Sentences

To form declarative sentences, we apply the Inflector C_{jk} not only to plain infinitives such as *vedere*, but also to extended infinitives including auxiliaries and modal verbs ([8], [19]). We assign the following types to this inflector:

$$\mathrm{C}_{j\,k} : \pi_k^r\, \mathrm{s}_j\, \bar{\bar{\imath}}^{\ell}\;,\;\; \mathrm{s}_j\, \bar{\bar{\imath}}^{\ell}\;,$$

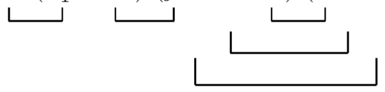
the former if the optional subject is present, as we shall assume from now on.

We conclude this section by looking at some examples of declarative sentences in the present tense involving pre-verbal cliticization.

(io) *te . lo do* (I give it to you)

$= io\;\; \mathrm{C}_{11}\,(\,te\;.\;lo\;\; dare)$

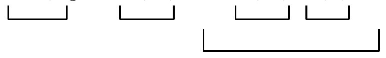
$\pi_1\; (\pi_1^r\, \mathrm{s}_1\, \bar{\bar{\imath}}^{\ell})\; (\bar{\jmath}\; \omega^{\ell\ell}\mathrm{o}^{\ell\ell}\; i^{\ell})\; (i\; \mathrm{o}^{\ell}\omega^{\ell}) \to\; \mathrm{s}_1 \qquad (\,\bar{\jmath} \to \bar{\bar{\imath}}\,)$



Dario lo vuole vedere (Dario wants to see it)

$= Dario\;\; \mathrm{C}_{13}\,(\;lo\;\; volere\;)\;\; vedere$

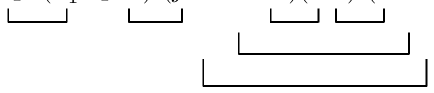
$\pi_3\;\; (\pi_3^r\, \mathrm{s}_1\, \bar{\bar{\imath}}^{\ell})\; (j\; \mathrm{o}^{\ell\ell}\; i^{\ell})\; (i\,\bar{\imath}^{\ell})(i\; \mathrm{o}^{\ell}) \to\; \mathrm{s}_1 \qquad (\,j \to \bar{\jmath} \to \bar{\bar{\imath}}\,)$



(io) *te . lo devo dare* (I must give it to you)

$= io\;\; \mathrm{C}_{11}\,(\;te\;.\;lo\;\; dovere\;)\;\; dare$

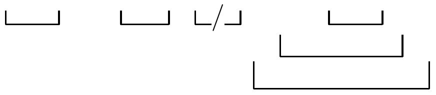
$\pi_1\;\; (\pi_1^r\, \mathrm{s}_1\, \bar{\bar{\imath}}^{\ell})\; (\bar{\jmath}\,\omega^{\ell\ell}\mathrm{o}^{\ell\ell}\; i^{\ell})(i\,\bar{\imath}^{\ell})\; (i\; \mathrm{o}^{\ell}\omega^{\ell}) \;\to\; \mathrm{s}_1 \qquad (\,\bar{\jmath} \to \bar{\bar{\imath}}\,)$



$* \,(io)$ *devo te . lo dare* ,

$= io\;\; \mathrm{C}_{11}\,(\,dovere\;)\;\; te\;.\;lo\;\; dare$

$\pi_1\;\; (\pi_1^r\, \mathrm{s}_1\, \bar{\bar{\imath}}^{\ell})\; (i\,\bar{\imath}^{\ell})(\bar{\jmath}\,\omega^{\ell\ell}\mathrm{o}^{\ell\ell}\; i^{\ell})\; (i\; \mathrm{o}^{\ell}\omega^{\ell}) \qquad (\,\bar{\jmath} \not\to \bar{\imath}\,)$



$$
\begin{array}{lllll}
 & (\textit{io}) & \textit{lo} \quad \textit{ho} \quad \textit{visto} & \text{(I have seen it)} & \\
= & \textit{io} & \mathrm{C}_{11}\,(\ \textit{lo} \quad \textit{avere}\ )\ \ \mathrm{Perf}\,(\textit{vedere}) & & \\
 & \multicolumn{4}{l}{\pi_1\ \ (\pi_1^r\ \mathrm{s}_1\ \bar{\bar{\imath}}^\ell)\ (j\ \mathrm{o}^{\ell\ell}\ i^\ell)(ip_2^\ell)(p_2\, i^\ell)(i\ \mathrm{o}^\ell)\ \to\ \mathrm{s}_1 \qquad (\, j \to \bar{j} \to \bar{\bar{\imath}}\,)}
\end{array}
$$

$$
\begin{array}{lllll}
 & {}^*\,(\textit{io}) & \textit{ho} \quad \textit{lo} \quad \textit{visto} & & \\
= & \textit{io} & \mathrm{C}_{11}\,(\ \textit{avere}\ )\ \ \mathrm{Perf}\,(\textit{lo} \quad \textit{vedere}) & & \\
 & \multicolumn{4}{l}{\pi_1\ \ (\pi_1^r\ \mathrm{s}_1\ \bar{\bar{\imath}}^\ell)\ (ip_2^\ell)(p_2\, i^\ell)(j\ \mathrm{o}^{\ell\ell}\ i^\ell)(i\ \mathrm{o}^\ell) \qquad (j \not\to i)}
\end{array}
$$

4 Conclusions

We have applied the calculus of pregroups to a selected set of Italian sentences involving clitic pronouns and agreement requirements; in doing so we have presented some evidence of the theoretical and computational advantages offered by the parallel computations of a calculus involving two pregroups: the free pregroup of syntactic types and the free pregroup independently taking care of features calculations. The paper also presents an essential, but relatively articulate, analysis of the syntax of Italian verbal constructions and cliticization domains, involving axiliaries, modal verbs and past participles.

References

1. Abrusci, M.: Classical conservative extensions of Lambek calculus. Studia Logica 71, 277–314 (2002)
2. Barr, M.: On subgroups of the Lambek pregroup. Theory and Application of Categories 12(8), 262–269 (2004)
3. Buszkowski, W.: Lambek grammars based on pregroups. In: de Groote, P., Morrill, G., Retoré, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, pp. 95–109. Springer, Heidelberg (2001)
4. Buszkowski, W.: Pregroups: Models and grammars. In: de Swart, H. (ed.) RelMiCS 2001. LNCS, vol. 2561, pp. 35–49. Springer, Heidelberg (2002)
5. Buszkowski, W.: Type logics and pregroups. Studia Logica 87(2-3), 145–169 (2007)
6. Casadio, C.: Non-commutative linear logic in linguistics. Grammars 4(3), 167–185 (2001)
7. Casadio, C.: Applying pregroups to Italian statements and questions. Studia Logica 87, 253–268 (2007)
8. Casadio, C., Lambek, J.: An algebraic analysis of clitic pronouns in Italian. In: de Groote, P., Morrill, G., Retoré, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, pp. 110–124. Springer, Heidelberg (2001)
9. Casadio, C., Lambek, J.: A tale of four grammars. Studia Logica 71(2), 315–329 (2002)
10. Casadio, C., Lambek, J. (eds.): Recent Computational Algebraic Approaches to Morphology and Syntax. Polimetrica, Milan (2008)

11. Chomsky, N.: Lectures on Government and Binding. Foris, Dordrecht (1981)
12. Klavans, J.L.: Some Problems in a Theory of Clitics. Indiana Ling. Club, Bloomington (1982)
13. Kusalik, T.: Product pregroups as an alternative to inflectors. In: Casadio, C., Lambek, J. (eds.) Recent Computational Algebraic Approaches to Morphology and Syntax, pp. 173–190. Polimetrica, Milan (2008)
14. Lambek, J.: The mathematics of sentence structure. American Math. Monthly 65, 154–169 (1958)
15. Lambek, J.: Type grammar revisited. In: Lecomte, A., Perrier, G., Lamarche, F. (eds.) LACL 1997. LNCS (LNAI), vol. 1582, pp. 1–27. Springer, Heidelberg (1999)
16. Lambek, J.: Type grammar meets German word order. Theoretical Linguistics 26, 19–30 (2000)
17. Lambek, J.: A computational algebraic approach to English grammar. Syntax 7(2), 128–147 (2004)
18. Lambek, J.: From word to sentence: a pregroup analysis of the object pronoun who(m). Journal of Logic, Language and Information 16, 303–323 (2007)
19. Lambek, J.: Exploring feature agreement in French with parallel pregroup computations. Journal of Logic, Language and Information 19(1), 75–88 (2010)
20. Monachesi, P.: A Grammar of Italian Clitics. ITK Dissertation Series, Tilburg (1995)
21. Moortgat, M.: Categorical type logics. In: van Benthem, J., ter Meulen, A. (eds.) Handbook of Logic and Language, pp. 93–177. Elsevier, Amsterdam (1997)
22. Preller, A., Lambek, J.: Free compact 2-categories. Math. Structures for Comp. Sciences 17, 309–340 (2007)
23. Preller, A., Prince, V.: Pregroup grammars with linear parsing of the French verb phrase. In: Casadio, C., Lambek, J. (eds.) Recent Computational Algebraic Approaches to Morphology and Syntax, pp. 53–84. Polimetrica, Milan (2008)
24. Stabler, E.P.: Tupled pregroup grammars. In: Casadio, C., Lambek, J. (eds.) Recent Computational Algebraic Approaches to Morphology and Syntax, pp. 23–52. Polimetrica, Milan (2008)
25. Wanner, D.: The Development of Romance Clitic Pronouns. In: From Latin to Old Romance. Mouton de Gruyter, Amsterdam (1987)
26. Zwicky, A.M., Pullum, G.K.: Cliticization vs. inflection: English n't. Language 59(3), 502–513 (1983)

Geometricity of Binary Regular Languages

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

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

Abstract. Our aim is to present an efficient algorithm for checking whether a regular language is geometrical or not, based on specific properties of its minimal automaton. Geometrical languages have interesting theoretical properties and they provide an original model for off-line temporal validation of real-time softwares. As far as implementation is concerned, the regular case is of practical interest, which motivates the design of an efficient geometricity test addressing the family of regular languages. This study generalizes the algorithm designed by the authors for the case of prolongable binary regular languages.

Keywords: regular language, minimal automaton, geometrical language.

1 Introduction

Geometrical languages were introduced in [2], with two motivations: the modeling of real-time task systems and the generalization of the class of regular languages commonly used to solve temporal validation problems. Computing the feasibility of a real-time software [1] consists of checking whether there exists a scheduling sequence that leads every task to be completed by its deadline. It can be achieved through a model based on regular languages [5] or a model based on discrete geometry [7,5]. Geometrical languages are intended to connect these two models. The challenge is to make use of geometrical properties in order to design new and efficient automata-based algorithms.

Given a d-dimensional space associated with an alphabet $\Sigma = \{a_1, \ldots, a_d\}$, a geometrical figure is a subset of points of $\mathbb{N}^d$ including the origin of the reference and such that, for any point in the figure, there exists a trajectory (from the origin) to this point. A geometrical figure can be seen as a (not-necessarily-finite) automaton: each point of the figure is a state, the origin of the reference is the initial state, every state is final, and given two points $P = (x_1, \ldots, x_d)$ and $P' = (x'_1, \ldots, x'_d)$ of the figure there exists an implicit transition with label a_i from P to P' if and only if $x'_i = x_i + 1$ and $\forall j \neq i$, $x'_j = x_j$. Hence we can define the language of a geometrical figure as well as the geometrical figure of a language. Finally, a language is said to be geometrical if and only if the language of its prefixes is equal to the language of its geometrical figure. Studying the properties of the (not-necessarily-regular) geometrical languages turns out to be of both theoretical and practical interest as reported in [2].

Geometrical regular languages have been characterized in [3] both in terms of languages and in terms of automata. These characterizations concern the whole family of regular languages. However, the algorithm presented in [3] only addresses the case where the construction of infinite and periodic trajectories is possible from any point of the figure, that is the case of prolongable languages (L is a prolongable language if and only if for all u in L, there exists x in Σ such that $u \cdot x \in L$). In this paper we show how to handle the general case and we present a new efficient algorithm for implementing the automaton-based characterization. As in [3], we investigate binary languages, geometrical arguments being more easily developed in the 2-dimensional case.

The following two sections recall fundamental notions concerning languages, finite automata and d-dimensional geometrical languages. The next sections are devoted to the 2-dimensional case and they address not-necessarily-prolongable languages. Section 4 introduces new properties of the trajectories of a geometrical figure that allow us to prove the correctness of the geometricity test presented in Section 5. A polynomial algorithm is described and analyzed in Section 6.

2 Preliminaries

Let us first review basic notions concerning regular languages and finite automata. For a comprehensive treatment of this domain, reference [4] can be consulted.

Let Σ be a nonempty finite set of symbols, called the *alphabet*. A *word* over Σ is a finite sequence of symbols, usually written $x_1 x_2 \cdots x_n$. The *length* of a word u, denoted by $|u|$, is the number of symbols in u. The *empty word*, denoted by ε, has a length equal to zero. If $u = x_1 \cdots x_n$ and $v = y_1 \cdots y_m$ are two words over the alphabet Σ, their concatenation $u \cdot v$, usually written uv, is the word $x_1 \cdots x_n y_1 \cdots y_m$. Let Σ^* be the set of words over Σ. Given two words u and w in Σ^*, u is said to be a *prefix* (resp. a *suffix*) of w if there exists a word v in Σ^* such that $uv = w$ (resp. $vu = w$). Let $u \in \Sigma^*$ and let $0 \leq k \leq |u|$; the prefix (resp. suffix) of length k of u is denoted by $[u]_k$ (resp. $[u]^k$). The set of words of Σ^* of length less than k is denoted by $\Sigma^{<k}$. A *language* L over Σ is a subset of Σ^*. The set of prefixes of the words of the language L is denoted by $\mathrm{Pref}(L)$. A language L is said to be *prefix-closed* if and only if $L = \mathrm{Pref}(L)$. The set of *regular* languages over an alphabet Σ contains the empty set and the set $\{a\}$ for all $a \in \Sigma$. and it is closed under finite concatenation, finite union and star.

A *deterministic finite automaton* (DFA) is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ where Q is a finite *set of states*, δ is a mapping from $Q \times \Sigma$ to Q, s_0 is the *initial state* and T is the *set of final states*. For all $(p, x) \in Q \times \Sigma$, we will write $p \cdot x$ instead of $\delta(p, x)$; the 3-tuple (p, x, q) in $Q \times \Sigma \times Q$ is said to be *a transition* if and only if $q = p \cdot x$. A DFA $\mathcal{A}$ is said to be *complete* if for any $q \in Q$ and any $a \in \Sigma$, $|q \cdot a| = 1$. In a complete DFA there may exist a *sink state* σ such that $\sigma \notin T$ and, for all $x \in \Sigma$, $\sigma \cdot x = \sigma$.

Let $p \in Q$ and $u = u_1 \cdots u_l \in \Sigma^*$. The *path* (p, u) of length l starting from p and labeled by u is the sequence of transitions (p_0, u_1, p_1), ..., $(p_{l-1}, u_l, p_l))$, with $p_0 = p$. A path (p, u) is said to be *successful* if $p = s_0$ and $p \cdot u \in T$. The language $L(\mathcal{A})$ *recognized* by the DFA $\mathcal{A}$ is the set of words that are labels of successful paths. Kleene's theorem [6] states that a language is recognized by a finite automaton if and only if it is regular. The *left language* $\overleftarrow{L}_q^{\mathcal{A}}$ (resp. *right language* $\overrightarrow{L}_q^{\mathcal{A}}$) of a state q is the set of words w such that there exists a path in $\mathcal{A}$ from s_0 to the state q (resp. from q to a final state) with w as label. A DFA $\mathcal{A}$ is said to be *accessible* if for any $q \in Q$ there exists a path from s_0 to q. A complete and accessible DFA $\mathcal{A}$ is *minimal* if and only if any two distinct states of $\mathcal{A}$ have distinct right languages. According to the theorem of Myhill-Nerode [8,9], the minimal DFA of a regular language is unique up to an isomorphism.

Let Σ be an ordered alphabet of size d and $\prec$ be the graded lexicographic order over Σ^*. Let $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ be a DFA. The d-ary *prefix tree* of $\mathcal{A}$ is defined from the mapping $\varphi : Q \rightarrow \Sigma^*$ such that $\varphi(q) = \min_{\prec}\{u \in \Sigma^* \mid s_0 \cdot u = q\}$. The set $\varphi(Q)$ is a prefix-closed set of Σ^*. The labeled tree $T_{\mathcal{A}} = (V, U, \Sigma)$ with $V = \varphi(Q)$ and $U = \{(\varphi(p), a, \varphi(q)) \mid \varphi(p) \prec \varphi(q) \wedge (p, a, q) \in \delta\}$ is the *prefix tree* of $\mathcal{A}$. The one-to-one φ mapping from Q to V is the *canonical labeling* of the automaton $\mathcal{A}$.

Let us denote by $<$ the lexicographic order over Σ^*. Given two words u and v in Σ^*, u is said to be *smaller* (resp. *greater*) than v if and only if $u < v$ (resp. $v < u$) and u is said to be *longer* (resp. *shorter*) than v if and only if $|u| > |v|$ (resp. $|v| > |u|$).

3 Geometrical Languages

Let us now review basic definitions and properties of geometrical languages, as introduced in [2]. Let d be a positive integer and $\Sigma = \{a_1, \ldots, a_d\}$ be an alphabet. The Parikh mapping [10] $c : \Sigma^* \longrightarrow \mathbb{N}^d$ maps a word w to its d-dimensional coordinate vector $(|w|_{a_1}, \ldots, |w|_{a_d})$. In particular, for all $1 \leqslant k \leqslant d, c(a_k)$ is the coordinate vector $(0, 0, \ldots, 1, \ldots, 0)$ where 1 is in the k-th position. The point with coordinate $(0, 0, \ldots, 0)$ is denoted by $\mathcal{O}$. For any point P in $\mathbb{N}^d$, we write P instead of $\overrightarrow{\mathcal{O}P}$. The *level* of the point $P = (x_1, \ldots, x_d)$ is $\text{level}(P) = x_1 + \ldots + x_d$.

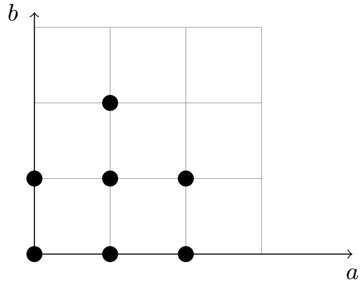


Fig. 1. $F_1 = \{(0,0), (0,1), (1,0), (1,1), (1,2), (2,0), (2,1)\}$

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

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

Figure 1 represents a 2-dimensional geometrical figure. The *geometrical figure of a language* $L \subseteq \Sigma^*$ is defined by $\mathcal{F}(L) = \bigcup_{w \in \mathrm{Pref}(L)} \mathrm{points}(\mathcal{O}, w)$. The *language of a geometrical figure* F is defined by $\mathcal{L}(F) = \{\mathrm{word}(T) \mid T \in \mathrm{Traj}(\mathcal{O}, F)\}$.

Definition 2. *The language L is* geometrical *if and only if* $\mathrm{Pref}(L) = \mathcal{L}(\mathcal{F}(L))$.

For any language L, $\mathrm{Pref}(L) \subseteq \mathcal{L}(\mathcal{F}(L))$. Some languages however are such that $\mathcal{L}(\mathcal{F}(L)) \nsubseteq \mathrm{Pref}(L)$. For instance, the two languages $\{a, ba\}$ and $\{ab, ba\}$ have the same geometrical figure; the former one is not geometrical, whereas the latter one is. The next proposition provides a characterization of geometricity in terms of languages.

Proposition 1. *[2] Let $\Sigma = \{a_1, \ldots, a_d\}$ and $L \subseteq \Sigma^*$. The following two conditions are equivalent:* (1) *L is geometrical.*
(2) $\forall u, v \in \mathrm{Pref}(L)$, $\underbrace{\exists k, 1 \le k \le d,\ c(u \cdot a_k) = c(v)}_{(*)} \Rightarrow u \cdot a_k \in \mathrm{Pref}(L)$

For example, let $\Sigma = \{a, b\}$ and $L = \{a, ba, bb\}$. Then the Condition $(*)$ is satisfied by the words $u = a$ and $v = ba$, since $c(u) + (0, 1) = c(v)$. According to Proposition 1 and since $ab \notin \mathrm{Pref}(L)$, L is not geometrical.

We now translate Proposition 1 in terms of automata. In the following, L is a regular language, $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ is the minimal DFA of $\mathrm{Pref}(L)$ and $F = \mathcal{F}(L)$ is the geometrical figure of L. The sink state of $\mathcal{A}$ is denoted by σ (if it exists). Proposition 2 is restated from Corollary 11 and Proposition 13 of [3]. The idea is that a transition should exist in the automaton $\mathcal{A}$ every time an implicit transition exists in the figure F.

Definition 3. *The relation* State *in $\mathbb{N}^d \times Q$ is defined as follows: given P in $\mathbb{N}^d$ and p in Q, (P, p) is in* State, *if and only if either $P \notin F$ and $p = \sigma$, or $P \in F$ and there exists $u \in L(\mathcal{A})$ such that $P = c(u) \in F$ and $p = s_0 \cdot u \in Q$.*

Proposition 2. *Let $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ be the minimal DFA of* $\mathrm{Pref}(L)$. *The following two conditions are equivalent:*
(1) *The language L is geometrical.*
(2) *The* State *relation is a mapping from $\mathbb{N}^d$ to Q, such that:*
$\forall P \in \mathbb{N}^d$, $\forall i \mid 1 \le i \le d$, $P - c(a_i) \in F \Rightarrow \mathrm{State}(P - c(a_i)) \cdot a_i = \mathrm{State}(P)$.

Definition 4. *The subset $F_Q = \bigcup_{u \in \varphi(Q)} \mathrm{points}(\mathcal{O}, u)$ of F, where φ is the canonical labeling of $\mathcal{A}$, is called the* basic geometrical figure *of $L(\mathcal{A})$.*

Notice that the mapping $\mathrm{Point} = c \circ \varphi$ from Q to F_Q is a one-to-one mapping; the inverse mapping of Point is the restriction of State to F_Q.

4 Words, Paths and Trajectories

We now revisit the notion of trajectory in order to address the case of non prolongable languages. In the following, L is a binary regular language and $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ is the minimal DFA of $\mathrm{Pref}(L)$, with $n = |Q|$. Notice that if L is not prolongable, then there exist a sink state σ and a state $\sigma' \in T$ such that $\sigma' \cdot a = \sigma' \cdot b = \sigma$. We set $Q = \{s_0 = 0, 1, \ldots, \sigma' = n-2, \sigma = n-1\}$. The geometrical figure F of L is supposed to be drawn so that points with the same level lie on a horizontal line (see Figure 2 for example).

4.1 Basic Definitions and Properties

According to Proposition 2, the image of a trajectory $(P, u) \in F \times \Sigma^*$ by the State mapping is the path $(\mathrm{State}(P), u)$. Conversely, a path (p, u) defines a trajectory (P, u) for any point P such that $\mathrm{State}(P) = p$. In the following we will consider specific paths (p, u), called *state paths*, such that for all $0 \le i < |u|$, the label u_{i+1} of the transition (p_i, u_{i+1}, p_{i+1}) only depends on the state p_i. For example, a trajectory that gives priority to moves to the right is associated to a state path, since it is generated by a word containing as many symbols b as possible. For a state path, it is equivalent to consider the transition sequence (p, u) or the associated state sequence $(p_0, \ldots, p_{|u|})$. A trajectory is *periodic* if the associated path is a periodic sequence (of transitions or of states according to the case). A sequence with μ as pre-period and π as period is said to be a (μ, π)*-periodic sequence*. The point $P_{j+\pi}$ is said to be *the shifted point* of P_j.

Definition 5. *Let p be a state in $Q \setminus \{\sigma\}$ such that $\overrightarrow{L}_p^{\mathcal{A}}$ is a finite language. Then the* order *of p is defined by $\rho(p) = \max\{|w| \mid w \in \overrightarrow{L}_p^{\mathcal{A}}\}$. We set $\rho(\sigma) = -1$.*

By definition, $\rho(\sigma') = 0$, and for all $p \in Q \setminus \{\sigma, \sigma'\}$, $\rho(p) \ge 1$. If $\overrightarrow{L}_p^{\mathcal{A}}$ is a finite language then $0 \le \rho(p) \le n-2$; otherwise the order of p is unbounded. A state p (resp. a point P) is said to be *bounded* if $\rho(p) \le n-2$ (resp. $\rho(\mathrm{State}(P)) \le n-2$) and *unbounded* otherwise. Let Q' (resp. Q'') be the set of unbounded (resp. bounded) states. We set $n' = |Q'|$ and $n'' = |Q''|$.

Definition 6. *Given a positive integer k, a word u of a language L is said to be k-*extensible *if and only if there exists a word $v \in \Sigma^+$ such that $|v| = k$ and $uv \in L$. A word u is said to be* extensible *if and only if it is k-extensible for any integer k and* inextensible *otherwise.*

A language L is *prolongable* if and only if for all u in L, there exists x in Σ such that $u \cdot x \in L$. We have: L is prolongable $\Leftrightarrow \forall u \in L$, u is extensible $\Leftrightarrow \forall p \in Q \setminus \{\sigma\}$, p is unbounded. The label $u = u_1 \cdots u_l$ of a path (p, u) is inextensible if and only if there exists k, $0 \le k \le l$, such that $p_k = p \cdot u_1 \cdots u_k$ is a bounded state.

4.2 Right Trajectories of a Point

We first define the *right sequence* of a point P of F and then motivate the notions introduced in this section.

Definition 7. *The* right sequence $\mathrm{RS}(P) = (P_k)_{0 \le k}$ *of a point P of F is defined by $P_0 = P$ and for all $0 \le k$, P_k is the rightmost point at level* $\mathrm{level}(P) + k$ *in F that is accessible from P.*

The *left sequence* $\mathrm{LS}(R)$ of a point R of F can be defined similarly. Let S, $P = S + c(a)$ and $R = S + c(b)$ be three points of F. The right sequence of P and the left sequence of R are of principal interest since an efficient geometricity test can be developed based on the shape of the region that they delimit.

Let $p = \mathrm{State}(P)$. According to Definition 7, for all $0 \le k$, P_k is a point of $\mathrm{RS}(P)$ if and only if it belongs to the trajectory starting from P associated with the greatest word w_k of $\overrightarrow{L}_p^{\mathcal{A}} \cap \Sigma^k$. If L is prolongable, for all $0 \le k$, w_k is a prefix of the greatest word of $\overrightarrow{L}_p^{\mathcal{A}}$; the sequence $\mathrm{RS}(P)$ is then the trajectory[1] associated with this word. If L is not prolongable, $\mathrm{RS}(P)$ is generally a sequence of segments computed from a sequence of inextensible words of $\overrightarrow{L}_p^{\mathcal{A}}$, possibly added with the greatest extensible word of $\overrightarrow{L}_p^{\mathcal{A}}$. Although the sequence $\mathrm{RS}(P)$ is generally not a trajectory, it is possible to define the *right trajectory* $\mathrm{RT}(P)$ of an unbounded point P as follows.

Definition 8. *Let P be an unbounded point and $p =$* $\mathrm{State}(P)$. *The* right trajectory $\mathrm{RT}(P)$ *of P is defined by the greatest extensible word $u_p^>$ of the set $\overrightarrow{L}_p^{\mathcal{A}}$.*

The word $u_p^> = u_1 u_2 \cdots$ can be computed as follows: $p_0 = p$ and for $0 \le k$, $u_{k+1} = b$ if $p_k \cdot b \in Q'$ and $u_{k+1} = a$ otherwise. Since for any unbounded state q the condition $q \cdot b \in Q'' \Rightarrow q \cdot a \in Q'$ holds, we have for all $0 \le k$, $p_k \in Q'$.

Lemma 1. *Let P be an unbounded point. The right trajectory* $\mathrm{RT}(P)$ *of P is a (μ, π)-periodic sequence, with $\mu + \pi \le n' \le n - 2$.*

Let L be non prolongable and p be unbounded. Then the extensible word $u_p^>$ is not necessarily the greatest word of $\overrightarrow{L}_p^{\mathcal{A}}$. Indeed, for any $v \in \Sigma^*$ such that va is a prefix of $u_p^>$ and $p \cdot vb \ne \sigma$, the state $p \cdot vb$ is bounded. As a consequence, for all $w \in \overrightarrow{L}_{p \cdot vb}^{\mathcal{A}}$, we have $vbw > va$ and thus $vbw > u_p^>$.

Let u be the greatest inextensible word of $\overrightarrow{L}_p^{\mathcal{A}}$. If $u < u_p^>$, then $u_p^>$ is the greatest word of $\overrightarrow{L}_p^{\mathcal{A}}$. Any inextensible word of $\overrightarrow{L}_p^{\mathcal{A}}$ is smaller than $u_p^>$, and thus, for all P such that $\mathrm{State}(P) = p$, the right sequence $\mathrm{RS}(P)$ and the right trajectory $\mathrm{RT}(P)$ of P coincide. Otherwise, the segment (P, u) is the first segment of the right sequence $\mathrm{RS}(P)$ and the word $u = u_1 \cdots u_l$ is computed as follows: for all $0 \le k < l$, $u_{k+1} = b$ if $p_k \cdot b \ne \sigma$ and $u_{k+1} = a$ otherwise. Since u is both inextensible and the greatest word of $\overrightarrow{L}_p^{\mathcal{A}}$ we have $p_l = \sigma'$ and, for all $0 \le k < l$, $p_k \cdot b = \sigma \Rightarrow p_k \cdot a \ne \sigma$.

Definition 9. *The sequence* $\mathrm{U}(p) = (u_p^i)_{1 \le i}$ *is defined as follows:*
(1) *The word u_p^1 is the greatest word of the set $\overrightarrow{L}_p^{\mathcal{A}}$.*
(2) *For all $k \ge 1$, if $u_p^k = u_p^>$ then* $\mathrm{U}(p) = (u_p^i)_{1 \le i \le k}$ *else u_p^{k+1} is the greatest word of $\overrightarrow{L}_p^{\mathcal{A}}$ smaller and longer than u_p^k.*

[1] It is called the rightmost trajectory of P in [3].

Table 1. The transition function δ of $\mathcal{A}$ (with $4 \leq i \leq 10$)

δ	s_0	p_0	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p'_3	p'_4	p'_5	p'_6	p'_7	r_0	r_1	r_2	r_3	r_i	r_{11}	r'_3	r'_4	σ'
a	p_0	p_1	p_2	p_3	p_4	p_5	p_6	σ	p_8	p_4	σ	p'_7	σ	σ	p'_6	σ	σ	r'_3	σ	r_{i+1}	σ'	r'_4	σ'	σ
b	r_0	σ	σ	p'_3	σ	p'_5	σ	p_7	p'_8	σ	p'_4	σ'	p'_6	σ'	σ	r_1	r_2	r_3	r_4	σ	r_3	σ	σ	σ

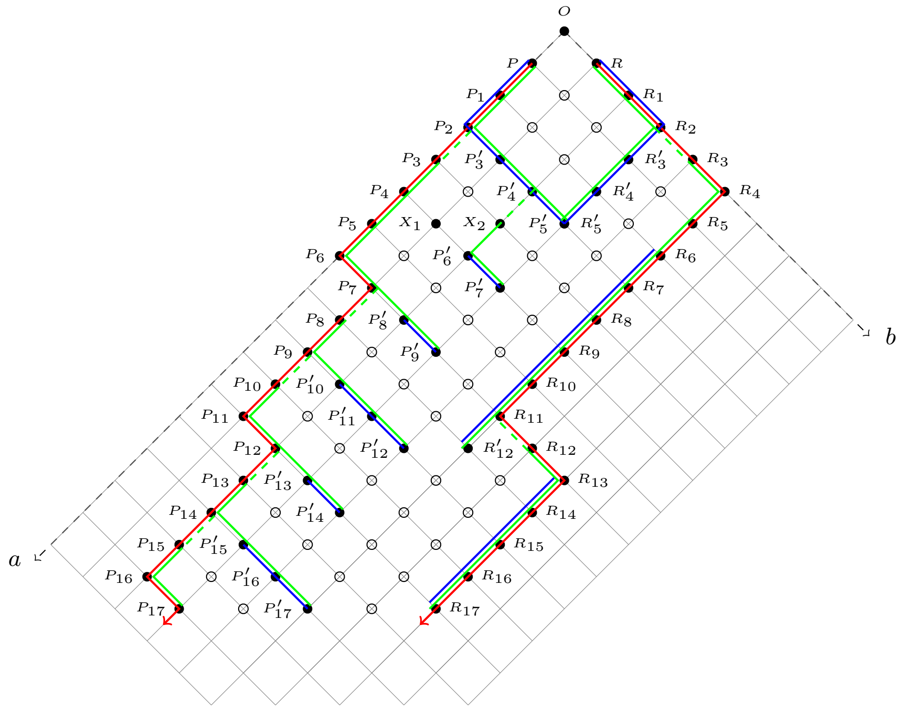


Fig. 2. The right (resp. left) trajectory and sequence starting from P (resp. R)

Example 1. *Let $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ be the minimal DFA the transition function δ of which is shown in the Table 1. The geometrical figure F of $L(\mathcal{A})$ is represented by the Figure 2. We assume that $P = P_0$ and that for all $0 \leq k \leq 3$, $P'_k = P_k$. For all $0 \leq k \leq 8$, $\text{State}(P_k) = p_k$ and p_k is an unbounded state. For all $k = 3, 4, 6$, $\text{State}(P'_k) = p'_k$; $\text{State}(P'_5) = \sigma'$. $\text{State}(X_1) = p'_5$; $\text{State}(X_2) = p'_7$. For all $3 \leq k \leq 7$, p'_k is a bounded state. The right trajectory $\text{RT}(P) = (P_k)_{0 \leq k}$ (in red lines) is the $(\mu = 4, \pi = 5)$-periodic sequence defined by the word $aaaa(aabaa)^\omega$. The right sequence $\text{RS}(P) = (P'_k)_{0 \leq k}$ (in blue lines) is defined from the sequence $\text{U}(p)$ (with $p = p_0$) that contains the words $u_p^1 = aabbb$, $u_p^2 = aabbaab$, $u_p^3 = aaaaaabbb$, $u_p^4 = aaaaaabaabbb$, $u_p^5 = aaaaaabaaaabbb$, ... Hence $\text{RS}(P)$ is the sequence of segments $((P, aabbb)$, (P'_6, b), (P'_8, b), (P'_{10}, bb), $(P'_{13}, b), \ldots)$.*

The sequence $\text{U}(p)$ is decreasing w.r.t. the lexicographic order and increasing w.r.t. the length order. If p is a bounded state, it is finite. Otherwise, it may be finite or infinite.

The sequence $\mathrm{U}(p)$ can be computed according to Proposition 3 and it allows us to compute the right sequence $\mathrm{RS}(P)$ of P following Proposition 4.

Proposition 3. *Let p be an unbounded state and u_p^i be a word of $\mathrm{U}(p)$, with $u_p^i \neq u_p^>$. Then the set $A = \{v \in \Sigma^* \mid (v \text{ is a prefix of } u_p^i) \wedge (\rho(p \cdot va) > \rho(p \cdot vb))\}$ is not empty and $u_p^{i+1} = v'aw'$, where v' is the longest word in A and w' the greatest word of $\overrightarrow{L}^{\mathcal{A}}_{p \cdot v'a}$.*

Proposition 4. *Let P be a point of F and $p = \mathrm{State}(P)$. Let $l_0 = 0$ and for all $1 \leq j$, $l_j = |u_p^j| + 1$. The* right sequence *$\mathrm{RS}(P) = (P_k)_{0 \leq k}$ is generated by the sequence $(P_{l_j}, z_p^{j+1})_{0 \leq j}$ of segments such that: $P_{l_j} = P + c([u_p^{j+1}]_{l_j})$ and $z_p^{j+1} = [u_p^{j+1}]^{l_{j+1} - l_j - 1}$.*

We now present an alternative way to compute the sequence $\mathrm{U}(p)$ in order to study the periodicity of $\mathrm{RS}(P)$. Let $\mathrm{X}(p) = (x_i)_{1 \leq i}$ be the prefix sequence of $u_p^>$ and $\mathrm{B}(p)$ be the subsequence of $\mathrm{X}(p)$ such that $x \in \mathrm{B}(p)$ if and only if $x \in \mathrm{X}(p)$ and $p \cdot xb$ is a bounded state. The *depth* $d(x)$ of a word x in $\mathrm{B}(p)$ is the maximal length of a word in $\mathrm{U}(p \cdot xb)$; we have $d(x) = |x| + 1 + \rho(p \cdot xb)$. We consider the subsequence $\mathrm{V}(p)$ of $\mathrm{B}(p)$ such that $x \in \mathrm{V}(p)$ if and only if $x \in \mathrm{B}(p)$ and xb is a prefix of some word in $\mathrm{U}(p)$. Let x_{min} be the first word in $\mathrm{B}(p)$. Let x and x' be two consecutive words in $\mathrm{V}(p)$, with $x < x'$. According to Definition 9, we have then $d(x') > d(x)$; this condition is called *the length condition*. A word $x' \in \mathrm{B}(p)$ is in $\mathrm{V}(p)$ if and only if either $x' = x_{min}$ or there exists a word $x \in \mathrm{V}(p)$ such that $d(x') > d(x)$. We set $top(x_{min}) = 0$ and $top(x') = d(x) + 1$. For all $x \in \mathrm{V}(p)$, the sequence $\mathrm{U}(p.xb)$ is finite. The sequence $\mathrm{W}(x) = (u \in \mathrm{U}(p.xb) \mid top(x) \leq |xbu|)$ is useful to factorize the sequence $\mathrm{U}(p)$.

Lemma 2. *Let $p \in Q$. We have $\mathrm{U}(p) = (xb\ \mathrm{W}(x))_{x \in \mathrm{V}(p)}$.*

According to Lemma 2, the periodicity of the right sequence $\mathrm{RS}(P)$ depends on the periodicity of the subsequence $\mathrm{V}(p)$. Let us first give a precise definition of this notion.

Definition 10. *Let p be an unbounded state and X' be a subsequence of $\mathrm{X}(p)$. We say that X' is* periodic *if and only if there exists $u_1 \in \mathrm{X}(p)$ and $u_2 \in \Sigma^*$ such that for all $x \in \mathrm{X}(p)$, $u_1 \leq x \Rightarrow (x \in X' \Leftrightarrow xu_2 \in X')$. Let u_1 be the smallest such word. Then X' is $(|u_1|, |u_2|)$-periodic and u_2 is the* period word *of X'.*

According to Lemma 1, the sequence $\mathrm{X}(p)$ is (μ, π)-periodic, Let us denote by ζ its period word. Since $p \cdot x = p \cdot x\zeta$ and $\rho(p \cdot xb) = \rho(p \cdot x\zeta b)$, we get that for all $x \in \mathrm{X}(p)$, $\mu \leq |x| \Rightarrow (x \in \mathrm{B}(p) \Leftrightarrow x\zeta \in \mathrm{B}(p))$. Hence the sequence $\mathrm{B}(p)$ is (μ, π)-periodic. This condition is not necessarily satisfied by $\mathrm{V}(p)$, due to the length condition. The following lemma proves that the sequence $\mathrm{V}(p)$ is however periodic.

Lemma 3. *Let p be an unbounded state such that the sequence $\mathrm{V}(p)$ is infinite. Then $\mathrm{V}(p)$ is (κ, π)-periodic, with $\kappa \leq \mu + \pi + n - 3 \leq 2(n-2) - 1$.*

Proposition 5. *Let P be an unbounded point such that $\mathrm{State}(P) = p$. We suppose that the sequence $\mathrm{V}(p)$ is infinite. Then the right sequence $\mathrm{RS}(P)$ of P is a (μ', π)-periodic sequence, with $\mu' \leq \mu + 2\pi + n - 3 \leq 3(n-2) - 1$.*

Example 2. *(Example 1 continued) We have* $\mathrm{V}(p) = (x_2 = aa, x_7 = x_2aaaab, x_9 = x_7aa, x_{12} = x_9aab, x_{14} = x_{12}aa, \ldots)$ *and* $\zeta = aaaab$*. Since* $x_5 \leq x \Rightarrow (x \in \mathrm{V}(p) \Leftrightarrow x\zeta \in \mathrm{V}(p))$*, we have* $u_1 = aaaaa$ *and* $u_2 = \zeta$*. Moreover* $top(x_7) = top(x_{12})$*; hence the sequences* $\mathrm{V}(p)$ *and* $\mathrm{RS}(P)$ *are both* $(\mu' = 5, \pi = 5)$*-periodic.*

5 A Geometricity Test for Binary Regular Languages

We now introduce the notions of boundary, hole and heart associated with a point. These notions make it possible to state a necessary and sufficient condition for a regular language to be geometrical.

Definition 11. *The right boundary* $\mathrm{RB}(P)$ *of a point* P *is the region delimited by (and including) its right trajectory* $\mathrm{RT}(P)$ *and its right sequence* $\mathrm{RS}(P)$*.*

Proposition 6. *There are two cases:*
Case 1 (P *is bounded): the trajectory* $\mathrm{RT}(P)$ *is not defined and the right boundary* $\mathrm{RB}(P)$ *coincides with the finite sequence* $\mathrm{RS}(P)$*.*
Case 2 (P *is unbounded): if* $\mathrm{RS}(P)$ *is* (μ', π)*-periodic, then the right boundary* $\mathrm{RB}(P)$ *is a* (μ', π)*-periodic region, with* $\mu' \leq \mu + 2\pi + n - 3 \leq 3(n-2) - 1$*. Otherwise* $\mathrm{RB}(P)$ *and* $\mathrm{RT}(P)$ *coincide on and after some level* k*, with* $k \leq 3(n-2) - 1$*.*

Now we have to go from right to left. The left sequence of a point P such that $\mathrm{State}(P) = p$ is defined by a sequence $\mathrm{U}'(p)$ of words of $\overrightarrow{L}_p^{\mathcal{A}}$ that is increasing w.r.t. the lexicographic order and w.r.t. the length order. The left trajectory is associated with the smallest extensible word of the set $\overrightarrow{L}_p^{\mathcal{A}}$. Finally, the following proposition is the left equivalent of Proposition 1, Proposition 5 and Proposition 6.

Proposition 7. *The left trajectory* $\mathrm{LT}(P)$ *of an unbounded point* P *is a* (λ, τ)*-periodic sequence, with* $\lambda + \tau \leq n' \leq n - 2$*. The left sequence* $\mathrm{LS}(P)$ *is a* (λ', τ)*-periodic sequence, with* $\lambda' \leq \lambda + 2\tau + n - 3 \leq 3(n-2) - 1$*. Concerning the left boundary* $\mathrm{LB}(P)$*, there are two cases:*
Case 1 (P *is bounded): the trajectory* $\mathrm{LT}(P)$ *not being defined, the left boundary* $\mathrm{LB}(P)$ *coincides with the finite sequence* $\mathrm{LS}(P)$*.*
Case 2 (P *is unbounded): if* $\mathrm{LS}(P)$ *is periodic, then the left boundary* $\mathrm{LB}(P)$ *is a* (λ', τ)*-periodic region, with* $\lambda' \leq \lambda + 2\tau + n - 3 \leq 3(n-2) - 1$*. Otherwise* $\mathrm{LB}(P)$ *and* $\mathrm{LT}(P)$ *coincide on and after some level* k*, with* $k \leq 3(n-2) - 1$*.*

We now revisit the notion of hole [3] of a point and we introduce the notion of heart. Then we restate the hole condition in the general case of binary regular languages.

If two points $P = (x, y)$ and $P' = (x', y')$ of a geometrical figure F have the same level (that is $x + y = x' + y'$), we say that (P, P') is a *pair of adjacent points* (we write $P < P'$) if and only if $x' = x - 1$ and $y' = y + 1$. We also say that such a pair (P, P') is a *conflict* if $\mathrm{State}(P) \cdot b \neq \mathrm{State}(P') \cdot a$. According to Proposition 2, a regular language is geometrical if and only if there exists no conflict in its geometrical figure. The main property is that it is sufficient to test the existence of a conflict only into some regions of the figure and only up to a bounded depth.

Definition 12. *We assume that the points S, $P = S + c(a)$ and $R = S + c(b)$ are in F. The* hole *of the point S is the region of $\mathbb{N}^2$ delimited by the trajectories $\mathrm{RT}(P)$ and $\mathrm{LT}(R)$. The* heart *of the point S is the region of $\mathbb{N}^2$ delimited by the sequences $\mathrm{RS}(P)$ and $\mathrm{LS}(R)$.*

Let $\mathrm{RT}(P) = (P_k)_{0\leq k}$, $\mathrm{RS}(P) = (P'_k)_{0\leq k}$, $\mathrm{LT}(R) = (R_k)_{0\leq k}$ and $\mathrm{LS}(R) = (R'_k)_{0\leq k}$. By construction, for any point S in F, the heart of S contains no point of F, except for points that belong to both $\mathrm{RT}(P)$ and $\mathrm{LT}(R)$. The depth of the hole of S is infinite if the trajectories $\mathrm{RT}(P)$ and $\mathrm{LT}(R)$ do not intersect; otherwise it is defined by $h_S = \min\{k > 0 \mid P_k = R_k\}$.

Definition 13. *We assume that the points S, $P = S + c(a)$ and $R = S + c(b)$ are in F. The* heart condition *is said to be satisfied by the point S if and only for all $0 \leq k < h_S$, the pair (P'_k, R'_k) is not a conflict. The* hole condition *is said to be satisfied by the point S if and only if the following two conditions hold:*
(1) *There is no conflict inside the right boundary of P nor inside the left boundary of R.*
(2) *The heart condition is satisfied by the point S.*

Since any point P of a geometrical figure (except for the origin) has at least one parent, the definition of a reverse right (or left) segment[2] starting from P is still possible.

Definition 14. *Let P be a point of a geometrical figure F.*
(1) The reverse right segment from P is a finite sequence $(P_i)_{0\leq i\leq f}$ such that $P_0 = P$, $P_f = \mathcal{O}$, and for all $0 \leq i < f$, $P_{i+1} = P_i - c(a)$ if $P_i - c(a) \in F$ and $P_{i+1} = P_i - c(b)$ otherwise.
(2) The reverse left segment from P is a finite sequence $(P_i)_{0\leq i\leq f}$ such that $P_0 = P$ and $P_f = \mathcal{O}$ and for all $0 \leq i < f$, $P_{i+1} = P_i - c(b)$ if $P_i - c(b) \in F$ and $P_{i+1} = P_i - c(a)$ otherwise.

If the points S, $S - c(a)$ and $S - c(b)$ are in F, then the reverse right segment from $S - c(b)$ and the reverse left segment from $S - c(a)$ necessarily intersect. The following lemma enlightens the relation between reverse segments and direct trajectories.

Lemma 4. *Let $(P_i)_{0\leq i\leq f}$ be a reverse right segment. If P_0 is unbounded, for all $0 \leq i \leq f$, P_0 belongs to the right trajectory of P_i. Otherwise, for all $0 \leq i \leq f$, either P_0 belongs to the right sequence of P_i or P_0 is inside the right boundary of P_i. Finally, for all $0 \leq i \leq f$, P_0 belongs to the right boundary of P_i. A similar property exists for a reverse left segment.*

Right and left sequences delimit the border of the heart, whereas right and left trajectories delimit the border of the hole and enable moving from the end of one segment of a right or left sequence to the beginning of the next one. Reverse right or left trajectories are useful to analyze the origin of a conflict and to prove the next proposition.

Proposition 8. *Let L be a binary language and $\mathcal{A} = (Q, \Sigma, \delta, s_0, T)$ be the minimal DFA of $\mathrm{Pref}(L)$. The following two conditions are equivalent:*
(1) *The language L is geometrical.*
(2) *For every point P in the basic figure F_Q, the hole condition is satisfied by P.*

[2] This segment is called reverse rightmost (or leftmost) trajectory in [3].

6 A Polynomial Algorithm for Checking Geometricity

From Proposition 8, a straightforward algorithm for checking the geometricity of a binary language can be designed, based on the fact that the hole condition must be satisfied for every point in F_Q. We consider three points S, $P = S + c(a)$ and $R = S + c(b)$ of F, such that $s = \mathrm{State}(S)$, $p = \mathrm{State}(P)$, $r = \mathrm{State}(R)$. The algorithm checks that there is no conflict inside the following three regions: the right boundary of P, the left boundary of R and the heart of S. By making use of the periodicity of the different trajectories or sequences, it is shown that for each region only a small number of levels needs to be inspected.

6.1 No Conflict Inside the Right Boundary of P and the Left Boundary of R

The $\mathrm{RB}(P)$ boundary is the region delimited by the trajectory $\mathrm{RT}(P)$ and the sequence $\mathrm{RS}(P)$, that are either finite or periodic. Thus the region itself is finite or periodic and checking that there is no conflict inside the $\mathrm{RB}(P)$ boundary needs only to be performed on its γ_P first levels. According to Proposition 6, the worst case is when $\mathrm{RB}(P)$ is (μ', π)-periodic, with $\mu' \leq \mu + 2\pi + n - 3$ We have $\gamma_P = \mu' + \pi$ and thus $\gamma_P \leq \mu + 3\pi + n - 3 \leq 4(n-2) - 1$. The width of a boundary being less than n, the worst case time complexity of the $\mathrm{RB}(P)$ test is in $O(n^2)$ time. Similarly, the worst case time complexity of the $\mathrm{LB}(R)$ test is in $O(n^2)$ time.

6.2 No Conflict on the Border of the Heart

The heart of the point S is delimited by the sequences $\mathrm{RS}(P)$ and $\mathrm{LS}(R)$. Since these sequences are either finite or periodic, the heart is itself finite or periodic. Hence checking the heart condition from the point S needs only to be performed on the γ_S first levels of the heart. According to Proposition 5 the $\mathrm{RS}(P)$ sequence is (μ', π)-periodic, with $\mu' \leq 3(n-2)-1$ and the $\mathrm{LS}(R)$ sequence is (λ', τ)-periodic), with $\lambda' \leq 3(n-2)-1$. Then the sequence $((P_k, R_k))_{0 \leq k}$ is $(\nu = max\{\mu', \lambda'\}, \pi' = lcm\{\pi, \tau\})$-periodic. We have $\gamma_S = \nu + \pi'$. Since $\nu \leq 3(n-2) - 1$ and $\pi' \leq (n-3)(n-2)$, we get $\gamma_S \leq n^2$.

An elementary conflict test on the pair (P'_k, R'_k) needs to check whether the two points P'_k and R'_k are adjacent or not, which can be implemented in $O(1)$ time, and to compute the pair (P'_{k+1}, R'_{k+1}) from (P'_k, R'_k), which can be performed in $O(n)$ time. Indeed, for all $1 \leq k$, P'_{k+1} is determined from P'_k either by considering the rightmost transition from p_k, which is in $O(1)$ time, or, when P'_k is the end of one segment generated by some word $u_p^i \in \mathrm{U}(p)$, by computing the next word u_p^{i+1}. This computation can be optimized by considering properties of shifted points, and is in $O(n)$ time. Hence looking for a conflict at a given level is in $O(n)$ time. Since for a given heart there are $O(n^2)$ levels to be checked, testing the border of a heart is in $O(n^3)$ time. Finally, there are at most $n-2$ unbounded states to be tested in Q and thus the following result can be stated.

Proposition 9. *The hole algorithm for testing whether a regular binary language is geometrical has a $O(n^4)$ worst case time complexity.*

This complexity is based on properties of state sequences. Geometrical considerations can improve the performance of the algorithm. For instance, the right trajectory $\mathrm{RT}(P)$

and the left trajectory $\mathrm{LT}(R)$ can converge, and thus the bottom of the hole of S can be reached before γ_S levels are investigated. However the worst case time complexity still holds, corresponding to parallel boundaries.

7 Conclusion

New tools have been developed for studying the properties of a geometrical binary regular language L, in relation with the minimal automaton of $\mathrm{Pref}(L)$. Thanks to periodicity features, a $O(n^4)$ worst case time algorithm has been designed for checking whether a binary regular language L is geometrical or not, generalizing the $O(n^3)$ algorithm that addresses the case where L is prolongable.

References

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

On the Expressive Power of FO[+][⋆]

Christian Choffrut[1], Andreas Malcher[2], Carlo Mereghetti[3], and Beatrice Palano[3]

[1] LIAFA, UMR 7089, 175 Rue du Chevaleret, Paris 13, France
cc@liafa.jussieu.fr

[2] Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany
malcher@informatik.uni-giessen.de

[3] DSI, Università degli Studi di Milano, via Comelico 39/41, 20135 Milano, Italy
{mereghetti,palano}@dsi.unimi.it

Abstract. The characterization of the class of FO[+]-definable languages by some generating or recognizing device is still an open problem. We prove that, restricted to bounded languages, this class coincides with the class of semilinear languages. We also study some closure properties of FO[+]-definable languages which, as a by-product, allow us to give an alternative proof that the Dyck languages cannot be defined in FO[+].

Keywords: Bounded languages, semilinear sets, first order logic.

1 Introduction

The aim of descriptive complexity is to provide logical characterizations of relevant classes of languages. The first result in this area dates back to Büchi [3] who gave a characterization of regular languages via monadic second order logic. Since then, a well consolidated trend in the literature is providing characterizations of several language classes via different logics. This turns out to be also of practical interest, since a logical description of a language often leads to a precise estimate of the parallel complexity of membership and related problems for that language (see, e.g., [1,14]).

Important first order logics for language description are FO[+1], FO[<], and FO[+]. All these logics are used to express properties of words, and their variables range over word positions. Along with the usual predicates $Q_a(x)$ holding true whenever the letter at position x is a and equality, they are provided with the predicates $x + 1 = y$, $x < y$, and $x + y = z$, respectively.

It is well known from the literature (see, e.g., [14]) that FO[+1] characterizes the class of locally threshold testable languages, while FO[<] characterizes the

⋆ This work was partially supported by CRUI/DAAD under the project "Programma Vigoni: Reducing complexity by introducing structure", by the European Science Foundation (ESF) for the activity entitled "Automata: from Mathematics to Applications", and by the Italian MURST under the project "PRIN: Aspetti matematici e applicazioni emergenti degli automi e dei linguaggi formali: metodi probabilistici e combinatori in ambito di linguaggi formali".

wider class of star-free languages, this latter class being strictly contained in that of regular languages. The class of languages described in FO[+] contains all the star-free languages, but not all the regular languages. Yet, it also contains nonregular languages. Nevertheless, a precise definition of the class of languages characterized by FO[+] is still unknown. Thus, it is natural to investigate the possibility of representing relevant classes of languages by FO[+] formulas.

Our main result is the following characterization of the bounded languages which are definable in FO[+], based on the well known notion of semilinear languages introduced by Ginsburg and Spanier in 1964 [7]:

Theorem. *A bounded language is definable in* FO[+] *if and only if it is semilinear.*

This is particularly interesting since such a logic characterization of bounded semilinear languages complements the known characterizations by formal grammars (e.g., simple matrix grammars [9]) and automata (e.g., certain variants of multi-head finite automata and multi-head pushdown automata [10]).

As a consequence of our characterization, we are able to state some negative results on closure properties of FO[+]-definable languages under certain operations. Yet, we also show some positive closure results. These investigations on closure properties provide further insights into the descriptive power of FO[+], e.g., with respect to the meaningful family of the Dyck languages [4,8]. It is known from [1] that the Dyck languages can be described by the logic FOC[+], i.e., FO[+] augmented with counting quantifiers. Moreover, from [12], one may easily get that the Dyck languages cannot be described in FO[+]. Here, we give a new proof of this latter result relying on the logic over words.

2 Preliminaries

The set of natural numbers is here denoted by $\mathbb{N}$. We assume basic notions on formal language theory [8]. Given an alphabet Σ, we denote by Σ^* the set of words on Σ, including the empty word ε. We denote by $|w|$ the length of a word $w \in \Sigma^*$ and by Σ^i the set of words of length i, with $\Sigma^0 = \{\varepsilon\}$. We let $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For any word $w \in \Sigma^*$ and letter $a \in \Sigma$, we let $|w|_a$ be the number of occurrences of the letter a in w. A *language* on Σ is any subset of Σ^*.

We assume familiarity with traditional circuits as a computational model to study the parallel complexity of problems (see, e.g., [15]). We recall that NC^k (AC^k) is the class of problems solved by families of bounded (unbounded) fan-in AND/OR/NOT-circuits of polynomial size and $O(\log^k n)$ depth. It is known from [13] that regular languages lie in NC^1. In [11], further interesting subclasses of context-free languages are shown to be in NC^1, such as the family of the Dyck languages over an arbitrary number of parentheses, and the family of bounded semilinear languages (see Sections 3 and 4.3 for a definition of these languages).

The connection between circuit complexity issues and first order logic formalisms for language description is extensively presented in [14]. In these formalisms, the words over Σ are represented as first order structures in the signature $\langle \{Q_a\}_{a\in\Sigma}, \{P_i\}_{1\le i\le m}, \texttt{last}\rangle$, so that the structure for a word w of length n has universe $\{1,\ldots,n\}$, Q_a is the unary predicate holding true for $1 \le j \le n$ if and only if the jth letter of w is a, P_i's are numerical predicates of different arities (e.g., $x < y$ or $x = y + z$), and $\texttt{last}$ is the constant n. In fact, all logics considered in the sequel assume the predicates Q_a, the numerical predicate $x = y$, and the constant $\texttt{last}$. Yet, they differ on the set $\mathcal{Z}$ of the assumed additional numerical predicates, e.g.: $+1$ for the immediate successor, $<$ for the usual ordering on the nonnegative integers, and $+$ for the ternary predicate $x = y+z$. In this way, a logic has a general designation as FO$[\mathcal{Z}]$, where FO stands for first order quantification. The formulas from FO$[\mathcal{Z}]$ are defined in the usual way, i.e.: every atomic predicate is a formula; if φ_1 and φ_2 are formulas, then $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$ and $\neg\varphi_1$ are formulas; if $\varphi(x_1,\ldots,x_n)$ is a formula whose free variables are $x_1,\ldots,x_n$, then $\exists x_i\, \varphi(x_1,\ldots,x_n)$ and $\forall x_i\, \varphi(x_1,\ldots,x_n)$, with $1 \le i \le n$, are formulas.

Formulas are meant for specifying languages. Indeed, if φ is a sentence (a formula without free variables), we let L_φ be the set of all words satisfying φ, formally $L_\varphi = \{w \in \Sigma^* \mid w \models \varphi\}$. In this case, we say that L_φ is the language *defined* (or described, or expressed) by φ. We denote by $\mathcal{L}(\mathrm{FO}[\mathcal{Z}])$ the class of languages definable in FO$[\mathcal{Z}]$, i.e., $\mathcal{L}(\mathrm{FO}[\mathcal{Z}]) = \{L \subseteq \Sigma^* \mid L = L_\varphi$, for some sentence $\varphi \in \mathrm{FO}[\mathcal{Z}]\}$. Several classes of languages have been logically characterized. For instance, FO$[+1]$ (resp., FO$[<]$) is the first order logic with numerical predicate $+1$ (resp., $<$). It is well known that $\mathcal{L}(\mathrm{FO}[+1])$ is the class of locally threshold testable languages, while $\mathcal{L}(\mathrm{FO}[<])$ is the class of star-free languages. No formal language characterization for $\mathcal{L}(\mathrm{FO}[+])$ is currently known. The following proper hierarchy (see, e.g., [14]) points out the descriptive power of different logics and the relation with circuit complexity:

$$\mathcal{L}(\mathrm{FO}[+1]) \subset \mathcal{L}(\mathrm{FO}[<]) \subset \mathcal{L}(\mathrm{FO}[+]) \subset \mathcal{L}(\mathrm{FO}[+,*]) = \mathrm{AC}^0 \subset \mathrm{NC}^1.$$

3 Bounded Languages

In this section, we exhibit a relevant class of languages contained in $\mathcal{L}(\mathrm{FO}[+])$. More precisely, we show that, *restricted to bounded languages,* FO$[+]$ *characterizes the semilinear languages.*

We recall that a set $X \subseteq \mathbb{N}^m$ is *linear* whenever, for some integer $r \in \mathbb{N}$, there exist vectors $v_0,\ldots,v_r \in \mathbb{N}^m$ such that $X = v_0 + \sum_{t=1}^{r} \mathbb{N}v_t$. A *semilinear set* is a finite union of linear sets.

Let $a_1,\ldots,a_m$ be a sequence of letters from the alphabet Σ with possible repetitions. We consider the *natural embedding of* $\mathbb{N}^m$ *into* Σ^* (we shall simply say "the natural embedding") *relative to this sequence* $a_1,\ldots,a_m$ as the mapping $\chi : \mathbb{N}^m \to \Sigma^*$ defined by $\chi(n_1,\ldots,n_m) = a_1^{n_1}\cdots a_m^{n_m}$. A language $L \subseteq \Sigma^*$ is *letter bounded* whenever $L = \chi(X)$ holds for some $X \subseteq \mathbb{N}^m$. Moreover, L is letter bounded *(linear) semilinear* whenever X is a (linear) semilinear set. Rigorously speaking, we should specify the sequence $a_1,\ldots,a_m$ with the mapping χ. However, the context should clearly determine which sequence is meant. For the sake of conciseness, throughout the rest of the paper *we will always say "bounded"* *instead of "letter bounded"*. Observe that, without loss of generality, we may assume $a_i \neq a_{i+1}$ for every $1 \leq i < m$. Indeed, it clearly suffices to consider the case where X is linear, i.e., of the form $v_0 + \sum_{t=1}^{r} \mathbb{N}v_t$. Suppose there exists $1 \leq i < m$ such that $a_i = a_{i+1}$, and denote with $v_{t,j}$ the jth component of the vector $v_t \in \mathbb{N}^m$. With each v_t, for $1 \leq t \leq r$, we associate the vector $v'_t \in \mathbb{N}^{m-1}$ defined by

$$v'_{t,j} = \begin{cases} v_{t,j} & \text{if } j < i \\ v_{t,j} + v_{t,j+1} & \text{if } j = i \\ v_{t,j+1} & \text{otherwise.} \end{cases}$$

By letting χ' be the natural embedding of $\mathbb{N}^{m-1}$ into $a_1^* \cdots a_i^* a_{i+2}^* \cdots a_m^*$, we get $L = \chi'(X')$ with the linear set $X' = v'_0 + \sum_{t=1}^{r} \mathbb{N}v'_t$.

The closure properties of the bounded semilinear languages are obtained as technical adaptations of the closure properties of the semilinear sets in $\mathbb{N}^m$, as we shall discuss in Proposition 1. A morphism of a free monoid into another is *nonincreasing* if the image of a letter is a letter or the empty word. A *length preserving substitution* is defined by a mapping $h : \Sigma \to 2^{\Sigma} \setminus \{\emptyset\}$, and assigns to the word $a_1 \cdots a_n$ the set of words $h(a_1) \cdot \ldots \cdot h(a_n)$. It extends to subsets of words in the usual way.

Proposition 1. *If $L, L' \subseteq a_1^* \cdots a_m^*$ are bounded semilinear languages and if f is a nonincreasing morphism, then $L \cup L'$, $L \setminus L'$, LL', and $f(L)$ are bounded semilinear. Furthermore, if h is a length preserving substitution, then $L \cap h(L')$ is bounded semilinear.*

Proof. The first three properties are consequences of the results in [6]. Concerning nonincreasing morphisms, observe that they are a composition of morphisms of two types: those renaming a letter and leaving all other letters invariant and those sending a letter to the empty word and leaving all other letters invariant. In the former case, the result directly follows from the above observation concerning the non-repetition of a letter. In the latter case, assume the morphism f satisfies $f(a) = \varepsilon$ and $f(c) = c$ for all $c \in \Sigma \setminus \{a\}$. Let I the subset of indices $i \in \{1, \ldots, m\}$ such that $a_i = a$, and denote by π the morphism of $\mathbb{N}^m$ into $\mathbb{N}^{m-|I|}$ assigning to any m-tuple the $(m - |I|)$-tuple obtained by ignoring the components whose positions are in I. If $L = \chi(X)$ then $f(L) = \chi(\pi(X))$, and we apply the closure property of the semilinear sets under morphism. The last statement follows from the fact that the intersection of a bounded semilinear language with the image of a bounded semilinear language under a substitution by regular sets is bounded semilinear [9, Thm 5.5]. □

We are now going to show that every bounded language is in $\mathcal{L}(\mathrm{FO}[+])$ if and only if it is semilinear. We start with the "if" part.

Theorem 1. *The class of bounded semilinear languages is in $\mathcal{L}(\mathrm{FO}[+])$.*

Proof. It clearly suffices to prove the result for bounded linear languages, i.e., languages of the form $L = \chi(X)$ where $X \subseteq \mathbb{N}^m$ is linear and χ is the natural embedding of $\mathbb{N}^m$ into $a_1^* \cdots a_m^*$, for some sequence $a_1, \ldots, a_m$ of letters from Σ.

As a further simplification, we may assume that no element in X has components equal to 0. Indeed, the sets of the form $M_1 \times \cdots \times M_m$, where $M_i = \{0\}$ or $M_i = \mathbb{N} \setminus \{0\}$, are linear and thus so are all intersections $X \cap M_1 \times \cdots \times M_m$. Let $\emptyset \subset I \subseteq \{1, \ldots, m\}$ be the set of indices i such that $M_i = \mathbb{N} \setminus \{0\}$, let π_I be the projection of $\mathbb{N}^m$ onto $\prod_{i \in I} M_i$, and let χ' be the unique mapping satisfying $\chi(v) = \chi'(\pi_I(v))$, for all $v \in \mathbb{N}^m$. Then, we get

$$\chi(X \cap M_1 \times \cdots \times M_m) = \chi'\left(\pi_I(X) \cap \prod_{i \in I} M_i\right).$$

Whenever $I = \emptyset$, we have $\chi(X \cap M_1 \times \cdots \times M_m) = \{\varepsilon\}$.

We are ready to give an FO[+] formula defining the bounded linear language $L = \chi(X) \subseteq a_1^+ \cdots a_m^+$, with $X = v_0 + \sum_{t=1}^r \mathbb{N} v_t$. By definition, a word $w \in \Sigma^*$ belongs to L if and only if:

(i) w is in $a_1^+ \cdots a_m^+$, and
(ii) for some $\alpha_1, \ldots, \alpha_r \in \mathbb{N}$, it holds $|w|_{a_j} = v_{0,j} + \sum_{t=1}^r \alpha_t v_{t,j}$ for $1 \leq j \leq m$, where $v_{t,j}$ denotes the jth component of the vector v_t.

Let $y_1, \ldots, y_m$ be the variables interpreted as the number of occurrences of the letters $a_1, \ldots, a_m$ in w. Then, our FO[+] formula defining L is of the form

$$\exists y_1 \cdots \exists y_m \; \psi_1(y_1, \ldots, y_m) \wedge \psi_2(y_1, \ldots, y_m),$$

where ψ_1 expresses condition (i) and ψ_2 expresses condition (ii). Using natural abbreviations in order to keep the formula readable, we have

$$\psi_1(y_1, \ldots, y_m) \equiv (y_1 + \cdots + y_m = \texttt{last}) \wedge (y_1 > 0) \wedge \cdots \wedge (y_m > 0) \wedge \\ \forall z \left((z \leq y_1 \Rightarrow Q_{a_1}(z)) \wedge \left(\bigwedge_{i=2}^m \left(\sum_{h=1}^{i-1} y_h < z \leq \sum_{h=1}^{i} y_h \Rightarrow Q_{a_i}(z) \right) \right) \right).$$

For condition (ii), by denoting with z_j the variables interpreted as the coefficients α_j, we have

$$\psi_2(y_1, \ldots, y_m) \equiv \exists z_1 \cdots \exists z_r \left(\bigwedge_{1 \leq j \leq m} \left(y_j = v_{0,j} + \sum_{t=1}^{r} \sum_{s=1}^{v_{t,j}} z_t \right) \right).$$

□

The converse of Theorem 1 goes by structural induction on FO[+] formulas. As a consequence, we must consider not only sentences (i.e., formulas with only bound variables) but more generally formulas with free variables, and therefore we must define what it means for such formulas to be satisfied by some model. We utilize the usual trick which consists of augmenting the letters of the alphabet Σ with a new component specifying subsets of free variables: by so doing, we encode the value of the free variables in the model. More precisely, a formula ϕ over a set $\mathcal{V}$ of free variables is interpreted on $\mathcal{V}$-*structures*, i.e., words of the form $u = (\sigma_1, \mathcal{V}_1) \cdots (\sigma_n, \mathcal{V}_n)$ over the alphabet $\Sigma \times 2^{\mathcal{V}}$, with: (i) $\mathcal{V}_i \subseteq \mathcal{V}$, (ii) $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$

for $i \neq j$, (iii) $\bigcup_{i=1}^{n} \mathcal{V}_i = \mathcal{V}$. We let $\mathcal{S}_{|\mathcal{V}|} \subseteq (\Sigma \times 2^{\mathcal{V}})^*$ denote the set of all $\mathcal{V}$-structures. Let us now explain what it means for a $\mathcal{V}$-structure to satisfy a formula with free variables (the figure below should facilitate the intuition). To fix ideas, let $\phi(x_1, \ldots, x_k)$ be a formula and $\mathcal{V} = \{x_1, \ldots, x_k\}$ the set of its free variables. We say that $u = (\sigma_1, \mathcal{V}_1) \cdots (\sigma_n, \mathcal{V}_n) \in \mathcal{S}_{|\mathcal{V}|}$ satisfies $\phi(x_1, \ldots, x_k)$, and we write $u \models \phi(x_1, \ldots, x_k)$, if $\phi(p_1, \ldots, p_k)$ holds true in the model $\sigma_1 \cdots \sigma_n$ where, for $1 \leq i \leq k$, the integer $1 \leq p_i \leq m$ is the unique position of the $\mathcal{V}$-structure u such that $x_i \in \mathcal{V}_{p_i}$. For instance, the following $\mathcal{V}$-structure

a	b	a	a	b	b
$\emptyset$	$\{x_2\}$	$\{x_3, x_4\}$	$\emptyset$	$\{x_1\}$	$\emptyset$

satisfies the formula

$$Q_b(x_1) \wedge Q_b(x_2) \wedge \neg Q_b(x_3) \wedge (x_2 < x_3) \wedge (x_3 < x_1) \wedge (x_2 < x_4) \wedge (x_4 < x_1).$$

The language defined by ϕ is the set $L_{\phi,\mathcal{V}} = \{u \in \mathcal{S}_{|\mathcal{V}|} \mid u \models \phi\}$. If ϕ is a sentence, i.e., a formula without free variables, then $L_\phi = L_{\phi,\emptyset} = \{w \in \Sigma^* \mid w \models \phi\}$.

The following lemma, which is useful in the proof of the main result, shows the boundedness and semilinearity of a language of $\mathcal{V}$-structures. Given a $\mathcal{V}$-structure $u = (\sigma_1, \mathcal{V}_1) \cdots (\sigma_n, \mathcal{V}_n)$, we let $\pi(u) = \sigma_1 \cdots \sigma_n$. Then:

Lemma 1. *Let $a_1, \ldots, a_m$ be a sequence of letters from an alphabet Σ and let $\mathcal{V}$ be a set of free variables with $|\mathcal{V}| = k$. The language*

$$B_k = \{u \in \mathcal{S}_k \mid \pi(u) \in a_1^* \cdots a_m^*\}$$

is bounded semilinear.

Proof. The set $a_1^* \cdots a_m^*$ is the (finite) union of subsets of the form $a_{i_1}^+ a_{i_2}^+ \cdots a_{i_s}^+$ where $0 \leq s \leq m$ and $i_1, i_2, \cdots, i_s$ is a subsequence of $1, 2, \cdots, m$. So, we are reduced to prove that the set $B'_k = \{u \in \mathcal{S}_k \mid \pi(u) \in a_1^+ \cdots a_m^+\}$ is bounded semilinear.

We claim that B'_k is a finite union of bounded linear languages. A subset in this union is specified by the choice of a sequence of nonempty subsets $\mathcal{V}_1, \ldots, \mathcal{V}_\ell$ defining a decomposition of $\mathcal{V}$ and a choice of letters from $a_1, a_2, \cdots, a_m$ which are associated with the $\mathcal{V}_i$'s. For instance, let $k = 3$ and $m = 3$, i.e., we are considering the words in $a_1^+ a_2^+ a_3^+$ and the formula has 3 free variables x_1, x_3, x_3. Consider the decomposition $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$ with $\mathcal{V}_1 = \{x_1, x_3\}$ and $\mathcal{V}_2 = \{x_2\}$. Associate $\mathcal{V}_1 = \{x_1, x_3\}$ with a_1 and $\mathcal{V}_2$ with a_3. Then, the associated subset is

$$(a_1, \emptyset)^*(a_1, \mathcal{V}_1)(a_1, \emptyset)^*(a_2, \emptyset)^*(a_3, \emptyset)^*(a_3, \mathcal{V}_2)(a_3, \emptyset)^*,$$

which indicates that the positions of the interpretations of the variables x_1 and x_3 are inside the factor of the word labeled by a_1, while the interpretation of the variable x_2 is inside the factor of the word labeled by a_3.

Formally, consider a sequence of the form $(i_1, \mathcal{V}_1), \ldots, (i_\ell, \mathcal{V}_\ell)$ satisfying the following conditions:

- $\mathcal{V}_\alpha \neq \emptyset$, for $1 \leq \alpha \leq \ell$,
- $1 \leq i_1 < i_2 < \ldots < i_\ell \leq m$,
- $\bigcup_{1\leq\alpha\leq\ell} \mathcal{V}_\alpha = \mathcal{V}$ and $\mathcal{V}_\alpha \cap \mathcal{V}_\beta = \emptyset$, for $1 \leq \alpha < \beta \leq \ell$.

Set $i_0 = 1$ and $i_{\ell+1} = m$, and define

$$L_\alpha = (a_{i_{\alpha-1}}, \emptyset)^* \cdots (a_{i_\alpha}, \emptyset)^* (a_{i_\alpha}, \mathcal{V}_\alpha)(a_{i_\alpha}, \emptyset)^* \cdots (a_{i_{\alpha+1}}, \emptyset)^*.$$

Then, each of these L_α is bounded linear and thus the product $L_1 \cdots L_\ell$ is bounded linear as well. Since B'_k is a finite union of such languages, it is bounded semilinear. □

In what follows, for the sake of conciseness, given an alphabet Σ we let $Q_\Sigma(x) \equiv \bigvee_{\sigma\in\Sigma} Q_\sigma(x)$. We are ready to obtain the converse of Theorem 1 as a corollary of

Theorem 2. *For every sequence $a_1, \ldots, a_m$ of letters in Σ and every language $L \subseteq \Sigma^*$ in $\mathcal{L}(\mathrm{FO}[+])$, the language $L \cap a_1^* \cdots a_m^*$ is semilinear.*

Proof. Let ϕ be an FO[+] formula with $\mathcal{V} = \{x_1, \ldots, x_k\}$ free variables. To prove the result, it is enough to show that $L_{\phi,\mathcal{V}} \cap B_k$ is semilinear, where $B_k = \{u \in \mathcal{S}_k \mid \pi(u) \in a_1^* \cdots a_m^*\}$ is the bounded semilinear language addressed in Proposition 1. We shall use the structural induction on ϕ, starting from atomic predicates and then considering more complex formulas

- $\phi \equiv Q_a(x)$: If a does not occur in the sequence $a_1, \ldots, a_m$, then $L_{\phi,\mathcal{V}} \cap B_1 = \emptyset$, so we assume $a = a_i$ for some $1 \leq i \leq m$. We have $L_{\phi,\{x\}} \cap B_1 = (a_1, \emptyset)^* \cdots (a_i, \emptyset)^* (a_i, \{x_1\})(a_i, \emptyset)^* \cdots (a_m, \emptyset)^*$, which is clearly bounded linear.
- $\phi \equiv (x_1 + x_2 = x_3)$: We have to show that $L_{\phi,\{x_1,x_2,x_3\}} \cap B_3$ is semilinear. The formula ϕ is equivalent to $(\phi \wedge (x_1 < x_2)) \vee (\phi \wedge (x_1 = x_2)) \vee (\phi \wedge (x_1 > x_2))$. We prove that the language $L_{\phi_<,\{x_1,x_2,x_3\}} \cap B_3$ with $\phi_< \equiv \phi \wedge (x_1 < x_2)$ is bounded linear, the two other cases being treated similarly. For $W \subseteq \{x_1, x_2, x_3\}$, let $A_W = \{(\sigma, W) \mid \sigma \in \{a_1, a_2, \ldots, a_m\}\}$. Then

$$L_{\phi_<,\{x_1,x_2,x_3\}} = \{{A_\emptyset}^\alpha A_{\{x_1\}} {A_\emptyset}^\beta A_{\{x_2\}} {A_\emptyset}^\alpha A_{\{x_3\}} {A_\emptyset}^\gamma \mid \alpha, \beta, \gamma \in \mathbb{N}\}.$$

  This language is the image under the length preserving substitution defined by $h(a) = A_\emptyset$, $h(b) = A_{\{x_1\}}$, $h(c) = A_{\{x_2\}}$ and $h(d) = A_{\{x_3\}}$ of the bounded linear language $L' = \{a^\alpha b a^\beta c a^\alpha d^\gamma \mid \alpha, \beta, \gamma \in \mathbb{N}\}$. So:

$$L_{\phi_<,\{x_1,x_2,x_3\}} \cap B_3 = h(L') \cap B_3,$$

  and the result follows from Lemma 1 and Proposition 1.
- $\phi \equiv \neg\psi$: We have

$$\begin{aligned} L_{\phi,\mathcal{V}} \cap B_k = L_{\neg\psi,\mathcal{V}} \cap B_k &= (\mathcal{S}_k \setminus L_{\psi,\mathcal{V}}) \cap B_k = (\mathcal{S}_k \cap L^c_{\psi,\mathcal{V}}) \cap B_k \\ &= L^c_{\psi,\mathcal{V}} \cap B_k = B_k \cap (L^c_{\psi,\mathcal{V}} \cup B^c_k) = B_k \setminus (L_{\psi,\mathcal{V}} \cap B_k). \end{aligned}$$

  By inductive hypothesis, we have that $L_{\psi,\mathcal{V}} \cap B_k$ is bounded semilinear. The result follows from Lemma 1 and Proposition 1.

– $\phi \equiv \psi_1 \wedge \psi_2$: We first transform ψ_1 and ψ_2 into equivalent formulas, each over the same set of free variables, say $\mathcal{V}$. To this purpose, let $\mathcal{W}_1$ and $\mathcal{W}_2$ be the set of free variables of ψ_1 and ψ_2, respectively, so that $\mathcal{V} = \mathcal{W}_1 \cup \mathcal{W}_2$. Define $\hat{\psi}_1 \equiv \psi_1 \wedge (\bigwedge_{x \in \mathcal{V} \setminus \mathcal{W}_1} Q_\Sigma(x))$ and $\hat{\psi}_2 \equiv \psi_2 \wedge (\bigwedge_{x \in \mathcal{V} \setminus \mathcal{W}_2} Q_\Sigma(x))$. Clearly, ϕ is equivalent to $\hat{\psi}_1 \wedge \hat{\psi}_2$. We have

$$\begin{aligned} L_{\phi,\mathcal{V}} \cap B_k &= L_{\hat{\psi}_1 \wedge \hat{\psi}_2,\mathcal{V}} \cap B_k = L_{\hat{\psi}_1,\mathcal{V}} \cap L_{\hat{\psi}_2,\mathcal{V}} \cap B_k \\ &= (L_{\hat{\psi}_1,\mathcal{V}} \cap B_k) \cap (L_{\hat{\psi}_2,\mathcal{V}} \cap B_k). \end{aligned}$$

By inductive hypothesis, we know that $L_{\psi_1,\mathcal{W}_1} \cap B_{|\mathcal{W}_1|}$ and $L_{\psi_2,\mathcal{W}_2} \cap B_{|\mathcal{W}_2|}$ are bounded semilinear. Let h be the length preserving substitution which assigns to every letter $(a, W) \in \mathcal{S}_{|\mathcal{W}_1|}$ the set of letters $(a, V) \in \mathcal{S}_k$ where $V = W \cup A$, for $A \subseteq \mathcal{W}_2 \setminus \mathcal{W}_1$. Then, we have

$$L_{\hat{\psi}_1,\mathcal{V}} \cap B_k = h(L_{\psi_1,\mathcal{W}_1} \cap B_{|\mathcal{W}_1|}) \cap B_k.$$

The result follows from Lemma 1 and Proposition 1.

– $\phi \equiv \exists x_{k+1}\ \psi(x_1, \ldots, x_k, x_{k+1})$: Let us define the nonincreasing morphism $\Pi : \mathcal{S}_{k+1} \to \mathcal{S}_k$ as $\Pi(a, V) = (a, V \setminus \{x_{k+1}\})$ For instance, if we let $u = (a, \emptyset)(a, \{x_3\})(b, \{x_1\})(b, \{x_2\}) \in \mathcal{S}_3$, we have $\Pi(u) = (a, \emptyset)(a, \emptyset)(b, \{x_1\}) (b, \{x_2\}) \in \mathcal{S}_2$. We extend Π to subsets of $\mathcal{S}_{k+1}$ in the usual way. Notice that $B_k = \Pi(B_{k+1})$. We have

$$\begin{aligned} L_{\phi,\mathcal{V}} \cap B_k &= L_{\exists x_{k+1}\,\psi(x_1,\ldots,x_{k+1}),\mathcal{V}} \cap B_k = \\ &= \Pi(L_{\psi(x_1,\ldots,x_{k+1}),\mathcal{V}\cup\{x_{k+1}\}}) \cap \Pi(B_{k+1}) \\ &= \Pi(L_{\psi(x_1,\ldots,x_{k+1}),\mathcal{V}\cup\{x_{k+1}\}} \cap B_{k+1}). \end{aligned}$$

The last equality follows from the saturation of B_{k+1} relative to Π, i.e., $\Pi(u) = \Pi(v)$ and $u \in B_{k+1}$ implies $v \in B_{k+1}$. By inductive hypothesis, $L_{\psi(x_1,\ldots,x_{k+1}),\mathcal{V}\cup\{x_{k+1}\}} \cap B_{k+1}$ is bounded semilinear, and the result follows from Proposition 1. □

In conclusion, from Theorems 1 and 2, we get

Theorem 3. *Let L be a bounded language. Then, L is semilinear if and only if L belongs to $\mathcal{L}$(FO[+]).*

4 Closure Properties of $\mathcal{L}$(FO[+])

In this section, we investigate the closure properties of the class $\mathcal{L}$(FO[+]) under various operations. The results of the previous section make it possible to show some negative closure results. On the other hand, also some positive closure results are obtained. As an application, we show that the Dyck languages are not definable in FO[+].

4.1 Negative Closure Properties

A morphism $h : \Sigma^* \to \Delta^*$ of a free monoid into another is *length preserving* whenever $|h(w)| = |w|$, for every $w \in \Sigma^*$.

Proposition 2. $\mathcal{L}(\mathrm{FO}[+])$ *is not closed under length preserving morphism.*

Proof. Let the length preserving homomorphism $h : \{a,b\}^+ \to \{a\}^+$ be defined as $h(a) = h(b) = a$. We prove that the language $L = \{abab^2ab^3a \cdots ab^ia \cdots ab^ka \mid k > 0\}$ belongs to $\mathcal{L}(\mathrm{FO}[+])$, but that $h(L)$ does not. Indeed, a word $w \in \{a,b\}^*$ belongs to L if and only if the following two conditions hold:

– $w = aba$ or $w \in aba\Sigma^*a$,
– if $w \neq aba$ and $w = uab^iav$, with $u, v \in \{a,b\}^*$ and $i > 0$, then there exists a suffix v' of v such that $v = b^{i+1}av'$.

These two conditions can be stated by the following FO[+] formula:

$$Q_a(1) \wedge Q_b(2) \wedge Q_a(3) \wedge Q_a(\mathtt{last}) \wedge$$
$$\forall i\,[(\exists j\,\exists k\; Q_a(i) \wedge Q_a(i+j) \wedge Q_a(i+j+k)) \Rightarrow (\exists j\;\phi(i,j))],$$

with

$$\begin{aligned}\phi(i,j) \equiv\; & Q_a(i) \wedge Q_a(i+j+1) \wedge Q_a(i+2j+3)\\ & \wedge \forall k\,(i < k < i+j+1 \Rightarrow Q_b(k))\\ & \wedge \forall k\,(i+j+1 < k < i+2j+3 \Rightarrow Q_b(k)).\end{aligned}$$

We have $h(L) = \{a^{\frac{(p+1)(p+2)}{2}} \mid p \in \mathbb{N}\backslash\{0\}\}$, but $h(L)$ is not semilinear. Hence, $h(L) \notin \mathcal{L}(\mathrm{FO}[+])$ due to Theorem 2 in Section 3. □

The *commutative image* of a language L is the language

$$\mathrm{COMM}(L) = \{x_1 \cdots x_n \in \Sigma^* \mid x_{i_1} \cdots x_{i_n} \in L \text{ and } \{i_1, \ldots, i_n\} = \{1, \ldots, n\}\}.$$

Proposition 3. $\mathcal{L}(\mathrm{FO}[+])$ *is not closed under commutative image.*

Proof. Consider the language $L = \{abab^2ab^3a \cdots ab^ia \cdots ab^ka \mid k > 0\}$. By the proof of Proposition 2, we have that $L \in \mathrm{FO}[+]$. Notice that the language $L' = \mathrm{COMM}(L) \cap a^*b^* = \{a^nb^{\frac{(n-1)n}{2}} \mid n > 1\}$. If $\mathcal{L}(\mathrm{FO}[+])$ were closed under commutative image, by Theorem 2 in Section 3, L' should be semilinear, a contradiction. □

To have a broader outlook, let us point out other operations under which $\mathcal{L}(\mathrm{FO}[+])$ is not closed. To this aim, we need the following elementary result:

Proposition 4. *The language* $L = \{w \in \{a,b\}^+ \mid |w|_a \bmod 2 = 0\}$ *does not belong to* $\mathcal{L}(\mathrm{FO}[+])$*.*

Proof. By contradiction, suppose that L is in $\mathcal{L}(\mathrm{FO}[+])$, thus in $\mathcal{L}(\mathrm{FO}[<])$ by [2, Cor. 4.2], which is the Crane-Beach conjecture for FO[+]. Let ϕ be an FO[<] formula defining L. Then, $\phi' = \phi \wedge \forall x\, Q_a(x)$ is in FO[<] and defines the language of all the words having no occurrences of the letter b and an even number of a's, namely $(a^2)^+$. Since $(a^2)^+ \notin \mathcal{L}(\mathrm{FO}[<])$ (see, e.g., [14]), we get the result. □

The *shuffle* operation on words can be defined recursively on the length of the words as follows:

$$w \sqcup\!\sqcup\, \varepsilon = \varepsilon \sqcup\!\sqcup\, w = w,$$
$$au \sqcup\!\sqcup\, bv = b(au \sqcup\!\sqcup\, v) \cup a(u \sqcup\!\sqcup\, bv), \text{ with } a, b \in \Sigma, u, v \in \Sigma^*.$$

E.g., if $u = aba$ and $v = aa$ then $u \sqcup\!\sqcup\, v = \{a^3ba, a^2ba^2, aba^3\}$. This notation extends to languages $A, B \subseteq \Sigma^*$ by defining $A \sqcup\!\sqcup\, B = \bigcup_{x \in A,\, y \in B} x \sqcup\!\sqcup\, y$.

Proposition 5. *$\mathcal{L}$(FO[+]) is not closed under: (i) shuffle, (ii) inverse morphism, (iii) Kleene star.*

Proof. *(i)* It is easy to see that the languages $(a^2)^+$ and b^+ belong to $\mathcal{L}$(FO[+]). Thus, if $\mathcal{L}$(FO[+]) were closed under shuffle, the language

$$(a^2)^+ \sqcup\!\sqcup\, b^+ = \{w \in \{a,b\}^+ \mid |w|_a \bmod 2 = 0\}$$

would be in $\mathcal{L}$(FO[+]). This contradicts Proposition 4.

(ii) Consider the language $(a^2)^* \in \mathcal{L}$(FO[+]) and the morphism $h : \{a,b\}^* \to \{a\}^*$ defined as $h(a) = a$ and $h(b) = \varepsilon$. It is easy to see that $h^{-1}((a^2)^*) = \{w \in \{a,b\}^* \mid |w|_a \bmod 2 = 0\}$ is not in $\mathcal{L}$(FO[+]) by Proposition 4.

(iii) Obviously, $\mathcal{L}$(FO[+]) is closed under union and contains all finite languages. If $\mathcal{L}$(FO[+]) were closed under Kleene star, by Proposition 6*(i)* (see below), the whole class of regular languages would be contained in $\mathcal{L}$(FO[+]) due to Kleene's Theorem. However, there exists a regular language that does not belong to $\mathcal{L}$(FO[+]) by Proposition 4, a contradiction. □

4.2 Positive Closure Properties

Given a word $x = x_1 \cdots x_n$, with $x_i \in \Sigma$, its reversal is the word $x^R = x_n \cdots x_1$ (with $\varepsilon^R = \varepsilon$). The *reversal* of a language L is $L^R = \{x^R \mid x \in L\}$.

Proposition 6. *$\mathcal{L}$(FO[+]) is closed under: (i) concatenation, (ii) reversal.*

Proof. Let L, L_1, L_2 be three languages in $\mathcal{L}$(FO[+]). Then, the statement to be proved writes as: *(i)* $L_1 \cdot L_2 \in \mathcal{L}$(FO[+]) and *(ii)* $L^R \in \mathcal{L}$(FO[+]).

(i) Let ϕ_1 and ϕ_2 be FO[+] formulas defining, respectively, the languages L_1 and L_2. We assume that in the formulas ϕ_1 and ϕ_2 all quantified variables are expressed in the form $\exists x \leq \texttt{last}$ and $\forall x \leq \texttt{last}$. We define $\psi = \exists m\ (\phi_1' \wedge \phi_2')$, where:

– ϕ_1' comes from ϕ_1 by replacing every occurrence of $\texttt{last}$ by m,
– ϕ_2' comes from ϕ_2 by replacing every occurrence of $Q_\sigma(i)$ by $Q_\sigma(m + i)$ and every occurrence of $\texttt{last}$ by $\texttt{last} - m$.

Clearly, ψ is an FO[+] formula for the language $L_1 \cdot L_2$.

(ii) Let ϕ be an FO[+] formula for the language L. An FO[+] formula for L^R is obtained by replacing any occurrences of $Q_\sigma(i)$ by $Q_\sigma(\texttt{last} - i + 1)$ in ϕ. □

The *conjugate* of a language L is CONJ$(L) = \{uv \in \Sigma^* \mid vu \in L\}$.

Proposition 7. *$\mathcal{L}$(FO[+]) is closed under conjugation.*

Proof. Let $L \subseteq \Sigma^*$ be a language defined by the FO[+] formula in prenex normal form $\phi \equiv \mathcal{Q}_1 x_1 \cdots \mathcal{Q}_m x_m\ \psi(x_1, \ldots, x_m)$, where $\mathcal{Q}_1, \ldots, \mathcal{Q}_m$ are quantifiers. We define the new predicate

$$Q_a(k, x) \equiv (x \leq k \wedge Q_a(\texttt{last} - k + x)) \vee (x > k \wedge Q_a(x - k)),$$

for free variables k, x. Then, the following is an FO[+] formula for CONJ(L):

$$\exists k\ \mathcal{Q}_1 x_1 \cdots \mathcal{Q}_m x_m\ \psi'(k, x_1, \ldots, x_m),$$

where ψ' is obtained from ψ by replacing every occurrence of $Q_a(x)$ with $Q_a(k,x)$. So, all the variables in the predicates Q_σ are modified with respect to k. □

4.3 An Application: Dyck Languages

Let A be a finite set of opening parentheses and let $\overline{A}$ be the set of (one-to-one) corresponding closing parentheses. Set $T = A \cup \overline{A}$; a word in T^* is correctly (or well) parenthesized if: (i) any opening parenthesis a is followed by a corresponding closing parenthesis $\overline{a}$, and (ii) if parenthesis a' follows a, then a' is closed before a. The *Dyck language* D_T is the set of correctly parenthesized words in T^*. From [1], we know that the Dyck languages are in $\mathcal{L}$(FOC[+]), while from [12] we get that they do not belong to $\mathcal{L}$(FO[+]). We are now going to provide an alternative proof of this latter fact. The following preliminary notions are useful.

The *majority* function $\mathcal{M}_{\sigma,\overline{\sigma}} : \{\sigma, \overline{\sigma}\}^* \to \{0,1\}$ and the *equality* function $\mathcal{E}_{\sigma,\overline{\sigma}} : \{\sigma, \overline{\sigma}\}^* \to \{0,1\}$ are defined, respectively, as:

$$\mathcal{M}_{\sigma,\overline{\sigma}}(x) = \begin{cases} 1 & \text{if } |x|_\sigma > |x|_{\overline{\sigma}} \\ 0 & \text{otherwise,} \end{cases} \qquad \mathcal{E}_{\sigma,\overline{\sigma}}(x) = \begin{cases} 1 & \text{if } |x|_\sigma = |x|_{\overline{\sigma}} \\ 0 & \text{otherwise.} \end{cases}$$

We also need the notion of AC^0-reduction between problems. Informally, a problem P is AC^0-reducible to a problem P' whenever P can be solved by a family of polynomial size, constant depth, unbounded fan-in AND/OR/NOT-circuits with *oracle gates* for P'. In this case, it is easy to see that $P' \in \mathrm{AC}^0$ implies $P \in \mathrm{AC}^0$ as well.

Theorem 4. *The Dyck language $D_{\{a,\overline{a}\}}$ does not belong to $\mathcal{L}$(FO[+]).*

Proof. We first prove that $\mathcal{E}_{a,\overline{a}}$ is not in AC^0. Indeed, since $\mathcal{M}_{a,\overline{a}}$ is not in AC^0 (see, e.g., [5]), it suffices to prove that $\mathcal{M}_{a,\overline{a}}$ is AC^0-reducible to $\mathcal{E}_{a,\overline{a}}$. Consider $x = x_1 \cdots x_n \in \{a, \overline{a}\}^n$. To compute $\mathcal{M}_{a,\overline{a}}(x)$, we build an AC^0-circuit C_n containing a first layer of oracle gates $O_0, \ldots, O_{\lfloor \frac{n}{2} \rfloor}$ for $\mathcal{E}_{a,\overline{a}}$. As input to O_i, we give the word $w(i) = a^{i+(n \bmod 2)} x_{i+1} \cdots x_n$. If there is an oracle O_i yielding 1, we have that $|x|_a \leq |w(i)|_a = |w(i)|_{\overline{a}} \leq |x|_{\overline{a}}$. On the contrary, if all O_i's yield 0, we get that $|x|_a > |x|_{\overline{a}}$. Thus, we can complete C_n by plugging all the outputs of O_i's into an OR gate whose output, in turn, is sent to a final NOT gate.

Let us now turn to the proof of the theorem. By contradiction, suppose that $D_{\{a,\overline{a}\}} \in \mathcal{L}$(FO[+]). It is not difficult to verify that the following holds

$$\mathrm{CONJ}(D_{\{a,\overline{a}\}}) = \{w \in \{a, \overline{a}\}^* \mid |w|_a = |w|_{\overline{a}}\}.$$

By Proposition 7, $\mathcal{L}$(FO[+]) is closed under conjugation, and consequently we have CONJ($D_{\{a,\overline{a}\}}$) $\in \mathcal{L}$(FO[+]). However, we have $w \in$ CONJ($D_{\{a,\overline{a}\}}$) if and only if $\mathcal{E}_{a,\overline{a}}(w) = 1$. Since $\mathcal{E}_{a,\overline{a}}$ is not in AC^0 and $\mathcal{L}$(FO[+]) $\subset$ AC^0, this completes the proof. □

Theorem 5. *The Dyck language D_T does not belong to $\mathcal{L}(\mathrm{FO}[+])$.*

Proof. Let $T = A \cup \overline{A}$ and $a \in A$ a type of parentheses of D_T. By contradiction, suppose there exists a FO[+] formula ϕ for D_T, and construct the formula $\phi' = \phi \wedge \forall x\,(Q_a(x) \vee Q_{\overline{a}}(x))$. Clearly, ϕ' is an FO[+] formula for the subset of D_T consisting of the well parenthesized words over the alphabet $\{a, \overline{a}\}$, namely $D_{\{a,\overline{a}\}}$. This contradicts Theorem 4. $\square$

References

1. Barrington, D.M., Corbett, J.: On the relative complexity of some languages in NC. Information Processing Letters 32, 251–256 (1989)
2. Barrington, D.M., Immerman, N., Lautemann, C., Schweikardt, N., Thérien, D.: First-order expressibility of languages with neutral letters or: The Crane Beach conjecture. Journal of Computer and System Sciences 70, 101–127 (2005)
3. Büchi, J.: Weak second order arithmetic and finite automata. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik 6, 66–92 (1960)
4. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 118–161. North Holland, Amsterdam (1963)
5. Furst, M., Saxe, J., Sipser, M.: Parity, circuits, and the polynomial-time hierarchy. Mathematical Systems Theory 17, 13–27 (1984)
6. Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, New York (1966)
7. Ginsburg, S., Spanier, E.: Bounded ALGOL-like languages. Trans. Amer. Math. Soc. 113, 333–368 (1964)
8. Harrison, M.: Introduction to Formal Languages. Addison-Wesley, Reading (1978)
9. Ibarra, O.: Simple matrix grammars. Information and Control 17, 359–394 (1970)
10. Ibarra, O.: A note on semilinear sets and bounded-reversal multihead pushdown automata. Information Processing Letters 3, 25–28 (1974)
11. Ibarra, O., Jiang, T., Ravikumar, B.: Some subclasses of context-free languages in NC^1. Information Processing Letters 29, 111–117 (1988)
12. Robinson, D.: Parallel algorithms for group word problems. Doctoral Dissertation, Mathematics Dept., University of California, San Diego (1993)
13. Ruzzo, W.: On uniform circuit complexity. Journal of Computer and System Sciences 22, 365–383 (1981)
14. Straubing, H.: Finite Automata, Formal Logic, and Circuit Complexity. Birkhäuser, Basel (1994)
15. Wegener, I.: The Complexity of Boolean Functions. Teubner, Stuttgart (1987)

Finding Consistent Categorial Grammars of Bounded Value: A Parameterized Approach

Christophe Costa Florêncio[1] and Henning Fernau[2]

[1] Department of Computer Science, K.U. Leuven, Leuven, Belgium
Chris.CostaFlorencio@cs.kuleuven.be
[2] Universität Trier, FB IV—Abteilung Informatik, D-54286 Trier, Germany
fernau@uni-trier.de

Abstract. Kanazawa ([8]) has studied the learnability of several parameterized families of classes of categorial grammars. These classes were shown to be learnable from text, in the technical sense of identifiability in the limit from positive data. They are defined in terms of bounds on certain parameters of the grammars. Intuitively, these bounds correspond to restrictions on linguistic aspects such as the amount of lexical ambiguity of the grammar.

The time complexity of learning these classes has been studied by Costa Florêncio ([4]). It was shown that for most of these classes, selecting a grammar from the class that is consistent with the data is NP-hard. In this paper existing complexity results are sharpened by demonstrating W[2]-hardness. Additional parameters allowing FPT-results are also studied, and it is shown that if these parameters are fixed, these problems become computable in polynomial time. As far as the authors are aware, this is the first such result for learning problems.

1 Introduction

We consider the complexity of *consistency problems* for some family $(\mathcal{L}_k)$ of language classes of the following form: Given a finite language sample D and some integer k, is there some language $L \supseteq D$ contained in $\mathcal{L}_k$? For many families of language classes, this type of consistency problem is trivial. Consider for example the class $\mathcal{A}_k$ of languages that can be accepted by a finite automaton with at most k states. In this case we can always answer YES to the consistency problem, since an automaton with just one state exists that accepts Σ^*.

However, this trivial type of reply is no longer possible if the universal language (i.e., Σ^* in the case of string languages) is not (automatically) in each of the language classes of interest. Examples are provided by classes of classical categorial grammars, see [8].

The language families $\mathcal{L}_k$ are usually defined via grammar families $\mathcal{G}_k$. As a variant of the mentioned consistency problem, we may be given a finite set of derivation structures and some parameter k and ask if there is a grammar $G \in \mathcal{G}_k$ that produces those structures (and possibly more). Note that this can be also seen as a special case of the first problem formulation, if we consider *languages of structures* (formally, labeled ordered trees).

We study the computational complexity of consistency problems both from a classical (P vs. NP) perspective, as well as from the perspective of parameterized complexity.

Since categorial grammars are mainly studied within computational and mathematical linguistics, our results may be especially relevant for researchers from these fields.

In order to keep the paper as self-contained as possible, we will try to provide the necessary background of all the relevant fields. Readers familiar with this material can of course skip these sections. Throughout the paper, we use standard notation from Formal Language Theory. If D is a finite language, then $\|D\|$ denotes the sum of all lengths of all words from D.

2 Categorial Grammars

The classes studied in [1,2] which are the focus of the present paper are based on a formalism for (ε-free) context-free languages called *classical categorial grammar* (CCG). In this section the relevant concepts of CCG will be defined. We will adopt the notation used in [8].

In CCG, each *symbol* (or *atom*) in some given alphabet Σ is assigned a finite number of *types*. In the remainder, we assume Σ to be fixed. This is technically convenient, and makes no difference in the context of learning, since only the subset of Σ that actually appears in the data is relevant for the learner. Types are constructed from *primitive types* by the operators \ and /. We let Pr denote the (countably infinite) set of primitive types. The set of types Tp is defined as the smallest set satisfying:

1. $\mathrm{Pr} \subseteq Tp$,
2. if $A \in Tp$ and $B \in Tp$, then $A\backslash B \in Tp$.
3. if $A \in Tp$ and $B \in Tp$, then $B/A \in Tp$.

One member t of Pr is called the *distinguished type*, and is considered a constant. In CCG there are only two modes of type combination, *backward application*, $A, A\backslash B \Rightarrow B$, and *forward application*, $B/A, A \Rightarrow B$. In both cases, type A is the *argument*, the complex type is the *functor*. Given an expression of the form A/B ($B\backslash A$), its main operator is '/' ('\'). *Grammars* consist of type assignments to symbols, i.e., $\texttt{symbol} \mapsto T$, where $\texttt{symbol} \in \Sigma$ and $T \in \mathrm{Tp}$.

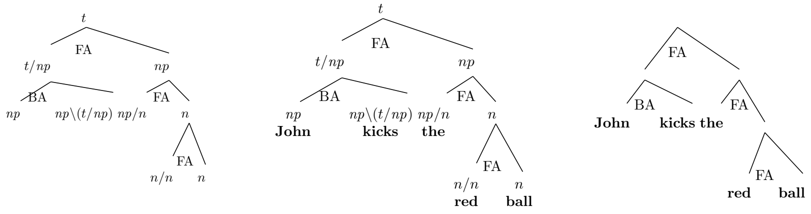
A *derivation* of B from $A_1, \ldots, A_n$ is a binary branching, labeled tree that encodes a proof of $A_1, \ldots, A_n \Rightarrow B$. Through the notion of derivation the association between grammar and language is defined. All structures contained in some given structure language correspond to a derivation of type t based solely on the type assignments contained in a given grammar. This is the *structure language* generated by G, denoted $\mathrm{FL}(G)$. The *string language* generated by G, $\mathrm{L}(G)$, consists of the strings corresponding to all the structures in its structure language, where the string corresponding to some derivation consists just of the leaves of that derivation (also known as the *yields*).

The symbol FL is an abbreviation of *functor-argument language*, the derivation language for a CCG that is obtained by suppressing types associated to inner nodes in the derivation (tree). Hence, structures correspond to terms. More precisely, structures are of the form `symbol`, `fa(s1,s2)` or `ba(s1,s2)`, where `symbol` $\in \Sigma$, `fa` stands for forward application, `ba` for backward application and `s1` and `s2` are also structures.

Example 1. The leftmost structure is a derivation for a proof of

$$np, np\backslash(t/np), np/n, n/n, n \Rightarrow t.$$

Note that all tree nodes carry types as labels, and that to inner nodes, in addition, labels BA and FA are associated, which indicate the operations applied at those points in the derivation.
The middle structure shows a CCG parse, which is a derivation where the leaves are labeled not just with types, but also with the lexical items (such as **John**) that these types are assigned to in the grammar used for the parse. The rightmost structure shows the corresponding functor-argument structure.



All learning functions in [8] are based on the function GF. This function receives a sample of structures D as input and yields a set of assignments (i.e., a grammar) called the *general form* as output, which generates exactly D. It is a homomorphism and runs in linear time. GFassigns t to each root node, assigns distinct variables to the argument nodes, and computes types for the functor nodes: if it is the case that `s1` $\mapsto A$, given `ba(s1,s2)` $\Rightarrow B$, then `s2` $\mapsto A\backslash B$. If `s1` $\mapsto A$, given `fa(s2,s1)` $\Rightarrow B$, then `s2` $\mapsto B/A$. When learning from strings, the structure language is not available to the learner, but given a set of strings there exist only finitely many possible sets of structures for the classes under discussion. These are then used to produce hypotheses.

Categorial types can be treated as terms, so natural definitions of substitution and unification apply. A substitution over a grammar is just a substitution over all of the types contained in its assignments. This notion can be used to unify distinct types assigned to the same word. Consider, for example, the following grammar:

$$G = \begin{array}{l} \texttt{a} \mapsto t/A, B\backslash t, C/(E/E), (C/D)/D \\ \texttt{b} \mapsto A, B, t/C, D, (E/E)/D \end{array}$$

(The reader can verify that L(G) consists of just the words `ab`, `ba` and `babb`.) The types t/A and $C/(E/E)$ can be unified to yield the single type $t/(E/E)$, obtained by applying the most general unifier $\sigma\{C = t, A = (E/E)\}$. The type $B\backslash t$ cannot be unified with any of the other types, since they all have '/' as main operator, while $B\backslash t$ has '\' as main operator. The types t/A and $(C/D)/D$ cannot be unified because their functors are a constant and a complex type, respectively. Finally, $C/(E/E)$ and $(C/D)/D$ cannot be unified because this would fail the *occurs check*: it would require that C is unified with C/D.

We state without proof that $\mathrm{FL}(G) \subseteq \mathrm{FL}(\sigma[G])$ for each substitution σ, see [8] for details.

A CCG G can be hence viewed as a mapping from Σ into finite subsets of types Tp. Accordingly, we can associate a *value function* v that maps $a \in \Sigma$ onto $|G(a)|$, i.e., the number of types that G maps to a. G is *k-valued* [4,8] if $\max_{a\in\Sigma} v(a) \leq k$. The according grammar and language classes are denoted by $\mathcal{G}_{k-\mathrm{valued}}$ and $\mathcal{L}_{k-\mathrm{valued}}$, resp. 1-valued grammars are also known as *rigid grammars*, and denoted as $\mathcal{G}_{\mathrm{rigid}}$. This class is known to be learnable from structures with polynomial update-time, by simply unifying all types assigned to the same symbol in the general form [8]. The other classes originally defined in [1,2] are generalizations thereof.

The class of structure languages that can be generated by grammars from $\mathcal{G}_{k-\mathrm{valued}}$ is written $\mathcal{FL}_{k-\mathrm{valued}}$.

3 Complexity Notions

We assume some familiarity with the basic notions of classical complexity on the side of the reader.

There has been recent interest in the development of parameterized complexity results to allow for a more fine-grained analysis of NP-hard problems. So, a problem (parameterized by k) is in FPT if we can develop an algorithm with running time $O(f(k)p(n))$, where n is the overall input size and k is the (size of the) parameter. f is an arbitrary function (only depending on k but not on n) and p is a polynomial. An algorithm that proves FPT-ness is also called an FPT-*algorithm*, or a *parameterized algorithm*. More details can be found in the monograph [5].

The classical example for a problem in FPT is the VERTEX COVER problem on undirected graphs. So, given a graph G and a parameter k, it is asked if there exists a *vertex cover* C with $|C| \leq k$, where a vertex cover set can be characterized by the fact that, upon removing it together with all incident edges, no edges will remain in the graph.

This approach makes sense in particular if the parameter of interest can be assumed to be small. The hierarchy level k in our formulation of the consistency problem might be such a small parameter: in linguistics, the amount of lexical ambiguity for natural language is assumed to be very small in relation to the number of symbols found in that lexicon. So, we arrive at problems that we call, for instance, $\mathcal{L}_{k-\mathrm{VALUED}}$-CONSISTENCY in order to make the parameter explicit. However, we cannot always hope to find nice parameterized algorithms. More specifically, we can derive as a corollary from [4, Theorem 5.32] (alternatively, [3]):

Corollary 1. *Unless* P = NP*, there is no* FPT *algorithm that decides* $\mathcal{L}_{k-\mathrm{VALUED}}$-CONSISTENCY*.*

Proof. If this were not the case, there would be an algorithm that decides the consistency problem in time $O(f(k)p(n))$. Setting $k = 1$, we would arrive at a polynomial-time algorithm that decides if, given a finite set D of strings, there exists a 1-max-valued (rigid) categorial grammar G such that $D \subseteq \mathrm{L}(G)$. This problem is known to be NP-hard. □

This corollary is surely disappointing from a parameterized point of view, since it seems to rule out to use k as a good choice of a parameter for the consistency problem for k-valued categorial grammars.

The proof of [4, Theorem 5.32] makes use of the fact that there is no bound on the size of the alphabet, so we still might hope for better parameterized results when we restrict our attention to languages over alphabets of size three, for example.

As we will see, this hope will not be fulfilled. However, in order to formulate and to establish the indicated result, we need some more notions from parameterized complexity.

As with classical complexity, we need an appropriate notion of reduction to prove hardness results, and some knowledge about classes of parameterized problems that are believed not to possess FPT-algorithms. Actually, there is a whole hierarchy of parameterized problems that is believed to be strict, the so-called W-hierarchy. Usually, its lowest level, W[0], is called FPT (which we have already defined).

A typical W[1]-complete problem is the following one: Given a graph G and a parameter k, is there an independent vertex set I of size k in G? Recall that C is a minimum vertex cover in a graph $G = (V, E)$ iff the subgraph induced by $V \setminus C$ is an *independent set*, see [6].

A typical W[2]-complete problem is the VERTEX COVER problem for hypergraphs, also known as HITTING SET: given a hypergraph G and a parameter k, we ask for a vertex set C of size at most k such that each edge e of G is hit, i.e., $e \cap C \neq \emptyset$ (note that a hyperedge e is simply the set of vertices it connects).

We still need a satisfying notion of reduction in order to define hardness (and completeness) for parameterized complexity classes. Given two parameterized problems P and P' with parameterizations k and k', resp., a *parameterized (many-one) reduction* translates an instance (I, k) of P in polynomial time into an instance (I', k') such that $k' = f(k)$ for some function f. Obviously, if P' is in $W[i]$, $i = 0, 1, 2$, then so is P. So, if P is W[2]-hard and we can provide such a reduction that translates P into P', then P' is W[2]-hard, as well. As an example, consider the reductions presented in Sections 5.3-5.5 from [4] that show NP-hardness of several variants of consistency problems (as we would call those here). These reductions use VERTEX COVER, and the reductions are actually parameterized reductions in the following sense: they show how an instance (G, k) of VERTEX COVER can be transformed in polynomial time into an instance (F, k) of $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY. This is why we originally hoped for FPT-ness results for this type of problems.

4 Results

In [4], only hardness results were shown. We first complement these results by demonstrating membership in NP. This was originally neglected, since the consistency problem was studied as an aspect of learning problems, specifically of identification in the limit. In this paradigm, the length of the input sequence before convergence is inherently unbounded. Thus, it makes little sense to consider questions such as membership in NP, which would require a polynomial number of steps before convergence.

Lemmas 6.1 and 6.2 from [8] (attributed to Buszkowski and Penn) underline the importance of the concept of the general form GF(D) (as discussed above), a CCG associated to a finite structure language D. Without giving details here, notice that it is further known that any reduced CCG G' consistent with D can be obtained from GF(D) by unification. Moreover, the size of GF(D) (and hence the size of G') is bounded by a polynomial over $\|D\|$. If D is a finite string language, then any finite structure language D' that yields D is of size polynomial in $\|D\|$. Hence, any GF(D') of interest is also of size polynomial in $\|D\|$.

Theorem 1. *$\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY is NP-complete.*

Proof. NP-hardness of $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY is shown in [4] (Theorem 5.16, which holds for the case where $|\Sigma| = 3$). To see membership in NP, let (D, k) be an instance of $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY. The nondeterministic procedure we propose first generates some $G \in \mathcal{G}_{k-\text{valued}}$, by unifying types in GF(D) that are assigned to the same symbol. Then, for each structure $s \in D$, the procedure tests whether $s \in \text{FL}(G)$ (which can be done in polynomial time). If (and only if) all these tests are passed, the algorithm returns YES. □

Theorem 2. *$\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY is NP-complete.*

Proof. NP-hardness of $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY is shown in [4] (Theorem 5.32, which holds for the case where $k = 1$ and $|\Sigma|$ is unbounded). To see membership in NP, let (D, k) be an instance of $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY. The nondeterministic procedure we propose consists of two parts; first, for each string in D, a derivation is chosen. Then, the union of the resulting structures, D' (note that $\|D'\|$ is obviously polynomial in $\|D\|$), is used as a sample for learning from structures, so that some $G \in \mathcal{G}_{k-\text{valued}}$ is generated by unifying types in GF(D') that are assigned to the same symbol. Then, for each string $w \in D$, the procedure tests whether $w \in \text{L}(G)$ (which can be done in polynomial time). If (and only if) all these tests are passed, the algorithm returns YES. □

We can actually sharpen the hardness assertion in the sense of parameterized complexity, by defining a polynomial-time transformation from HITTING SET to a dataset for a language in $\mathcal{FL}_{k-\text{valued}}$. Deciding that the data is consistent with a language in that class, i.e., $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY, then corresponds to deciding the existence of a cover of a specified size c. We will call any grammar that generates a language in the class consistent with the given input a *consistent grammar.*

We now define a construction that is based on these ideas.

Definition 1. *Let $hg(HG, c)$ be the algorithm that maps instances of the vertex cover problem for hypergraphs to samples of structure languages defined in the following way:*

The hypergraph $HG = (V, E)$ consists of a set V of vertices numbered $1, \ldots, v$ and a set E of (hyper)edges numbered $1, \ldots, e$. It is characterized by the functions

$d(i)$, $1 \leq i \leq e$, which gives the degree of edge E_i, and $n(i,j)$, which gives the index of the jth vertex that edge i is incident on. The constant c specifies the maximal size of the cover.

The sample D output by hg is the smallest set that fulfills the following requirements:

For each i, $1 \leq i \leq e$, the structures
$\mathtt{fa}(\ldots\mathtt{fa}(\mathtt{e}_i, \underbrace{\mathtt{fa}(\mathtt{c}_i,\mathtt{x}))\ldots\mathtt{fa}(\mathtt{c}_i,\mathtt{x})),}_{d(i)\ \textit{times}}\ \underbrace{\mathtt{u}_i),\ldots\mathtt{u}_i)}_{d(i)+1\ \textit{times}}$ *and*
$\mathtt{fa}(\ldots\mathtt{fa}(\mathtt{e}_i,\mathtt{f}_{(i,1)}),\ldots\mathtt{f}_{(i,d(i))}),\mathtt{fa}(\mathtt{tt},\mathtt{t})),\mathtt{s}_{(i,1)}),\ldots\mathtt{s}_{(i,d(i)-1)}),$
$\mathtt{fa}(\mathtt{ttt},\mathtt{t}))$ *are in D. Additionally, for each j, $1 \leq j \leq d(i)$,*
$\mathtt{fa}(\ldots\mathtt{fa}(\mathtt{e}_i,G_{(j,1)}),\ldots G_{(j,d(i))}),T_{(j,1)}),\ldots T_{(j,d(i)+1)})$ *where $G_{(j,x)}$ is $\mathtt{fa}(\mathtt{v}_{n(i,x)},\mathtt{x})$ if $x = j$ and $\mathtt{g}_{(i,x)}$ otherwise, and where $T_{(j,x)}$ is $\mathtt{fa}(\mathtt{t}_{j,j},\mathtt{x})$ if $x = j$ or $x = j+1$ and $\mathtt{t}_{(j,x)}$ otherwise.*

For each i, $1 \leq i \leq e$, $\mathtt{fa}(\mathtt{c},\mathtt{fa}(\mathtt{c}_i,\mathtt{x}))$ is in D. If $c = 1$, the padding structure $\mathtt{ba}(\mathtt{x},\mathtt{c})$ is in D.

For each i, $1 \leq i \leq c$, the padding structure $\mathtt{ba}(\mathtt{x},\mathtt{c}_i)$ is in D.

For each i, $1 \leq i \leq v$, the padding structure $\mathtt{ba}(\mathtt{x},\mathtt{v}_i)$ is in D.

For each $\mathtt{t}_{i,j}$, the padding structure $\mathtt{ba}(\mathtt{x},\mathtt{t}_{i,j})$ is in D.

We add $k-2$ padding structures for each $\mathtt{e}_i$, and $k-1$ such structures for each $\mathtt{v}_i$, $\mathtt{c}_i$, and for $\mathtt{tt}$ and $\mathtt{ttt}$.

In order to make clear why the sample is built up in this way, we now discuss the types as they occur in any grammar in $\mathcal{G}_{k-\text{valued}}$ that is consistent with this sample.

Let $\Upsilon_i = C_{i,1}/\ldots C_{i,v}/(U_{s(1,i,0)}/U_{s(1,i,1)})/\ldots(U_{s(d(i),i,0)}/U_{s(d(i),i,1)})$,
$\Gamma_{n(i,j)} = G_{i,1}/\ldots G_{i,v}/\ (T_{t(1,i,0)}/T_{t(1,i,1)})/\ldots(T_{t(d(i),i,0)}/T_{t(d(i),i,1)})$,
$T_{t(k,i,0)} = T_{t(\ell,i,1)}$ if $\ell = n(i,j)$, and the $T_{t(\ell,i,j)}$s are distinct (primitive) types, otherwise.

Every type $G_{i,j}$, $i = j$, is equal to some type $\Delta_n(i,j)$. These are based strictly on alternating forward- and backward slashes, with the main operator always the backward slash. The type Δ_1 is $X\backslash t$. For any two u, v such that $u \neq v$, Δ_u and Δ_v are not unifiable. Note that this allows any two $\Gamma_{n(i,x)}$ and $\Gamma_{n(i,y)}$, $x \neq y$, to be unifiable, since the Δ-subtypes appear in different positions of these Γ terms.

Define $\Sigma_i = F_{i,1}/\ldots F_{i,v}/(S_{g(1,i,0)}/S_{g(1,i,1)})/\ldots(S_{g(v,i,0)}/S_{g(v,i,1)})$ for $1 \leq i \leq e$. From GF(D), the following grammar is derived by unifying all types assigned to $\mathtt{x}$. This simplifies the presentation without affecting the proof of W[2]-hardness in any way. Note that in the interest of clarity we omit type assignments to symbols $\mathtt{f}_{n(x,y)}$, $\mathtt{g}_{n(x,y)}$, $\mathtt{s}_{x,y}$, $\mathtt{t}_{x,y}$ and $\mathtt{u}_x$.

$$\begin{aligned}\mathtt{e}_1 \mapsto\ & t/\Upsilon_1,\\ & t/\Gamma_{n(1,1)},\ldots t/\Gamma_{n(1,d(1))},\\ & t/\Sigma_1,\\ & \textit{Padding}\\ \ldots & \end{aligned}$$

$$G' : \begin{array}{ll}
\mathtt{e}_e & \mapsto t/\Upsilon_e, \\
 & \quad t/\Gamma_{n(e,1)}, \ldots t/\Gamma_{n(e,d(e))}, \\
 & \quad t/\Sigma_e, \\
 & \quad \mathit{Padding} \\
 & \\
\mathtt{v}_1 & \mapsto \Delta_1, \mathit{Padding} \\
\ldots & \\
\mathtt{v}_v & \mapsto \Delta_v, \mathit{Padding} \\
 & \\
\mathtt{c}_1 & \mapsto C_{1,1}/X, \ldots, C_{1,v}/X, C_1/X, \mathit{Padding} \\
\ldots & \\
\mathtt{c}_e & \mapsto C_{e,1}/X, \ldots, C_{e,v}/X, C_e/X, \mathit{Padding} \\
 & \\
\mathtt{c} & \mapsto t/C_1, \ldots, t/C_e, \mathit{Padding} \\
 & \\
\mathtt{x} & \mapsto X \\
 & \\
\mathtt{t} & \mapsto t \\
\mathtt{tt} & \mapsto t/t, \mathit{Padding} \\
\mathtt{ttt} & \mapsto (t/t)/t, \mathit{Padding}
\end{array}$$

Where, for $1 \leq i \leq e$, $j = n(i, d(i))$.

Note that *hg* runs in time polynomial in the size of the hypergraph. There are bounds on parameters of the grammar: given hypergraph $HG = (V, E)$ and stipulated size of the cover c, $k = \max(2, c)$, and $|\Sigma| = 5 + |V| + 2|E| + 2\sum_{i=1}^{|E|} d(i) + 2\sum_{i=1}^{|E|} d(i)^2$.

The construction works just for $k \geq 2$, but this does not affect the result in any way. Note that for $k = 1$, the consistency problem is known to be solvable in polynomial time.

Theorem 3. $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY *is* W[2]*-hard.*

Proof. By modifying the mentioned NP-hardness proofs, we show how to transform an instance (G, k) of HITTING SET to $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY, preserving the parameter. To be more precise, the HITTING SET problem can be reduced in polynomial time to finding a grammar consistent with structures D and in the class $\mathcal{G}_{k-\text{valued}}$. We achieve this using the algorithm *hg* as given in Definition 1.

Let hypergraph $HG = (V, E)$, $G = \text{GF}(hg(HG))$, and c such that a cover of size c exists for HG.

For any symbol $\mathtt{e}_i$, unification of all types $t/\Gamma_{n(i,1)}, \ldots t/\Gamma_{n(i,d(i))}$ to t/Σ_i will lead to a substitution such that $S_{g(1,i,0)} = S_{g(1,i,1)} = \ldots = S_{g(v,i,0)} = S_{g(v,i,1)}$. Since this is not possible, for each symbol $\mathtt{e}_i$, at most one of the types $t/\Gamma_{n(i,1)}, \ldots t/\Gamma_{n(i,d(i))}$ can be unified with t/Υ_i instead. For each i, only one of these t/Γ types can be chosen for this, since it will block unification of t/Υ_i with any of the other t/Γ types: for any given i, the U types in Υ_i all have to be of the same type, and such a unification step will result in a substitution such that some Δ type will be substituted for all these U types.

For every i, the T types in Γ_i overlap: $T_{j,j}$ occurs twice in every Γ_i.

Thus, they cannot all be unified with Σ_i, since the pair of the first and last S type in every Σ-type is not unifiable.

Hence, for each i, *exactly* one of the t/Γ types has to be unified with the t/Υ type, and the rest with the Σ type. This implies a substitution such that for each C_i, a Δ_ℓ is substituted such that $\ell = n(i,j)$, $1 \leq j \leq d(i)$. This corresponds to choosing vertex ℓ in the original hypergraph to cover edge i. Since, for all i, t/C_i is assigned to symbol c, and given the number of padding types assigned to this symbol (0 if $c \geq 2$, 1 if $c = 1$), the number of distinct Δ types that substitute for the C types can be no more than c. This proves that, if $k \geq c$, the answer to $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY is YES.

Let c and HG be such that a minimum cover for HG is of size $c' > c$, and let $G = \text{GF}(hg(HG))$ as before. Following the same line of reasoning as earlier in this proof, it is clear that for each C_i, a Δ_k must be substituted in order to obtain a consistent grammar that is in the class. Given the definition of hg, these Δ types correspond to one of the vertices that edge i is incident on. Given that $c' > c$, there are at least c' distinct such Δ types, and since for all i, t/C_i is assigned to c, at least c' distinct types are assigned to c in a consistent grammar, which thus cannot be in $\mathcal{G}_{k-\text{valued}}$ for $k = max(c, 2)$. Thus the answer to $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY is NO.

This proves that the answer to $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY for the sample $hg(HG, c)$ is the same as for HITTING SET for HG with c as size of the cover. Since the reduction hg runs in polynomial time, this proves W[2]-hardness. □

As for the problem $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY, note that $\mathcal{L}_{k-\text{valued}}$ contains Σ^* for $k \geq 2$, which immediately trivializes the problem. We refer to [3] for a proof of NP-hardness for the case $k = 1$. Notice that these results render the parameterization discussed in this section meaningless from the viewpoint of parameterized complexity.

As an aside, let us mention that in the literature (see Corollary 5.13 from [4], for example) also consistency questions related to the *least-k-valued* grammars and languages were considered. This means we are looking for the smallest k such that there is a grammar $G \in \mathcal{G}_{k-\text{valued}}$ and $D \subseteq \text{FL}(G)$. These problems are also known to be NP-hard, but it is an open question whether they belong to NP.

5 Reparameterizations

As already seen with the example of VERTEX COVER versus INDEPENDENT SET, basically the same problem can be parameterized in different ways, possibly leading to positive (FPT) results or to negative (W[2]-hardness) results. So it might be that other choices of parameterization may lead to FPT-algorithms.

One other natural choice of a parameter is the number of unification steps u needed to transform the general form of D into some k-valued grammar G such that $D \subseteq \text{L}(G)$. This leads to problems like u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY. The input to such a problem would be a triple (D, u, k), where D is the finite input sample. A variant could be UNIFORM u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY, where the input would be (D, k), and the question would be whether there exists a u such that (D, u, k) is a YES-instance of u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY. Hence, the inputs to UNIFORM u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY and to $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY are the same.

Although the following is not stated explicitly in [8], it follows from Proposition 6.32 and the preceding description of algorithm VG_k, and the description of algorithm LVG (Section 6.3) in that book:

Theorem 4. *(D, k) is a YES-instance to* $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY *iff (D, k) is a YES-instance to* UNIFORM u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY.

In conclusion, the uniform problem variants do not offer new insights. However, they immediately provide:

Corollary 2. UNIFORM u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY *is* NP*-complete.* UNIFORM u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY *is* W[2]*-hard.*

Instead of a single parameter, one could also consider two or more parameters. (Formally, this is captured by our definition by combining those multiple parameters into one single parameter.) So, the FPT-question of FIXED $|\Sigma|$ u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY would be whether an algorithm exists that runs in time $f(u, k, |\Sigma|)p(\|D\|)$ for some function f and some polynomial p.

Theorem 5. FIXED $|\Sigma|$ u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY *is in* FPT.

Proof. It is easy to see that, given that u bounds the number of unification steps, for a sample larger than $u + |\Sigma| \cdot k$, the answer to FIXED $|\Sigma|$ u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY is always NO.

When the sample is smaller than this, we can obtain $\frac{1}{2} \cdot \|\mathrm{GF}(D)\| \cdot (\|\mathrm{GF}(D)\| - 1)$ as a bound for pairs of types and thus a bound on the size of the search-space of $\frac{u!}{(u+|\Sigma|\cdot k)!(|\Sigma|\cdot k)!}$. This is a constant, since u, k and $|\Sigma|$ are fixed. □

Theorem 6. FIXED $|\Sigma|$ u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY *is in* FPT.

Proof. As in the case for structure languages, for a sample larger than $u + |\Sigma| \cdot k$ the answer to FIXED $|\Sigma|$ u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY is NO. For smaller samples a consistent grammar may exist, so consider the number of derivations compatible with the strings in D. The length of the strings in D is upper bounded by $\|D\|$, and thus by $u + |\Sigma| \cdot k$, and the number of strings is $|D|$, which is upper bounded by $u + |\Sigma| \cdot k$, as well. Thus, given D, the number of possible structure samples D' is bounded by

$$\left(2^{u+|\Sigma|\cdot k-1} \frac{1}{u + |\Sigma| \cdot k} \binom{2(u + |\Sigma| \cdot k - 1)}{(u + |\Sigma| \cdot k - 1)}\right)^{u+|\Sigma|\cdot k}$$

which is a constant, since u, k and $|\Sigma|$ are fixed. For each of the possible D', FIXED $|\Sigma|$ u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY can be considered, which is in FPT. □

One could also study the step number u (or other subsets of parameters) as another parameterization. One problem formulation could be the following one: Given a finite sample D and a categorial grammar G (and the parameter u), does there exist a sequence of at most u unification steps, starting from the general form GF(D) and leading to G? This might be an interesting subject for future study. However, note that the slightly more general problem of deciding the existence of such a sequence from some arbitrary given grammar (not necessarily in general form) is already at least as hard as the well-known GRAPH ISOMORPHISM problem for $u = 0$.[1] We can encode a graph into a grammar by assigning to one single symbol the types T_i/T_j for every edge from vertex i to vertex j in the graph (and assigning T_ℓ and t/T_ℓ for every vertex ℓ to avoid useless types). Let G_1 and G_2 be two such grammars encoding graphs $Graph_1$ and $Graph_2$, then $u = 0$ (i.e., an empty sequence of unification steps) just if there exists a renaming such that, when it is applied to G_1, G_2 is obtained. It is easy to see that this is only the case if $Graph_1$ and $Graph_2$ are isomorphic.

6 Consequences for the Complexity of Learning

We have studied the parameterized complexity, for several classes of categorial grammars, of selecting a grammar consistent with a given (string- or structure) sample. Our results have a direct consequence for the complexity of learning: if this problem is computable in polynomial time, then a learning algorithm with polynomial update time may exist.

Our complexity results are summarized in Table 6. As far as the authors are aware, these are the first such results for learning problems.

Problem	Complexity
$\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY	W[2]-hard
UNIFORM u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY	W[2]-hard
UNIFORM u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY	NP-complete
FIXED $\lvert\Sigma\rvert$ u-STEP $\mathcal{FL}_{k-\text{VALUED}}$-CONSISTENCY	FPT
FIXED $\lvert\Sigma\rvert$ u-STEP $\mathcal{L}_{k-\text{VALUED}}$-CONSISTENCY	FPT

From a technical point of view, it would be nice to complement our W[2]-hardness result (Theorem 3) by demonstrating membership in W[2]. A natural idea would be to design a multi-tape Turing machine with one tape storing or counting the rule (applications) for each symbol. However, it is not clear if such a Turing machine would need only $f(k)$ many steps to decide consistency. Such a question might also be interpreted in the direction of parallelizability of derivations in categorial grammars. We are not aware of any such study for this type of mechanisms, but we would like to point to the fact that studies in this direction were undertaken for the weakly equivalent mechanism of context-free grammars, see [11] and the references therein.

[1] Though it is not known if GRAPH ISOMORPHISM is NP-complete, this problem is also believed not to be solvable in polynomial time, see [7,9].

The obvious interpretation of our positive (FPT) results would be that, as long as the parameters k, u, and $|\Sigma|$ are kept low, the classes are efficiently learnable. The last parameter is the most problematic, since for typical (NLP) applications the lexicon is very large. Thus, our analysis suggests the approach of choosing the total number of distinct types in the grammar as a parameter.

It would be interesting to study the consistency problem for other language class hierarchies, where each class has finite elasticity. One such example might be those based on elementary formal systems as examined by Moriyama and Sato [10]. This would be an interesting topic for future research.

References

1. Buszkowski, W.: Discovery procedures for categorial grammars. In: Klein, E., van Benthem, J. (eds.) Categories, Polymorphism and Unification. University of Amsterdam (1987)
2. Buszkowski, W., Penn, G.: Categorial grammars determined from linguistic data by unification. Studia Logica 49, 431–454 (1990)
3. Costa Florêncio, C.: Consistent identification in the limit of rigid grammars from strings is NP-hard. In: Adriaans, P.W., Fernau, H., van Zaanen, M. (eds.) ICGI 2002. LNCS (LNAI), vol. 2484, pp. 49–62. Springer, Heidelberg (2002)
4. Costa Florêncio, C.: Learning Categorial Grammars. PhD thesis, Universiteit Utrecht, The Netherlands (2003)
5. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Heidelberg (1999)
6. Gallai, T.: Über extreme Punkt- und Kantenmengen. Ann. Univ. Sci. Budapest, Eötvös Sect. Math. 2, 133–138 (1959)
7. Johnson, D.S.: The NP-completeness column. ACM Transactions on Algorithms 1(1), 160–176 (2005)
8. Kanazawa, M.: Learnable Classes of Categorial Grammars. PhD, CSLI (1998)
9. Köbler, J., Schöning, U., Torán, J.: Graph Isomorphism Problem: The Structural Complexity. Birkhäuser, Basel (1993)
10. Moriyama, T., Sato, M.: Properties of language classes with finite elasticity. In: Tomita, E., Kobayashi, S., Yokomori, T., Jantke, K.P. (eds.) ALT 1993. LNCS (LNAI), vol. 744, pp. 187–196. Springer, Heidelberg (1993)
11. Reinhardt, K.: A parallel context-free derivation hierarchy. In: Ciobanu, G., Păun, G. (eds.) FCT 1999. LNCS, vol. 1684, pp. 441–450. Springer, Heidelberg (1999)

Operator Precedence and the Visibly Pushdown Property[*]

Stefano Crespi Reghizzi and Dino Mandrioli

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
P.za Leonardo da Vinci 32, I–20133 Milano
{stefano.crespireghizzi,dino.mandrioli}@polimi.it

Abstract. Operator precedence languages, designated as Floyd's Languages (FL) to honor their inventor, are a classical deterministic context-free family. FLs are known to be a boolean family, and have been recently shown to strictly include the Visibly Pushdown Languages (VPDL); the latter are FLs characterized by operator precedence relations determined by the alphabet partition. In this paper we give the non-obvious proves that FLs have the same closure properties that motivated the introduction of VPDLs, namely under reversal, concatenation and Kleene's star. Thus, rather surprisingly, the historical FL family turns out to be the largest known deterministic context-free family that includes the VPDL and has the same closure properties needed for applications to model checking and for defining mark-up languages such as HTML. As a corollary, an extended regular expression of precedence-compatible FLs is a FL and a deterministic parser for it can be algorithmically obtained.

1 Introduction

From the very beginning of formal language science, research has struggled with the wish and need to extend as far as possible the elegant and practical properties of regular languages to other language families that overcome the limitations of finite-state models in terms of expressivity and allow more accurate modelling of relevant phenomena. In particular, it is well known that closure properties under basic operations allow to automatically construct complex models from simple ones, and to decide important problems: e.g. model checking relies on closure w.r.t. boolean operations and on decidability of the emptiness problem. Since, among the classic formal language families, only regular languages enjoy closure w.r.t. the needed operations, the search for new subclasses – mostly of context-free (CF) languages – exhibiting the wished properties, is a long-standing research concern.

A major step has been made by McNaughton with parenthesis grammars [12], whose productions are characterized by enclosing any righthand side (r.h.s) within a pair of parentheses; the alphabet is the disjoint union of internal characters and the pair. By considering instead of strings the *stencil* (or skeletal) trees encoded by parenthesized strings, some typical properties of regular languages that do not hold for CF languages are still valid: uniqueness of the minimal grammar, and boolean closure within the

[*] Partially supported by PRIN 2007TJNZRE-002, CNR-IEIIT and ESF AutoMathA.

class of languages having the same production stencils. Further mathematical developments of those ideas have been pursued in the setting of tree automata [16]. Short after McNaughton's results, we investigated similar closure properties of *Floyd's* operator precedence *Grammars* and *Languages* [10] [1] (FG and FL), elegant precursors of LR(k) grammars and Deterministic CF (DCF) languages, also exploited in early work on grammar inference [5]. The production set of an operator grammar determines three binary precedence relations (greater/less/equal) over the alphabet, that for a FG grammar are disjoint, and are presented in a matrix. The precedence matrix, even in the absence of the productions, fully determines the topology (or stencil) of the syntax tree, for any word that is generated by any FG having the same precedence matrix. The families of FGs that share the same precedence matrix and the corresponding languages are boolean algebras [8,5].

We also extended the notion of *non-counting* regular languages of McNaughton and Papert [13] to parenthesis languages and to FLs [6].

Decades later, novel interest for parentheses-like languages arose from research on mark-up languages such as XML, and produced the family of *balanced grammars* and languages [3]. They generalize parenthesis grammars in two ways: several pairs of parentheses are allowed, and the r.h.s of grammar rules allow for regular expressions over nonterminal and internal symbols to occur between matching parentheses. The property of uniqueness of the minimal grammar is preserved, and the family has the closure property w.r.t. concatenation and Kleene star, which was missing in parenthesis languages. Clearly, balanced as well as parenthesis languages, are closed under reversal.

Model checking and static program analysis provide another motivation for such families of languages – those that extend the typical finite-state properties to infinite-state pushdown systems. The influential paper by Alur and Madhusudan [1] (also in [2]) defines the *visibly pushdown automata* and *languages* (VPDA, VPDL), a subclass of realtime pushdown automata and DCF. The input alphabet is partitioned into three sets named calls (or opening), returns (or closing), and internals. The decision of the type of move to perform (push, pop, or a stack neutral move) is purely driven by the membership of an input character in one of the three sets, a property that justifies the name "visibly pushdown". VPDLs extend balanced grammars in ways that are important for modelling symbolic program execution. For each partitioned alphabet, the corresponding language family is closed under reversal and boolean operations, concatenation and Kleene star.

Guided by the intuition that precedence relations between terminals in a FG determine the action on the pushdown stack in a more flexible way than in a VPDA, we recently [7] proved that VPDLs are a proper subclass of FLs, characterized by a fixed partition of the precedence matrix, induced by the alphabetic partition into opening, closing and internal letters.

From the standpoint of their generative capacity and expressivity, FGs are more powerful than VPDAs in various practical ways. An example of structural adequacy possible with FG but not with VPDA is the semantic precedence of multiplication operators over

[1] We propose to name them *Floyd grammars* to honor the memory of Robert Floyd and also to avoid confusion with other similarly named but quite different types of precedence grammars.

additive ones (which inspired Floyd in the definition of operator precedence). An example of the higher generative capacity are the nested constructs opened and closed by means of a sequence of characters, e.g. `/* .... */`, rather than by two distinct single characters, one for the opening and one for the closing of the scope. Overall FGs, though less general than LR(1) grammars, offer a comfortable notation for specifying syntaxes of the complexity of programming languages, and are still occasionally used in compilation [11]. Surprisingly enough, nothing was known on the closure of FL under concatenation and Kleene star. This paper provides a positive but not immediate answer, in contrast to the fact that for the main language families closure under the two operations is either trivially present or trivially absent: examples of the former case are CF and VPDLs with a fixed alphabetic partitioning, while DCF is an example of the latter. In this perspective FGs represent an interesting singularity: they are closed but in a far from obvious way. Precisely, although the parse tree of a string $x \cdot y$ is solely determined by the given precedence relations of the grammars generating the two factors, the tree of $x \cdot y$ may be sharply different from the pasting together of the trees of x and y. The difficulty increases for Kleene star, because the syntax tree of, say, $x \cdot x \cdot x$ cannot be obtained by composing the trees of either $x \cdot x$ and x or x and $x \cdot x$, but may have an entirely different structure.

Thus, rather surprisingly, a classical, half-forgotten language family turns out to enjoy all the desirable properties that motivated the recent invention of VPDLs! To the best of our knowledge FG currently qualifies as the largest DCF family closed under boolean operations, reversal, concatenation and Kleene star.

The paper proceeds as follows: Section 2 lists the essential definitions of FG, and a summary of known results; Section 3 proves closure under concatenation; Section 4 proves closure under Kleene star and shows an application of the closure properties to regular expressions. Brevity precludes inclusion of the complete proofs, which are available from the first author's web page. Section 5 concludes.

2 Basic Definitions and Properties

We list the essential definitions of Floyd grammars. For the basic definitions of CF grammars and languages, we refer to any textbook, such as [15]. The empty string is denoted ε, the terminal alphabet is Σ. For a string x and a letter a, $|x|_a$ denotes the number of occurrences of letter a, and the same notation $|x|_\Delta$ applies also to a set $\Delta \subseteq \Sigma$; $first(x)$ and $last(x)$ denote the first and last letter of $x \neq \varepsilon$. The projection of a string $x \in \Sigma^*$ on Δ is denoted $\pi_\Delta(x)$.

A *Context-Free* CF grammar is a 4-tuple $G = (V_N, \Sigma, P, S)$, where V_N is the nonterminal alphabet, P is the production set, S is the axiom, and $V = V_N \cup \Sigma$. An *empty rule* has ε as right hand side (r.h.s.). A *renaming rule* has one nonterminal as r.h.s. A grammar is *reduced* if every production can be used to generate some terminal string. A grammar is *invertible* if no two productions have identical r.h.s.

The following naming convention will be adopted, unless otherwise specified: lowercase Latin letters a, b, c denote terminal characters; letters u, v, x, y, w, z denote terminal strings; Latin capital letters A, B, C denote nonterminal symbols, and Greek letters $\alpha, \ldots, \omega$ denote strings over V. The strings may be empty, unless stated otherwise.

For a production $A \to u_0 A_1 u_1 A_2 \dots u_{k-1} A_k$, $k \geq 0$, the *stencil* is the production $N \to u_0 N u_1 N \dots u_{k-1} N$, where N is not in V_N.

A production is in *operator form* if its r.h.s. has no adjacent nonterminals, and an *operator grammar* (OG) contains just such productions. Any CF grammar admits an equivalent OG, which can be also assumed to be invertible [15].

For a CF grammar G over Σ, the associated *parenthesis grammar* [12] $\widetilde{G}$ has the rules obtained by enclosing each r.h.s. of a rule of G within the parentheses '[' and ']' that are assumed not to be in Σ.

Two grammars G, G' are *equivalent* if they generate the same language, i.e., $L(G) = L(G')$. They are *structurally equivalent* if in addition the corresponding parenthesis grammars are equivalent, i.e., $L(\widetilde{G}) = L(\widetilde{G'})$.

For a grammar G consider a *sentential form* α with $S \overset{*}{\Rightarrow} \alpha$ and $\alpha = \beta A$, $A \in V_N$. Then A is a *Suffix of the Sentential Form* (SSF); similarly we define the *Prefix of a Sentential Form* (PSF).

The coming definitions for operator precedence grammars [10], here named *Floyd Grammars*, are from [8]. (See also [11] for a recent practical account.)

For a nonterminal A of an OG G, the *left and right terminal sets* are

$$\mathcal{L}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} Ba\alpha\} \qquad \mathcal{R}_G(A) = \{a \in \Sigma \mid A \overset{*}{\Rightarrow} \alpha a B\} \tag{1}$$

where $B \in V_N \cup \{\varepsilon\}$. The two definitions are extended to a set W of nonterminals and to a string $\beta \in V^+$ via

$$\mathcal{L}_G(W) = \bigcup_{A \in W} \mathcal{L}_G(A) \qquad \text{and} \qquad \mathcal{L}_G(\beta) = \mathcal{L}_{G'}(D) \tag{2}$$

where D is a new nonterminal and G' is the same as G except for the addition of the production $D \to \beta$. Notice that $\mathcal{L}_G(\epsilon) = \emptyset$. The definitions for $\mathcal{R}$ are similar. The grammar name G will be omitted unless necessary to prevent confusion.

R. Floyd took inspiration from the traditional notion of precedence between arithmetic operators, in order to define a broad class of languages, such that the shape of the parse tree is solely determined by a binary relation between terminals that are consecutive, or become consecutive after a bottom-up reduction step.

For an OG G, let α, β range over $(V_N \cup \Sigma)^*$ and $a, b \in \Sigma$. Three binary operator precedence (OP) relations are defined:

$$\begin{aligned} \text{equal-precedence: } & a \doteq b \iff \exists A \to \alpha a B b \beta, B \in V_N \cup \{\varepsilon\} \\ \text{takes precedence: } & a \gtrdot b \iff \exists A \to \alpha D b \beta \text{ and } a \in \mathcal{R}_G(D) \\ \text{yields precedence: } & a \lessdot b \iff \exists A \to \alpha a D \beta \text{ and } b \in \mathcal{L}_G(D) \end{aligned} \tag{3}$$

For an OG G, the *operator precedence matrix* (OPM) $M = OPM(G)$ is a $|\Sigma| \times |\Sigma|$ array that to each ordered pair (a, b) associates the set M_{ab} of OP relations holding between a and b. Between two OPMs M_1 and M_2, we define set inclusion and operations.

$$M_1 \subseteq M_2 \text{ if } \forall a, b : M_{1,ab} \subseteq M_{2,ab}, \quad M = M_1 \cup M_2 \text{ if } \forall a, b : M_{ab} = M_{1,ab} \cup M_{2,ab} \tag{4}$$

Definition 1. *G is an operator precedence or Floyd grammar (FG) if, and only if, $M = OPM(G)$ is a conflict-free matrix, i.e., $\forall a, b$, $|M_{ab}| \leq 1$. Two matrices are compatible if their union is conflict-free. A matrix is total if it contains no empty case.*

In the following all precedence matrices are conflict-free.

2.1 Known Properties of Floyd Grammars

We recall some relevant definitions and properties of FGs. To ease cross-reference, we follow the terminology of [8].

Definition 2. *Normal forms of FG*
A FG is in Fischer normal form [9] if it is invertible, the axiom S does not occur in the r.h.s. of any production, the only permitted renaming productions have S as left hand side, and no ε-productions exist, except possibly $S \to \varepsilon$.

An FG is in homogeneous normal form [8,5] if it is in Fischer normal form and, for any production $A \to \alpha$ with $A \neq S$, $\mathcal{L}(\alpha) = \mathcal{L}(A)$ and $\mathcal{R}(\alpha) = \mathcal{R}(A)$.

Thus in a homogeneous grammar, for every nonterminal symbol, all of its alternative productions have the same pairs of left and right terminal sets.

Statement 1. *For any FG G a structurally equivalent homogeneous FG H can be effectively constructed [8].*

Next we consider FGs having identical or compatible precedence relations and we state their boolean properties.

Definition 3. *Precedence-compatible grammars*
For a precedence matrix M, the class [8] C_M of precedence-compatible FGs is

$$C_M = \{G \mid OPM(G) \subseteq M\} .$$

The equal-precedence relations of a FG have to do with an important parameter of the grammar, namely the maximal length of the r.h.s. of the productions. Clearly, a production $A \to A_1 a_1 \ldots A_t a_t A_{t+1}$, where each A_i is a possibly missing nonterminal, is associated with $a_1 \doteq a_2 \doteq \ldots \doteq a_t$. If the $\doteq$ relation is circular, the grammar can have productions of unbounded length. Otherwise the length of any r.h.s. is bounded by $(2.c)+1$, where c is the length of the longest $\doteq$-chain. For both practical and mathematical reasons, when considering the class of FG associated to a given OPM, it is convenient to restrict attention to grammars with bounded r.h.s. This can be done in two ways.

Definition 4. *Right-bounded grammars*
The class $C_{M,k}$ of FGs with right bound $k \geq 1$ is defined as

$$C_{M,k} = \{G \mid G \in C_M \wedge (\forall \textit{ production } A \to \alpha \textit{ of } G, |\alpha| \leq k)\}$$

The class of $\doteq$-acyclic FGs is defined as

$$C_{M,\doteq} = \{G \mid G \in C_M \mid \textit{ matrix } M \textit{ is } \doteq\textit{-acyclic}\}$$

A class of FGs is right bounded if it is k-right-bounded for some k.

The class of $\doteq$-acyclic FGs is obviously right-bounded. Notice also that, for any matrix M, the set of the production stencils of the grammars in $C_{M,k}$ (or in $C_{M,\doteq}$) is finite.

The following closure properties are from [8] (Corol. 5.7 and Theor. 5.8).

Statement 2. *For every precedence matrix M, the class of FLs*

$$\{L(G) \mid G \in C_{M,k}\}$$

is a boolean algebra.

In other words, the proposition applies to languages generated by right-bounded FGs having precedence matrices that are included in, or equal to some matrix M. Notice that the top element of the boolean lattice is the language of the FG that generates all possible syntax trees compatible with the precedence matrix; in particular, if M is total, the top element is Σ^*.

We observe that the boolean closure properties of VPDL immediately follow from Statement 2 and from the fact that a VPDL is a FL characterized by a particular form of precedence matrix [7].

Other simple properties

Statement 3. *The class of FG languages is closed with respect to reversal.*

This follows from the fact that, if $a \lessdot b$ for a FG grammar G, then it holds $b \gtrdot a$ for the grammar G^R obtained by reversing the r.h.s. of the productions of G; and similarly for $a \gtrdot b$. The $a \doteq b$ relation is turned into $b \doteq a$ by production reversal. It follows that G^R is a FG.

It is interesting to briefly discuss the cases of very simple precedence matrices. First consider a matrix M containing only $\lessdot$, hence necessarily conflict free. Then the non-empty r.h.s.'s of the productions of any grammar in C_M may only be of the types aN or a. Therefore the grammar is *right-linear*. (Conversely for $\gtrdot$ and left-linearity.) Second, suppose M does not contain $\doteq$. Then any production of any grammar in C_M only admits one terminal character. Notice that *linear* CF grammars can be cast in that form, but not all linear CF languages are FG since they may be non-DCF.

To finish, we compare regular languages and FGs.

Statement 4. *Let $R \subseteq \Sigma^*$ be a regular language. There exists a FG grammar for R in the family $C_{M,2}$, where M is the precedence matrix such that $M_{ab} = \lessdot$ for all $a, b \in \Sigma$.*

The statement follows from the fact that every regular language is generated by a right-linear grammar. If the empty string is in R, the FG has the axiomatic rule $S \to \varepsilon$.

A stronger statement holding for any precedence matrix will be proved in Subsection 4.1 as a corollary of the main theorems.

3 Closure under Concatenation

Although FGs are the oldest deterministic specialization of CF grammars, the fundamental but non-trivial questions concerning their closure under concatenation and

Kleene star have never been addressed, to the best of our knowledge. This theoretical gap is perhaps due to the facts that DCF languages are not closed under these operations, and that the constructions used for other grammar or PDA families do not work for FG, because they destroy the operator grammar form or introduce precedence conflicts. The closure proofs to be outlined, though necessarily rather involved, are constructive and practical. The grammars produced for the concatenation (or the Kleene star) structurally differ from the grammars of the two languages to be combined in rather surprising ways: the syntax tree of the prefix string may invade the other syntax tree, or conversely, and such trespasses may occur several times.

A simple case is illustrated by $L \cdot L$ where $L = a^+ \cup b^+ \cup ab$ with precedences $a \lessdot a$, $b \gtrdot b$, $a \doteq b$. Then for $y_1 = aaa$ the structure is $\Big(a\big(a(a)\big)\Big)$, for $y_2 = bb$ the structure is $\big((b)b\big)$, but the structure of $y_1 \cdot y_2$ is $\Big(a\big(a(ab)b\big)\Big)$, which is not a composition of the two.

The following notational conventions apply also to Section 4. Let the grammars be $G_1 = (V_{N_1}, \Sigma, P_1, S_1)$ and $G_2 = (V_{N_2}, \Sigma, P_2, S_2)$, and the nonterminals be conveniently named $V_{N_1} = \{S_1, A_1, A_2, \ldots\}$ and $V_{N_2} = \{S_2, B_1, B_2, \ldots\}$, in order to have distinct names for the axioms and nonterminals of the two grammars.

To simplify the proofs we operate on FGs in homogeneous normal form.

For two sets $\Delta_1, \Delta_2 \subseteq \Sigma$ and a precedence sign, say $\lessdot$, the notation $\Delta_1 \lessdot \Delta_2$ abbreviates $\forall a \in \Delta_1, \forall b \in \Delta_2, a \lessdot b$. Moreover, we extend precedence relations from $\Sigma \times \Sigma$ to pairs of strings $\alpha, \beta \in (\Sigma \cup V_N)^+$ such that $\alpha\beta \notin \big((\Sigma \cup V_N)^* \cdot V_N V_N \cdot (\Sigma \cup V_N)\big)^*$ by positing $\alpha \lessdot \beta \iff \mathcal{R}(\alpha) \lessdot \mathcal{L}(\beta)$, and similarly for $\gtrdot$; for $\doteq$ the condition is $last\,(\pi_\Sigma(\alpha)) \doteq first\,(\pi_\Sigma(\beta))$.

When writing derivations and left/right terminal sets, we usually drop the grammar name when no confusion is possible.

Theorem 1. *Let G_1, G_2 be FGs such that $OPM(G_1)$ is compatible with $OPM(G_2)$. Then a FG grammar G can be effectively constructed such that*

$$L(G) = L(G_1) \cdot L(G_2) \text{ and } OPM(G) \supseteq OPM(G_1) \cup OPM(G_2)\,.$$

Proof. We give a few hints on the construction of G and support the intuition by means of figures and examples. For simplicity, we assume that $M = OPM(G_1) \cup OPM(G_2)$ is a total matrix. This does not affect generality, because at every step, the algorithm checks the precedence relation between two letters a and b, and if $M_{ab} = \emptyset$, it can *arbitrarily* assign a value to M_{ab}, thus obtaining a matrix compatible with M.

The core of the algorithm builds a "thread" of productions that joins the parse trees of x_1 and x_2, $x_1 \in L_1, x_2 \in L_2$. The thread is recursively built in accordance with the precedence relations that connect the letters occurring at the right of the parse tree of x_1 and at the left of the parse tree of x_2, respectively. Since the parsing driven by operator precedence relations is bottom-up, the initialization of the construction is based on the possible "facing" of the rightmost letter of x_1 and the leftmost one of x_2. If $x_1 = y_1 \cdot a$, $x_2 = b \cdot y_2$ and $a \doteq b$, then we build a production of the type $[AB] \to \ldots ab \ldots$, $[AB]$ being a new nonterminal (see Figure 1). If instead the rightmost part of x_1 can be parsed without affecting x_2 up to a derivation $N \overset{*}{\Rightarrow} y_1$ because $\mathcal{R}(N) \gtrdot b$, then,

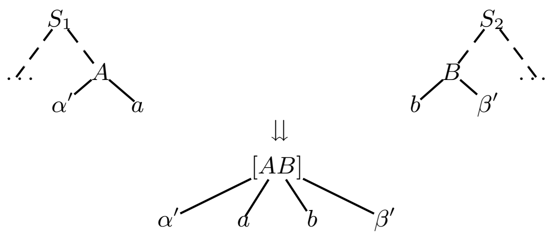


Fig. 1. Cross-border production constructed when the facing letters are equal in precedence

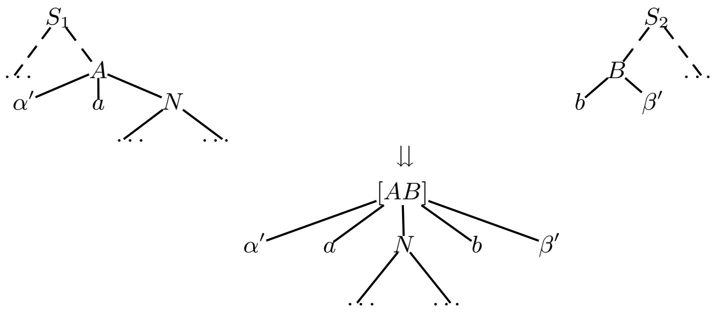


Fig. 2. Another case of cross-border production when $\mathcal{R}(N) \gtrdot b$

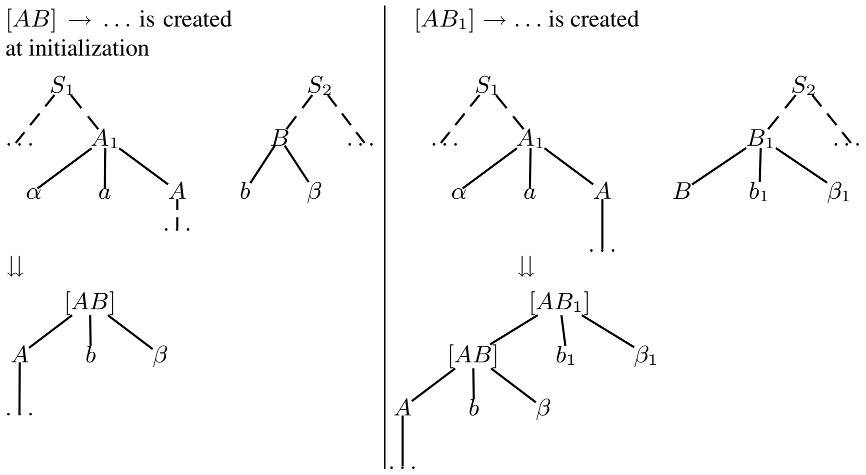


Fig. 3. Productions created by growing the cross-border thread

when the parsing of x_1 leads to a production such as $A \rightarrow \alpha \cdot a \cdot N$ with $a \doteq b$, the junction of the two syntax trees begins at that point by means of a production such as $[AB] \rightarrow \alpha \cdot a \cdot N \cdot b \cdot \beta$ (see Figure 2) so that the original precedence relations of G_1 and G_2 are unaffected.

Similar rules apply if instead $a \lessdot b$.

After this initialization, the construction of the "joint parsing thread" follows the natural bottom-up parsing. For instance, suppose that a nonterminal of type $[AB]$ has been built; this means that A is a SSF, B is a PSF and $[AB]$ "joins" two derivations $A \overset{*}{\Rightarrow} y_1$, at the end of a parse tree for some string x_1 of L_1 and $B \overset{*}{\Rightarrow} y_2$ at the start of a string x_2 of L_2; thus, if G_1 contains a rule $A_1 \to \alpha \cdot a \cdot A$ (A_1 being a SSF) and symmetrically $B_1 \to B \cdot b \cdot \beta$, with $a \doteq b$, then the new production $[A_1 B_1] \to \alpha \cdot a \cdot [AB] \cdot b \cdot \beta$ is created.

The cases $a \gtrdot b$ and $a \lessdot b$ are treated accordingly. The last situation is illustrated in Figure 3 (right). $\square$

4 Closure under Kleene Star

In many language families the closure under Kleene star comes together with the closure under union and concatenation. Thus for a CF language L, the syntax tree of a string $x = y_1 y_2 \dots y_i \in L^i$ with $y_j \in L$, is simply obtained by linking, in a left- or right-linear structure, the syntax trees of components $y_1, y_2, \dots, y_i$. In the case of FG, a similar composition is in general not possible, because the syntax tree of x may have a sharply different structure, as already observed for the concatenation $L \cdot L$.

A case is illustrated by the third power of language $L \supset a^+ \cup b^+ \cup c^+$, assuming the precedences (induced by further sentences not considered) to be: $a \lessdot a, a \doteq b, b \gtrdot b, b \doteq c, c \lessdot c$. Then the structure of a string such as $y_1 \cdot y_2 \cdot y_3 = a^3 \cdot b^2 \cdot c^2 \in L^3$ is not the composition of the structures of either $y_1 \cdot y_2$ and y_3, or of y_1 and $y_2 \cdot y_3$.

Before we enter the main topic, it is useful to return to the $\doteq$-acyclicity condition of Definition 4. Consider language $L = \{aa\}$ with the circular precedence relation $M = \{a \doteq a\}$, and a string $a^{2p}, p \geq 0$, in the Kleene closure of L. The FG grammar of L^* with OPM M would then need to contain an unbounded production set $\{S \to a^{2p}, p \geq 0\}$, which is not permitted by the standard definition[2] of CF grammar. For this reason we make the hypothesis that the precedence matrix is $\doteq$-acyclic.

Theorem 2. *Let $G = (V_N, \Sigma, P, S)$ be a FG such that $OPM(G)$ is $\doteq$-acyclic. Then a FG $\widehat{G} = (\widehat{V}_N, \Sigma, \widehat{P}, \widehat{S})$ with $OPM(\widehat{G}) \supseteq OPM(G)$ can be effectively built such that $L(\widehat{G}) = (L(G))^*$.*

As in Theorem 1 we assume without loss of generality the precedence matrix to be total, and in this case also $\doteq$-acyclic. Not surprisingly the construction of $\widehat{G}$ is based on the construction in Theorem 1 for $L.L$, but the required extensions involve some technical difficulties.

We need to consider only the irreflexive closure L^+, since for L^* it suffices to add the production $S \to \varepsilon$. We assume the form of grammar G to be homogeneous.

For brevity we give here just an intuitive description of the construction. $\widehat{G}$ is built as the last element of a series of grammars that begins with $G_1 = G$ and continues with the grammar G_2 that generates $L \cdot L \cup L$, computed according to the concatenation algorithm outlined in Section 3 and the union algorithm implied by Statement 2. Then G_3 is built

[2] We do not discuss the possibility of allowing regular expressions in the productions.

by iterating the application of the concatenation algorithm to L_2 and L itself. Notice, however, that this new application produces new nonterminals of the type $[[AB]C]$. Obviously this process cannot be iterated indefinitely since it would produce a grammar with infinite nonterminals and productions. Thus, nonterminals of type $[[AB]C]$ are "collapsed" into $[AC]$. Intuitively, this operation is justified by the observation that the production of an "intermediate" nonterminal of the type $[[AB]C]$ means that, in G, $A \overset{*}{\Rightarrow} x_1$ a suffix of some string $x \in L$, $B \overset{*}{\Rightarrow} y$ belonging to L, and $C \overset{*}{\Rightarrow} z_1$, a prefix of some $z \in L$. In this way, the number of possible new nonterminals is bounded and the construction of $\widehat{G}$ terminates when no new nonterminals and productions are generated. Figure 4 gives an idea of how the sequence G_1, G_2, G_3 is built.

Notice also that the details of the construction involve the production of so called *compound* nonterminals of type $\{[AB], [CD], E\}$, i.e., collection of "boundary nonterminals". This is due to the need to iteratively apply a normalization procedure that eliminates repeated r.h.s. For instance, suppose that during the process the following productions are built:

$$[AB] \to \alpha \mid \beta, \quad [CD] \to \alpha \mid \gamma, \quad E \to \alpha \mid \delta$$

where $A, B, C, D, E \in V_N$, and we recall that the nonterminals of the form of a pair $[AB]$ are created by the concatenation algorithm. Then, elimination of repeated r.h.s.'s produces a normalized homogeneous grammar containing the rules:

$$\{[AB], [CD], E\} \to \alpha, \quad \{[AB]\} \to \beta, \quad \{[CD]\} \to \gamma, \quad \{E\} \to \delta$$

$\square$

It is possible to prove that the closure of FL under boolean operations, concatenation and Kleene star, implies that certain subfamilies of FG languages are closed under the same set of operations: the cases of regular languages and of visibly pushdown languages over the same partitioned alphabet are straightforward.

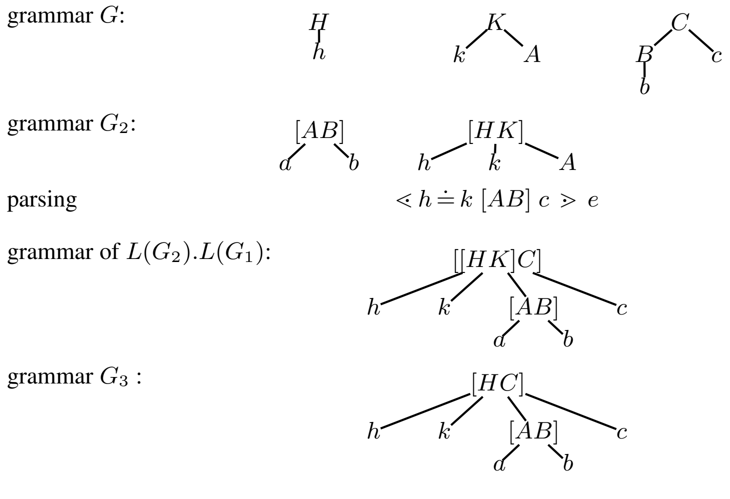


Fig. 4. Productions used for parsing a string in $L.L.L$

4.1 Regular Languages with Prescribed Precedences

For regular languages, we have already observed that their standard Chomsky grammar of type 3, say right-linear, is a very special FG containing only $\lessdot$ relations. Let $R \subseteq \Sigma^*$ be a regular language and let M be any precedence matrix. A more interesting question is whether it is possible to find a FG that generates R, with M as OPM. The positive answer follows from Theorems 1 and 2 under fairly general hypotheses.

Corollary 1. *Let M be any total $\doteq$-acyclic precedence matrix over Σ. Then the family of regular languages over Σ is (strictly) included in the family of languages generated by FGs ranging over C_M.*

Proof. Let R be defined by a regular expression. In order to construct a FG grammar with the given matrix M, we analyze the regular expression starting from the atomic subexpressions. Anytime two subexpressions are combined by union or concatenation respectively, the constructions of [8] or of Theorem 1 respectively produce a grammar, compatible with M, for the union or concatenation. Similarly, anytime a subexpression is under star, the construction of Theorem 2 produces the corresponding grammar. □

This result gives a procedure, based on the previous algorithms, for constructing from a regular expression, a FG grammar with the specified precedences. In particular, when the assigned precedences correspond to the left-linear (or right-linear) structure, the procedure returns a left-linear (or right linear) grammar. When the precedences are those of a VPL [1,7], the procedure returns a grammar with the specified partition of the alphabet into call, return and internal symbols.

Notice that the procedure works as well for ambiguous regular expressions. We are not aware of comparable methods for constructing a deterministic CF grammar, in order to parse in accordance with a prescribed syntactic structure, a language specified by a regular expression.

The same procedure, when applied to a regular expression, possibly augmented with intersection and complement, over precedence-compatible FG languages, permits to obtain an equivalent FG. From a practical standpoint, this approach would permit to construct a precedence parser for a grammar featuring "extended" regular expressions in the right-hand sides. This should of course be contrasted with the well-known non-closure of DCF under intersection and union.

5 Conclusions

We mention some questions raised by the present study.

Every class of Floyd languages over a given $\doteq$-acyclic precedence matrix includes (possibly after filling the precedence matrix to totality) the regular language family and is closed with respect to the basic operations: concatenation, Kleene star, reversal, and all boolean operations. The FG family appears at present to be the largest family exhibiting such properties; in particular it strictly includes [7] the visibly pushdown language (VPDL) family. On the other hand, several other language families fall in between the VPDL and the CF, such as the height-deterministic family of [14] or the synchronized pushdown languages of [4], but some of the basic closure properties are missing or the language family is non-deterministic. We wonder whether a significant family of deterministic CF languages, more general than FL, yet preserving the same closure properties, can be found.

In another direction, one could ask a similar question about the invariance property of FLs with respect to the non-counting (or aperiodicity) property [6].

It would be interesting to assess the suitability of Floyd languages for applications, especially to XML documents and to model checking, that have motivated other language models such as the balanced grammars and the visibly pushdown languages. We observe that the greater generative capacity of FGs should improve the realism of the intended models, by permitting the use of more flexible multi-level and recursively nested structures.

Finally, to apply these theoretical results in practice, e.g. to derive model checking algorithms therefrom, the computational complexity aspects should be investigated. Admittedly, the closure algorithms presented in this paper have a typical combinatorial nature: the worst case complexity in fact is dominated by the size of nonterminal alphabets which are constructed as power sets of the original ones. Notice also that the algorithms assume as starting point grammars in homogeneous normal form, which in turn require a nonterminal alphabet constructed on top of the power set of the terminal alphabet.

On the other hand, the risk of combinatorial explosion is rather intrinsic in these families of algorithms, starting from the seminal papers on model checking and VPDL. We hope that further research will produce suitable heuristics and techniques to manage such a complexity with the same success obtained by the long-standing research in model checking.

Aknowledgment. We thank the referees for their valuable suggestions and Jean Berstel for lively discussions.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC: ACM Symposium on Theory of Computing, STOC (2004)
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. J. ACM 56(3) (2009)
3. Berstel, J., Boasson, L.: Balanced grammars and their languages. In: Brauer, W., Ehrig, H., Karhumäki, J., Salomaa, A. (eds.) Formal and Natural Computing. LNCS, vol. 2300, pp. 3–25. Springer, Heidelberg (2002)
4. Caucal, D.: Synchronization of pushdown automata. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 120–132. Springer, Heidelberg (2006)
5. Crespi Reghizzi, S.: The mechanical acquisition of precedence grammars. PhD thesis, University of California UCLA, School of Engineering (1970)
6. Crespi-Reghizzi, S., Guida, G., Mandrioli, D.: Operator precedence grammars and the noncounting property. SICOMP: SIAM Journ. on Computing 10, 174–191 (1981)
7. Crespi-Reghizzi, S., Mandrioli, D.: Algebraic properties of structured context-free languages: old approaches and novel developments. In: WORDS 2009 - 7th Int. Conf. on Words, preprints (2009), http://arXiv.org/abs/0907.2130
8. Crespi-Reghizzi, S., Mandrioli, D., Martin, D.F.: Algebraic properties of operator precedence languages. Information and Control 37(2), 115–133 (1978)

9. Fischer, M.J.: Some properties of precedence languages. In: STOC '69: Proc. first annual ACM Symp. on Theory of Computing, pp. 181–190. ACM, New York (1969)
10. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM 10(3), 316–333 (1963)
11. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008)
12. McNaughton, R.: Parenthesis grammars. J. ACM 14(3), 490–500 (1967)
13. McNaughton, R., Papert, S.: Counter-free Automata. MIT Press, Cambridge (1971)
14. Nowotka, D., Srba, J.: Height-deterministic pushdown automata. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 125–134. Springer, Heidelberg (2007)
15. Salomaa, A.K.: Formal Languages. Academic Press, London (1973)
16. Thatcher, J.: Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. Journ. of Comp. and Syst. Sc. 1, 317–322 (1967)

On the Maximal Number of Cubic Runs in a String[⋆]

Maxime Crochemore[1,3], Costas Iliopoulos[1,4], Marcin Kubica[2], Jakub Radoszewski[2], Wojciech Rytter[2,5,⋆⋆], and Tomasz Waleń[2]

[1] King's College London, London WC2R 2LS, UK
maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk
[2] Dept. of Mathematics, Computer Science and Mechanics, University of Warsaw, Warsaw, Poland
{kubica,jrad,rytter,walen}@mimuw.edu.pl
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics, Copernicus University, Toruń, Poland

Abstract. A run is an inclusion maximal occurrence in a string (as a subinterval) of a repetition v with a period p such that $2p \leq |v|$. The maximal number of runs in a string of length n has been thoroughly studied, and is known to be between $0.944\,n$ and $1.029\,n$. In this paper we investigate cubic runs, in which the shortest period p satisfies $3p \leq |v|$. We show the upper bound of $0.5\,n$ on the maximal number of such runs in a string of length n, and construct an infinite sequence of words over binary alphabet for which the lower bound is $0.406\,n$.

1 Introduction

Repetitions and periodicities in strings are one of the fundamental topics in combinatorics on words [2,13]. They are also important in other areas: lossless compression, word representation, computational biology etc. Repetitions are studied from different points of view: classification of words not containing repetitions of a given exponent, efficient identification of factors being repetitions of different types and, finally, computing the bounds on the number of repetitions of a given exponent that a string may contain, which we consider in this paper. Both the known results in the topic and a deeper description of the motivation can be found in a survey by Crochemore et al. [5].

The concept of runs (also called maximal repetitions) has been introduced to represent all repetitions in a string in a succinct manner. The crucial property of

⋆ Research supported in part by the Royal Society, UK.
⋆⋆ Supported by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education.

runs is that their maximal number in a string of length n (denoted as $\mathsf{runs}(n)$) is $O(n)$ [10]. Due to the work of many people, much better bounds on $\mathsf{runs}(n)$ have been obtained. The lower bound $0.927\,n$ was first proved in [8]. Afterwards, it was improved by Kusano et al. [12] to $0.944\,n$ employing computer experiments, and very recently by Simpson [18] to $0.944575712\,n$. On the other hand, the first explicit upper bound $5\,n$ was settled in [15], afterwards it was systematically improved to $3.44\,n$ [17], $1.6\,n$ [3,4] and $1.52\,n$ [9]. The best known result $\mathsf{runs}(n) \leq 1.029\,n$ is due to Crochemore et al. [6], but it is conjectured [10] that $\mathsf{runs}(n) < n$. The maximal number of runs was also studied for special types of strings and tight bounds were established for Fibonacci strings [10,16] and more generally Sturmian strings [1].

The combinatorial analysis of runs is strongly related to the problem of estimation of the maximal number of squares in a string. In the latter problem the gap between the upper and lower bound is much larger than for runs [5,7]. However, a recent paper [11] by some of the authors shows that introduction of integer exponents larger than 2 may lead to obtaining tighter bounds for the number of corresponding repetitions.

In this paper we introduce and study the concept of cubic runs in which the period is at least three times shorter than the run itself. We show the following bounds on their maximal number, $\mathsf{cubic\text{-}runs}(n)$, in a string of length n:

$$0.406\,n < \mathsf{cubic\text{-}runs}(n) < 0.5\,n\ .$$

The upper bound is achieved by analysis of Lyndon words (i.e. words that are primitive and minimal/maximal in the class of their cyclic equivalents) that appear as periods of cubic runs. As for the lower bound, we describe an infinite family of binary words that contain more than $0.406\,n$ cubic runs.

2 Preliminaries

We consider *words* (*strings*) over a finite alphabet A, $u \in A^*$; the empty word is denoted by ε; the positions in u are numbered from 1 to $|u|$. By $\mathsf{Alph}(u)$ we denote the set of all letters of u. For $u = u_1 u_2 \dots u_m$, let us denote by $u[i\,..\,j]$ a *factor* of u equal to $u_i \dots u_j$ (in particular $u[i] = u[i\,..\,i]$). For the sake of simplicity, we assume that $u[i]$ is also defined for $i < 0$ or $i > |u|$, but it is different from any letter appearing in u.

Words $u[1\,..\,i]$ are called prefixes of u, and words $u[i\,..\,|u|]$ — suffixes of u. We say that a positive integer p is the (shortest) *period* of a word $u = u_1 \dots u_m$ (notation: $p = \mathsf{per}(u)$) if p is the smallest number such that $u_i = u_{i+p}$ holds for all $1 \leq i \leq m - p$.

If $w^k = u$ (k is a non-negative integer), that is $u = ww \dots w$ (k times), then we say that u is the k^{th} power of the word w. A *square* is the 2^{nd} power of some non-empty word. The *primitive root* of a word u, denoted $\mathsf{root}(u)$, is the shortest word w such that $w^k = u$ for some positive integer k. We call a word u *primitive* if $\mathsf{root}(u) = u$, otherwise it is called *non-primitive*. We say that words u and v

are cyclically equivalent (or that one of them is a cyclic rotation of the other) if $u = xy$ and $v = yx$ for some $x, y \in A^*$. It is a simple observation that if u and v are cyclically equivalent then $|\mathsf{root}(u)| = |\mathsf{root}(v)|$.

A *run* (also called a maximal repetition) in a string u is an interval $[i \,..\, j]$ such that:

– the associated factor $u[i \,..\, j]$ has period p,
– the length of the interval is at least $2p$, that is $2p \le j - i + 1$,
– the interval cannot be extended to the right or to the left, without violating the above properties, that is: $u[i-1] \neq u[i+p-1]$ and $u[j-p+1] \neq u[j+1]$.

A *cubic run* is a run $[i \,..\, j]$ for which the shortest period p satisfies $3p \le j-i+1$. By $\mathsf{cubic\text{-}runs}(u)$ we denote the number of cubic runs in a string u.

For simplicity, in the rest of the text we sometimes refer to runs or cubic runs as to occurrences of corresponding factors of u.

3 Fibonacci Strings

Let us start by analyzing the behavior of function cubic-runs for a very common benchmark in text algorithms, i.e., the Fibonacci strings, defined recursively as:

$$F_0 = a, \quad F_1 = ab, \quad F_n = F_{n-1}F_{n-2} \quad \text{for} \;\; n \ge 2 \; .$$

Denote by $\Phi_n = |F_n|$ the nth Fibonacci number (we assume that for $n < 0$, $\Phi_n = 1$) and by g_n the word F_n with the last two letters removed.

Lemma 1. [14,16] *Each run in F_n is of the form $F_k \cdot F_k \cdot g_{k-1}$ (short runs) or $F_k \cdot F_k \cdot F_k \cdot g_{k-1}$ (long runs), each of the runs of period Φ_k.*

Obviously, in Lemma 1 only runs of the form $F_k^3 \cdot g_{k-1}$ are cubic runs.

Denote by $\#occ(u, v)$ the number of occurrences (as a factor) of a word u in a word v.

Lemma 2. *For every $k, n \ge 0$,*

$$\#occ(F_k^3 \cdot g_{k-1},\ F_n) \;=\; \#occ(F_k^3,\ F_n) \; .$$

Proof. Each occurrence of F_k^3 within F_n must be followed by g_{k-1}, since otherwise it would form a run different from the ones specified in Lemma 1. □

Lemma 3. *For every $k \ge 2$ and $m \ge 0$,*

a) $\#occ(F_k^3,\ F_{m+k}) = \#occ(aaba,\ F_m)$,
b) $\#occ(aaba,\ F_m) = \Phi_{m-3} - 1$.

The (technical) proof of Lemma 3 will be included in the full version of the paper.

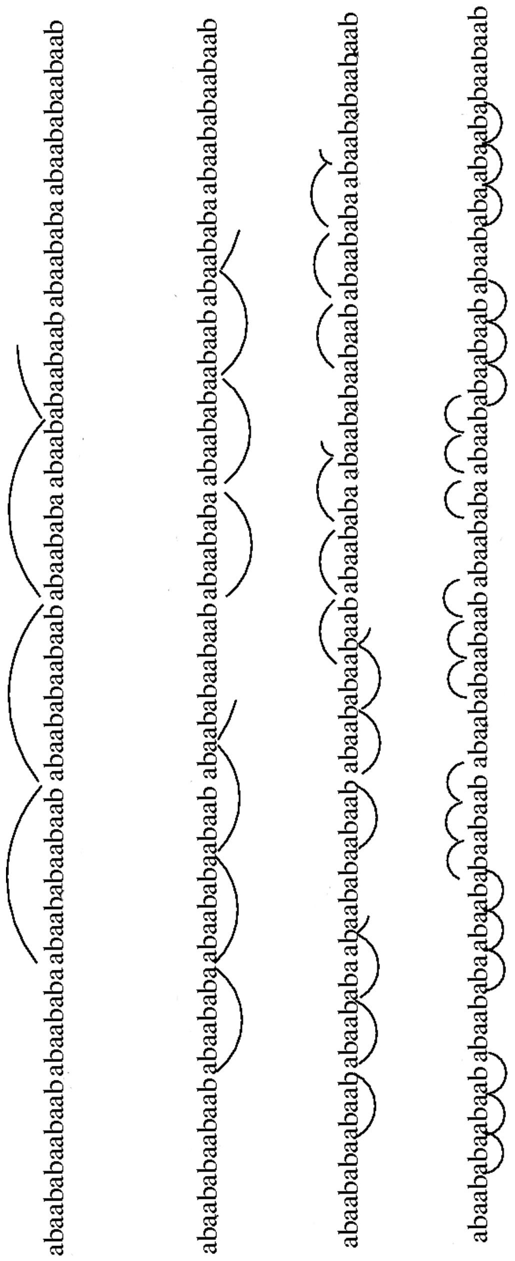

abaababaabaababaababaabaababaabaababaababaabaababaababaabaababaabaababaababaabaababaabaab

Fig. 1. The structure of cubic runs in the Fibonacci word F_9. The cubic runs are distributed as follows: 1 run $F_5^3 \cdot g_4$, 2 runs $F_4^3 \cdot g_3$, 4 runs $F_3^3 \cdot g_2$, and 7 runs F_2^3.

Lemma 4. *For $n > 5$, the word F_n contains (see Fig. 1):*

- $\Phi_{n-5} - 1$ *cubic runs* $F_2^3 \cdot g_1$
- $\Phi_{n-6} - 1$ *cubic runs* $F_3^3 \cdot g_2$

– ...
– $\Phi_1 - 1$ *cubic runs* $F_{n-4}^3 \cdot g_{n-5}$.

Words $F_0, F_1, \ldots, F_5$ *do not contain cubic runs.*

Proof. Let $n > 5$ and $k \in \{2, 3, \ldots, n-4\}$. Denote $m = n - k$. Combining the formulas from Lemmas 2 and 3, we obtain that:

$$\#occ(F_k^3 \cdot g_{k-1},\ F_n) \ = \ \#occ(F_k^3 \cdot g_{k-1},\ F_{m+k}) \ = \ \Phi_{m-3} - 1 \ = \ \Phi_{n-k-3} - 1. \square$$

We are now ready to describe the behaviour of the function $\mathsf{cubic\text{-}runs}(F_n)$. The following theorem not only provides an exact formula for it, but also shows a relationship between the number of cubic runs and the number of distinct cubes in Fibonacci words. This relationship is even more elegant than the corresponding relationship between the number of (ordinary) runs and the number of (distinct) squares in Fibonacci words, which always differ exactly by 1, see [14,16].

Theorem 1

a) $\mathsf{cubic\text{-}runs}(F_n) = \Phi_{n-3} - n + 2$.
b) $\lim_{n\to\infty} \frac{\mathsf{cubic\text{-}runs}(F_n)}{|F_n|} = \frac{1}{\phi^3} \approx 0.2361$, *where* $\phi = \frac{1+\sqrt{5}}{2}$ *is the golden ratio.*
c) The total number of cubic runs in F_n *equals the number of cubes in* F_n.

Proof. a) From Lemma 4 we obtain:

$$\mathsf{cubic\text{-}runs}(F_n) \ = \ \sum_{i=1}^{n-5} (\Phi_i - 1) \ = \ \Phi_{n-3} - 3 - (n-5) \ = \ \Phi_{n-3} - n + 2 \ .$$

b) It is a straightforward application of the formula from (a):

$$\lim_{n\to\infty} \frac{\mathsf{cubic\text{-}runs}(F_n)}{|F_n|} = \lim_{n\to\infty} \frac{\Phi_{n-3} - n + 2}{\Phi_n} = \frac{1}{\phi^3} \ .$$

c) It suffices to note that the number of cubes in $F_{k+1}^3 \cdot g_k$ is $|g_k| + 1 = \Phi_k - 1$, and thus the total number of distinct cubes in F_n equals $\sum_{k=1}^{n-5} (\Phi_k - 1)$. $\square$

4 Upper Bound

In this section we assume that A is totally ordered by $\leq$, what induces a lexicographical order on A^*, also denoted by $\leq$. We say that $v \in A^*$ is a *Lyndon word* if it is primitive and minimal or maximal in the class of words that are cyclically equivalent to it. It is known (see [13]) that a Lyndon word has no non-trivial prefix that is also its suffix.

Let $u \in A^*$ be a given word of length n. We will show the upper bound of $0.5\,n$ on the number of cubic runs in u.

Let us denote by $P = \{p_1, p_2, \ldots, p_{n-1}\}$ the set of inter-positions in u that are located *between* pairs of consecutive letters of u. We define a function H

assigning to each cubic run v in u a set of some inter-positions within v (called later on *handles*) — H is a mapping from the set of cubic runs occurring in u to the set 2^P of subsets of P. Let v be a cubic run with period p and let w be the prefix of v of length p. Let $w_{\min}$ and $w_{\max}$ be the minimal and maximal words (in lexicographical order) cyclically equivalent to w. $H(v)$ is defined as follows:

a) if $w_{\min} \neq w_{\max}$ then $H(v)$ contains inter-positions between consecutive occurrences of $w_{\min}$ in v and between consecutive occurrences of $w_{\max}$ in v,
b) if $w_{\min} = w_{\max}$ then $H(v)$ contains all inter-positions within v.

Example 1. If $w = abaab$ then $w_{\min} = aabab$, $w_{\max} = babaa$.

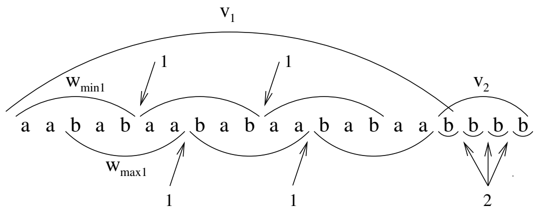


Fig. 2. An example of a word with two cubic runs v_1 and v_2. For v_1 we have $w_{min1} \neq w_{max1}$ and for v_2 the corresponding words are equal to b (a one-letter word). The inter-positions belonging to the sets $H(v_1)$ and $H(v_2)$ are pointed by arrows.

Lemma 5. *$w_{\min}$ and $w_{\max}$ are Lyndon words.*

Proof. By the definition of $w_{\min}$ and $w_{\max}$, it suffices to show that both words are primitive. This follows from the fact that, due to the minimality of p, w is primitive and that $w_{\min}$ and $w_{\max}$ are cyclically equivalent to w. □

Lemma 6. *Case (b) in the definition of $H(v)$ implies that $|w_{\min}| = 1$.*

Proof. $w_{\min}$ is primitive, therefore if $|w_{\min}| \geq 2$ then $w_{\min}$ contains at least two distinct letters, $a = w_{\min}[1]$ and $b = w_{\min}[i] \neq a$. If $b < a$ ($b > a$) then the cyclic rotation of $w_{\min}$ by $i-1$ letters would be lexicographically smaller (greater) than $w_{\min}$, so $w_{\min} \neq w_{\max}$. □

Note that in case (b) in the definition of H, $H(v)$ contains at least two distinct handles. The following lemma concludes that the same property also holds in case (a).

Lemma 7. *Each of the words $w_{\min}^2$ and $w_{\max}^2$ is a factor of v.*

Proof. Recall that $3p \leq |v|$, where $p = \mathsf{per}(v)$. By Lemma 6, this concludes the proof in case (b). In case (a), it suffices to note that the first occurrences of each of the words $w_{\min}$ and $w_{\max}$ within v start no further than p positions from the beginning of v. □

Now we show a crucial property of H.

Lemma 8 (Key lemma). *$H(v_1) \cap H(v_2) = \emptyset$ for any two distinct cubic runs v_1 and v_2 in u.*

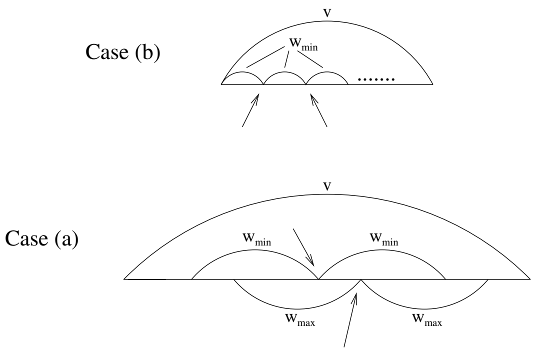


Fig. 3. Illustration of the definition of H and Lemma 7. The arrows in the figure show the elements of $H(v)$.

Proof. Assume, to the contrary, that $p_i \in H(v_1) \cap H(v_2)$ is a handle of two different cubic runs v_1 and v_2. By Lemmas 5 and 7, p_i is located in the middle of two squares of Lyndon words: w_1^2 and w_2^2, where $|w_1| = \mathsf{per}(v_1)$ and $|w_2| = \mathsf{per}(v_2)$. Note that $w_1 \neq w_2$, since otherwise runs v_1 and v_2 would be the same. Without the loss of generality, we can assume that $|w_1| < |w_2|$. So, the word w_1 is both a prefix and a suffix of w_2 (see Fig. 4), what contradicts the fact that w_2 is a Lyndon word. $\square$

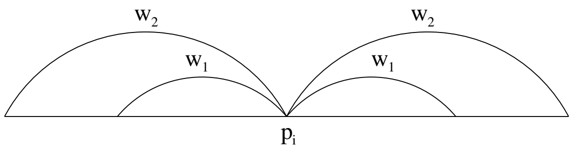


Fig. 4. A situation where p_i is in the middle of two different squares w_1^2 and w_2^2

The following theorem concludes the analysis of the upper bound.

Theorem 2. *A word $u \in A^*$ of length n may contain at most $\frac{n-1}{2}$ cubic runs.*

Proof. Due to Lemma 7, for each cubic run v in u, $|H(v)| \geq 2$. Since $|P| = n-1$, Lemma 8 implies the bound from the theorem. $\square$

5 Lower Bound

We start this section by a simple argument that leads to $0.4\,n$ lower bound for the number of cubic runs in a string, however over an arbitrarily large alphabet.

Theorem 3. *For any word s there exists an infinite sequence of words $(s_n)_{n=0}^{\infty}$, such that $s_0 = s$ and*

$$\lim_{n\to\infty} \frac{r_n}{\ell_n} = \frac{r}{\ell} + \frac{1}{5\ell}$$

where $r_n = \textit{cubic-runs}(s_n)$, $\ell_n = |s_n|$, $r = \textit{cubic-runs}(s)$, $\ell = |s|$.

Proof. The sequence $(s_n)_{n=0}^{\infty}$ is defined recursively. Let $s_0 = s$. Let $A = \mathsf{Alph}(s_n)$ and let $\overline{A}$ be a disjoint copy of A. By $\overline{u}$ let us denote a word obtained from word $u \in A^*$ by substituting letters from A with the corresponding letters from $\overline{A}$. The word s_{n+1} is defined as $s_{n+1} = (s_n\overline{s_n})^3$.

Recall that $\ell_0 = \ell$, $r_0 = r$, and note that for $n \geq 1$ we have $\ell_n = 6\ell_{n-1}$ and $r_n = 6r_{n-1} + 1$. Thus:

$$\tfrac{r_n}{\ell_n} \;=\; \tfrac{6r_{n-1}+1}{6\ell_{n-1}} \;=\; \tfrac{r_{n-1}}{\ell_{n-1}} + \tfrac{1}{6\ell_{n-1}}$$

and by simple induction we obtain:

$$\tfrac{r_n}{\ell_n} \;=\; \tfrac{r}{\ell} + \tfrac{1}{\ell}\textstyle\sum_{j=1}^{n}\tfrac{1}{6^j} \;=\; \tfrac{r}{\ell} + \tfrac{1}{5\ell}\left(1 - \tfrac{1}{6^{n+1}}\right) .$$

Taking $n \to \infty$ in the above formula we conclude the proof. $\square$

Starting with a 3-letter word $s = a^3$, for which $r/\ell = 1/3$, we obtain the following sequence of words:

$$\begin{aligned} s_0 &= a^3 \\ s_1 &= \left(a^3b^3\right)^3 \\ s_2 &= \left(\left(a^3b^3\right)^3\left(c^3d^3\right)^3\right)^3 \\ &\dots \end{aligned}$$

which gives, by Theorem 3, the lower bound of $0.4\,n$.

However, this bound is not optimal — we will show an example sequence of *binary* words which gives the bound of $0.406\,n$. We introduce the following morphism:

$$\begin{aligned} \psi(a) &= \left(a^3b^3\right)^3 a^4b^3a \;=\; aaabbbaaabbbaaabbbaaaabbba, \\ \psi(b) &= \left(a^3b^3\right)^3 a \;=\; aaabbbaaabbbaaabbba . \end{aligned}$$

Recall that F_n is the n-th Fibonacci word.

Theorem 4. *There are infinitely many binary strings $\psi(F_n)$ such that*

$$\frac{r_n}{\ell_n} > 0.406 ,$$

where $r_n = \textsf{cubic-runs}(\psi(F_n))$, $\ell_n = |\psi(F_n)|$.

Proof. Let us denote

$$X = \left(a^3b^3\right)^3, \quad Y = a^4b^3a .$$

Thus $\psi(a) = XY$, $\psi(b) = Xa$. Also denote $w_n = \psi(F_n)$.

Example 2.

$$\begin{aligned} w_0 &= \left(a^3b^3\right)^3a^4b^3a \\ w_1 &= \left(a^3b^3\right)^3a^4b^3a\left(a^3b^3\right)^3a \\ w_2 &= \left(a^3b^3\right)^3a^4b^3a\left(a^3b^3\right)^3a\left(a^3b^3\right)^3a^4b^3a \end{aligned}$$

n	r_n	ℓ_n	r_n/ℓ_n
0	9	26	0.3462
1	17	45	0.3778
2	26	71	0.3662
3	45	116	0.3879
4	71	187	0.3796
5	119	303	0.3927
6	192	490	0.3918

Fig. 5. Characteristics of a first few elements of the sequence (w_n)

We will show that for sufficiently large n we have $\frac{r_n}{\ell_n} > 0.406$. Note that

$$\ell_n = \ell_{n-1} + \ell_{n-2} \ . \tag{1}$$

Also note that:

$$w_n = \psi(F_n) = \psi(F_{n-1}F_{n-2}) = \psi(F_{n-1})\psi(F_{n-2}) = w_{n-1}w_{n-2} \ .$$

Let us define recursively a sequence:

$$\begin{array}{lll} t_n = r_n & \text{for} & n \le 4 \\ t_n = t_{n-1} + t_{n-2} + n - 4 & \text{for} & 2 \mid n \text{ and } n \ge 6 \\ t_n = t_{n-1} + t_{n-2} + n - 2 & \text{for} & 2 \nmid n \text{ and } n \ge 5 \ . \end{array} \tag{2}$$

Claim 1. $r_n \ge t_n$.

Proof. For each word w_n we will identify t_n cubic runs appearing in it, and we will show that the runs identified in w_{n-1} and w_{n-2} do not merge in $w_n = w_{n-1}w_{n-2}$. Hence, we obtain the recursive part $(t_{n-1} + t_{n-2})$ of the equations defining t_n. Moreover, we will identify a number of new cubic runs overlapping the concatenation $w_n = w_{n-1} \cdot w_{n-2}$.

First, let us have a look at the building blocks, from which the words w_n are constructed, that is the words $\psi(a) = XY = aaabbbaaabbbaaabbbaaaabbba$ and $\psi(b) = Xa = aaabbbaaabbbaaabbba$. The word XY contains $r_0 = t_0 = 9$ cubic runs: 8 with period 1 and one with period 6, and the word Xa contains 7 cubic runs: 6 with period 1 and one with period 6. But how these building blocks can be combined together?

Recall the following simple property of Fibonacci words: none of the words F_n contains a factor bb. Thus, in each of these words, each letter b can be surrounded only by letters a. As a consequence, words $\psi(b) = Xa$ used to construct words w_n can by surrounded only by $\psi(a) = XY$. If we have a look at the words $XYXa$ and $XaXY$, and analyze cubic runs appearing in Xa, it turns out that none of them merges with the cubic runs appearing in XY. Also, cubic runs appearing in XY do not merge with cubic runs appearing in Xa. Similarly, if we look at the word $XYXY$, it also turns out that the cubic runs appearing in one of the XY's do not merge with the cubic runs appearing in the other XY.

Hence, all the cubic runs of periods 1 and 6 in w_n, contributed by $\psi(a) = XY$ and $\psi(b) = Xa$, can be counted separately.

On the other hand, concatenations of XY and Xa appearing in w_n can introduce new runs with longer periods. Concatenation XYX introduces a new cubic run $(aaabbba)^3aa$ with period 7 (that cannot be further extended). Hence, both $\psi(aa) = XYXY$ and $\psi(ab) = XYXa$ introduce such a cubic run. Thus we obtain $r_1 = t_1 = 17$ and $r_2 = t_2 = 26$.

For $w_3 = w_2w_1 = (XYXaXY)(XYXa)$ we have 43 cubic runs contributed by w_2 and w_1, and two cubic runs overlapping their concatenation. One of them comes from XYX and (as previously observed) cannot be further extended. The other one is a suffix of w_3 of the form $a^3b^3aXYXYXa = (a^3b^3aXa)^3$. It cannot be extended to the left, but we have to show that when w_3 is used to build longer words w_n, it does not merge with any other cubic run. Let us note that in such a case it is always followed by w_2. Hence, this cubic run can extend to the right, but only for 7 characters, and does not merge with any other cubic runs. For $w_4 = w_3w_2$ we have only $45 + 26 = 71$ cubic runs contributed by w_3 and w_2.

Now, let us consider words w_n for $n \geq 5$. We have shown that cubic runs contributed by w_1, w_2, w_3 and w_4 used to build w_n do not merge and can be counted separately.

A new type of cubic runs that appears in w_n for $n \geq 5$ are runs present in the words F_n — each cubic run v in F_n corresponds to a cubic run $\psi(v)$. Due to Theorem 1, we obtain

$$\begin{aligned}&\textsf{cubic-runs}(F_n) - \textsf{cubic-runs}(F_{n-1}) - \textsf{cubic-runs}(F_{n-2}) = \\ &= \Phi_{n-3} - n + 2 - (\Phi_{n-4} - n + 3) - (\Phi_{n-5} - n + 4) = n - 5\end{aligned}$$

such cubic runs overlapping the concatenation of F_{n-1} and F_{n-2}, consequently new cubic runs overlapping the concatenation of w_{n-1} and w_{n-2}. Obviously, these runs do not merge with the ones that were considered previously.

We only have to show just three more cubic runs for odd n and one more for even n.

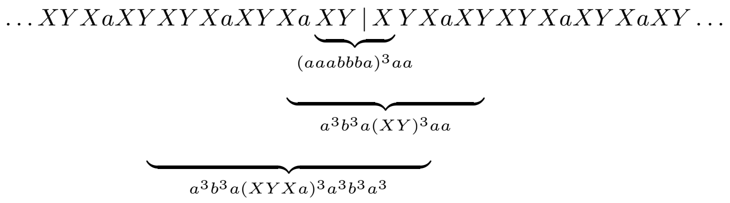
First, let us assume that n is odd. Let us note that w_4 is a suffix of w_{n-1} and w_3 is a prefix of w_{n-2}. Hence, every cubic run overlapping the concatenation $w_5 = w_4 \cdot w_3$ also overlaps the concatenation $w_n = w_{n-1} \cdot w_{n-2}$. There are three such cubic runs that we have not considered yet — they are shown in Figure 6.

XY is a suffix of w_4 and X is a prefix of w_3, consequently XYX introduces a cubic run $(aaabbba)^3aa$ with period 7 (that cannot be further extended).

Let us note that Y is a prefix of aX. Therefore, $\psi(a) = XY$ is a prefix of $\psi(ba) = XaXY$. Hence, since $XYXaXY$ is a prefix of w_3, $XYXYXaXY$ introduces a cubic run $a^3b^3a(XY)^3aa$ with period 78 (that cannot be further extended).

Because a is a prefix of Y and $(XYXa)^2XY$ is a suffix of w_4, $(XYXa)^2XYXY$ overlaps the concatenation of w_4 and w_3, and introduces a cubic run $(XYXa)^3$ with period 135, which can be extended to $a^3b^3a(XYXa)^3a^3b^3a^3$ (but no further).

Now, let us assume that n is even. Note that w_5 is a suffix of w_{n-1} and w_4 is a prefix of w_{n-2}. Hence, every cubic run overlapping the concatenation $w_6 = w_5 \cdot w_4$



$\ldots XYXaXYXYXaXYXa\,XY\,|\,X\,YXaXYXYXaXYXaXY\ldots$
$(aaabbba)^3aa$
$a^3b^3a(XY)^3aa$
$a^3b^3a(XYXa)^3a^3b^3a^3$

Fig. 6. Additional cubic runs overlapping concatenation $w_n = w_{n-1}w_{n-2}$, for odd $n \geq 5$

also overlaps the concatenation $w_n = w_{n-1} \cdot w_{n-2}$. There is one such cubic run that we have not considered yet. $XYXa$ is a suffix of w_5 and $XYXaXYXY$ is a prefix of w_4. Because a is a prefix of Y, $XYXaXYXaXYXY$ introduces a cubic run $(XYXa)^3$ with period 135, which can be extended to $a^3b^3a(XYXa)^3a^3b^3a^3$ (but no further).

$$\ldots XYX\underbrace{YXYXa\,|\,XYXaXYXYX}_{a^3b^3a(XYXa)^3a^3b^3a^3}aXY\ldots$$

This ends the proof of Claim 1. □

Completing the proof of Theorem [4]. We prove by induction that for $n \geq 19$, $r_n \geq 0.406 \cdot \ell_n$. The following inequalities:

$$\frac{r_{19}}{\ell_{19}} \geq \frac{103\,664}{255\,329} > 0.406\ ,$$

$$\frac{r_{20}}{\ell_{20}} \geq \frac{167\,740}{413\,131} > 0.406\ ,$$

are consequences (obtained by heavily using a calculator) of the formulas ([1]), ([2]) and Claim 1. The inductive step (for $n \geq 21$) follows from:

$$r_n - 0.406 \cdot \ell_n \ \geq\ t_n - 0.406 \cdot \ell_n \ \geq\ t_{n-1} + t_{n-2} - 0.406(\ell_{n-1} + \ell_{n-2}) \ >\ 0\ .$$

This concludes the inductive proof and also the proof of the whole theorem. □

Remark. A naive approach to obtain arbitrarily long binary words with large number of cubic runs would be to concatenate many copies of the same word $\psi(F_{19})$. However, it would not work, since some boundary runs can be glued together. Hence, a more complicated machinery was needed to prove Theorem [4].

References

1. Baturo, P., Piatkowski, M., Rytter, W.: The number of runs in Sturmian words. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 252–261. Springer, Heidelberg (2008)
2. Berstel, J., Karhumäki, J.: Combinatorics on words: a tutorial. Bulletin of the EATCS 79, 178–228 (2003)

3. Crochemore, M., Ilie, L.: Analysis of maximal repetitions in strings. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 465–476. Springer, Heidelberg (2007)
4. Crochemore, M., Ilie, L.: Maximal repetitions in strings. J. Comput. Syst. Sci. 74(5), 796–807 (2008)
5. Crochemore, M., Ilie, L., Rytter, W.: Repetitions in strings: Algorithms and combinatorics. Theor. Comput. Sci. 410(50), 5227–5235 (2009)
6. Crochemore, M., Ilie, L., Tinta, L.: Towards a solution to the "runs" conjecture. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 290–302. Springer, Heidelberg (2008)
7. Crochemore, M., Rytter, W.: Squares, cubes, and time-space efficient string searching. Algorithmica 13(5), 405–425 (1995)
8. Franek, F., Yang, Q.: An asymptotic lower bound for the maximal number of runs in a string. Int. J. Found. Comput. Sci. 19(1), 195–203 (2008)
9. Giraud, M.: Not so many runs in strings. In: Martín-Vide, C., Otto, F., Fernau, H. (eds.) LATA 2008. LNCS, vol. 5196, pp. 232–239. Springer, Heidelberg (2008)
10. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: Proceedings of the 40th Symposium on Foundations of Computer Science, pp. 596–604 (1999)
11. Kubica, M., Radoszewski, J., Rytter, W., Walen, T.: On the maximal number of cubic subwords in a string. In: Fiala, J., Kratochvíl, J., Miller, M. (eds.) IWOCA 2009. LNCS, vol. 5874, pp. 345–355. Springer, Heidelberg (2009)
12. Kusano, K., Matsubara, W., Ishino, A., Bannai, H., Shinohara, A.: New lower bounds for the maximum number of runs in a string. CoRR abs/0804.1214 (2008)
13. Lothaire, M.: Combinatorics on Words. Addison-Wesley, Reading (1983)
14. Mignosi, F., Pirillo, G.: Repetitions in the Fibonacci infinite word. ITA 26, 199–204 (1992)
15. Rytter, W.: The number of runs in a string: Improved analysis of the linear upper bound. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 184–195. Springer, Heidelberg (2006)
16. Rytter, W.: The structure of subword graphs and suffix trees in Fibonacci words. Theor. Comput. Sci. 363(2), 211–223 (2006)
17. Rytter, W.: The number of runs in a string. Inf. Comput. 205(9), 1459–1469 (2007)
18. Simpson, J.: Modified Padovan words and the maximum number of runs in a word. Australasian Journal of Combinatorics (to appear, 2010)

On the Hamiltonian Operators for Adiabatic Quantum Reduction of SAT

William Cruz-Santos and Guillermo Morales-Luna[⋆]

Computer Science Department, Cinvestav-IPN,
Mexico City, Mexico
gmorales@cs.cinvestav.mx

Abstract. We study the Hamiltonians resulting from the Adiabatic Quantum Computing treatment of the Satisfiability Problem SAT. We provide respective procedures for explicit calculation of the involved Hamiltonians. The statement of the ending Hamiltonians allows us to pose a variant of SAT which is also NP-complete.

1 Introduction

Adiabatic quantum computing (AQC) appeared [4,3] as a procedure to efficiently solve NP-hard problems and it has been extensively studied with respect to its computational capabilities [2,1]. The prototypical NP-complete problem, SAT, consisting of deciding the satisfiability of arbitrary Boolean conjunctive forms, has been dealt with adiabatic methods [5,6] as well as with other approaches based on Quantum Computing (QC) [7].

In Quantum Mechanics, a Hamiltonian is an operator acting on quantum states such that its eigenvalues determine the energy of the corresponding eigenstates. The spectrum of the Hamiltonian, i. e. the collection of its eigenvalues, is a bounded set in the complex plane and of particular interest are the eigenstates corresponding to extreme eigenvalues, in modulus.

In AQC, given a problem, say Π, two Hamiltonian operators H_{init} and H_{end} are defined such that the *ground states*, i.e. the eigenvectors corresponding to smallest (in absolute value) eigenvalues, of H_{init} are easily calculated, and the ground states of H_{end} codify solutions of Π. When the linear interpolation of the Hamiltonians, $H(t) = (1-t)H_{\text{init}} + tH_{\text{end}}$ is considered then the quantum system with Hamiltonian $H(t)$ will evolve ground states into ground states, provided the hypothesis of the Adiabatic Theorem are fulfilled (basically no eigenvalues paths cross during the system evolution, or the minimal spectral gap, i.e., the difference between the smallest and the greatest eigenvalues, is at least inverse polynomial).

The advantage of AQC is its physical foundation: the evolution of the quantum system will approximate the solutions of a given instance of the problem Π. Nevertheless the computer simulation of the process may require the explicit calculation of the involved Hamiltonians and this is an expensive procedure.

⋆ The authors acknowledge the partial support by Mexican Conacyt.

Namely, any instance $\mathbf{x}$ of size n determines a pair of Hamiltonians within the Hilbert space of n-quregisters, thus actually each Hamiltonian is represented by a $(2^n \times 2^n)$-complex matrix, hence the computation of $\Pi(\mathbf{x})$ entails an exponential space complexity.

In this paper we follow a SAT coding similar to the already standard codings [4,6] into AQC. We will consider 3-SAT: The satisfiability decision problem for 3-clauses. In section 3.3 we provide a procedural construction of an initial Hamiltonian for the given instances, in terms of the initial Hamiltonians corresponding to 3-clauses, whose ground states are the qu-registers determining uniform probability distributions. In section 3.2 we give two algorithms to calculate the same ending Hamiltonians, which are represented by diagonal matrices. This allows us to pose a polynomial reduction of SAT (hence also NP-complete).

2 SAT and AQC

2.1 Satisfiability Problem

Let $\mathcal{X} = (X_j)_{j=0}^{n-1}$ be a set of n Boolean variables. A *literal* has the form X^δ, with $X \in \mathcal{X}$, and $\delta \in \{0,1\}$: $X^1 = X$ and $X^0 = \neg X$. A *clause* is a disjunction of literals, and a *conjunctive form* (CF) is a conjunction of clauses. An assignment is a point $\varepsilon = (\varepsilon_j)_{j=1}^n \in \{0,1\}^n$ in the n-dimensional hypercube. Such an assignment *satisfies* the literal X_j^δ if and only if $\varepsilon_j = \delta$; it *satisfies* a clause whenever it satisfies a literal in the clause; and it *satisfies* a CF whenever it satisfies all clauses in the CF. An *m-clause* is a clause consisting of exactly m literals, and an *m-CF* is a CF consisting just of m-clauses.

The *satisfiability problem* SAT consists of deciding whether a given CF has a satisfying assignment. SAT is NP-complete and 3-SAT (the restriction of SAT to 3-CF's) is also NP-complete.

For any clause C, let $h_C : \{0,1\}^n \to \mathbb{R}$, be the map such that

$$\begin{aligned} \varepsilon \text{ satisfies } C &\Longrightarrow h_C(\varepsilon) = 0, \\ \varepsilon \text{ does not satisfy } C &\Longrightarrow h_C(\varepsilon) = 1. \end{aligned}$$

And for any CF $\phi = (C_i)_{i=0}^{m-1}$ let $h_\phi : \{0,1\}^n \to \mathbb{R}$ be $h_\phi = \sum_{i=0}^{m-1} h_{C_i}$. Clearly:

$$\forall \varepsilon \in \{0,1\}^n : \ [h_\phi(\varepsilon) = 0 \iff \varepsilon \text{ satisfies } \phi],$$

thus deciding the satisfiability of ϕ is reduced to decide whether the global minimum of h_ϕ is 0.

2.2 AQC Formulation of SAT

Let $e_0 = (1,0)$ and $e_1 = (0,1)$ be the vectors in the canonical basis of the Hilbert space $\mathbb{H}_1 = \mathbb{C}^2$. Let, for each $n > 1$, $\mathbb{H}_n = \mathbb{H}_{n-1} \otimes \mathbb{H}_1$ be the n-fold tensor power of $\mathbb{H}_1$. A basis of $\mathbb{H}_n$ is $(e_\varepsilon)_{\varepsilon \in \{0,1\}^n}$ where

$$\varepsilon = (\varepsilon_j)_{j=1}^n \implies e_\varepsilon = \bigotimes_{j=1}^n e_{\varepsilon_j}. \tag{1}$$

Let $\sigma_z : \mathbb{H}_1 \to \mathbb{H}_1$ be the Pauli quantum gate with matrix $\sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. For any bit $\delta \in \{0,1\}$ let $\tau_{\delta z} = \frac{1}{2}(I_2 - (-1)^\delta \sigma_z)$. Independently of δ, the characteristic polynomial of $\tau_{\delta z}$ is $p_z(\lambda) = (\lambda - 1)\lambda$ and its eigenvalues are 0 and 1 with unit eigenvectors e_0 and e_1. The correspondence among eigenvalues and eigenvectors is determined by δ, namely:

$$\forall \varepsilon \in \{0,1\} : \quad \tau_{\delta z} e_\varepsilon = (\delta \oplus \varepsilon) e_\varepsilon, \tag{2}$$

in words: if $\delta = 0$ the index of each eigenvector coincides with the eigenvalue, otherwise, it is the complementary value. Thus, the zero eigenvalue of the map $\tau_{\delta z}$ corresponds to the eigenvector e_δ.

For any $\delta \in \{0,1\}$ and $j_1 \in [\![1,n]\!]$, let $R_{E\delta j_1 n} = \bigotimes_{j_2=1}^n \rho_{z\delta j_2} : \mathbb{H}_n \to \mathbb{H}_n$ where $\rho_{z\delta j_2} = \text{identity}_{\mathbb{H}_1}$ if $j_2 \neq j_1$ and $\rho_{z\delta j_1} = \tau_{\delta z}$, thus the effect of $R_{E\delta jn}$ in an n-quregister is to apply $\tau_{\delta z}$ to the j-th qubit. Consequently,

$$\forall \varepsilon \in \{0,1\}^n : \quad R_{E\delta jn}\left(e_\varepsilon\right) = \left(\delta \oplus \varepsilon_j\right) e_\varepsilon, \tag{3}$$

thus the zero eigenvalue corresponds to the basic vectors giving a satisfying assignment for the literal X_j^δ. Given a 3-clause $C = X_{j_1}^{\delta_{j1}} \vee X_{j_2}^{\delta_{j2}} \vee X_{j_3}^{\delta_{j3}}$ let

$$H_{EC} = R_{E\delta_3 j_3 n} \circ R_{E\delta_2 j_2 n} \circ R_{E\delta_1 j_1 n} : \mathbb{H}_n \to \mathbb{H}_n.$$

Thus, for any $\varepsilon \in \{0,1\}^n$, $H_{EC}\left(e_\varepsilon\right) = 0$ if and only if ε satisfies the clause C; and it coincides with the linear map that on the basis vectors acts as $e_\varepsilon \mapsto h_C(\varepsilon) e_\varepsilon$. Thus, if $x = \sum_{\varepsilon \in \{0,1\}^n} x_\varepsilon e_\varepsilon$ then $H_{EC}(x) = \sum_{\varepsilon \in \{0,1\}^n} x_\varepsilon h_C(\varepsilon)\, e_\varepsilon$ and

$$\langle x | H_{EC}(x) \rangle = \sum_{\varepsilon \in \{0,1\}^n} \overline{x_\varepsilon} x_\varepsilon h_C(\varepsilon) = \sum_{\varepsilon \in \{0,1\}^n} |x_\varepsilon|^2 h_C(\varepsilon) \geq 0. \tag{4}$$

Hence H_{EC} is a positive operator. Indeed, we have $\langle x | H_{EC}(x) \rangle = 0$ if and only if $H_{EC}(x) = 0 \in \mathbb{H}_n$, and this happens if and only if x is a linear combination of those basic vectors indexed by assignments satisfying the clause C.

For a given CF $\phi = (C_i)_{i=0}^{m-1}$ let $H_{E\phi} : \mathbb{H}_n \to \mathbb{H}_n$ be $H_{E\phi} = \sum_{i=0}^{m-1} H_{EC_i}$. Again, $H_{E\phi}$ is positive and $H_{E\phi}(x) = 0$ if and only if x is a linear combination of those basic vectors indexed by assignments satisfying the CF ϕ.

An unit n-quregister $x \in \mathbb{H}_n$ such that $H_{E\phi}(x) = 0$ is called a *ground state* for $H_{E\phi}$. Thus:

Remark 1. In order to find a satisfying assignment for ϕ it is sufficient to find a ground state for $H_{E\phi}$.

Let $\sigma_x : \mathbb{H}_1 \to \mathbb{H}_1$ be the Pauli quantum gate with matrix $\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The map $\tau_{\delta x} = \frac{1}{2}(I_2 - (-1)^\delta \sigma_x)$ also has, independently of δ, characteristic polynomial $p_x(\lambda) = (\lambda - 1)\lambda$ and its eigenvalues are 0 and 1, now with corresponding unit eigenvectors $c_0 = \frac{1}{\sqrt{2}}\left(e_0 + e_1\right)$ and $c_1 = \frac{1}{\sqrt{2}}\left(-e_0 + e_1\right)$, which form an orthonormal basis of $\mathbb{H}_1$. The correspondence among eigenvalues and eigenvectors is determined as in relation (2) by δ, namely:

$$\forall \varepsilon \in \{0,1\} : \quad \tau_{\delta x} c_\varepsilon = (\delta \oplus \varepsilon) c_\varepsilon. \tag{5}$$

Let us also make

$$\varepsilon = (\varepsilon_j)_{j=1}^n \implies c_\varepsilon = \bigotimes_{j=1}^n c_{\varepsilon_j}.$$

For any $j_1 \in [\![1, n]\!]$, let $R_{Z\delta j_1 n} = \bigotimes_{j_2=1}^n \mu_{\delta j_2} : \mathbb{H}_n \to \mathbb{H}_n$ where $\mu_{\delta j_2} = \text{identity}_{\mathbb{H}_1}$ if $j_2 \neq j_1$ and $\mu_{\delta j_1} = \tau_{\delta x}$, thus the effect of $R_{Z\delta jn}$ in an n-quregister is to apply $\tau_{\delta x}$ to the j-th qubit. Consequently, as in relation (3):

$$\forall \varepsilon \in \{0,1\}^n : \quad R_{Z\delta jn}\,(c_\varepsilon) = (\delta \oplus \varepsilon_j)\, c_\varepsilon. \tag{6}$$

Hence whenever $\varepsilon_j = \delta$, c_ε is a ground state of the operator $R_{Z\delta jn}$.

Let us consider $\delta = 0$ and let us write $R_{Zjn} = R_{Z0jn}$. Given a 3-clause $C = X_{j_1}^{\delta_{j_1}} \vee X_{j_2}^{\delta_{j_2}} \vee X_{j_3}^{\delta_{j_3}}$ let $H_{ZC} = R_{Zj_1n} + R_{Zj_2n} + R_{Zj_3n} : \mathbb{H}_n \to \mathbb{H}_n$. Then H_{ZC} does not depend on the "signs" $\delta_{j_1}, \delta_{j_2}, \delta_{j_3}$ of the literals, but just on the variables appearing in the clause. The following implication holds:

$$\left[\varepsilon_{j_1} = \varepsilon_{j_2} = \varepsilon_{j_3} = 0 \implies H_{ZC}\,(z_\varepsilon) = 0\right]. \tag{7}$$

Given a CF $\phi = (C_i)_{i=0}^{m-1}$ let $H_{Z\phi} : \mathbb{H}_n \to \mathbb{H}_n$ be $H_{Z\phi} = \sum_{i=0}^{m-1} H_{ZC_i}$.

Remark 2. From relation of equation (6), $c_{00\cdots0} = \frac{1}{2^{\frac{n}{2}}} \sum_{\varepsilon\in\{0,1\}^n} e_\varepsilon$ is a ground state of $H_{Z\phi}$.

Remark 3. The following equation holds:

$$H_{Z\phi} = \sum_{j=1}^n d_j R_{Zjn} \tag{8}$$

where, for each $j \in [\![1, n]\!]$, $d_j = \text{card}\{i \in [\![1, m]\!] |\ X_j \text{ appears in } C_i\}$.

From remark 2 we have that there is a "natural" ground state, $c_{00\cdots0}$, for the operator $H_{Z\phi}$, while, after remark 1, to solve the SAT instance given by ϕ it is necessary to find a ground state for the operator $H_{E\phi}$. In summary, $c_{00\cdots0}$ is a ground state for $H_{Z\phi}$ but our aim is to find a ground state for $H_{E\phi}$.

For any 3-clause C, let us consider the map $I \to \text{GL}(\mathbb{H}_n)$, where $\text{GL}(\mathbb{H}_n)$ is the group of invertible linear automorphisms of the space $\mathbb{H}_n$, and $I = [0, 1]$ is the unit real interval, given as $t \mapsto H_C(t) = (1-t)H_{ZC} + tH_{EC}$.

For a CF $\phi = (C_i)_{i=0}^{m-1}$, let

$$H_\phi : t \mapsto H_\phi(t) = \sum_{i=0}^{m-1} H_{C_i}(t) = \sum_{i=0}^{m-1} \left[(1-t)H_{ZC} + tH_{EC}\right].$$

Let

$$\forall t \in [0,1] : \; i\frac{d}{dt}\psi(t) = H_\phi(t)\psi(t). \tag{9}$$

be the proper Schrödinger equation, with Hamiltonian H_ϕ.

Let $\{\eta_\nu\}_{\nu=0}^{2^n-1} \subset (\mathbb{R}^I)^{2^n}$ be the sequence of curves giving the eigenvalues of H_ϕ (indexed according to their absolute values at the initial points for $t = 0$). Then it is possible to see that η_0 and η_1 never cross on I, and, by the Adiabatic Theorem, there exists a $t_0 > 0$ such that the solutions ψ_{t_0} of the "scaled" equation

$$\forall t \in [0, t_0] : \ i\frac{d}{dt}\psi_{t_0}(t) = H_\phi\left(\frac{t}{t_0}\right)\psi_{t_0}(t) \tag{10}$$

are such that $\psi_{t_0}(t)$ gets arbitrarily close, as $t \nearrow t_0$, to a ground state for $H_{E\phi}$. A measurement of such ground state provides an assignment that either satisfies ϕ or maximizes the number of satisfied clauses in ϕ.

3 Procedural Construction of the Hamiltonian Operators

For any two integers $i, j \in \mathbb{N}$, $i \leq j$, let $[\![i, j]\!]$ denote the collection of integers ranging from i to j, $[\![i, j]\!] = \{i, i+1, \ldots, j-1, j\}$.

3.1 Hyperplanes in the Hypercube

Let us enumerate the n-dimensional hypercube with indexes in $[\![0, 2^n - 1]\!]$ associating each $i \in [\![0, 2^n - 1]\!]$ with its length n big-endian base-2 representation:

$$i \leftrightarrow \texttt{rev}\left((i)_2\right) = (\varepsilon_0, \ldots, \varepsilon_{n-1}) \in \{0,1\}^n \quad \text{where } i = \sum_{\nu=0}^{n-1} \varepsilon_\nu 2^\nu. \tag{11}$$

By putting each such representation as the i-th row of a rectangular array, a $(2^n \times n)$-matrix $\mathbf{E} \in \{0,1\}^{2^n \times n}$ is obtained. Let us denote by $\mathbf{e}_j^{(1)} \in \{0,1\}^{2^n}$ its j-th column, $j = 0, \ldots, n-1$. On one side, $\mathbf{e}_j^{(1)}$ can be written as the list $(0^{2^j}1^{2^j})^{2^{n-1-j}} = \mathbf{e}_j^{(1)}$, and on the other hand it can be seen as the Boolean map that has as support the hyperplane $E_j^1 : \varepsilon_j = 1$. Let $\mathbf{e}_j^{(0)}$ be the 2^n-vector obtained from $\mathbf{e}_j^{(1)}$ by taking the complement value at each entry. Then $\mathbf{e}_j^{(0)} = (1^{2^j}0^{2^j})^{2^{n-1-j}}$, and it represents the Boolean map with support the hyperplane $E_j^0 : \varepsilon_j = 0$. Clearly:

Remark 4. Each hyperplane E_j^δ is a $(n-1)$-dimensional affine variety at the hypercube and its characteristic map can be written as the list

$$\mathbf{e}_j^{(\delta)} = (\overline{\delta}^{2^j}\delta^{2^j})^{2^{n-1-j}}.$$

The lists $\mathbf{e}_j^{(\delta)}$ are easily computable:

Procedure. $(n-1)$-`DimensionalVarieties`.
Input: $\delta \in \{0,1\}$, $j \in [\![0, n-1]\!]$ and $k \in [\![0, 2^n - 1]\!]$.
Output: The k-th entry of the list $\mathbf{e}_j^{(\delta)}$.

```
1. Let $k_0 := k \bmod (2^{n-1-j})$.
2. If $k_0 \geq 2^j$ then output $\delta$ else output $\overline{\delta}$.
```

Two $(n-1)$-dimensional affine varieties are *parallel* if they are of the form E^0_j and E^1_j, for some index $j \in [\![0, n-1]\!]$.

Remark 5. The intersection of two parallel $(n-1)$-dimensional varieties is empty, while the intersection of any two non-parallel $(n-1)$-dimensional varieties is a $(n-2)$-dimensional affine variety, thus the intersection of any two non-parallel $(n-1)$-dimensional varieties has cardinality 2^{n-2}. Also, the intersection of three pairwise non-parallel $(n-1)$-dimensional affine varieties has cardinality 2^{n-3}.

3.2 The Hamiltonian Operators H_E

For any $\delta \in \{0,1\}$ and $j \in [\![0, n-1]\!]$, the transform $R_{E\delta jn} : \mathbb{H}_n \to \mathbb{H}_n$ defined in section 2.2, being the tensor product of transforms represented by diagonal matrices with respect to the canonical basis, is represented, with respect to the basis $(e_\varepsilon)_{\varepsilon\in\{0,1\}^n}$, by a diagonal matrix. Indeed:

Remark 6. The 2^n-length diagonal determining the diagonal matrix of $R_{E\delta jn}$ coincides with the list $\mathbf{e}^{(\delta)}_j = (\overline{\delta}^{2^j}\, \delta^{2^j})^{2^{n-1-j}}$.

For a 3-clause $C = X^{\delta_{j_1}}_{j_1} \vee X^{\delta_{j_2}}_{j_2} \vee X^{\delta_{j_3}}_{j_3}$, the operator $H_{EC} = R_{E\delta_3 j_3 n} \circ R_{E\delta_2 j_2 n} \circ R_{E\delta_1 j_1 n}$ is also represented by a diagonal matrix and its diagonal is the component-wise product of the lists $\mathbf{e}^{(\delta_1)}_{j_1}$, $\mathbf{e}^{(\delta_2)}_{j_2}$ and $\mathbf{e}^{(\delta_3)}_{j_3}$. Since the indexes j_1, j_2, j_3 are pairwise different, the lists are the characteristic maps of three pairwise non-parallel $(n-1)$-dimensional affine varieties. From remark 5:

Remark 7. With respect to the canonical basis $(e_\varepsilon)_{\varepsilon\in\{0,1\}^n}$ of $\mathbb{H}_n$, for any 3-clause $C = X^{\delta_{j_1}}_{j_1} \vee X^{\delta_{j_2}}_{j_2} \vee X^{\delta_{j_3}}_{j_3}$, the operator H_{EC} is represented by a diagonal matrix and its diagonal, $D_C(C) = D_C\left((j_1,\delta_1),(j_2,\delta_2),(j_3,\delta_3)\right)$, consisting of 2^{n-3} 1's, is such that each entry can be calculated by a slight modification of the procedure $(n-1)$-`DimensionalVarieties` outlined above. Namely:

Procedure. `3-ClauseDiagonal`.
Input: A 3-clause $C = \{(j_1,\delta_1),(j_2,\delta_2),(j_3,\delta_3)\}$, and $k \in [\![0, 2^n-1]\!]$.
Output: The k-th entry of the list D_C.

```
1. For $r = 1$ to 3 do
   (a) $k_{r0} := k \bmod (2^{n-1-j_r})$.
   (b) If $k_{r0} \geq 2^{j_r}$ then $x_r := \delta_r$ else $x_r := \overline{\delta}_r$.
2. Output $x_1 \cdot x_2 \cdot x_3$.
```

Remark 8. With respect to the canonical basis $(e_\varepsilon)_{\varepsilon\in\{0,1\}^n}$ of $\mathbb{H}_n$, for any CF $\phi = (C_i)_{i=0}^{m-1}$ the operator $H_{E\phi} = \sum_{i=0}^{m-1} H_{EC_i}$ is represented by a diagonal matrix, and its diagonal is $D_F(\phi) = \sum_{i=0}^{m-1} D_C(C_i)$.

For any 3-clause C, let $\mathrm{Spt}_C(C) = \{j \in [\![0, 2^n - 1]\!] |\ D_C(C)[j] \neq 0\}$ be the collection of indexes corresponding to non-zero entries at the vector in the diagonal $D_C(C)$. Then $\mathrm{card}(\mathrm{Spt}_C(C)) = 2^{n-3}$. Similarly, let $\mathrm{Spt}_F(\phi)$ be the collection of indexes corresponding to non-zero entries at the vector in the diagonal $D_F(\phi)$. Clearly:

$$\phi = (C_i)_{i=0}^{m-1} \implies \mathrm{Spt}_F(\phi) = \bigcup_{i=0}^{m-1} \mathrm{Spt}_C(C_i).$$

The entries at $D_F(\phi)$ are the eigenvalues of the operator $H_{E\phi}$, and the satisfying assignments are determined by the eigenvectors corresponding to the zero eigenvalue (if zero indeed is an eigenvalue). From remark 1 the following results:

Remark 9. Any zero entry in the 2^n-vector $D_F(\phi)$ determines a satisfying assignment for ϕ. Namely, if $D_F(\phi)[i] = 0$ then $\phi(\mathtt{rev}\,((i)_2)) = \mathtt{True}$.

This can also be stated as follows:

Remark 10. For a given CF $\phi = (C_i)_{i=0}^{m-1}$, ϕ is satisfiable if and only if the following happends $\mathrm{Spt}_F(\phi) \neq [\![0, 2^n - 1]\!]$.

Thus, the satisfiability problem can be rephrased as follows:

Problem QASAT.
Instance: A CF $\phi = (C_i)_{i=0}^{m-1}$.
Solution: "Yes" if $\mathrm{Spt}_F(\phi) \neq [\![0, 2^n - 1]\!]$; "No", if $\mathrm{Spt}_F(\phi) = [\![0, 2^n - 1]\!]$.

SAT is thus reducible to QUSAT in polynomial time, consequently QUSAT is NP-complete as well.

As a second construction of the vector at the diagonal $D_C(C)$ for any 3-clause, let us enumerate these clauses in another rather conventional manner.

In a general setting, let $k \geq 3$. Then the number of k-clauses, $C = \bigvee_{j \in J} X_j^{\delta_j}$, with $\mathrm{card}(J) = k$, in n variables, is $\nu_{kn} = \binom{n}{k} 2^k$. For any $i \in [\![0, \nu_{kn} - 1]\!]$ let $i_0 = i \bmod 2^k$ and $i_1 = (i - i_0)/2^k$. Then the map $\eta : i \mapsto (i_1, i_0)$ allows us to identify $[\![0, \nu_{kn} - 1]\!]$ with the Cartesian product $[\![0, \binom{n}{k} - 1]\!] \times [\![0, 2^k - 1]\!]$. The map η can also be seen as the function that to each index $i \in [\![0, \nu_{kn} - 1]\!]$ associates the clause $C = \bigvee_{j \in J_{i_1}} X_j^{\delta_j}$ where J_{i_1} is the i_1-th k-set of $[\![0, n - 1]\!]$ and $i_0 = \sum_{\kappa=0}^{k-1} \delta_{j_\kappa} 2^\kappa$.

Remark 11. Let $C = X_{j_1}^{\delta_{j_1}} \vee X_{j_2}^{\delta_{j_2}} \vee X_{j_3}^{\delta_{j_3}}$ be a 3-clause, $0 \leq j_1 < j_2 < j_3 < n$. Then the collection $\mathrm{Spt}_C(C)$ of indexes corresponding to non-zero entries at $D_C(C)$ is characterized as follows: For any $k \in [\![0, 2^n - 1]\!]$, $k \in \mathrm{Spt}_C(C) \iff$

$$\begin{aligned}
&\exists (k_0, k_1, k_2, k_3) \in K : \\
&(k_1 = \delta_1 \bmod 2) \,\&\, (k_2 = \delta_2 \bmod 2) \,\&\, (k_3 = \delta_3 \bmod 2) \,\& \\
&k = k_0 + 2^{j_1} k_1 + 2^{j_2} k_2 + 2^{j_3} k_3
\end{aligned}$$

where $K = [\![0, 2^{j_1} - 1]\!] \times [\![0, 2^{j_2 - j_1} - 1]\!] \times [\![0, 2^{j_3 - j_2} - 1]\!] \times [\![0, 2^{n - j_3} - 1]\!]$.

The remark 11 is consistent with the calculated cardinality of $\mathrm{Spt}_C(C)$ because: $2^{n-3} = 2^{j_1}2^{j_2-j_1-1}2^{j_3-j_2-1}2^{n-j_3-1}$. And also, it justifies an algorithm to compute $D_C(C)$. Namely:

Procedure `3-ClauseDiagonalBis`.
Input: A 3-clause $C = \{(j_1,\delta_1),(j_2,\delta_2),(j_3,\delta_3)\}$, and $k \in [\![0, 2^n-1]\!]$.
Output: The k-th entry of the list D_C.

```
1. $flg :=$ True ; $crk := k$ ;
2. $k_0 := crk \bmod 2^{j_1}$ ; $crk := (crk - k_0)/2^{j_1}$ ;
3. $k_1 := crk \bmod 2^{j_2-j_1}$ ; $crk := (crk - k_1)/2^{j_2-j_1}$ ;
4. $flg := (k_1 == \delta_1 \bmod 2)$ ;
5. If $flg$ then
   (a) $k_2 := crk \bmod 2^{j_3-j_2}$ ; $crk := (crk - k_2)/2^{j_3-j_2}$ ;
   (b) $flg := (k_2 == \delta_2 \bmod 2)$ ;
   (c) If $flg$ then
         i. $k_3 := crk \bmod 2^{j_3-j_2}$ ; $crk := (crk - k_3)/2^{n-j_3}$ ;
        ii. $flg := (k_3 == \delta_3 \bmod 2)$ ;
6. If $flg$ then $b := 1$ else $b := 0$;
7. Output $b$.
```

3.3 Hamiltonian Operator H_{Z_ϕ}

Now let us consider the operators with subindex Z defined in section 2.2.

Let us define the following matrices:

$$\begin{array}{lcl} A_0 = [1] & ; & B_0 = [1] \\ A_1 = I_2 \otimes A_0 - \frac{1}{2}\sigma_x \otimes B_0 & ; & B_1 = I_2 \otimes B_0 \\ A_2 = I_2 \otimes A_1 - \frac{1}{2}\sigma_x \otimes B_1 & ; & B_2 = I_2 \otimes B_1 \\ A_3 = I_2 \otimes (\frac{1}{2}B_2 + A_2) - \frac{1}{2}\sigma_x \otimes B_2 & ; & B_3 = I_2 \otimes B_2 \end{array} \quad (12)$$

where I_2 is the (2×2)-identity matrix. For each $k \le 3$, A_k, B_k are matrices of order $(2^k \times 2^k)$, indeed we have $B_k = I_{2^k}$.

For $n = 3$ and any 3-clause $C_{012} = X_0^{\delta_0} \vee X_1^{\delta_1} \vee X_2^{\delta_2}$ involving the three variables, the transform $H_{ZC_{012}} : \mathbb{H}_3 \to \mathbb{H}_3$ is represented, with respect to the canonical basis of $\mathbb{H}_3$, by the matrix

$$H_{[012],3} = A_3 = \frac{1}{2}\begin{pmatrix} 3 & -1 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & 0 & -1 & 0 & 0 \\ -1 & 0 & 3 & -1 & 0 & 0 & -1 & 0 \\ 0 & -1 & -1 & 3 & 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 & 3 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & -1 & 3 & 0 & -1 \\ 0 & 0 & -1 & 0 & -1 & 0 & 3 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & -1 & 3 \end{pmatrix}. \quad (13)$$

which is a band matrix with the following properties: its upper-right boundary is its diagonal at distance $4 = 2^{3-1}$ above the main diagonal, the lower-left boundary is also at distance 4 below the main diagonal, the main diagonal has constant value $\frac{3}{2}$ and the only values appearing in the matrix are $\frac{3}{2}, 0, -\frac{1}{2}$.

Naturally, for any $n > 3$ the transform $H_{ZC_{012}} : \mathbb{H}_n \to \mathbb{H}_n$ is represented by the matrix

$$H_{[012],n} = H_{[012],n-1} \otimes I_2. \tag{14}$$

The tensor product at eq. (14) substitutes each current entry at $H_{[012],n-1}$ by the product of that entry by the (2×2)-identity matrix. Thus also $H_{[012],n}$ is a band matrix, its boundaries are diagonals at distance 2^{n-1} from the main diagonal, the main diagonal has constant value $\frac{3}{2}$ and the only values appearing in the matrix are $\frac{3}{2}, 0, -\frac{1}{2}$. The following algorithm results:

Procedure `HZ012`.
Input: An integer $n \geq 3$ and a pair $(i, j) \in [\![0, 2^n - 1]\!]^2$.
Output: The (i, j)-th entry of the matrix $H_{[012],n}$.

```
 1. $k := j - i$ ;
 2. Case $k$ of
     0 : $v := \frac{3}{2}$.
  $\pm 2^\ell$ : (a power of 2, with $\ell \geq 1$)
          i. $\iota := \min\{i, j\}$ ;
         ii. $\iota_0 := \iota \bmod 2^{\ell+1}$ ;
        iii. If $\iota_0 < 2^\ell$ Then $v := -\frac{1}{2}$ Else $v := 0$ ;
  Else: $v := 0$ ;
 3. Output $v$.
```

For an arbitrary 3-clause $C_{j_1 j_2 j_3} = X_{j_1}^{\delta_1} \vee X_{j_2}^{\delta_2} \vee X_{j_3}^{\delta_3}$, with $0 \leq j_1 < j_2 < j_3 < n$ let $\pi_{j_1 j_2 j_3}$ be a permutation $[\![0, n-1]\!] \to [\![0, n-1]\!]$ such that $j_1 \mapsto 0$, $j_2 \mapsto 1$, $j_3 \mapsto 2$ and the restriction $\pi_{j_1 j_2 j_3}|_{[\![0,n-1]\!]-\{j_1,j_2,j_3\}}$ is a bijection $[\![0, n-1]\!] - \{j_1, j_2, j_3\} \to [\![3, n-1]\!]$. Then, there is a permutation $\rho_{j_1 j_2 j_3} : [\![0, 2^n - 1]\!] \to [\![0, 2^n - 1]\!]$, which can be determined in terms of $\pi_{j_1 j_2 j_3}$, such that the matrix $H_{[j_1 j_2 j_3],n}$ representing the transform $H_{ZC_{j_1 j_2 j_3}}$ is the action of $\rho_{j_1 j_2 j_3}$ over rows and columns on the matrix $H_{[012],n}$. Namely, let

$$\rho_{j_1 j_2 j_3} : [\![0, 2^n - 1]\!] \to [\![0, 2^n - 1]\!] \quad , \quad \sum_{\kappa=0}^{n-1} \varepsilon_\kappa 2^\kappa \mapsto \sum_{\kappa=0}^{n-1} \varepsilon_{\pi_{j_1 j_2 j_3}(\kappa)} 2^\kappa, \tag{15}$$

then when writing $H_{[012],n} = \left[h_{ij}^{(0)}\right]_{0 \leq i,j \leq 2^n - 1}$ one has

$$H_{[j_1 j_2 j_3],n} = \left[h^{(0)}_{\rho_{j_1 j_2 j_3}(i)\, \rho_{j_1 j_2 j_3}(j)}\right]_{0 \leq i,j \leq 2^n - 1}.$$

The following algorithm results:

Procedure `HZFor3Clauses`.
Input: An integer $n \geq 3$, a 3-clause $C = \{(j_1, \delta_1), (j_2, \delta_2), (j_3, \delta_3)\}$ and a pair $(i, j) \in [\![0, 2^n - 1]\!]^2$.
Output: The (i, j)-th entry of the matrix $H_{[j_1 j_2 j_3],n}$.

1. Compute the permutation $\rho_{j_1 j_2 j_3} : [\![0, n-1]\!] \to [\![0, n-1]\!]$ as in (15) ;
2. Output $\texttt{HZ012}[n; (\rho_{j_1 j_2 j_3}(i), \rho_{j_1 j_2 j_3}(j))]$.

(Evidently, the permutation $\rho_{j_1 j_2 j_3}$ can be computed as a preprocess to be used later for several entries (i, j).)

For a CF $\phi = (C_i = \{(j_{i1}, \delta_{i1}), (j_{i2}, \delta_{i2}), (j_{i3}, \delta_{i3}))_{i=0}^{m-1}$, the Hamiltonian operator $H_{Z\phi} : \mathbb{H}_n \to \mathbb{H}_n$ is represented by the matrix $H_{\phi,n} = \sum_{i=0}^{m-1} H_{[j_{i1} j_{i2} j_{i3}],n}$. Thus it can be computed directly by an iteration of algorithm `HZFor3Clauses`.

Remark 12. The ground eigenvector of the matrix $H_{\phi,n}$ will tend to a ground state of the matrix $H_{E\phi}$ when solving the Schrödinger equation (10), providing thus a solution of SAT for the instance ϕ.

4 Conclusions

The Adiabatic Theorem has been a powerful tool in Quantum Computing (QC) and it can be used to build efficient algorithms within the frame of QC for classical hard problems. The reduction techniques of these last problems to adiabatic terms allows us to pose equivalent polynomial reductions, which obviously are still hard in the classical sense. We have given explicit calculations of the involved Hamiltonians for a treatment of SAT through AQC. Besides its mathematical interest, this calculation allows us to check the correctness of the method through formal symbolic calculations of the eigensystems and to make computational simulations of the method.

References

1. Aharonov, D., van Dam, W., Kempe, J., Landau, Z., Lloyd, S., Regev, O.: Adiabatic quantum computation is equivalent to standard quantum computation. SIAM J. Comput. 37(1), 166–194 (2007)
2. Dam, W.V., Mosca, M., Vazirani, U.: How powerful is adiabatic quantum computation? In: FOCS 2001: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science, Washington, DC, USA, p. 279. IEEE Computer Society, Los Alamitos (2001)
3. Farhi, E., Goldstone, J., Gutmann, S., Lapan, J., Lundgren, A., Preda, D.: A quantum adiabatic evolution algorithm applied to random instances of an NP-complete problem. Science 292(5516), 472–475 (2001); Also arXiv:quant-ph/0104129v1
4. Farhi, E., Goldstone, J., Gutmann, S., Sipser, M.: Quantum computation by adiabatic evolution (2000), arXiv:quant-ph/0001106v1
5. Hogg, T.: Solving random satisfiability problems with quantum computers (2001), arXiv:quant-ph/0104048
6. Hogg, T.: Adiabatic quantum computing for random satisfiability problems (2002), arXiv:quant-ph/0206059
7. Leporati, A., Felloni, S.: Three "quantum" algorithms to solve 3-SAT. Theor. Comput. Sci. 372(2-3), 218–241 (2007)

Parametric Metric Interval Temporal Logic*

Barbara Di Giampaolo, Salvatore La Torre, and Margherita Napoli

Università degli Studi di Salerno, Via Ponte Don Melillo - 84084 Fisciano, Italy

Abstract. We study an extension of the logic MITL with parametric constants. In particular, we define a logic, denoted PMITL (parametric MITL), where the subscripts of the temporal operators are intervals with possibly a parametric endpoint. We consider typical decision problems, such as emptiness and universality of the set of parameter valuations under which a given parametric formula is satisfiable, or whether a given parametric timed automaton is a model of a given parametric formula. We show that when each parameter is used with a fixed polarity and only parameter valuations which evaluate parametric intervals to non-singular time intervals are taken into consideration, then the considered problems are decidable and EXPSPACE-complete. We also investigate the computational complexity of these problems for natural fragments of PMITL, and show that in meaningful fragments of the logic they are PSPACE-complete. Finally, we discuss other natural parameterizations of MITL, which indeed lead to undecidability.

1 Introduction

Temporal logic is a simple and standard formalism to specify the wished behaviour of a reactive system. Its use as a specification language was first suggested by Pnueli [15] who proposed the propositional linear temporal logic (LTL). This logic presents natural operators to express temporal requests on the time ordering of occurrences of events, such as "always", "eventually", "until", and "next".

The logic MITL [3] extends LTL with a real-time semantics where the changes of truth values happen according to a splitting of the line of non-negative reals into intervals. Syntactically, MITL augments the temporal operators of LTL (except for the next operator which has no clear meaning in a real-time semantics) with a subscript which expresses an interval of interest for the expressed property. Thus, properties such as "every time an a occurs then a b must occur within time $t \in [3, 5]$" become expressible. Also in this setting equality (which corresponds to use singular intervals in the subscripts of temporal operators) is a major problem for decidability, therefore the definition of MITL formulas syntactically excludes singular intervals as time constraints.

In this paper, we extend MITL with parametric constants, i.e., we allow the intervals in the subscripts of the temporal operators to have as an endpoint a *parametric expression* of the form $c + x$, for a parameter x and a constant c. Therefore, typical time properties which are expressible in MITL can now be analysed by varying the scope of the temporal operators depending on the values of the parameters. As an example,

* This work was partially funded by the MIUR grants FARB 2008-2009 Università degli Studi di Salerno (Italy).

consider a parameterized version of the above property: “every time an a occurs then a b must occur within a time $t \in [3, 5+x]$” where x is a parameter. One could be interested in determining if there exists a value of x such that this property holds on a given timed sequence, or if this is true for any possible value of x, or more, the set of x values such that the property holds.

The use of parametric constants has been advocated by many authors as a support to designers in the early stages of a design when not much is known on the system under construction (see for example [4,2,7,10,8,12,13,16]). Unfortunately, the unrestricted use of parameters leads to undecidability, and in particular, one should avoid testing a parameter for equality [4,2]. We define a logic, which we denote PMITL (parametric MITL), where in each interval at most one endpoint can be a parametric expression. Furthermore, we impose that each parameter is used with a fixed “polarity” and a parameter valuation is *admissible* if the evaluated intervals are non-singular and non-empty. The concept of polarity is semantic and is related to whether the space of the values for a parameter such that the formula is satisfied is upward or downward closed. For example, the set of values of x for the assertion “an a will eventually occur within time x” is upward closed: a model which satisfies this for $x = 3$ also satisfies it for every $x > 3$.

We study the analogous of satisfiability and model-checking for non-parametric temporal logic. As model of the formula we consider a model of timed automata with parameters, called L/U automaton [13], thus in the model-checking we allow parameters both in the formula and the model. L/U automata share the same kind of restriction on the parameters as PMITL and if a parameter is used in the formula and the model we just require that it is used with the same polarity. In particular, we define the set $S(\varphi)$ of all the admissible valuations such that the formula φ is satisfiable and the set $S(\mathcal{A}, \varphi)$ of all the admissible valuations such that the L/U automaton $\mathcal{A}$ is a model of the formula φ. We show that the universality and the emptiness problems for such sets are decidable and EXPSPACE-complete. The proof goes through a reduction to decidable problems for Büchi L/U automata [5].

We refine our complexity results on PMITL by showing several results. First, we prove that the considered decision problems are still EXPSPACE-hard in its fragment PMITL$_{0,\infty}$ where each non-parametric interval has one end-point which is either 0 or ∞, and that hardness still holds in the fragments of PMITL$_{0,\infty}$ with only parametric operators of one polarity. By restricting further the syntax of the operators coupled with parametric intervals whose end-points are neither 0 or ∞, we show that: (1) deciding the emptiness of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ for formulas φ where such operators are all of the form $\Diamond_{(c,d+x)}$ is in PSPACE, and (2) deciding the universality of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ for formulas ϕ where such operators are of all the form $\Box_{(c,d+x)}$ is in PSPACE. To the best of our knowledge, these fragments capture the most general formulation of parametric constraints in PMITL with the considered decision problems in PSPACE (indeed the logic studied in [5] allows parametric expressions of the form $c_1x_1 + \ldots + c_nx_n$, however parametric intervals have as end-points either 0 or ∞ and our results still hold when allowing such form of parametric expressions).

In the definition of PMITL we impose some restrictions on the parameters. In particular, we do not allow a parameter valuation to evaluate a parametric interval to a singular interval, each parameter is used with a fixed polarity consistently through all the formula, and in each interval we can use parameters in only one of the end-points. We show that if we relax any of these restrictions the resulting logic becomes undecidable. In fact, we have already mentioned that testing parameters for equality (and thus allowing singular intervals) leads to undecidability. Also, if a parameter is used with both polarities in a formula, then it is possible to express equality, and again the logic becomes undecidable. We thus consider some natural ways of defining parameterized intervals with parameters in both the end-points: both the end-points are added with the same parameter or with different parameters. In all such cases, the studied decision problems become undecidable in the resulting logic.

Related work. The need for restricting the use of parameters (in order to obtain decidability) such that each parameter is always used with a fixed polarity was addressed already in [2] for obtaining the decidability of a parametric extension of LTL (denoted PLTL). As already mentioned a parameterized fragment of PMITL is studied also in [5]. Both in [2] and [5], time constraints on temporal operators do not allow intervals with arbitrary end-points: one of the end-points is always 0 or ∞. Here, we give a thorough study of the parameterization with intervals and get a deeper insight on some concepts expressed there. The techniques used to show decidability results in [2,5], cannot be used directly in our settings, and we would like to stress that it was not clear to us in the beginning that PMITL was even decidable. Recently, the results on PLTL have been shown using different techniques in [14]. Parametric branching time specifications were first investigated in [16,12] where decidability is shown for logics obtained as extensions of TCTL [1] with parameters. In [6], decidability is extended to full TCTL with Presburger constraints over parameters. In [7], decidability is established for the model checking problem of *discrete-time* timed automata with *one parametric clock* against parametric TCTL without equality.

2 Parametric Dense-Time Metric Interval Temporal Logic

Notation. We consider non-empty intervals (convex sets) of non-negative real numbers. We use the standard notation $[a, b]$, $]a, b[$, $[a, b[$, and $]a, b]$ to denote respectively the closed, open, left-closed/right-open and left-open/right-closed intervals with end-points a and b. When we do not need to specify if an end-point is included or not in an interval, we simply use parentheses: for example, we denote with (a, b) any of the possible intervals with end-points a and b. A *time interval* I is an interval (a, b) such that $0 \leq a \leq b$, and $a < b$ if I is not closed. A closed time interval $I = [a, a]$ is called *singular*.

In the rest of the paper, we fix a set of atomic propositions AP and two disjoint sets of parameters U and L. A *parametric expression* is an expression of the form $c + x$, where $c \in \mathbb{N}$ and x is a parameter. With $\mathcal{E}(U)$ (resp. $\mathcal{E}(L)$) we denote the set of all the parametric expressions over parameters from U (resp. L). A *parameterized time interval* is an interval (a, b) such that either a or b belong to $\mathcal{E}(L) \cup \mathcal{E}(U)$. In the following, we sometimes use the term interval to indicate either a parameterized interval or a time interval. A *parameter valuation* $v : L \cup U \longrightarrow \mathbb{N}$ assigns a natural number to each parameter. Given a parameter valuation v and an interval I, with I_v we denote the time interval obtained by evaluating the end-points of I by v (in particular, if I is a time interval then $I_v = I$).

Syntax. The *Parametric dense-time Metric Interval Temporal Logic* (PMITL) extends MITL [3] by allowing parameterized time intervals as subscripts of temporal operators. The PMITL formulas over AP are defined by the following grammar:

$$\varphi := p \mid \neg p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \, \mathcal{U}_H \, \varphi \mid \varphi \, \mathcal{R}_J \, \varphi,$$

where $p \in AP$, and H and J are either non-singular time intervals with end-points in $\mathbb{N} \cup \{\infty\}$ or parameterized time intervals such that:

- for $H = (a, b)$ either (1) $a \in \mathbb{N}$ and $b \in \mathcal{E}(U)$ or (2) $a \in \mathcal{E}(L)$ and $b \in (\mathbb{N} \cup \{\infty\})$;
- for $J = (a, b)$ either (1) $a \in \mathbb{N}$ and $b \in \mathcal{E}(L)$ or (2) $a \in \mathcal{E}(U)$ and $b \in (\mathbb{N} \cup \{\infty\})$.

As usual, we use the abbreviations $\Diamond_H \varphi$ and $\Box_J \varphi$ for $true \; \mathcal{U}_H \; \varphi$ and $false \; \mathcal{R}_J \; \varphi$, respectively.

For a given formula φ, a parameter valuation v is *admissible* for φ if for each time interval I in the subscripts of φ, I_v is a non-singular time interval. With $\mathcal{D}(\varphi)$ we denote the set of all the admissible valuations for φ.

The logic $\mathrm{MITL}_{0,\infty}$ is defined in [3] as a syntactic restriction of MITL where all the time intervals (a, b), that are used as subscripts of the temporal operators, are such that either $a = 0$ or $b = \infty$. With $\mathrm{PMITL}_{0,\infty}$ we denote the parametric extension of $\mathrm{MITL}_{0,\infty}$ which corresponds to the fragment of PMITL where all the non-parameterized intervals are as in $\mathrm{MITL}_{0,\infty}$. Note that in [5], the acronym $\mathrm{PMITL}_{0,\infty}$ is used to denote the parametric extension of $\mathrm{MITL}_{0,\infty}$ where also the parameterized intervals are restricted such that one of the end-points is either 0 or ∞. Here, we prefer to denote such extension of $\mathrm{MITL}_{0,\infty}$ as $\mathrm{P}_{0,\infty}\mathrm{MITL}_{0,\infty}$ to stress the fact that the imposed syntactic restriction concerns both time and parametric intervals.

Semantics. PMITL formulas are interpreted over timed sequences. A *timed sequence* over AP is an infinite sequence $\alpha = (\alpha_0, I_0)(\alpha_1, I_1) \ldots$ such that:

- for all i, $\alpha_i \in 2^{AP}$ and I_i is a time interval with real end-points;
- for all i, $I_i \cap I_{i+1} = \emptyset$ and I_{i+1} follows I_i along the time line;
- each real number $t \geq 0$ belongs to some interval I_i.

For each $t \geq 0$, $\alpha(t)$ denotes the unique α_i such that $t \in I_i$. Given an interval $I = (a, b)$ and $t \geq -a$, with $I + t$ we denote the interval $(a + t, b + t)$ such that I is left-closed (resp. right-closed) iff $I + t$ is left-closed (resp. right-closed).

For a formula φ, a timed sequence α, a parameter valuation v, and $t \in \mathbb{R}_+$, the *satisfaction relation under valuation* v, $(\alpha, v, t) \models \varphi$, is defined as follows (we omit the standard clauses for boolean connectives):

- $(\alpha, v, t) \models p \Leftrightarrow p \in \alpha(t)$;
- $(\alpha, v, t) \models \varphi \, \mathcal{U}_H \, \psi \Leftrightarrow$ for some $t' \in H_v + t$, $(\alpha, v, t') \models \psi$, and $(\alpha, v, t'') \models \varphi$ for all $t < t'' < t'$;
- $(\alpha, v, t) \models \varphi \, \mathcal{R}_J \, \psi \Leftrightarrow$ for all $t' \in J_v + t$, either $(\alpha, v, t') \models \psi$, or $(\alpha, v, t'') \models \varphi$ for some $t < t'' < t'$.

A timed sequence α *satisfies* φ *under valuation* v, denoted $(\alpha, v) \models \varphi$, if $(\alpha, v, 0) \models \varphi$.

Polarity of parameterized temporal operators. A temporal operator of PMITL is *upward-closed* if the interval in its subscript has a parameter from U in one of its end-points. Analogously, a temporal operator of PMITL is *downward-closed* if the interval in its subscript has a parameter from L in one of its end-points. The meaning of these definition is clarified by the following lemma.

We first introduce two relations over parameter valuations. Let $\approx \in \{\leq, \geq\}$. Given any parameter valuations v and v', and a parameter z, the relation $\approx_z$ is defined such that: $v \approx_z v'$ iff $v(z) \approx v'(z)$ and $v(z') = v'(z')$ for any other parameter $z' \neq z$.

Lemma 1. *Let z be a parameter occurring in φ, α be a timed word and v and v' be parameter valuations.*
For $z \in U$ and $v \leq_z v'$, if $(\alpha, v) \models \varphi$ then $(\alpha, v') \models \varphi$.
For $z \in L$ and $v \geq_z v'$, if $(\alpha, v) \models \varphi$ then $(\alpha, v') \models \varphi$.

Parametric Timed Automata. *Parametric Timed Automata* extend Timed Automata, allowing the use of parameters in the clock constraints, see [4]. We briefly recall the definition. Given a finite set of clocks X and a finite set of parameters P, a *clock constraint* is a positive boolean combination of terms of the form $\xi \approx e$ where $\xi \in X$, $\approx \in \{<, \leq, \geq, >\}$, and either $e \in \mathbb{N}$ or e is a parametric expression. In the following, with Ξ we denote the set of all clock constraints over X and P. A *parametric timed automaton* (PTA) is a tuple $\mathcal{A} = \langle Q, Q^0, X, P, \beta, \Delta, \lambda \rangle$, where Q is a finite set of *locations*, $Q^0 \subseteq Q$ is the set of *initial locations*, X is a finite set of clocks, P is a finite set of parameters, β is a function assigning to each location q a *parametric clock constraint* $\beta(q)$, $\Delta \subseteq Q \times \Xi \times Q \times 2^X$ is a *transition relation*, and $\lambda : Q \rightarrow 2^{AP}$ is a function labeling each location with a set of atomic propositions.

A *clock interpretation* $\sigma : X \rightarrow \mathbb{R}_+$ assigns real values to each clock. A clock interpretation $\sigma + t$, for $t \in \mathbb{R}_+$, assigns $\sigma(\xi) + t$ to each $\xi \in X$. For $\gamma \subseteq X$, $\sigma[\gamma := 0]$ denotes the clock interpretation that assigns 0 to all clocks in γ, and $\sigma(\xi)$ to all the other clocks ξ. Given $q \in Q$, we say that a clock interpretation σ and a parameter valuation v satisfy a parametric clock constraint δ, denoted $(\sigma, v) \models \delta$, iff evaluating each clock of δ according to σ and each parameter of δ according to v, the resulting boolean expression holds true.

For locations $q_i \in Q$, clock interpretations σ_i, clock constraints $\delta_i \in \Xi$, clock sets $\gamma_i \subseteq X$, and intervals I_i, a *run* ρ of a PTA $\mathcal{A}$, under a parameter valuation v, is an infinite sequence $\xrightarrow{\sigma_0} (q_0, I_0) \xrightarrow[\delta_1,\gamma_1]{\sigma_1} (q_1, I_1) \xrightarrow[\delta_2,\gamma_2]{\sigma_2} (q_2, I_2) \xrightarrow[\delta_3,\gamma_3]{\sigma_3} \ldots$, such that:

- $q_0 \in Q^0$, and $\sigma_0(\xi) = 0$, for each clock ξ;
- for all $i \geq 0$, denoting $I_i = (a_i, b_i)$ and $\sigma'_i = \sigma_i + b_i - a_i$:
  (1) $(q_i, \delta_{i+1}, q_{i+1}, \gamma_{i+1}) \in \Delta$, $(\sigma'_i, v) \models \delta_{i+1}$ and $\sigma_{i+1} = \sigma'_i[\gamma_{i+1} := 0]$;
  (2) $\forall t \in I_i$, $\sigma_i + (t - a_i)$ and v satisfy $\beta(q_i)$.

The timed sequence associated with ρ is $(\lambda(q_0), I_0)(\lambda(q_1), I_1)(\lambda(q_2), I_2)\ldots$.

Recall that two timed sequences α' and α'' are equivalent if $\alpha'(t) = \alpha''(t)$ for all $t \geq 0$. With $L(\mathcal{A}, v)$ we denote the set of of all timed sequences over AP which are equivalent to those associated with a run of $\mathcal{A}$ under a parameter valuation v.

An interesting class of PTA is that of *lower bound/upper bound* (L/U) *automata* which are defined as PTA such that the set L of the parameters which can occur as a lower bound in a parametric clock constraint is disjoint from set U of the parameters which can occur as an upper bound. For L/U automata both acceptance criteria on finite runs and on infinite runs have been considered [13,5]. Here, we recall the Büchi acceptance condition. A *Büchi* L/U automaton $\mathcal{A}$ is an L/U automaton coupled with a subset F of locations. A run ρ is accepting for $\mathcal{A}$ if at least one location in F repeats infinitely often along ρ. We denote with $\Gamma(\mathcal{A})$ the set of parameter valuations v such that there exists an accepting run under v. We recall the following result.

Theorem 1 ([5]). *The problems of checking the emptiness and the universality of $\Gamma(\mathcal{A})$, for a Büchi L/U automaton $\mathcal{A}$, are* PSPACE*-complete.*

Decision problems. For the logic PMITL, we study the satisfiability and the model-checking problems, with respect to an L/U automaton. More precisely, given an PMITL formula φ and an L/U automaton $\mathcal{A}$, we consider the emptiness and universality of:

– the set $S(\varphi)$ of all the parameter valuations $v \in \mathcal{D}(\varphi)$ such that there is a timed sequence that satisfies φ under valuation v;
– the set $S(\mathcal{A}, \varphi)$ of all the parameter valuations $v \in \mathcal{D}(\varphi)$ such that there is a timed sequence in $L(\mathcal{A}, v)$ that satisfies φ under the valuation v.

Observe that we have defined PMITL by imposing some restrictions on the parameters. First, we require the sets of parameters L and U to be disjoint. Second, we force each interval to have at most one parameter, either in the left or in the right end-point. Third, we define admissibility for parameter valuations such that a parameterized interval cannot be evaluated neither as an empty nor a singular set. In section 5, we briefly discuss the impact of the first two restrictions on the decidability of the considered problems. In particular, we show that relaxing any of the first two restrictions leads to undecidability. Concerning to the notion of admissibility of parameter valuations, the restriction to non-empty intervals seems reasonable since evaluating an interval to an empty set would cancel a portion of our specification (which will remain unchecked). The restriction to non-singular intervals is instead needed for achieving decidability (see [3]).

Example of application of PMITL. As an example of the use of the logic PMITL we consider a model of the SPSMALL memory, a commercial product of STMicroeletronics, from [9].

The memory is modeled as the synchronous product of the timed automata corresponding to the input signals and the internal components of the memory, such as latches, wires and logical blocks. In Figure 1, we recall the model for a wire. Observe that differently from [9] our model is a PTA where the system constants $x^{\uparrow}$, $x^{\downarrow}$, $y^{\uparrow}$, and $y^{\downarrow}$ are parameters. In particular, $x^{\uparrow}$ and $x^{\downarrow}$ are upward parameters (i.e., belong to U) and $y^{\uparrow}$ and $y^{\downarrow}$ are downward parameters (i.e., belong to L). The behavior of the wire is related

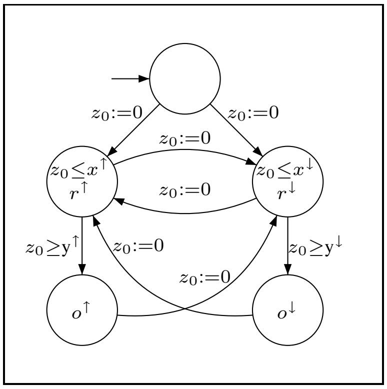


Fig. 1. A PTA model for the wire component

to the intervals $[y^{\uparrow}, x^{\uparrow}]$ and $[y^{\downarrow}, x^{\downarrow}]$. The first interval represents the delay interval for the component traversal when the input signal is rising (similarly for the second interval w.r.t. the falling signal). The model has one clock variable z_0 and five locations. The symbol $r^{\uparrow}$ (resp. $r^{\downarrow}$) labels the location which is entered when the input signal r is rising (resp. falling), and similarly $o^{\uparrow}$ (resp. $o^{\downarrow}$) for the output signal o. Each edge corresponds to a discrete event in the system. The locations are associated with parametric clock constraints modeling the desired behaviour.

When the data-sheet of a circuit does not provide enough information we can use parameters to formulate the wished system requirements and then perform a parameterized analysis of the circuit. As sample formulas, consider $\varphi_1 = \Box(r^{\uparrow} \Rightarrow \Diamond_{[c,d+x]}\, o^{\uparrow})$ and $\varphi_2 = \Box(r^{\downarrow} \Rightarrow \Diamond_{[c+y,d]}\, o^{\downarrow})$, where $c, d \in \mathbb{N}$, $x \in U$ and $y \in L$. These are variations with parameters of the standard time response property. In particular, φ_1 asserts that "every rising edge of the input signal r is followed by a rising edge of the output signal o within the interval $[c, d + x]$". In the first formula, we fix a constant lower bound on the response property and restrict the upper bound to be at least d. In the second formula instead, we fix an upper bound d and require the lower bound to be at least c.

3 Decidability of PMITL

In this section, we prove that the satisfiability and model-checking problems stated in the previous section are decidable and EXPSPACE-complete thus matching the computational complexity for MITL formulas [3]. A central step in our argument is a translation to the emptiness and the universality problems for Büchi L/U automata.

We start showing a normal form for PMITL formulas. Let $Sub(\varphi)$ denote the number of subformulas of φ. The following lemma is central in our approach.

Lemma 2. *For every PMITL formula φ, there is an equivalent PMITL formula ψ using only $\Diamond$ and $\Box$ as parameterized temporal operators, and parameterized intervals of the form $(0, z)$ or $(c+z, d)$, for $c, d \in \mathbb{N}$ and $z \in U \cup L$. Moreover, $Sub(\psi) = O(Sub(\varphi))$.*

Proof. First, we show that the parameterized $\Diamond_H$ and $\Box_J$, along with MITL operators, are sufficient to define all parameterized operators.

Consider first a parametric interval $(c, d + z)$. We have the equivalences:
$\varphi\, \mathcal{U}_{(c,d+z)}\, \psi \equiv (\varphi\, \mathcal{U}_I\, \psi) \wedge \Diamond_{(c,d+z)}\, \psi$, and $\varphi\, \mathcal{R}_{(c,d+z)}\, \psi \equiv (\varphi\, \mathcal{R}_I\, \psi) \vee \Box_{(c,d+z)}\, \psi$, where $I = (c, \infty[$ and I is left-closed iff $(c, d + z)$ is left-closed.

Now consider a parametric interval $(c + z, d)$. We have the equivalences:
$\varphi\, \mathcal{U}_{(c+z,d)}\, \psi \equiv \Box_{[0,z]}(\varphi \wedge \varphi\, \mathcal{U}_I\, \psi) \wedge \Diamond_{(c+z,d)}\, \psi$, and $\varphi\, \mathcal{R}_{(c+z,d)}\, \psi \equiv \Diamond_{[0,z]}(\varphi \vee \varphi\, \mathcal{R}_I\, \psi) \vee \Box_{(c+z,d)}\, \psi$, where $I = (c, \infty[$ and I is left-closed iff $(c, d + z)$ is left-closed.

We can further transform formulas such that the parametric intervals of the form $(c, d + z)$ are replaced with parameter intervals of the form $[0, z)$. For closed intervals, we use the following formulas: $\Diamond_{[c,d+x]}\, \varphi \equiv \Diamond_{[c,d]}(\varphi \vee \Diamond_{[0,x]}\, \varphi)$ and $\Box_{[c,d+y]}\, \varphi \equiv \Box_{[c,d]}(\varphi \wedge \Box_{[0,y]}\, \varphi)$. The equivalences for the remaining cases can be obtained similarly and thus we omit them.

Therefore, by applying the above described equivalences we can rewrite a PMITL formula φ into an equivalent formula ψ such that $Sub(\psi) \leq 3(Sub(\varphi))$ and the lemma holds. $\square$

Observe that the transformations used in [3] to reduce MITL formulas in normal form when applied to PMITL formulas do not alter the polarity of parameters. Therefore, from Lemma 2 and by adapting such transformations from [3], we have the following lemma:

Lemma 3. *Given a PMITL formula φ, there is an equivalent PMITL formula ψ whose temporal subformulas are of one of the following types:*

1. $\Diamond_{(0,b)}\,\varphi'$, *or* $\Box_{(0,b)}\,\varphi'$ *where* $b \in \mathbb{N}$*;*
2. $\varphi_1\,\mathcal{U}_{(a,b)}\,\varphi_2$, *or* $\varphi_1\,\mathcal{R}_{(a,b)}\,\varphi_2$, *where* $a > 0$ *and* $b \in \mathbb{N}$*;*
3. $\varphi_1\,\mathcal{U}_{[0,\infty[}\,\varphi_2$*;*
4. $\Box_{[0,\infty[}\,\varphi'$*;*
5. $\Diamond_{[0,x)}\,\varphi'$, *or* $\Box_{(a+x,b)}$, *where* $x \in U$*;*
6. $\Box_{[0,y)}\,\varphi'$, *or* $\Diamond_{(a+y,b)}$, *where* $y \in L$*.*

Moreover, $Sub(\psi) = O(Sub(\varphi))$*.*

If we restrict to PMITL formulas which do not contain parametric intervals of the form $(c+z,d)$, by Lemma 3 we are able to repeat the constructions given in [3] for MITL and in [5] for $\mathrm{P}_{0,\infty}\mathrm{MITL}_{0,\infty}$ to obtain equivalent PTAs. This result is precisely stated in the following theorem, where given a formula φ we denote with K_φ the maximal constant used in φ augmented by 1 and with N_φ the number of subformulas of φ.

Theorem 2. *Given a PMITL formula φ which does not contain parametric intervals of the form $(c+z,d)$, one can construct a Büchi L/U automaton $\mathcal{A}_\varphi$ such that, $\mathcal{A}_\varphi$ accepts a timed sequence α under a parameter valuation v if and only if $(\alpha, v) \models \varphi$. Moreover, $\mathcal{A}_\varphi$ has $O(2^{N_\varphi \times K_\varphi})$ locations, $O(N_\varphi \times K_\varphi)$ clocks, and constants bounded by K_φ.*

Note that, the L/U automaton $\mathcal{A}_\varphi$ from the above theorem uses exactly the parameters of the formula φ such that each parameter of φ from L is a lower bound parameter for $\mathcal{A}_\varphi$ and each parameter of φ from U is an upper bound parameter for $\mathcal{A}_\varphi$. We can now show the main theorem of this section.

Theorem 3. *Given a PMITL formula φ and an L/U automaton $\mathcal{A}$, checking the emptiness and the universality of the sets $S(\varphi)$ and $S(\mathcal{A},\varphi)$ is* EXPSPACE*-complete.*

Proof. Hardness follows from EXPSPACE-hardness of both satisfiability and model-checking problems for MITL [3].

To show membership to EXPSPACE of the emptiness problem for $S(\varphi)$ we use the following algorithm. First, assign each parameter x appearing in a subformula of φ of the form $\Diamond_{(c+x,d)}\,\psi$ with the minimum value assigned by an admissible parameter valuation, and each parameter y in a subformula of φ' of the form $\Box_{(c'+y,d')}\,\psi'$ with the maximum value assigned by an admissible parameter valuation. Note that these values are well defined since from the considered intervals we get the admissibility constraints $0 \le x \le d - c - 1$ and $0 \le y \le d' - c' - 1$, and thus the admissible values for both kinds of parameters are bounded. Now, denote with φ' the resulting formula after these assignments, construct the Büchi L/U automaton $\mathcal{A}_{\varphi'}$ as in Theorem 2 and then check $\Gamma(\mathcal{A}_{\varphi'})$ for emptiness.

Since $x \in L$ and $y \in U$, by Lemma 1 we get that $S(\varphi)$ is empty iff $S(\varphi')$ is empty. Thus, by Theorems 1 and 2, and since the number of subformulas of φ' is at most that of φ, we get that the above algorithm correctly checks $S(\varphi)$ for emptiness and takes exponential space.

To show membership to EXPSPACE of the universality problem for $S(\varphi)$ we can reason analogously, we only need to switch the role of the parameters in L and U in the first step of the algorithm. Membership to EXPSPACE of deciding emptiness and universality of $S(\varphi, \mathcal{A})$ can be shown with similar arguments, we just need to change the last step of the above algorithm to check the desired property of the set Γ for the intersection of $\mathcal{A}$ and $\mathcal{A}_{\varphi'}$, and not just for $\mathcal{A}_{\varphi'}$. $\square$

4 Computational Complexity in Fragments of PMITL

In this section, we address the complexity of the parameterized operators in PMITL, and thus of the corresponding logic fragments. Since MITL is already EXPSPACE-hard, we focus only on fragments of PMITL$_{0,\infty}$. We start with some more hardness results and then show PSPACE membership for some decision problems in the considered fragments.

Lemma 4. *Let φ be a PMITL$_{0,\infty}$ formula and $\mathcal{A}$ be an L/U automaton.*
If we restrict to formulas φ where all the parameterized operators are of the form $\Diamond_{(c,d+x)}$, then deciding the universality of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ is EXPSPACE-*hard.*

If we restrict to formulas φ where all the parameterized operators are of the form $\Box_{(c,d+x)}$, then deciding the emptiness of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ is EXPSPACE-*hard.*

If all the parameterized operators of φ are either of the form $\Box_{(c+x,d)}$ or of the form $\Diamond_{(c+x,d)}$, then deciding the emptiness and the universality of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ is EXPSPACE-*hard.*

Proof. We briefly sketch the proofs for $S(\varphi)$. The proofs for $S(\mathcal{A}, \varphi)$ can be obtained reducing the corresponding results for $S(\varphi)$, similarly to what is done for reducing satisfiability to model checking in temporal logics (see [11]).

We start showing EXPSPACE-hardness of the universality of $S(\varphi)$ when only parameterized operators of the form $\Diamond_{(c,d+x)}$ are allowed. We reduce the satisfiability problem of the fragment of MITL where, except for the operator $\Diamond$, one of the end-points of the intervals used as subscripts in the formulas is either 0 or ∞. Take any such formula φ, fix a parameter $x \in U$ and rewrite φ to a formula φ' where each operator of the form $\Diamond_{(c,d)}$ is rewritten to $\Diamond_{(c,d+x)}$ and any other part of the formula stays unchanged. We claim that φ is satisfiable if and only if $S(\varphi')$ is universal. To see this first observe that all the parameter valuations are admissible, and therefore, "$S(\varphi')$ is universal" means that $S(\varphi') = \mathbb{N}$. Thus, if $S(\varphi')$ is universal then $0 \in S(\varphi')$, and hence φ is satisfiable. Vice-versa, if φ is satisfiable then by Lemma 1, we get that $S(\varphi')$ is universal. Therefore, since the satisfiability for the considered fragment of MITL is EXPSPACE-hard, we get EXPSPACE-hardness for checking the universality of $S(\varphi)$.

To show EXPSPACE-hardness of the emptiness problem of $S(\varphi)$ when only parameterized operators of the form $\Box_{(c,d+x)}$ are allowed, we reason similarly. We reduce now the satisfiability problem of the fragment of MITL where, except for the operator $\Box$, one of the end-points of the intervals used as subscripts in the formulas is either 0 or ∞. For any such formula φ, we fix a parameter $y \in L$ and rewrite φ to a formula φ' where each operator of the form $\Box_{(c,d)}$ is rewritten to $\Box_{(c,d+y)}$ and any other part of the formula stays unchanged. Thus, if φ is satisfiable then trivially $S(\varphi')$ is not empty (it contains at least 0). Vice-versa, if $S(\varphi')$ is not empty, then by Lemma 1, 0 must belong to $S(\varphi')$ and therefore φ is satisfiable. Therefore, the claimed result follows from the fact that satisfiability for the considered fragment of MITL is EXPSPACE-hard.

For the fragments of $\mathrm{PMITL}_{0,\infty}$ where the only parameterized temporal operator are either of the form $\Box_{(c+x,d)}$ or of the form $\Diamond_{(c+x,d)}$, the proofs can be obtained similarly, and thus we omit further details. □

From the above lemma and Theorem 3, the following theorem holds.

Theorem 4. *Given a $\mathrm{PMITL}_{0,\infty}$ formula φ and an L/U automaton $\mathcal{A}$, checking the emptiness and the universality of each of the sets $S(\varphi)$, and $S(\mathcal{A}, \varphi)$ is* EXPSPACE-*complete.*

The results from Lemma 4 leave open the computational complexity for some of the considered decision problems.

Denote with $\mathrm{PMITL}_{\Diamond}$ the fragment of $\mathrm{PMITL}_{0,\infty}$ where the only parameterized operators are either of the form $\Diamond_{(c,d+x)}$ or one of the interval end-points is 0 or ∞, and with $\mathrm{PMITL}_{\Diamond}$ the fragment of $\mathrm{PMITL}_{0,\infty}$ where the only parameterized operators are either of the form $\Box_{(c,d+x)}$ or one of the interval end-points is 0 or ∞. (Observe that both fragments syntactically include $\mathrm{P}_{0,\infty}\mathrm{MITL}_{0,\infty}$.) In the rest of this section, we show that for such fragments some of the considered decision problems are PSPACE-complete. This is an interesting result, since they capture meaningful properties (see the example from Section 2).

For a sequence α, we denote with $S_\alpha(\varphi)$ the set of parameter valuations v such that the sequence α satisfies φ under valuation v. In the next lemma, without loss of generality we assume that for each parametric interval $(c, d+x)$, $c \leq d$ holds. In fact, if $c > d$, we can substitute the interval with $(c, c+x')$ for a fresh parameter x' such that $x' = x + d - c$. The results which are obtained on the resulting formula can thus be translated back to the original formula by reversing the linear transformation. The next lemma is the crucial result in our reduction.

Lemma 5. *Let $I = (c, d+x)$ where $c > 0$ and x is a parameter. The following holds:*

(a) For a $\mathrm{PMITL}_{\Diamond}$ formula $\varphi = \Diamond_I \psi$, define $\varphi' = \Box_{]0,c]} \Diamond_{I-c} \psi$. Then, $S_\alpha(\varphi') \subseteq S_\alpha(\varphi)$, and $S_\alpha(\varphi) = \emptyset \iff S_\alpha(\varphi') = \emptyset$.

(b) For a $\mathrm{PMITL}_{\Box}$ formula $\varphi = \Box_I \psi$, define $\varphi' = \Diamond_{]0,c]} \Box_{I-c} \psi$. Then, $S_\alpha(\varphi) \subseteq S_\alpha(\varphi')$. Moreover, if $(\alpha, v) \models \varphi'$ and $v(x) > c$, then $(\alpha, v') \models \varphi$ where $v'(x) = v(x) - c$ and $v'(y) = v(y)$ for each $y \neq x$.

Given a formula φ of $\mathrm{PMITL}_{\Diamond}$, by iteratively applying the transformation given in part (a) of the above lemma, from the innermost subformulas of the form $\Diamond_I \psi$ out, we obtain a formula φ' of $\mathrm{P}_{0,\infty}\mathrm{MITL}_{0,\infty}$ such that $S(\varphi) = \emptyset$ iff $S(\varphi') = \emptyset$. Since emptiness of $S(\varphi')$ can be decided in polynomial space ([5]) and size of φ' is $O(|\varphi|)$, we get that checking emptiness of $S(\varphi)$ is in PSPACE. We can repeat the same arguments for showing PSPACE-membership for checking the emptiness of $S(\mathcal{A}, \varphi)$ for a given PTA $\mathcal{A}$. Similarly, we use part (b) of the above lemma to show PSPACE-membership of checking the universality of $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ in PMITL$_\Box$. Therefore, we have the following theorem.

Theorem 5. *Given a formula φ in PMITL$_\Diamond$, and an L/U automaton $\mathcal{A}$, then checking the emptiness of the sets $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ is* PSPACE*-complete.*

Given a formula φ in PMITL$_\Box$, and an L/U automaton $\mathcal{A}$, then checking the universality of the sets $S(\varphi)$ and $S(\mathcal{A}, \varphi)$ is PSPACE*-complete.*

5 Parameterization of Time Intervals

The need for restricting the use of each parameter with temporal operators of the same polarity has been already addressed in [2,13,5] for parametric temporal logics and parametric timed automata. The argument used there also apply to PMITL and therefore we omit further discussion on this aspect. In this section, we relax the restriction that at most one of the end-points of an interval is a parametric expression. We define three natural ways of adding parameters to both the end-points of the intervals. Unfortunately, none of the proposed parameterizations leads to a decidable logic.

For simplicity, in this section we consider only the satisfiability problem, that is the problem of checking the emptiness of the set $S(\varphi)$.

In the first parameterization we consider, parameterized time-shifts of intervals. More precisely, with $\mathcal{L}_1$ we denote the logic obtained by augmenting MITL with parameterized intervals of the form $(c+x, d+x)$, such that (c, d) is not singular. Observe, that operators with this kind of intervals do not have polarity.

Theorem 6. *The problem of checking the emptiness of $S(\varphi)$ for any φ in $\mathcal{L}_1$ is undecidable. In particular, this holds already for the fragment of $\mathcal{L}_1$ with a single parameter x and where all parametric intervals are of the form $(x, x + 1)$.*

Proof's sketch. The idea of the proof is to reduce the membership problem for Turing machines by encoding computations such that all configurations are encoded with the same number of cells and each cell is encoded in an open interval of width 1. Parameter x is used to guess the length of the configurations and thus relate the content of a cell in the current configuration with its content in the next configuration. The other aspects of this reduction are quite standard, and therefore, we omit further details. $\Box$

Consider now an extension of PMITL with parameterized intervals where both left end-points and right end-points are in $\mathcal{E}(L) \cup \mathcal{E}(U)$. More precisely, given $x \in U$ and $y \in L$, we consider *fully parameterized* intervals which can be of the form $(c+y, d+x)$, when used as subscripts of *until* operators, and of the form $(c + x, d + y)$, when used as subscripts of *release* operators. We denote this logic $\mathcal{L}_2$.

Theorem 7. *The problem of checking the emptiness of $S(\varphi)$ for any φ in $\mathcal{L}_2$ is undecidable.*

Proof. Consider the formula $\varphi = \Diamond_{[y,x+1]} \psi \wedge \Box_{[x+1,y+2]} true$. For an admissible parameter valuation v, it holds that $v(y) < v(x) + 1$ and $v(x) + 1 < v(y) + 2$ from which we obtain that $v(x) = v(y)$. Thus, φ is equivalent to the formula $\Diamond_{[z,z+1]} \psi$ of $\mathcal{L}_1$. Therefore, the theorem follows from Theorem 6. $\Box$

Another way of obtaining fully parametrization of the intervals is to use a parameter for translating the interval in time and the other to adjust the width of the interval. Let $\mathcal{L}_3$ denote the corresponding logic. We can show that this logic is also undecidable by using the following reduction. Given an interval $(c + y, d' + y + x)$, we obtain the interval $(c + y', d + x')$ by the linear transformation: $y' = y$, $x' = c + 1 + x + y' - d$, and $d' = c + 1$. Thus, from Theorem 7 we get:

Theorem 8. *The problem of checking the emptiness of $S(\varphi)$ for any φ in $\mathcal{L}_3$ is undecidable.*

References

1. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Inf. Comput. 104(1), 2–34 (1993)
2. Alur, R., Etessami, K., La Torre, S., Peled, D.: Parametric temporal logic for "model measuring". ACM Trans. Comput. Log. 2(3), 388–407 (2001)
3. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM 43(1), 116–146 (1996)
4. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: STOC, pp. 592–601 (1993)
5. Bozzelli, L., La Torre, S.: Decision problems for lower/upper bound parametric timed automata. Formal Methods in System Design 35(2), 121–151 (2009)
6. Bruyère, V., Dall'Olio, E., Raskin, J.F.: Durations and parametric model-checking in timed automata. ACM Trans. Comput. Log. 9(2) (2008)
7. Bruyère, V., Raskin, J.F.: Real-time model-checking: Parameters everywhere. Logical Methods in Computer Science 3(1) (2007)
8. Campos, S.V.A., Clarke, E.M., Grumberg, O.: Selective quantitative analysis and interval model checking: Verifying different facets of a system. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 257–268. Springer, Heidelberg (1996)
9. Chevallier, R., Encrenaz-Tiphène, E., Fribourg, L., Xu, W.: Timed verification of the generic architecture of a memory circuit using parametric timed automata. Formal Methods in System Design 34(1), 59–81 (2009)
10. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Formal Methods in System Design 1(4), 385–415 (1992)
11. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science. Formal Models and Sematics (B), vol. B, pp. 995–1072 (1990)
12. Emerson, E.A., Trefler, R.J.: Parametric quantitative temporal reasoning. LICS, 336–343 (1999)
13. Hune, T., Romijn, J., Stoelinga, M., Vaandrager, F.W.: Linear parametric model checking of timed automata. J. Log. Algebr. Program. 52-53, 183–220 (2002)
14. Kupferman, O., Piterman, N., Vardi, M.Y.: From liveness to promptness. Formal Methods in System Design 34(2), 83–103 (2009)
15. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE, Los Alamitos (1977)
16. Wang, F.: Parametric timing analysis for real-time systems. Inf. Comput. 130(2), 131–150 (1996)

Short Witnesses and Accepting Lassos in ω-Automata[⋆]

Rüdiger Ehlers

Reactive Systems Group,
Saarland University
ehlers@react.cs.uni-sb.de

Abstract. Emptiness checking of ω-automata is a fundamental part of the automata-theoretic toolbox and is frequently applied in many applications, most notably verification of reactive systems. In this particular application, the capability to extract accepted words or alternatively accepting runs in case of non-emptiness is particularly useful, as these have a diagnostic value. However, non-optimised such words or runs can become huge, limiting their usability in practice, thus solutions with a small representation should be preferred. In this paper, we review the known results on obtaining these and complete the complexity landscape for all commonly used automaton types. We also prove upper and lower bounds on the approximation hardness of these problems.

1 Introduction

In the last decades, model checking has emerged as an increasingly successful approach to the verification of complex systems [20,1]. This development is witnessed by the existence of a significant number of industrial-scale model checkers and successful experiments on integrating the usage of model checkers into the development cycle of industrial products [12,17,22]. Compared to deductive verification approaches, model checking has the advantage of being a push-button technology: the designer of a system only has to state the desired properties and (a model of) the system implementation, but the proof of correctness/incorrectness of the system is done automatically. In case of an error in the implementation, the model checker usually constructs an example run of the system in which this error occurs, which in turn is useful for the system designer to correct the system. It has been observed that this makes model checking particularly useful in the early development stages of a complex system [17,13], as the automatic generation of such *counter-examples* saves valuable time.

Finding good counter-examples in model checking is however a non-trivial task as the question which of the often infinitely many counter-examples is most

⋆ This work was supported by the German Research Foundation (DFG) within the program "Performance Guarantees for Computer Systems" and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

useful for the designer heavily depends on the particular problem instance [13,11]. Consequently, the length of a counter-example is the predominant quality metric that researchers in this area have agreed on [9].

As an example, in the context of model checking finite state machines against properties written in *linear time temporal logic* (LTL), the specification is usually negated and transformed into an equivalent non-deterministic Büchi automaton. Finding a witness for the non-satisfaction of the specification then amounts to finding an *accepting lasso* in the product of the finite state machine and the Büchi automaton. In this context, shorter lassos are preferred as they simplify analysing the cause of the problem. Nowadays, efficient polynomial algorithms for finding a shortest accepting lasso in such a setting exist [19,11], allowing for the extraction of such shortest lassos. On the other hand, for systems that obey some *fairness* constraints, the problem of finding short counter-examples reduces to finding short accepting lassos in *generalized Büchi automata*. For this case, it is known that finding a shortest accepting lasso is NP-complete [4].

Independently of these complexity considerations, it has been argued that for debugging models, a shortest accepting lasso is not what the designer of a system is usually interested in [16]. In fact, some input to the system of the form uv^ω (for u and v being finite sequences) such that $|u| + |v|$ is minimal is likely to be more helpful for debugging as such a representation is independent of the actual automaton-encoding of the violated property. For this modified setting, Kupferman and Sheinfeld-Faragy proved that also for ordinary model checking without fairness, finding shortest such counter-examples (called witnesses in this case) is NP-complete, rendering the problem difficult.

From a more high-level view of these results, the existence of efficient algorithms for some of the cases just discussed on the one hand and the NP-completeness of the other cases leads to a natural question: where exactly is the borderline that separates the hard problems from the simple ones for finding short counter-examples in model checking? Furthermore, from a practical point of view, another question naturally arises: what is the approximation hardness of these problems? For example, while finding a shortest witness for the non-satisfaction of a specification might be NP-hard, finding a 2-approximate shortest witness might be doable in polynomial time. Obviously, such a result would have practical consequences. Nevertheless, to the best of our knowledge, this question has not been discussed in the literature yet.

In this paper, we give a thorough discussion of the complexity of finding short non-emptiness certificates for various types of ω-automata, which answers the question how hard obtaining short counter-examples in regular model checking (which reduces to Büchi automaton emptiness) and model checking under fairness (which reduces to generalized Büchi automaton emptiness) actually is. We discuss both types of certificates mentioned above: short accepting lassos and short witnesses. As finding short lassos and witnesses is also useful in other contexts in which automata-theoretic methods are applied, like *synthesis of closed systems* [3] or deciding the validity of formulas in logics such as S1S [2], we give a unified overview for all commonly used types of ω-automata, namely those with safety, Büchi, co-Büchi, parity, Rabin, Streett, generalized Büchi and Muller acceptance conditions. For all of these cases, we review the known complexity results for the exact minimization of the size of accepting lassos or witnesses and complete the complexity landscape for the cases not considered in the literature so far. This results in the first complete exposition of the borderline between the hard and simple problems in this context. We also examine the approximation hardness of the NP-complete problems of this landscape, which, from a practical point of view, is an important question to raise as approximate solutions often suffice for the good usability of a method in which finding short accepting lassos or witnesses is a sub-step. The results we obtain for the approximability of the problems considered are mostly negative: For example, we prove that approximating the minimal witness size within any polynomial is NP-complete even for the simple safety acceptance condition. We also give some positive results, e.g., a simple algorithm for approximating the minimal witness size within any (fixed) exponential function that runs in time polynomial in the number of states of some given ω-automaton. Table 1 contains a summary of the other results contained in this paper.

The structure of our presentation is as follows: In the next section, we state the preliminaries. Sections 3 and 4 contain the precise definitions of the problems of finding shortest accepting lassos and witnesses and present hardness results and algorithms for them. Section 5 concludes the findings and sketches the open problems.

2 Preliminaries

An ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ is a five-tuple consisting of some finite state set Q, some finite alphabet Σ, some initial state $q_0 \in Q$, some transition function $\delta : Q \times \Sigma \to 2^Q$ and some *acceptance component* $\mathcal{F}$ (to be defined later). We say that an automaton is deterministic if for every $q \in Q$ and $x \in \Sigma$, $|\delta(q, x)| \leq 1$.

Given an infinite word $w = w_1 w_2 \ldots \in \Sigma^\omega$ and an ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$, we say that some sequence $\pi = \pi_0 \pi_1 \ldots$ is a run for w if $\pi_0 = q_0$ and for all $i \in \{1, 2, \ldots\}$, $\pi_i \in \delta(\pi_{i-1}, w_i)$. We say that π is accepting if for $\inf(\pi) = \{q \in Q \mid \exists^\infty j \in \mathbb{N} : \pi_j = q\}$, $\inf(\pi)$ is accepted by $\mathcal{F}$. The acceptance of π by $\mathcal{A}$ is defined with respect to the type of $\mathcal{F}$, for which many have been proposed in the literature [10].

- For a *safety winning condition*, all infinite runs are accepting. In this case, the $\mathcal{F}$-symbol can also be omitted from the automaton definition.
- For a *Büchi acceptance condition* $\mathcal{F} \subseteq Q$, π is accepting if $\inf(\pi) \cap \mathcal{F} \neq \emptyset$.
- For a *co-Büchi acceptance condition* $\mathcal{F} \subseteq Q$, π is accepting if $\inf(\pi) \cap \mathcal{F} = \emptyset$.
- For a *generalized Büchi acceptance condition* $\mathcal{F} \subseteq 2^Q$, π is accepting if for all $F \in \mathcal{F}$, $\inf(\pi) \cap F \neq \emptyset$.
- For a *Rabin acceptance condition* $\mathcal{F} \subseteq 2^Q \times 2^Q$, π is accepting if for $\mathcal{F} = \{(F_1, G_1), \ldots, (F_n, G_n)\}$, there exists some $1 \leq i \leq n$ such that $\inf(\pi) \subseteq F_i$ and $\inf(\pi) \cap G_i \neq \emptyset$.

- For a *parity acceptance condition*, $\mathcal{F} : Q \rightarrow \mathbb{N}$ and π is accepting in the case that $\max\{\mathcal{F}(v) \mid v \in \inf(\pi)\}$ is even.
- For a *Streett acceptance condition* $\mathcal{F} \subseteq 2^Q \times 2^Q$, π is accepting if for $\mathcal{F} = \{(F_1, G_1), \ldots, (F_n, G_n)\}$ and for all $1 \leq i \leq n$, we have $\inf(\pi) \nsubseteq F_i$ or $\inf(\pi) \cap G_i = \emptyset$.
- For a *Muller acceptance condition* $\mathcal{F} \subseteq 2^Q$, π is accepting if $\inf(\pi) \in \mathcal{F}$.

The language of $\mathcal{A}$ is defined as the set of words for which there exists a run that is accepting with respect to the type of the acceptance condition. We also call automata with a t-type acceptance condition t-automata (for $t \in \{$safety, Büchi, co-Büchi, generalized Büchi, parity, Rabin, Streett, Muller$\}$). For all acceptance condition types stated above, $|\mathcal{F}|$ is defined as the cardinality of $\mathcal{F}$ (for safety automata we set $|\mathcal{F}| = 0$).[1] We define the size of $\mathcal{A}$, written as $|\mathcal{A}|$ to be $|Q| + |\Sigma| + |\delta| + |\mathcal{F}|$ for $|\delta| = |\{(q, q', e) \in Q \times Q \times \Sigma \mid q' \in \delta(q, x)\}|$.

We say that an algorithm approximates the minimal lasso/witness within some function $f(n)$ if for every problem instance with a shortest accepting lasso/witness having some size $n \in \mathbb{N}$, it always finds a solution of size not more than $f(n)$. An algorithm is said to approximate within a constant factor/within a polynomial if there exists some $c \in \mathbb{N}$/some polynomial function $p(n)$ such that it approximates within $f(n) = c \cdot n / f(n) = p(n)$, respectively. For the hardness and non-approximability results, we assume that P$\neq$NP (otherwise all problems discussed here are solvable in polynomial time).

An automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ can also be thought of as a graph $\langle V, E \rangle$ with vertices $V = Q$ and edges $E \subseteq V \times V$ such that for all $v_1, v_2 \in V$, $(v_1, v_2) \in E$ if there exists some $a \in \Sigma$ such that $v_2 \in \delta(v_1, a)$. A path in $\langle V, E \rangle$ going from v to v' is a sequence $\pi = v_1 \ldots v_n$ with $v_1 = v$ and $v_n = v'$ such that for all $i \in \{1, \ldots, n\}$, $v_i \in V$ and for all $i \in \{1, \ldots, n-1\}$, $(v_i, v_{i+1}) \in E$. A *strongly connected subset* of $\mathcal{A}$ is a set of states $Q' \subseteq Q$ such that there exist paths in $\langle Q', E|_{Q'} \rangle$ between all pairs of states in Q'.

For all acceptance condition types given above, the emptiness of the language of an automaton (i.e., whether there exists no accepted word) can be decided in time polynomial in the size of the automaton. For Rabin and Muller automata, this follows from standard automata constructions (see, e.g., the folk theorems in [18]). For Streett automata, this follows from the existence of efficient emptiness checking constructions [14]. Likewise, checking if a word uv^ω is accepted by some automaton $\mathcal{A}$ can also be performed in time polynomial in $|uv|$ and $|\mathcal{A}|$ for all of these acceptance condition types.

3 Finding Shortest Accepting Lassos

In this section, we deal with finding shortest accepting lassos in ω-regular automata. Given an ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$, we formally define lassos as pairs (l, l') such that:

[1] As in this paper, we are only interested in the borderline between NP-complete problems and those that are in P (assuming P$\neq$NP), we can safely ignore the fact that an explicit encoding of $\mathcal{F}$ might actually be slightly bigger.

- $l = l_0 \dots l_n \in Q^n$ for some $n \in \mathbb{N}_0$
- $l' = l'_0 \dots l'_{n'} \in Q^{n'}$ for some $n' \in \mathbb{N}_{>0}$
- $l_0 = q_0$, $l_n = l'_0 = l'_{n'}$
- For all $i \in \{0, \dots, n-1\}$, $\exists x \in \Sigma$ such that $\delta(l_i, x) = l_{i+1}$
- For all $i \in \{0, \dots, n'-1\}$, $\exists x \in \Sigma$ such that $\delta(l'_i, x) = l'_{i+1}$

The length of such a lasso is defined to be $n + n'$. Given a lasso (l, l'), we call l' the *lasso cycle* of (l, l').

3.1 The Rabin Acceptance Condition and Its Special Cases

First of all, we consider safety, Büchi, co-Büchi, parity and Rabin acceptance conditions and show that finding shortest accepting lassos for all of these acceptance condition types is doable in time (and thus, space) polynomial in the input size. Note that conversions from safety, Büchi, co-Büchi or parity acceptance components to equivalent Rabin acceptance components can easily be done with only polynomial blow-up (see, e.g., [7]).

For Büchi automata (and thus also safety automata as a special case), efficient algorithms for finding shortest accepting lassos are known, requiring roughly $O(|Q||\delta|)$ time (see [9,19,13] for entry points to the literature).

For the remaining cases, we show that finding shortest accepting lassos is solvable in polynomial time for Rabin automata, leading to the same result also for co-Büchi and parity automata. Without loss of generality, we can assume that a Rabin automaton has only one acceptance pair, i.e., $\mathcal{F} = \{(F, G)\}$ for some $F, G \subseteq Q$ as a word is accepted by a Rabin automaton $(Q, \Sigma, q_0, \delta, \mathcal{F})$ if and only if there exists an acceptance pair $(F, G) \in \mathcal{F}$ such that $\mathcal{A}' = (Q, \Sigma, q_0, \delta, \{(F, G)\})$ accepts the word. Therefore, by iterating over all elements in $\mathcal{F}$ and taking the shortest lasso found, we can extend a polynomial algorithm for a single acceptance pair to a polynomial algorithm for general Rabin automata.

Note that for a lasso (l, l') with $l' = l'_0 \dots l'_{n'}$ to be accepting for $(Q, \Sigma, q_0, \delta, \{(F, G)\})$, we must have $\{q \in Q \mid \exists i : l'_i = q\} \setminus F = \emptyset$. So, states in $Q \setminus F$ may not occur on the cycle-part of the lasso. For each state $q \in F$, we can apply one of the basic shortest-lasso algorithms for Büchi automata on $(F, \Sigma, q, \delta|_F, G)$ and compute a shortest accepting lasso in it. Let the lasso length for each starting state $q \in F$ be called $c(q)$.

For actually obtaining a shortest accepting lasso over $(Q, \Sigma, q_0, \delta, \{(F, G)\})$, we can apply a standard shortest-path algorithm by interpreting $\mathcal{A}$ as a graph, adding a goal vertex to it, adding edges from each state $q \in Q$ where $c(q)$ is defined to this goal vertex with cost $c(q)$, and taking q_0 as the starting vertex. The remaining transitions have cost 1. By taking the shortest path up to the point where an added edge is taken and then replacing it by the corresponding lasso computed in the previous step, we easily obtain a shortest accepting lasso for $(Q, \Sigma, q_0, \delta, \{(F, G)\})$.

The overall complexity of this procedure is clearly polynomial in $|\mathcal{A}|$.

3.2 Generalized Büchi and Streett Automata

Rabin automata and their special cases have a certain property: On every shortest accepting lasso, no state can occur twice. This property does not hold for generalized Büchi and Streett acceptance conditions. Intuitively, this can make finding short accepting lassos significantly harder as the corresponding search space is larger. Indeed, the length of a shortest accepting lasso cannot be approximated within any constant in polynomial time if P≠NP. We prove this fact by reducing the Ek-Vertex-Cover problem [15] onto finding short accepting lassos.

Problem 1. A k-uniform hypergraph is a 2-tuple $G = \langle V, E\rangle$ such that V is a finite set and $E \subseteq 2^Q$ such that all elements in E are of cardinality k. Given a k-uniform hypergraph $H = \langle V, E\rangle$, the Ek-Vertex-Cover problem is to find a subset $V' \subseteq V$ of minimal cardinality such that for all $e \in E$: $e \cap V' \neq \emptyset$. It has been proven that approximating the minimal size of such a subset within a factor of $(k - 1 - \epsilon)$ for some $\epsilon > 0$ is NP-hard [5].

Consider a k-uniform hypergraph $G = \langle V, E\rangle$ for some arbitrary $k \in \mathbb{N}$. We can easily reduce the problem of finding a small Ek-Vertex-Cover to finding short accepting lassos in a generalized Büchi automaton over a one-element alphabet $\Sigma = \{\cdot\}$. We define $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ with $Q = V$, $\delta(q, \cdot) = Q$ for all $q \in Q$ (so we have a complete graph) and $\mathcal{F} = E$. Furthermore q_0 is set to some arbitrary element of Q. Given some vertex cover $V' \subseteq V$, it is clear from the definition of $\mathcal{A}$ that for $V' = \{v_1, \ldots, v_m\}$, the lasso (l, l') with $l = q_0 v_1$ and $l' = v_1 v_2 \ldots v_m v_1$ is accepting. On the other hand, an accepting lasso (l, l') with $l = q_0 v_1$ and $l' = v_1 v_2 \ldots v_m v_1$ induces a vertex cover $V' \subseteq V$ with $V' = \{v_1, \ldots, v_m\}$. Therefore, this reduction preserves the quality of the solutions up to a possible deviation of 1 (for the initial state of the lasso).

As the Ek-Vertex-Cover problem is reducible to finding short lassos (up to a deviation of 1) and is NP-hard to approximate within a factor of $(k - 1 - \epsilon)$ for all $k \in \mathbb{N}$ and $\epsilon > 0$, we obtain the following result:

Theorem 2. *Approximating the length of a shortest accepting lasso in generalized Büchi automata is NP-hard within any constant factor.*

As generalized Büchi automata have a simple translation to Streett automata, the same result holds for Streett automata as well. Note that these problems are also in NP as verifying the validity of an accepting lasso is simple and the length of a shortest accepting lasso in $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ is bounded by $|Q|^2$.

Thus, NP-completeness of these problems follows. Note that this line of reasoning also holds for the Muller acceptance condition to be discussed next.

3.3 Muller Automata

For finding short accepting lassos in Muller automata, we can use the same scheme as for Rabin automata: Given a Muller automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ with $\mathcal{F} = \{F_1, \ldots, F_m\}$, we can search for short accepting lassos in each of the automata $(Q, \Sigma, q_0, \delta, F_1)$, …, $(Q, \Sigma, q_0, \delta, F_m)$ and take the shortest accepting lasso we find in these automata as a shortest lasso for $\mathcal{A}$. Thus, assuming that we have a $f(n)$-approximation algorithm for a Muller automaton with a single acceptance set running in polynomial time, this immediately gives rise to a polynomial $f(n)$-approximation algorithm for general Muller automata.

For a lasso (l, l') to be accepting for some Muller acceptance set F, all states in F must occur in l'. As we can furthermore assume that the states in $F \subseteq Q$ form a strongly connected subset in Q (as otherwise F cannot be precisely the set of states occurring infinitely often on a run), the problem of finding a short accepting lasso is related to the *asymmetric metric travelling salesman problem* (AMTSP), as we explain in the remainder of this section.

Problem 3. Given a set of cities C with $|C| = n$ and a distance function $d : C \times C \rightarrow \mathbb{N}_0$ such that $d(c, c) = 0$ for all $c \in C$ and for every $c_1, c_2, c_3 \in C$, $d(c_1, c_2) + d(c_2, c_3) \leq d(c_1, c_3)$, the AMTSP-problem is to find a cycle $c_0, \ldots, c_{n-1}$ such that the cost of the cycle (i.e., $\sum_{i=0}^{n-1} d(c_i, c_{(i+1) \bmod n})$) is as small as possible.

It has been proven that in a special case of the AMTSP problem in which the distance between two cities is either 1 or 2, the cost of the cheapest cycle cannot be approximated within a factor of $\frac{321}{320} - \epsilon$ for some $\epsilon > 0$ in polynomial time, unless P=NP [6]. A simple reduction shows that this is also the case for finding shortest accepting lassos in Muller automata:

Theorem 4. *Approximating the length of a shortest accepting lasso within a factor of $\frac{321}{320} - \epsilon$ for some $\epsilon > 0$ in a Muller automaton is NP-hard.*

Proof. Given a AMTSP-Problem $\langle C, d \rangle$ in which the distance between two different cities is always 1 or 2, we reduce finding the length of a shortest cycle to finding the shortest accepting lasso in a Muller automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ over $\Sigma = \{\cdot\}$ with $\mathcal{F} = \{F_1, F_2\}$ by defining $Q = C \uplus \{\Gamma\}$, $\delta(c, \cdot) = \{c' \in C \mid d(c, c') = 1\} \cup \{\Gamma\}$ (for all $c \in C$), $\delta(\Gamma, \cdot) = C$, $F_1 = C$, $F_2 = Q$ and set $q_0 = c$ for some arbitrary $c \in C$.

For every cycle of length j for some $j \in \mathbb{N}$, there exists an accepting lasso of the same length starting with q_0. Whenever an edge with cost 2 is taken, the lasso is routed through Γ, the other edges can be taken directly.

On the other hand, each lasso cycle in $\mathcal{A}$ induces a cycle in $\langle C, d \rangle$ with a cost equal to the length of the lasso by skipping over all visits to Γ. Without loss of generality, we can assume that such an accepting lasso (l, l') has $l = q_0$.

As this way, the cost of the cycle and the lasso length coincide and approximating the cost of a shortest cycle in $\langle C, d \rangle$ within $\frac{321}{320} - \epsilon$ is NP-complete for all $\epsilon > 0$, the claim follows.

Thus, also in the Muller automaton case, we cannot approximate the size of a shortest accepting lasso arbitrarily well. However, the close connection between the AMTSP problem and Muller automaton emptiness allows us to make use of a positive approximation result for the AMTSP problem:

Theorem 5. *Given a Muller automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ with $\mathcal{F} = \{F_1\}$, we can compute a lasso of length not more than $\ulcorner \log_2 |F_1| \urcorner$ times the length of a shortest one in polynomial time.*

Proof. The problem can be solved using a $\ulcorner \log_2 |n| \urcorner$-approximation algorithm for the AMTSP problem [8]. We construct an AMTSP instance $\langle C, d \rangle$ by taking $C = F_1$ and for each pair of cities $c_1, c_2 \in C$ with $c_1 \neq c_2$, we use a standard shortest-path finding algorithm for computing $d(c_1, c_2)$, i.e., the length of the shortest path through the graph of $\mathcal{A}$ restricted to F_1 from state c_1 to c_2. For every computed value, we store the corresponding path for later retrieval. Then, we apply the approximation algorithm on $\langle C, d \rangle$ and obtain a tour of length at most $\ulcorner \log_2 |F_1| \urcorner \cdot m$, where m is the length of the optimal tour. As we can assume that F_1 is a strongly connected subset in $\mathcal{A}$, taking the tour and stitching together the individual respective parts we stored in the previous step results in a lasso cycle with a length equal to the cost of the tour. By finding a shortest path in $\mathcal{A}$ from q_0 to one of the states in F_1 and adding this path as first part of the lasso, we obtain a complete accepting lasso. The approximation quality of the solution follows directly from the definition of $\langle C, d \rangle$ and the fact that the first part of the lasso is indeed as short as possible as all elements in F_1 have to occur on the cycle.

4 Finding Shortest Witnesses

In this section, we consider finding shortest witnesses, i.e., given some ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$, the task is to find a word uv^ω for $u, v \in \Sigma^*$ that is accepted by $\mathcal{A}$ with $|u| + |v|$ being as small as possible. We show that approximating the length of a shortest such word within any polynomial is NP-complete for all acceptance condition types considered in this paper, but we can approximate this length within any exponential function in polynomial time (for every fixed alphabet Σ). We start with the hardness result.

Theorem 6. *Given some polynomial function p, approximating the length of a minimal witness in some safety-type ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ within p over a ternary alphabet $\Sigma = \{0, 1, \#\}$ is NP-hard.*

Proof. The proof is based on a reduction from the satisfiability (SAT) problem, which is known to be NP-hard (see, e.g., [21] for details).

We define a conjunctive normal form SAT-instance to consist of a set of variables $V = \{v_1, \ldots, v_m\}$ and a set of clauses $C = \{c_1, \ldots, c_n\}$ (with $c_i : V \times \{0,1\} \to \mathbb{B}$ for all $1 \leq i \leq n$) which are formally functions such that $c_i(v_k, 1) = \mathbf{true}$ if and only if v_k is a literal in clause i and $c_i(v_k, 0) = \mathbf{true}$ if $\neg v_k$ is a literal in clause i (for all $1 \leq k \leq m$, $1 \leq i \leq n$).

We reduce the problem of determining whether there exists some valuation of the variables that satisfies all clauses in C to finding some short witness in some safety automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta)$ over $\Sigma = \{0, 1, \#\}$ as follows:

- $Q = \{(i, j, k, b) \in \mathbb{N}^3 \times \mathbb{B} \mid 1 \leq i \leq n, 1 \leq j \leq p(m), 1 \leq k \leq m+1, b \Rightarrow k > 1\} \cup \{\perp\}$
- $q_0 = (1, 1, 1, \mathbf{false})$

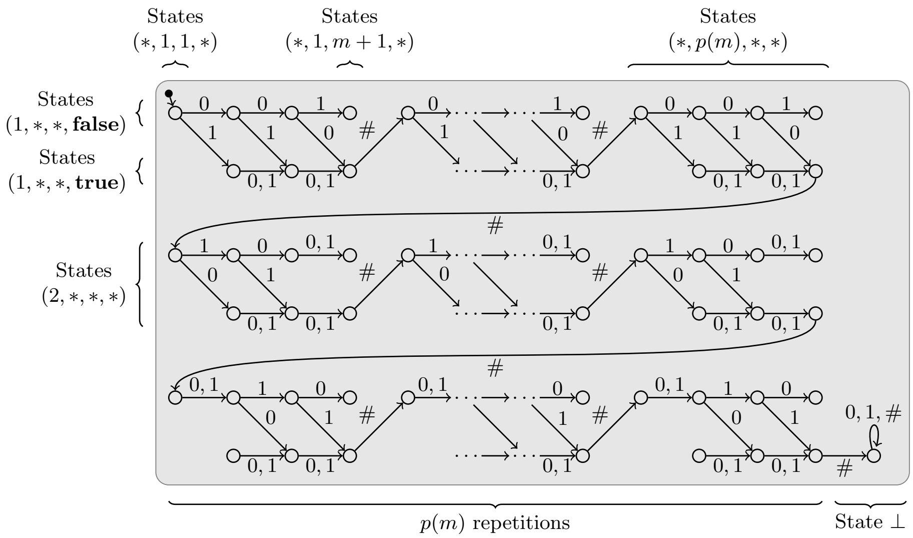


Fig. 1. Example automaton constructed from the SAT-instance $(v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_2 \vee v_3)$ as described in the proof of Theorem 6. In this example, we have $m = 3$ and $n = 3$ with $V = \{v_1, v_2, v_3\}$. The labels next to the braces explain the structure of the automaton generated (with $*$ denoting that the states corresponding to any suitable value at this point in the tuple are contained in the state set).

- For all $(i,j,k,b) \in Q$, $a \in \{0,1,\#\}$, $\delta((i,j,k,b),a)$ is the union of:
  - $\{(i,j,k+1,b') \mid k \leq m, b' = (b \vee c_i(v_k,a))\}$
  - $\{(i,j+1,1,\mathbf{false}) \mid b = \mathbf{true}, a = \#, j \leq p(m), k = m+1\}$
  - $\{(i+1,1,1,\mathbf{false}) \mid b = \mathbf{true}, j = p(m), k = m+1, a = \#, i \leq n\}$
  - $\{\perp\}$ if $b = \mathbf{true}$, $j = p(m)$, $k = m+1$, $a = \#$, b and $i = n$
- $\delta(\perp, a) = \{\perp\}$ for all $a \in \{0,1,\#\}$

Figure 1 gives an example of such an automaton for an example SAT-instance.

The key idea of this reduction is the following: The automaton built only accepts input words on which during the first $p(m)(m+1)n$ input letters, precisely every $(m+1)$th letter is a #. Furthermore, the letters in between represent valuations to the variables in the SAT instance. During the first $p(m)(m+1)$ input letters, it is checked that the solution given satisfies the first clause. Subsequent parts of the input words are then checked against the next clause (and so on). Now assume that a word uv^ω for which $|u|+|v| \leq p(m)(m+1)$ holds is accepted by the automaton. All parts in between two occurrences of # in the word represent variable valuations satisfying all clauses. On the other hand, if there exists some valuation for the variables satisfying all clauses, then there exists a simple word with $|u| = 0$ and $|v| = m+1$ such that uv^ω is accepted. Therefore, by using a p-approximation algorithm for finding the length of a shortest accepting witness, we can check if there exists a valuation of V satisfying C.

This non-approximability result for finding (or even determining the minimal size of) short witnesses is surprising. While finding short accepting lassos is doable in polynomial time even for the more complex Rabin condition, approximating the size of a shortest witness is NP-hard even for safety automata and thus considerably harder. For the other acceptance condition types, the same result holds as only the state $\perp$ can be visited infinitely often on any accepting run. It is trivial to build corresponding acceptance components for any of the other acceptance condition types defined in this paper. The hardness proof given above also holds for a binary alphabet with only a slight modification.

As in the case of finding short accepting lassos, the fact that the problem of finding a shortest witness is actually contained in NP is easy to show: for all automaton types considered, the problem of checking whether a word uv^ω is in the language of the automaton is solvable in polynomial time. Furthermore, if the language of the automaton is non-empty, then there exists some witness of length not more than the square of the automaton's number of states. By taking together these facts, membership in NP trivially follows.

A natural question to ask at this point is which positive statements about the approximability of this problem can be given. In this paper, we show the following:

Theorem 7. *Let $c > 1$ and Σ be some fixed finite alphabet. Given some ω-automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, \mathcal{F})$ with any of the acceptance types considered in this paper, computing a word uv^ω such that $|u| + |v|$ is not longer than c^n for n being the minimal witness length can be done in time polynomial in $|\mathcal{A}|$.*

Proof. Note that for all acceptance types considered in this paper, checking whether a word uv^ω is accepted by $\mathcal{A}$ is possible in time polynomial in $|u| + |v|$ and $|\mathcal{A}|$.

Furthermore, for all acceptance condition types, emptiness checking and the extraction of an accepting lasso of size no longer than $|Q|^2$ can be performed in polynomial time. Therefore, we can iterate over all words uv^ω such that $|uv| \leq \ulcorner \log_c |Q|^2 \urcorner$ (which are only polynomially many) and check for each of them whether they are in the language of the automaton. A c^n-approximation algorithm can thus return the shortest such witness, if found. In all other cases,

Table 1. Summary of the approximability results on finding short accepting lassos and witnesses for the acceptance condition types considered in this paper. In all cases, it is assumed that P$\neq$NP and only algorithms running in time polynomial in the size of the input are considered.

Acceptance cond. type	**Shortest accepting lassos**	**Shortest witnesses**
Safety, Büchi, co-Büchi, parity, Rabin	solvable precisely in polynomial time	not approximable within any polynomial, approximable within every exponential function for a fixed alphabet
Generalized Büchi, Streett	not approximable within any constant, approximable within every exponential function for a fixed alphabet	
Muller	not approximable within $\frac{321}{320} - \epsilon$, approximable within $\ulcorner \log_2 \|Q\| \urcorner$	

the simple accepting lasso of size not more than $|Q|^2$ can be converted to an accepting word by copying the edge labels. The fact that exponentially shorter words would have been found by the first step suffices for proving the approximation quality of this algorithm.

Taking the results obtained in this section together, we obtain a quite precise characterisation of the approximation hardness of finding short witnesses in ω-automata: Approximating the size within any polynomial is NP-complete, but the problem is approximable within any exponential function in polynomial time for every fixed alphabet.

As a final note, the exponential-quality approximation algorithm presented in this section is also useful for finding short accepting lassos. Therefore, we obtain the same upper bound on the approximation hardness of that problem.

5 Conclusion

In this paper, we have examined the problem of finding short accepting lassos and witnesses for ω-automata of various acceptance condition types. We bounded the borderline between NP-complete approximation problems and those in P from both above and below (assuming that P$\neq$NP) by giving NP-hardness proofs for numerous variations of the problem along with polynomial approximation algorithms of lower approximation quality. Table 1 summarises the details of the findings.

Additionally, for the case of short accepting lassos for Muller automata, we have established its connection to the travelling salesman problem by identifying it as special case of the asymmetric metric TSP.

We considered the automata types currently employed in model checking applications as well as those that currently mainly serve as models in theoretical works in order to fill the automata-theoretic toolbox for use cases which have not been discovered yet.

At a first glance, the non-approximability results for Büchi and generalized Büchi automata are discouraging: Assuming that P$\neq$NP, the implementation of methods for extracting approximate shortest witnesses (or approximate shortest lassos in the case of fair systems) for the non-satisfaction of a specification in future model checkers appears not to be a fruitful idea. However, it should be noted that the identification of these problems as being hard helps preparing the field for the development of suitable heuristics. Also, the hardness results obtained may serve as justification for developing counter-example quality metrics which also base on other objectives than only their size.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
2. Büchi, J.R.: On a decision method in restricted second-order arithmetic. In: Proc. 1960 Int. Congr. for Logic, Methodology, and Philosophy of Science, pp. 1–11 (1962)

3. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)
4. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC, pp. 427–432 (1995)
5. Dinur, I., Guruswami, V., Khot, S., Regev, O.: A new multilayered PCP and the hardness of hypergraph vertex cover. SIAM J. Comput. 34(5), 1129–1146 (2005)
6. Engebretsen, L., Karpinski, M.: Approximation hardness of TSP with bounded metrics. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 201–212. Springer, Heidelberg (2001)
7. Farwer, B.: ω-automata. In: [10], pp. 3–20
8. Frieze, A.M., Galbiati, G., Maffioli, F.: On the worst-case performance of some algorithms for the asymmetric traveling salesman problem. Networks 12(1), 23–39 (1982)
9. Gastin, P., Moro, P.: Minimal counterexample generation for SPIN. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 24–38. Springer, Heidelberg (2007)
10. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games: A Guide to Current Research. LNCS, vol. 2500. Springer, Heidelberg (2002)
11. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
12. Grumberg, O., Veith, H. (eds.): 25 Years of Model Checking - History, Achievements, Perspectives. LNCS, vol. 5000. Springer, Heidelberg (2008)
13. Hansen, H., Geldenhuys, J.: Cheap and small counterexamples. In: Cerone, A., Gruner, S. (eds.) SEFM, pp. 53–62. IEEE Computer Society, Los Alamitos (2008)
14. Henzinger, M.R., Telle, J.A.: Faster algorithms for the nonemptiness of Streett automata and for communication protocol pruning. In: Karlsson, R.G., Lingas, A. (eds.) SWAT, pp. 16–27. Springer, Heidelberg (1996)
15. Khot, S., Regev, O.: Vertex cover might be hard to approximate to within 2-ϵ. J. Comput. Syst. Sci. 74(3), 335–349 (2008)
16. Kupferman, O., Sheinvald-Faragy, S.: Finding shortest witnesses to the nonemptiness of automata on infinite words. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 492–508. Springer, Heidelberg (2006)
17. Mitra, R.S.: Strategies for mainstream usage of formal verification. In: Fix, L. (ed.) DAC, pp. 800–805. ACM, New York (2008)
18. Safra, S.: Complexity of Automata on Infinite Objects. PhD thesis, Weizmann Institute of Science, Rehovot, Israel (March 1989)
19. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
20. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata, pp. 238–266. Springer, New York (1996)
21. Wegener, I.: Complexity Theory. In: Exploring the Limits of Efficient Algorithms. Springer, Heidelberg (2004)
22. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)

Grammar-Based Compression in a Streaming Model

Travis Gagie[1,⋆] and Paweł Gawrychowski[2]

[1] Department of Computer Science, University of Chile
travis.gagie@gmail.com

[2] Institute of Computer Science, University of Wrocław, Poland
gawry1@gmail.com

Abstract. We show that, given a string s of length n, with constant memory and logarithmic passes over a constant number of streams we can build a context-free grammar that generates s and only s and whose size is within an $\mathcal{O}\Big(\min\Big(g \log g, \sqrt{n/\log n}\Big)\Big)$-factor of the minimum g. This stands in contrast to our previous result that, with polylogarithmic memory and polylogarithmic passes over a single stream, we cannot build such a grammar whose size is within any polynomial of g.

1 Introduction

In the past decade, the ever-increasing amount of data to be stored and manipulated has inspired intense interest in both grammar-based compression and streaming algorithms, resulting in many practical algorithms and upper and lower bounds for both problems. Nevertheless, there has been relatively little study of grammar-based compression in a streaming model. In a previous paper [13] we proved limits on the quality of the compression we can achieve with polylogarithmic memory and polylogarithmic passes over a single stream. In this paper we show how to achieve better compression with constant memory and logarithmic passes over a constant number of streams.

For grammar-based compression of a string s of length n, we try to build a small context-free grammar (CFG) that generates s and only s. This is useful not only for compression but also for, e.g., indexing [10,23] and speeding up dynamic programs [24]. (It is sometimes desirable for the CFG to be in Chomsky normal form (CNF), in which case it is also known as a straight-line program.) We can measure our success in terms of universality [18], empirical entropy [30] or the ratio between the size of our CFG and the size $g = \Omega(\log n)$ of the smallest such grammar. In this paper we consider the third and last measure. Storer and Szymanski [37] showed that determining the size of the smallest grammar is NP-complete; Charikar et al. [8] showed it cannot be approximated to within a small

⋆ Funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

constant factor in polynomial time unless $\mathrm{P} = \mathrm{NP}$, and that even approximating it to within a factor of $o(\log n / \log \log n)$ in polynomial time would require progress on a well-studied algebraic problem. Charikar et al. and Rytter [31] independently gave $\mathcal{O}(\log(n/g))$-approximation algorithms, both based on turning the LZ77 [38] parse of s into a CFG and both initially presented at conference in 2002; Sakamoto [32] then proposed another $\mathcal{O}(\log(n/g))$-approximation algorithm, based on Re-Pair [22]. Sakamoto, Kida and Shimozono [33] gave a linear-time $\mathcal{O}((\log g)(\log n))$-approximation algorithm that uses $\mathcal{O}(g \log g)$ workspace, again based on LZ77; together with Maruyama, they [34] recently modified their algorithm to run in $\mathcal{O}(n \log^* n)$ time but achieve an $\mathcal{O}((\log n)(\log^* n))$ approximation ratio.

A few years before Charikar et al.'s and Rytter's papers sparked a surge of interest in grammar-based compression, a paper by Alon, Matias and Szegedy [2] did the same for streaming algorithms. We refer the reader to Babcock et al. [4] and Muthukrishnan [28] for a thorough introduction to streaming algorithms. In this paper, however, we are most concerned with more powerful streaming models than these authors consider, ones that allow the use of multiple streams. A number of recent papers have considered such models, beginning with Grohe and Schweikardt's [16] definition of (r, s, t)-bounded Turing Machines, which use at most r reversals over t "external-memory" tapes and a total of at most s space on "internal-memory" tapes to which they have unrestricted access. While Munro and Paterson [27] proved tight bounds for sorting with one tape three decades ago, Grohe and Schweikardt proved the first tight bounds for sorting with multiple tapes. Grohe, Hernich and Schweikardt [15] proved lower bounds in this model for randomized algorithms with one-sided error, and Beame, Jayram and Rudra [6] proved lower bounds for algorithms with two-sided error (renaming the model "read/write streams"). Beame and Huynh [5] revisited the problem considered by Alon, Matias and Szegedy, i.e., approximating frequency moments, and proved lower bounds for read/write stream algorithms. Hernich and Schweikardt [17] related results for read/write stream algorithms to results in classical complexity theory, including results by Chen and Yap [9] on reversal complexity. Hernich and Schweikardt's paper drew our attention to a theorem by Chen and Yap implying that, if a problem can be solved deterministically with read-only access to the input and logarithmic workspace then, in theory, it can be solved with constant memory and logarithmic passes (in either direction) over a constant number of read/write streams. This theorem is the key to our main result in this paper. Unfortunately, the constants involved in Chen and Yap's construction are enormous; we leave as future work finding a more practical proof of our results.

The study of compression in something like a streaming model goes back at least a decade, to work by Sheinwald, Lempel and Ziv [36] and De Agostino and Storer [11]. As far as we know, however, our joint paper with Manzini [14] was the first to give nearly tight bounds in a standard streaming model. In that paper we proved nearly matching bounds on the compression achievable with a constant amount of internal memory and one pass over the input, as well as upper and lower bounds for LZ77 with a sliding window whose size grows as a function of the number of characters encoded. (Our upper bound for LZ77 used a theorem due to Kosaraju and Manzini [20] about quasi-distinct parsings, a subject recently revisited by Amir, Aumann, Levy and Roshko [3].) Shortly thereafter, Albert, Mayordomo, Moser and Perifel [1] showed the compression achieved by LZ78 [39] is incomparable to that achievable by pushdown transducers; Mayordomo and Moser [26] then extended their result to show both kinds of compression are incomparable with that achievable by online algorithms with polylogarithmic memory. (A somewhat similar subject, recognition of the context-sensitive Dyck languages in a streaming model, was recently broached by Magniez, Mathieu and Nayak [25], who gave a one-pass algorithm with one-sided error that uses polylogarithmic time per character and $\mathcal{O}\big(n^{1/2}\log n\big)$ space.) In a recent paper with Ferragina and Manzini [12] we demonstrated the practicality of streaming algorithms for compression in external memory.

In a recent paper [13] we proved several lower bounds for compression algorithms that use a single stream, all based on an automata-theoretic lemma: suppose a machine implements a lossless compression algorithm using sequential accesses to a single tape that initially holds the input; then we can reconstruct any substring given, for every pass, the machine's configurations when it reaches and leaves the part of the tape that initially holds that substring, together with all the output it generates while over that part. (We note similar arguments appear in computational complexity, where they are referred to as "crossing sequences", and in communication complexity.) It follows that, if a streaming compression algorithm is restricted to using polylogarithmic memory and polylogarithmic passes over one stream, then there are periodic strings with polylogarithmic periods such that, even though the strings are very compressible as, e.g., CFGs, the algorithm must encode them using a linear number of bits; therefore, no such algorithm can approximate the smallest-grammar problem to within any polynomial of the minimum size. Such arguments cannot prove lower bounds for algorithms with multiple streams, however, and we left open the question of whether extra streams allow us to achieve a polynomial approximation. In this paper we use Chen and Yap's result to confirm they do: we show how, with logarithmic workspace, we can compute the LZ77 parse and turn that into a CFG in CNF while increasing the size by a factor of $\mathcal{O}\Big(\min\Big(g\log g, \sqrt{n/\log n}\Big)\Big)$ — i.e., at most polynomially in g. It follows that we can achieve that approximation ratio while using constant memory and logarithmic passes over a constant number of streams.

2 LZ77 in a Streaming Model

Our starting point is the same as that of Charikar et al., Rytter and Sakamoto, Kida and Shimozono, but we pay even more attention to workspace than the last set of authors. Specifically, we begin by considering the variant of LZ77 considered by Charikar et al. and Rytter, which does not allow self-referencing phrases but still produces a parse whose size is at most as large as that of the smallest grammar. Each phrase in this parse is either a single character or a substring of the prefix of s already parsed. For example, the parse of "how-much-wood-would-a-woodchuck-chuck-if-a-woodchuck-could-chuck-wood?" is "h|o|w|-|m|u|c|h|-|w|o|o|d|-wo|u|l|d-|a|-wood|ch|uc|k|-|chuck-|i|f|-a-woodchuck-c|ould-|chuck-|wood|?".

Lemma 1 (Charikar et al., 2002; Rytter, 2002). *The number of phrases in the LZ77 parse is a lower bound on the size of the smallest grammar.*

As an aside, we note that Lemma 1 and results from our previous paper [13] together imply we cannot compute the LZ77 parse with one stream when the product of the memory and passes is sublinear in n.

It is not difficult to show that this LZ77 parse — like the original — can be computed with read-only access to the input and logarithmic workspace. Pseudocode for doing this appears as Algorithm 1. On the example above, this pseudocode produces "how-much-wood(9,3)ul(13,2)a(9,5)(7,2)(6,2)k-(27,6)if(20,14)(16,5)(27,6)(10,4)?".

Lemma 2. *We can compute the LZ77 parse with logarithmic workspace.*

Proof. The first phrase in the parse is the first letter in s; after outputting this, we always keep a pointer t to the division between the prefix already parsed and the suffix yet to be parsed. To compute each later phrase in turn, we check the length of the longest common prefix of $s[i..t-1]$ and $s[t..n]$, for $1 \leq i < t$; if the longest match has length 0 or 1, we output $s[t]$; otherwise, we output the value of the minimal i that maximizes the length of the longest common prefix, together with that length. This takes a constant number of pointers into s and a constant number of $\mathcal{O}(\log n)$-bit counters. □

Combined with Chen and Yap's theorem below, Lemma 2 implies that we can compute the LZ77 parse with constant workspace and logarithmic passes over a constant number of streams.

Theorem 1 (Chen and Yap, 1991). *If a function can be computed with logarithmic workspace, then it can be computed with constant workspace and logarithmic passes over a constant number of streams.*

As an aside, we note that Chen and Yap's theorem is actually much stronger than what we state here: they proved that, if $f(n) = \Omega(\log n)$ is reversal-computable (see [9] or [17] for an explanation) and a problem can be solved deterministically in $f(n)$ time, then it can be solved with constant workspace and $\mathcal{O}(f(n))$ passes over a constant number of tapes. Chen and Yap showed how a reversal-bounded Turing machine can simulate a space-bounded Turing machine by building a table of the possible configurations of the space-bounded machine. Schweikardt [35] pointed out that "this is of no practical use, since the resulting algorithm produces huge intermediate results, but it is of major theoretical interest" because it implies that a number of lower bounds are tight. We leave as future work finding a more practical proof our our results.

```
$t \leftarrow 1$;
while $t \leq n$ do
    $max\_match \leftarrow 0$;
    $max\_length \leftarrow 0$;
    for $i \leftarrow 1 \ldots t-1$ do
        $j \leftarrow 0$;
        while $s[i+j] = s[t+j]$ do
            $j \leftarrow j+1$;
        if $j > max\_length$ then
            $max\_match \leftarrow i$;
            $max\_length \leftarrow j$;
    if $max\_length \leq 1$ then
        print $s[t]$;
        $t \leftarrow t+1$;
    else
        print $(max\_match, max\_length)$;
        $t \leftarrow t + max\_length$;
```

Algorithm 1. Pseudocode for computing the LZ77 parse in logarithmic workspace

In the next section we prove the following lemma, which is the most technical part of this paper. By the size of a CFG, we mean the number of symbols on the righthand sides of the productions; notice this is at most a logarithmic factor less than the number of bits needed to express the CFG.

Lemma 3. *With logarithmic workspace we can turn the LZ77 parse into a CFG whose size is within a* $\mathcal{O}\Big(\min\big(g \log g, \sqrt{n/\log n}\big)\Big)$*-factor of minimum.*

Together with Lemma 2 and Theorem 1, Lemma 3 immediately implies our main result.

Theorem 2. *With constant workspace and logarithmic passes over a constant number of streams, we can build a CFG generating s and only s whose size is within a* $\mathcal{O}\Big(\min\big(g \log g, \sqrt{n/\log n}\big)\Big)$*-factor of minimum.*

3 Logspace CFG Construction

Unlike the LZ78 parse, the LZ77 parse cannot normally be viewed as a CFG, because the substring to which a phrase matches may begin or end in the middle of a preceding phrase. We note this obstacle has been considered by other authors in other circumstances, e.g., by Navarro and Raffinot [29] for pattern matching. Fortunately, we can remove this obstacle in logarithmic workspace, without increasing the number of phrases more than quadratically. To do this, for each phrase for which we output (i, ℓ), we ensure $s[i]$ is the first character in a phrase and $s[i+\ell]$ is the last character in a phrase (by breaking phrases in two, if necessary). For example, the parse

“h|o|w|-|m|u|c|h|-|w|o|o|d|-wo|u|l|d-|a|-wood|ch|uc|k|-|chuck
-|i|f|-a-woodchuck-c|ould-|chuck-|wood|?”

becomes

“h|o|w|-|m|u|c|h|-|w|o|o|d|-w ┃$_1$o|u|l|d ┃$_2$-|a|-wood|c ┃$_3$h|uc|k|-|c ┃$^3_4$huck
-|i|f|^2-a-woodchuck-c|^{1,4}ould-|chuck-|wood|?”,

where the thick lines indicate new breaks and superscripts indicate which breaks cause the new ones (which are subscripted). Notice the break “a-woodchuck-c|^{1,4}ould” causes both “w ┃$_1$ould” (matching “ould”) and “a-woodchuck-c ┃$^3_4$huck” (matching “a-woodchuck-c”); in turn, the latter new break causes “woodc ┃$_3$huck” (matching “huck”), which is why it has a superscript 3.

Lemma 4. *Breaking the phrases takes at most logarithmic workspace and at most squares the number of phrases. Afterwards, every phrase is either a single character or the concatenation of complete, consecutive, preceding phrases.*

Proof. Since the phrases' start points are the partial sums of their lengths, we can compute them with logarithmic workspace; therefore, we can assume without loss of generality that the start points are stored with the phrases. We start with the rightmost phrase and work left. For each phrase's endpoints, we compute the corresponding position in the matching, preceding substring (notice that the position corresponding to one phrase's finish may not be the one corresponding to the start of the next phrase to the right) and insert a new break there, if there is not one already. If we have inserted a new break, then we iterate, computing the position corresponding to the new break; eventually, we will reach a point where there is already a break, so the iteration will stop. This process requires only a constant number of pointers, so we can perform it with logarithmic workspace. Also, since each phrase is broken at most twice for each of the phrases that initially follow it in the parse, the final number of phrases is at most the square of the initial number. By inspection, after the process is complete every phrase is the concatenation of complete, consecutive, preceding phrases. □

Notice that, after we break the phrases as described above, we can view the parse as a CFG. For example, the parse for our running example corresponds to

$$\begin{array}{lll}
\mathsf{X}_0 \to \mathsf{X}_1 \ldots \mathsf{X}_{35} & \mathsf{X}_{14} \to \mathsf{X}_9\ \mathsf{X}_{10} & \vdots \\
\mathsf{X}_1 \to \mathsf{h} & \mathsf{X}_{15} \to \mathsf{o} & \mathsf{X}_{34} \to \mathsf{X}_{10} \ldots \mathsf{X}_{13} \\
\vdots & \vdots & \mathsf{X}_{35} \to \mathsf{?} \\
\mathsf{X}_{13} \to \mathsf{d} & \mathsf{X}_{31} \to \mathsf{X}_{19} \ldots \mathsf{X}_{27} &
\end{array}$$

where X_0 is the starting nonterminal. Unfortunately, while the number of productions is polynomial in the number of phrases in the LZ77 parse, it is not clear the size is and, moreover, the grammar is not in CNF. Since all the right-hand sides of the productions are either terminals or sequences of consecutive nonterminals, we could put the grammar into CNF by squaring the number of nonterminals — giving us an approximation ratio cubic in g. This would still be enough for us to prove our main result but, fortunately, such a large increase is not necessary.

Lemma 5. *Putting the CFG into CNF takes logarithmic workspace and increases the number of productions by at most a logarithmic factor. Afterwards, the size of the grammar is proportional to the number of productions.*

Proof. We build a forest of complete binary trees whose leaves are the nonterminals: if we consider the trees in order by size, the nonterminals appear in order from the leftmost leaf of the first tree to the rightmost leaf of the last tree; each tree is as large as possible, given the number of nonterminals remaining after we build the trees to its left. Notice there are $\mathcal{O}(\log g)$ such trees, of total size at most $\mathcal{O}(g^2)$. We then assign a new nonterminal to each internal node and output a production which takes that nonterminal to its children. This takes logarithmic workspace and increases the number of productions by a constant factor.

Notice any sequence of consecutive nonterminals that spans at least two trees, can be written as the concatenation of two consecutive sequences, one of which ends with the rightmost leaf in one tree and the other of which starts with the leftmost leaf in the next tree. Consider a sequence ending with the rightmost leaf in a tree; dealing with one that starts with a leftmost leaf is symmetric. If the sequence completely contains that tree, we can write a binary production that splits the sequence into the prefix in the preceding trees, which is the expansion of a new nonterminal, and the leaves in that tree, which are the expansion of its root. We need do this $\mathcal{O}(\log g)$ times before the remaining subsequence is contained within a single tree. After that, we repeatedly produce new binary productions that split the subsequence into prefixes, again the expansions of new nonterminals, and suffixes, the expansions of roots of the largest possible complete subtree. Since the size of the largest possible complete subtree shrinks by a factor of two at each step (or, equivalently, the height of its root decreases by 1), we need repeat $\mathcal{O}(\log g)$ times. Again, this takes logarithmic workspace (we will give more details in the full version of this paper).

In summary, we may replace each production with $\mathcal{O}(\log g)$ new, binary productions. Since the productions are binary, the number of symbols on the righthand sides is linear in the number of productions themselves. □

Lemma 5 is our most detailed result, and the diagram below showing the construction with our running example is also somewhat detailed. On the left are modifications of the original productions, now made binary; in the middle are productions for the internal nodes of the binary trees; and on the right are productions breaking down the consecutive subsequences that appear on the righthand sides of the productions in the left column, until the subsequences are single, original nonterminals or nonterminals for nodes in the binary trees (i.e., those on the lefthand sides of the productions in the middle column).

$$\begin{array}{lll}
\mathsf{X}_0 \rightarrow \mathsf{X}_{1,32}\ \mathsf{X}_{33,35} & \mathsf{X}_{1,32} \rightarrow \mathsf{X}_{1,16}\ \mathsf{X}_{17,32} & \\
\mathsf{X}_1 \rightarrow \mathsf{h} & \mathsf{X}_{1,16} \rightarrow \mathsf{X}_{1,8}\ \mathsf{X}_{9,16} & \\
\vdots & \mathsf{X}_{1,8} \rightarrow \mathsf{X}_{1,4}\ \mathsf{X}_{5,8} & \\
\mathsf{X}_{13} \rightarrow \mathsf{d} & \vdots & \\
\mathsf{X}_{14} \rightarrow \mathsf{X}_9\ \mathsf{X}_{10} & \mathsf{X}_{17,32} \rightarrow \mathsf{X}_{17,24}\ \mathsf{X}_{18,32} & \mathsf{X}_{19,24} \rightarrow \mathsf{X}_{19,20}\ \mathsf{X}_{21,24} \\
\mathsf{X}_{15} \rightarrow \mathsf{o} & \mathsf{X}_{17,24} \rightarrow \mathsf{X}_{17,20}\ \mathsf{X}_{21,24} & \mathsf{X}_{25,27} \rightarrow \mathsf{X}_{25,26}\ \mathsf{X}_{27} \\
\vdots & \vdots & \vdots \\
\mathsf{X}_{31} \rightarrow \mathsf{X}_{19,24}\ \mathsf{X}_{25,27} & \mathsf{X}_{29,32} \rightarrow \mathsf{X}_{29,30}\ \mathsf{X}_{31,32} & \mathsf{X}_{10,12} \rightarrow \mathsf{X}_{10}\ \mathsf{X}_{11,12} \\
\vdots & \mathsf{X}_{33,35} \rightarrow \mathsf{X}_{33,34}\ \mathsf{X}_{35} & \\
\mathsf{X}_{34} \rightarrow \mathsf{X}_{10,12}\ \mathsf{X}_{13} & \mathsf{X}_{1,2} \rightarrow \mathsf{X}_1\ \mathsf{X}_2 & \\
\mathsf{X}_{35} \rightarrow \mathsf{?} & \vdots & \\
 & \mathsf{X}_{33,34} \rightarrow \mathsf{X}_{33}\ \mathsf{X}_{34} &
\end{array}$$

Combined with Lemma 1, Lemmas 4 and 5 imply that with logarithmic workspace we can build a CFG in CNF whose size is $\mathcal{O}\left(g^2 \log g\right)$. We can use a similar approach with binary trees to build a CFG in CNF of size $\mathcal{O}(n)$ that generates s and only s, still using logarithmic workspace. If we combine all non-terminals that have the same expansion, which also takes logarithmic workspace, then this becomes Kieffer, Yang, Nelson and Cosman's [19] Bisection algorithm, which gives an $\mathcal{O}\left(\sqrt{n/\log n}\right)$-approximation [8]. By taking the smaller of these two CFGs we achieve an $\mathcal{O}\left(\min\left(g \log g, \sqrt{n/\log n}\right)\right)$-approximation. Therefore, as we claimed in Lemma 3, with logarithmic workspace we can turn the LZ77 parse into a CFG whose size is within a $\mathcal{O}\left(\min\left(g \log g, \sqrt{n/\log n}\right)\right)$-factor of minimum.

4 Recent Work

We recently improved the bound on the approximation ratio in Lemma 3 from $\mathcal{O}\left(\min\left(g \log g, \sqrt{n/\log n}\right)\right)$ to $\mathcal{O}\left(\min\left(g, \sqrt{n/\log n}\right)\right)$. The key observation is that, by the definition of the LZ77 parse, the first occurrence of any substring must touch or cross a break between phrases. Consider any phrase in the parse obtained by applying Lemma 4 to the LZ77 parse. By the observation above, that phrase can be written as the concatenation of some consecutive new phrases (all contained within one old phrase and ending at that old phrase's right end), some consecutive old phrases, and some more consecutive new phrases (all contained within one old phrase and starting at the old phrase's left end). Since there are $\mathcal{O}(g)$ old phrases, there are $\mathcal{O}\left(g^2\right)$ sequences of consecutive old phrases; since there are $\mathcal{O}\left(g^2\right)$ new phrases, there are $\mathcal{O}\left(g^2\right)$ sequences of consecutive new phrases that are contained in one old phrase and either start at that old phrase's right end or end at that old phrase's left end.

While working on the improvement above, we realized how to improve the bound further, to $\mathcal{O}\left(\min\left(g, 4^{\sqrt{\log n}}\right)\right)$. To do this, we choose a value b between

2 and n and, for $0 \leq i \leq \log_b n$, we associate a nonterminal to each of the b blocks of $\lceil n/b^i \rceil$ characters to the left and right of each break; we thus start building the grammar with $\mathcal{O}(bg \log_b n)$ nonterminals. We then add $\mathcal{O}(bg \log_b n)$ binary productions such that any sequence of nonterminals associated with a consecutive sequence of blocks, can be derived from $\mathcal{O}(1)$ nonterminals. Notice any substring is the concatenation of 0 or 1 partial blocks, some number of full blocks to the left of a break, some number of blocks to the right of a break, and 0 or 1 more partial blocks. We now add more binary productions as follows: we start with s (the only block of length $\lceil n/b^0 \rceil = n$); find the first break it touches or crosses (in this case it is the start of s); consider s as the concatenation of blocks of size $\lceil n/b^1 \rceil$ (in this case only the rightmost block can be partial); associate nonterminals to the partial blocks (if they exist); add $\mathcal{O}(1)$ productions to take the symbol associated to s (in this case, the start symbol) to the sequence of nonterminals associated with the smaller blocks in order from left to right; and recurse on each of the smaller blocks. To guarantee each smaller block touches or crosses a break, we work on the first occurrence in s of the substring contained in that block. We stop recursing when the block size is 1, and add $\mathcal{O}(bg)$ productions taking those blocks' nonterminals to the appropriate characters.

Analysis shows that the number of productions we add during the recursion is proportional to the number of blocks involved, either full or partial. Since the number of distinct full blocks in any level of recursion is $\mathcal{O}(bg)$ and the number of partial blocks is at most twice the number of blocks (full or partial) in the previous level of recursion, the number of productions we add during the recursion is $\mathcal{O}\big(2^{\log_b n} bg\big)$. Therefore, the grammar has size $\mathcal{O}\big(2^{\log_b n} bg\big)$; when $b = 2^{\sqrt{\log n}}$, this is $\mathcal{O}\Big(4^{\sqrt{\log n}} g\Big)$. The first of the two key observations that let us build the grammar in logarithmic workspace, is that we can store the index of a block (full or partial) with respect to the associated break, in $\mathcal{O}\big(\sqrt{\log n}\big)$ bits; therefore, we can store $\mathcal{O}(1)$ indices for each of the $\mathcal{O}\big(\sqrt{\log n}\big)$ levels of recursion, in a total of $\mathcal{O}(\log n)$ bits. The second key observation is that, given the indices of the block we are working on in each level of recursion, with respect to the appropriate break, we can compute the start point and end point of the block we are currently working on in the deepest level of recursion. We will give details of these two improvements in the full version of this paper.

While working on this second improvement, we realized that we can use the same ideas to build a compressed representation that allows efficient random access. We refer the reader to the recent papers by Kreft and Navarro [21] and Bille, Landau and Weimann [7] for background on this problem. Suppose that, for each of the $\mathcal{O}\Big(2^{\sqrt{\log n}} g \sqrt{\log n}\Big)$ full blocks described above, we store a pointer to the first occurrence in s of the substring in that block, as well as a pointer to the first break that first occurrence touches or crosses. Notice this takes a total of $\mathcal{O}\Big(2^{\sqrt{\log n}} g (\log n)^{3/2}\Big)$ bits. Then, given a block's index and an offset in that block, in $\mathcal{O}(1)$ time we can compute a smaller block's index and offset in that smaller block, such that the characters in those two positions are equal; if the larger block has size 1, in $\mathcal{O}(1)$ time we can return the character. Since s itself is a block, an offset in it is just a character's position, and there are $\mathcal{O}\left(\sqrt{\log n}\right)$ levels of recursion, it follows that we can access any character in $\mathcal{O}\left(\sqrt{\log n}\right)$ time. Further analysis shows that it takes $\mathcal{O}\left(\sqrt{\log n} + \ell\right)$ time to access a substring of length ℓ. Of course, for any positive constant ϵ, if we are willing to use $\mathcal{O}(n^{\epsilon} g)$ bits of space, then we can access any character in constant time. If we make the data structure slightly larger and more complicated — e.g., storing searchable partial sums at the block boundaries — then, at least for strings over fairly small alphabets, we can also support fast rank and select queries.

This implementation makes it easy to see the data structure's relation to LZ77 and grammar-based compression. We can use a simpler implementation, however, and use LZ77 only in the analysis. Suppose that, for $0 \leq i \leq \log_b n$, we break s into consecutive blocks of length $\lceil n/b^i \rceil$ (the last block may be shorter), always starting from the first character of s. For each block, we store a pointer to the first occurrence in s of that block's substring. Given a block's index and an offset in that block, in $\mathcal{O}(1)$ time we can again compute a smaller block's index and offset in that smaller block, such that the characters in those two positions are equal: the new block's index is the sum of pointer and the old offset, divided by the new block length and rounded down; the new offset is the sum of the pointer and the old offset, modulo the new block length. We can discard any block that cannot be visited during a query, so this data structure takes at most a constant factor more space than the one described above. Indeed, this data structure seems likely to be smaller in practice, because blocks of the same size can overlap in the previous data structure but cannot in this one. We plan to implement this data structure and report the results in a future paper.

References

1. Albert, P., Mayordomo, E., Moser, P., Perifel, S.: Pushdown compression. In: Proceedings of the Symposium on Theoretical Aspects of Computer Science, pp. 39–48 (2008)
2. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and System Sciences 58(1), 137–147 (1999)
3. Amir, A., Aumann, Y., Levy, A., Roshko, Y.: Quasi-distinct parsing and optimal compression methods. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 12–25. Springer, Heidelberg (2009)
4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the Symposium on Database Systems, pp. 1–16 (2002)
5. Beame, P., Huynh, T.: On the value of multiple read/write streams for approximating frequency moments. In: Proceedings of the Symposium on Foundations of Computer Science, pp. 499–508 (2008)
6. Beame, P., Jayram, T.S., Rudra, A.: Lower bounds for randomized read/write stream algorithms. In: Proceedings of the Symposium on Theory of Computing, pp. 689–698 (2007)
7. Bille, P., Landau, G., Weimann, O.: Random access to grammar compressed strings (2010), http://arxiv.org/abs/1001.1565

8. Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., shelat, a.[1]: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)
9. Chen, J., Yap, C.-K.: Reversal complexity. SIAM Journal on Computing 20(4), 622–638 (1991)
10. Claude, F., Navarro, G.: Self-indexed text compression using straight-line programs. In: Královič, R., Niwiński, D. (eds.) MFCS 2009. LNCS, vol. 5734, pp. 235–246. Springer, Heidelberg (2009)
11. De Agostino, S., Storer, J.A.: On-line versus off-line computation in dynamic text compression. Information Processing Letters 59(3), 169–174 (1996)
12. Ferragina, P., Gagie, T., Manzini, G.: Lightweight data indexing and compression in external memory. In: Proceedings of the Latin American Theoretical Informatics Symposium (to appear, 2010)
13. Gagie, T.: On the value of multiple read/write streams for data compression. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 68–77. Springer, Heidelberg (2009)
14. Gagie, T., Manzini, G.: Space-conscious compression. In: Kučera, L., Kučera, A. (eds.) MFCS 2007. LNCS, vol. 4708, pp. 206–217. Springer, Heidelberg (2007)
15. Grohe, M., Hernich, A., Schweikardt, N.: Lower bounds for processing data with few random accesses to external memory. Journal of the ACM 56(3), 1–58 (2009)
16. Grohe, M., Schweikardt, N.: Lower bounds for sorting with few random accesses to external memory. In: Proceedings of the Symposium on Database Systems, pp. 238–249 (2005)
17. Hernich, A., Schweikardt, N.: Reversal complexity revisited. Theoretical Computer Science 401(1-3), 191–205 (2008)
18. Kieffer, J.C., Yang, E.-H.: Grammar-based codes: A new class of universal lossless source codes. IEEE Transactions on Information Theory 46(3), 737–754 (2000)
19. Kieffer, J.C., Yang, E.-H., Nelson, G.J., Cosman, P.C.: Universal lossless compression via multilevel pattern matching. IEEE Transactions on Information Theory 46(4), 1227–1245 (2000)
20. Kosaraju, S.R., Manzini, G.: Compression of low entropy strings with Lempel-Ziv algorithms. SIAM Journal on Computing 29(3), 893–911 (1999)
21. Kreft, S., Navarro, G.: LZ77-like compression with fast random access. In: Proceedings of the Data Compression Conference (to appear, 2010)
22. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)
23. Lifshits, Y.: Processing compressed texts: A tractability border. In: Proceedings of the Symposium on Combinatorial Pattern Matching, pp. 228–240 (2007)
24. Lifshits, Y., Mozes, S., Weimann, O., Ziv-Ukelson, M.: Speeding up HMM decoding and training by exploiting sequence repetitions. Algorithmica 54(3), 379–399 (2009)
25. Magniez, F., Mathieu, C., Nayak, A.: Recognizing well-parenthesized expressions in the streaming model. Technical Report TR09-119, Electronic Colloquium on Computational Complexity (2009)
26. Mayordomo, E., Moser, P.: Polylog space compression is incomparable with Lempel-Ziv and pushdown compression. In: Proceedings of the Conference on Current Trends in Theory and Practice of Informatics, pp. 633–644 (2009)
27. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theoretical Computer Science 12, 315–323 (1980)

[1] abhi shelat is calling himself using lowercases, for more references see `http://www.cs.virginia.edu/~shelat/research/`

28. Muthukrishnan, S.: Data Streams: Algorithms and Applications. In: Foundations and Trends in Theoretical Computer Science, vol. 1(2). Now Publishers (2005)
29. Navarro, G., Raffinot, M.: Practical and flexible pattern matching over Ziv-Lempel compressed text. Journal of Discrete Algorithms 2(3), 347–371 (2004)
30. Navarro, G., Russo, L.M.S.: Re-pair achieves high-order entropy. In: Proceedings of the Data Compression Conference, p. 537 (2008)
31. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theoretical Computer Science 302(1-3), 211–222 (2003)
32. Sakamoto, H.: A fully linear-time approximation algorithm for grammar-based compression. Journal of Discrete Algorithms 3(2-4), 416–430 (2005)
33. Sakamoto, H., Kida, T., Shimozono, S.: A space-saving linear-time algorithm for grammar-based compression. In: Apostolico, A., Melucci, M. (eds.) SPIRE 2004. LNCS, vol. 3246, pp. 218–229. Springer, Heidelberg (2004)
34. Sakamoto, H., Maruyama, S., Kida, T., Shimozono, S.: A space-saving approximation algorithm for grammar-based compression. IEICE Transactions 92-D(2), 158–165 (2009)
35. Schweikardt, N.: Machine models and lower bounds for query processing. In: Proceedings of the Symposium on Principles of Database Systems, pp. 41–52 (2007)
36. Sheinwald, D., Lempel, A., Ziv, J.: On encoding and decoding with two-way head machines. Information and Computation 116(1), 128–133 (1995)
37. Storer, J.A., Szymanski, T.G.: Data compression via textual substitution. Journal of the ACM 29(4), 928–951 (1982)
38. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory 23(3), 337–343 (1977)
39. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)

Simplifying Regular Expressions
A Quantitative Perspective

Hermann Gruber[1] and Stefan Gulan[2]

[1] Institut für Informatik, Universität Gießen,
Arndtstraße 2, D-35392 Gießen, Germany
hermann.gruber@informatik.uni-giessen.de
[2] Fachbereich IV—Informatik, Universität Trier,
Campus II, D-54296 Trier, Germany
gulan@uni-trier.de

Abstract. We consider the efficient simplification of regular expressions and suggest a quantitative comparison of heuristics for simplifying regular expressions. To this end, we propose a new normal form for regular expressions, which outperforms previous heuristics while still being computable in linear time. This allows us to determine an exact bound for the relation between the two prevalent measures for regular expression - size: alphabetic width and reverse polish notation length. In addition, we show that every regular expression of alphabetic width n can be converted into a nondeterministic finite automaton with ε-transitions of size at most $4\frac{2}{5}n+1$, and prove this bound to be optimal. This answers a question posed by Ilie and Yu, who had obtained lower and upper bounds of $4n-1$ and $9n-\frac{1}{2}$, respectively [15]. For reverse polish notation length as input size measure, an optimal bound was recently determined by Gulan and Fernau [14]. We prove that, under mild restrictions, their construction is also optimal when taking alphabetic width as input size measure.

1 Introduction

It is well known that simplifying regular expressions is hard, since alone deciding whether a given regular expression describes the set of all strings, is **PSPACE** - complete [17]. As witnessed by a number of recent studies, e.g. [5,10,11,12,13], the descriptional complexity of regular expressions is of great interest, and several heuristics for simplifying regular expressions appear in the literature. These mostly deal with removing only the most obvious redundancies, such as iterated Kleene stars or superfluous occurrences the empty word [15,4,8,9].

We take a quantitative viewpoint to compare such simplifications; namely, we compare the total size of a regular expression (disregarding parentheses) to its alphabetic width. The intuition behind this is as follows: Certain simplifications for regular expressions are of an ad-hoc nature, e.g. the rule $r + r = r$ cannot simplify $a^* + (a + b)^*$. Also, there are rules that are difficult to apply, e.g. if $L(r) \subseteq L(s)$, then $r + s = s$. But there are also simplifications that do not fall in either category, such as the reduction rules suggested in [15,4,8,9,16]. In this paper, we suggest a *strong star normal form* of regular expressions, which is a variation of the star normal form defined in [4]. This normal form achieves an optimal ratio when comparing expression size to alphabetic width, and can be computed as efficiently as the original star normal form.

For converting regular expressions into small ε-NFAs, an optimal construction was found recently in [14]. Here, *optimal* means that the algorithm attains the best possible ratio of *expression size* to *automaton size*. Ilie and Yu [15] asked for the optimal quotient if expression size is replaced with *alphabetic width*; they obtained an upper bound of roughly 9. We resolve this open problem by showing that the quotient equals $4\frac{2}{5}$. In fact, we prove that the construction from [14] attains this bound if the input expression is in strong star normal form. We move on to show that this still holds, under very mild restrictions, also for expressions not in star normal form. Our results suggest that this construction of ε-NFAs from regular expressions is optimal in a robust sense.

2 Basic Notions

Let Σ be a set of symbols, called *letters*. Regular expression over Σ, or just *expressions*, are defined as follows: Every letter is an expression and if r_1 and r_2 are expressions, so are (r_1+r_2), $(r_1 \cdot r_2)$, $(r_1)^?$ and $(r_1)^*$. The language denoted by an expression r, written $L(r)$, is defined inductively: $L(a) = \{a\}$, $L(r_1 + r_2) = L(r_1) \cup L(r_2)$, $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$, $L(r_1^?) = \{\varepsilon\} \cup L(r_1)$ and $L(r_1^*) = L(r_1)^*$. A language is called regular if it is definable by an expression.

We deviate from the convention by omitting symbols denoting the empty set and the empty word, while allowing for a special operator that adds the empty word to a language. The disadvantages of our definition are minor—we cannot describe the degenerate languages $\emptyset$ and $\{\varepsilon\}$; on the plus side, our syntax prevents *a priori* the construction of many kinds of unnatural and redundant expressions, such as $\varepsilon \cdot r$ or $\emptyset^*$.

There are two prevalent measures for the length of expressions: The *alphabetic width* of r, denoted $\mathrm{alph}(r)$ is defined as the total number of occurrences of letters in r. The second measure is the reverse polish notation length. To allow for comparison with related works, e.g., [8,15], we define the (abbreviated) reverse polish notation length of r as $\mathrm{arpn}(r) = |r|_\Sigma + |r|_+ + |r|_\cdot + |r|_* + |r|_?$, and its unabbreviated rpn-length as $\mathrm{rpn}(r) = \mathrm{arpn}(r) + |r|_?$. This reflects the fact that replacing each subexpression of the form $s^?$ with $s + \varepsilon$ increases the overall length by 1 each time. The alphabetic width of a regular language L is defined as the minimum alphabetic width among all expressions denoting L, and is denoted $\mathrm{alph}(L)$. The notions $\mathrm{rpn}(L)$ and $\mathrm{arpn}(L)$ are defined correspondingly.

Some notions from term rewriting are needed:Let S be a set, and let $\to$ be a relation on S. Let $\to^*$ denote the transitive closure of $\to$. Two elements $b, c \in S$ are called *joinable*, if some $d \in S$ satisfies $b \to^* d$ and $c \to d$. The relation $\to$ is *confluent*, if for all $a, b, c \in S$ with $a \to^* b$ and $a \to^* c$, the elements b and c are joinable. It is *locally confluent*, if for all $a, b, c \in S$ with $a \to b$ and $a \to c$, the elements b and c are joinable. The relation is *terminating*, if there is no infinite descending chain $a_1 \to a_2 \to \cdots$.

It is easily proven that if $\to$ is confluent and terminating, then each element has a unique normal form, see e.g. [2, Thm. 2.1.9]. Indeed for unique normal forms, we only need to establish local confluence instead of confluence: Newman's Lemma states that if a terminating relation is locally confluent, then it is confluent ([18], see also [2, Lem. 2.7.2]).

3 Alphabetic Width versus Reverse Polish Notation Length

We adapt the *star normal form* of expressions, proposed by Brueggemann-Klein [4], to our needs.

Definition 1. *The operators* $\circ$ *and* $\bullet$ *are defined on expressions as follows: For the first operator, let* $a^\circ = a$, *for* $a \in \Sigma$, $(r+s)^\circ = r^\circ + s^\circ$, $r^{?\circ} = r^\circ$, $r^{*\circ} = r^\circ$, *and*

$$(rs)^\circ = \begin{cases} rs, & \textit{if } \varepsilon \notin L(rs) \\ r^\circ + s^\circ & \textit{else} \end{cases}.$$

The second operator is given by: $a^\bullet = a$, *for* $a \in \Sigma$, $(r+s)^\bullet = r^\bullet + s^\bullet$, $(rs)^\bullet = r^\bullet s^\bullet$, $r^{*\bullet} = r^{\bullet\circ *}$, *and*

$$r^{?\bullet} = \begin{cases} r^\bullet & , \textit{if } \varepsilon \in L(r) \\ r^{\bullet ?} & \textit{otherwise} \end{cases}.$$

The strong star normal form *of an expression* r *is then defined as* $r^\bullet$.

Observe that, e.g., the expression $(\emptyset + a)^* + \varepsilon \cdot b + \emptyset \cdot c \cdot (d + \varepsilon + \varepsilon)$ in unabbreviated syntax is in star normal form, so the relative advantage of strong star normal form should be obvious. The difference to star normal form merely consists in using abbreviated syntax and in the addition of a rule for computing $r^{?\bullet}$. All the statements in the original work [4, Thm. 3.1, Lem. 3.5, 3.6, 3.7] regarding $^\circ$ and $^\bullet$ carry over to our variation.

We compare rpn-length and alphabetic width of expressions in strong star normal form. To this end, for an expression r in abbreviated syntax, define $\omega(r) = |r|_? + |r|_*$, that is, ω counts the total number of occurrences of unary operators in r. The following property is evident from the definition of $\bullet$ and $\circ$; a similar statement concerning rpn-length is found in [4].

Lemma 1. *Let* r *be an expression. Then* $\omega(r^\bullet), \omega(r^\circ) \leq \omega(r)$, *and* $\omega(r^{*\circ}) \leq \omega(r^*) - 1$.

Lemma 2. *Let* r *be an expression, then* $\omega(r^\bullet) \leq \operatorname{alph}(r^\bullet)$, *if* $\varepsilon \in L(s)$, *and* $\omega(r^\bullet) \leq \operatorname{alph}(r^\bullet) - 1$ *otherwise.*

Proof. By lexicographic induction on the pair (n, h), where $n = \operatorname{alph} r^\bullet$, and h is the height of the parse of r. The base case is $(1, 1)$, i.e., $r^\bullet \in \Sigma$, for which the statement clearly holds. Assume the claim is true for expressions of alphabetic width at most $n-1$ and for expressions of alphabetic width n and height at most $k-1$. The nontrivial cases for the induction step are $r = s^?$ and $r = s^*$. In the first case, we have $r^\bullet = s^\bullet$, if $\varepsilon \in L(s)$. Applying the induction hypothesis to $s^\bullet$ yields

$$\operatorname{alph}(r^\bullet) = \operatorname{alph}(s^\bullet) \geq \omega(s^\bullet) = \omega(r^\bullet).$$

If $\varepsilon \notin L(s)$, then $r^\bullet = s^{\bullet ?}$, where again the induction hypothesis applies for s. This time, we obtain

$$\operatorname{alph}(r^\bullet) = \operatorname{alph}(s^\bullet) \geq \omega(s^\bullet) + 1 = \omega(r^\bullet).$$

In case $r = s^*$, we need to distinguish by the structure of r. The easy cases are $r = s^{?*}$ and $r = s^{**}$: here, $r^\bullet = s^{*\bullet}$ and the claim holds by induction. If $r = (s+t)^*$, expansion of the definition gives

$$r^\bullet = (s^{*\bullet\circ} + t^{*\bullet\circ})^*.$$

Since both $s^{*\bullet}$ and $t^{*\bullet}$ must have alphabetic width strictly less than n, and since both describe the empty word, we apply the inductive hypothesis to obtain

$$\mathrm{alph}(r^\bullet) = \mathrm{alph}(s^{*\bullet}) + \mathrm{alph}(t^{*\bullet}) \geq \omega(s^{*\bullet}) + \omega(t^{*\bullet}).$$

Now $\omega(s^{*\bullet\circ}) \leq \omega(s^{*\bullet}) - 1$, and similar for $t^{*\bullet\circ}$, we deduce that $\omega(r^\bullet) \leq \mathrm{alph}(r^\bullet) - 2$, which completes the induction step for this case.

For the case where $r = (st)^*$, we have $r^\bullet = (s^\bullet t^\bullet)^{\circ *}$ and the induction goes through if at least one of s and t does not describe the empty word. If however $\varepsilon \in L(s) \cap L(t)$, then it is easy to prove under this condition that $r^\bullet = (s+t)^{*\bullet}$, a case we already dealt with a few lines above in this proof. □

Theorem 1. *Any regular language L satisfies* $\mathrm{arpn}(L) \leq 3\,\mathrm{alph}(L) - 1$ *and* $\mathrm{rpn}(L) \leq 4\,\mathrm{alph}(L) - 1$.

Proof. Let r be an expression, in abbreviated syntax, of minimum alphabetic width denoting L. Then the parse tree of $r^\bullet$ has $\mathrm{alph}(r)$ many leaves. Disregarding unary operators, this is a binary tree with $\mathrm{alph}(r) - 1$ internal vertices that correspond to occurrences of binary operators in r. Since there are at most $\mathrm{alph}(r)$ many occurrences of unary operators, we have $\mathrm{arpn}(r^\bullet) \leq 3\,\mathrm{alph}(L) - 1$ and $\mathrm{rpn}(r^\bullet) \leq \mathrm{arpn}(r^\bullet) + \omega(r^\bullet) \leq 4\,\mathrm{alph}(L) - 1$. □

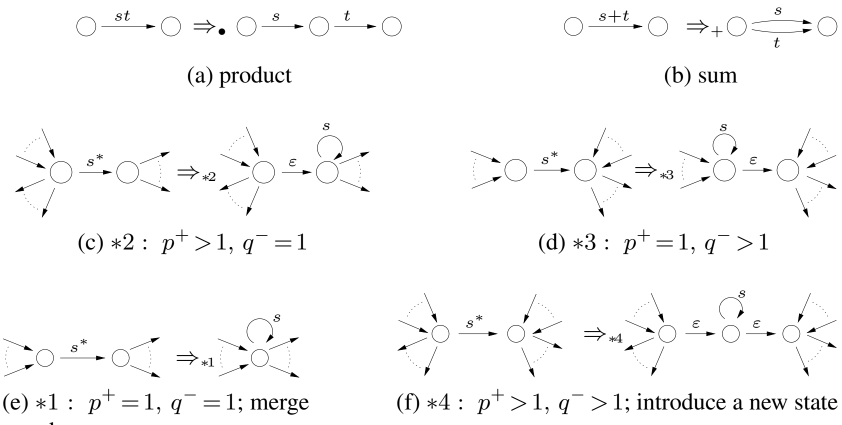
Thus size and alphabetic width can differ at most by a factor of 4 in unabbreviated syntax. Previous bounds, which were based on other simplification paradigms, by Ilie and Yu [15] and by Ellul et al. [8] only achieved factors of 6 and 7, respectively, in place of 4. For abbreviated syntax, we will later show that the bound of the form $3n - 1$ is best possible. Also note that strong star normal form subsumes all of the previous simplification heuristics from [4,8,15].

4 Constructing ε-NFAs from Regular Expressions, Revisited

We show that under mild restrictions, the construction given by Gulan and Fernau [14] subsumes the conversion of the input expression into strong star normal form. This construction is essentially a replacement system on digraphs, that are arc-labeled by regular expressions or the symbol ε. Such objects are called *extended finite automata* (EFAs), as they generalize (conventional) finite automata; consult Wood [20] for a proper introduction. The replacements are called *conversions*; they come in two flavors:

- A transition labeled by a regular expression may be replaced wrt. the labels root. These conversions, called *expansions*, are depicted in Fig. 1.
- A substructure defined by ε-transitions may be replaced by a smaller equivalent. These conversions are also called *eliminations*, they are shown in Fig. 2.

Since ε-transitions are allowed in EFAs, we treat $r^?$ implicitly as $r + \varepsilon$. We call the lhs of i-expansion or i-elimination an *i-anchor*, and write $E \Rightarrow_i E'$ if E' is derived from replacing an i-anchor in E with its according rhs. If the type of conversion is irrelevant, we simply write $E \Rightarrow E'$, and denote a (possibly empty) series of conversions from


(a) product

(b) sum

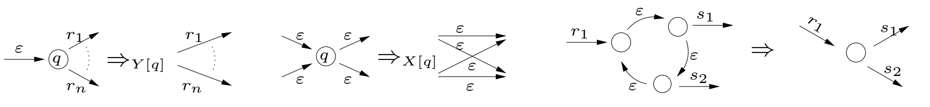
(c) $*2: \ p^+ > 1,\ q^- = 1$

(d) $*3: \ p^+ = 1,\ q^- > 1$

(e) $*1: \ p^+ = 1,\ q^- = 1$; merge p and q

(f) $*4: \ p^+ > 1,\ q^- > 1$; introduce a new state

Fig. 1. Expanding transitions (p, r, q) for nontrivial r. If $r = s^*$, the out-degree p^+ of p and the in-degree q^- of q need to be considered.


(a) Y-elimination, requires $q^- = 1$

(b) X-elimination, requires $q^- = q^+ = 2$

(c) O-elimination

Fig. 2. Eliminating substructures with ε-labeled transitions. Reverting all transitions in (a), and demanding that $q^+ = 1$ yields a further Y-rule.

E to E' with $E \Rightarrow^* E'$. An expression r over Σ is identified with the trivial EFA $A_r^0 := (\{q_0, q_f\}, \Sigma, \{(q_0, r, q_f)\}, q_0, q_f)$. On input r, the construction is initialized with A_r^0, which is successively and exhaustively converted to an ε-NFA, denoted A_r. We slightly restrict the applicability of conversions by two rules:

(R1) As long as any conversion other than $\Rightarrow_X$ is possible, X-elimination must not be applied.

(R2) If two X-anchors share ε-transitions the one from which they are leaving is to be eliminated.

Other than that, conversions may be applied in any order. Note that (R2) is sound: cyclic elimination preference among X-anchors would imply the existence of an O-anchor, which, due to (R1), would be eliminated first. The conversion process is split into a sequence of conversions without X-eliminations, followed by one with X- eliminations only. This is due to

Proposition 1. *Let $E \Rightarrow_X E'$ respect (R1). Then E' contains only X-anchors, if any.*

Proof. Since $E \Rightarrow_X E'$ respects (R1), E contains only X-anchors. Neither complex labels nor cycles, particularly no O-anchors, are introduced upon X-elimination.

Assume $E \Rightarrow_{X[q]} E' \Rightarrow_{Y[p]} E''$ is a valid conversion sequence, then p and q are adjacent in E, since the Y-anchor in E' results from the preceding X-elimination. Let (p, ε, q) be the transition connecting p and q in E, then in E', $p^+ = 2$, hence $p^- = 1$. But the in-degree of p is not changed by this X-elimination, so $p^- = 1$ in E, too. But then, E contains an Y-anchor centered in p, contradicting the assumption that the conversion respects (R1).

To designate the transition between the two phases, let A_r^k be the first EFA in the sequence $A_r^0 \Rightarrow A_r^1 \Rightarrow \cdots \Rightarrow A_r$ that allows for no conversion besides possibly X-elimination; we denote this automaton X_r. If X-elimination does not occur at all upon full conversion, then $X_r = A_r$.

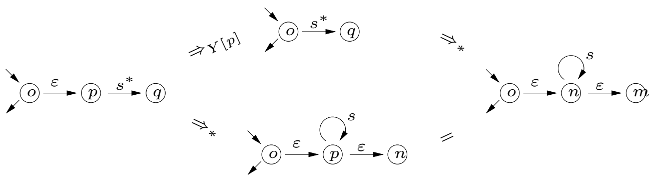
We show that the conversions other than X-elimination are locally confluent. To this end, we write $E_1 \cong E_2$, if E_1 and E_2 are joinable. Since no infinite conversion sequences are possible, Newman's Lemma implies that X_r is unique.

Theorem 2. *The replacement-system consisting of* $\Rightarrow_+$, $\Rightarrow_\bullet$, $\Rightarrow_{*i}$, $\Rightarrow_Y$ *and* $\Rightarrow_O$ *is locally confluent on the class of EFAs.*

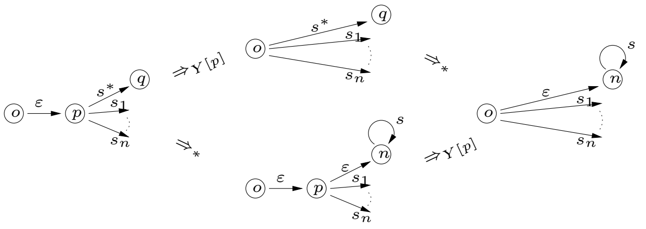
Proof. We claim that $E \Rightarrow_i E_1$ and $E \Rightarrow_j E_2$ for $i, j \in \{+, \bullet, *1, *2, *3, *4, Y, O\}$, implies $E_1 \cong E_2$. This is trivial if the conversions occur in disjoint subautomata, so assume the relevant anchors share at least a state. We assume that at least one of i, j is Y or O, the remaining cases are discussed in [14, Lem. 6]. We distinguish by i:

– $i = Y[p]$: Let (o, ε, p) be the ε-transition to be removed, and assume $\Rightarrow_j$ is an expansion, then one of the labels r_k as in Fig. 2(a) is a product, a sum or a starred expression. If r_k is a sum or a product, it is easy to see that the order of $\Rightarrow_i$ and $\Rightarrow_j$ is interchangeable. We sketch the cases involving $*$-expansion in Fig. 3. The three cases arising when both conversions are Y-eliminations, are illustrated in Fig. 4.
– $i = O$: O-elimination comes down to removing the ε-transitions forming a cycle, followed by merging the cycle-states into a selected one among them, call this the *merge-state*. If $\Rightarrow_j$ is the expansion of $t = (p, s, q)$, assume p lies on the cycle, while q does not. Choose p as the merge-state, then t remains unaffected from O-elimination, hence expansion introduces the same elements before and after O-elimination. If q is part of the cycle but not p, or $p = q$, choose q as the merge-state. If both p and q lie on the cycle and $p \neq q$, the case of $j = *4$ is detailed in Fig. 5, the remaining cases where j is an expansion are easily dealt with in the same way. Next consider the case that $\Rightarrow_j$ is $Y[q]$-elimination, for some state q, and where q is part of the ε-cycle relevant for O-elimination—the case where q is not on the ε-cycle in question would be again easy. By definition of Y-elimination, we must have $q^- = 1$ (resp. $q^+ = 1$ in the case of reverse Y-elimination), and there must be exactly one ε-transition entering (resp. leaving) the state q. Since $q^- = 1$ (resp. $q^+ = 1$), this transition is necessarily part of the ε-cycle in question. Hence, if O-elimination is applied first, it subsumes Y-elimination; otherwise, Y-elimination may be considered as the first merging step of O-elimination, followed by merging a smaller cycle.

  Finally, if $\Rightarrow_j$ also denotes O-elimination, there is at least one common state c to both cycles, which we chose as the merge-state. Regardless of the order, both cycles may be merged into c, thus yielding the same EFA. □

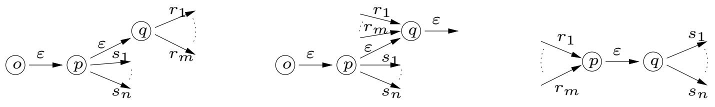


(a) Degenerate case where $p^- = p^+ = 1$; the particular type of $*$-expansion is determined by o^+ and q^-



(b) General case

Fig. 3. Local confluence of cases involving Y-elimination and $*$-expansion. The state denoted n is either q or a newly introduced state, according to q^-. Note that reverting all transitions in the figures yields further valid cases.



(a) (b) (c)

Fig. 4. Elimination-conflicts between overlapping Y-anchors centered in p and q. In (a) and (b), the resulting EFA is invariant under the order of removal. In (c) only one anchor may be eliminated, however, the resulting EFAs are isomorphic.

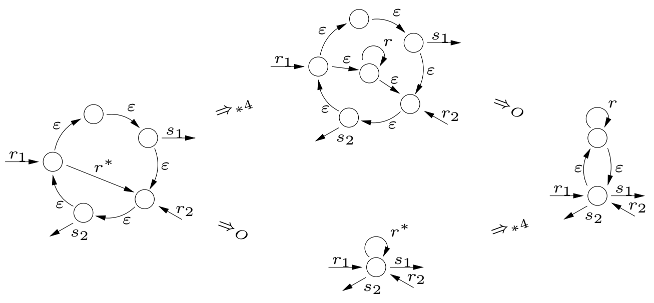


Fig. 5. Conflict between cycle-elimination and expanding a transition connecting two distinct states of the cycle

We omit proving that the conversion from X_r to A_r is also locally confluent, which is due to restriction (R2). This implies that A_r is unique, too.

We add an almost trivial linear-time preprocessing step on the input expression, called *mild simplification*: Every occurrence of $s^?$ in r, s.t. $\varepsilon \in L(s)$ is replaced with s. The expression such built from r is denoted $\mathrm{simp}(r)$, it can be computed in linear time on the parse of r in a bottom-up manner. Without proof, we mention that computing the strong star normal form subsumes mild simplification:

Lemma 3. *Let r be a regular expression, then* $\mathrm{simp}(r)^\bullet = \mathrm{simp}(r^\bullet) = r^\bullet$

On input r, we mildly simplify it first and then compute $A^0_{\mathrm{simp}(r)}$. The size $|A|$ of an EFA A is defined as the number of its states and transitions *combined*. Mild simplification is a reasonable first step in order to get smaller ε-NFAs:

Lemma 4. *For any expression r,* $|A_{\mathrm{simp}(r)}| \leq |A_r|$

Proof. Let E_1 be an EFA with transition $t = (p, s^?, q)$, and let E_2 be the EFA obtained from E_1 by replacing t with (p, s, q). Expanding t in E_1 yields E_1'; the difference between E_1' and E_2 is an additional transition (p, ε, q) in E_1'. Now p^+ and q^- are bigger in E_1' than in E_2 — if $s = t^*$, expanding (p, s, q) in E_1' introduces at least as many elements in E_2. On the other hand, removal of p or q in E_1' may result from X- or cycle-elimination, then however, Y- or cycle-elimination would be applicable in E_2. In short, converting E_1' does not yield an ε-NFA which is smaller than the one reached by converting E_2. Since mildly simplifying an expression boils down to replacing some occurrences of $s^?$ with s (in labels), the statement follows. □

The remaining part of this section deals with invariant cases of the construction under $^\circ$ and $^\bullet$. To this end, for a transition $t = (p, r, q)$ let $t^\circ := (p, r^\circ, q)$ and $t^\bullet := (p, r^\bullet, q)$. Note that since the conversions are locally confluent when respecting (R1) and (R2), $\cong$ is an equivalence relation on the class of EFAs.

Lemma 5. *Let E_1 be an EFA with looping transition $l = (q, r, q)$, and let E_2 be the EFA obtained from E_1 by replacing l with l°. Then $E_1 \cong E_2$.*

Proof. If $r \in \Sigma$ then $r = r^\circ$, satisfying the claim. Let E_1 and E_2 be as above and assume the claim is true for loops labeled s or t. Let r be

- $s + t$: l is replaced by $(q, s, q), (q, t, q)$, while l° is replaced by $(q, s^\circ, q), (q, t^\circ, q)$. By assumption, the pairs are interchangeable, hence so are l and l°
- $s^?$: l is replaced by loops $(q, \varepsilon, q), (q, s, q)$, the first of which is an ε-cycle, hence eliminated, while the second may by assumption be replaced with $(q, s^\circ, q) = (q, s^{?\circ}, q) = l^\circ$.
- s^*: $*4$-expansion is applied, introducing an ε-cycle $\{(q, \varepsilon, q'), (q', \varepsilon, q)\}$ and a loop (q', s, q'). Eliminating the cycle identifies q and q', yielding (q, s, q) which may by assumption be replaced with $(q, s^\circ, q) = l^\circ$
- st: If $\varepsilon \notin L(st)$, then $(st)^\circ = st$ and nothing needs to be proven. So assume $\varepsilon \in L(st)$, implying $\varepsilon \in L(s)$ and $\varepsilon \in L(t)$. Let E_1' be the EFA after fully expanding r, without intermediate elimination steps. The first expansion-step replaces t_l with $\{(q, s, q'), (q', t, q)\}$ — both q and q' are still present in E_1', where they lie on an

ε-cycle. Consider cycle-elimination in 'slow-motion': in a first step, only q and q' are merged, resulting in a volatile intermediate which happens to be isomorphic to the EFA constructed from fully expanding $l^\circ = (q, s^\circ + t^\circ, q)$ in E_2. A second step merges the remaining states, which is equivalent to two cycle-eliminations. □

A more general result can be established for mildly simplified expressions:

Lemma 6. *Let $A_r^0 \Rightarrow^* E_1$ for mildly simplified r. Let $t = (p, r, q)$ be any transition in E_1, and let E_2 be as E_1 except that t is replaced with $t^\bullet$. Then $E_1 \cong E_2$.*

Proof. The statement is true for letters. Assume it is true for labels s and t, and let E_1 and E_2 be as above. Let r be

- $s^?$: expansion replaces t with $\{(p, s, q), (p, \varepsilon, q)\}$, the first of which may by assumption be replaced with $(p, s^\bullet, q)$. Since r is mildly simplified, $\varepsilon \notin L(s)$ therefore $r^\bullet = s^{?\bullet} = s^{\bullet ?}$; this implies that $(p, r^\bullet, q)$ is expanded into $(p, s^\bullet, q)$ and (p, ε, q) as well.
- s^*: expanding t yields a looping transition $l = (p', s, p')$, which may by assumption be replaced with $l^\bullet$ and by Lemma 5 with $l^{\bullet\circ}$. Clearly, expanding $t^\bullet = (q, s^{\bullet\circ *}, q')$ results in $l^{\bullet\circ}$, too.

The remaining cases are straightforward. □

Theorem 3. *Let r be mildly simplified, then the ε-*NFA *constructed from r is isomorphic to the one constructed from its strong star normal form, that is, $A_r \cong A_{r^\bullet}$.*

Proof. Lemma 6 implies $A_r^0 \cong A_{r^\bullet}^0$. □

Together with Lemma 3, this shows that the construction is invariant under taking strong star normal form. Differently put, strong star normal form is implicitly computed upon conversion of mildly simplified regular expressions.

5 Alphabetic Width and the Size of ε-NFAs

Let the *size* of an ε-NFA be its number of states plus its number of transitions. The following question regarding the size of ε-NFAs was posed by Ilie and Yu.

Problem 1. Given a regular expression of alphabetic width n, what is the optimal bound on the size of an equivalent ε-NFA in terms of n?

Ilie and Yu provide a bound of $9n - \frac{1}{2}$; they remark that this does not appear to be close to optimal. The construction we discussed in the previous section was shown to give following bound in terms of rpn-length on the size of the constructed ε-NFA:

Theorem 4 ([14]). *Let r be a regular expression of unabbreviated rpn-length n. Then the constructed ε-*NFA *A_r has size at most $22/15(n+1)+1$. There are infinitely many regular languages for which this bound is tight.*

The original work does not consider abbreviated syntax for regular expressions. Fortunately, subexpressions of the form $r + \varepsilon$ do not contribute to the hardness of the conversion problem. The following bound in terms of *abbreviated* rpn-length is slightly stronger.

Theorem 5. *Let r be an expression of abbreviated rpn-length n. Then the constructed ε-NFA A_r has size at most $22/15(n+1)+1$. There are infinitely many regular languages for which this bound is tight.*

Proof. The analysis is the same as given in [14], except for obvious modifications to the proof of [14, Thm. 10], which is the only place where we take the use of abbreviated syntax into account. The fact that this bound is tight for infinitely many regular languages trivially carries over. □

Together with Thms. 1 and 3, we obtain the following upper bound:

Theorem 6. *Let r be a regular expression of alphabetic width n. If r is mildly simplified, then the constructed ε-NFA A_r has size at most $4\frac{2}{5}n+1$. There are infinitely many regular languages for which this bound is tight.*

Proof. Let r be mildly simplified with $\mathrm{alph}(r) = n$. Then Thm. 3 implies that A_r is identical to $A_{r^\bullet}$ and we know from Thm. 1 that $\mathrm{arpn}(r^\bullet) \leq 3n-1$. Plugging this into the statement of Thm. 5, it follows that the ε-NFA $A_{r^\bullet}$, constructed from $r^\bullet$, has size at most $22/15(3n-1+1)+1 = 4\frac{2}{5}n+1$.

Gulan and Fernau [14] also give an infinite family of regular expressions r_n showing that the bound $22/15(m-1)+1$ on the size of an ε-NFA equivalent to a regular expression of rpn-length m is optimal: For $k \geq 1$, they define the regular expression

$$r_k = \prod_{i=1}^{k} (a_i^* + b_i^*) \cdot (c_i^* + d_i^* + e_i^*)$$

of rpn-length $m = 15k-1$ and prove that every equivalent ε-NFA has size at least $22k+1 = 22/15(m+1)+1$. Since the alphabetic width of r_k is $\ell = 5k$, this shows that the bound of $22k+1 = 4\frac{2}{5}\ell+1$ stated in the theorem is tight for infinitely many regular languages. □

The examples from the last proof can be used to prove that the bound from Thm. 1 is tight in the abbreviated case:

Theorem 7. *There is an infinite family L_n of regular languages such that $\mathrm{alph}(L_n) \leq n$, whereas $\mathrm{arpn}(L_n) \geq 3n-1$.*

Proof. Consider the language L_n described by the expression

$$r_k = \prod_{i=1}^{k} (a_i^* + b_i^*)(c_i^* + d_i^* + e_i^*).$$

For $n = 5k$ and $L_n = L(r_{5k})$, we have $\mathrm{alph}(L_n) = 5k = n$. But the existence of an expression of abbreviated rpn-length less than $3n-1 = 15k-1$ would imply with Theorem 5 that there exists an ε-NFA of size less than $22k+1$ accepting L_n, which contradicts Thm. 4. □

6 Conclusion and Further Research

As equivalence of expressions is **PSPACE**-complete [17] and not finitely axiomatizable [1,7], a normal form that assigns a unique expression to each regular language, might be difficult to obtain. Ideally, we would like a normal form that realizes minimum alphabetic width and minimum rpn-length, and that is efficiently computable — two criteria, that would apparently contradict the above negative theoretical results.

We have suggested a robust notion of reduced expressions, the strong star normal form. This notion satisfies at least the latter two criteria, in the sense that each regular language, admits at least one regular expression in star normal form of minimum rpn-length and of minimum alphabetic width, while being computable in linear time. Our notion subsumes previous attempts at defining such a notion [4,8,15].

Furthermore, we showed that the strong star normal form proves useful in various contexts: Apart from a prior application in the context of the construction of ε-free NFAs [6], we gave two further applications.

The first concerns the relation between different complexity measures for regular expressions, namely alphabetic width and (abbreviated) rpn-length. With the aid of strong star normal form, we were able to determine the optimal bound, witnessing superiority of this concept over previous attempts at defining such a notion of irreducibility, which yield only loose bounds [8,15].

The second application concerns the comparison of descriptional complexity measures across different representations, namely alphabetic width on the one hand, and the minimum size of equivalent ε-NFAs on the other hand. Here, we applied a construction proposed recently by Gulan and Fernau [14]: Under a mild additional assumption, this construction already incorporates all simplifications offered by strong star normal form. While this alone adds to the impression of robustness of the construction, we also proved an optimal bound on the relation between alphabetic width and the size of finite automata, and we showed that this bound is attained by the mentioned construction.

We believe that there are various further applications outside the theoretical domain. For instance, the fastest known algorithm [3] for regular expression matching is still based on the classical construction due to Thompson [19]. While better constructions for ε-NFAs may not improve the asymptotic worst-case running time, we hope that these can still lead to noticeably better practical performance of NFA-based regular expression engines.

References

1. Aceto, L., Fokkink, W., Ingólfsdóttir, A.: On a question of A. Salomaa: the equational theory of regular expressions over a singleton alphabet is not finitely axiomatizable. Theoretical Computer Science 209(1), 163–178 (1998)
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
3. Bille, P., Thorup, M.: Faster regular expression matching. In: ICALP 2009. LNCS, vol. 5555, pp. 171–182. Springer, Heidelberg (2009)
4. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science 120(2), 197–213 (1993)

5. Caron, P., Champarnaud, J.M., Mignot, L.: Multi-tilde operators and their Glushkov automata. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 290–301. Springer, Heidelberg (2009)
6. Champarnaud, J.M., Ouardi, F., Ziadi, D.: Normalized expressions and finite automata. International Journal of Algebra and Computation 17(1), 141–154 (2007)
7. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall, Boca Raton (1971)
8. Ellul, K., Krawetz, B., Shallit, J., Wang, M.: Regular expressions: New results and open problems. Journal of Automata, Languages and Combinatorics 10(4), 407–437 (2005)
9. Frishert, M., Cleophas, L.G., Watson, B.W.: The effect of rewriting regular expression on their accepting automata. In: Ibarra, O.H., Dang, Z. (eds.) CIAA 2003. LNCS, vol. 2759, pp. 304–305. Springer, Heidelberg (2003)
10. Gelade, W., Martens, W., Neven, F.: Optimizing schema languages for XML: Numerical constraints and interleaving. SIAM Journal on Computing 38(5), 2021–2043 (2009)
11. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. In: Symposium on Theoretical Aspects of Computer Science. Number 08001 in Dagstuhl Seminar Proceedings, pp. 325–336 (2008)
12. Gruber, H., Holzer, M.: Finite automata, digraph connectivity, and regular expression size. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 39–50. Springer, Heidelberg (2008)
13. Gruber, H., Johannsen, J.: Optimal lower bounds on regular expression size using communication complexity. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 273–286. Springer, Heidelberg (2008)
14. Gulan, S., Fernau, H.: An optimal construction of finite automata from regular expressions. In: FSTTCS 2008. Number 08004 in Dagstuhl Seminar Proceedings, pp. 211–222 (2008)
15. Ilie, L., Yu, S.: Follow automata. Information and Computation 186(1), 140–162 (2003)
16. Lee, J., Shallit, J.: Enumerating regular expressions and their languages. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 2–22. Springer, Heidelberg (2005)
17. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: FOCS 1972, pp. 125–129. IEEE Computer Society, Los Alamitos (1972)
18. Newman, M.: On theories with a combinatorial definition of "equivalence". Annals of Mathematics 43(2), 223–243 (1942)
19. Thompson, K.: Regular expression search algorithm. Communications of the ACM 11(6), 419–422 (1968)
20. Wood, D.: Theory of Computation. John Wiley & Sons, Inc., Chichester (1987)

A Programming Language Tailored to the Specification and Solution of Differential Equations Describing Processes on Networks

Reinhard Hemmerling, Katarína Smoleňová, and Winfried Kurth

University of Göttingen,
Göttingen, Germany
{rhemmer,ksmolen,wk}@informatik.uni-goettingen.de

Abstract. We present an extension to the graph-transformation based programming language XL that allows easy specification and solution of differential equations on graphs.

1 Introduction

When making decisions on economical or ecological processes, models are used to predict the results of those decisions. To simplify the description of such models, modelling systems and modelling languages exist. One such modelling language is XL, which stands for eXtended L-system language.

XL allows a user-friendly specification of graph transformation rules. Its main purpose is to facilitate the implementation of functional-structural plant models (FSPMs, [5]), but it is general enough to also be used in other domains where the problem can be mapped to a graph, like modelling cellular networks. In each XL program run, there exists a global graph with arbitrary objects as nodes and relations of various types as edges. This graph can undergo transformations in discrete timesteps, controlled by a *relational growth grammar* (RGG) – a special sort of parallel graph grammar which can directly be written down in XL code.

An RGG formally consists of a finite family of RGG rules, together with a control flow (regulating the order of rule application) and a start graph. RGG rules are transformation rules with application conditions on typed, attributed, directed graphs with inheritance, which are applied in parallel and follow basically the single-pushout principle from categorical graph-grammar theory (see [6] for the exact definitions). RGGs allow an L-system style embedding of the right-hand side of a rule into the host graph by a special choice of connection transformation – so the application of L-systems on strings, often employed in plant models, emerges as a special case of RGG application on graphs.

When modelling biological or physical systems, the functional aspects of those systems are often described in terms of differential equations. XL extends the Java programming language, so that calculations for functional aspects of a model can be expressed using imperative programming.

Solving those differential equations often requires advanced numerical methods to calculate the correct solution and to keep the error within well-defined bounds. Using such numerical methods requires special knowledge of numerics and the model must be expressed in a suitable way. Applying the differential equations to arbitrary graphs becomes even unmanageable.

We therefore propose an extension of the XL programming language that allows easy specification of differential equations on graphs. We will demonstrate the usefulness of the approach on one example.

An XL compiler is provided by the software GroIMP[1] (Growth-grammar related Interactive Modelling Platform), which is an open-source project primarily intended for the use in research and teaching in the life sciences, but with applications in other fields as well, like, e.g., architecture [7]. GroIMP also offers interactive 3D modelling and navigation to visualize the structures modelled with the XL language. Several renderers, a 2D viewer for the complete generated graph, an attribute editor enabling interactive choice of shaders for realistic surface representation, a raytracer for radiation modelling, and various analysis tools are also included.

2 Related Work

A 3D model of the branching system of spruce crowns, generated from an L-system, has been used as input for a simulator of xylem sap flow, based on a finite-differences solver with a predictor-corrector scheme for the Darcy equation [2]. Spatial and temporal discretization was locally adapted to the expected water potential gradients. However, this simulator was restricted to a static structure; it was not possible to apply it to a growing branch system. No feedback from the simulation results to the growth process (i.e., from functioning to structure) was included. Furthermore, the implementation was an ad-hoc solution, written in the general-purpose language C and resulting in a lengthy and quite intransparent code. The motivation for and design of the language XL resulted from such experiences. It is supposed to give support for easy and transparent encoding of function-structure interactions, as they occur in dynamical systems with a dynamical structure [3].

An application of ODEs to growing structures was presented in [12] and named dL-systems (differential L-systems). The geometrical structures were described by parametric L-systems, whereas the functional aspects were described by ODEs. Basically the integration takes place until some specified boundary is hit. Then a production is executed to modify the structure and the integration continues. In the paper many examples are given, but only in the form of a formal description and not as a program. The paper states that the simulations were carried out using a programming language based on parametric L-systems, but the user had to implement the numerical integrator on its own (the forward Euler method was actually used). Yet, it was not investigated if and how more advanced integration methods could be used for dL-systems.

However, this was done in [1]. The diffusion-based developmental model of the blue-green alga *Anabaena catenula* was defined using a dL-system and solved

[1] http://sourceforge.net/projects/groimp

using an implicit Crank-Nicholson integrator. The ODEs were transformed into a banded matrix (by hand) and then mapped to an L-system string. Using some advanced features of L+C allowed to solve the linear system by two scans (forward, then backward) through the matrix. Unfortunately, the generalization of this method to arbitrary graphs is not straightforward [11,9]. Furthermore, its implementation in L+C requires a considerable technical overhead.

3 Solving Differential Equations on Graphs

In this section we will start with a short introduction to XL. We will do this on a simple example. Later on this example will be extended by differential equations and we will show how to solve these using the Runge-Kutta method. In the end we will point out the problems of the solution and suggest a better approach.

3.1 XL

The programming language XL is based on the Java programming language[4]. In fact, every valid Java program is also a valid XL program, but not the other way around. The language specification of XL[8] introduces many features not available in Java.

One of the most important extensions are transformation rules. A rule consists of a query on the left hand side, a rule arrow in the middle, and a production statement on the right hand side. A query is an expression that searches the graph for some structure. For each match the production statement is invoked to modify the structure of that match. A complete description of queries, production statements and rules can be found in [6].

Rules are organized in rule blocks. Whereas a block of Java statements uses braces { and } to mark beginning and end of the block, a block of rules uses brackets [and] instead.

An example for a simple XL program describing a binary tree is:

```
module A(float len) extends Sphere(0.1) {
    { setShader(GREEN); }
}

protected void init ()
[
    Axiom ==> A(1);
]

public void run ()
[
    A(x) ==> F(x) [RU(30) RH(90) A(x*0.8)]
                  [RU(-30) RH(90) A(x*0.8)];
]
```

The program starts with a declaration of a module `A`. This is similar to declaring a Java class, but the compiler automatically generates additional code. In this case the module expects a single parameter of type `float` and is derived from the type `Sphere`, therefore it will be displayed as a sphere. The radius of the sphere is set to 0.1 units. Inside the body of the module declaration there is a class initializer that sets the shader for the sphere to green.

After the module declaration follows the method `init`. This is called automatically by `GroIMP` when the model is started. The body of the `init` method consists of a single rule block. Inside of the block there is a single rule that replaces each node of type `Axiom` by a node of type `A` with the parameter `len` set to one. Initially there is exactly one node of type `Axiom` in the graph serving as starting symbol for the derivation.

The method `run` finally describes the derivation process to form the binary tree. Each occurrence of the type `A` in the graph is replaced by a node of type `F` (basically a cylinder) and two branches with a new node of type `A` at each end. The rotation commands `RU` and `RH` (rotate around up and head axis) are responsible for a different orientation of the two branches. When a match of the left hand side of the rule is found, the parameter `len` of the module `A` is stored in a local variable `x`. This can then be used to steer the production on the right hand side. In this case, the length of the cylinder `F` will be set to `x` and the two branches will only grow 80% as much.

The result of five derivation steps can be seen in Fig. 1.

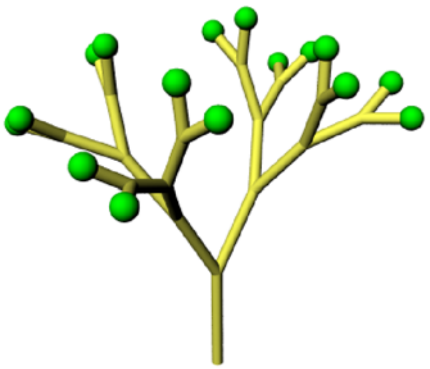


Fig. 1. A simple, growing graph-based structure

3.2 Functional Dependence of Growth

Right now the model describes only structure. For functional-structural plant models, however, also the functional aspects need to be considered. An example of such a functional process is the basipetal transport of carbon assimilates within the plant. We will extend the model from above to simulate transport of carbon via diffusion. We do not claim that this models the way transport actually works in real plants, instead it should merely demonstrate how to apply diffusion processes on a graph.

Diffusion is a process that leads to a homogeneous distribution of particles or molecules and obeys the following law (where c is the concentration, t is time, x is spatial dimension, D is the coefficient of diffusion):

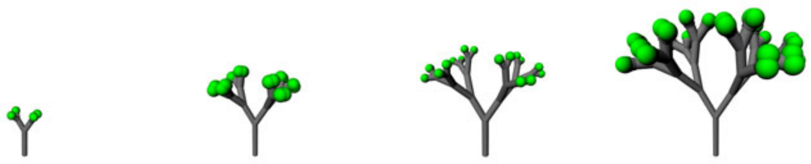
$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2}$$

To apply this equation to a graph, it must be discretized in space. Concretely this means there must be some exchange between neighboured nodes in the graph. In XL this can be descrzibed with the following rule:

```
ca:C (-->)+ : (cb:C) ::> {
    double rate = D * (cb[carbon] - ca[carbon]);
    ca[carbon] :+= h * rate;
    cb[carbon] :-= h * rate;
}
```

Basically two nodes of type `C` are searched in the graph and are named `ca` and `cb`. The two nodes must be connected, but nodes of other types (i.e. the `RU` and `RH` nodes) may be on the path from `ca` to `cb`. This is specified in the code by the transitive closure (`+`), applied to the "edge" relation (`->`) and followed by the "colon" operator which restricts the search along the path to the next occurring node of type `C`. The production statement of the rule (which does in this case not change the structure of the graph, but only updates some attributes of nodes) calculates the difference of carbon concentration and then the exchange rate. This rate is then applied to the carbon concentrations of `ca` and `cb` using a time step `h`. Note that the deferred assignment operators `:+=` and `:-=` are used to perform a parallel application of the rule.

Extending the model by this rule and also by some other rules describing carbon production in leaves (imitating photosynthesis), carbon allocation for growth and branching leads to the following sequence of a growth process (radius of cylinder or sphere indicates carbon concentration):



A closer look at the solution of the ODE from a numerical point of view reveals that an explicit Euler method [10] with fixed integration step size was implemented. This seems to give reasonable results for this simple example, but it is well-known that an explicit Euler method is not a good method to numerically integrate more complex systems of ODEs (because of stability issues for instance).

However, we discovered that users of GroIMP/XL implemented the ODEs in their models in the same or a rather similar way to the one described above. The reason therefore could be that the explicit Euler scheme naturally fits to how rules are applied to the graph: The current state is used to calculate the rate of change which is then used to modify the current state to yield the next state and therefore to advance the simulation one step in time.

3.3 Improving the Numerical Solution

In mathematical terminology, the integration problem as described above poses an *initial value problem.* An initial value problem of first order is described by an ordinary differential equation of the form $y'(t) = f(t, y(t))$ and an *initial condition* $y(t_0) = y_0$. The state y can be a single value, but more generally also a vector of values for systems of ODEs. An initial value problem of higher order can be transformed into a first order initial value problem.

Besides the explicit Euler method, more advanced methods exist to solve such a problem. A very well-known method is the *classical Runge-Kutta* method [10], which is a Runge-Kutta method of 4th order (often abbreviated as RK4). For a current state y_n at a time t_n and a step size h it calculates the new state y_{n+1} at time t_{n+1} as:

$$\begin{aligned} y_{n+1} &= y_n + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \\ t_{n+1} &= t_n + h \end{aligned}$$

where

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \ . \end{aligned}$$

The intermediate values k_i are used by the RK4 method to test the slope of the ODE for different states at different times to find a good average slope that better follows the solution compared to the Euler method.

Applying RK4 to the example from above requires to find a mapping between the properties of the nodes in the graph (like `carbon`) and the entries in the state vector. For the dynamic structure in the example, another challenging problem to solve is how to modify this mapping when the structure changes and how to adjust the state vector then.

Obviously the numerical integration and the structural modification has to be separated, so that one or more integration steps are executed alternately to one structural derivation step. For the integration, the user has to provide the rate function f, which may look like this:

```
void getRate(double[] rate, double t, double[] y) [
    ca:C (-->)+ : (cb:C) ::> {
        double rate = D * (y[cb.index] - y[ca.index]);
        rate[ca.index] += rate;
        rate[cb.index] -= rate;
    }
]
```

Here, `index` is a mapping from the node properties to the elements of the rate/state vector (`rate` and `y`). The calculation of the rates is as before, but instead of modifying the state in the rule this is up to the integrator. So instead the rates are written to the rate vector (first parameter). The time and state, for which the rates should be calculated, are passed in the second and third parameter.

As can be seen, if implemented this way the user needs to manually assign indices that refer to the value stored in the state vector. Also the node properties have to be copied into the state vector before integration and copied back into the graph afterwards, or they are kept in the state vector all the time. If the structure of the graph changes (i.e. a new node is inserted or deleted), the state and rate vector must be reallocated and the indices must be reassigned, resulting in a form of manual memory management. Instead of referring to the state via indices, it may also be provided in form of node properties (by copying the state from the state vector to the node properties before calling the rate function). But then still the calculated rates must be returned in the rate vector, requiring the rate function to know the mapping between node properties and elements of the rate vector.

3.4 Extending XL to Support ODEs

Although specification and solution of ODEs as described above is possible, it is not very user-friendly and therefore prone to errors. For instance the user could easily mix up the mapping and therefore calculate a wrong solution in the end. The task to gather information about the mapping between node properties and elements of the state vector can be automatized. But this requires the user to specify which values need to be integrated and how this should be done (i.e. how the ODE looks like).

Ideally the user should be able to specify the ODEs in a manner similar to the way described in Sect. 3.2. Our idea is to extend the programming language XL by a single new operator, the *rate assignment operator*. The specification of the diffusion process using the rate assignment operator would look like this:

```
ca:C (-->)+ : (cb:C) ::> {
    double rate = D * (cb[carbon] - ca[carbon]);
    ca[carbon] :'= +rate;
    cb[carbon] :'= -rate;
}
```

Note that the deferred assignment operators `:+=` and `:-=` have been replaced by the rate assignment operator `:'=` and that there is no step size h anymore (because this is provided by the numerical integrator and not the rate function).

Again each pair of neighboured nodes of type `C` are searched for in the graph. For each match, the rate is calculated. Then the rate assignment operator is used to specify the rate of carbon exchange for the two involved node properties.

The system works as follows. During compilation each occurrence of a rate assignment operator is analyzed. The expression on the left hand side yields the type of node (in this case `C`) and the property of this node (in this case `carbon`).

At runtime, before integration starts, the size of the state vector and also of the rate vector is calculated by searching the graph for nodes that may participate in the integration according to the information obtained at compiletime. This gives an upper bound for the size of the state and rate vector. A state and a rate vector of proper size is allocated and the current node properties (only those that may participate in integration) are copied into the state vector as initial values.

Then, when evaluating the rate function, each rate assignment causes a lookup in a table to find the correct index for a property of a specific node. The value on the right hand side of the rate assignment operator is then accumulated to the correct entry in the rate vector. If no mapping exists yet, a new index in the state/rate vector is allocated and the mapping is kept.

A runtime-library handles the task of allocating the state and rate vector and copying the state from the state vector into the properties of the nodes as well as managing the mapping between elements of the rate/state vector and node properties.

The user can therefore describe his ODEs in the form as presented above and can directly use the values of the node properties to calculate the rates.

3.5 Monitor Functions

Right now, numerical integration and modification of the graph alternates in fixed intervals. A better and more natural way to simulate the growth process would be to perform integration until some condition is fulfilled and triggers a structural modification of the graph. This also happens in real plants, for instance, when the concentration of a substance crosses a threshold and therefore causes flowering.

The concept used here is called a *monitor function* (sometimes also called *trigger function* or *switching function*). The user provides a function $g : \mathbb{R}^n \rightarrow \mathbb{R}$ that calculates a single value for a given state. When g changes its sign during the course of integration, root finding methods (like Bisection or Newton) can be applied to find the exact time when the event occurred.

Then the integration is halted and structural modifications may be applied. Afterwards, the integration continues on the new graph structure.

The language XL provides an easy way to specify functors [6]. This feature is used to install a monitor function. The syntax for the specification of a functor in XL is

```
X x => Y e
```

where `X` and `Y` are types, `x` is an identifier which can be accessed within the expression `e` as a local variable of type `X`, and the type of `e` has to be assignable to `Y`. The created functor then maps `x` to the value computed by `e`. If the type `X` is void, no `x` may be specified.

As an example consider the case that the carbon concentration of the nodes should be monitored if it rises above some threshold. Installing the monitor for every node of type `C` with a threshold of 10 would be performed with this rule:

```
c:C ::> monitor(void=>double c[carbon] - 10);
```

The functor is created to compute values of type `double`, but the current state of the graph is given implicitly when the monitor function is executed. If an action should be performed when the monitor function triggers an event, this may be passed as second parameter after the monitor function.

4 Results

We tested the ODE framework on some examples. The binary tree has already been presented above. The differential equations were used to compute carbon production at leaves (green spheres), carbon consumption at internodes (converted into growth) and exchange between internodes and/or leaves via diffusion. A monitor function is installed for every leaf to observe when the concentration reaches a threshold and triggers replacement of the leaf by two branches, each with a new leaf. The carbon of the leaf that was replaced is distributed evenly among the two new leaves.

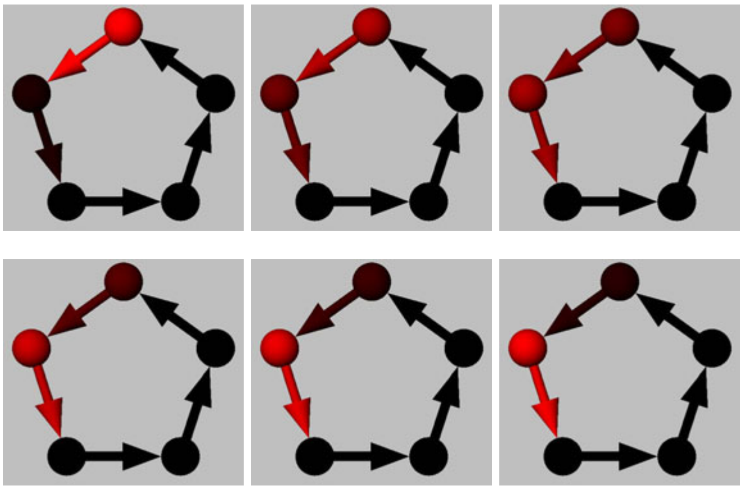


Fig. 2. From left to right, top to bottom: Transport with inhibition

Another test case is the simulation of active transport with inhibition of a substrate through a sequence of cells (see Fig. 2). The five cells are organized in a ring and transport may be performed as indicated by the arrows. Transport of the substrate from a cell B to the successor cell C may be performed, if the predecessing cell A has a substrate concentration below some threshold value.

Without the framework one would have to write the ODE by manually managing which entries in the state vector correspond to which value of a cell. This would result in an implementation of the rule with code like this:

```
for (int i = 0; i < rate.length; i++) {
    int a = i;
    int b = (a + 1) % out.length;
    int c = (b + 1) % out.length;
    double r = y[a] > 0.001 ? 0 : 0.4 * y[b];
    rate[b] -= r;
    rate[c] += r;
}
```

With the ODE framework, the same process can be implemented like this:

```
a:S -EDGE_0-> b:S -EDGE_0-> c:S ::> {
    double rate = a[s] > 0.001 ? 0 : 0.4 * b[s];
    b[s] :'= -rate;
    c[s] :'= +rate;
}
```

The nodes are connected by edges of a user-defined type EDGE_0 forming a ring structure, as was indicated by the arrows. If the concentration of substrate `s` at `a` is below 0.001, then substrate from `b` may be transported to `c` with a rate of 40% of the concentration of `s` at `b`.

Although in this example the ODE framework just makes the code shorter to write and more understandable, it becomes much more useful when simulating reaction networks, especially if those networks can be dynamic (like the binary tree above).

Another example is a model of partitioning during vegetative growth in plants by Thornley [13]. This model describes assimilation of carbon and nitrogen. While the model using RK4 (classical Runge-Kutta) was hard to implement

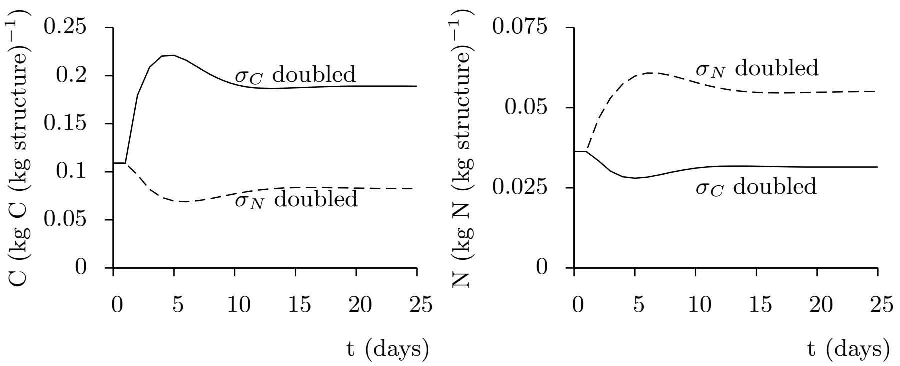


Fig. 3. Diagrams showing the effect of doubling the parameter σ_C respective σ_N at time $t = 2$ on carbon and nitrogen concentration

without the ODE framework, this became much simpler and shorter when the ODE framework was used. The resulting graphs produced by running the simulation with RK4 and plotting the state at intervals of one day can be seen in Fig. 3.

Here σ_C and σ_N are used to calculate the assimilation rate of carbon and nitrogen depending on the shoot drymass W_{sh} and root drymass W_r as $\sigma_C W_{sh}$ and $\sigma_N W_r$. The diagrams show the effect of doubling σ_C and σ_N after day one of the simulation.

5 Conclusions and Future Work

In this paper we presented an ODE framework that allows specification and integration of ODEs on graphs. We demonstrated the motivation why such a framework is needed and how it is used. The framework was tested on several examples which show that it was successfully used in practice.

The basis of the framework is the extension of the XL-compiler to capture all rate specifications with a new operator `:'=` at compile-time and a runtime-library that manages the mapping between node properties and the elements of the rate/state vector.

In the future we would like to apply the ODE framework to more complex examples like a model of a growing plant with internal reaction mechanisms steering the growth of that plant. This will then also lead to further extensions of the framework that would make it more robust and usable.

References

1. Federl, P., Prusinkiewicz, P.: Solving differential equations in developmental models of multicellular structures expressed using L-systems. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3037, pp. 65–72. Springer, Heidelberg (2004)
2. Früh, T., Kurth, W.: The hydraulic system of trees: Theoretical framework and numerical simulation. J. Theor. Biol. 201, 251–270 (1999)
3. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational Models for Integrative and Developmental Biology. In: Modelling and Simulation of Biological Processes in the Context of Genomics, Hermes (July 2002)
4. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Amsterdam (June 2005)
5. Hemmerling, R., Kniemeyer, O., Lanwert, D., Kurth, W., Buck-Sorlin, G.: The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. Functional Plant Biology 35(9/10), 739–750 (2008)
6. Kniemeyer, O.: Design and Implementation of a Graph Grammar Based Language for Functional-Structural Plant Modelling. PhD thesis, University of Technology at Cottbus, Fakultät für Mathematik, Naturwissenschaften und Informatik (2008), `http://opus.kobv.de/btu/volltexte/2009/593/`
7. Kniemeyer, O., Barczik, G., Hemmerling, R., Kurth, W.: Relational Growth Grammars - a parallel graph transformation approach with applications in biology and architecture. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 152–167. Springer, Heidelberg (2008)

8. Kniemeyer, O., Hemmerling, R., Kurth, W.: The XL Language Specification (2009), http://www.grogra.de/xlspec
9. Parter, S.: The use of linear graphs in Gauss elimination. SIAM Review 3(2), 119–130 (1961)
10. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing, 3rd edn. Cambridge University Press, New York (2007)
11. Prusinkiewicz, P., Allen, M., Escobar-Gutiérrez, A., DeJong, T.M.: Numerical methods for transport-resistance source-sink allocation models. In: Vos, J., Marcelis, L.F.M., de Visser, P.H.B., Struik, P.C., Evers, J.B. (eds.) Functional-Structural Plant Modelling in Crop Production, 1st edn. Wageningen UR Frontis Series, vol. 22. Springer, Heidelberg (2007)
12. Prusinkiewicz, P., Hammel, M.S., Mjolsness, E.: Animation of plant development. In: SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp. 351–360. ACM, New York (1993)
13. Thornley, J.H.M., Johnson, I.R.: Plant and crop modelling: a mathematical approach to plant and crop physiology. Clarendon Press/Oxford University Press, Oxford/New York (1990)

The Inclusion Problem for Regular Expressions

Dag Hovland

Institutt for Informatikk, Universitetet i Bergen, Norway
dag.hovland@uib.no

Abstract. This paper presents a new polynomial-time algorithm for the inclusion problem for certain pairs of regular expressions. The algorithm is not based on construction of finite automata, and can therefore be faster than the lower bound implied by the Myhill-Nerode theorem. The algorithm automatically discards unnecessary parts of the right-hand expression. In these cases the right-hand expression might even be 1-ambiguous. For example, if r is a regular expression such that any DFA recognizing r is very large, the algorithm can still, in time independent of r, decide that the language of ab is included in that of $(a+r)b$. The algorithm is based on a syntax-directed inference system. It takes arbitrary regular expressions as input, and if the 1-ambiguity of the right-hand expression becomes a problem, the algorithm will report this.

1 Introduction

The inclusion problem for regular expressions was shown PSPACE-complete by Meyer & Stockmeyer [10]. The input to the problem is two expressions, which we will call the *left-hand* expression and the *right-hand* expression, where the question is whether the language of the left-hand expression is included in the language of the right-hand expression. The classical algorithm starts with constructing non-deterministic finite automata (NFAs) for each of the expressions, then constructs a DFA from the NFA recognizing the language of the right-hand expression, and a DFA recognizing the complement of this language, then constructs an NFA recognizing the intersection of the language of the left-hand expression with the complement of the language of the right-hand expression, and finally checks that no final state is reachable in the latter NFA. The super-polynomial blowup occurs when constructing a DFA from the NFA recognizing the right-hand expression. A lower bound to this blowup is given by the Myhill-Nerode theorem [11,7]. All the other steps, seen separately, are polynomial-time.

1-unambiguous regular expressions were introduced by Brüggemann-Klein & Wood [3,2]. They show a polynomial-time construction of DFAs from 1-unambiguous regular expressions. The algorithm above can therefore be modified to solve the inclusion problem in polynomial time when the right-hand expression is 1-unambiguous. This paper presents an alternative algorithm for inclusion of 1-unambiguous regular expressions. As in the algorithm above, the left-hand expression can be an arbitrary regular expression. An implementation of the algorithm is available from the website of the author. The algorithm can of course also be run twice to test whether the languages of two 1-unambiguous regular expressions are equal.

A consequence of the Myhill-Nerode theorem is that for many regular expressions, the minimal DFA recognizing this language, is of super-polynomial size. For example, there are no polynomial-size DFAs recognizing expressions of the form $(b+c)^*c(b+c)\cdots(b+c)$. An advantage of the algorithm presented in this paper is that it only treats the parts of the right-hand expression which are necessary; it is therefore sufficient that these parts of the expression are 1-unambiguous. For some expressions, it can therefore be faster than the algorithm above. For example, the algorithm described in this paper will (in polynomial time) decide that the language of ab is included in that of $(a+(b+c)^*c(b+c)\cdots(b+c))b$, and the sub-expression $(b+c)^*c(b+c)\cdots(b+c)$ will be discarded. The polynomial-time algorithm described above cannot easily be modified to handle expressions like this, without adding complex and ad hoc pre-processing.

To summarize: Our algorithm always terminates in polynomial time. If the right-hand expression is 1-unambiguous, the algorithm will return a positive answer if and only if the expressions are in an inclusion relation, and a negative answer otherwise. If the right-hand expression is 1-ambiguous, three outcomes are possible: The algorithm might return a positive or negative answer, which is then guaranteed to be correct, or the algorithm might also decide that the 1-ambiguity of the right-hand expression is a problem, report this, and terminate.

Section 2 defines operations on regular expressions and properties of these. Section 3 describes the algorithm for inclusion, and Sect. 4 shows some important properties of the algorithm. The last section covers related work and a conclusion.

2 Regular Expressions

Fix an *alphabet* Σ of *letters*. Assume a, b, and c are members of Σ. $l, l_1, l_2, \ldots$ are used as variables for members of Σ.

Definition 1 (Regular Expressions). *The* regular expressions *over the language Σ are denoted R_Σ and defined in the following inductive manner:*

$$R_\Sigma ::= R_\Sigma + R_\Sigma \mid R_\Sigma \cdot R_\Sigma \mid R_\Sigma^* \mid \Sigma \mid \epsilon$$

$r, r_1, r_2, \ldots$ are used as variables for regular expressions. The sign for concatenation, $\cdot$, will often be omitted. The regular expressions denoting the empty language are not included, as they are irrelevant to the results in this paper.

The semantics of regular expressions is defined in terms of sets of words over the alphabet Σ. We lift concatenation of words to sets of words, such that if $L_1, L_2 \subseteq \Sigma^*$, then $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \land w_2 \in L_2\}$. ϵ denotes the *empty word* of zero length, such that for all $w \in \Sigma^*$, $\epsilon \cdot w = w \cdot \epsilon = w$. Therefore we also assume $r\epsilon = \epsilon r = r$ for regular expressions r. Integer exponents are short-hand for repeated concatenation of the same set, such that for a set L of words, e.g., $L^2 = L \cdot L$, and we define $L^0 = \{\epsilon\}$. $\mathsf{sym}(r)$ denotes the set of letters from Σ occurring in r.

Definition 2 (Language of a Regular Expression). *The* language *of a regular expression r is denoted $\|r\|$ and is defined by the following inductive rules: $\|r_1 + r_2\| = \|r_1\| \cup \|r_2\|$, $\|r_1 \cdot r_2\| = \|r_1\| \cdot \|r_2\|$, $\|r^*\| = \bigcup_{0 \leq i} \|r\|^i$ and for $a \in \Sigma \cup \{\epsilon\}$, $\|a\| = \{a\}$.*

All subexpressions of the forms $\epsilon \cdot \epsilon$, $\epsilon + \epsilon$ or ϵ^* can be removed in linear time, working bottom up. We therefore can safely assume there are no subexpressions of these forms. We use r^i as a short-hand for r concatenated with itself i times.

The First-set of a regular expression is the set of letters that can occur first in a word in the language, while the followLast-set is the set of letters which can follow a word in the language. An easy, linear time, algorithm for calculating the First-set has been given by many others, e.g., Glushkov [6] and Yamada & McNaughton [9].

Definition 3 (First and followLast). *[2,6,9]*

$$\mathsf{first}(r) = \{l \in \Sigma \mid \exists w : lw \in \|r\|\}$$

$$\mathsf{followLast}(r) = \{l \in \mathsf{sym}(r) \mid \exists u, v \in \mathsf{sym}(r)^* : (u \in L(r) \wedge ulv \in L(r))\}$$

Definition 4 (Nullable Expressions). *[6,9] The* nullable *regular expressions are denoted $\mathfrak{N}$ and are defined inductively as follows:*

$$\mathfrak{N} ::= \mathfrak{N} + R_\Sigma \mid R_\Sigma + \mathfrak{N} \mid \mathfrak{N} \cdot \mathfrak{N} \mid R_\Sigma^* \mid \epsilon$$

It can be proved by induction on the regular expressions, that $\mathfrak{N}$ are exactly the regular expressions that have ϵ in the language.

Definition 5 (Marked Expressions). *[6,9] If $r \in R_\Sigma$ is a regular expression, $\mu(r)$ is the marked expression, that is, the expression where every instance of any symbol from Σ is subscripted with an integer, starting with 1 at the left and increasing.*

For example, $\mu((a + b)^* a) = (a_1 + b_2)^* a_3$. The mapping $\sharp$ removes subscripts on letters, such that $\sharp(\mu(r)) = r$.

Definition 6 (Star Normal Form). *[3,2]: A regular expression is in* star normal form *iff for all subexpressions r^*: $r \notin \mathfrak{N}$ and $\mathsf{first}(\mu(r)) \cap \mathsf{followLast}(\mu(r)) = \varnothing$.*

Brüggemann-Klein & Wood described also in [3,2] a linear time algorithm mapping a regular expression to an equivalent expression in star normal form. We can therefore safely assume that all regular expressions are in star normal form.

Definition 7 (Header-form). *A regular expression is in header-form if it is of the form ϵ, $l \cdot r_1$, $(r_1 + r_2) \cdot r_3$ or $r_1^* \cdot r_2$, where $l \in \Sigma$ and $r_1, r_2, r_3 \in R_\Sigma$.*

A regular expression can in linear time be put in header-form by applying the mapping hdf. We need the auxiliary mapping header, which maps a pair of regular expressions to a single regular expression. It is defined by the following inductive rules:

$$\mathsf{header}(\epsilon, r) = r$$
$$\mathsf{header}(r_1, r_2) = \begin{cases} \text{if } r_1 \text{ is of the form } r_3 \cdot r_4 : & \mathsf{header}(r_3, r_4 \cdot r_2) \\ \text{else:} & r_1 \cdot r_2 \end{cases}$$

For any regular expression r, $\mathsf{hdf}(r) = \mathsf{header}(r, \epsilon)$ is in header-form and recognizes the same language as r. hdf also preserves star normal form, as starred subexpressions are not altered.

2.1 1-Unambiguous Regular Expressions

Intuitively, a regular expression is 1-unambiguous if there is only one way a word in its language can be matched when working from left to right with only one letter of look-ahead.

Definition 8. *[3,2] A regular expression r is 1-unambiguous if for any two $upv, uqw \in \|\mu(r)\|$, where $p, q \in \mathsf{sym}(\mu(r))$ and $u, v, w \in \mathsf{sym}(\mu(r))^*$ such that $\sharp(p) = \sharp(q)$, we have $p = q$.*

Examples of 1-unambiguous regular expressions are $(a^* + b)^*$, $a(a + b)^*$ and $b^*a(b^*a)^*$, while $(\epsilon + a)a$ and $(a + b)^*a$ are not 1-unambiguous. An expression which is not 1-unambiguous is called 1-ambiguous. A language is called 1-unambiguous if there is a 1-unambiguous regular expression denoting it. Otherwise, the language is called 1-ambiguous.

1-unambiguity is different from, though related with, *unambiguity*, as used to classify grammars in language theory, and studied for regular expressions by Book et al [1]. From [1]: "A regular expression is called unambiguous if every tape in the event can be generated from the expression in one way only". It follows almost directly from the definitions that the class of 1-unambiguous regular expressions is included in the class of unambiguous regular expressions. The inclusion is strict, as for example the expression $(a+b)^*a$ is both unambiguous and 1-ambiguous. See also [3,2] for comparisons of unambiguity and 1-unambiguity.

Brüggemann-Klein and Wood [3] showed that there exist 1-ambiguous regular languages, e.g., $\|(a+b)^*(ac+bd)\|$. They also showed that a regular expression is 1-unambiguous if and only if all of its subexpressions also are 1-unambiguous. We will use this property below. Note at this point that hdf preserves 1-unambiguity.

Taking $u = \epsilon$ in Definition 8 it follows that if $l_n, l_m \in \mathsf{first}(\mu(r))$ and r 1-unambiguous, then $n = m$. This fact is employed by the algorithm below.

3 Rules for Inclusion

The algorithm is based on an inference system described inductively in Table 1 for a binary relation $\sqsubseteq$ over regular expressions. The core of the algorithm is a goal-directed, depth first search using this inference system. We will show

Table 1. The rules for the relation $\sqsubseteq$

$$\text{(Axm)}\ \frac{}{\epsilon \sqsubseteq r}\ [r \in \mathfrak{N}] \qquad \text{(Letter)}\ \frac{r_1 \sqsubseteq r_2}{l \cdot r_1 \sqsubseteq l \cdot r_2} \qquad \text{(LetterStar)}\ \frac{l \cdot r_1 \sqsubseteq r_2 r_2^* r_3}{l \cdot r_1 \sqsubseteq r_2^* r_3}\ [l \in \mathsf{first}(r_2)]$$

$$\text{(LetterChoice)}\ \frac{l \cdot r_1 \sqsubseteq r_i r_4}{l \cdot r_1 \sqsubseteq (r_2 + r_3) r_4} \begin{bmatrix} i \in \{2,3\} \\ l \in \mathsf{first}(r_i) \end{bmatrix} \qquad \text{(LeftChoice)}\ \frac{\begin{matrix} r_1 r_3 \sqsubseteq r_4 \\ r_2 r_3 \sqsubseteq r_4 \end{matrix}}{(r_1 + r_2) r_3 \sqsubseteq r_4}$$

$$\text{(LeftStar)}\ \frac{\begin{matrix} r_1 r_1^* r_2 \sqsubseteq r_3 r_4 \\ r_2 \sqsubseteq r_3 r_4 \end{matrix}}{r_1^* r_2 \sqsubseteq r_3 r_4} \begin{bmatrix} \mathsf{first}(r_1) \cap \mathsf{first}(r_3) \neq \varnothing \\ r_4 \neq \epsilon \vee r_2 \neq \epsilon \\ \exists l, r_5 : r_3 = l \vee r_3 = r_5^* \end{bmatrix} \qquad \text{(StarStarE)}\ \frac{r_1 \sqsubseteq r_2^*}{r_1^* \sqsubseteq r_2^*}$$

$$\text{(StarChoice1)}\ \frac{r_1^* r_2 \sqsubseteq r_i r_5}{r_1^* r_2 \sqsubseteq (r_3 + r_4) r_5} \begin{bmatrix} i \in \{3,4\} \\ \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_i) \neq \varnothing \\ \mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_i r_5) \\ r_2 \notin \mathfrak{N} \vee r_i \in \mathfrak{N} \end{bmatrix}$$

$$\text{(StarChoice2)}\ \frac{\begin{matrix} r_1 r_1^* r_2 \sqsubseteq (r_3 + r_4) r_5 \\ r_2 \sqsubseteq (r_3 + r_4) r_5 \end{matrix}}{r_1^* r_2 \sqsubseteq (r_3 + r_4) r_5} \begin{bmatrix} \mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3 + r_4) \neq \varnothing \\ (r_2 \in \mathfrak{N} \wedge r_3 \notin \mathfrak{N}) \vee \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_3 r_5) \\ (r_2 \in \mathfrak{N} \wedge r_4 \notin \mathfrak{N}) \vee \mathsf{first}(r_1^* r_2) \not\subseteq \mathsf{first}(r_4 r_5) \end{bmatrix}$$

$$\text{(ElimCat)}\ \frac{r_1 \sqsubseteq r_3}{r_1 \sqsubseteq r_2 r_3} \begin{bmatrix} \exists l, r_4, r_5 : r_1 = l \cdot r_4 \ \vee \ r_1 = r_4^* r_5 \\ r_2 \in \mathfrak{N} \\ \mathsf{first}(r_1) \subseteq \mathsf{first}(r_3) \end{bmatrix}$$

later that a pair of regular expressions are in the relation $\sqsubseteq$ if and only if their languages are in the inclusion relation.

We will say that $r_1 \sqsubseteq r_2$ *holds*, if it is also true that $\|r_1\| \subseteq \|r_2\|$. Each rule consists of a horizontal line with a conclusion below it, and none, one, or two premises above the line. Some rules also have *side-conditions* in square brackets. We only allow rule instances where the side-conditions are satisfied. Note that (StarChoice1) and (LetterChoice) each have only one premise.

Figure 1 describes the algorithm for deciding inclusion of regular expressions. The algorithm takes a pair of regular expressions as input, and if it returns "Yes" they are in an inclusion relation, if it returns "No" they are not, and if it returns "1-ambiguous", the right-hand expression is 1-ambiguous. The stack T is used for a depth-first search, while the set S keeps track of already treated pairs of regular expressions.

```
Input: Two regular expressions $r_1$ and $r_2$
Output: "Yes", "No" or "1-ambiguous"
Initialize stack $\mathsf{T}$ and set $\mathsf{S}$ both consisting of pairs of regular expressions ;
push $(r_1, r_2)$ on $\mathsf{T}$;
while $\mathsf{T}$ not empty do
    pop $(r_1, r_2)$ from $\mathsf{T}$;
    if $(r_1, r_2) \notin \mathsf{S}$ then
        if $\mathsf{first}(r_1) \not\subseteq \mathsf{first}(r_2)$ or $r_1 \in \mathfrak{N} \wedge r_2 \notin \mathfrak{N}$ or $r_2 = \epsilon \wedge r_1 \neq \epsilon$ then
            return "No";
        end
        if $r_1 \sqsubseteq r_2$ matches conclusion of more than one rule instance then
            return "1-ambiguous";
        end
        add $(r_1, r_2)$ to $\mathsf{S}$;
        for all premises $r_3 \sqsubseteq r_4$ of the rule instance where $r_1 \sqsubseteq r_2$ matches
        conclusion do
            push $(\mathsf{hdf}(r_3), \mathsf{hdf}(r_4))$ on $\mathsf{T}$;
        end
    end
end
return "Yes";
```

Fig. 1. Algorithm for inclusion of regular expressions

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\mathit{Store}}{(\mathsf{Letter})\ \ a^*b^* \sqsubseteq (a+b)^*}}{(\mathsf{LetterChoice})\ \ aa^*b^* \sqsubseteq a(a+b)^*}}{(\mathsf{LetterStar})\ \ aa^*b^* \sqsubseteq (a+b)(a+b)^*}}{(\mathsf{LeftStar})\ \ aa^*b^* \sqsubseteq (a+b)^*} \quad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(\mathsf{Axm})}{(\mathsf{Letter})\ \ \epsilon \sqsubseteq (a+b)^*}}{(\mathsf{LetterChoice})\ \ b \sqsubseteq b(a+b)^*}}{(\mathsf{LetterStar})\ \ b \sqsubseteq (a+b)(a+b)^*}}{(\mathsf{StarStarE})\ \ b \sqsubseteq (a+b)^*}}{b^* \sqsubseteq (a+b)^*}}{a^*b^* \sqsubseteq (a+b)^*}$$

Fig. 2. Example usage of the inference rules to decide $a^*b^* \sqsubseteq (a+b)^*$

Figures 2 and 3 show examples of how to use the inference rules. The example noted in the introduction, deciding whether $\|ab\| \subseteq \|(a + (b + c)^*c(b + c) \cdots (b + c))b\|$ is shown in Fig. 3. Note that branches end either in an instance of the rule (Axm), usage of the store of already treated relations, or a failure. In addition to correctness of the algorithm, termination is of course of paramount importance. It is natural to ask how the algorithm possibly can terminate, when the rules $(\mathsf{LetterStar})$, $(\mathsf{LeftStar})$, and $(\mathsf{StarChoice2})$ have more complex premises than conclusions. This will be answered in the next section.

4 Properties of the Algorithm

To aid the understanding of the algorithm and the rules, Table 2 shows what rules might apply for each combination of header-forms of the left-hand and right-hand expressions. The only combinations that are never matched are when the right-hand expression is ϵ while the left-hand expression is not, and the combinations where the left-hand expression is ϵ while the right-hand is of the form $l \cdot r$. That the former are not in the inclusion relation follows from the fact that subexpressions of the forms $\epsilon \cdot \epsilon$, $\epsilon + \epsilon$ and ϵ^* are not allowed, while the latter combinations follow from that $\epsilon \notin \|l \cdot r\|$.

Table 2. The rules that might apply for any combination of header-forms of the left-hand and right-hand expressions

Left \ Right	ϵ	$l \cdot r$	$(r_1 + r_2) \cdot r_3$	$r_1^* \cdot r_2$
ϵ	(Axm)		(Axm)	(Axm)
$l \cdot r$		(Letter)	(ElimCat) (LetterChoice)	(ElimCat) (LetterStar)
$(r_1 + r_2) \cdot r_3$		(LeftChoice)	(LeftChoice)	(LeftChoice)
$r_1^* \cdot r_2$		(LeftStar)	(ElimCat) (StarChoice1) (StarChoice2)	(ElimCat) (LeftStar) (StarStarE)

4.1 Preservation of 1-Unambiguity

We must make sure that the rules given in Table 1 preserve 1-unambiguity for the *right-hand* expressions. That is, if the right-hand expression in the conclusion is 1-unambiguous, then the right-hand expression in all the premises are 1-unambiguous. For most rules we either have that the right-hand expression is the same in the premise and the conclusion, or we can use the fact that all subexpressions of a 1-unambiguous regular expression are 1-unambiguous. The latter fact was shown by Brüggemann-Klein & Wood [3]. The only remaining rule is (LetterStar), where the right-hand expression of the premise is of the form $r_1 r_1^* r_2$ and we know that $r_1^* r_2$ is 1-unambiguous. We must use the fact that all expressions are in star normal form (see Definition 6), thus $r_1 \notin \mathfrak{N}$, and $\mathsf{first}(\mu(r_1)) \cap \mathsf{followLast}(\mu(r_1)) = \varnothing$. Take u, p, q, v and w as in Definition 8, and assume (for contradiction) that $\natural(p) = \natural(q)$ and $p \neq q$. Since $r_1^* r_2$ and r_1 are 1-unambiguous and $r_1 \notin \mathfrak{N}$, we can by symmetry assume that p is from r_1 while q is from $r_1^* r_2$. This is only possible if $u \in \|\mu(r_1)\|$, $p \in \mathsf{followLast}(\mu(r_1))$, and q corresponds to a member of $\mathsf{first}(\mu(r_1))$ or of $\mathsf{first}(\mu(r_2))$. But since $\mathsf{first}(\mu(r_1)) \cap \mathsf{followLast}(\mu(r_1)) = \varnothing$, this means that also $r_1^* r_2$ is 1-ambiguous, which is a contradiction.

Secondly, we must substantiate the claim that if the side-conditions of more than one applicable rule hold, the right-hand expression is 1-ambiguous.

Lemma 1. *For any two regular expressions r_1 and r_2, where r_2 is 1-unambiguous, there is at most one rule instance with $r_1 \sqsubseteq r_2$ in the conclusion.*

Proof. This is proved by comparing each pair of rule instances of rules occurring in Table 2 and using Definition 8. For each case, we show that the existence

$$\dfrac{\dfrac{\dfrac{\dfrac{(\mathsf{Axm})}{(\mathsf{Letter})\ \epsilon \sqsubseteq \epsilon}}{(\mathsf{Letter})\ b \sqsubseteq b}}{(\mathsf{LetterChoice})\ ab \sqsubseteq ab}}{ab \sqsubseteq (a + (b+c)^*c(b+c)\cdots(b+c))b} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{Store}{(\mathsf{Letter})\ (ab)^*a \sqsubseteq a(ba)^*}}{(\mathsf{LetterStar})\ b(ab)^*a \sqsubseteq ba(ba)^*}}{(\mathsf{Letter})\ b(ab)^*a \sqsubseteq (ba)^*}}{(\mathsf{LeftStar})\ ab(ab)^*a \sqsubseteq a(ba)^*} \quad \dfrac{\dfrac{(\mathsf{Axm})}{(\mathsf{Letter})\ \epsilon \sqsubseteq (ba)^*}}{a \sqsubseteq a(ba)^*}}{(ab)^*a \sqsubseteq a(ba)^*}$$

Fig. 3. Example usages of the inference rules

of several rule instances with the same conclusion implies either that the right hand expression is 1-ambiguous, or that the side-conditions do not hold.

- The only rules of which there can be several instances with the same conclusion are (StarChoice1) and (LetterChoice). For (LetterChoice), the conclusion is of the form $l \cdot r_1 \sqsubseteq (r_2 + r_3) \cdot r_4$, and the existence of two instances implies that $l \in \mathsf{first}(r_2) \cap \mathsf{first}(r_3)$. This can only be the case if the right-hand expression is 1-ambiguous. For (StarChoice1), the conclusion is of the form $r_1^* r_2 \sqsubseteq (r_3 + r_4)r_5$, and the existence of two instances of this rule would imply that $\mathsf{first}(r_1^* r_2)$ and $\mathsf{first}(r_4)$ have a non-empty intersection, which furthermore is included in the first-set of $r_3 r_5$. The expression $(r_3 + r_4)r_5$ is therefore 1-ambiguous.
- If instances of both (ElimCat) and either (LetterStar) or (LetterChoice) match the pair of expressions, then the right-hand expression is of the form $r_2 r_3$, where $r_2 \in \mathfrak{N}$ and there is an l such that both $l \in \mathsf{first}(r_2)$ and $l \in \mathsf{first}(r_3)$. Therefore $r_2 r_3$ is 1-ambiguous.
- If instances of both (ElimCat) and (LeftStar) match the pair of expressions, then the relation is of the form $r_1^* r_2 \sqsubseteq r_3 r_4$, where $r_3 \in \mathfrak{N}$ and both $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_4)$ and $\mathsf{first}(r_3) \cap \mathsf{first}(r_1) \neq \varnothing$. This can only hold if $r_3 r_4$ is 1-ambiguous.
- If instances of both (ElimCat) and either (StarChoice1) or (StarChoice2) had the same conclusion, then this conclusion is of the form $r_1^* r_2 \sqsubseteq (r_3 + r_4)r_5$, where $r_3 + r_4 \in \mathfrak{N}$ and both $\mathsf{first}(r_1^* r_2) \subseteq \mathsf{first}(r_5)$ and $\mathsf{first}(r_1^* r_2) \cap \mathsf{first}(r_3 + r_4) \neq \varnothing$. Therefore the right-hand expression $(r_3 + r_4)r_5$ is 1-ambiguous.
- It is not possible that instances of (ElimCat) and (StarStarE) have the same conclusion, because that would mean that r_3 in (ElimCat) would be ϵ, and that cannot satisfy the side-conditions of (ElimCat).
- It is neither possible to instantiate (LeftStar) and (StarStarE) with the same expressions below the line, as this would not satisfy the side-conditions of (LeftStar).
- Finally, it is not possible to instantiate (StarChoice1) and (StarChoice2) with the same expressions below the line. The two last lines in the side-conditions of both rules prevent this.

4.2 Invertibility of the Rules

It is now not hard to prove that each of the rules given in Table 1 are *invertible*, in the sense that, for each rule instance, assuming that (1) the side-conditions hold and (2) no other rule instance matches the conclusion, then the conclusion holds if and only if the conjunction of the premises hold.

Proof

- For (Axm), we only note that the side-condition is that the right-hand expression is nullable, and then $\{\epsilon\}$ is of course a subset of the language. The absence of any premises is to be treated as an empty conjunction, which is always true.
- For (Letter) we are just adding (removing) a single letter prefix to (from) both languages, and this preserves the inclusion relation.
- For (LetterStar), the conclusion is of the form $lr_1 \sqsubseteq r_2^* r_3$. We note first that $\|r_2 r_2^* r_3\| \subseteq \|r_2^* r_3\|$, and therefore the premise implies the conclusion. For the other direction, note that since $l \in \mathsf{first}(r_2)$ and (ElimCat) does not match the conclusion, the l in r_2 must be the position used to match the first l in a word, and the premise must therefore also hold.
- For (LetterChoice), that the premise implies the conclusion follows from Definition 2, while showing the other direction depends on the fact that no other instance of (LetterChoice) nor (ElimCat) match the conclusion. The latter implies then that $l \notin (\mathsf{first}(r_{5-i}) \cup \mathsf{first}(r_4))$, so we must have the premise.
- For (LeftChoice), the implications follow from Definition 2.
- (LeftStar) and (StarChoice2) hold by Definition 2, as $\|r_1^* r_2\| = \|r_1 r_1^* r_2\| \cup \|r_2\|$.
- For (StarStarE), note $\|r_1\| \subseteq \|r_1^*\|$. So, obviously, if $r_1^* \sqsubseteq r_2^*$, then also $r_1 \sqsubseteq r_2^*$. The other direction holds by first seeing that $\|r_1\| \subseteq \|r_2^*\|$ implies $\|r_1^*\| \subseteq \|r_2^{**}\|$, and secondly that $\|r_2^{**}\| = \|r_2^*\|$. Both are standard results from language theory.
- For (StarChoice1), that $\|r_1^* r_2\| \subseteq \|r_i r_5\|$ implies $\|r_1^* r_2\| \subseteq \|(r_3 + r_4) r_5\|$, when $i \in \{3, 4\}$, follows from Definition 2. The other direction follows from the assumption that no other rule instance matches the conclusion, combined with the third side-condition, which together imply that $\|r_1^* r_2\| \cap \|r_{7-i} r_5\| = \varnothing$.
- For (ElimCat), the fact that the premise implies the conclusion, can be seen using Definition 2 and $r_2 \in \mathfrak{N}$. For the other direction, note that since no other rule instance matches the conclusion $r_1 \sqsubseteq r_2 r_3$, and since $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_3)$, we must have $\mathsf{first}(r_1) \cap \mathsf{first}(r_2) = \varnothing$. Therefore $\|r_1\| \cap \|r_2 r_3\| = \|r_1\| \cap \|r_3\|$, and we get that $\|r_1\| \subseteq \|r_2 r_3\|$ implies $\|r_1\| \subseteq \|r_3\|$. □

Invertibility implies that, at any point during an execution of the algorithm, the pair originally given as input is in the inclusion relation if and only if all the pairs in both the store S and the stack T are in the inclusion relation. These properties are used in the proofs of soundness and completeness below.

4.3 Termination and Polynomial Run-Time

The algorithm always terminates in polynomial time. Termination is guaranteed by two properties. First, the use of the store S means that any pair of regular expressions is treated at most once. Secondly, all regular expressions occurring in conclusions are either ϵ or of the form $r_1 \cdot r_2$, where r_1 is a subexpression of the corresponding expression input to the algorithm, while r_2 is unique for each r_1. Both properties can be shown by induction on the steps in an execution of the algorithm.

Since a regular expression has only a quadratic number of subexpressions, then the number of possible different rule instances in a run of the algorithm is $O(n^4)$, where n is the sum of the length of the regular expressions input to the algorithm. Since the work at each rule instance is polynomial in the size of the input to the algorithm, we get a polynomial run-time for the whole algorithm.

4.4 Soundness and Completeness

The only obstacle to showing soundness of the algorithm, is to show that our usage of the store is safe. Most critical is the use of the store to eliminate loops. To get an intuition as to why this is safe, we refer the reader to the right hand example in Fig. 3. Note that the conclusion holds if and only if $\forall i, i \geq 0$: $\|ab\|^i\{a\} \subseteq \|a(ba)^*\|$ This can be proved by an induction on i. The right-hand branch in Fig. 3 corresponds to the base case $i = 0$. And we get the induction case by taking the left-hand branch and replacing the $*$ in the left-hand expressions by ab repeated $i - 1$ times. We will use a similar observation to show that the use of the store is safe.

We model an execution of the algorithm as a directed tree. The internal nodes in this tree are rule instances, and the leaves are pairs where the first element is a pair of regular expressions and the second element is either (Axm), *Store* or *Fail*. Each node has, for each premise, an edge going either to a node with that conclusion, or to a leaf containing the corresponding pair of regular expressions.

With a *loop* in an execution of the algorithm, we mean a directed path in its tree, the start being an internal node and the end a leaf containing *Store*, such that the conclusion in the rule instance in the first node, corresponds to the pair of regular expressions in the leaf. The intuition is that this path would have been repeated indefinitely, *looped*, if the store S had not prevented it.

Let the *size* of a regular expression be the sum of the number of letters and operators $*$ and $+$ occurring in the expression. We will say that a rule instance in a directed path is *left-increasing* or *right-increasing*, respectively, if the left-hand or right-hand expression in the conclusion has smaller size than the corresponding expression in the next node in the loop. *Left-decreasing* and *right-decreasing* instances are defined similarly.

Instances of (StarChoice2) and (LeftStar) are either left-increasing or left-decreasing, while an instance is right-increasing if and only if it is an instance of (LetterStar). Instances of all other rules, except (Axm) and (ElimCat) are always left-decreasing, right-decreasing, or both. An instance which is neither left-increasing nor left-decreasing has the same expression on the left-hand side in the conclusion and the premise corresponding to the next node in the path. Except for certain instances of (ElimCat), the same holds for the right-hand side.

Lemma 2. *In any loop, there is at least one right-increasing and one left-increasing instance.*

The proof is omitted for space considerations.

Remark at this point, that only the rules (LeftStar) and (StarChoice2) can have premises not containing starred sub-expressions which are in the left-hand expression of the conclusion. Thus, given a tree modeling an execution of the algorithm, in any directed path starting at a node where the left-hand expression has a subexpression r_1^* and going to a node where the left-hand expression does not contain such a subexpression, there is a left-decreasing instance where the conclusion has left-hand side $r_1^* r_2$ for some r_2.

Theorem 1 (Soundness). $(r_1 \sqsubseteq r_2) \Rightarrow \|r_1\| \subseteq \|r_2\|$

Proof. Assume a successful execution of the algorithm. Since the rules are invertible, and the base case (Axm) holds by definition of $\mathfrak{N}$, we only need to show that the usage of the store S was sound. The store is used in two different situations. The cases where the pair was added to the store in a different branch hold because the rules are used depth-first. The other cases correspond to the loops. From Lemma 2, every such leaf has a left-increasing parent node. If we can show that the conclusion $r_1^* r_2 \sqsubseteq r_3$ of these left-increasing nodes are true, we are done. We only need to show that for any $i > 0$, the branch rooted in the child (in the loop) of this node can be used to show that $\|r_1\|^i \|r_2\| \subseteq \|r_3\|$. This can be done by replacing $r_1 r_1^* r_2$ by $r_1^i r_2$ in the conclusion. The steps in the branch can be used in a similar way, except that the loop(s) will be *unrolled* at most $i-1$ times, and that at least $i-1$ left-increasing instances will be removed together with the subbranches corresponding to the premises with smaller left-hand expressions. At the ith minimal left-increasing instance we get that the conclusion is the same as the premise with the smaller left-hand expression, and can be treated by the corresponding branch. □

Theorem 2 (Completeness). *If* $\|r_1\| \subseteq \|r_2\|$*, the algorithm will either accept* $r_1 \sqsubseteq r_2$*, or it will report that the 1-ambiguity of* r_2 *is a problem.*

Proof. Since the rules are invertible and the algorithm always terminates, all that remains is to show that for all regular expressions r_1 and r_2, where their languages are in an inclusion relation, there is at least one rule instance with conclusion $r_1 \sqsubseteq r_2$. This is done by a case distinction on the header-forms of r_1 and r_2, using Tables 1 and 2 and Definitions 2 and 3, and noting that $\|r_1\| \subseteq \|r_2\|$ implies $\mathsf{first}(r_1) \subseteq \mathsf{first}(r_2)$. □

5 Related Work and Conclusion

Hosoya et al [8] study the inclusion problem for XML Schemas. They also use a syntax-directed inference system, but the algorithm is not polynomial-time. Salomaa [12] presents an axiom systems for equality of regular expressions, but does not treat the run-time of doing inference in the system. The inference system used by our algorithm has some inspiration from the concept of derivatives of regular expressions, first defined by Brzozowski [4]. Chen & Chen [5] describe an algorithm for inclusion of 1-unambiguous regular expressions, which is based on derivatives, and which has some similarities with the algorithm presented in the present paper. They do not treat the left-hand and right-hand together in the way that the rules of the algorithm in this paper does. The analysis of their algorithm depends on both the left-hand and the right-hand regular expressions being 1-unambiguous.

We have described a polynomial-time algorithm for inclusion of regular expressions. The algorithm is based on a syntax-directed inference system, and is guaranteed to give an answer if the right-hand expression is 1-unambiguous. If the right-hand expression is 1-ambiguous the algorithm either reports an error or gives the answer. In addition, unnecessary parts of the right-hand expression are automatically discarded. This is an advantage over the classical algorithms for inclusion. An implementation of the algorithm is available on the author's website.

References

1. Book, R., Even, S., Greibach, S., Ott, G.: Ambiguity in graphs and expressions. IEEE Transactions on Computers c-20(2), 149–153 (1971)
2. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science 120(2), 197–213 (1993)
3. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and Computation 140(2), 229–253 (1998)
4. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM 11(4), 481–494 (1964)
5. Chen, H., Chen, L.: Inclusion test algorithms for one-unambiguous regular expressions. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigün, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 96–110. Springer, Heidelberg (2008)
6. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys 16(5), 1–53 (1961)
7. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
8. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)
9. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Transactions on Electronic Computers 9, 39–47 (1960)
10. Meyer, A.R., Stockmeyer, L.J.: The equivalence problem for regular expressions with squaring requires exponential space. In: Proceedings of FOCS, pp. 125–129. IEEE, Los Alamitos (1972)
11. Nerode, A.: Linear automaton transformations. Proceedings of the American Mathematical Society 9(4), 541–544 (1958)
12. Salomaa, A.: Two complete axiom systems for the algebra of regular events. J. ACM 13(1), 158–169 (1966)

Learnability of Automatic Classes

Sanjay Jain[1,⋆], Qinglong Luo[1], and Frank Stephan[2,⋆⋆]

[1] Department of Computer Science,
National University of Singapore, Singapore 117417, Republic of Singapore
sanjay@comp.nus.edu.sg, luoqingl@comp.nus.edu.sg

[2] Department of Mathematics and Department of Computer Science,
National University of Singapore, Singapore 117543, Republic of Singapore
fstephan@comp.nus.edu.sg

Abstract. The present work initiates the study of the learnability of automatic indexable classes which are classes of regular languages of a certain form. Angluin's tell-tale condition characterizes when these classes are explanatorily learnable. Therefore, the more interesting question is when learnability holds for learners with complexity bounds, formulated in the automata-theoretic setting. The learners in question work iteratively, in some cases with an additional long-term memory, where the update function of the learner mapping old hypothesis, old memory and current datum to new hypothesis and new memory is automatic. Furthermore, the dependence of the learnability on the indexing is also investigated. This work brings together the fields of inductive inference and automatic structures.

1 Introduction

Consider the following scenario for learning. A learner is receiving, one piece at a time, data about a target concept. As the learner is receiving the data, it conjectures its hypothesis about what the target concept might be. The hypothesis may be modified/changed as more data is received. One can consider the learner to be successful if the sequence of hypotheses converges to a correct hypothesis which explains the target concept. This is essentially the model of explanatory learning proposed by Gold [11].

The concept classes of interest to us in this paper are the classes of languages (a language is a subset of Σ^*, for some finite alphabet Σ). The data provided to the learner then becomes a sequential presentation of all the elements of the target language, in arbitrary order, with repetition allowed. Such a presentation of data is called a *text* for the language. Note that in a text, only positive data is presented to the learner, and negative data is not given. If both positive and negative data are presented to the learner, then the mode of presentation is called *informant*. Here we will only be concentrating on learning from texts.

⋆ Supported in part by NUS grant number R252-000-308-112.

⋆⋆ Supported in part by NUS grant numbers R252-000-308-112 and R146-000-114-112.

In most cases, one considers only recursive learners. The hypotheses produced by the learner describe the language to be learnt in some form, for example, they are grammars generating the language. The learner is said to **Ex**-learn the target language if the sequence of hypotheses converges to one correct hypothesis describing the language to be learnt (here "**Ex**-learn" stands for explanatory learning). Learning of one language L is not interesting, as the learner might ignore all inputs and always output the same hypothesis which is correct for L. Thus, what is considered is whether all languages from a class of languages are **Ex**-learnt by some particular learner. Finally, a class of languages is **Ex**-learnable if some learner **Ex**-learns it.

Since Gold, several other models of learning have been considered by the researchers. For example, in behaviourally correct learning (**BC**-learning) [6] the learner is not required to converge syntactically to one correct hypothesis, but it is just required that all hypotheses are correct from some time onwards. Besides the mode of convergence, researchers have also considered several properties of learner such as *consistency*, where the hypothesis of the learner is required to contain the elements seen in the input so far (see [5,7]), *conservativeness*, where the learner is not allowed to change a hypothesis which is consistent with the data seen so far (see [2]) and *iterativeness*, where the new hypothesis of the learner depends only on the previous hypothesis and the latest datum (see [21,22]) (iterative learning is often also called incremental learning). The formal definitions of the above criteria are given in Section 2 below.

Besides considering models of learning, there has also been interest in considering learning of practical and concrete classes such as pattern languages [1,15], elementary formal systems [20] and the class of regular languages [4]. As the class of all regular languages is not learnable from positive data [11], Angluin [3] initiated the study of the learnability of certain subclasses of regular languages from positive data. In particular, she showed the learnability of the class of k-reversible languages. These studies were later extended [9,12]. The classes considered in these studies were all superclasses of the class of all 0-reversible languages, which is not automatic; for example, every language $\{0,1\}^*\{2\}^n\{0,1\}^*$ is 0-reversible but the class of these languages is not automatic.

In this work, we consider those subclasses of regular languages where the membership problem is regular in the sense that one automaton accepts a combination (called "convolution") of an index and a word if and only if the word is in the language given by the index. This is formalized in the framework of automatic structures [8,13,14,19]. Here are some examples of automatic classes:

- The class of sets with up to k elements for a constant k;
- The class of all finite and cofinite subsets of $\{0\}^*$;
- The class of all intervals of an automatic linear order on a regular set;
- Given an automatic presentation of the ordered group of integers $(\mathbb{Z}, +, <)$ and a first-order formula $\Phi(x, a_1, \ldots, a_n)$ with parameters $a_1, \ldots, a_n \in \mathbb{Z}$, the class consisting of all sets $\{x \in \mathbb{Z} : \Phi(x, a_1, \ldots, a_n)\}$ with $a_1, \ldots, a_n \in \mathbb{Z}$.

It is known that the automatic relations are closed under first-order theory, as proven by Khoussainov and Nerode [14]. This makes several properties of such classes regular and thus decidable; it also makes it possible to define learners using first-order definitions. Studies in automatic structures have connections to model checking and Boolean algebra [8,14].

A tell-tale set for a language L in a class $\mathcal{L}$ is a finite subset D of L such that, for every $L' \in \mathcal{L}$, $D \subseteq L' \subseteq L$ implies $L' = L$. A class $\mathcal{L}$ satisfies Angluin's tell-tale condition iff every language L in $\mathcal{L}$ has a tell-tale set (with respect to $\mathcal{L}$). Angluin [2] showed that any class of languages which is learnable (even by a non-recursive learner, for which **Ex**-learning and **BC**-learning are the same) must satisfy Angluin's tell-tale condition. We show in Theorem 6 that every automatic class that satisfies Angluin's tell-tale condition is **Ex**-learnable by a recursive learner which is additionally consistent and conservative. Additionally, it is decidable whether an automatic class satisfies Angluin's tell-tale condition and thus whether it is **Ex**-learnable (see Corollary 7).

As we are considering learning of automatic structures, it is natural to also consider learners which are simpler than just being recursive. A natural idea would be to consider learners which are themselves described via automatic structures. This would put both, the learners and the classes to be learnt into a unified framework. Furthermore, the learners will be linear time computable and additional constraints on the memory of the whole process can be satisfied. This approach is justified by the observation that a learner might observe much more data than it can remember and therefore it is not realistic to assume that the whole learning history can be remembered. To model the above, we consider variants of iterative learners [21,22] and learners with bounded long-term memory [10]. The basic idea is that the learner reads in each round a datum and updates the long term memory and the hypothesis based on this datum; for automatic learners, this update function is then required to be automatic.

As automatic structures are relatively simple to implement and analyze, it is interesting to explore the capabilities of such learners. In Section 3 we formally define automatic learners (iterative learners as well as iterative learners with long-term memory). Specifically we consider the following bounds on memory: memory bounded by the size of the hypothesis, memory bounded by the size of the largest word seen in the input so far, besides the default cases of no memory (iterative learning) and the case where we do not put any specific bounds on memory except as implicit from the definition of automatic learners. Theorem 11 shows that there are automatic classes which are **Ex**-learnable (even iteratively) but not learnable by any automatic learners (with unconstrained long-term memory except implicitly due to the definition of automatic learner).

In Section 3 we show the relationship between various iterative automatic learners and iterative automatic learners with long-term memory. For example, if long-term memory is not explicitly bounded, then automatic **Ex**-learning is the same as automatic **BC**-learning (in contrast to the situation in learning of recursively enumerable languages by recursive learners, where there is a difference [6]). Additionally, for **BC**-learning, different bounds on long-term memory do not make a difference, as all automatically **BC**-learnable classes (with no explicit long-term memory bound) are iteratively automatically **BC**-learnable (see Theorem 12). However, for explanatory learning there is a difference if one considers long-term memory bounded by hypothesis size, or whether long-term memory is bounded by the size of maximum word seen in the input so far (see Theorem 13). It is open at this point whether long-term memory size bounded by the longest word seen so far is equivalent to there being no explicit bound on long-term memory. Furthermore, it is open whether maximum word-size memory can simulate hypothesis-size memory.

In Section 4 we consider consistent learning by automatic learners. Unlike Theorem 6, where we saw that general learners for automatic classes can be made consistent, automatic learners cannot in general be made consistent. Theorem 16 shows that there is an automatic class $\mathcal{L}$ which is **Ex**-learnable by an automatic iterative learner but not **Ex**-learnable by a consistent automatic learner (with no constraints on long-term memory, except that implicit due to the learner being automatic). Theorem 17 shows that there is an automatic class $\mathcal{L}$, which is **Ex**-learnable by a consistent automatic learner or an iterative automatic learner, but not by a consistent iterative learner. Theorem 18 shows the existence of an automatic class $\mathcal{L}$ which is **Ex**-learnable by a consistent and iterative automatic learner using a class comprising hypothesis space (i.e., using hypotheses from an automatic class which is a superset of the class $\mathcal{L}$), but not **Ex**-learnable by a consistent automatic learner (with no constraints on long-term memory, except that implicit due to the learner being automatic) using a class preserving hypothesis space (i.e., using a hypothesis space which contains languages only from $\mathcal{L}$).

One of the reasons for the difficulty of learning by iterative learners is that they forget past data. An attempt to overcome this is to require that every datum appears infinitely often in the text — such a text is called a *fat text* [18]. Fat texts are quite frequently studied in learning theory. In Section 5 we investigate the natural question of whether requiring fat texts permits to overcome the limitations of iterative learning and related criteria. In Theorem 21 we show that every automatic class that satisfies Angluin's tell-tale condition is **Ex**-learnable (using the automatic class itself as the hypothesis space) from fat texts by an automatic learner with long-term memory bounded by the size of the largest word seen so far. If one allows an arbitrary class preserving hypothesis space, then one can even do **Ex**-learning in the above case by iterative automatic learners and no additional long-term memory besides the hypothesis is needed.

In Theorem 22, we show the existence of automatic classes which are automatically iteratively learnable (even from normal texts) using a class preserving hypothesis space, but not conservatively iteratively learnable using a class preserving hypothesis space (even by arbitrary recursive learners) on a fat text.

Partial identification is a very general learning criterion, where one requires that some fixed correct hypothesis is output infinitely often by the learner while all other hypotheses are output only finitely often [18]. In Theorem 24 we show that every automatic class is partially learnable by an automatic iterative learner. This corresponds to the result by Osherson, Stob and Weinstein [18] that the whole class of all recursively enumerable languages is partially learnable by some recursive learner.

2 Preliminaries

Let $\mathbb{N}$ denote the set of natural numbers. $\mathbb{Z}$ denotes the set of integers. An alphabet Σ is any non-empty finite set. Σ^* is the set of all strings (words) over the alphabet Σ. ϵ denotes the empty string. A string of length n over Σ will be treated as a function from the set $\{0, 1, 2, \ldots, n-1\}$ to Σ. A language is a subset of Σ^* and a class is a set of languages. The relation $x <_{ll} y$ denotes that x is length lexicographically before y. In the present work we will only consider classes of regular sets. Furthermore, Σ will always refer to the alphabet on which languages and language classes are defined.

An *indexing* of a class $\mathcal{L}$ is a sequence of sets L_α with $\alpha \in I$, for some domain I, such that $\mathcal{L} = \{L_\alpha : \alpha \in I\}$. Often we will refer to both, the class and the indexing, as $\{L_\alpha : \alpha \in I\}$, where the indexing is implicit. The I above is called the set of legal indices. We will always assume that the indices in I are taken as words over an alphabet and we usually denote this alphabet with the letter Γ.

Now we consider notions related to automatic structures. Let $n > 0$ and $\Sigma_1, \Sigma_2, \ldots, \Sigma_n$ be alphabets not containing #. Let $x_1 \in \Sigma_1^*, x_2 \in \Sigma_2^*, \ldots, x_n \in \Sigma_n^*$ be given. Let $\ell = \max\{|x_1|, |x_2|, \ldots, |x_n|\}$ and let $y_i = x_i \#^{\ell - |x_i|}$. Define z to be a string of length ℓ such that $z(j)$ is the symbol made up of the j-th symbols of the strings $y_1, y_2, \ldots, y_n$: $z(j) = (y_1(j), y_2(j), \ldots, y_n(j))$, where $z(j)$ is a symbol in the alphabet $(\Sigma_1 \cup \{\#\}) \times (\Sigma_2 \cup \{\#\}) \times \ldots \times (\Sigma_n \cup \{\#\})$. We call z the *convolution of* $x_1, x_2, \ldots, x_n$ and denote it as $\mathrm{conv}(x_1, x_2, \ldots, x_n)$. Let $R \subseteq \Sigma_1^* \times \Sigma_2^* \times \ldots \times \Sigma_n^*$. We call the set $S = \{\mathrm{conv}(x_1, x_2, \ldots, x_n) : (x_1, x_2, \ldots, x_n) \in R\}$, the *convolution of* R. Furthermore, we say that R is *automatic* if and only if the convolution of R is regular. An indexing $\{L_\alpha : \alpha \in I\}$ is *automatic* if and only if I is regular and $E = \{(\alpha, x) : x \in L_\alpha, \alpha \in I\}$ is automatic. A class is *automatic* if and only if it has an automatic indexing.

The following Fundamental Theorem of Automatic Structures is useful to define automatic learners and to decide the learnability of automatic classes.

Fact 1 (Khoussainov, Nerode [14]). *Any relation that is first-order definable from existing automatic relations is automatic.*

Next, we recall a few definitions of learning, followed by a result from Angluin [2] that characterizes learnable classes. For any alphabet Σ, Γ, we let $\square$ be a special character not in Σ^* which is called the *pause symbol* and ? be a special character not in Γ^* which is called the *no-conjecture symbol.* Let Σ be the alphabet over which languages are being considered. We use σ, τ to denote finite sequences over $\Sigma^* \cup \{\square\}$ and T to denote infinite sequences over $\Sigma^* \cup \{\square\}$. Furthermore, λ denotes the empty sequence. The length of a sequence σ is denoted by $|\sigma|$. $T[m]$ denotes the initial segment of T of length m. $\sigma \diamond \tau$ (respectively, $\sigma \diamond T$) denotes the concatenation of σ and τ (respectively, σ and T). Furthermore, $\sigma \diamond x$ denotes the concatenation of a sequence σ with a sequence containing just x. We let $\mathrm{cnt}(\sigma) = \{x \in \Sigma^* : \exists\, n < |\sigma|\, (\sigma(n) = x)\}$ and $\mathrm{cnt}(T) = \{x \in \Sigma^* : \exists\, n \in \mathbb{N}\, (T(n) = x)\}$. For every set L and every infinite sequence T over $\Sigma^* \cup \{\Box\}$ with $L = \mathrm{cnt}(T)$, we call T a *text for* L. For every $L \subseteq \Sigma^*$, let $\mathrm{txt}(L) = \{T \in (\Sigma^* \cup \{\Box\})^\omega : \mathrm{cnt}(T) = L\}$ and $\mathrm{seq}(L) = \{\sigma \in (\Sigma^* \cup \{\Box\})^* : \mathrm{cnt}(\sigma) \subseteq L\}$.

Given a class $\mathcal{L}$, a *hypothesis space* for $\mathcal{L}$ is a class $\{H_\alpha : \alpha \in J\} \supseteq \mathcal{L}$ with corresponding indexing, where J is the set of indices for the hypothesis space. We will only consider automatic hypothesis spaces. A hypothesis space is *class preserving* iff $\mathcal{L} = \{H_\alpha : \alpha \in J\}$. A hypothesis space is *class comprising* iff $\mathcal{L} \subseteq \{H_\alpha : \alpha \in J\}$. A hypothesis space is *one-one* iff it is class preserving and, for every $L \in \mathcal{L}$, there is exactly one $\alpha \in J$ with $L = H_\alpha$.

A *learner* is a function $\mathbf{F} : (\Sigma^* \cup \{\Box\})^* \to J \cup \{?\}$. We use $\mathbf{F}$ for learners which may not be recursive. We use $\mathbf{P}$ for iterative learners and $\mathbf{Q}$ for iterative learners with additional long-term memory. The learners $\mathbf{P}$ and $\mathbf{Q}$ are usually automatic. Iterative and automatic learners are defined in Section 3 below.

Definition 2. Fix a class $\mathcal{L}$ and a hypothesis space $\{H_\alpha : \alpha \in J\}$ with J being the set of indices. Let $\mathbf{F}$ be a learner.
(a) [11] We say that $\mathbf{F}$ **Ex**-*learns* $\mathcal{L}$ if and only if for every $L \in \mathcal{L}$ and every $T \in \mathrm{txt}(L)$, there exists an $n \in \mathbb{N}$ and an $\alpha \in J$ with $H_\alpha = L$ such that, for every $m \geq n$, $\mathbf{F}(T[m]) = \alpha$.
(b) [6] We say that $\mathbf{F}$ **BC**-*learns* $\mathcal{L}$ if and only if for every $L \in \mathcal{L}$ and every $T \in \mathrm{txt}(L)$, there exists an $n \in \mathbb{N}$ such that for every $m \geq n$, $H_{\mathbf{F}(T[m])} = L$.
(c) [18] We say that $\mathbf{F}$ **Part**-*learns* $\mathcal{L}$ if and only if for every $L \in \mathcal{L}$ and every $T \in \mathrm{txt}(L)$, there exists an $\alpha \in J$ such that (i) $L_\alpha = L$, (ii) for every $n \in \mathbb{N}$, there exists a $k \geq n$ such that $\mathbf{F}(T[k]) = \alpha$ and (iii) for every $\beta \in J$ with $\beta \neq \alpha$, there exists an $n \in \mathbb{N}$ such that for every $k \geq n$, $\mathbf{F}(T[k]) \neq \beta$.

For **Ex**, **BC** and **Part** learning, one can assume without loss of generality that the learner never outputs ?. However, for some other criteria of learning, this may not be the case.

Definition 3. Let Σ and Γ be alphabets. Let $\{H_\alpha : \alpha \in J\}$ be a hypothesis space with some J being the set of indices. Let $\mathbf{F}$ be a learner.
(a) [5] We say that $\mathbf{F}$ is *consistent* on L if and only if for every $\sigma \in \mathrm{seq}(L)$, if $\mathbf{F}(\sigma) \in J$, then $\mathrm{cnt}(\sigma) \subseteq H_{\mathbf{F}(\sigma)}$. We say that $\mathbf{F}$ is consistent on $\mathcal{L}$ if it is consistent on each $L \in \mathcal{L}$.
(b) [2] We say that $\mathbf{F}$ is *conservative* on L if and only if for every $\sigma, \sigma' \in \mathrm{seq}(L)$, if $\mathbf{F}(\sigma) \in J$ and $\mathrm{cnt}(\sigma \diamond \sigma') \subseteq H_{\mathbf{F}(\sigma)}$, then $\mathbf{F}(\sigma \diamond \sigma') = \mathbf{F}(\sigma)$. We say that $\mathbf{F}$ is conservative on $\mathcal{L}$ if it is conservative on each $L \in \mathcal{L}$.
(c) [17] We say that $\mathbf{F}$ is *set-driven* if and only if for every $\sigma_1, \sigma_2 \in (\Sigma^* \cup \{\Box\})^*$, if $\mathrm{cnt}(\sigma_1) = \mathrm{cnt}(\sigma_2)$, then $\mathbf{F}(\sigma_1) = \mathbf{F}(\sigma_2)$.

When we are considering learning consistently (conservatively, set-drivenly) a class $\mathcal{L}$, we mean learning of the class by a learner which is consistent (conservative, set-driven) on $\mathcal{L}$. For each learning criterion **LC** such as **Ex**, **BC** and **Part**, we let **LC** also denote the collection of all classes which are **LC**-learned by a recursive learner using some class comprising hypothesis space.

The following recalls the definition of tell-tale set and introduces the notion of tell-tale cut-off word.

Definition 4 (Angluin's Tell-Tale Condition [2]). Suppose $\mathcal{L}$ is a class of languages.
(a) For every $L \in \mathcal{L}$, we say that D is a *tell-tale set of* L (*in* $\mathcal{L}$) if and only if D is a finite subset of L and for every $L' \in \mathcal{L}$ with $D \subseteq L' \subseteq L$ we have $L' = L$.
(b) For every $L \in \mathcal{L}$ and $x \in \Sigma^*$, we say that x is a *tell-tale cut-off word of* L (*in* $\mathcal{L}$) if and only if $\{y \in L \ : \ y \leq_{ll} x\}$ is a tell-tale set of L (in $\mathcal{L}$).
(c) We say that $\mathcal{L}$ satisfies *Angluin's tell-tale condition* if and only if every $L \in \mathcal{L}$ has a tell-tale set (in $\mathcal{L}$), or equivalently, a tell-tale cut-off word (in $\mathcal{L}$).

Fact 5 (Angluin [2]). *Let Σ be an alphabet. A class $\mathcal{L}$ of recursively enumerable languages is* **Ex***-learnable* (*by a not necessarily recursive learner*) *if and only if $\mathcal{L}$ satisfies Angluin's tell-tale condition.*

Note that for non recursive learners, **Ex** and **BC** learning are equivalent. Given a uniformly recursive class $\{L_\alpha : \alpha \in J\}$, Angluin [2] proved that the learner can be chosen to be recursive iff there is a uniformly recursively enumerable class of sets E_α such that each E_α is a tell-tale set for L_α. Using the Fundamental Theorem for automatic structures, the following theorem shows that any automatic class satisfying Angluin's tell-tale condition is **Ex**-learnable and the learner can be made to be recursive, consistent, conservative and set-driven.

Theorem 6. *Suppose $\mathcal{L}$ is automatic. Then, there is a learner which recursively, consistently, conservatively and set-drivenly* **Ex***-learns $\mathcal{L}$ if and only if $\mathcal{L}$ satisfies Angluin's tell-tale condition.*

Corollary 7. *It is decidable whether an automatic family $\mathcal{L} = \{L_\alpha : \alpha \in I\}$ is* **Ex***-learnable, where the input given to the decision-procedure is any DFA accepting the regular language $\{\mathrm{conv}\,(\alpha, x) : x \in L_\alpha, \alpha \in I\}$.*

3 Automatic Learning of Automatic Classes

It was shown above that all automatic classes that satisfy Angluin's tell-tale condition, can be learnt using a recursive learner. Learners that are able to memorize all past data are not practical. Rather, most learners in the setting of artificial intelligence are iterative, in the sense that these learners conjecture incrementally as they are fed the input inductively, one word at a time [21,22]. Iterative learners base their new conjectures only on their previous conjecture and the new datum (in other words they do not remember their past data, except as coded in the hypothesis).

In the realm of automatic structures, it is natural to consider automatic learners, where the learning function is in some way automatic. In the case of general recursive learners, there does not seem to be any natural correspondence which will lead to an interesting model. However, for iterative learners, there is a natural corresponding definition for automatic learners where the update function is automatic. Below we formally define automatic iterative learning and its variant, iterative learning with long-term memory.

Definition 8 (Wexler, Culicover [21], Wiehagen [22]). Let Σ, Γ and Δ be alphabets. Let $\mathcal{L}$ be a class (defined over alphabet Σ) and $\{H_\alpha : \alpha \in J\}$ be a hypothesis space with $J \subseteq \Gamma^*$. An *iterative learner* is any function $\mathbf{P} : (J \cup \{?\}) \times (\Sigma^* \cup \{\Box\}) \to J \cup \{?\}$. An *iterative learner with long-term memory* is any function $\mathbf{Q} : ((J \cup \{?\}) \times \Delta^*) \times (\Sigma^* \cup \{\Box\}) \to (J \cup \{?\}) \times \Delta^*$, where Δ is a suitable alphabet for memory.

For an iterative learner $\mathbf{P}$, we write $\mathbf{P}(w_0 w_1 \ldots w_n)$ as a short hand for the expression $\mathbf{P}(\ldots \mathbf{P}(\mathbf{P}(?, w_0), w_1), \ldots, w_n)$. Similar notation applies when considering iterative learners $\mathbf{Q}$ with long term memory. With these modifications, $\mathbf{P}$ and $\mathbf{Q}$ are seen as learners and the definitions of all the learning criteria carry over. Note that convergence of a learner $\mathbf{Q}$ is defined only with respect to the hypothesis and not the memory.

Definition 9. Suppose $\mathcal{L}$ is defined over alphabet Σ, and $\{H_\alpha : \alpha \in J\}$ is a hypothesis space. Suppose $\mathbf{P}$ is an iterative learner and $\mathbf{Q}$ is an iterative learner with long-term memory over some alphabet Δ.
(a) We say that $\mathbf{P}$ is *automatic* if and only if $\mathbf{P}$, as a relation over $(J \cup \{?\}) \times (\Sigma^* \cup \{\Box\}) \times (J \cup \{?\})$, is automatic. We say that $\mathbf{Q}$ is *automatic* if and only if $\mathbf{Q}$, as a relation over $((J \cup \{?\}) \times \Delta^*) \times (\Sigma^* \cup \{\Box\}) \times ((J \cup \{?\}) \times \Delta^*)$, is automatic.
(b) We say that the long-term memory of $\mathbf{Q}$ is *bounded by the longest datum seen so far* if and only if there exists a constant $c \in \mathbb{N}$ such that for every $\sigma \in (\Sigma^* \cup \{\Box\})^*$, if $\mathbf{Q}(\sigma) = (\alpha, \mu)$, then $|\mu| \leq \max\{|x| : x \in \mathrm{cnt}(\sigma)\} + c$.
We say that the long-term memory of $\mathbf{Q}$ is *bounded by the hypothesis size* if and only if there exists a constant $c \in \mathbb{N}$ such that for every $\sigma \in (\Sigma^* \cup \{\Box\})^*$, if $\mathbf{Q}(\sigma) = (\alpha, \mu)$, then $|\mu| \leq |\alpha| + c$.

Automatic iterative learners with long-term memory are called automatic learners from now on.

Definition 10. For the following, the hypothesis space is allowed to be any class comprising automatic family. Let **LC** be one of **Ex**, **BC** and **Part**. We let
(a) **AutoLC** be the set of all classes of languages that are **LC**-learned by some automatic learner with arbitrary long-term memory,
(b) **AutoWordLC** be the set of all classes of languages that are **LC**-learned by some automatic learner with long-term memory that is bounded by the longest datum seen so far,
(c) **AutoIndexLC** be the set of all classes of languages that are **LC**-learned by some automatic learner with long-term memory that is bounded by the hypothesis size,
(d) **AutoItLC** be the set of all classes of languages that are **LC**-learned by some automatic iterative learner.

We first show that automatic learners are not as powerful as general learners.

Theorem 11. *There exists an automatic $\mathcal{L}$ that is* **Ex** *learnable by some recursive iterative learner, but which is not* **AutoEx***-learnable.*

We now consider the relationship between various long-term memory limitations for the main criteria of learning: **Ex** and **BC**. Interestingly, if the memory is not explicitly constrained, then every automatic class which is **BC**-learnable can be **Ex**-learnt. For **BC**-learning, long-term memory is not useful (for automatic learners), as such memory can be coded into the hypothesis itself, as long as one is allowed padding of the hypothesis.

Theorem 12
(a) **AutoEx** = **AutoBC**.
(b) **AutoBC** = **AutoWordBC** = **AutoIndexBC** = **AutoItBC**.

The next theorem shows that, for **Ex** learning, there are classes which can be learnt by automatic learners having long-term memory bounded by longest word size seen so far while they cannot be learnt by automatic learners having long-term memory bounded by hypothesis size. The following theorem holds even when one considers class preserving hypothesis spaces.

Theorem 13
(a) **AutoItEx** $\subseteq$ **AutoWordEx** $\subseteq$ **AutoEx**.
(b) **AutoItEx** $\subseteq$ **AutoIndexEx** $\subseteq$ **AutoEx**.
(c) **AutoWordEx** $\not\subseteq$ **AutoIndexEx**.

Open Problem 14
The following problems are currently open:
(a) *Is* **AutoEx** = **AutoWordEx***?*
(b) *Is* **AutoIndexEx** $\subseteq$ **AutoWordEx***?*
(c) *Is* **AutoIndexEx** $\subseteq$ **AutoItEx***?*

If the alphabet is unary, then every **AutoEx**-learner can be replaced by an **AutoWordEx**-learner which answers (a) and (b) above in the affirmative for this special case. Also, the separation in Theorem 13 (c) is witnessed by a family of languages defined over unary alphabet.

Theorem 15. *Suppose that $\Sigma = \{0\}$ and $\mathcal{L} \subseteq \mathrm{powerset}(\Sigma^*)$ is an automatic class. Then $\mathcal{L}$ is in* **AutoWordEx** *as witnessed by a conservative, consistent and set-driven learner if and only if $\mathcal{L}$ satisfies Angluin's tell-tale condition.*

Hence, for language classes over a unary alphabet, **AutoWordEx** and **AutoEx** coincide and properly contain **AutoIndexEx**.

4 Consistent Learning

Note that for general recursive learners, all learnable automatic classes have a consistent, conservative and set-driven recursive learner, see Theorem 6 above.

Thus, on one hand, consistency, conservativeness and set-drivenness are not restrictive for learning automatic classes by recursive learners. On the other hand, in this section, we will show that consistency is a restriction when learning automatic classes by automatic learners. It will be interesting to explore similar questions for conservativeness and set-drivenness.

The following theorem gives an automatic class which can be **Ex**-learnt by an iterative automatic learner but which cannot be **Ex**-learnt by any consistent automatic learner.

Theorem 16. *There exists an automatic $\mathcal{L}$ such that*
(a) *$\mathcal{L}$ is* **AutoItEx** *learnable using a class preserving hypothesis space;*
(b) *$\mathcal{L}$ is not consistently* **AutoEx** *learnable even using a class comprising hypothesis space.*

The following theorem gives an automatic class $\mathcal{L}$ which can be **Ex**-learnt by a consistent automatic learner or **Ex**-learnt by an iterative automatic learner but which cannot be **Ex**-learnt by a consistent iterative automatic learner. Thus, requiring both consistency and iterativeness is more of a restriction compared to requiring only one of them.

Theorem 17. *There exists an automatic $\mathcal{L}$ such that*
(a) *$\mathcal{L}$ is consistently* **AutoEx** *learnable using $\mathcal{L}$ as the hypothesis space;*
(b) *$\mathcal{L}$ is* **AutoItEx** *learnable using $\mathcal{L}$ as the hypothesis space;*
(c) *$\mathcal{L}$ is not consistently* **AutoItEx** *learnable.*

The following theorem shows the existence of an automatic class which can be **Ex**-learnt by a consistent automatic iterative learner using a class comprising hypothesis space, but cannot be **Ex**-learnt by a consistent automatic learner using a class preserving hypothesis space. Thus, in some cases having a larger hypothesis space makes consistency problem easier to handle. Similar phenomenon for monotonic learning (for recursive learners) has been observed by [16].

Theorem 18. *There exists an automatic class $\mathcal{L}$ such that*
(a) *$\mathcal{L}$ is* **AutoItEx***-learnable using a class preserving hypothesis space;*
(b) *$\mathcal{L}$ is consistently* **AutoItEx***-learnable using some class comprising hypothesis space for $\mathcal{L}$;*
(c) *$\mathcal{L}$ is not consistently* **AutoEx***-learnable using any class preserving hypothesis space for $\mathcal{L}$.*

5 Automatic Learning from Fat Text

One of the reasons why iterative learning and its variations are restrictive is because the learners forget past data. So it is interesting to study the case when each datum appears infinitely often (such a text is called fat text). In the case of learning recursively enumerable sets, it has been shown that every explanatorily learnable class is also iteratively learnable from fat text [18]. In the following, it is investigated to which extent this result transfers to automatic learners.

Definition 19. [18] Let Σ be an alphabet. Let $T \in (\Sigma^* \cup \{\Box\})^\omega$. We say that T is *fat* if and only if for every $x \in \mathrm{cnt}(T)$ and $n \in \mathbb{N}$, there exists a $k \geq n$ such that $T(k) = x$. For $L \subseteq \Sigma^*$, we let $\mathrm{ftxt}(L) = \{T \in \mathrm{txt}(L) \ : \ T \text{ is fat}\}$.

Definition 20. Let Σ be an alphabet. Let $\{H_\alpha \ : \ \alpha \in J\}$ be a hypothesis space with some J being the set of indices. Let $\mathbf{P}$ be an iterative learner. We say that $\mathbf{P}$ **Ex**-*learns* $\mathcal{L}$ *from fat text* if and only if for every $L \in \mathcal{L}$ and every $T \in \mathrm{ftxt}(L)$, there exists an $n \in \mathbb{N}$ and an $\alpha \in J$ with $H_\alpha = L$ such that, for every $m \geq n$, $\mathbf{P}(T[m]) = \alpha$. Similarly one can define other criteria of learning from fat text.

Theorem 21. *The following conditions are equivalent for an automatic class L:*
(a) $\mathcal{L}$ *satisfies Angluin's tell-tale condition;*
(b) $\mathcal{L}$ *is* **AutoWordEx**-*learnable from fat text using the given class itself as the hypothesis space;*
(c) $\mathcal{L}$ *is* **AutoItEx**-*learnable from fat text using a class preserving hypothesis space.*

The next result shows that one cannot learn every given class iteratively from fat text using a one-one hypothesis space. So "padding", that is, the usage of the hypothesis as an auxiliary memory, is necessary for iterative learning from fat text. Furthermore, the following also shows constraints of iterative conservative automatic learners.

Theorem 22. *Some automatic class is class preservingly* **AutoItEx**-*learnable from normal text, class comprisingly conservatively* **AutoItEx** *learnable from normal text, but neither conservatively iteratively learnable from fat text using a class preserving hypothesis space nor iteratively learnable from fat text using a one-one class preserving hypothesis space.*

One might ask whether there are classes which can be learnt using some one-one hypothesis space but not be learnt using some other hypothesis space. The answer is "no". In the next result, the option "(from fat text)" has either to be taken at both places or at no place in the theorem.

Proposition 23. *If* $\{L_\alpha : \alpha \in I\}, \{H_\beta : \beta \in J\}$ *are automatic indexings, the mapping* $\alpha \mapsto L_\alpha$ *is one-one, every* L_α *is equal to some* H_β *and* $\{L_\alpha : \alpha \in I\}$ *is* **AutoItEx**-*learnable* (*from fat text*) *using the hypothesis space* $\{L_\alpha : \alpha \in I\}$*, then* $\{L_\alpha : \alpha \in I\}$ *is also* **AutoItEx**-*learnable* (*from fat text*) *using the hypothesis space* $\{H_\beta : \beta \in J\}$.

The next theorem shows that every automatic class (even those that may not satisfy Angluin's tell-tale condition) is partially learnable by an automatic iterative learner. This corresponds to the result by [18] that the whole class of all recursively enumerable languages is partially learnable by some recursive learner.

Theorem 24. *Every automatic* $\mathcal{L}$ *is* **AutoWordPart**-*learnable from fat text.*

Acknowledgments. We would like to thank John Case, Henning Fernau, Pavel Semukhin and Thomas Zeugmann for discussions about the subject of learning classes with automatic indexings. We also thank the anonymous referees for several helpful comments.

References

1. Angluin, D.: Finding patterns common to a set of strings. Journal of Computer and System Sciences 21, 46–62 (1980)
2. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45, 117–135 (1980)
3. Angluin, D.: Inference of reversible languages. Journal of the ACM 29, 741–765 (1982)
4. Angluin, D.: Learning regular sets from queries and counter-examples. Information and Computation 75, 87–106 (1987)
5. Bārzdiņš, J.: Inductive inference of automata, functions and programs. In: Proceedings of the 20th International Congress of Mathematicians, Vancouver, pp. 455–560 (1974)
6. Bārzdiņš, J.: Two theorems on the limiting synthesis of functions. In: Theory of Algorithms and Programs, vol. 1, pp. 82–88. Latvian State University (1974)
7. Blum, L., Blum, M.: Toward a mathematical theory of inductive inference. Information and Control 28, 125–155 (1975)
8. Blumensath, A., Grädel, E.: Automatic structures. In: 15th Annual IEEE Symposium on Logic in Computer Science (LICS), pp. 51–62. IEEE Computer Society, Los Alamitos (2000)
9. Fernau, H.: Identification of function distinguishable languages. Theoretical Computer Science 290, 1679–1711 (2003)
10. Freivalds, R., Kinber, E., Smith, C.: On the impact of forgetting on learning machines. Journal of the ACM 42, 1146–1168 (1995)
11. Gold, E.M.: Language identification in the limit. Information and Control 10, 447–474 (1967)
12. Head, T., Kobayashi, S., Yokomori, T.: Locality, reversibility, and beyond: Learning languages from positive data. In: Richter, M.M., Smith, C.H., Wiehagen, R., Zeugmann, T. (eds.) ALT 1998. LNCS (LNAI), vol. 1501, pp. 191–204. Springer, Heidelberg (1998)
13. Hodgson, B.R.: On direct products of automaton decidable theories. Theoretical Computer Science 19, 331–335 (1982)
14. Khoussainov, B., Nerode, A.: Automatic presentations of structures. In: Leivant, D. (ed.) LCC 1994. LNCS, vol. 960, pp. 367–392. Springer, Heidelberg (1995)
15. Lange, S., Wiehagen, R.: Polynomial time inference of arbitrary pattern languages. New Generation Computing 8, 361–370 (1991)
16. Lange, S., Zeugmann, T.: Language learning in dependence on the space of hypotheses. In: Proceedings of the Sixth Annual Conference on Computational Learning Theory, pp. 127–136. ACM Press, New York (1993)
17. Osherson, D., Stob, M., Weinstein, S.: Learning strategies. Information and Control 53, 32–51 (1982)
18. Osherson, D., Stob, M., Weinstein, S.: Systems that Learn: An Introduction to Learning Theory for Cognitive and Computer Scientists. MIT Press, Cambridge (1986)
19. Rubin, S.: Automata presenting structures: A survey of the finite string case. The Bulletin of Symbolic Logic 14, 169–209 (2008)
20. Shinohara, T.: Rich classes inferable from positive data: Length–bounded elementary formal systems. Information and Computation 108, 175–186 (1994)
21. Wexler, K., Culicover, P.: Formal Principles of Language Acquisition. MIT Press, Cambridge (1980)
22. Wiehagen, R.: Limes-Erkennung rekursiver Funktionen durch spezielle Strategien. Electronische Informationverarbeitung und Kybernetik 12, 93–99 (1976)

Untestable Properties Expressible with Four First-Order Quantifiers[⋆]

Charles Jordan[⋆⋆] and Thomas Zeugmann[⋆⋆⋆]

Division of Computer Science
Hokkaido University, N-14, W-9, Sapporo 060-0814, Japan
{skip,thomas}@ist.hokudai.ac.jp

Abstract. In property testing, the goal is to distinguish between structures that have some desired property and those that are *far* from having the property, after examining only a small, random sample of the structure. We focus on the classification of first-order sentences based on their quantifier prefixes and vocabulary into testable and untestable classes. This classification was initiated by Alon *et al.* [1], who showed that graph properties expressible with quantifier patterns $\exists^*\forall^*$ are testable but that there is an untestable graph property expressible with quantifier pattern $\forall^*\exists^*$. In the present paper, their untestable example is simplified. In particular, it is shown that there is an untestable graph property expressible with each of the following quantifier patterns: $\forall\exists\forall\exists$, $\forall\exists\forall^2$, $\forall^2\exists\forall$ and $\forall^3\exists$.

Keywords: property testing, logic.

1 Introduction

In property testing, we take a small, random sample of a large structure and wish to determine if the structure has some desired property or if it is far from having the property. The hope is that we can gain efficiency in return for not deciding the problem exactly. We focus on the classification problem for testability, where the goal is to determine exactly which prefix vocabulary classes of first-order logic are testable and which are not.

Property testing was first introduced in the context of program verification (cf. Rubinfeld and Sudan [11] and Blum *et al.* [4]). The study of combinatorial property testing was initiated by Goldreich *et al.* [8], who focused on graphs. Alon *et al.* [1] first considered the classification problem for testability, although they restricted their attention to undirected, loop-free graphs. They showed that all such first-order sentences[1] with quantifier prefixes of the form $\exists^*\forall^*$ express testable properties. They also showed that there exists an untestable property

[⋆] An earlier version with additional proofs is available as [9]. We would like to thank an anonymous referee for significant improvements to Theorem 2.

[⋆⋆] Supported by a Grant-in-Aid for JSPS Fellows under Grant No. 2100195209.

[⋆⋆⋆] Supported by MEXT Grant-in-Aid for Scientific Research on Priority Areas under Grant No. 21013001.

[1] We assume throughout that all sentences are in prenex normal form.

expressible with the prefix $\forall^*\exists^*$. The example they give is (essentially) an encoding of graph isomorphism that can be expressed with a quantifier prefix of the form $\forall^{12}\exists^5$.

In studying the classification problem, it is necessary to determine the *minimum* number of quantifiers needed to express untestable properties. Additionally, the first-order theory of graphs is not restricted to undirected, loop-free graphs. Here, we show that there exists an untestable property of directed graphs that is expressible in first-order sentences with prefixes $\forall\exists\forall\exists$, $\forall\exists\forall^2$, $\forall^2\exists\forall$, and $\forall^3\exists$, when equality ($=$) is allowed (see Theorem 2 for a more formal statement). That is, with prefixes of these patterns and when equality is allowed, four first-order quantifiers suffice to express an untestable graph property. The proof is a modification of the proof by Alon *et al.* [1], which is made possible by the presence of directed edges and loops.

The results in Jordan and Zeugmann [10] show that one universal quantifier is not sufficient to express an untestable property (regardless of the vocabulary), and so it would be interesting to determine the status of the remaining prefixes with two universal quantifiers.

A class of first-order logic has the *finite model property* if every satisfiable formula in the class has a finite model. Classes without the finite model property contain *infinity axioms*, i.e. satisfiable formulas without finite models. The current classification for testability closely resembles the classification for the finite model property. It would be particularly interesting to determine the testability of the "minimal" classes with infinity axioms $[\forall^3\exists, (0,1)]$ and $[\forall\exists\forall, (0,1)]$[2].

2 Preliminaries

In property testing, the goal is always to distinguish structures that have some property from those that are far from having the property. We are particularly interested in properties that are first-order definable, and so we begin by defining our logic. Enderton [7] provides a more detailed introduction.

The atomic terms are the (countable) variable symbols x_i. There are no function or constant symbols, and so the terms are exactly the atomic terms. The atomic formulas are $E(x_i, x_j)$ and $x_i = x_j$, for any two variable symbols x_i and x_j. The formulas are built from the atomic formulas using the usual Boolean connectives (i.e., $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$), negation ($\neg$) and first-order quantifiers ($\forall$, $\exists$) in the canonical way. The *well-formed* formulas or *sentences* are the formulas which contain no free variables. We have no further use for formulas that are not well-formed, and so we will refer to the well-formed formulas simply as *formulas*.

Our logic contains a special equality predicate symbol ($=$) which is always interpreted as true equality (i.e., $x_i = x_j$ is true iff the two symbols x_i and x_j refer to the same object). It also contains a single binary predicate symbol, which we have given the name E. Of course, the name of this symbol is not important; any fixed, unique name could have been chosen.

[2] These classes do not permit equality and so Theorem 2 does not immediately imply that the first is untestable.

A structure is an object that allows us to interpret a sentence in our logic. It consists of a finite universe U over which the variable symbols are allowed to range, and a binary relation E corresponding to the symbol E in our logic. Any such object can be considered as a (directed) graph, so from now on we refer to these structures as graphs. See Diestel [6] for an introduction to graph theory.

Definition 1. *A* graph $A = (U^A, E^A)$ *is a pair consisting of a finite set of vertices* U^A *and a binary edge relation* $E^A \subseteq U^A \times U^A$*.*

The natural numbers are denoted by $\mathbb{N} := \{0, 1, \ldots\}$. For any set U we write $|U|$ to denote the cardinality of U and generally identify U with the set $\{0, \ldots, |U| - 1\}$. We denote the set of graphs on exactly n vertices by $\mathcal{G}^n$ and the set of all graphs by $\mathcal{G} := \cup_{n\in\mathbb{N}}\mathcal{G}^n$. The *size* of the universe of a graph $A = (U^A, E^A)$ is denoted by $\#(A)$ and defined as $\#(A) := |U^A|$.

A *property* P is any subset of $\mathcal{G}$. Sentences are interpreted in the usual way, and so we can decide $A \models \varphi$ for any fixed graph A and first-order sentence φ. Each sentence φ therefore defines a property, namely the set of its models,

$$P_\varphi := \{A \mid A \in \mathcal{G} \text{ and } A \models \varphi\} .$$

The properties that we use in the proof of Theorem 2 involve encodings of isomorphisms. Graphs $A = (U^A, E^A)$ and $B = (U^B, E^B)$ are *isomorphic* if there is a bijection $f : U^A \to U^B$ such that for all $(x, y) \in U^A \times U^A$, $(x, y) \in E^A$ iff $(f(x), f(y)) \in E^B$. We say that a property P is *closed under isomorphisms* if for all isomorphic $A, B \in \mathcal{G}$, it is true that $A \in P$ iff $B \in P$. All properties expressible in our logic are closed under isomorphisms.

The goal in property testing is to distinguish between structures that have properties and structures that are *far* from having them. This requires a distance measure, which we define next. In the following, $\oplus$ denotes exclusive-or and E^A and E^B are the edge relations of A and B, respectively.

Definition 2. *Let* $n \in \mathbb{N}$ *and let* U *be any universe such that* $|U| = n$*. Furthermore, let* $A = (U, E^A)$ *and* $B = (U, E^B)$ *be any two graphs with universe* U*. The* distance *between* A *and* B *is*

$$\text{dist}(A, B) := \frac{|\{(x_1, x_2) \mid x_1, x_2 \in U \text{ and } E^A(x_1, x_2) \oplus E^B(x_1, x_2)\}|}{n^2} .$$

Note that by definition, $\#(A) = \#(B) = n$. The dist distance is the fraction of edges on which the two graphs disagree. This is the *dense graph* model introduced by Goldreich *et al.* [8] and is essentially based on the adjacency matrix representation. The dist distance generalizes to properties in the following way.

Definition 3. *Let* $P \subseteq \mathcal{G}$ *be a property of graphs and let* $A \in \mathcal{G}^n$ *be a graph with* n *vertices. Then,*

$$\text{dist}(A, P) := \min_{A' \in \mathcal{G}^n \cap P} \text{dist}(A, A') .$$

We are now able to define property testing itself. The following definitions are typical, but we will also mention several variations.

Definition 4. *An* ε-tester *for property* P *is a randomized algorithm given an oracle which answers queries for the universe size and queries for the existence of edges connecting given nodes in a graph* A*. The tester must accept with probability at least* $2/3$ *if* A *has* P *and must reject with probability at least* $2/3$ *if* $\mathrm{dist}(A, P) \geq \varepsilon$*.*

Definition 5. *A property* P *is* testable *if for every* $\varepsilon > 0$ *there is an* ε*-tester for* P *making a number of queries which is bounded from above by a function depending only on* ε*.*

We allow different ε-testers for each $\varepsilon > 0$ and our definitions are therefore non-uniform. The uniform case is strictly more difficult (see, e.g., Alon and Shapira [3]). We are interested in proving untestability, and our results hold even in the non-uniform case. In *oblivious* testing (see Alon and Shapira [2]), the testers are not given access to the size of the universe. Again, our results hold in the more general case where the testers may make decisions based on the size of the universe. In a similar way, the number of loops in a graph is asymptotically insignificant compared to the number of possible non-loops. Modifying the definition of distance to account for this makes testing strictly more difficult (see Jordan and Zeugmann [10]) and so we use the more general definition above.

However, the (possible) loops seem to affect the notion of *indistinguishability* defined by Alon *et al.* [1]. We use the following modification of Definition 2.

Definition 6. *Let* $n \in \mathbb{N}$ *and let* U *be any universe such that* $|U| = n$*. Furthermore, let* $A = (U, E^A)$ *and* $B = (U, E^B)$ *be any two graphs with universe* U*. For notational convenience, let*

$$d_1(A,B) := \frac{|\{x \mid x \in U \text{ and } E^A(x,x) \oplus E^B(x,x)\}|}{n}\,, \textit{ and}$$

$$d_2(A,B) := \frac{|\{(x_1,x_2) \mid x_1,x_2 \in U,\, x_1 \neq x_2,\ \textit{and } E^A(x_1,x_2) \oplus E^B(x_1,x_2)\}|}{n^2}\,.$$

The mr-distance *between* A *and* B *is*

$$\mathrm{mrdist}(A,B) := \max\{d_1(A,B), d_2(A,B)\}\ .$$

Again, note that $\#(A) = \#(B) = n$. Although the number of loops is asymptotically insignificant, a tester can easily restrict its queries to the form (x, x) and distinguish between graphs that differ only in loops. Definition 6 is a special case of a definition from Jordan and Zeugmann [10], and mrdist abbreviates "maximum relational distance." We use the following simple variation of indistinguishability for graphs that may contain loops.

Definition 7. *Two properties* P *and* Q *of graphs are* indistinguishable *if they are closed under isomorphisms and for every* $\varepsilon > 0$ *there exists an* N_ε *such that for any graph* A *with universe of size* $n \geq N_\varepsilon$*, if* A *has* P *then* $\mathrm{mrdist}(A, Q) \leq \varepsilon$ *and if* A *has* Q *then* $\mathrm{mrdist}(A, P) \leq \varepsilon$*.*

The important fact to note is that indistinguishability preserves testability. The proof of the following theorem is analogous to that given in Alon *et al.* [1].

Theorem 1. *If P and Q are indistinguishable, then P is testable if and only if Q is testable.*

Our classification definitions are from Börger *et al.* [5] except that we omit function symbols. We omit a detailed discussion, but the following is for completeness. Let Π be a string over the four-character alphabet $\{\exists, \exists^*, \forall, \forall^*\}$. Then $[\Pi, (0, 1)]_=$ is the set of sentences in prenex normal form which satisfy the following conditions:

1. The quantifier prefix is contained in the language specified by the regular expression Π.
2. There are zero (0) monadic predicate symbols.
3. In addition to the equality predicate (=), there is at most one (1) binary predicate symbol.
4. There are no other predicate symbols.

That is, $[\Pi, (0, 1)]_=$ is the set of sentences in the logic that we have defined above whose quantifier prefixes in prenex normal form match Π.

3 An Untestable Property

We will begin by defining property P, which is essentially the graph isomorphism problem for undirected loop-free graphs encoded in directed graphs that may contain loops. We will begin by showing in Lemma 1 that P is indistinguishable from property P_f (cf. Definition 9) which is expressible in any of the prefix vocabulary classes mentioned in Theorem 2. We will then show that P is not testable. Indistinguishability preserves testability and so this implies that P_f is also untestable, which will suffice to show the following theorem.

Theorem 2. *The following prefix classes are not testable:*

1. $[\forall\exists\forall\exists, (0, 1)]_=$
2. $[\forall\exists\forall^2, (0, 1)]_=$
3. $[\forall^2\exists\forall, (0, 1)]_=$
4. $[\forall^3\exists, (0, 1)]_=$

We define property P as follows. First, a graph that has property P must consist of an even number of vertices, of which exactly half have loops. The subgraph induced by the vertices with loops must be isomorphic to that induced by the vertices without loops, ignoring all loops, and there must be no edges connecting the vertices with loops to those without loops. Finally, all edges must be *undirected* (i.e., an edge from x to y implies an edge from y to x). We refer to such undirected edges as *paired edges.*

Definition 8. *A graph $G \in \mathcal{G}^n$ has P iff the following conditions are satisfied:*

1. *For some s, $n = 2s$.*
2. *There are exactly s vertices x satisfying $E(x,x)$. We will refer to the set of such vertices as H_1 and to the remaining s vertices as H_2.*
3. *The substructure induced by H_1 is isomorphic to that induced by H_2 when all loops are removed. That is, there is a bijection f from H_1 to H_2 such that for distinct $x, y \in H_1$, it is true that $G \models E(x,y)$ iff $G \models E(f(x), f(y))$.*
4. *There are no edges between H_1 and H_2.*
5. *All edges are paired.*

Graph isomorphism is not directly expressible in first-order logic, and so we use the following encoding where the bijection f is made explicit by adding n edges between H_1 and H_2.

Definition 9. *A graph $G \in \mathcal{G}^n$ has P_f iff the following conditions are satisfied:*

1. *For every vertex x, if $E(x,x)$ then there is an edge from x to exactly one y such that $\neg E(y,y)$.*
2. *For every vertex x, if $\neg E(x,x)$ then there is an edge from x to exactly one y such that $E(y,y)$.*
3. *For all vertices x and y, $E(x,y)$ iff $E(y,x)$.*
4. *For all pairwise distinct vertices x_1, x_2, x_3, x_4, if $E(x_1,x_1)$, $\neg E(x_2,x_2)$, $E(x_3,x_3)$, $\neg E(x_4,x_4)$, $E(x_1,x_2)$ and $E(x_3,x_4)$, then $E(x_1,x_3)$ iff $E(x_2,x_4)$.*

Expressing Conditions 1 and 2 as "there is at most one such y" and "there is at least one such y," P_f can be expressed in each of the classes $[\forall\exists\forall\exists, (0,1)]_=$, $[\forall\exists\forall^2, (0,1)]_=$, $[\forall^2\exists\forall, (0,1)]_=$ and $[\forall^3\exists, (0,1)]_=$.

For example, in the class $[\forall^3\exists, (0,1)]_=$, we can express P_f by

$$\begin{aligned}
&\forall x_1 \forall x_3 \forall x_4 \exists x_2 : \Big[\\
&\quad \big((E(x_1,x_1) \leftrightarrow \neg E(x_2,x_2)) \wedge E(x_1,x_2) \big) \quad \wedge \\
&\quad \big[\big((E(x_1,x_1) \leftrightarrow \neg E(x_3,x_3)) \wedge (E(x_3,x_3) \leftrightarrow \neg E(x_4,x_4)) \wedge \\
&\qquad E(x_1,x_3) \wedge E(x_1,x_4) \big) \to x_3 = x_4 \big] \quad \wedge \\
&\quad \big(E(x_1,x_3) \to E(x_3,x_1) \big) \quad \wedge \\
&\quad \big([E(x_1,x_1) \wedge E(x_3,x_3) \wedge x_1 \neq x_3 \wedge \neg E(x_4,x_4) \wedge E(x_3,x_4)] \to \\
&\qquad (\neg E(x_2,x_2) \wedge E(x_1,x_2) \wedge (E(x_1,x_3) \leftrightarrow E(x_2,x_4))) \big) \Big] .
\end{aligned}$$

To express P_f with prefixes $\forall^2\exists\forall$ and $\forall\exists\forall^2$, it suffices to reorder the quantifiers (keeping x_2 existential and x_1 first). The prefix $\forall\exists\forall\exists$ requires a few additional modifications.

The two properties P and P_f differ only in the edges which make the isomorphism explicit in P_f but are forbidden in P. There are at most n such edges, none of which are loops. This suffices to prove the following.

Lemma 1. *Properties P and P_f are indistinguishable.*

Proof. Let $\varepsilon > 0$ be arbitrary and let $N_\varepsilon = \varepsilon^{-1}$. Assume that G is a structure that has property P and that $\#(G) > N_\varepsilon$. We will show that $\mathrm{mrdist}(G, P_f) < \varepsilon$.

Structure G has P and so there is a bijection f satisfying Condition 3 of Definition 8. For all $x \in H_1$, we add the edges $E(x, f(x))$ and $E(f(x), x)$ and call the result G'. Property P_f differs from P only in that the isomorphism is made explicit by the edges connecting loops and non-loops, and so G' has P_f. Indeed, it satisfies Conditions 1 and 2 of Definition 9 because G had no edges between loops and non-loops and we have connected each to exactly one of the other, following the bijection f. Next, G' satisfies Condition 3 of Definition 9 because G satisfied Condition 5 of Definition 8 and we added only paired edges. Finally, G' satisfies Condition 4 of Definition 9 because the edges between loops and non-loops follow the isomorphism f from Condition 3 of Definition 8.

We have added exactly n (directed) edges, none of which are loops and so $\mathrm{mrdist}(G, P) \leq \mathrm{mrdist}(G, G') = 0 + n/n^2 < \varepsilon$, where the inequality holds for $n > N_\varepsilon$. The converse is analogous; given a G that has property P_f, we simply remove the n edges between loops and non-loops after using them to construct the isomorphism f. □

Properties P and P_f are indistinguishable. Testability is preserved by indistinguishability (cf. Theorem 1) and thus showing that P is not testable suffices to prove that P_f is not testable (and therefore Theorem 2). The proof closely follows that of Alon *et al.* [1]. The crucial lemma is the following, a combination of Lemmata 7.3 and 7.4 from Alon *et al.* [1]. We use $\mathrm{count}_H(T)$ to refer to the number of times that a graph T occurs as an induced subgraph in H. A *bipartite graph* is a graph where we can partition the vertices into two sets H_1 and H_2 such that there are no edges "internal" to the partitions. That is, for all $x_1, y_1 \in H_1$ and $x_2, y_2 \in H_2$, $\neg E(x_1, y_1)$ and $\neg E(x_2, y_2)$. See Jordan and Zeugmann [9] for an explicit proof of Lemma 2, which is technical and long.

Lemma 2 (Alon *et al.* [1]). *There exists a constant $\varepsilon' > 0$ such that for every $D \in \mathbb{N}$, there exist two undirected bipartite graphs $H = H(D)$ and $H' = H'(D)$ satisfying the following conditions.*

1. *Both H and H' have a bipartition into classes U_1 and U_2, each of size t.*
2. *In both H and H', for all subgraphs X with size $t/3 \leq \#(X) \leq t$, there are more than $t^2/18$ undirected edges between X and the remaining part of the graph.*
3. *The minimum degree of both H and H' is at least $t/3$.*
4. $\mathrm{dist}(H, H') \geq \varepsilon'$.
5. *For all D-element graphs T,* $\mathrm{count}_H(T) = \mathrm{count}_{H'}(T)$.

It is worth noting that the above is for undirected, loop-free graphs. However, bipartite graphs never have loops and "undirected" in our setting results in paired edges. It is easy to show that if two structures agree on the counts for all size D induced subgraphs, they agree on the counts for all induced subgraphs of size at most D. This is done by applying Lemma 3 inductively.

Lemma 3. *Let H and H' be two graphs, both of size s, and let $2 < D \leq s$. If for every graph T of size D, $\mathrm{count}_H(T) = \mathrm{count}_{H'}(T)$, then for every graph T' of size $D-1$, $\mathrm{count}_H(T') = \mathrm{count}_{H'}(T')$.*

Proof. Assume H and H' satisfy the initial conditions of Lemma 3, but that there exists a T' of size $D-1$ such that $\mathrm{count}_H(T') \neq \mathrm{count}_{H'}(T')$. Let $C = \{T \mid \#(T) = D \text{ and } T \text{ contains } T' \text{ as an induced subgraph}\}$.

Note that $\sum_{T \in C} \mathrm{count}_H(T)\,\mathrm{count}_T(T') = \mathrm{count}_H(T')(s-D+1)$ and likewise for $\sum_{T \in C} \mathrm{count}_{H'}(T)\,\mathrm{count}_T(T')$. We have assumed that H and H' satisfy $\mathrm{count}_H(T) = \mathrm{count}_{H'}(T)$ for $T \in C$, but $\mathrm{count}_H(T') \neq \mathrm{count}_{H'}(T')$, giving a contradiction and the Lemma follows. □

Lemma 4. *Property P is not testable.*

Proof. Assume that P is testable. Then, there exists an ε-tester for

$$\varepsilon := \min\{\varepsilon'/8, 1/144\},$$

where ε' is the constant from Lemma 2. We can assume without loss of generality that the tester queries all edges in a random sample of $D := D(\varepsilon)$ vertices.

Consider the graph G which contains two copies of the $H = H(D)$ from Lemma 2, where one of the copies is marked by loops on each vertex and there are no edges between the copies. This graph has property P, and so the tester must accept it with probability at least $2/3$. Next, consider the graph G' which contains one copy of H marked by loops and one copy of H', again where there are no edges between the two (induced) subgraphs. Graph G' is such that $\mathrm{dist}(G', P) \geq \varepsilon$ (cf. Lemma 5) and so it must be rejected with probability at least $2/3$. Both G and G' consist of two bipartite graphs, each of which has a bipartition into two classes of size t, and so $\#(G) = \#(G') = 4t$.

However, G and G' both contain exactly the same number of each induced subgraph with D vertices. This is because both have loops on exactly half of the vertices and the two halves are not connected by any edges. Some of the D vertices must be in the first copy of H and the others in the second H (resp. H'). By Lemma 3, H and H' contain the same number of each induced subgraph with size at most D. The tester therefore obtains any fixed sample with the same probability in G and G' and is unable to distinguish between them. Hence, it is unable to accept G with probability $2/3$ and also reject G' with probability $2/3$. This completes the proof, taking into account Lemma 5 below. □

Recall that testing is easiest under the dist definition, and so Lemma 4 also implies P is not testable under other definitions.

Lemma 5. *The graph G' is such that $\mathrm{dist}(G', P) \geq \varepsilon$.*

Proof. Suppose that $\mathrm{dist}(G', P) < \varepsilon$. Then, there is an $M \in P$ such that $\mathrm{dist}(G', M) < \varepsilon$. Let M_1 be the set of vertices with loops in M and let M_2 be the set of vertices without loops. We will refer to the subgraph induced by the vertices with loops in G' as H and to that induced by those without loops as H'.

Without loss of generality, assume that $|M_1 \cap H| \geq |M_1 \cap H'|$. Then, $|M_1 \cap H| \geq t$. We let α_1 be the set $M_1 \backslash H$ and α_2 be $M_2 \backslash H'$. Note that $|\alpha_1| = |\alpha_2|$ and $|\alpha_1| \leq t$ because $|M_1 \cap H| \geq t$.

Informally, M is formed by moving the vertices α_1 from H' to H and the vertices α_2 from H to H', and then possibly making other changes. There are three cases, which we will consider in order.

1. $|\alpha_1| = 0$.
2. $|\alpha_1| \geq t/3$.
3. $0 < |\alpha_1| < t/3$.

If $|\alpha_1| = 0$, then we can construct M from G' without exchanging vertices between H and H', and in particular, construct H' from H (ignoring loops), by making less than $\varepsilon(4t)^2$ modifications. However, $\text{dist}(H, H') \geq \varepsilon'$ by Lemma 2 and so this must require at least $\varepsilon'(2t)^2$ modifications. By definition, $\varepsilon < \varepsilon'/4$ so $\varepsilon(4t)^2 < \varepsilon'(2t)^2$. The first case is therefore not possible.

Recall that $|\alpha_1| \leq t$. If $|\alpha_1| \geq t/3$, then by Condition 2 of Lemma 2 there exists at least $t^2/18$ undirected edges between α_1 and $H' \backslash \alpha_1$ and between α_2 and $H \backslash \alpha_2$. All of these edges must be removed to satisfy P because each would connect a vertex with a loop to a vertex without a loop. Therefore,

$$\text{dist}(G', M) \geq \frac{4t^2/18}{(4t)^2} = 1/72 .$$

But, $\varepsilon < 1/72$ and so the second case is not possible.

Therefore, it must be that $0 < |\alpha_1| < t/3$. Here, we will show that it must be the case that α_1 and α_2 are relatively far apart. If they are not far apart, then it is possible to modify them instead of swapping them. This essentially results in the first case considered above. Condition 3 of Lemma 2 requires that each vertex has relatively high degree. These edges can be either internal to α_1 (resp. α_2) or connecting α_1 (α_2) with $H' \backslash \alpha_1$ ($H \backslash \alpha_2$). If α_1 and α_2 are relatively far apart, then we will see that this forces too many edges "outside" of α_1 (resp. α_2), resulting in a similar situation to the second case considered above.

We have assumed that $\text{dist}(G', M) < \varepsilon$ and that we can construct M from G' by making less than $\varepsilon(4t)^2$ modifications if we move α_1 to H and α_2 to H'. This entails the following modifications.

1. Removing all edges connecting α_1 to $H' \backslash \alpha_1$.
2. Removing all edges connecting α_2 to $H \backslash \alpha_2$.
3. Adding any required edges between α_1 and $H \backslash \alpha_2$.
4. Adding any required edges between α_2 and $H' \backslash \alpha_1$.
5. Changing α_1, α_2, $H \backslash \alpha_2$ and $H' \backslash \alpha_1$ to their final forms.

We can assume that the total number of modifications is less than $\varepsilon(4t)^2$. It must be that $\text{dist}(\alpha_1, \alpha_2)|\alpha_1|^2/(4t)^2 + \varepsilon \geq \varepsilon'/4$. If this does not hold, then we could first modify α_1 to make it identical to α_2 and then make H' identical to M_2.

Next, M_2 is identical to M_1, which we could make identical to H. This would require less than $\varepsilon'(2t)^2$ modifications, which would violate Lemma 2. Therefore,

$$\operatorname{dist}(\alpha_1, \alpha_2) \;\geq\; \frac{16(\varepsilon'/4 - \varepsilon)t^2}{|\alpha_1|^2}\,. \qquad (1)$$

If both α_1 and α_2 are complete graphs then they cannot be far apart. Given that all vertices in α_1 (α_2 is analogous) have degree at least $t/3$, then there must be at least

$$|\alpha_1|(t/3 - |\alpha_1| + 1) + 2r$$

edges connecting α_1 to $H'\backslash\alpha_1$, where r is the number of edges internal to α_1 that must be omitted to satisfy (1). The simple lower bound on r, the number of edges needed for two graphs with at most r edges to be $\operatorname{dist}(\alpha_1, \alpha_2)$-far, that follows from $\operatorname{dist}(\alpha_1, \alpha_2) \leq 2r/|\alpha_1|^2$ is sufficient. Finally, combining this with Inequality (1) yields

$$r \;\geq\; 8(\varepsilon'/4 - \varepsilon)t^2\,. \qquad (2)$$

The number of edges connecting α_1 to $H'\backslash\alpha_1$ is therefore, by (2), at least

$$|\alpha_1|(t/3 - |\alpha_1| + 1) + 16(\varepsilon'/4 - \varepsilon)t^2 \;\geq\; 16(\varepsilon'/4 - \varepsilon)t^2\,.$$

All of these edges must be removed to move α_1 (resp. α_2), and so

$$\operatorname{dist}(G', M) \;\geq\; \frac{16(\varepsilon'/4 - \varepsilon)t^2}{(4t)^2} \;=\; \frac{\varepsilon'}{4} - \varepsilon\,.$$

We have defined $\varepsilon \leq \varepsilon'/8$ and so $\operatorname{dist}(G', M) \geq \varepsilon$, a contradiction.

The cases are exhausted and so $\operatorname{dist}(G', P) \geq \varepsilon$ as desired. □

4 Conclusion

Property testing is an application of induction (in the philosophy of science sense), in which we obtain a small, random sample of a structure and seek to determine whether the structure has a desired property or is far from having the property. We have considered the *classification problem* for testability, wherein we classify the prefix vocabulary classes of first-order logic according to their testability. In particular, we simplified the untestable property, expressible with quantifier prefix $\forall^{12}\exists^5$, from Alon *et al.* [1] for the case of directed graphs which may contain loops. This implies that there exists an untestable property expressible with quantifier prefixes $\forall\exists\forall\exists$, $\forall\exists\forall^2$, $\forall^2\exists\forall$ and $\forall^3\exists$.

In the classification problem for testability, it is necessary to determine the *minimal* untestable classes. Informally, we seek the untestable properties that are *easiest* to express. Jordan and Zeugmann [10] showed that classes with (at most) one universal quantifier are testable, and so there are at least two minimal untestable classes which have either two or three universal quantifiers.

The current classification for testability closely resembles the classification for the finite model property (see, e.g., Section 6.5 of Börger *et al.* [5]). The "minimal" classes without this property (i.e., those with infinity axioms) are $[\forall^3\exists, (0, 1)]$ and $[\forall\exists\forall, (0, 1)]$, while the case of $[\forall^2\exists, (0, 1)]_=$ is apparently open. It would be particularly interesting to determine the testability of these classes.

Acknowledgments. We would like to thank an anonymous referee for significantly improving Theorem 2 by eliminating one variable from each prefix, adding $\forall\exists\forall\exists$, and also for simplifying the example following Definition 9.

References

[1] Alon, N., Fischer, E., Krivelevich, M., Szegedy, M.: Efficient testing of large graphs. Combinatorica 20(4), 451–476 (2000)

[2] Alon, N., Shapira, A.: A characterization of the (natural) graph properties testable with one-sided error. SIAM J. Comput. 37(6), 1703–1727 (2008)

[3] Alon, N., Shapira, A.: A separation theorem in property testing. Combinatorica 28(3), 261–281 (2008)

[4] Blum, M., Luby, M., Rubinfeld, R.: Self-testing/correcting with applications to numerical problems. J. of Comput. Syst. Sci. 47(3), 549–595 (1993)

[5] Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer, Heidelberg (1997)

[6] Diestel, R.: Graph Theory, 3rd edn. Springer, Heidelberg (2006)

[7] Enderton, H.B.: A Mathematical Introduction to Logic, 2nd edn. Academic Press, London (2000)

[8] Goldreich, O., Goldwasser, S., Ron, D.: Property testing and its connection to learning and approximation. J. ACM 45(4), 653–750 (1998)

[9] Jordan, C., Zeugmann, T.: Contributions to the classification for testability: Four universal and one existential quantifier. Technical Report TCS-TR-A-09-39, Hokkaido University, Division of Computer Science (November 2009)

[10] Jordan, C., Zeugmann, T.: Relational properties expressible with one universal quantifier are testable. In: Watanabe, O., Zeugmann, T. (eds.) SAGA 2009. LNCS, vol. 5792, pp. 141–155. Springer, Heidelberg (2009)

[11] Rubinfeld, R., Sudan, M.: Robust characterizations of polynomials with applications to program testing. SIAM J. Comput. 25(2), 252–271 (1996)

The Copying Power of Well-Nested Multiple Context-Free Grammars

Makoto Kanazawa[1] and Sylvain Salvati[2,*]

[1] National Institute of Informatics, Tokyo, Japan
[2] INRIA Bordeaux – Sud-Ouest, Talence, France

Abstract. We prove a copying theorem for well-nested multiple context-free languages: if $L = \{ w\#w \mid w \in L_0 \}$ has a well-nested m-MCFG, then L has a 'non-branching' well-nested m-MCFG. This can be used to give simple examples of multiple context-free languages that are not generated by any well-nested MCFGs.

1 Introduction

For a long time, the formalism of *multiple context-free grammars* [18], together with many others equivalent to it, has been regarded as a reasonable formalization of Joshi's [9] notion of *mildly context-sensitive grammars*. Elsewhere [10], we have made a case that a smaller class of grammars, consisting of MCFGs whose rules are *well-nested*, might actually provide a better formal approximation to Joshi's informal concept. Well-nested MCFGs are equivalent to *non-duplicating macro grammars* [5] and to *coupled-context-free grammars* [7]. Kanazawa [11] proves the pumping lemma for well-nested multiple context-free languages. The well-nestedness constraint has also been a focus of attention recently in the area of *dependency grammars* (e.g., [12]).

Seki and Kato [17] present a series of languages that are generated by MCFGs of dimension m, but not by any well-nested MCFGs of the same dimension. These examples illustrate the limiting effect that the well-nestedness constraint has on the class of generated languages at each level m of the infinite hierarchy of m-multiple context-free languages ($m \geq 1$).

An interesting fact is that the examples of Seki and Kato [17] all belong to the class of well-nested MCFLs at some higher level of the hierarchy, so they do not serve to separate the whole class of MCFLs from the whole class of well-nested MCFLs. In fact, to our knowledge, the only example that has appeared in the literature of an MCFL which is not a well-nested MCFL is the language discussed by Michaelis [13], originally due to Staudacher [20]. Staudacher uses Hayashi's [6] theorem to show that this language is not an indexed language, while Michaelis gives a (non-well-nested) 3-MCFG generating it. Since well-nested MCFLs are all indexed languages, it follows that this language is an MCFL which is not

* We are grateful to Uwe Mönnich for pointing us to Engelfriet and Skyum's [4] paper in connection with the question of what languages are in MCFL − $\mathrm{MCFL}_{\mathrm{wn}}$.

a well-nested MCFL. As it happens, the definition of this language is rather complex, and Staudacher's proof is not easy to understand.

As a matter of fact, what we would like to call the "triple copying theorem" for OI (the class of OI macro languages), due to Engelfriet and Skyum [4], can be used to give a simple example of a language that separates MCFL (the class of MCFLs) from $\mathrm{MCFL}_{\mathrm{wn}}$ (the class of well-nested MCFLs). This theorem says that $L = \{ w\#w\#w \mid w \in L_0 \} \in \mathrm{OI}$ implies $L_0 \in \mathrm{EDT0L}$.[1] (Here and henceforth, L_0 is a language over some alphabet Σ and # is a symbol not in Σ.) Since OI is the same as the class of indexed languages [5] and includes the class of well-nested MCFLs, and $L = \{ w\#w\#w \mid w \in L_0 \} \in$ 3-MCFL for all $L_0 \in \mathrm{CFL}$, this theorem implies that $L \in \text{3-MCFL} - \mathrm{MCFL}_{\mathrm{wn}}$ whenever $L_0 \in \mathrm{CFL} - \mathrm{EDT0L}$. Examples of such L_0 are D_2^*, the one-sided Dyck language over two pairs of parentheses [2,3] and D_1^*, the one-sided Dyck language over a single pair of parentheses [15]. A question that immediately arises is the status of the "double copying theorem" for OI: when is $L = \{ w\#w \mid w \in L_0 \}$ in OI? We do not yet have an answer to this open question. In this paper, we prove a double copying theorem for well-nested multiple context-free languages, which implies, among other things, that $L = \{ w\#w \mid w \in L_0 \} \in \text{2-MCFL} - \mathrm{MCFL}_{\mathrm{wn}}$ for all $L_0 \in \mathrm{CFL} - \mathrm{EDT0L}$. Unlike Staudacher's [20] proof, our proof of this result does not depend on a pumping argument but instead makes use of simple combinatorial properties of strings.

In addition to shedding light on the difference between the class of MCFLs and the class of well-nested MCFLs, the double copying theorem for well-nested MCFLs also highlights a general question underlying Joshi's notion of mild context-sensitivity: what are the limitations found in the kind of cross-serial dependency exhibited in natural language? For, if $\mathcal{L}$ is a family of languages closed under rational transductions, $\{ w\#w \mid w \in L_0 \} \in \mathcal{L}$ implies $\{ w\,h(w) \mid w \in L_0 \} \in \mathcal{L}$ for any homomorphism h, and languages of the latter form, together with some restriction on L_0, may serve as a model of natural language constructions exhibiting cross-serial dependency. This may offer a more fruitful approach than concentrating on languages like $\{ w\,h(w) \mid w \in \Sigma^* \}$, which has been a common practice in the mathematical study of natural language syntax.

2 The Double Copying Theorem for Context-Free Languages

Let us first look at the double copying theorem for context-free languages. This has a rather simple proof, which is omitted here in the interests of space. The implication (i) ⇒ (iii) may be proved using the pumping lemma for context-free languages; it also follows from a closely related result proved by Ito and Katsura [8].

[1] See [3] for the definition of EDT0L.

Theorem 1. *Let $L = \{ w\#w \mid w \in L_0 \}$. The following are equivalent:*

(i) *L is a context-free language.*
(ii) *L is a linear context-free language.*
(iii) *L_0 is a finite union of languages of the form rRs, where $r, s \in \Sigma^*$ and R is a regular subset of t^* for some $t \in \Sigma^+$.*

3 Combinatorics on Words

The statement of the double copying theorem for well-nested multiple context-free languages is similar to that for context-free languages, but we do not need to invoke the pumping lemma for well-nested MCFLs in order to prove it. Instead, we rely on some basic results in the combinatorics on words.

A string x is a *conjugate* of a string y if $x = uv$ and $y = vu$ for some u, v. Elements of u^* are called *powers* of u. A nonempty string is *primitive* if it is not a power of another string. For every nonempty string x, there is a unique primitive string u such that x is a power of u; this string u is called the *primitive root* of x. When two nonempty strings are conjugates, their primitive roots are also conjugates.

We use the following basic results from the combinatorics on words (see, e.g., [19]):

Lemma 2. *Let $x, y, z \in \Sigma^+$. Then $xy = yz$ if and only if there exist $u \in \Sigma^+$, $v \in \Sigma^*$, and an integer $k \geq 0$ such that $x = uv$, $z = vu$, and $y = (uv)^k u = u(vu)^k$.*

Lemma 3. *Let $x, y \in \Sigma^+$. The following are equivalent:*

(i) *$xy = yx$.*
(ii) *There exist $z \in \Sigma^+$ and $i, j \geq 1$ such that $x = z^i$ and $y = z^j$.*
(iii) *There exist $i, j \geq 1$ such that $x^i = y^j$.*

4 Multiple Context-Free Grammars

A *ranked alphabet* is a finite set $\Delta = \bigcup_{n \geq 0} \Delta^{(n)}$ such that $\Delta^{(i)} \cap \Delta^{(j)} = \varnothing$ if $i \neq j$. An element d of Δ has *rank* n if $d \in \Delta^{(n)}$. A *tree* over a ranked alphabet Δ is an expression of the form $(dT_1 \dots T_n)$, where $d \in \Delta^{(n)}$ and $T_1, \dots, T_n$ are trees over Δ; the parentheses are omitted when $n = 0$. In writing trees, we adopt the abbreviatory convention of dropping the outermost parentheses.

Let Δ be a ranked alphabet and Σ an unranked alphabet. Let $\mathcal{X}$ be a countably infinite set of *variables* ranging over Σ^*. We use boldface italic letters $\boldsymbol{x}_1, \boldsymbol{y}_1, \boldsymbol{z}_1$, etc., as variables in $\mathcal{X}$. A *rule* over Δ, Σ is an expression of the form

$$A(\alpha_1, \dots, \alpha_q) :- B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n}),$$

where $n \geq 0$, $A \in \Delta^{(q)}$, $B_i \in \Delta^{(q_i)}$, $\boldsymbol{x}_{i,j}$ are pairwise distinct variables, and α_i is a string over $\Sigma \cup \{ \boldsymbol{x}_{i,j} \mid i \in [1, n], j \in [1, q_i] \}$ satisfying the following condition:

– for each i, j, the variable $\boldsymbol{x}_{i,j}$ occurs in $\alpha_1 \dots \alpha_q$ at most once.

Rules with $n = 0$ are called *terminating* and written without the :– symbol. When we deal with rules over Δ, Σ, we view elements of Δ as predicates, and call q the *arity* of A if $A \in \Delta^{(q)}$. Thus, rules are *definite clauses* (in the sense of logic programming) built from strings and predicates on strings.

A *multiple context-free grammar* (MCFG) is a quadruple $G = (N, \Sigma, P, S)$, where N is a ranked alphabet of *nonterminals*, Σ is an unranked alphabet of *terminals*, P is a finite set of rules over N, Σ, and $S \in N^{(1)}$. When $A \in N^{(q)}$ and $w_1, \dots, w_q \in \Sigma^*$, we write $\vdash_G A(w_1, \dots, w_q)$ to mean that $A(w_1, \dots, w_q)$ is derivable using the following inference schema:

$$\frac{\vdash_G B_1(w_{1,1}, \dots, w_{1,q_1}) \quad \dots \quad \vdash_G B_n(w_{n,1}, \dots, w_{n,q_n})}{\vdash_G A(\alpha_1, \dots, \alpha_q)\sigma}$$

where $A(\alpha_1, \dots, \alpha_q) \coloneq B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n})$ is in P and σ is the substitution mapping each $\boldsymbol{x}_{i,j}$ to $w_{i,j}$. The language of G is defined as $L(G) = \{\, w \in \Sigma^* \mid \vdash_G S(w) \,\}$.

In order to speak of derivation trees of derivable facts, we put the elements of P in one-to-one correspondence with the elements of a ranked alphabet Δ_P, so that a rule $\pi \in P$ with n occurrences of nonterminals on the right-hand side corresponds to a symbol in $\Delta_P^{(n)}$, which we confuse with π itself. In order to refer to contexts in which derivation trees appear, we augment Δ_P with a set $\mathbf{Y}$ of variables ($\mathbf{y}, \mathbf{z}$, etc.), whose rank is always 0. The following inference system associates derivation trees (trees over Δ_P) with derivable facts and derivation tree contexts (trees over $\Delta_P \cup \mathbf{Y}$) with facts derivable from some premises:

$$\frac{}{\mathbf{y} : A(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q) \vdash_G \mathbf{y} : A(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q)}$$

$$\frac{\Gamma_1 \vdash_G T_1 : B_1(\beta_{1,1}, \dots, \beta_{1,q_1}) \quad \dots \quad \Gamma_n \vdash_G T_n : B_n(\beta_{n,1}, \dots, \beta_{n,q_n})}{\Gamma_1, \dots, \Gamma_n \vdash_G \pi T_1 \dots T_n : A(\alpha_1, \dots, \alpha_q)\sigma}$$

In the second schema, π is the rule

$$A(\alpha_1, \dots, \alpha_q) \coloneq B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n})$$

and σ is the substitution mapping each $\boldsymbol{x}_{i,j}$ to $\beta_{i,j}$; each Γ_i is a finite sequence of premises of the form $\mathbf{z} : C(\boldsymbol{y}_1, \dots, \boldsymbol{y}_p)$, and it is understood that Γ_i and Γ_j do not share any variables if $i \neq j$. It is clear that $\vdash_G A(w_1, \dots, w_q)$ if and only if $\vdash_G T : A(w_1, \dots, w_q)$ for some tree T over Δ_P. The set $\{\, T \mid \vdash_G T : S(w)$ for some $w \in \Sigma^* \,\}$ is a recognizable set of trees; as a consequence, the Parikh image of $L(G)$ is semilinear [21].

A nonterminal $A \in N^{(q)}$ is *useful* if $\vdash_G A(w_1, \dots, w_q)$ for some $w_1, \dots, w_q$ and $\mathbf{y} : A(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q) \vdash_G T : S(\alpha)$ for some T and α; otherwise it is *useless*.

Example 4. Let G be the MCFG consisting of the following rules:

$$\pi_1 \colon S(\boldsymbol{x}_1 \boldsymbol{y}_1 \mathtt{b\#a} \boldsymbol{y}_2 \boldsymbol{x}_2) \coloneq A(\boldsymbol{x}_1, \boldsymbol{x}_2), B(\boldsymbol{y}_1, \boldsymbol{y}_2).$$
$$\pi_2 \colon A(\mathtt{a}, \epsilon). \quad \pi_3 \colon A(\mathtt{ab}\boldsymbol{x}_1\mathtt{ba}, \mathtt{ab}\boldsymbol{x}_2\mathtt{ab}) \coloneq A(\boldsymbol{x}_1, \boldsymbol{x}_2).$$
$$\pi_4 \colon B(\epsilon, \mathtt{b}). \quad \pi_5 \colon B(\mathtt{ba}\boldsymbol{y}_1\mathtt{ba}, \mathtt{ba}\boldsymbol{y}_2\mathtt{ab}) \coloneq B(\boldsymbol{y}_1, \boldsymbol{y}_2).$$

For example,

$$\vdash_G \pi_1(\pi_3(\pi_3\pi_2))(\pi_5\pi_4) : S(\texttt{abababababababab\#abababababababab}),$$
$$\mathbf{y} : A(\boldsymbol{x}_1, \boldsymbol{x}_2) \vdash_G \pi_1(\pi_3\mathbf{y})(\pi_5\pi_4) : S(\texttt{ab}\boldsymbol{x}_1\texttt{bababab\#abababab}\boldsymbol{x}_2\texttt{ab}),$$

and we have $L(G) = \{\, (\texttt{ab})^n\texttt{\#}(\texttt{ab})^n \mid n \geq 1 \,\}$.

The *dimension* of an MCFG G is the maximal arity of nonterminals of G. The *branching factor* (or *rank*) of G is the maximal number of occurrences of nonterminals on the right-hand side of rules of G. We write m-MCFG(f) for the class of MCFGs whose dimension is at most m and whose branching factor is at most f. (Note that this notation is the opposite of the one used by Seki and Kato [17], but is more consistent with [18].) We write m-MCFG and MCFG for $\bigcup_f m$-MCFG(f) and $\bigcup_m \bigcup_f m$-MCFG(f), respectively. The corresponding classes of languages are denoted by m-MCFL(f), m-MCFL, etc.[2]

An MCFG rule

$$A(\alpha_1, \dots, \alpha_q) :\!-\; B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n})$$

is *non-deleting* if each $\boldsymbol{x}_{i,j}$ occurs in $\alpha_1 \dots \alpha_q$; it is *non-permuting* if $j < k$ implies that the occurrence (if any) of $\boldsymbol{x}_{i,j}$ in $\alpha_1 \dots \alpha_q$ precedes the occurrence (if any) of $\boldsymbol{x}_{i,k}$ in $\alpha_1 \dots \alpha_q$. It is known that every $G \in m$-MCFG(f) has an equivalent $G' \in m$-MCFG(f) whose rules are all non-deleting and non-permuting. A non-deleting and non-permuting rule is *well-nested* if it moreover satisfies the following condition:

– if $i \neq i'$, $j < q_i$, and $j' < q_{i'}$, then $\alpha_1 \dots \alpha_q \notin (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{i,j} (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{i',j'} (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{i,j+1} (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{i',j'+1} (\Sigma \cup \mathcal{X})^*$.

In other words, if $\boldsymbol{x}_{i',j'}$ occurs between $\boldsymbol{x}_{i,j}$ and $\boldsymbol{x}_{i,j+1}$ in $\alpha_1 \dots \alpha_q$, then $\boldsymbol{x}_{i',1}, \dots, \boldsymbol{x}_{i',q_{i'}}$ must all occur between $\boldsymbol{x}_{i,j}$ and $\boldsymbol{x}_{i,j+1}$.

We attach the subscript "wn" to "MCFG" and "MCFL" to denote classes of well-nested MCFGs and corresponding classes of languages, as in m-MCFG$_{\text{wn}}$(f), m-MCFG$_{\text{wn}}$, m-MCFL$_{\text{wn}}$(f), m-MCFL$_{\text{wn}}$, etc. The grammar in Example 4 belongs to 2-MCFG$_{\text{wn}}$(2). Note that m-MCFL(1) = m-MCFL$_{\text{wn}}$(1).

Lemma 5. *For each $m \geq 1$, m-MCFL$_{wn}$ = m-MCFL$_{wn}$(2).*

Proof (sketch). A well-nested rule

$$\pi = A(\alpha_1, \dots, \alpha_q) :\!-\; B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n})$$

with $n \geq 3$ can always be replaced by two rules whose right-hand side has at most $n-1$ nonterminals, as follows. The replacement introduces one new nonterminal C, whose arity does not exceed $\max\{q, q_1, \dots, q_n\}$. We assume without loss of

[2] See [14] and [16] for relations among the classes m-MCFL(f) with different values of m and f.

generality that $1 \le i < j \le n$ implies that $\boldsymbol{x}_{i,1}$ occurs to the left of $\boldsymbol{x}_{j,1}$ in $\alpha_1 \dots \alpha_q$. Since π is well-nested, there must be an $l \in [1, n]$ such that $\alpha_1 \dots \alpha_q \in (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{l,1} \Sigma^* \boldsymbol{x}_{l,2} \Sigma^* \dots \Sigma^* \boldsymbol{x}_{l,q_l} (\Sigma \cup \mathcal{X})^*$. Let i, j be such that $\alpha_i \in (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{l,1} (\Sigma \cup \mathcal{X})^*$ and $\alpha_j \in (\Sigma \cup \mathcal{X})^* \boldsymbol{x}_{l,q_l} (\Sigma \cup \mathcal{X})^*$.

Case 1. $i < j$. We can write $\alpha_i = \beta_1 \boldsymbol{x}_{l,1} \beta_2$ and $\alpha_j = \gamma_1 \boldsymbol{x}_{l,q_l} \gamma_2$. Let C be a new nonterminal of arity $q' = i + q - j + 1 \le q$. We can replace π with the following two rules:

$$\begin{aligned}
&B(\boldsymbol{y}_1, \dots, \boldsymbol{y}_{i-1}, \boldsymbol{y}_i \boldsymbol{x}_{l,1} \beta_2, \alpha_{i+1}, \dots, \alpha_{j-1}, \gamma_1 \boldsymbol{x}_{l,q_l} \boldsymbol{y}_j, \boldsymbol{y}_{j+1}, \dots, \boldsymbol{y}_q) :- \\
&\qquad C(\boldsymbol{y}_1, \dots, \boldsymbol{y}_i, \boldsymbol{y}_j, \dots, \boldsymbol{y}_q), B_l(\boldsymbol{x}_{l,1}, \dots, \boldsymbol{x}_{l,q_l}). \\
&C(\alpha_1, \dots, \alpha_{i-1}, \beta_1, \gamma_2, \alpha_{j+1}, \dots, \alpha_q) :- \\
&\qquad B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_{l-1}(\boldsymbol{x}_{l-1,1}, \dots, \boldsymbol{x}_{l-1,q_{l-1}}), \\
&\qquad B_{l+1}(\boldsymbol{x}_{l+1,1}, \dots, \boldsymbol{x}_{l+1,q_{l+1}}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n}).
\end{aligned}$$

Case 2. $i = j$. We can write $\alpha_i = \beta_1 \boldsymbol{x}_{l,1} \beta_2 \boldsymbol{x}_{l,q_l} \beta_3$.

Case 2a. $\beta_1 \beta_3 \in \Sigma^*$. Let C be a new nonterminal of arity $q - 1$. We can replace π with the following two rules:

$$\begin{aligned}
&A(\boldsymbol{y}_1, \dots, \boldsymbol{y}_{i-1}, \alpha_i, \boldsymbol{y}_{i+1}, \dots, \boldsymbol{y}_q) :- \\
&\qquad C(\boldsymbol{y}_1, \dots, \boldsymbol{y}_{i-1}, \boldsymbol{y}_{i+1}, \dots, \boldsymbol{y}_q), B_l(\boldsymbol{x}_{l,1}, \dots, \boldsymbol{x}_{l,q_l}). \\
&C(\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_q) :- \\
&\qquad B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_{l-1}(\boldsymbol{x}_{l-1,1}, \dots, \boldsymbol{x}_{l-1,q_{l-1}}), \\
&\qquad B_{l+1}(\boldsymbol{x}_{l+1,1}, \dots, \boldsymbol{x}_{l+1,q_{l+1}}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n}).
\end{aligned}$$

Case 2b. $\beta_1 = \gamma \boldsymbol{x}_{k,p} w$ with $w \in \Sigma^*$. Let C be a new nonterminal of arity q_k. We can replace π with the following two rules:

$$\begin{aligned}
&A(\alpha_1, \dots, \alpha_{i-1}, \gamma \boldsymbol{x}_{k,p} \beta_3, \alpha_{p+1}, \dots, \alpha_q) :- \\
&\qquad B_1(\boldsymbol{x}_{1,1}, \dots, \boldsymbol{x}_{1,q_1}), \dots, B_{k-1}(\boldsymbol{x}_{k-1,1}, \dots, \boldsymbol{x}_{k-1,q_{k-1}}), C(\boldsymbol{x}_{k,1}, \dots, \boldsymbol{x}_{k,q_k}), \\
&\qquad B_{k+1}(\boldsymbol{x}_{k+1,1}, \dots, \boldsymbol{x}_{k+1,q_{k+1}}), \dots, B_{l-1}(\boldsymbol{x}_{l-1,1}, \dots, \boldsymbol{x}_{l-1,q_{l-1}}), \\
&\qquad B_{l+1}(\boldsymbol{x}_{l+1,1}, \dots, \boldsymbol{x}_{l+1,q_{l+1}}), \dots, B_n(\boldsymbol{x}_{n,1}, \dots, \boldsymbol{x}_{n,q_n}). \\
&C(\boldsymbol{x}_{k,1}, \dots, \boldsymbol{x}_{k,p-1}, \boldsymbol{x}_{k,p} w \boldsymbol{x}_{l,1} \beta_2 \boldsymbol{x}_{l,q_l}, \boldsymbol{x}_{k,p+1}, \dots, \boldsymbol{x}_{k,q_k}) :- \\
&\qquad B_k(\boldsymbol{x}_{k,1}, \dots, \boldsymbol{x}_{k,q_k}), B_l(\boldsymbol{x}_{l,1}, \dots, \boldsymbol{x}_{q_l}).
\end{aligned}$$

Case 2c. $\beta_3 = w \boldsymbol{x}_{k,p} \gamma$ with $w \in \Sigma^*$. Similar to Case 2b. □

Seki and Kato [17] show that for all $m \ge 2$, $\mathrm{RESP}_m \in m\text{-MCFL}(2) - m\text{-MCFL}_{\mathrm{wn}}$, where RESP_m is defined by

$$\mathrm{RESP}_m = \{ \mathtt{a}_1^i \mathtt{a}_2^i \mathtt{b}_1^j \mathtt{b}_2^j \dots \mathtt{a}_{2m-1}^i \mathtt{a}_{2m}^i \mathtt{b}_{2m-1}^j \mathtt{b}_{2m}^j \mid i, j \ge 0 \}.$$

It is easy to see that $\mathrm{RESP}_m \in 2m\text{-MCFL}(1) = 2m\text{-MCFL}_{\mathrm{wn}}(1)$.

5 The Double Copying Theorem for Well-Nested Multiple Context-Free Languages

The following theorem about possibly non-well-nested MCFGs is easy to prove. For part (ii), note that there is a rational transduction that maps L to L_0.[3]

Theorem 6. *Let* $L = \{\, w\#w \mid w \in L_0 \,\}$.

(i) *If* $L_0 \in m\text{-}MCFL(f)$, *then* $L \in 2m\text{-}MCFL(f)$.
(ii) *If* $L \in m\text{-}MCFL(f)$, *then* $L_0 \in m\text{-}MCFL(f)$.

A consequence of Theorem 6 is that the class of all MCFGs has an unlimited copying power in the sense that $L = \{\, w\#w \mid w \in L_0 \,\}$ is an MCFL whenever L_0 is. We will see that the copying power of well-nested MCFGs is much more restricted (Corollary 9).

The following lemma is used in the proof of our main theorem (Theorem 8). Its proof is straightforward and is omitted.

Lemma 7. *Let M be a semilinear subset of $\mathbb{N}^{2m}$ and $r_i, s_i, t_i, u_i, v_i \in \Sigma^*$ for $i \in [1, m]$. Then there are some $G = (N, \Sigma, P, S) \in m\text{-}MCFG(1)$ and nonterminal $A \in N^{(m)}$ such that*

$$\{\, (x_1, \dots, x_m) \mid \vdash_G A(x_1, \dots, x_m) \,\} = \\ \{\, (r_1 s_1^{n_1} t_1 u_1^{n_2} v_1, \dots, r_m s_m^{n_{2m-1}} t_m u_m^{n_{2m}} v_m) \mid (n_1, \dots, n_{2m}) \in M \,\}.$$

Theorem 8. *Let* $L = \{\, w\#w \mid w \in L_0 \,\}$. *The following are equivalent:*

(i) $L \in m\text{-}MCFL_{wn}$.
(ii) $L \in m\text{-}MCFL(1)$.

Proof. The implication from (ii) to (i) immediately follows from m-MCFL(1) $=$ m-MCFL$_{\mathrm{wn}}$(1). To show that (i) implies (ii), suppose that $L = L(G)$ for some $G = (N, \Sigma \cup \{\#\}, P, S) \in m\text{-MCFG}_{\mathrm{wn}}(2)$. If L is finite, L clearly belongs to 1-MCFL(1), so we assume that L is infinite. Without loss of generality, we may suppose that G has no useless nonterminal and satisfies the following property:

– For each nonterminal $A \in N^{(q)}$, the set $\{\, (x_1, \dots, x_q) \mid \vdash_G A(x_1, \dots, x_q) \,\}$ is infinite.

To show that L belongs to m-MCFL(1), we prove that for each binary rule

$$\pi = A(\alpha_1, \dots, \alpha_q) :\!-\, B(\boldsymbol{y}_1, \dots, \boldsymbol{y}_k), C(\boldsymbol{z}_1, \dots, \boldsymbol{z}_l)$$

in P, there are $G_\pi = (N_\pi, \Sigma \cup \{\#\}, P_\pi, S_\pi) \in m$-MCFG(1) and a nonterminal $A_\pi \in N_\pi^{(q)}$ such that

$$\{\, (x_1, \dots, x_q) \mid \vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q) \text{ for some derivation trees } T_1, T_2 \,\} \\ = \{\, (x_1, \dots, x_q) \mid \vdash_{G_\pi} A_\pi(x_1, \dots, x_q) \,\}. \quad (1)$$

This is a consequence of the following claim. We assume without loss of generality that $\boldsymbol{y}_1$ occurs to the left of $\boldsymbol{z}_1$ in $(\alpha_1, \dots, \alpha_q)$.

[3] See the discussion following the proof of Theorem 8 for a possible strengthening of part (ii) of Theorem 6.

Claim. There exist $t \in \Sigma^+$ and $r, s \in \Sigma^*$ such that if

$$\vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q)$$

for some T_1, T_2, then $x_1, \dots, x_q$ are non-overlapping substrings of $rt^i s\#rt^i s$ for some $i \geq 0$.

Proof. We write $\Sigma_\#$ for $\Sigma \cup \{\#\}$. Let $U[\mathbf{x}]$ be a (smallest, for concreteness) derivation tree context such that for some $\gamma \in \Sigma_\#^* \boldsymbol{x}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{x}_q \Sigma_\#^*$,

$$\mathbf{x} : A(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q) \vdash_G U[\mathbf{x}] : S(\gamma).$$

We write $\gamma[\vec{\beta}]$ for $\gamma[\boldsymbol{x}_1 := \beta_1, \dots, \boldsymbol{x}_q := \beta_q]$. Our goal is to find $t \in \Sigma^+$ and $r, s \in \Sigma^*$ such that

$$\vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q) \text{ implies } \gamma[\vec{x}] = rt^i s\#rt^i s \text{ for some } i \geq 0. \tag{2}$$

We have

$$\mathbf{y} : B(\boldsymbol{y}_1, \dots, \boldsymbol{y}_k), \mathbf{z} : C(\boldsymbol{z}_1, \dots, \boldsymbol{z}_l) \vdash_G U[\pi\mathbf{y}\mathbf{z}] : S(\gamma[\vec{\alpha}]).$$

Let us write $\gamma[\vec{\alpha}][\vec{y}, \vec{z}]$ for the result of substituting $y_1, \dots, y_k, z_1, \dots, z_l$ for $\boldsymbol{y}_1, \dots, \boldsymbol{y}_k, \boldsymbol{z}_1, \dots, \boldsymbol{z}_l$ in $\gamma[\vec{\alpha}]$. Since π is well-nested, either

$$\gamma[\vec{\alpha}] \in \Sigma_\#^* \boldsymbol{y}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{y}_k \Sigma_\#^* \boldsymbol{z}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{z}_l \Sigma_\#^*$$

or else

$$\gamma[\vec{\alpha}] \in \Sigma_\#^* \boldsymbol{y}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{y}_h \Sigma_\#^* \boldsymbol{z}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{z}_l \Sigma_\#^* \boldsymbol{y}_{h+1} \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{y}_k \Sigma_\#^*$$

for some $h \in [1, k-1]$. Since $\gamma[\vec{\alpha}][\vec{y}, \vec{z}] \in L$ for all $y_1, \dots, y_k, z_1, \dots, z_l$ such that $\vdash_G B(y_1, \dots, y_k)$ and $\vdash_G C(z_1, \dots, z_l)$, and $y_1, \dots, y_k$ and $z_1, \dots, z_l$ can vary independently, it is easy to see that the former possibility is ruled out; thus we must have

$$\gamma[\vec{\alpha}] = \delta_1 \delta_2 \delta_3$$

where

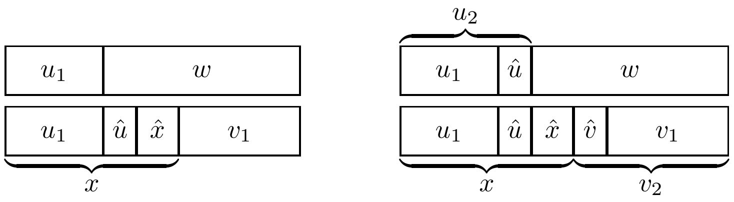
$$\delta_1 \in \Sigma^* \boldsymbol{y}_1 \Sigma^* \dots \Sigma^* \boldsymbol{y}_h \Sigma^*, \quad \delta_2 \in \boldsymbol{z}_1 \Sigma_\#^* \dots \Sigma_\#^* \boldsymbol{z}_l, \quad \delta_3 \in \Sigma^* \boldsymbol{y}_{h+1} \Sigma^* \dots \Sigma^* \boldsymbol{y}_k \Sigma^*.$$

Let

$$L_B = \{\, \delta_1 \# \delta_3[\vec{y}] \mid \vdash_G B(y_1, \dots, y_k) \,\} \quad \text{and} \quad L_C = \{\, \delta_2[\vec{z}] \mid \vdash_G C(z_1, \dots, z_l) \,\}.$$

Note that both L_B and L_C are infinite subsets of $\Sigma^* \# \Sigma^*$, and for every $u\#v \in L_B$ and $w\#x \in L_C$, the string $uw\#xv$ is an element of L. Let $u\#v, u'\#v' \in L_B$ with $|u| \leq |u'|$. By taking $w\#x \in L_C$ with $|w| \geq |v'|$ (or equivalently, $|x| \geq |u'|$), we see that u must be a prefix of u', since both are prefixes of x. We also see that $|u'| - |u| = |v'| - |v|$. By the same token, v must be a suffix of v'.

Let $u_1\#v_1$ and $u_2\#v_2$ be the two shortest strings in L_B. Then $u_2 = u_1\hat{u}$ and $v_2 = \hat{v}v_1$ for some $\hat{u}, \hat{v} \in \Sigma^+$ such that $|\hat{u}| = |\hat{v}|$. Let $w\#x \in L_C$, and suppose $|x| > |u_2|$.



From $u_1w = xv_1$ and $u_2w = xv_2$, we see that there is an $\hat{x} \in \Sigma^+$ such that

$$x = u_2\hat{x}, \quad w = \hat{x}v_2, \quad \text{and} \quad \hat{u}\hat{x} = \hat{x}\hat{v}.$$

By Lemma 2, there are $\hat{u}_1 \in \Sigma^+, \hat{u}_2 \in \Sigma^*$ such that

$$\hat{u} = \hat{u}_1\hat{u}_2, \quad \hat{v} = \hat{u}_2\hat{u}_1, \quad \text{and} \quad \hat{x} = \hat{u}^k\hat{u}_1 = \hat{u}_1(\hat{u}_2\hat{u}_1)^k \quad \text{for some } k \geq 0.$$

Now let t be the primitive root of $\hat{u}$. There are some $i_1, i_2 \geq 0$ and t_1, t_2 such that $t_1 \neq \epsilon$ and

$$t = t_1t_2, \quad \hat{u}_1 = t^{i_1}t_1, \quad \hat{u}_2 = t_2t^{i_2}.$$

Then

$$\hat{u}\hat{x} = \hat{x}\hat{v} = \hat{u}^{k+1}\hat{u}_1 \in t^*t_1.$$

It follows that for all $w\#x \in L_C$ such that $|x| > |u_2|$,

$$w \in t^*t_1v_1, \tag{3}$$

$$x \in u_1t^*t_1. \tag{4}$$

Now let $u\#v$ be an arbitrary element of L_B. Take $w\#x \in L_C$ such that $|x| > |u_2|$ and $|w| \geq |t| + |v|$. Since $uw = xv$, there is an x' such that $|x'| \geq |t|$ and

$$w = x'v, \tag{5}$$

$$x = ux'. \tag{6}$$

Since $|v| \geq |v_1|$ and $|x'| \geq |t|$, (3) and (5) implies

$$x' = t_1(t_2t_1)^jt_3$$

for some $j \geq 0$ and some prefix t_3 of t_2t_1 such that $t_3 \neq t_2t_1$. Let t_4 be such that $t_3t_4 = t_2t_1$. Since (4) and (6) imply that x' ends in t_2t_1, we see

$$t_4t_3 = t_3t_4.$$

Since $t_3t_4 = t_2t_1$ is a conjugate of t and hence is primitive, Lemma 3 implies that $t_3 = \epsilon$. Hence $x' \in t^*t_1$. By (4) and (6), we see

$$u \in u_1t^*. \tag{7}$$

By a reasoning symmetric to that leading up to (7), we can infer that there exist some primitive non-empty string $\tilde{t}$ and some string w_1 such that for all $w\#x \in L_C$,

$$w \in \tilde{t}^* w_1. \tag{8}$$

By taking sufficiently long w, (3) and (8) together imply

$$t^{|\tilde{t}|} = \tilde{t}^{|t|}.$$

Since t and $\tilde{t}$ are both primitive, Lemma 3 implies $t = \tilde{t}$. Thus, for all $w\#x \in L_C$,

$$w \in t^* w_1. \tag{9}$$

From (7) and (9), we obtain

$$uw \in u_1 t^* w_1$$

for all $u\#v \in L_B$ and all $w\#x \in L_C$. Now (2) follows with $r = u_1$ and $s = w_1$. □

We continue with the proof of Theorem 8. Let $c = \max\{|r|, |s|, |t|\}$. By the above claim, one of the following two cases must obtain.

Case 1. Every $(x_1, \dots, x_q)$ such that $\vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q)$ for some T_1, T_2 is of the form

$$(r_1 t^{n_1} s_1, \dots, r_q t^{n_q} s_q).$$

for some $r_1, \dots, r_q, s_1, \dots, s_q \in \Sigma^{\leq c}$.

Case 2. Every $(x_1, \dots, x_q)$ such that $\vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q)$ for some T_1, T_2 is of the form

$$\begin{aligned}(r_1 t^{n_1} s_1, \dots, r_{j-1} t^{n_{j-1}} s_{j-1}, r_j t^{n_j} s_j \# r_{j+1} t^{n_{j+1}} s_{j+1},\\ r_{j+2} t^{n_{j+2}} s_{j+2}, \dots, r_{q+1} t^{n_{q+1}} s_{q+1})\end{aligned}$$

for some $r_1, \dots, r_{q+1}, s_1, \dots, s_{q+1} \in \Sigma^{\leq c}$.

In Case 1, for any fixed $r_1, \dots, r_q, s_1, \dots, s_q$, the set

$$\{\, (n_1, \dots, n_q) \mid \vdash_G \pi T_1 T_2 : A(r_1 t^{n_1} s_1, \dots, r_q t^{n_q} s_q) \text{ for some } T_1, T_2 \,\}$$

is semilinear. To see this, it suffices to note that $L_\pi = \{\, x_1 \$ \dots \$ x_q \mid \vdash_G \pi T_1 T_2 : A(x_1, \dots, x_q) \text{ for some } T_1, T_2 \,\}$ is an m-MCFL and there is a rational transduction that relates $r_1 t^{n_1} s_1 \$ \dots \$ r_q t^{n_q} s_q$ to $\mathtt{a}_1^{n_1} \dots \mathtt{a}_q^{n_q}$. Thus, by Lemma 7, there are a $G_\pi = (N_\pi, \Sigma \cup \{\#\}, P_\pi, S_\pi) \in q$-MCFG(1) and a nonterminal $A_\pi \in N_\pi^{(q)}$ such that (1) holds.

In Case 2, we can derive the same conclusion in a similar way.

Let P_2 be the set of all binary rules of G. We can now form a $G' = (N', \Sigma \cup \{\#\}, P', S) \in m$-MCFG(1) generating L by setting

$$N' = N \cup \bigcup_{\pi \in P_2} N_\pi,$$

$$\begin{aligned}P' = (P - P_2) \cup \bigcup_{\pi \in P_2} P_\pi \cup \{\, A(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q) \coloneq A_\pi(\boldsymbol{x}_1, \dots, \boldsymbol{x}_q) \mid \pi \in P_2 \\ \text{and } A \in N^{(q)} \text{ is the head nonterminal of } \pi \,\}.\end{aligned}$$

This completes the proof of Theorem 8. □

It would be desirable to have a precise characterization of the class of languages L_0 for which $L = \{ w\#w \mid w \in L_0 \}$ belongs to m-MCFL$_{\text{wn}}$, as in the double copying theorem for context-free languages (Theorem 1). In a previous version of the paper, we hastily stated that $L \in m\text{-MCFL}(f)$ implies $L_0 \in \lceil m/2 \rceil\text{-MCFL}(f)$ (compare part (ii) of Theorem 6), which would give us such a characterization for even m. While this still seems to us to be a reasonable conjecture, we currently see no easy way to prove it.

Since it is easy to see[4]

$$m\text{-MCFL}(1) = \text{EDT0L}_{\text{FIN}(m)},$$

Theorems 6 and 8 give

Corollary 9. *Let $L = \{ w\#w \mid w \in L_0 \}$. The following are equivalent:*

(i) $L \in \mathit{MCFL}_{wn}$.
(ii) $L \in \mathit{EDT0L}_{FIN}$.
(iii) $L_0 \in \mathit{EDT0L}_{FIN}$.

Since CFL − EDT0L $\neq \varnothing$ and $\{ w\#w \mid w \in L_0 \} \in$ 2-MCFL for all $L_0 \in$ CFL − EDT0L, Corollary 9 implies

Corollary 10. $2\text{-}\mathit{MCFL} - \mathit{MCFL}_{wn} \neq \varnothing$.

6 Conclusion

We have shown that imposing the well-nestedness constraint on the rules of multiple context-free grammars causes severe loss of the copying power of the formalism. The restriction on the languages L_0 that can be copied is similar to the restriction in Engelfriet and Skyum's [4] triple copying theorem for OI. It is worth noting that the crucial claim in the proof of Theorem 8 does not depend on the non-duplicating nature of the MCFG rules, and one can indeed prove that an analogous claim also holds of OI. This leads us to conjecture that a double copying theorem holds of OI with the same restriction on L_0 as in Engelfriet and Skyum's triple copying theorem (namely, membership in EDT0L).[5] We hope to resolve this open question in future work.

References

1. Arnold, A., Dauchet, M.: Un théorem de duplication pour les forêts algébriques. Journal of Computer and System Science 13, 223–244 (1976)
2. Ehrenfeucht, A., Rozenberg, G.: On some context-free languages that are not deterministic ET0L languages. R.A.I.R.O. Informatique théorique/Theoretical Computer Science 11, 273–291 (1977)

[4] See [3] for the definition of EDT0L$_{\text{FIN}(m)}$ and EDT0L$_{\text{FIN}}$.

[5] Arnold and Dauchet [1] prove a copying theorem for *OI context-free tree languages*, which is an exact tree counterpart to this conjecture.

3. Engelfriet, J., Rozenberg, G., Slutzki, G.: Tree transducers, L systems, and two-way machines. Journal of Computer and System Sciences 20, 150–202 (1980)
4. Engelfriet, J., Skyum, S.: Copying theorems. Information Processing Letters 4, 157–161 (1976)
5. Fisher, M.J.: Grammars with Macro-Like Productions. Ph.D. thesis, Harvard University (1968)
6. Hayashi, T.: On derivation trees of indexed gramamrs —an extension of the uvwxy-theorem—. Publications of the Research Institute for Mathematical Sciences 9, 61–92 (1973)
7. Hotz, G., Pitsch, G.: On parsing coupled-context-free languages. Thoretical Computer Science 161, 205–253 (1996)
8. Ito, M., Katsura, M.: Context-free languages consisting of non-primitive words. International Journal of Computer Mathematics 40, 157–167 (1991)
9. Joshi, A.K.: Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions? In: Dowty, D.R., Karttunen, L., Zwicky, A.M. (eds.) Natural Language Parsing: Psychological, Computational and Theoretical Perspectives, pp. 206–250. Cambridge University Press, Cambridge (1985)
10. Kanazawa, M.: The convergence of well-nested mildly context-sensitive grammar formalisms. In: An invited talk given at the 14th Conference on Formal Grammar, Bordeaux, France (July 2009), http://research.nii.ac.jp/~kanazawa/
11. Kanazawa, M.: The pumping lemma for well-nested multiple context-free languages. In: Diekert, V., Nowotka, D. (eds.) Developments in Language Theory: 13th International Conference, DLT 2009, pp. 312–325. Springer, Berlin (2009)
12. Kuhlmann, M.: Dependency Structures and Lexicalized Grammars. Ph.D. thesis, Saarland University (2007)
13. Michaelis, J.: An additional observation on strict derivational minimalism. In: Rogers, J. (ed.) Proceedings of FG-MoL 2005: The 10th conference on Formal Grammar and the 9th Meeting on Mathematics of Language, pp. 101–111. CSLI Publications, Stanford (2009)
14. Rambow, O., Satta, G.: Independent parallelism in finite copying parallel rewriting systems. Theoretical Computer Science 223, 87–120 (1999)
15. Rozoy, B.: The Dyck language ${D'_1}^*$ is not generated by any matrix grammar of finite index. Information and Computation 74, 64–89 (1987)
16. Satta, G.: Trading independent for synchronized parallelism in finite copying parallel rewriting systems. Journal of Computer and System Sciences 56, 27–45 (1998)
17. Seki, H., Kato, Y.: On the generative power of multiple context-free grammars and macro grammars. IEICE Transactions on Information and Systems E91–D, 209–221 (2008)
18. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88, 191–229 (1991)
19. Shallit, J.: A Second Course in Formal Langauges and Automata Theory. Cambridge University Press, Cambridge (2009)
20. Staudacher, P.: New frontiers beyond context-freeness: DI-grammars and DI-automata. In: 6th Conference of the European Chapter of the Association for Computational Linguistics (EACL '93), pp. 358–367 (1993)
21. Vijay-Shanker, K., Weir, D.J., Joshi, A.K.: Characterizing structural descriptions produced by various grammatical formalisms. In: 25th Annual Meeting of the Association for Computational Linguistics, pp. 104–111 (1987)

Post Correspondence Problem with Partially Commutative Alphabets

Barbara Klunder[2,*] and Wojciech Rytter[1,2,**]

[1] Institute of Informatics, University of Warsaw, ul. Banacha 2, 02-097 Warszawa, Poland
[2] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Toruń, Poland

Abstract. We introduce a version of Post Correspondence Problem (PCP, in short) generalized to words over partially commutative alphabets. Several observations are presented about the algorithmic status of the introduced problem. In particular solvability is shown for the partially commutative PCP for two special cases: the binary case of PCP (denoted by PCP(2)), and the case with one periodic morphism. This extends solvability results for the classical PCP for these cases. Also a weaker version of PCP, named here Weak-PCP, is discussed. This version distinguishes (in the sense of solvability) the case of noncommutative from the case of partially commutative alphabets. We consider also a solvable (though NP-hard) simple version of Weak-PCP. Our solvability results demonstrate the power of Ibarra's algorithms for reversal bounded multi-counter machines.

Keywords: Post Correspondence Problem, morphism, partially commutative alphabet, solvability, equality set, weak equality set, reversal bounded multicounter machine.

1 Introduction

The origins of partial commutativity is the theory of traces (i.e. monoids with partial commutations). In the fundamental Mazurkiewicz's paper [20], trace languages are regarded as a powerful means for description of behaviors of concurrent systems. The formal language theory over traces, limited to recognizable and rational trace languages, is the subject of [21].

Usually traces are more complicated than standard noncommutative words, for example rational expressions with classical meaning are less powerful then expressions for alphabets which are partially commutative. In the theory of traces the symbols represent some atomic processes and two symbols commute iff they are concurrent (the corresponding processes can be executed in any order).

* Supported by grant N N206 258035 of the Polish Ministry of Science and Higher Education.

** Supported by grant N206 004 32/0806 of the Polish Ministry of Science and Higher Education.

A partially commutative alphabet (p.c. alphabet, in short) is a finite set A of symbols with a relation $\mathcal{I} \subseteq A \times A$ which is symmetric and irreflexive.

Such a relation is named the *independency relation* or the relation of *partial commutativity*. The complement $\mathcal{D}$ of $\mathcal{I}$ is named the *dependency relation*.

For a given p.c. alphabet A and two words x, y we write $x \approx_I y$ iff the word x can be transformed to y commuting neighboring symbols which are in the relation $\mathcal{I}$. In other words x, y are equivalent modulo pairs of adjacent symbols which commute.

Example. Let $A = \{a, b, c\}$ and $\mathcal{I} = \{(a, b),\ (b, a)\}$, then

$$aaabbcab \approx_I bbaaacba.$$

There are many algorithmic problems easy in case of noncommutative alphabets which become unsolvable in the partially commutative case. Typical examples are equivalence of regular sets and the unique decipherability problem.

In this paper we consider yet another classical problem in the setting of partially commutative alphabets.

1.1 The Classical Post Correspondence Problem

The Post Correspondence Problem (PCP, in short) is probably the first published algorithmically undecidable combinatorial problem. It is a useful tool to establish undecidability results in different fields of theoretical computer science. As an example let us recall the problem of emptiness of intersection of two context-free languages.

The PCP problem with lists of size n (denoted by PCP(n)), is defined as follow.:

Given two lists of words over an alphabet A:

$$(u_1, u_2, \ldots, u_n),\ (v_1, v_2, \ldots, v_n)$$

decide if there exists a nonempty sequence $i_1, \ldots i_m$ of indices such that

$$u_{i_1} \ldots u_{i_m} = v_{i_1} \ldots v_{i_m}.$$

Equivalently, for an n-element alphabet X we are given two morphisms

$$h, g : X^\star \mapsto A^\star ,$$

and the problem is to decide whether the following set, called the *equality set*, is nonempty:

$$\text{EQ-SET}(h, g) = \{w \in X^+ : h(w) = g(w)\}.$$

1.2 Post Correspondence Problem with p.c. Alphabets

In the case of p.c. alphabet we define the equality set with respect to the relation $\mathcal{I}$ of partial commutation:

$$\text{EQ-SET}_I(h, g) = \{w \in X^+ : h(w) \approx_I g(w)\}$$

Now the partially commutative PCP problem is defined as follows:

given h, g and an independency relation $\mathcal{I}$,
check if EQ-SET$_I(h,g) = \emptyset$.

The only known (positive) result related to PCP with commutative alphabet is of [12] and deals with the Parikh equivalence, i.e. the case when the p.c. alphabet is fully commutative.

It is known that the classical PCP is solvable for the lists of size $n = 2$ and unsolvable for $n \geq 7$, the case of $2 < n < 7$ is not well understood.

We show that for partially commutative PCP the situation is similar.

1.3 Reversal Bounded Multicounter Machines

As an algorithmic tool (to show solvability) we use the algorithm testing emptiness of reversal bounded multicounter machines. In this subsection we define these machines and state the basic result needed later. A two-way k-counter machine M is defined by an 8-tuple $M =< k, K, \Sigma, \$, \bar{c}, \delta, q_0, F >$ where:

1. $K, \Sigma, \bar{c}, \$, q_0, F$ are the states, inputs, left and right endmarkers, initial and final states respectively;
2. δ is a mapping from $K \times (\Sigma \cup \{\bar{c}, \$\}) \times \{0,1\}^k$ into $K \times \{-1,0,1\} \times \{-1,0,1\}^k$.

Assume the counters cannot be decreased below zero.

A configuration $\bar{c}x\$$, $x \in \Sigma$, is defined by a tuple $(q, \bar{c}x\$, i, c_1, \ldots, c_k)$, denoting that M is in the state q with the head reading the i-th symbol of $\bar{c}x\$$ and $c_1, \ldots, c_k$ are integers stored in the k counters.

We define a relation $\rightarrow$ among configurations as follows:

$$(q, \bar{c}x\$, i, c_1, \ldots, c_k) \rightarrow (p, \bar{c}x\$, i + d, c_1 + d_1, \ldots, c_k + d_k)$$

if a is the ith symbol of $\bar{c}x\$$ and $\delta(q, a, \lambda(c_1), \ldots, \lambda(c_k))$ contains $(p, d, d_1 \ldots, d_k)$, where

$$\lambda(c_i) = \begin{cases} 0 & \text{if } c_i = 0 \\ 1 & \text{if } c_i \neq 0 \end{cases}$$

The reflexive and transitive closure of $\rightarrow$ is denoted as $\rightarrow^*$. A string x in Σ^* is accepted by M if

$$(q_0, \bar{c}x\$, 1, 0, \ldots, 0) \rightarrow^* (q, \bar{c}x\$, i, c_1, \ldots, c_k),$$

for some $q \in F$, $1 \leq i \leq |\bar{c}x\$|$ and nonnegative integers $c_1, \ldots, c_k$. The set of strings accepted by M is denoted by $L(M)$.

A *reversal-bounded* k-counter machine operates in such a way that in every accepting computation the input head reverses direction at most p times and the count in each counter alternately increases and decreases at most q times, where p, q are some constants.

The emptiness problem for M is to check if $L(M) = \emptyset$. We will use one of the results of Ibarra's paper.

Lemma 1. [13]
The emptiness problem for reversal-bounded multicounter machines is solvable.

2 Two Special Cases of Partially Commutative PCP

For a pair of symbols (a, b) we denote by $\pi_{a,b}$ the projection which for a word w removes all letters but a, b.

Example. $\pi_{a,b}(accbacb) = abab, \ \ \pi_{a,c}(accbacb) = accac.$

The following result reduces the relation $\approx$ to multiple application of equality of classical strings over noncommutative alphabet.

Lemma 2. [8] $u \approx_I w \Leftrightarrow (\forall \ (a, b) \notin \mathcal{I} \ ) \ \pi_{a,b}(u) = \pi_{a,b}(w)$.

According to Lemma 2 we can express the equality set for PCP with p.c. alphabets as a finite intersection of equality sets for standard (noncommutative) alphabets.

Lemma 3. $\mathit{EQ\text{-}SET}_I(h, g) = \bigcap_{(a,b)\notin\mathcal{I}} \mathit{EQ\text{-}SET}(\pi_{a,b} \cdot h, \pi_{a,b} \cdot g)$.

2.1 Partially Commutative PCP(2)

In this section we assume that $n = 2$ and $X = \{0, 1\}$. The problem PCP(2) is solvable as was proved by Ehrenfeucht, Karhumäki and Rozenberg in 1982, see [3]. On the other hand Matiyasevich and Sénizergues showed that PCP(7) is unsolvable, [19]. Before we state the first new result we shall recall the main results of [4] and [9], which can be found in [8], too, concerning the structure of equality sets in the case of free monoids.

We say that a morphism $h : X^\star \mapsto A^\star$ is periodic if $h(X) \subseteq u^\star$ for some word u. For a symbol s by $|w|_s$ denotes the number of occurrences of s in w.

Lemma 4. ([4])

(a) *If h and g are periodic then either*
$\mathit{EQ\text{-}SET}(h, g) = \emptyset$ *or* $\mathit{EQ\text{-}SET}(h, g) = \{w \in X^\star \ : \ r(w) = \frac{|w|_0}{|w|_1} = k\}$
for some $k \geq 0$ or $k = \infty$.

(b) *If h is periodic and g is not then $\mathit{EQ\text{-}SET}(h, g)$ is empty or equal to u^+ for some nonempty word u.*

Hence equality sets are regular or accepted by a reversal bounded one-counter machines. The number $r(w) = \frac{|w|_0}{|w|_1}$ is called the *ratio* of a word and it is decidable if the intersection of regular sets and (or) sets of words of a given ratio is nonempty.

In the case of two non-periodic morphisms the equality set is always regular. For two periodic morphisms, EQ-SET(h, g) can be nonempty only when $h(X), g(X) \subseteq u^\star$ and then r can be easily found. The main result of [10] says that in the nonperiodic case the equality set is of a very simple form.

Lemma 5. [10]
Let (h, g) be a pair of non-periodic morphisms over a binary alphabet. If the equality set $\mathit{EQ\text{-}SET}(h, g)$ is nonempty then it is of the form $(u + v)^+$ for some words u, v.

The following constructive result has been shown in [9] (as Corollary 5.7).

Lemma 6. [9]
Assume the size of the lists is $n = 2$ and h, g are two nonperiodic morphisms. Then EQ-SET(h, g) can be effectively found (as a regular expression or finite automaton).

A combination of these results implies easily the following fact.

Theorem 1
Partially commutative PCP(2) is solvable

Proof
Lemmas 2,3,4,5 imply that the p.c. PCP(2) can be reduced to the emptiness of intersection of a finite set of languages, each of them is regular or a reversal bounded one-counter language. These languages can be effectively presented by corresponding finite automata or reversal bounded one-counter automata. Hence the intersection language can be accepted by a reversal-bounded multicounter machine M which is a composition of all these automata. Due to Lemma 1 the emptiness problem for M is solvable. Consequently the emptiness of the intersection of related languages is also solvable. □

2.2 Partially Commutative PCP with One Periodic Morphism

We shall consider here another easily solvable case: periodic morphisms. We say that a morphism h into a p.c. alphabet is periodic if there is a word w such that for each x, $h(x) \approx_I w^i$ for some natural i. The proof of the next theorem adopts similar arguments as the classical one, see [8].

Theorem 2
Partially commutative PCP is decidable for instances (h, g), where h is periodic.

Proof
Let $h, g : X^\star \mapsto A^\star$ and assume that h is periodic Let $(a, b) \in \mathcal{D}$, then the equality set of $(\pi_{a,b} \cdot h, \pi_{a,b} \cdot g)$ is a multicounter reversal bounded language.

Now the equality set of (h, g) is the intersection of multicounter reversal bounded languages too. Define the morphism ρ by $\rho(a) = |h(a)| - |g(a)|$ for all $a \in X$. Define also the set $R = g^{-1}(u^\star \setminus \{\varepsilon\}\}$. We have:

1. $\rho^{-1}(0) = \{v : |h(v)| = |g(v)|\}$
2. $w \in \rho^{-1}(0) \cap R \Leftrightarrow w \neq \varepsilon, g(w) \in u^\star$ and $|g(w)| = |h(w)|$.

Hence $g(w) = h(w)$ for some $w \neq \varepsilon$ if and only if $\rho^{-1}(0) \cap R \neq \emptyset$. The language $\rho^{-1}(0) \cap R$ is recognizable by a reversal-bounded multicounter machine. Hence emptiness is solvable due to [13]. □

3 Partially Commutative Weak PCP

There is a version of PCP which is easily solvable for noncommutative alphabets but surprisingly the same version is unsolvable for partially commutative alphabets.

Define the partially commutative problem, named here *Weak* PCP, with parameters r, s as follows:

given p.c. words $x_1, x_2, \ldots, x_r,\ y_1, y_2, \ldots, y_s$,
test if there are nonempty sequences

$$(i_1, i_2, \ldots, i_p),\ (j_1, j_2, \ldots j_q)$$

such that

$$x_{i_1} x_{i_2} \ldots x_{i_p} \quad \approx_I \quad y_{j_1} y_{j_2} \ldots y_{j_q}.$$

We can redefine it using the concept of equality sets as follows:

given two morphisms h, g into p.c. words, test emptiness of the set

$$\text{Weak-EQ-SET}(h, g) = \{(z1, z2)\ :\ h(z1)\ \approx_I\ g(z2)\}.$$

The set Weak-EQ-SET(h, g) is called here the weak equality set. If we do not write *partially commutative* this means that we consider the case of classical noncommutative alphabet (special case of partially commutative).

Observation. The language $\{x\#y^R\ :\ (x, y) \notin \text{Weak-EQ-SET}(h, g)\ \}$ is a linear context free language.

Proof
The one-turn pushdown automaton can nondeterministically guess two symbols $a \neq b,\ (a, b) \notin I$. Then it reads x and writes on the stack the word $\pi_{a,b}(x)$, then it checks, while reading y^R that $\pi_{a,b}(x) \neq \pi_{a,b}(y)$. □

Denote by Weak-PCP(s, r) the weak PCP in which the domain of h is of size s and the domain of g is of size r.

The natural relation of PCP for noncommutative alphabets to Weak-PCP for p.c. alphabets is as follows.

Assume we have an instance of PCP given by (u_i, v_i) for $i = 1 \ldots k$, where $u_i, v_i \in A^+$ and $A \cap \{1, \ldots, k\}\ =\ \emptyset$.

Let the p.c. alphabet be $A \cup \{1, \ldots, k\}$, where all letters in A commute with all letters in $\{1, \ldots, k\}$, and no other pairs of different letters commute.

Define

$$h(i) = i \cdot u_i,\ g(i) = i \cdot v_i \ \text{ for each } 1 \leq i \leq k$$

Then we can express in a natural way the PCP(k) problem as a Weak-PCP(k, k) with morphisms h, g defined above.

It is known, [19] that PCP(7) is unsolvable, hence we have proved that partially commutative . Weak-PCP$(7, 7)$ is unsolvable. We improve this slightly below.

We know that PCP(2) is decidable (also for p.c. alphabets), this would suggest that Weak-PCP$(2,k)$ is solvable. However this suggestion is wrong.

Theorem 3
(a) *Weak-PCP(s,r) is solvable for any s, r and noncommutative alphabets.*
(b) *Partially commutative Weak-PCP$(2,7)$ is unsolvable.*

Proof
(a) It easy to see that the problem Weak-PCP(s,r) for totally noncommutative alphabets is reducible to the emptiness of a language of a form:

$$(x_1 \cup \ldots \cup x_s)^+ \ \cap \ (y_1 \cup \ldots \cup y_r)^+.$$

This is a simple instance of emptiness problem for a classical regular language, hence it is obviously solvable.

(b) Let us consider an instance of PCP(7) for lists $(u_1, u_2, \ldots u_7)$ and $(v_1, v_2, \ldots v_7)$. Assume the alphabet of words u_i, v_i is $\Sigma = \{a, b\}$. Let $\overline{\Sigma}$ be the disjoint copy of Σ, by $\overline{v}$ we mean the word v with each letter $v[i]$ changed to its copy $\overline{v[i]}$. The instance of PCP(7) is reduced to the problem Weak-PCP$(2,7)$ as follows:

$$h(1) = a\,\overline{a},\ h(2) = b\,\overline{b}$$

$$g(i) = u_i\,\overline{v_i} \ \text{ for each } 1 \leq i \leq 7$$

Assume that the commutation relation is $\Sigma \times \overline{\Sigma} \cup \overline{\Sigma} \times \Sigma$. Then $PCP(k)$ has a solution iff Weak-PCP$(2,7)$ has a solution for the morphisms h, g constructed above. Hence unsolvable problem PCP(7) is reduced to p.c Weak PCP(2,7). Consequently the partially commutative Weak PCP(2,7) is unsolvable. □

4 Weak-PCP$(1,k)$

In this section we consider a solvable case of Weak PCP, the situation when one of the lists is of size 1. Especially simple is the case $k = 1$, i.e. the partially commutative Weak-PCP$(1,1)$. The case of totally noncommutative alphabet is simple: for two words u, v we have

$$(\exists\ i,j)\ u^i = v^j \ \Leftrightarrow\ (\ uv = vu\ ).$$

Using projections $\pi_{a,b}$ we can reduce the p.c. case to the noncommutative case:

Observation 1
Partially commutative Weak-PCP$(1,1)$ for the words u, v is reducible to the test of $uv \approx_I vu$, in other words:
$(\ (\exists\ (natural)\ i,j > 0)\ u^i \approx_I v^j\ )\ \Leftrightarrow (\ uv \approx_I vu\ ).$

Corollary 1. *Partially commutative Weak-PCP$(1,1)$ is solvable in deterministic polynomial time.*

Theorem 4. *Weak-PCP$(1,k)$ is solvable.*

Proof
Assume we have an instance of Weak-PCP$(1,k)$, given by the words $x_1, x_2, \ldots x_k$ and the word w.

In this problem we ask if there is a word $x \in \{1,\ldots,k\}^+$ and a natural m such that $h(x) \approx_I w^m$.

We can construct a reversal-bounded multicounter machine M which accepts all such strings x. Assume we have r pairs of the letters a, b which do not commute. The machine M has r counters, intially it is guessing the number m and is storing it in each counter.

Assume the i-th pair is (a_i, b_i), the machine M reads the input x on-line from left to right and using the i-th counter checks if $\pi_{a_i,b_i}(h(x)) = \pi_{a_i,b_i}(w^m)$.

Then the problem Weak-PCP$(1,k)$ is reducible to emptiness of reversal-bounded multicounter machine, which is solvable due to [13]. □

Theorem 5. *Assume k is a part of the input, then Weak-PCP$(1,k)$ is NP-hard.*

Proof. The following problem called Exact Cover by 3-sets is NP-complete: given family of sets $X_i \subset U = \{1,2,...,n\}$, where $1 \le i \le r$, each of cardinality 3, check if U is a disjoint union of a subfamily of these sets.

For a subset X_i let x_i be the string which is a list of elements of X_i. We can take the alphabet U, totally commutative, then the problem above is reduced to the problem if the string $z = 1\ 2\ 3\ ...n$ is equivalent (modulo permutation) to a concatenation of some of strings x_i.

W construct the instance of PCP(1,r+1) with lists:

$$w = z \cdot \#, \quad (x_1,\ x_2,\ \ldots x_r,\ x_{r+1} = \#),$$

where # is an additional symbol noncommuting with any other symbol.

Then Exact Cover by 3-sets is reduced to the problem if some concatenation of strings from the family $x_1, x_2, \ldots x_{r+1}$ is equivalent (modulo our partial commutation) to w^m, for some natural m. In this way we have a deterministic polynomial time reduction of Exact Cover by 3-sets to partially commutative PCP(1,r+1). Therefore the last problem is NP-hard. □

We do not know if partially commutative $Weak\ PCP(1,k)$ is in NP, however we prove that it is in P for the lists of words over an alphabet of a constant size.

Define:

$$\Delta(\Sigma) = \{w \in \Sigma^+ : (\forall\ s1, s2 \in \Sigma)\ |w|_{s1} = |w|_{s2}\}.$$

In other words $\Delta(\Sigma)$ is the set of words over the alphabet Σ in which the number of occurrences of letters are the same. Let $L(M)$ be the language accepted by a nodeterministic finite automaton M.

We consider the following problem for M.

(diagonal emptiness problem) $L(M) \cap \Delta(\Sigma) = \emptyset$?

The following lemma can be shown using techniques from [5, 17]. One of these techniques is an interesting application of Euler theorem about Euler tours in multi-graphs. This allows to describe the membership problem as an integer linear program, where multiplicities are treated as variables, and the Euler condition related to indegree-outdegree of nodes can be expressed as a set of equations. This gives singly exponential upper bounds for the size of the solution.

Lemma 7
The diagonal emptiness *problem for finite automata is in NP;*
If z is a shortest word in $L(M) \cap \Delta(\Sigma)$ then it is of singly exponential length (if there is any such z).

We use the following fact, shown recently by Eryk Kopczynski.

Lemma 8. [15]
Assume the alphabet is of a constant size. Then the membership problem for a commutative word, given as a Parikh vector with coefficients written in binary, in a given regular language is in P.

Lemma 9
For a nondeterministic automaton M with input alphabet Σ of a constant size the diagonal emptiness problem is in P (solved by a deterministic polynomial time algorithm).

Proof. We can transform M to an equivalent nondeterministic automaton of polynomial size such that its accepting states are sinks (there are no outgoing edges). Assume now that M is of this form.

For an integer j let

$$\Sigma = \{a_1, a_2, \dots a_r\}, \; \Sigma^{(j)} = a_1^j a_2^j \dots a_r^j$$

Change M to the automaton M', by adding for each accepting state q the loop (transition) from q to q labeled by $\Sigma^{(1)}$.

Due to Lemma 7 there is a constant c such that the length of the shortest word in $L(M) \cap \Delta(\Sigma)$ is upper bounded by $K = 2^{cn}$.

Now the diagonal emptiness problem is reduced to the problem whether $\Sigma^{(K)}$ is commutatively equivalent to some word accepted by M'. This problem is in P due to [15].

Hence our problem is also in P. □

Remark
The above problem is NP-complete for nonconstant alphabet, as pointed in [18]. However we do not use this result here.

We can use Lemma 9 to show the following fact.

Theorem 6
Let A be the alphabet of words in the lists defining a partially commutative Weak PCP(1,k). If $|A| = O(1)$ and $k = O(1)$ then partially commutative Weak PCP(1,k) is in P.

Sketch of the proof
Let r be the number of pairs of symbols a, b which do not commute. Let (a_j, b_j) be the j-th such pair and denote:

$$w_{(j)} = \pi_{a_j,b_j}(w)$$

We are to check if there is a sequence of indexes $i_1, i_2, \ldots i_m$ such that:

$$\exists\ (N \geq 1)\ \forall\ (1 \leq j \leq r)\ \ \pi_{a_j,b_j}(x_{i_1} x_{i_2} \ldots x_{i_m}) = w^N_{(j)}.$$

We construct an automaton similar to the construction of a graph for testing unique decipherability of a set of words.

In our graph (the automaton M) each node is a tuple of r words. The j-th component is a prefix α (possibly empty) of $w_{(j)}$.

Let $A' = \{a_1, a_2, \ldots a_r\}$ be some additional symbols (acting as counters).

Then there is an edge labeled a_j^k from α to β iff for some x_i we have:

$$\alpha \cdot \pi_{a_j,b_j}(x_i) = w^k_{(j))} \cdot \beta,$$

where β is a prefix of $w_{(j)}$. For each component α in a given tuple we add such string a_j^k to the transition, and this is done for each component. Hence each edge of the graph of the automaton M is labeled by a string over the alphabet A' of counters.

In this way the automaton A is following some nondeterministically guessed x_i's and keeps on the edges the count of the number of copies of $w_{(j)}$. Hence it is enough to check additionally if for any two a_j, a_s we have the same number of occurrences of these symbols on some (the same for all components) nondeterministically guessed path from a source (empty prefix) to a sink (also empty prefix).

The path corresponds to the choice of a sequence $x_{i_1} x_{i_2} \ldots x_{i_m}$ and some natural nonzero N such that

$$x_{i_1} x_{i_2} \ldots x_{i_m} = w^N$$

Hence our problem is reduced to the diagonal emptiness problem for M. The machine M is of a polynomial size since we have only a constant number of noncommuting pairs (a_j, b_j). We omit details. □

Open Problems

1. Is partially commutative Weak-PCP$(2, 2)$ solvable ?
2. What about the complexity status of partially commutative Weak-PCP$(1, k)$, we showed it is NP-hard when k is a part of the input, and is in P for constant sized alphabet when k is fixed . What about general alphabets, is it in NP ? Is it NP-hard in case of a fixed k ?

3. For which partially commutative alphabets I the problem Weak-PCP is solvable ? We suspect that it is solvable in case of transitively closed dependency relations $\mathcal{D}$ (the complement of $\mathcal{I}$).

4. What is the minimal k such that partially commutative PCP(k) is unsolvable (in case of noncommutative alphabet the smallest known k is $k = 7$).

Acknowledgment

The authors thank J. Leroux, S. Lasota and Eryk Kopczyński for helpful comments related to Lemma 9.

References

1. Clerbout, M., Latteux, M.: Semi-commutations. Information & Computation 73(1), 59–74 (1987)
2. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific Publishing Co., Inc., Singapore (1995)
3. Ehrenfeucht, A., Karhumäki, J., Rozenberg, G.: The (generalized) Post correspondence problem with lists consisting of two words is decidable. Theor. Comput. Sci. 21, 119–144 (1982)
4. Ehrenfeucht, A., Karhumäki, J., Rozenberg, G.: On binary equality sets and a solution to the set conjecture in the binary case. Journal of Algebra 85, 76–85 (1983)
5. Esparza, J.: Petri nets, commutative context-free grammars, and basic parallel processes. Fundamenta Informaticae 31, 13–26 (1997)
6. Gibbons, A., Rytter, W.: On the decidability of some problems about rational subsets of free partially commutative monoids. Theor. Comput. Sci. 48(2-3), 329–337 (1986)
7. Halava, V., Harju, T., Hirvensalo, M.: Binary (generalized) Post correspondence problem. Theor. Comput. Sci. 276(1-2), 183–204 (2002)
8. Harju, T., Karhumäki, J.: Morphisms. In: Handbook of formal languages, vol. 1 (1997)
9. Harju, T., Karhumäki, J., Krob, D.: Remarks on generelized Post correspondence problem. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046, pp. 39–48. Springer, Heidelberg (1996)
10. Holub, S.: Binary equality sets are generated by two words. Int. J. Algebra (259), 1–42 (2003)
11. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (2001)
12. Ibarra, O.H., Kim, C.E.: A useful device for showing the solvability of some decision problems. In: STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing, pp. 135–140 (1976)
13. Ibarra, O.: Reversal-bounded multicounter machines and their decision problems. J. ACM 25(1), 116–133 (1978)
14. Kopczyǹski, E.: Personal Communication (2009)
15. Kopczyński, E.: Complexity of problems of commutative grammars, draft (2009)
16. Lasota, S.: Personal Communication (2009)

17. Leroux, J.: A polynomial time Presburger criterion and synthesis for number decision diagrams. In: LICS, pp. 147–156 (2005)
18. Leroux, J.: Personal Communication (2009)
19. Matiyasevich, Y., Sénizergues, G.: Decision problems for semi-Thue systems with a few rules, pp. 523–531 (1996)
20. Mazurkiewicz, A.: Concurrent program schemes and their interpretations. Technical report, Aarhus University (1977)
21. Ochmański, E.: Recognizable Trace Languages, pp. 167–204. World Scientific Publishing Co., Inc., Singapore (1995)

Reversible Pushdown Automata

Martin Kutrib and Andreas Malcher

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
{kutrib,malcher}@informatik.uni-giessen.de

Abstract. Reversible pushdown automata are deterministic pushdown automata having the property that any configuration occurring in any computation has exactly one predecessor. In this paper, the computational capacity of reversible computations in pushdown automata is investigated and turns out to lie properly in between the regular and deterministic context-free languages. Furthermore, it can be shown that a deterministic context-free language cannot be accepted reversibly if more than realtime is necessary for acceptance. Closure properties as well as decidability questions for reversible pushdown automata are studied. Finally, the question of whether a given (nondeterministic) pushdown automaton can be algorithmically tested for reversibility is answered in the affirmative, whereas it is shown to be undecidable whether the language accepted by a given nondeterministic pushdown automaton is reversible.

1 Introduction

Computers are information processing devices which are physical realizations of abstract computational models. It may be difficult to define exactly what information is or how information should be measured suitably. It may be even more difficult to analyze in detail how a computational device processes or transmits information while working on some input. Thus, one first step towards a better understanding of information is to study computations in which no information is lost. Another motivation to study information preserving computations is the physical observation that a loss of information results in heat dissipation. A first study of this kind has been done in [2] for Turing machines where the notion of *reversible* Turing machines is introduced. Deterministic Turing machines are called reversible when they are also backward deterministic, i.e., each configuration has exactly one predecessor. One fundamental result shown in [2] is that every, possibly irreversible, Turing machine can always be simulated by a reversible Turing machine in a constructive way. This construction is significantly improved in [7] with respect to the number of tapes and tape symbols. Thus, for the powerful model of Turing machines, which describe the recursively enumerable languages, every computation can be made information preserving. At the other end of the Chomsky hierarchy there are the regular languages. Reversible variants of deterministic finite automata have been defined and investigated in [1,8]. It turns out that there are regular languages for which no reversible deterministic finite automaton exists. Thus, there are computations in which a loss of information is inevitable. Another result of [8] is that the existence of a reversible automaton can be decided for a regular language in polynomial time.

Reversible variants of the massively parallel model of cellular automata and iterative arrays, which are interacting deterministic finite automata, have been studied in [5,6]. One main result there is the identification of data structures and constructions in terms of closure properties which can be implemented reversibly. Another interesting result is that, in contrast to regular languages, there is no algorithm which decides whether or not a given cellular device is reversible.

In this paper, the investigation of reversibility in computational devices is complemented by the study of *reversible pushdown automata*. These are deterministic pushdown automata with the property that any configuration occurring in any computation has exactly one predecessor. First, it is shown that all regular languages as well as some non-regular languages are accepted by reversible deterministic pushdown automata. On the other hand, we prove that there is a deterministic context-free language which cannot be accepted in a reversible way. Thus, the computational capacity of reversible pushdown automata lies properly in between the regular and deterministic context-free languages. Moreover, every deterministic context-free language which needs more than realtime is shown not to be acceptable by reversible pushdown automata. In the second part of the paper, closure properties and decidability questions of the language class are investigated. It turns out that the closure properties of reversible pushdown automata are similar to those of deterministic pushdown automata. The main difference is the somehow interesting result that the language class accepted by reversible pushdown automata is *not* closed under union and intersection with regular languages. Finally, the questions of whether a given automaton is a reversible pushdown automaton, and whether its language is reversible are investigated. We show that it is decidable whether a given nondeterministic or deterministic pushdown automaton is reversible, whereas it is undecidable whether a nondeterministic pushdown automaton accepts a reversible language.

2 Preliminaries and Definitions

Let Σ^* denote the set of all words over the finite alphabet Σ. The empty word is denoted by λ, and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. The set of words of length at most $n \geq 0$ is denoted by $\Sigma^{\leq n}$. For convenience, we use Σ_λ for $\Sigma \cup \{\lambda\}$. The reversal of a word w is denoted by w^R and for the length of w we write $|w|$. The number of occurrences of a symbol $a \in \Sigma$ in $w \in \Sigma^*$ is written as $|w|_a$. Set inclusion is denoted by $\subseteq$, and strict set inclusion by $\subset$.

A *deterministic pushdown automaton* is a system $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F\rangle$, where Q is a finite set of states, Σ is the finite input alphabet, Γ is a finite pushdown alphabet, $q_0 \in Q$ is the initial state, $\perp \in \Gamma$ is a distinguished pushdown symbol, called the bottom-of-pushdown symbol, which initially appears on the pushdown store, $F \subseteq Q$ is the set of accepting states, and δ is a mapping from $Q \times \Sigma_\lambda \times \Gamma$ to $Q \times \Gamma^*$ called the transition function. In particular, there must never be a choice of using an input symbol or of using λ input. So, it is required that for all q in Q and Z in Γ: if $\delta(q, \lambda, Z)$ is defined, then $\delta(q, a, Z)$ is undefined for all a in Σ.

A *configuration* of a pushdown automaton is a quadruple (v, q, w, γ), where q is the current state, v is the already read and w the unread part of the input, and γ the current content of the pushdown store, the leftmost symbol of γ being the top symbol. On input w the initial configuration is defined to be $(\lambda, q_0, w, \bot)$. For $q \in Q$, $a \in \Sigma_\lambda$, $v, w \in \Sigma^*$, $\gamma \in \Gamma^*$, and $Z \in \Gamma$, let $(v, q, aw, Z\gamma)$ be a configuration. Then its *successor configuration* is $(va, p, w, \beta\gamma)$, where $\delta(q, a, Z) = (p, \beta)$. We write $(v, q, aw, Z\gamma) \vdash (va, p, w, \beta\gamma)$ in this case. As usual, the reflexive transitive closure of $\vdash$ is denoted by $\vdash^*$. In order to simplify matters, we require that in any configuration the bottom-of-pushdown symbol appears exactly once at the bottom of the pushdown store, that is, it can neither appear at some other position in the pushdown store nor be deleted. Formally, we require that if $\delta(q, a, Z) = (p, \beta)$ then either $Z \neq \bot$ and β does not contain $\bot$, or $Z = \bot$ and $\beta = \beta'\bot$, where β' does not contain $\bot$. The *language accepted* by $\mathcal{M}$ with accepting states is $L(\mathcal{M}) = \{\, w \in \Sigma^* \mid (\lambda, q_0, w, \bot) \vdash^* (w, q, \lambda, \gamma),$ for some $q \in F$ and $\gamma \in \Gamma^* \,\}$. In general, the family of all languages that are accepted by some device X is denoted by $\mathscr{L}(X)$.

Now we turn to reversible pushdown automata. Basically, reversibility is meant with respect to the possibility of stepping the computation back and forth. In particular, for reverse computation steps the head of the input tape is always moved to the *left*. Therefore, the automaton rereads the input symbol which has been read in a preceding forward computation step. So, for reversible pushdown automata there must exist a reverse transition function.

A *reverse transition function* $\delta_R : Q \times \Sigma_\lambda \times \Gamma \to Q \times \Gamma^*$ maps a configuration to its *predecessor configuration*. For $q \in Q$, $a \in \Sigma_\lambda$, $v, w \in \Sigma^*$, $\gamma \in \Gamma^*$, and $Z \in \Gamma$, let $(va, q, w, Z\gamma)$ be a configuration. Then its *predecessor configuration* is $(v, p, aw, \beta\gamma)$, where $\delta_R(q, a, Z) = (p, \beta)$. We write $(va, q, w, Z\gamma) \dashv (v, p, aw, \beta\gamma)$ in this case. Let $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ be any sequence of configurations passed through by $\mathcal{M}$ beginning with an initial configuration c_0. Then $\mathcal{M}$ is said to be *reversible* (REV-PDA), if there exists a reverse transition function δ_R such that $c_{i+1} \dashv c_i$, for $0 \leq i \leq n-1$ (cf. Figure 1).

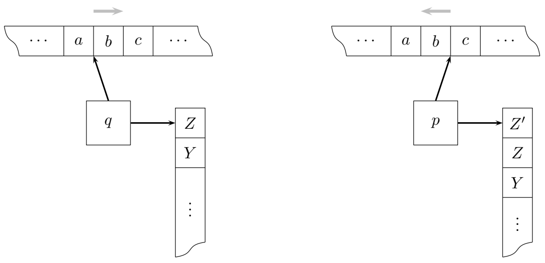


Fig. 1. Successive configuration of a reversible deterministic pushdown automaton, where $\delta(q, b, Z) = (p, Z'Z)$ (left to right) and $\delta_R(p, b, Z') = (q, \lambda)$ (right to left)

In order to clarify our notion we continue with an example.

Example 1. The linear language $\{ wcw^R \mid w \in \{a,b\}^* \}$ is accepted by the reversible DPDA $\mathcal{M} = \langle \{q_0,q_1,q_2\}, \{a,b,c\}, \{a,b,\perp\}, \delta, q_0, \perp, \{q_2\} \rangle$, where the transition functions δ and δ_R are as follows.

Transition function δ	Reverse transition function δ_R
(1) $\delta(q_0,a,\perp) = (q_0,a\perp)$	(1) $\delta_R(q_0,a,a) = (q_0,\lambda)$
(2) $\delta(q_0,b,\perp) = (q_0,b\perp)$	(2) $\delta_R(q_0,b,b) = (q_0,\lambda)$
(3) $\delta(q_0,a,a) = (q_0,aa)$	(3) $\delta_R(q_1,c,\perp) = (q_0,\perp)$
(4) $\delta(q_0,a,b) = (q_0,ab)$	(4) $\delta_R(q_1,c,a) = (q_0,a)$
(5) $\delta(q_0,b,a) = (q_0,ba)$	(5) $\delta_R(q_1,c,b) = (q_0,b)$
(6) $\delta(q_0,b,b) = (q_0,bb)$	(6) $\delta_R(q_1,a,a) = (q_1,aa)$
(7) $\delta(q_0,c,\perp) = (q_1,\perp)$	(7) $\delta_R(q_1,a,b) = (q_1,ab)$
(8) $\delta(q_0,c,a) = (q_1,a)$	(8) $\delta_R(q_1,b,a) = (q_1,ba)$
(9) $\delta(q_0,c,b) = (q_1,b)$	(9) $\delta_R(q_1,b,b) = (q_1,bb)$
(10) $\delta(q_1,a,a) = (q_1,\lambda)$	(10) $\delta_R(q_1,a,\perp) = (q_1,a\perp)$
(11) $\delta(q_1,b,b) = (q_1,\lambda)$	(11) $\delta_R(q_1,b,\perp) = (q_1,b\perp)$
(12) $\delta(q_1,\lambda,\perp) = (q_2,\perp)$	(12) $\delta_R(q_2,\lambda,\perp) = (q_1,\perp)$

As expected, the transitions (1) through (6) of δ are used by $\mathcal{M}$ to store the input prefix w. When a c appears in the input, transitions (7) through (9) are used to change to state q_1 while the pushdown store remains unchanged. By the transitions (10) and (11) the input suffix w^R is matched with the stored prefix w. Finally, if the bottom-of-pushdown symbol is seen in state q_1, automaton $\mathcal{M}$ changes into the sole accepting state q_2 and the computation necessarily stops.

For the backward computation the transitions of δ_R are used. Since there is only one transition of δ that changes to state q_2, transition (12) reverses this step. For input symbols a and b, the only transitions of δ that change to state q_1 are (8) and (9) which pop the symbol from the top of the pushdown store if it matches the current input symbol. So, transitions (6) through (11) of δ_R are easily constructed in order to reverse the popping by pushing the current input symbol. In forward computations $\mathcal{M}$ changes from state q_0 to q_1 if and only if the current input symbol is a c, whereby the pushdown store remains unchanged. These steps can uniquely be reversed by the transitions (3) through (5) of δ_R. While in state q_0, in any forward step a current input symbol a or b is pushed. Therefore, δ_R reverses the pushing by popping whenever the pushdown store is not empty and an a or b appears in the input by transitions (1) and (2). This concludes the construction of δ_R. □

Example 2. Only slight modifications of the construction given in Example 1 show that the languages $\{ a^n cb^n \mid n \geq 0 \}$, and $\{ a^n cb^n \mid n \geq 0 \}^*$, as well as $\{ a^m cb^n ea^m \mid m,n \geq 0 \} \cup \{ a^n db^n ea^m \mid m,n \geq 0 \}$ are accepted by REV-PDAs as well. □

3 Structural Properties and Computational Capacity

In this section the computational capacity of REV-PDAs is considered. First, we examine the structure of transitions that enable reversibility, and investigate the role played by λ-steps.

Fact 1. *Let $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, \bot, F\rangle$ be a REV-PDA.*

1. *As for the transition function also for the reverse transition function δ_R we have necessarily that for all q in Q and Z in Γ: if $\delta_R(q, \lambda, Z)$ is defined, then $\delta_R(q, a, Z)$ is undefined for all a in Σ. Otherwise the predecessor configuration would not be unique and, thus, $\mathcal{M}$ not be reversible.*
2. *All transitions of $\mathcal{M}$ are either of the form $\delta(q, a, Z) = (p, \lambda)$, or $\delta(q, a, Z) = (p, Y)$, or $\delta(q, a, Z) = (p, YZ)$, where $q, p \in Q$, $a \in \Sigma_\lambda$, $Y, Z \in \Gamma$. In particular, there is no transition that modifies the pushdown store except for the topmost symbol, since the reverse transition has only access to the topmost symbol.*

Now we turn to λ-steps. It is well known that general deterministic pushdown automata that are not allowed to perform λ-steps are weaker than DPDAs that may move on λ input [4]. In order to go a little more into details we consider the maximal number of consecutive λ-steps. A REV-PDA is said to be *quasi realtime* if there is a constant that bounds this number for all computations. The REV-PDA is said to be *realtime* if this constant is 0, that is, if there are no λ-steps at all.

Lemma 1. *Every REV-PDA $\mathcal{M}$ with $L(\mathcal{M}) \neq \emptyset$ and $L(\mathcal{M}) \neq \{\lambda\}$ is quasi realtime.*

Proof. Let $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, \bot, F\rangle$ be a REV-PDA accepting at least one non-empty input, and assume in contrast to the assertion that $\mathcal{M}$ is not quasi realtime. Then there is a computation on some input w such that at least $|Q| \cdot |\Gamma|$ consecutive λ-steps are performed. If these steps appear before any non-λ-step, $\mathcal{M}$ starts each computation with an infinite loop on λ input. So, depending on whether or not this loop includes an accepting state $L(\mathcal{M})$ is either $\{\lambda\}$ or $\emptyset$, a contradiction.

Next assume that at least $|Q| \cdot |\Gamma|$ consecutive λ-steps appear after some non-λ-step, and let $r : \Sigma^* \times Q \times \Sigma^* \times \Gamma^+ \to Q \times \Gamma$ be a mapping that maps a configuration to its state and the topmost pushdown symbol. Then there is a (partial) computation $c_{k-1} \vdash c_k \vdash^* c_{k+i} \vdash^* c_{k+i+j-1} \vdash c_{k+i+j}$, where the transition from c_{k-1} to c_k reads some non-λ input $a \in \Sigma$ and all the other transitions are on λ input. Moreover, we have $r(c_{k+i}) = r(c_{k+i+j})$, for some minimal $0 \leq i$, $1 \leq j$ such that $i + j \leq |Q| \cdot |\Gamma|$. Let $r(c_{k+i}) = (p, Z)$. Then, for $i = 0$, $\delta_R(p, \lambda, Z)$ has to be defined in order to get back from configuration c_{k+j} to configuration c_{k+j-1}. At the same time $\delta_R(p, a, Z)$ has to be defined in order to get back from configuration c_k to configuration c_{k-1}, a contradiction. For $i \geq 1$ we know that $r(c_{k+i-1})$ and $r(c_{k+i+j-1})$ are different since i has been chosen to be minimal. Since for this case $\delta_R(p, \lambda, Z)$ has to be defined in such a way that the computation steps back from configuration c_{k+i} to configuration c_{k+i-1}, and at the same time such that the computation steps back from c_{k+i+j} to $c_{k+i+j-1}$, we obtain a contradiction, too. □

In order to conclude the consideration of λ-steps we present the result that, in fact, the family $\mathscr{L}(\text{REV-PDA})$ is a subfamily of the realtime deterministic context-free languages.

Theorem 2. *For every REV-PDA there is an equivalent realtime REV-PDA.*

Theorem 2 provides us with a large class of deterministic context-free languages that are not reversible. Every deterministic context-free language that is not realtime is not accepted by any REV-PDA. For example, the language $\{ a^m e b^n c a^m \mid m, n \geq 0 \} \cup \{ a^m e b^n d a^n \mid m, n \geq 0 \}$ does not belong to the family $\mathscr{L}(\text{REV-PDA})$ (see, for example, [4,3]). This result immediately raises the question of whether or not *all* realtime deterministic context-free languages are also reversible. The next lemma answers this question negatively.

Lemma 2. *The deterministic linear realtime language* $L = \{ a^n b^n \mid n \geq 0 \}$ *is not accepted by any REV-PDA.*

Proof. Assume in contrast to the assertion that L is accepted by some REV-PDA $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$. Without loss of generality, we may assume that $\mathcal{M}$ is realtime. During the computation of $\mathcal{M}$ on input prefixes a^+ no combination of state and content of the pushdown store may appear twice. If otherwise

$$(\lambda, q_0, a^n b^n, \perp) \vdash^* (a^{m_1}, q_1, a^{n-m_1} b^n, \sigma_1) \vdash^+ (a^{m_1+m_2}, q_1, a^{n-m_1-m_2} b^n, \sigma_1)$$

would be the beginning of an accepting computation, then $(\lambda, q_0, a^{n-m_2} b^n, \perp) \vdash^* (a^{m_1}, q_1, a^{n-m_1-m_2} b^n, \sigma_1)$ would also be the beginning of an accepting computation, but $a^{n-m_2} b^n$ does not belong to L. In particular, this implies that each height of the pushdown store may appear only finitely often and, thus, that the height increases arbitrarily. So, $\mathcal{M}$ runs into a loop while processing a's, that is, the combination of a state and, for any fixed number k, some k topmost pushdown store symbols α appear again and again. To render the loop more precisely, let $(a^{n-x}, q, a^x b^n, \alpha\gamma)$ be a configuration of the loop. Then there is a successor configuration with the same combination of state and topmost pushdown store symbols $(a^{n-x+y}, q, a^{x-y} b^n, \alpha\beta)$. Moreover, we may choose α such that during the computation starting in $(a^{n-x}, q, a^x b^n, \alpha\gamma)$ no symbol of γ is touched, that is, $\alpha\beta = \alpha\gamma'\gamma$. Therefore, the computation continues as $(a^{n-x+y}, q, a^{x-y} b^n, \alpha\gamma'\gamma) \vdash^+ (a^{n-x+2y}, q, a^{x-2y} b^n, \alpha\gamma'\gamma'\gamma)$.

Next, we turn to the input suffixes. While $\mathcal{M}$ processes the input suffixes b^+, again, no combination of state and content of the pushdown store may appear twice. If otherwise

$$(a^n, q_2, b^n, \sigma_2) \vdash^* (a^n b^{m_1}, q_3, b^{n-m_1}, \sigma_3) \vdash^+ (a^n b^{m_1+m_2}, q_3, b^{n-m_1-m_2}, \sigma_3)$$

would result in an accepting computation, then also

$$(a^n, q_2, b^{n-m_2}, \sigma_2) \vdash^* (a^n b^{m_1}, q_3, b^{n-m_1-m_2}, \sigma_3)$$

would result in an accepting computation but $a^n b^{n-m_2}$ does not belong to L. In particular, this implies that each height of the pushdown store may appear only finitely often. Moreover, in any accepting computation the pushdown store has to be decreased until some symbol of γ appears. Otherwise, we could increase the number of a's by y in order to drive $\mathcal{M}$ through an additional loop while processing the input prefix. The resulting computation would also be accepting but the input does not belong to L. Together we conclude that $\mathcal{M}$ runs into a loop that decreases the height of the pushdown store while processing the b's, and that there are only finitely many combinations of state and content of the pushdown store which are accepting.

Now, consider two different numbers $n_1 < n_2$ such that $\mathcal{M}$ accepts $a^{n_1}b^{n_1}$ and $a^{n_2}b^{n_2}$ in the same combinations of state and content of the pushdown store, say in state q_a with γ_a in the pushdown store. We have the forward computations $(\lambda, q_0, a^{n_1}b^{n_1}, \bot) \vdash^{n_1} (a^{n_1}, q_1, b^{n_1}, \gamma_1) \vdash^{n_1} (a^{n_1}b^{n_1}, q_a, \lambda, \gamma_a)$ and $(\lambda, q_0, a^{n_2}b^{n_2}, \bot) \vdash^{n_1} (a^{n_1}, q_1, a^{n_2-n_1}b^{n_2}, \gamma_1) \vdash^{n_2-n_1} (a^{n_2}, q_2, b^{n_2}, \gamma_2) \vdash^{n_2} (a^{n_2}b^{n_2}, q_a, \lambda, \gamma_a)$. Since $\mathcal{M}$ is reversible and runs through loops while processing the b's, the backward computation also runs through loops that now increase the height of the pushdown store. This backward loop cannot be left while reading b's. So, we have $(a^{n_1}b^{n_1}, q_a, \lambda, \gamma_a) \dashv^{n_1} (a^{n_1}, q_1, b^{n_1}, \gamma_1)$ and $(a^{n_2}b^{n_2}, q_a, \lambda, \gamma_a) \dashv^{n_1} (a^{n_2}b^{n_2-n_1}, q_1, b^{n_1}, \gamma_1) \dashv^{n_2-n_1} (a^{n_2}, q_2, b^{n_2}, \gamma_2)$. Due to the deterministic behavior and the reversibility the last step implies $(a^{n_2}, q_2, b^{n_2}, \gamma_2) \vdash^{n_2-n_1} (a^{n_2}b^{n_2-n_1}, q_1, b^{n_1}, \gamma_1)$.

Finally, we consider the input $a^{n_2}b^{n_2-n_1}a^{n_2-n_1}b^{n_2}$ which does not belong to L. However, we obtain the accepting computation

$$\begin{aligned}(\lambda, q_0, a^{n_2}b^{n_2-n_1}a^{n_2-n_1}b^{n_2}, \bot) &\vdash^{n_2} (a^{n_2}, q_2, b^{n_2-n_1}a^{n_2-n_1}b^{n_2}, \gamma_2) \vdash^{n_2-n_1}\\ (a^{n_2}b^{n_2-n_1}, q_1, a^{n_2-n_1}b^{n_2}, \gamma_1) &\vdash^{n_2-n_1} (a^{n_2}b^{n_2-n_1}a^{n_2-n_1}, q_2, b^{n_2}, \gamma_2) \vdash^{n_2}\\ &(a^{n_2}b^{n_2-n_1}a^{n_2-n_1}b^{n_2}, q_a, \lambda, \gamma_a),\end{aligned}$$

a contradiction. □

Lemma 2 together with Theorem 2 show that the family $\mathscr{L}$(REV-PDA) is strictly included in the family of languages accepted by realtime deterministic pushdown automata. So, let us impose another natural restriction on languages accepted by realtime deterministic pushdown automata. Not only in connection with reversibility it is interesting to consider realtime deterministic context-free languages whose reversals are also realtime deterministic context-free languages.

By Example 2 the language $\{ a^m c b^n e a^m \mid m, n \geq 0 \} \cup \{ a^n d b^n e a^m \mid m, n \geq 0 \}$ belongs to $\mathscr{L}$(REV-PDA), but its reversal is known not to be accepted by any realtime deterministic pushdown automaton. Conversely, language $\{ a^n b^n \mid n \geq 0 \}$ as well as its reversal are realtime deterministic context free, but not accepted by any reversible pushdown automaton. So, we derive the following corollary.

Corollary 1. *The family $\mathscr{L}$(REV-PDA) is incomparable with the family of realtime deterministic context-free languages whose reversals are also realtime deterministic context-free languages.*

Furthermore, Lemma 2 together with the language $\{ a^n cb^n \mid n \geq 0 \}^*$ of Example 2 reveals the following corollary.

Corollary 2. *The families of linear context-free languages and $\mathscr{L}$(REV-PDA) are incomparable.*

In [8] it has been shown that there are regular languages which are not accepted by any reversible finite automaton. Next, we show that the regular languages are included in $\mathscr{L}$(REV-PDA).

Theorem 3. *The regular languages are strictly included in $\mathscr{L}$(REV-PDA).*

Proof. By Example 1 the non-regular language $\{ wcw^R \mid w \in \{a,b\}^* \}$ belongs to $\mathscr{L}$(REV-PDA).

On the other hand, given a deterministic finite automaton $\mathcal{M}$ with state set Q, input alphabet Σ, initial state q_0, set of accepting states F, and transition function $\delta : Q \times \Sigma \to Q$, we construct an equivalent REV-PDA $\mathcal{M}'$. Basically, the idea is to simulate $\mathcal{M}$ in the finite control of $\mathcal{M}'$ directly, and to store the state history on the pushdown store. Formally, let $\mathcal{M}' = \langle Q, \Sigma, \Gamma, \delta', q_0, \bot, F \rangle$, where $\Gamma = Q \cup \{\bot\}$ and $\delta'(q,a,q') = (\delta(q,a), qq')$, for all $q \in Q$, $q' \in \Gamma$, and $a \in \Sigma$. The reverse transition δ'_R is derived as $\delta'_R(p,a,q) = (q,\lambda)$.

By construction, $\mathcal{M}'$ and $\mathcal{M}$ are equivalent and $\mathcal{M}'$ is reversible. □

Summarizing the results so far, we have obtained the following hierarchy, where REG denotes the regular and $\mathscr{L}_{rt}$(DPDA) the realtime deterministic context-free languages: REG $\subset \mathscr{L}$(REV-PDA) $\subset \mathscr{L}_{rt}$(DPDA) $\subset \mathscr{L}$(DPDA)

4 Closure Properties

In this section, we study the closure properties of REV-PDAs. It turned out that REV-PDAs and DPDAs have similar closure properties, but the former are interestingly not closed under union and intersection with regular languages.

Lemma 3. *$\mathscr{L}$(REV-PDA) is closed under complementation.*

Next, we consider the operations intersection and union with regular languages and first give another example which enables us to show the non-closure under both operations.

Example 3. The language $\{ w \in \{a,b\}^* \mid |w|_a = |w|_b \}$ is accepted by the REV-PDA $\mathcal{M} = \langle \{q_0, q_1\}, \{a,b\}, \{A, A', B, B', \bot\}, \delta, q_0, \bot, \{q_0\} \rangle$ where the transition functions δ and δ_R are as follows.

Transition function δ	Reverse transition function δ_R
(1) $\delta(q_0, a, \bot) = (q_1, A'\bot)$	(1) $\delta_R(q_0, a, \bot) = (q_1, B'\bot)$
(2) $\delta(q_0, b, \bot) = (q_1, B'\bot)$	(2) $\delta_R(q_0, b, \bot) = (q_1, A'\bot)$
(3) $\delta(q_1, a, A') = (q_1, AA')$	(3) $\delta_R(q_1, b, A') = (q_1, AA')$
(4) $\delta(q_1, a, A) = (q_1, AA)$	(4) $\delta_R(q_1, b, A) = (q_1, AA)$
(5) $\delta(q_1, b, A) = (q_1, \lambda)$	(5) $\delta_R(q_1, a, A) = (q_1, \lambda)$
(6) $\delta(q_1, b, A') = (q_0, \lambda)$	(6) $\delta_R(q_1, a, A') = (q_0, \lambda)$
(7) $\delta(q_1, b, B') = (q_1, BB')$	(7) $\delta_R(q_1, a, B') = (q_1, BB')$
(8) $\delta(q_1, b, B) = (q_1, BB)$	(8) $\delta_R(q_1, a, B) = (q_1, BB)$
(9) $\delta(q_1, a, B) = (q_1, \lambda)$	(9) $\delta_R(q_1, b, B) = (q_1, \lambda)$
(10) $\delta(q_1, a, B') = (q_0, \lambda)$	(10) $\delta_R(q_1, b, B') = (q_0, \lambda)$

□

Lemma 4. *$\mathscr{L}$(REV-PDA) is not closed under union and intersection with regular languages.*

Proof. Consider $L = \{ w \in \{a, b\}^* \mid |w|_a = |w|_b \} \cap a^*b^*$. □

On the other hand, we obtain the closure under intersection and union with regular languages under the condition that the regular language can be accepted by a reversible deterministic finite automaton. In [8] reversibility in finite automata is defined as the property of having only deterministic forward and backward computations. Additionally, the automata may possess several initial and accepting states. Here, we define a regular language as *reversible* if it is accepted by some reversible deterministic finite automaton which possesses one initial state only and obtain a proper subclass of the languages defined in [8].

Lemma 5. *$\mathscr{L}$(REV-PDA) is closed under union and intersection with reversible regular languages.*

Remark 1. In this context the question may arise whether the union or intersection of a non-regular language from $\mathscr{L}$(REV-PDA) with a non-reversible regular language is always a non-reversible language. The following example shows that there are cases which lead to REV-PDAs although the regular language is not reversible. The union of the languages $\{ a^n cb^n \mid n \geq 0 \}$ and a^*b^*, where the latter is not reversible [8], is accepted by some REV-PDA. □

Lemma 6. *$\mathscr{L}$(REV-PDA) is not closed under concatenation, Kleene star, λ-free homomorphism, and reversal.*

Remark 2. It is worth mentioning that there are two situations in which closure results of the above-mentioned operations are obtained. The first result is that $\mathscr{L}$(REV-PDA) is closed under marked concatenation and marked Kleene star.

A second result one can easily observe is that the reversal L^R of a language $L \in \mathscr{L}$(REV-PDA) belongs to $\mathscr{L}$(REV-PDA) if L is accepted by a REV-PDA which has one accepting state only and in which every accepting computation ends in a configuration with empty (up to $\bot$) pushdown store. □

5 Decidability Questions

Problems which are decidable for DPDAs are decidable for REV-PDAs as well. Therefore, emptiness, universality, equivalence, and regularity are decidable for REV-PDAs. On the other hand, inclusion is known to be undecidable for DPDAs. We now show that inclusion is undecidable for REV-PDAs, too. To this end, we use a reduction from Post's correspondence problem (PCP) which is known to be undecidable (see, e.g., [9]). Let Σ be an alphabet and an instance of the PCP be given by two lists $\alpha = u_1, u_2, \ldots, u_k$ and $\beta = v_1, v_2, \ldots, v_k$ of words from Σ^+. Furthermore, let $A = \{a_1, a_2, \ldots, a_k\}$ be an alphabet with k symbols, $\Sigma \cap A = \emptyset$, and $d = \max\{|u_i|, |v_i| \mid 1 \leq i \leq k\}$ be the maximal length of words occurring in α and β. Now, consider two languages L_α and L_β.

$$L_\alpha = \{u_{i_1} u_{i_2} \ldots u_{i_m} \$ a_{i_m}^{d+2} a_{i_{m-1}}^{d+2} \ldots a_{i_1}^{d+2} \mid m \geq 1, 1 \leq i_j \leq k, 1 \leq j \leq m\}$$
$$L_\beta = \{v_{i_1} v_{i_2} \ldots v_{i_m} \$ a_{i_m}^{d+2} a_{i_{m-1}}^{d+2} \ldots a_{i_1}^{d+2} \mid m \geq 1, 1 \leq i_j \leq k, 1 \leq j \leq m\}$$

Lemma 7. *The languages L_α and L_β as well as their reversals L_α^R and L_β^R are accepted by REV-PDAs.*

Lemma 8. *Let $\mathcal{M}_1$ and $\mathcal{M}_2$ be two REV-PDAs. Then it is undecidable whether $L(\mathcal{M}_1) \subseteq L(\mathcal{M}_2)$.*

Theorem 4. *Let $\mathcal{M}$ be a nondeterministic pushdown automaton. Then it is undecidable whether $L(\mathcal{M}) \in \mathscr{L}(\text{REV-PDA})$.*

Proof. We consider an instance of the PCP and define the languages $L_1 = L_\alpha \# L_\beta^R$ and $L_2 = \{\, w \# w^R \mid w \in (\Sigma \cup A \cup \{\$\})^* \,\} \cap \Sigma^* \$ A^* \# A^* \$ \Sigma^*$.

Language L_1 belongs to $\mathscr{L}$(REV-PDA) due to Lemma 7 and the closure under marked concatenation discussed in Remark 2. Language L_2 belongs to $\mathscr{L}$(REV-PDA) due to Example 1 and the closure under intersection with reversible regular languages shown in Lemma 5. Due to the closure under complementation we obtain that $\overline{L_1} \cup \overline{L_2}$ is context free. We will now show that $\overline{L_1} \cup \overline{L_2}$ belongs to $\mathscr{L}$(REV-PDA) if and only if the given instance of the PCP has no solution. If the instance has no solution, then $L_1 \cap L_2 = \emptyset$ and, thus, its complement $\overline{L_1} \cup \overline{L_2}$ is the regular language $(\Sigma \cup A \cup \{\#, \$\})^*$, which belongs to $\mathscr{L}$(REV-PDA) due to Theorem 3. On the other hand, if $\overline{L_1} \cup \overline{L_2}$ belongs to $\mathscr{L}$(REV-PDA), then its complement $L_1 \cap L_2$ belongs to $\mathscr{L}$(REV-PDA) as well. We have to show that the given instance of the PCP has no solution. By way of contradiction we assume that the instance has a solution. Then, $L_1 \cap L_2$ is an infinite, context-free language. Let $w = u_1 u_2 \ldots u_m \$ a_m^{d+2} a_{m-1}^{d+2} \ldots a_1^{d+2} \# a_1^{d+2} a_2^{d+2} \ldots a_m^{d+2} \$ v_m v_{m-1} \ldots v_1$ be a word in $L_1 \cap L_2$ long enough such that the pumping lemma for context-free languages applies. Pumping leads to words which are not in $L_1 \cap L_2$ and we obtain a contradiction.

Now, if we could decide whether the context-free language $\overline{L_1} \cup \overline{L_2}$ belongs to $\mathscr{L}$(REV-PDA), then we could decide whether or not the given instance of the PCP has a solution which is a contradiction. □

The same problem of Theorem 4 for deterministic pushdown automata is open. However, we have the following decidable property which contrasts the result that there is no algorithm which decides whether or not, for example, a given cellular automaton or iterative array is reversible [6,5].

Theorem 5. *Let $\mathcal{M}$ be a deterministic pushdown automaton. Then it is decidable whether $\mathcal{M}$ is a REV-PDA.*

Proof. In order to decide whether a given deterministic pushdown automaton $\mathcal{M} = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F\rangle$ is reversible or not, in general, it is not sufficient to inspect the transition function. Whether or not a transition can be reversed depends on the information that is available after performing it. If this information is unique for all in-transitions to a state, then the transition can be reversed. For example, $\delta(q, a, Z) = (q', Z'Z)$ or $\delta(q, a, Z) = (q', Z')$ provides the state q', the input symbol a, and the topmost pushdown symbol Z'. On the other hand, consider $\delta(q, a, Z) = (q', \lambda)$ which provides only the state q and the input symbol a. The necessary information is complemented by the second symbol on the pushdown store, which cannot be determined by inspecting the transition function only.

In order to cope with the problem, we first construct an equivalent DPDA $\mathcal{M}' = \langle Q, \Sigma, \Gamma', \delta', q_0, \perp, F\rangle$, where $\Gamma' = \Gamma^2 \cup \{\perp\}$ and

$$\delta'(q, a, \perp) = \begin{cases} (q', \perp) & \text{if } \delta(q, a, \perp) = (q', \perp) \\ (q', (Z\perp)\perp) & \text{if } \delta(q, a, \perp) = (q', Z\perp) \end{cases}$$

$$\delta'(q, a, (Y_1Y_2)) = \begin{cases} (q', (ZY_2)) & \text{if } \delta(q, a, Y_1) = (q', Z) \\ (q', (ZY_1)(Y_1Y_2)) & \text{if } \delta(q, a, Y_1) = (q', ZY_1) \\ (q', \lambda) & \text{if } \delta(q, a, Y_1) = (q', \lambda) \end{cases} .$$

By construction there is a bijection φ between the configurations passed through by $\mathcal{M}$ and $\mathcal{M}'$, where

$$\varphi(v, q, w, Z_1Z_2Z_3 \cdots Z_k\perp) = (v, q, w, (Z_1Z_2)(Z_2Z_3) \cdots (Z_{k-1}Z_k)(Z_k\perp)\perp).$$

Moreover, $\mathcal{M}$ and $\mathcal{M}'$ have the same initial configurations (up to the second component of the initial state of $\mathcal{M}'$) and a configuration c_a of $\mathcal{M}$ is accepting if and only if $\varphi(c_a)$ is an accepting configuration of $\mathcal{M}'$. Therefore, $\mathcal{M}$ and $\mathcal{M}'$ accept the same language.

Basically, the idea of the construction is to store information of the topmost pushdown symbol in the state and information of the second pushdown symbol in the topmost pushdown symbol. The construction may introduce also transitions for situations that cannot appear. For example, if in any computation there is never a Z on top of a Y in the pushdown store, then the transition $\delta'(q, a, (ZY))$ is useless. However, if a transition of the form $\delta'(q, a, (Y_1Y_2)) = (q', \lambda)$ is applied, then we do now have the necessary information to test for uniqueness after having performed the transition as mentioned above. That is, we know the state q', the input symbol a, and the topmost pushdown symbol Y_2. So, basically, it remains to be tested whether a transition is applied in some computation or whether it is useless.

To this end, we label the transitions of δ' uniquely, say by the set of labels $B = \{l_1, l_2, \ldots, l_k\}$. Then we apply an old trick and consider words over the alphabet B. On input $u \in B^*$ a DPDA $\tilde{\mathcal{M}}$ with all states final tries to imitate a computation of $\mathcal{M}'$ by applying in every time step the transition whose label is currently read. If $\tilde{\mathcal{M}}$ accepts some input $u_1u_2 \cdots u_n$, then there is a computation (not necessarily accepting) of $\mathcal{M}'$ that uses the transitions $u_1u_2 \cdots u_n$ in this order. If conversely there is a computation of $\mathcal{M}'$ that uses the transitions $u_1u_2 \cdots u_n$ in this order, then $u_1u_2 \cdots u_n$ is accepted by $\tilde{\mathcal{M}}$. So, in order to determine whether a transition with label l_i of $\mathcal{M}'$ is useful, it suffices to decide whether $\tilde{\mathcal{M}}$ accepts an input containing the letter l_i. This decision can be done by testing the emptiness of the deterministic context-free language $L(\tilde{\mathcal{M}}) \cap B^* l_i B^*$.

Assume that $\mathcal{M}''$ is constructed from $\mathcal{M}'$ by deleting all useless transitions that never appear in any computation. Clearly, $\mathcal{M}''$ and $\mathcal{M}$ are equivalent. Now, for any state we consider all in-transitions and check whether the corresponding information after performing it (state, input symbol and pushdown symbol) is unique. If this is true for all states, then $\mathcal{M}$ is reversible and irreversible otherwise. □

Corollary 3. *Let $\mathcal{M}$ be a nondeterministic pushdown automaton. Then it is decidable whether $\mathcal{M}$ is a REV-PDA.*

Proof. By inspecting the transition function it is easy to decide whether or not $\mathcal{M}$ is a DPDA. If the answer is yes, then it can be decided whether or not $\mathcal{M}$ is a REV-PDA by Theorem 5. If $\mathcal{M}$ is not a DPDA, then it cannot be a REV-PDA. □

References

1. Angluin, D.: Inference of reversible languages. J. ACM 29, 741–765 (1982)
2. Bennet, C.H.: Logical reversibility of computation. IBM J. Res. Dev. 17, 525–532 (1973)
3. Ginsburg, S., Greibach, S.A.: Deterministic context-free languages. Inform. Control 9, 620–648 (1966)
4. Harrison, M.A.: Introduction to Formal Language Theory. Addison-Wesley, Reading (1978)
5. Kutrib, M., Malcher, A.: Fast reversible language recognition using cellular automata. Inform. Comput. 206, 1142–1151 (2008)
6. Kutrib, M., Malcher, A.: Real-time reversible iterative arrays. Theor. Comput. Sci. 411, 812–822 (2010)
7. Morita, K., Shirasaki, A., Gono, Y.: A 1-tape 2-symbol reversible Turing machine. Trans. IEICE E72, 223–228 (1989)
8. Pin, J.E.: On reversible automata. In: Simon, I. (ed.) LATIN 1992. LNCS, vol. 583, pp. 401–416. Springer, Heidelberg (1992)
9. Salomaa, A.: Formal Languages. Academic Press, London (1973)

String Extension Learning Using Lattices

Anna Kasprzik[1] and Timo Kötzing[2]

[1] FB IV – Abteilung Informatik, Universität Trier, 54286 Trier, Germany
kasprzik@informatik.uni-trier.de

[2] Department 1: Algorithms and Complexity, Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
koetzing@mpi-inf.mpg.de

Abstract. The class of regular languages is not identifiable from positive data in Gold's language learning model. Many attempts have been made to define interesting classes that *are* learnable in this model, preferably with the associated learner having certain advantageous properties. Heinz '09 presents a set of language classes called *String Extension (Learning) Classes*, and shows it to have several desirable properties.

In the present paper, we extend the notion of String Extension Classes by basing it on *lattices* and formally establish further useful properties resulting from this extension. Using lattices enables us to cover a larger range of language classes including the *pattern languages*, as well as to give various ways of *characterizing* String Extension Classes and its learners. We believe this paper to show that String Extension Classes are learnable in a *very natural way*, and thus worthy of further study.

1 Introduction

In this paper, we are mostly concerned with learning as defined by Gold [11] which is sometimes called *learning in the limit from positive data*.

Formally, for a class of (computably enumerable) languages $\mathcal{L}$ and an algorithmic learning function h, we say that h *TxtEx-learns* $\mathcal{L}$ [11,13] iff, for each $L \in \mathcal{L}$, for every function T enumerating (or presenting) all and only the elements of L, as h is fed the succession of values $T(0), T(1), \ldots$, it outputs a corresponding succession of programs $p(0), p(1), \ldots$ from some hypothesis space, and, for some i_0, for all $i \geq i_0$, $p(i)$ is a correct program for L, and $p(i+1) = p(i)$. The function T is called a *text* or *presentation* for L.

There are two main viewpoints in research on language learning: Inductive Inference (II) and Grammatical Inference (GI). The area of Inductive Inference is mainly concerned with the question if a certain target concept, which in our case usually represents a formal language class, can be identified in the limit, i.e., after any finite number of steps. The area of Grammatical Inference is mainly concerned with the concrete algorithms solving that task and with their efficiency, i.e., with the question if the number of steps needed can be bounded by some polynomial with respect to a relevant measure such as the input size or the number of queries asked, if admissible. As a result, research on GI is more involved with the task of inferring a specific description of a formal language (e.g., a grammar or an automaton) than just the language as an abstract item, as the inference strategy of any concrete learning algorithm is intrinsically linked to the description it yields as output (for an overview of GI, see [8]). In this paper, we have tried to include results of importance from both perspectives.

Gold [11] already showed that the class of regular languages is *not* TxtEx-learnable. Several papers, for example [9,12], are concerned with finding interesting classes of languages that *are* TxtEx-learnable. Furthermore, frequently it is desirable for a learner to have additional properties, and one wants to find interesting classes learnable by a learner having these properties.

In this paper, we extend and analyze the notion of *String Extension Learning* as given in [12]. We do this by applying Birkhoff-style lattice theory and require all conjectures a learner makes to be drawn from a lattice. Section 2 makes String Extension Learning precise. Importantly, in Theorem 6 we show the resulting learners to have a long list of advantageous properties.

Many simple, but also several more complex languages classes are learnable this way. Some examples are given in [12]; we show in Section 3 how *Pattern Languages* can be learned as a subclass of a String Extension Language Class. Furthermore, Section 3 discusses in what respect *Distinction Languages* [9] are String Extension Languages as well.

Section 4 analyzes String Extension Learners (SELs) and String Extension Classes (SECs) further. We give *two* insightful characterization of SELs in Theorem 14, and *three* characterizations of SECs in Theorem 15. This establishes the SECs as very naturally arising learnable language classes.

Section 5 studies how String Extension Classes and special cases thereof are learnable via *queries* [2]. Complexity issues are discussed and it is shown that String Extension Languages can be learned in a particularly straightforward way from *equivalence queries.*

As an anonymous referee pointed out, the learning setting presented in [7] is similar to String Extension Learning – in fact, it corresponds to String Extension Learning with *complete lattices.*

Familiarity with lattice theory is useful to understand this paper, but not completely necessary. For introductions into lattice theory, the reader is referred to the textbooks [4] (a classic) and [16] (available online).

We omit many proofs due to space constraints. The proof of Theorem 6 is given below and exemplary for several of the omitted proofs. A complete version can be found at `http://www.mpi-inf.mpg.de/~koetzing/StringExtensionLearnersTR.pdf`.

2 Definitions and Basic Properties

Any unexplained complexity-theoretic notions are from [19]. All unexplained general computability-theoretic notions are from [18].

$\mathbb{N}$ denotes the set of natural numbers, $\{0, 1, 2, \ldots\}$. We let Σ be a countable alphabet (a non-empty countable set; we allow for – countably – infinite alphabets), and by Σ^* we denote the set of all finite words over Σ. A *language* is any set $L \subseteq \Sigma^*$. For each k, Σ^k denotes the set of all words of length exactly k. We denote the empty word by ε and the length of a word x by $|x|$.

By $\mathbb{S}\mathrm{eq}$ we denote the set of finite sequences over $\Sigma^* \cup \{\#\}$, where $\#$ is a special symbol called "pause". We denote the empty sequence by $\emptyset$. For a sequence $\sigma \in \mathbb{S}\mathrm{eq}$, we let $\mathrm{len}(\sigma)$ denote the length σ, and, for all $i < \mathrm{len}(\sigma)$ we let $\sigma(i)$ be the $i+1$-th element of σ. Concatenation on sequences is denoted by $\diamond$. For all $\sigma \in \Sigma^*$, we let $\mathrm{content}(\sigma) = \{x \in \Sigma^* \mid \exists i < \mathrm{len}(\sigma) : \sigma(i) = x\}$.

The symbols $\subseteq, \subset, \supseteq, \supset$ respectively denote the subset, proper subset, superset and proper superset relation between sets. For sets A, B, we let $A \setminus B = \{a \in A \mid a \notin B\}$, $\overline{A}$ be the complement of A and $\mathrm{Pow}(A)$ ($\mathrm{Pow}_{\mathrm{fin}}(A)$) be the set of all (finite) subsets of A.

The quantifier $\forall^\infty x$ means "for all but finitely many x", the quantifier $\exists^\infty x$ means "for infinitely many x". For any set A, $|A|$ denotes the cardinality of A.

We let dom and range denote, respectively, domain and range of a given function. We sometimes denote a function f of $n > 0$ arguments $x_1, \ldots, x_n$ in lambda notation (as in Lisp) as $\lambda x_1, \ldots, x_n \centerdot f(x_1, \ldots, x_n)$. For example, with $c \in \mathbb{N}$, $\lambda x \centerdot c$ is the constantly c function of one argument.

A function ψ is *partial computable* iff there is a deterministic, multi-tape Turing machine computing ψ. $\mathcal{P}$ and $\mathcal{R}$ denote, respectively, the set of all partial computable and the set of all total (partial) computable functions $\mathbb{N} \to \mathbb{N}$. We say that ψ is *polytime* iff ψ is computable in polynomial time. If a function f is defined for $x \in \mathrm{dom}(f)$ we write $f(x)\downarrow$, and we say that f on x *converges.*

For all p, W_p denotes the computably enumerable ($\mathtt{ce}$) set $\mathrm{dom}(\varphi_p)$.

We say that a function f *converges to* p iff $\forall^\infty x : f(x)\downarrow = p$.

Whenever we consider (partial) computable functions on objects like finite sequences or finite sets, we assume those objects to be efficiently coded as natural numbers. We also assume words to be so coded. The size of any such finite object is the size of its code number.

Note that, for infinite alphabets, the size of words of length 1 is unbounded.

String Extension Learning

After these general definitions, we will now turn to definitions more specific to this paper. First we introduce lattices and String Extension Spaces and then show how we use them for learning.

Definition 1. A pair $(V, \sqsubseteq)$ is a *partially ordered set* iff

- $\forall a, b \in V : a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$;
- $\forall a, b \in V : a \sqsubseteq a$;
- $\forall a, b, c \in V : a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$.

Let $(V, \sqsubseteq)$ be a partially ordered set. For any set $S \subseteq V$, $v \in V$ is called

- an *upper bound of* S iff $\forall a \in S : a \sqsubseteq v$;
- an *lower bound of* S iff $\forall a \in S : v \sqsubseteq a$;
- a *maximum of* S iff v is upper bound of S and $v \in S$;
- a *minimum of* S iff v is lower bound of S and $v \in S$;

– a *least upper bound* or *supremum of S* iff v is the minimum of the set of upper bounds of S;
– a *greatest lower bound* or *infimum of S* iff v is the maximum of the set of lower bounds of S;

Note that, for a given set, there is at most one supremum and at most one infimum. If V has a minimum element, we denote it by $\bot_V$, a maximum element by $\top_V$. $(V, \sqsubseteq)$ is called

– an *upper semi-lattice* iff each two elements of V have a supremum;
– a *lower semi-lattice* iff each two elements of V have an infimum;
– a *lattice* iff each two elements of V have a supremum and an infimum.

In an upper semi-lattice, the supremum of two elements $a, b \in V$ is denoted by $a \sqcup b$ and we use $\bigsqcup$ to denote suprema of sets D (note that, in an upper semi-lattice, each non-empty finite set has a supremum, which equals the iterated supremum of its elements, as the binary supremum is an associative operation); if V has a minimum element, then, by convention, $\bigsqcup \emptyset = \bot_V$. In a lower semi-lattice, the infimum of two elements $a, b \in V$ is denoted by $a \sqcap b$.

For two partially ordered sets V, W a function $h : V \to W$ is called an *order embedding* iff, for all $a, b \in V$, $a \sqsubseteq_V b \Leftrightarrow h(a) \sqsubseteq h(b)$. An *order isomorphism* is a bijective order embedding.

For example, for each $k \in \mathbb{N}$, the set of all finite sets that contain only words of length k, with inclusion as the order, is a lattice. We call this lattice V_{fac-k}.

Definition 2. For an upper semi-lattice V and a function $f : \Sigma^* \to V$ such that f and $\sqcup$ are (total) computable, (V, f) is called a *String Extension Space (SES)* iff, for each $v \in V$, there is a finite $D \subseteq \text{range}(f)$ with $\bigsqcup_{x \in D} x = v$.[1]
(V, f) is called *polytime* iff f and suprema in V are polytime.

We get the special case of [12] if we take the lattices of all finite subsets of a finite set A with inclusion.

As an example, for each k, we let $fac_k : \Sigma^* \to V_{fac-k}, x \mapsto \{v \in \Sigma^k \mid \exists u, w \in \Sigma^* : x = uvw\}$. Then (V_{fac-k}, fac_k) is an SES.[2]

Definition 3. Let (V, f) be an SES.

– A *grammar* is any $v \in V$.[3]
– The *language of grammar* v is $L_f(v) = \{w \in \Sigma^* \mid f(w) \sqsubseteq v\}$.

[1] This definition might seem a little strange at first, and in general one could define SESes without those restrictions. However, elements that are not the finite suprema of elements from range(f) are not directly useful for our purposes, and many of our theorems would have to be stated in terms of the "stripped" sub semi-lattice one gets from restricting to all elements which are finite union of elements from range(f). Thus, for notational purposes, we only allow for "stripped" SESes in the first place.

[2] Any substring of length k of a word x is called a *k-factor* of x.

[3] Note that we assume our grammars to be finite with respect to a relevant measure, i.e., containing for example a finite number of admissible substrings, or other rules.

- The *class of languages* obtained by all possible grammars is $\mathcal{L}_f = \{L_f(v) \mid v \in V\}$.

We define ϕ_f such that $\forall \sigma : \phi_f(\sigma) = \bigsqcup_{x \in \text{content}(\sigma)} f(x)$.

Any class of languages $\mathcal{L}$ such that there is an SES (V, f) with $\mathcal{L} = \mathcal{L}_f$ is called a *String Extension Class (SEC)*, ϕ_f a *String Extension Learner (SEL)*.[4] We use SEC to denote the set of all String Extension Classes. Further, we will omit the subscript of f if it is clear from context.

For example, with respect to (V_{fac-2}, fac_2), $\{aa, ab\}$ is a grammar for the set of all words for which any contiguous subword of length 2 is either aa or ab. Example such words include aaa, ab, $aaaab$, ccc, ...

Next we define what we mean by "learning".

Definition 4. Let $L \subseteq \Sigma^*$ and $T : \mathbb{N} \to \Sigma^*$. T is called a *text for* L iff $\text{content}(T) = L$. For any text T and any k, let $T[k]$ denote the sequence of the first k elements of T. Let $h : \mathbb{S}\text{eq} \to \mathbb{N}$ be a (total computable) learner. We assume the outputs of h to be mapped by a function $L(\cdot)$ to a language. Whenever no concrete function $L(\cdot)$ is given, we assume the mapping $\lambda p \boldsymbol{.} W_p$.

The learner $h \in \mathcal{P}$ is said to **TxtEx**-*identify* a languages L with respect to $L(\cdot)$ iff, for each text T for L, there is $k \in \mathbb{N}$ such that

(i) $L(h(T[k])) = L$; and
(ii) for all $k' \geq k$, $h(T[k']) = h(T[k])$.

For the minimum such k, we then say that h on T has *converged* after k steps, and denote this by $\text{Conv}(h, T) = k$.

We denote the set of all languages **TxtEx**-*identified* by a learner h with **TxtEx**(h). We say that a class of languages $\mathcal{L}$ is **TxtEx**-*identified* (possibly with certain properties) iff there is a learner $h \in \mathcal{P}$ (observing those properties) **TxtEx**-learning every set in $\mathcal{L}$. Further, we say that h learns a language using a uniformly decidable hypothesis space iff $\lambda x, p \boldsymbol{.} x \in L(p)$ is (total) computable.

The following learner properties have been studied in the literature.

Definition 5. Let a learner $h : \mathbb{S}\text{eq} \to \mathbb{N}$ be given. We call h

- iterative [10,22], iff there is a function $h^{it} : \mathbb{N} \times \Sigma^* \to \mathbb{N}$ such that $\forall \sigma \in \mathbb{S}\text{eq}, w \in \Sigma^* : h^{it}(h(\sigma), w) = h(\sigma \diamond w)$;
- polytime iterative, iff there is a polytime such function h^{it};
- set-driven [21,13], iff there is a function $h^{set} : \text{Pow}_{\text{fin}}(\Sigma^*) \to \mathbb{N}$ such that $\forall \sigma \in \mathbb{S}\text{eq} : h^{set}(\text{content}(\sigma)) = h(\sigma)$;
- globally consistent [3,5,22], iff $\forall \sigma \in \mathbb{S}\text{eq} : \text{content}(\sigma) \subseteq L(h(\sigma))$;
- locally conservative [1], iff $\forall \sigma \in \mathbb{S}\text{eq}, x \in \Sigma^* : h(\sigma) \neq h(\sigma \diamond x) \Rightarrow x \notin L(h(\sigma))$;
- strongly monotone [14], iff $\forall \sigma \in \mathbb{S}\text{eq}, x \in \Sigma^* : L(h(\sigma)) \subseteq L(h(\sigma \diamond x))$;

[4] In general, in formal language theory, several descriptions may define the same language. Observe that for the language classes defined here this is not the case – we have $L_f(u) \neq L_f(v)$ for any two elements $u, v \in V$ with $u \neq v$. See Theorem 10.

- prudent [20,17], iff $\forall \sigma \in \mathbb{S}\mathrm{eq} : L(h(\sigma)) \in \mathbf{TxtEx}(h)$;
- optimal [11], iff, for all learners h' with $\mathbf{TxtEx}(h) \subseteq \mathbf{TxtEx}(h')$,

$$\begin{gathered}\exists L \in \mathbf{TxtEx}(h), T \in \mathbf{Txt}(L) : \mathrm{Conv}(h', T) < \mathrm{Conv}(h, T) \\ \Rightarrow \\ \exists L \in \mathbf{TxtEx}(h), T \in \mathbf{Txt}(L) : \mathrm{Conv}(h, T) < \mathrm{Conv}(h', T).\end{gathered} \tag{1}$$

We briefly show that SELs have a number of desirable properties.

Theorem 6. Let (V, f) be an SES. Then ϕ_f **TxtEx**-learns $\mathcal{L}_f$

(i) iteratively;
(ii) if (V, f) is a polytime SES, polytime iteratively;
(iii) set-drivenly;
(iv) globally consistently;
(v) locally conservatively;
(vi) strongly monotonically;
(vii) prudently; and
(viii) optimally.

Proof. Regarding **TxtEx**-learnability: Let $L \in \mathcal{L}_f$ and let $v \in V$ be such that $L(v) = L$. Let T be a text for L. As (V, f) is an SES, let $D \subseteq \Sigma^*$ such that $v = \bigsqcup_{x \in D} f(x)$. Let k be such that $D \subseteq \mathrm{content}(T[k])$. Then, obviously, $\forall k' \geq k : \phi_f(T[k']) = v$. Regarding the different items of the list, we have:

(i) We let $\phi_f^{it} \in \mathcal{P}$ be such that

$$\forall v, x : \phi_f^{it}(v, x) = \begin{cases} v, & \text{if } x = \#; \\ v \sqcup f(x), & \text{otherwise.} \end{cases} \tag{2}$$

(ii) Clearly, ϕ_f^{it} from (i) is polytime, if (V, f) is a polytime SES.
(iii) Let $\phi^{set} \in \mathcal{P}$ be such that $\forall D : \phi^{set}(D) = \bigsqcup_{x \in D} f(x)$.
(iv) Let σ be a sequence in Σ^*, let $v = \phi_f(\sigma)$ and $x \in \mathrm{content}(\sigma)$. Then $f(x) \sqsubseteq \bigcup_{y \in \mathrm{content}(\sigma)} f(y) = \phi_f(\sigma) = v$. Thus, $x \in L(v)$.
(v) Let $\sigma \in \mathbb{S}\mathrm{eq}$ and $x \in \Sigma^*$ with $\phi_f(\sigma) \neq \phi_f(\sigma \diamond x)$. Thus, $\phi_f(\sigma) \neq \phi_f(\sigma) \cup f(x)$, in particular, $f(x) \not\sqsubseteq \phi_f(\sigma)$. Therefore, $x \notin L(\phi_f(\sigma))$.
(vi) Let $\sigma \in \mathbb{S}\mathrm{eq}$ and $x \in \Sigma^*$. Clearly, $\phi_f(\sigma) \sqsubseteq \phi_f(\sigma \diamond x)$. Thus,

$$\begin{aligned} L_f(\phi_f(\sigma)) &= \{w \in \Sigma^* \mid f(w) \sqsubseteq \phi_f(\sigma)\} \\ &\subseteq \{w \in \Sigma^* \mid f(w) \sqsubseteq \phi_f(\sigma \diamond x)\} \\ &= L_f(\phi_f(\sigma \diamond x)). \end{aligned}$$

(vii) Prudence is clear, as, for all $\sigma \in \mathbb{S}\mathrm{eq}$ and $x \in L_f(\phi_f(\sigma))$, we have $f(x) \sqsubseteq \phi_f(\sigma)$. Hence, for all texts T for $L_f(\phi_f(\sigma))$, ϕ_f on T will converge to $\phi_f(\sigma)$.
(viii) Optimality follows from consistency, conservativeness and prudence, as stated in [17, Proposition 8.2.2A]. □

For each SES (V, f) we will use ϕ_f^{it} and ϕ_f^{set} as shown existent just above.

3 Example SECs

We already came across the example of k-factor languages and its SES (V_{fac-k}, fac_k). Many more examples like this can be found in [12]. In this section we define a more complex example.

Definition 7. Let Σ be an alphabet and let X be a countably infinite set (of *variables*) disjoint from Σ.

Let $\mathbb{P}\mathrm{at} = (\Sigma \cup X)^*$ be the set of all patterns. For any $\pi \in \mathbb{P}\mathrm{at}$, with $w_0, \dots w_{n+1} \in \Sigma^*$ and $x_0, \dots x_n \in X$ such that $\pi = w_0 x_0 w_1 x_1 \dots x_n w_{n+1}$, let

$$L(\pi) = \{w_0 v_{x_0} w_1 v_{x_1} \dots v_{x_n} w_{n+1} \mid \forall x \in X : v_x \in \Sigma^* \setminus \{\varepsilon\}\}$$

denote the set of all strings *matching* the pattern π. We call any L such that there is a pattern π with $L = L(\pi)$, a (non-erasing) *pattern language*. For each $w \in \Sigma^*$, let $\mathrm{pat}(w) = \{\pi \in \mathbb{P}\mathrm{at} \mid w \in L(\pi)\}$ denote the set of patterns matched by w. Note that, for each $w \in \Sigma^*$, $\mathrm{pat}(w)$ is finite.

The pattern languages are not learnable globally consistently and iteratively in a *non-redundant* hypothesis space, see [6, Corollary 12]. The usual iterative algorithm is first published in [15].

Theorem 8. For any finite set $D \subseteq \Sigma^*$, we let $\mathrm{pat}(D) = \bigcap_{w \in D} \mathrm{pat}(w)$.[5] Let V_{pat} be the lattice $\{\mathrm{pat}(D) \mid D \subseteq \Sigma^* \text{ finite}\}$ with order relation $\supseteq$.[6] Then (V_{pat}, pat) is an SES.

Now ϕ_{pat} learns the pattern languages globally consistently and iteratively (as well as with all other properties as given in Theorem 6). Note that some of the grammars of (V_{pat}, pat) are not for pattern languages, for example $\mathrm{pat}(\{a^3, b^4\}) = \{x_1, x_1x_2, x_1x_2x_3, x_1x_1x_2, x_1x_2x_1, x_1x_2x_2\}$.

Also note: One can code the elements of V_{pat}, as all but $\bot_{V_{pat}}$ are finite sets.

Fernau [9] introduced the notion of *function distinguishable languages* (DLs). The following shows that the concept of DLs is subsumed by the concept of SECs, while the concept of SECs is *not* subsumed by the concept of DLs.

Theorem 9

$$\mathrm{DL} \subset \mathrm{SEC}.$$

The inequality is witnessed by a class of regular languages as stated below.

Proof. "$\neq$": Obviously, the class of all finite languages is an SEC but not a DL.

"$\subseteq$": Let $\mathcal{L}$ be a DL. Let h be the learner for $\mathcal{L}$ given in [9, § 6]. By [9, Theorem 35], h fulfills the condition of Theorem 14(iii). Hence, h is a String Extension Learner by Theorem 14 and $\mathcal{L}$ is an SEC. □

For the reader familiar with [9] we specify a concrete SES (V, f) such that ϕ_f learns the class of f'-DLs for any distinguishing function $f' : \Sigma^* \to X$.

[5] By convention, we let $\mathrm{pat}(\emptyset) = \mathbb{P}\mathrm{at}$.

[6] Note that the order is inverted with respect to the usual powerset lattice.

Define V as the set of all stripped[7] f'-distinguishable DFA $\cup \{(\{q_0\}, \Sigma, q_0, \emptyset, \emptyset)\}$, and $\sqsubseteq$ such that $B_1 \sqsubseteq B_2$ iff $L(B_1) \subseteq L(B_2)$ for $B_1, B_2 \in V$.

Obviously, V is a partially ordered set. $(V, \sqsubseteq)$ is also an upper semi-lattice – the supremum B of B_1, B_2 is obtained as follows: Compute the stripped minimal DFA B_0 for $L(B_1) \cup L(B_2)$ (algorithms can be found in the literature). If $B_0 \in V$ then $B := B_0$. Else build a finite positive sample set I_+ by adding all shortest strings leading to an accepting state in B_0, and then for every hitherto unrepresented transition $\delta(q_1, a) = q_2$ $(a \in \Sigma)$ of B_0 adding the string resulting from concatenating a string leading to q_1, a, and a string leading from q_2 to an accepting state. Use the learner h from [9] on I_+. By Lemma 34 and Theorem 35 in [9] the result is a stripped DFA recognizing the smallest f'-distinguishable language containing $L(B_1) \cup L(B_2)$, and since the elements of V are all stripped there is only one such DFA in V, which is the supremum of B_1 and B_2. Also note that $(V, \sqsubseteq)$ has a minimum element $\bot_V = (\{q_0\}, \Sigma, q_0, \emptyset, \emptyset)$.

For any distinguishing function $f' : \Sigma^* \to X$ define $f : \Sigma^* \to V$ by setting $f(w) := A_w$ where A_w is the minimal stripped DFA with $L(A_w) = \{w\}$ (A_w is f'-distinguishable by [9], Lemma 15). We show that (V, f) is an SES.

Obviously, f is computable. For each $v \in V$ there is a finite set $D \in \text{range}(f)$ such that $\bigsqcup_{x \in D} x = v$: Take any two elements $B_1, B_2 \in V$ such that $B_1 \sqcup B_2 = v$ and construct the set I_+ as specified above. We can set $D := I_+$.

Thus, the class of f'-DLs is learnable by ϕ_f.[8]

4 Properties of SECs

In this section we give a number of interesting theorems pertaining to SECs and their learnability. Most importantly, we characterize SELs (Theorem 14) and SECs (Theorem 15).

Theorem 10. Let (V, f) be an SES. Then $(V, \sqsubseteq)$ and $(\mathcal{L}_f, \subseteq)$ are order-isomorphic, with order-isomorphism $L_f(\cdot)$.

Corollary 11. Let (V, f) and (W, g) be two SESes with $\mathcal{L}_f \subseteq \mathcal{L}_g$. Then there is an order embedding $h : V \to W$.

Lemma 12. Let (V, f) be an SES. We have the following.

(i) For all $a, b \in V$, $L(a) \cup L(b) \subseteq L(a \sqcup b)$.
(ii) For all $a \in V$, $L(a) = \Sigma^*$ iff $a = \top_V$.
(iii) If $\mathcal{L}_f$ is closed under (finite) union, then we have, for all $a, b \in V$, $L(a) \cup L(b) = L(a \sqcup b)$.
(iv) If V is a lattice, then we have, for all $a, b \in V$, $L(a) \cap L(b) = L(a \sqcap b)$.

[7] An automaton is stripped when taking away any state or transition would change the language recognized by the automaton.

[8] Note that in a concrete implementation we would not have to construct I_+ when computing suprema in V as we can just use the text seen so far. Also, it seems relatively easy to define an incremental version of the learner from [9].

Theorem 13. Let (V, f) be an SES. We have the following.

(i) $\lambda v, x \centerdot x \in L(v)$ is computable (i.e., $(L(v))_{v \in V}$ is uniformly decidable).
(ii) If (V, f) is polytime, then $\lambda v, x \centerdot x \in L(v)$ is computable in polynomial time (i.e., $(L(v))_{v \in V}$ is uniformly decidable in polynomial time).
(iii) $\mathcal{L}$ is closed under intersection iff V is a lattice.

Now we get to our main theorem of this section, which shows that all learners having a certain subset of the properties listed above in Theorem 6 can necessarily be expressed as SELs.

Theorem 14. Let $h \in \mathcal{R}$. The following are equivalent.

(i) There is an SES (V, f) such that $h = \phi_f$.
(ii) h **TxtEx**-learns $\mathcal{L}$ set-drivenly, globally consistently, locally conservatively and strongly monotonically.
(iii) There is a 1-1 $L(\cdot)$ such that there is a computable function t such that, for all x, v, $t(x, v)$ halts iff $x \in L(v)$ and, for all $\sigma \in \mathbb{S}\mathrm{eq}$, $L(h(\sigma))$ is the $\subseteq$-minimum element of $\mathbf{TxtEx}(h)$ containing all of content(σ).

Theorem 15. Let $\mathcal{L}$ be a set of languages. The following are equivalent.

(i) $\mathcal{L}$ is an SEC.
(ii) $\mathcal{L}$ can be **TxtEx**-learned by a globally consistent, locally conservative, set-driven and strongly monotonic learner.
(iii) There is a 1-1 $L(\cdot)$ such that there is a computable function t such that, for all x, v, $t(x, v)$ halts iff $x \in L(v)$ and a (total) computable function g such that, for all $D \subseteq \Sigma^*$ $L(g(D))$ is the $\subseteq$-minimum element of $\mathcal{L}$ containing D.
(iv) $\mathcal{L}$ can be **TxtEx**-learned by a strongly monotonic set-driven learner using a uniformly decidable hypothesis space.

Proposition 16 just below gives a sufficient condition for a language to be an SEC.

Proposition 16. Let $\mathcal{L}$ be a class of languages closed under intersection and **TxtEx**-learnable set-drivenly, globally consistently and locally conservatively as witnessed by $h \in \mathcal{P}$. Then h is strongly monotone, and, in particular, $\mathcal{L}$ is an SEC.

5 Query Learning of SECs

This section is concerned with learning SECs from queries. We address the issue from a more GI-oriented view, inasmuch as for example some concrete algorithms are given and complexity questions are considered.

Definition 17. Let (V, f) be an SES and $v \in V$ the target to identify.[9] A *membership query (MQ)* for $w \in \Sigma^*$ and $L \subseteq \Sigma^*$ is a query '$w \in L$?' receiving an answer from $\{0, 1\}$ with MQ $= 1$ if $w \in L$ and MQ $= 0$ otherwise.[10] An *equivalence query (EQ)* for $v_0 \in V$ is a query '$v_0 = v$?' receiving an answer from $\Sigma^* \cup \{\texttt{yes}\}$ $(\Sigma^* \cap \{\texttt{yes}\} = \emptyset)$ such that $\mathrm{EQ}(v_0) = \texttt{yes}$ for $L(v_0) = L(v)$ and $\mathrm{EQ}(v_0) = c$ with $[f(c) \sqsubseteq v \wedge \neg f(c) \sqsubseteq v_0] \vee [f(c) \sqsubseteq v_0 \wedge \neg f(c) \sqsubseteq v]$ otherwise.

Let (V, f) be an SES. As $\mathcal{L}_f$ is identifiable in the limit from text (see [12]) $\mathcal{L}_f$ is also identifiable in the limit from MQs: Consider a learner just querying all strings of Σ^* in length-lexical order – at some point the set of all strings w with $\mathrm{MQ}(w) = 1$ queried so far necessarily includes a text for the target.

If we are interested in complexity, unfortunately in general we cannot bound the number of MQs needed in any interesting way. For each $v \in V$, define $\mathbb{T}_v := \{T \subseteq \mathrm{range}(f) \mid \bigsqcup_{t \in T} t = v\}$ and let T_0 be an element of $\mathbb{T}_v$ with minimal cardinality. Obviously, $|T_0|$ is a lower bound on the number of MQs needed to converge on the target. However, note that there exist SECs with properties that allow more specific statements:

Theorem 18. Let (V, f) be an SES. If $\mathcal{L}_f$ is the class of k-factor languages or the class of k-piecewise testable languages (see [12]) identification is possible with a query complexity of $O(|\Sigma|^k)$ MQs.

Proof. We give a simple learning algorithm in pseudo-code that can be used to identify any SEC $\mathcal{L}_f$ such that there is a finite set Q with $\forall v \in V : \exists S \subseteq Q : f(S) \in \mathbb{T}_v$. Observe that we can set $Q := \Sigma^k$ for the k-factor languages and $Q := \Sigma^{\leq k}$ for the k-piecewise testable languages.

```
[Initialize $Q_0$ with the respective $Q$ as given just above];
$v_0 := \bot_V$;
for all $s \in Q_0$ do:
    if MQ($s$) = 1 then $v_0 := v_0 \sqcup f(s)$
return $v_0$.
```

It is easy to see that this algorithm yields the target after $|Q_0|$ loop executions which corresponds to having asked $|Q_0|$ MQs, where $|Q_0| = |\Sigma^k| = |\Sigma|^k$ for the k-factor languages and $|Q_0| = |\Sigma^{\leq k}| = (|\Sigma|^{k+1} - 1)/(|\Sigma| - 1)$ such that $O(|Q_0|) = O(|\Sigma|^k)$ for the k-piecewise testable languages. □

Remark: If the SES is polytime it follows from Theorem 18 that in these special cases identification is possible in polytime as well.

However, as stated above, polytime identification cannot be ensured in the general case. The situation changes if we allow EQs instead of MQs:

[9] To be precise, the concept to infer is a language. However, as no two elements of V define the same language (see Footnote 4) our potential targets are elements of V.

[10] Algorithmically, using MQs only makes sense if the membership problem is decidable. As for SECs we have $f : \Sigma^* \to V$ an MQ for $w \in \Sigma^*$ amounts to checking if $f(w) \sqsubseteq v$.

Theorem 19. Let (V, f) be an SES. Then $\mathcal{L}_f$ is identifiable in the limit from EQs. In particular, for all $v \in V$, if the length of each ascending path from $\bot_V$ to v is at most n then v can be identified using $O(n)$ EQs.

Proof. Let $v \in V$ be the target. We give a concrete learning algorithm:

```
$v_0 := \bot_V$;
$c$ := EQ($v_0$);
while ($c \neq$ yes):
    $v_0 := v_0 \sqcup f(c)$
    $c$ := EQ($v_0$);
return $v_0$.
```

Lemma 20. The algorithm identifies any target $v \in V$ using $O(n)$ EQs.

As $v_0 \sqsubseteq v$ before each loop execution counterexample $c = \text{EQ}(v_0)$ must be chosen such that $f(c) \sqsubseteq v$, which entails $v_0 \sqcup f(c) \sqsubseteq v$. The fact that c is a counterexample implies $\neg(f(c) \sqsubseteq v_0)$ and $v_0 \sqcup f(c) \neq v_0$. Consequently, the successive values of v_0 in the execution of the algorithm form a path from $\bot_V$ to v, where the length of this path equals the number of loop executions. Hence, the target v is identified in finitely many steps using $O(n + 1) = O(n)$ EQs (which corresponds to receiving at most n counterexamples). □

If n can be bounded by some polynomial relating the length of the longest ascending path from $\bot_V$ to v to the size of the grammar, and if (V, f) is polytime, then $\mathcal{L}_f$ is identifiable in polytime from EQs as well. Remark: EQs as such cost as much as it costs the teacher to compare two elements of the lattice.

EQs have two advantages: First, the learner actually *knows* when he has identified the target, namely when the teacher has no more counterexamples to give and answers the next EQ in the positive. And second, unlike in the general cases of learning from text or MQs where the learner has to handle strings $s \in \Sigma^* \setminus L_f(v)$ that do not change the current hypothesis at all, EQs can be used in such a way that every EQ results in the retrieval of at least one more hitherto unrevealed element of the target (which can be seen from the fact that we "make progress" in every loop execution of the algorithm given above).

6 Conclusion and Outlook

We have given a general definition of String Extension Classes and have shown that several natural examples are SECs. We have argued further for the naturality of these language classes by giving various characterizations and properties.

It seems to us that the lattice theoretic framework can be highly beneficial to the analysis of *classes* of learning algorithms. For example, one can analyze the probabilistic learnability of String Extension Classes – we have some promising preliminary results pertaining to certain lattices.

References

1. Angluin, D.: Inductive inference of formal languages from positive data. Information and Control 45, 117–135 (1980)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
3. Bārzdiņš, J.: Inductive inference of automata, functions and programs. In: Proceedings of the 20th International Congress of Mathematicians, Vancouver, Canada, pp. 455–560 (1974); English translation in American Mathematical Society Translations 2, 109, 107–112 (1977)
4. Birkhoff, G.: Lattice Theory. American Mathematical Society, Providence (1984)
5. Blum, L., Blum, M.: Toward a mathematical theory of inductive inference. Information and Control 28, 125–155 (1975)
6. Case, J., Jain, S., Lange, S., Zeugmann, T.: Incremental concept learning for bounded data mining. Information and Computation 152, 74–110 (1999)
7. de Brecht, M., Kobayashi, M., Tokunaga, H., Yamamoto, A.: Inferability of closed set systems from positive data. In: Washio, T., Satoh, K., Takeda, H., Inokuchi, A. (eds.) JSAI 2006. LNCS (LNAI), vol. 4384, pp. 265–275. Springer, Heidelberg (2007)
8. de la Higuera, C.: Grammatical Inference. Cambridge University Press, Cambridge (2010) (in Press)
9. Fernau, H.: Identification of function distinguishable languages. Theoretical Computer Science 290(3) (2003)
10. Fulk, M.: A Study of Inductive Inference Machines. PhD thesis, SUNY at Buffalo (1985)
11. Gold, E.: Language identification in the limit. Information and Control 10, 447–474 (1967)
12. Heinz, J.: String extension learning (2009), http://phonology.cogsci.udel.edu/~heinz/papers/heinz-sel.pdf
13. Jain, S., Osherson, D., Royer, J., Sharma, A.: Systems that Learn: An Introduction to Learning Theory, 2nd edn. MIT Press, Cambridge (1999)
14. Jantke, K.P.: Monotonic and non-monotonic inductive inference. New Generation Computing 8(4) (1991)
15. Lange, S., Wiehagen, R.: Polynomial time inference of arbitrary pattern languages. New Generation Computing 8, 361–370 (1991)
16. Nation, J.: Notes on lattice theory (2009), http://www.math.hawaii.edu/~jb/books.html
17. Osherson, D., Stob, M., Weinstein, S.: Systems that Learn: An Introduction to Learning Theory for Cognitive and Computer Scientists. MIT Press, Cambridge (1986)
18. Rogers, H.: Theory of Recursive Functions and Effective Computability. McGraw Hill, New York (1967); Reprinted by MIT Press, Cambridge, Massachusetts (1987)
19. Royer, J., Case, J.: Subrecursive Programming Systems: Complexity and Succinctness. In: Research monograph in Progress in Theoretical Computer Science. Birkhäuser, Boston (1994)
20. Weinstein, S.: Private communication at the Workshop on Learnability Theory and Linguistics. University of Western Ontario (1982)
21. Wexler, K., Culicover, P.: Formal Principles of Language Acquisition. MIT Press, Cambridge (1980)
22. Wiehagen, R.: Limes-Erkennung rekursiver Funktionen durch spezielle Strategien. Elektronische Informationsverarbeitung und Kybernetik 12, 93–99 (1976)

The Equivalence Problem of Deterministic Multitape Finite Automata: A New Proof of Solvability Using a Multidimensional Tape

Alexander A. Letichevsky[1], Arsen S. Shoukourian[2], and Samvel K. Shoukourian[3]

[1] National Academy of Sciences of Ukraine, Glushkov Institute of Cybernetics
let@cyfra.net
[2] National Academy of Sciences of Armenia, Institute for Informatics and Automation Problems
arsen.shoukourian@gmail.com
[3] Yerevan State University, IT Educational and Research Center
samshouk@sci.am

Abstract. This publication presents a new proof of solvability for the equivalence problem of deterministic multitape finite automata, based on modeling their behavior via a multidimensional tape. It is shown that for a decision on equivalence of two automata it is necessary and sufficient to consider finite sets of their execution trace words built over the mentioned multidimensional tape.

1 Introduction

Deterministic multitape finite automata and their equivalence problem were introduced by M. O. Rabin and D. Scott in 1959 [6]. The solvability of the equivalence problem for two tape automata was proven by M. Bird in 1973 [1]. The way of solution is based on equivalent transformation of source automata graphs to a special finite commutative diagram, which takes into account commutativity assumptions for symbols of the alphabet used in different tapes. Meantime, the suggested way is not acceptable for the case when the number of tapes is more than two, because, for that case, the suggested transformation may not lead to a finite commutative diagram. In 1993 T. Harju and J. Karhumäki proved the solvability of the equivalence problem for multitape automata without any restriction on the number of tapes [5]. It is based on a purely algebraic technique. Per T. Harju "the main argument of the proof uses an embedding result for ordered groups into division rings".

A new combinatorial proof of solvability is presented in this article. It is similar to the solution suggested by M. Bird, but instead of a transformation of source automata to a commutative diagram, the commutativity assumptions are taken into account via a special multidimensional tape [3][4] used for coding execution traces of source automata. The advantages and disadvantages of the proposed solution imply from its combinatorial nature.

A special representation of an element in a partially commutative semigroup is described in section 2. The new proof of solvability of the equivalence problem of multitape automata is adduced in section 3.

This is the third new application of multidimensional tapes for solving problems in theory of automata. The first applications were in [4][7] for problems open for many years.

2 Partially Commutative Semigroups

If X is an alphabet, then the semigroup of all words in the alphabet X, including the empty word, will be denoted by F_X, and the semigroup of all n-element vectors of words will be denoted by F_X^n.

Let G be a semigroup with a unit, generated by the set of generators $Y = \{y_1, \ldots, y_n\}$. G is called *free partially commutative semigroup*, if it is defined by a finite set of definitive assumptions of type $y_i y_j = y_j y_i$ [2].

Let $K : F_Y \longrightarrow F^n_{\{0,1\}}$ be a homomorphism over the semigroup F_Y which maps words from F_Y to n-element vectors in binary alphabet $\{0, 1\}$ [3]. The homomorphism K over the set of generators of the semigroup F_Y is defined by the equation

$$K(y_i) = (\alpha_{1i}, \ldots, \alpha_{ni}), \quad \text{where} \quad \alpha_{ij} = \begin{cases} 1, \, i = j \\ e, \, y_i y_j = y_j y_i \\ 0, \, y_i y_j \neq y_j y_i \end{cases} \tag{1}$$

At the same time $K(e) = (e, \ldots, e)$.

Lemma 1. *Let y_i, y_j be generators of G, $y_i \neq y_j$, $g_1 = y_i y_j$, $g_2 = y_j y_i$ be elements of G, obtained after applying the operation of the semigroup G to generators y_i and y_j. $g_1 = g_2 \Longleftrightarrow K(y_i y_j) = K(y_j y_i)$* [3].

This statement allows to consider the homomorphism K as a mapping not only over the semigroup of words F_Y, but also over the free partially commutative semigroup G.

A *linear order* $<$ is specified over the set of generators Y [3]. This order *coincides with the order of enumeration of generators* (lexicographical order). The order $<$ is used for sorting the representations of elements of the semigroup G and for defining their canonical representation.

Let $h \in F_Y$. A word in the alphabet $\{0, 1\}$ is called a mask of occurrences of a generator y in the word h, if it is ensued from the substitution of all occurrences of the generator y by 1, and from the substitution of occurrences of all other generators by 0. If the masks are considered not only as words in the alphabet $\{0, 1\}$, but also as binary integers, then it will be possible to compare them. It is assumed that the lexicographical order defines the order for comparing binary representations of generators.

Let g be an element of the semigroup G, and $h, q \in F_Y$ be its representations. The representation h is less than the representation q, $h < q$, if there exists such a generator y that:

1. if $y' < y$ then masks of occurrences of the generator y' in words h and q are equal,
2. the mask of occurrences of the generator y in the word h is less than the mask of occurrences of that generator in the word q.

Lemma 2. *Any two different representations of the same element of the semigroup G are comparable in terms of the order $<$* [3].

It is implied, from Lemma 2, that there exists a minimal representation for every element of G. That representation is called the *canonical form* of the element, induced by the order $<$ over the set of generators [3]. The following property of the canonical representation of an element is true: any two symbols that stand next to each other in the canonical representation are either not commutative or the left one is less (in terms of the order $<$) than the right one.

An equivalence relation ρ over the semigroup F_Y is specified as follows. If w_1 and w_2 are words from F_Y, $w_1, w_2 \in F_Y$, then $w_1 \rho w_2$ if and only if w_1 coincides with w_2 up to the commutativity assumptions.

The relation ρ partitions the semigroup F_Y to disjoint classes. These classes will further be called *classes of commutation.* It is obvious, that a commutation class C_g corresponds to an element g from the semigroup G. An element from C_g, which is the canonical form of g, in turn, will be called the *representative of the commutation class C_g.*

Lemma 3. *Any free partially commutative semigroup of n generators is isomorphic to some sub-semigroup of Cartesian product of n free semigroups with two generators* [3].

Let $Y = \{y_1, y_2\}$, $y_1 y_2 \neq y_2 y_1$. Using the binary coding considered above, i.e. $(1, 0)$ for y_1 and $(0, 1)$ for y_2, the following correspondence between words in F_Y

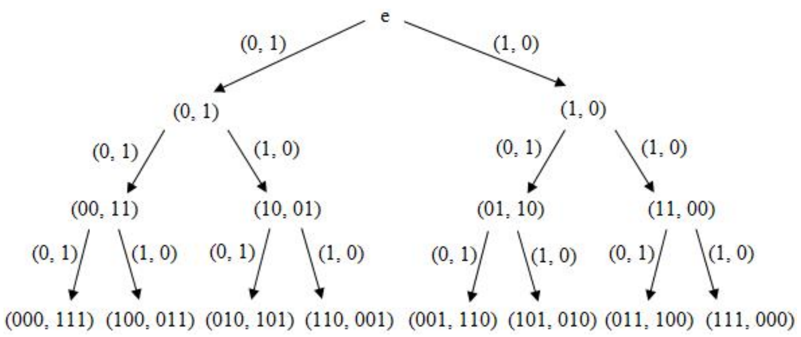


Fig. 1. Correspondence between the words in F_Y and vectors in binary alphabet $\{0, 1\}$

and two-element vectors in binary alphabet $\{0,1\}$ (see Fig. 1) will be obtained: $K(e) = (e,e)$, $K(y_1) = (1,0)$, $K(y_2) = (0,1)$, $K(y_1y_1) = (11,00) = (3,0)$, $K(y_1y_2) = (01,10) = (1,2)$, $K(y_2y_1) = (10,01) = (2,1)$, $K(y_2y_2) = (00,11) = (0,3)$, $\dots$.

As it was mentioned above each element of a vector can be considered as an integer. Thus, a new correspondence between words in F_Y and elements of N^2, where $N = \{0,1,\dots\}$ and (e,e) corresponds to $(0,0)$, is obtained (see Fig. 2). The elements of N^2 which correspond to words in F_Y are located on diagonals marked bold in the Fig. 2.

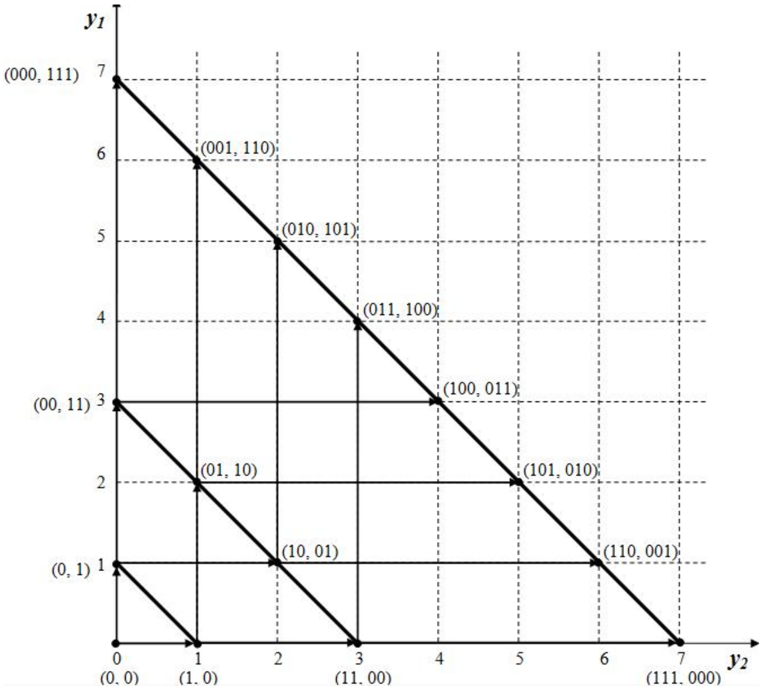


Fig. 2. Correspondence between the words in F_Y and elements of N^2

The correspondence between words in F_Y and elements of N^3, in more complicated case when $Y = \{y_1, y_2, y_3\}$, $y_1y_3 = y_3y_1$, $y_2y_3 = y_3y_2$, $y_1y_2 \neq y_2y_1$ is depicted in the Fig. 3.

Thus, introduction of the described binary coding for elements of a partially commutative semigroup and its justification in Lemmas 1, 2 and 3 allows to consider cells of a multidimensional tape introduced below instead of semigroup elements and, therefore, to avail of the opportunity to compare them as integer vectors when analyzing behaviors of two automata on a given element of the semigroup. Specifically, it becomes possible to measure the distance between

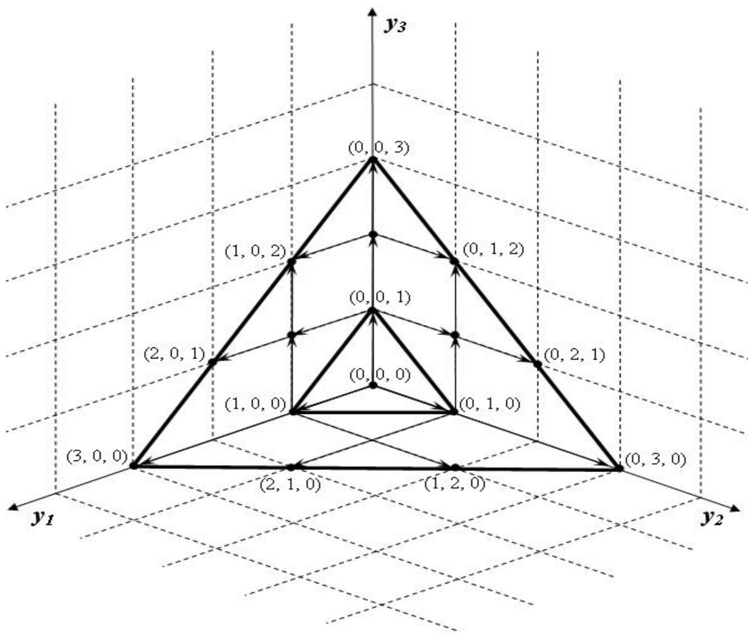


Fig. 3. Correspondence between the words in F_Y and elements of N^3

two automata during their movement on different representations of the same element of the semigroup.

3 A New Proof of Solvability

Some definitions from [3][4], which are necessary for further consideration, will be repeated below.

Let r be a positive integer, $N = \{0, 1, \ldots\}$. The set N^r is called an *r-dimensional tape.* Any element of N^r - $(a_1, \ldots, a_r)$ is called a *cell* of the tape and the numbers $a_1, \ldots, a_r$ are called the *coordinates* of the corresponding cell. The cell $(0, \ldots, 0)$ is the *initial* cell. Let Y be a finite alphabet. Any mapping $N^r \longrightarrow Y$ is called a fill of the tape with the symbols of Y.

It is considered that the alphabet Y is ordered $Y = \{y_1, \ldots, y_n\}$. It is divided into disjoint ordered subsets $Y = \bigcup_{i=1}^{p} Y_i$, $Y_i \bigcap_{i \neq j} Y_j = \varnothing$, preserving the given order for Y: $Y_i = \{y_{f(i)}, \ldots, y_{f(i)+|Y_i|-1}\}$ where $|Y_i|$ is the number of elements in Y_i, $f(1) = 1$, $f(i+1) = f(i) + |Y_i|$ when $i > 1$.

The n-dimensional tape N^n is considered, $n = |Y_1| + \ldots + |Y_p|$. Each $|Y_i|$ dimensions are used for expressing the movement on symbols from Y_i. The subset of dimensions corresponding to Y_i will be denoted by $D(Y_i)$. The position of the first coordinate corresponding to the subset $D(Y_i)$ is denoted by $f(i)$.

Correspondingly, the position of the last coordinate for the subset of dimensions $D(Y_i)$ can be expressed as $f(i) + |Y_i| - 1$. Several definitions are adduced below.

Suppose that $A =< Y, S, \delta, F, s_0 >$ is a deterministic p-tape automaton with the input alphabet Y, has a set $S = S_1 \bigcup \ldots \bigcup S_p$ as the set of states, $S_i \bigcap\limits_{i \neq j} S_j = \emptyset$, δ as the completely defined transition function, F as the set of final states, and s_0 as the initial state. Y_i is the alphabet of the tape i, S_i is the set of states for the head i.

The notion of a predecessor is naturally demonstrated via binary representations with variable length for coordinates of cells adduced in the section 2. The binary representation of the initial cell consists of one digit codes: $(0, \ldots, 0)$. The binary representation of any other cell is built basing on the binary representation of a predecessor for the considered cell, because each successor has only one predecessor here. The length of any coordinate code for the successor is one more than the length of that coordinate code for the predecessor. At the same time, for a given coordinate, only one successor among successors of a given predecessor has the most left bit of the binary representation equal to 1.

A cell $a_1 = (\alpha_{11}, \ldots, \alpha_{1n})$ is called a *predecessor* of a cell $a_2 = (\alpha_{21}, \ldots, \alpha_{2n})$ if and only if there exists a number $j \in \{1, \ldots, p\}$, such that for any k from $\{1, \ldots, f(j) - 1, f(j) + |Y_j|, f(j) + |Y_j| + 1, \ldots, n\}$, $\alpha_{2k} = \alpha_{1k}$ and $\exists l \in \{f(j), \ldots, f(j) + |Y_j| - 1\}$ that $\forall m \in \{f(j), \ldots, f(j) + |Y_j| - 1\}$

1. $\alpha_{2m} = \alpha_{1m}, m \neq l,$
2. $\alpha_{2m} = \alpha_{1m} + (L + 1),\ m = l,$ where $L = \alpha_{11} + \ldots + \alpha_{1n}$.

Two predicates are introduced to represent if one cell is a predecessor of another. $\pi(a_1, a_2)$ is true if and only if the cell a_1 is the predecessor of the cell a_2. Another predicate $\pi_q(a_1, a_2)$ is introduced also. It is true if and only if the predicate $\pi(a_1, a_2)$ is true and a_1, a_2 differ only by the value of the coordinate q as follows $\alpha_{2q} = \alpha_{1q} + (L + 1)$, where $L = \alpha_{11} + \ldots + \alpha_{1n}$.

For a given automaton A a partial mapping $\phi_A : N^n \longrightarrow S$ is introduced:

1. $\phi_A(0, \ldots, 0) = s_0$
2. $\forall a \in N^n \setminus (0, \ldots, 0)\ \exists j \in \{1, \ldots, n\}\ \exists a^{(j)}_{pred}$ that $\pi_j(a^{(j)}_{pred}, a)$ is true and $\phi_A(a^{(j)}_{pred})$ is defined $\Longrightarrow \phi_A(a) = \delta(\phi_A(a^{(j)}_{pred}), y_j)$.
3. $\phi_A(a)$ is considered defined if and only if it is defined according to points 1 and 2 above.

Lemma 4. *For a given automaton A and a given cell $a \in N^n \setminus (0, \ldots, 0)$ if $\phi_A(a)$ is defined then there exists a unique cell a' such that $\phi_A(a')$ is defined and $\pi(a', a)$ is true.*

The graph of the mapping ϕ_A will be named a *set of all execution traces* for the automaton A.

It is evident that there exists such a mapping for any automaton with completely defined transition function.

A part of a set of execution traces, where the sum of coordinates of each cell is less or equal to $k-1$ is called a *trace word* of the length k. The set of all trace words will be denoted further by Ω_A. The set of all cells used in a given trace word ω will be denoted by U_ω.

The part of a trace word ω, where the sum of coordinates of each cell is equal to k, is called the k^{th} *diagonal* of the word ω and is denoted by $d_k(\omega)$. The set of all cells used in a given diagonal d will be denoted by U_d. The length of $d_k(\omega)$ is equal to k+1.

The length of a trace word ω is equal to the number of diagonals it contains and is denoted by $length(\omega)$.

The diagonals which lengths are less by one than the powers of 2 : $2^0 - 1$, $2^1 - 1$, ... will be named *essential*.

The set of all essential diagonals is denoted by $D^{(E)} = \{d_h^{(E)} | h = 0, 1, \ldots\}$, and the length of a given essential diagonal $l_{d_{h+1}^{(E)}} = l_{d_h^{(E)}} + 2^h, l_{d_0^{(E)}} = 0$.

A trace word for a 3-tape automaton is adduced in Figure 4.

For a given trace word ω a path $p = a_{p_1} \ldots a_{p_m}$, $m \geq 1$, $a_{p_j} \in U_\omega$, $j \in \{1, \ldots, m\}$, is defined as a sequence of cells for which $\pi(a_{p_v}, a_{p_{v+1}})$ is true, $v = 1, \ldots, m-1$.

For a given path $p = a_{p_1} \ldots a_{p_m}$ a word $\chi_p = y_{q_1} \ldots y_{q_m - 1}$ in the alphabet Y is called a *characteristics* of the path p if and only if $\forall j \in \{1, \ldots, m\}$, $\pi_{q_j}(a_{p_j}, a_{p_{j+1}})$ is true.

A path $p = a_{p_1} \ldots a_{p_m}$ is called *complete* if $a_{p_1} = (0, \ldots, 0)$. A complete path $p = a_{p_1} \ldots a_{p_m}$ will be called accepted by a given automaton A if $\phi_A(a_{p_m})$ is a final state of A. Basing on considerations in section 2 we can consider the binary

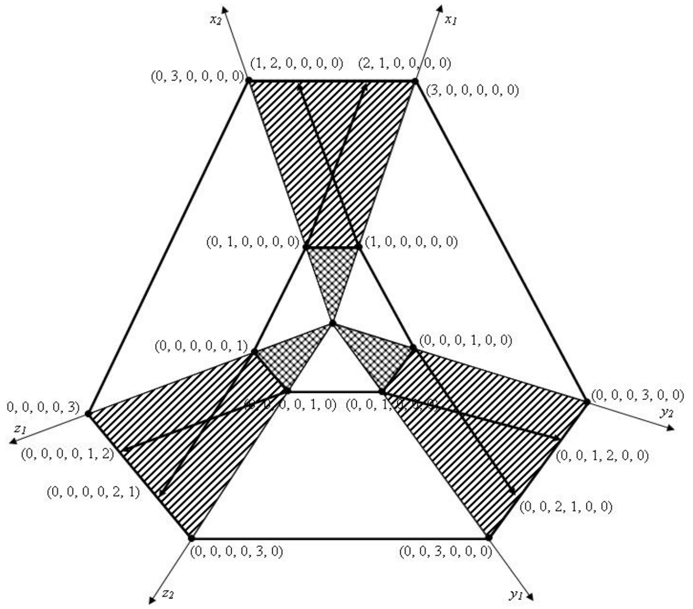


Fig. 4. A trace word for a 3-tape automaton over a two-symbol alphabet for each tape

coding of the characteristics $\chi_p \in F_Y$ for the path p. Its canonical representation coincides with the binary representation of a_{p_m} coordinates. Thus, the binary representation of a_{p_m} coordinates can be considered as a *canonical form* of the complete path $p = a_{p_1} \dots a_{p_m}$.

The set of all accepted paths in a trace word ω for a given automaton A is denoted by $AP_A(\omega)$ and the set of their canonical forms is denoted by $CF_{AP_A(\omega)}$.

Suppose A_1 and A_2 are multitape automata. Suppose also that $k = s^2 \times (2^s - 1)$, $s = |S_1| + |S_2|$, where S_j is the set of states of the automaton A_j, $j = 1, 2$, and ω_j is a trace word, $length(\omega_j) = k$.

Let A be an automaton. It is evident that $\forall d_h^{(E)} \in D^{(E)}\ \exists a \in U_{d_h^{(E)}}$ that $\phi_A(a)$ is defined.

A new mapping $\psi_A : D^{(E)} \longrightarrow 2^{N^n \times S}$ is introduced in the following way: $\psi_A(d_h^{(E)}) = \bigcup\limits_{a \in U_{d_h^{(E)}}} (a, \phi_A(a)),\ \forall d_h^{(E)} \in D^{(E)}$.

A subset of $\psi_A(d_h^{(E)})$ containing all pairs with final states and only them is denoted by $\xi_A(d_h^{(E)})$.

Set of all first components of $\psi_A(d_h^{(E)})$ and $\xi_A(d_h^{(E)})$ will be denoted by $\psi_A^{(1)}(d_h^{(E)})$ and $\xi_A^{(1)}(d_h^{(E)})$ correspondingly. Similarly, the set of all second components will be denoted by $\psi_A^{(2)}(d_h^{(E)})$ and $\xi_A^{(2)}(d_h^{(E)})$.

Lemma 5. *For any diagonal* $d_h^{(E)} \in D^{(E)}$ $\psi_A(d_h^{(E)}) \neq \emptyset$.

Lemma 5 shows that if the transition function of a given automaton is completely defined then any essential diagonal can be reached during the execution of the algorithm.

Lemma 6. *If* $A_1 \sim A_2$ *then for any diagonal* $d_h^{(E)} \in D^{(E)}$
$\xi_{A_1}^{(1)}(d_h^{(E)}) = \xi_{A_2}^{(1)}(d_h^{(E)})$.

Lemma 7 below shows that if two automata are not equivalent then there exists a diagonal (not necessary essential) and a cell on that diagonal such that either both automata reach the cell - one in a final state and the other not in a final state or one automata reaches the cell and the second does not reach it.

Lemma 7. *If* $A_1 \not\sim A_2$ *then there exists a diagonal* d *and a cell* a_j, $a_j \in U_d^{(A_j)}$, $j \in \{1, 2\}$ *and a number* j', $j' \neq j$, $j' \in \{1, 2\}$ *that* $\phi_{A_j}(a_j)$ *is defined,* $\phi_{A_j}(a_j) \in F_{A_j}$ *and* $\phi_{A_{j'}}(a_j)$ *is not defined or* $\phi_{A_{j'}}(a_j)$ *is defined, but* $\phi_{A_{j'}}(a_j) \notin F_{A_{j'}}$.

Lemma 8. *For any number* $m > k$ *there exists a number* $m' < m$, $m'' \leq m$, $m' < m''$, $d_{m'}^{(E)}$, $d_{m''}^{(E)}$ *are essential diagonals* $\psi_{A_1}^{(2)}(d_{m'}^{(E)}) = \psi_{A_1}^{(2)}(d_{m''}^{(E)})$ *and* $\psi_{A_2}^{(2)}(d_{m'}^{(E)}) = \psi_{A_2}^{(2)}(d_{m''}^{(E)})$.

Lemma 9 shows that if there exist two essential diagonals with repeating set of states then it is possible to determine next essential diagonal with the same set of states. The distance between diagonals is not constant, but it is computable.

Lemma 9. *$\forall m' < k$, $m'' \leq k$, $m' < m''$ $d^{(E)}_{m'}$, $d^{(E)}_{m''}$ are essential diagonals and $\psi^{(2)}_A(d^{(E)}_{m'}) = \psi^{(2)}_A(d^{(E)}_{m''})$ then $\psi^{(2)}_A(d^{(E)}_{m''}) = \psi^{(2)}_A(d^{(E)}_{m'''})$ $m''' = m'' + 2^{m''} + \ldots + 2^{m''+(m''-m')-1}$.*

Lemma 10. *Let $p = a_{p_1} \ldots a_{p_m}$, $m > k$ be a complete path for the automaton A. Let also $d^{(E)}_{m_1}, \ldots, d^{(E)}_{m_u}$, $m_1 < \ldots < m_u < m$, $u \geq |S|$ be a set of essential diagonals such that $\psi^{(2)}_A(d^{(E)}_{m_t}) = \psi^{(2)}_A(d^{(E)}_{m_1})$ $t = 1, \ldots, u$. Then there exist numbers m', $m'' \in \{m_1, \ldots, m_u\}$ such that $\phi_A(a_{p_{m'}}) = \phi_A(a_{p_{m''}})$, $a_{p_{m'}} \in U_{d^{(E)}_{m'}}$, $a_{p_{m''}} \in U_{d^{(E)}_{m''}}$.*

Lemma 11 below shows that if the set of states is repeating then it is possible to find essential diagonals with repeating states for each state of the set.

Lemma 11. *Let $d^{(E)}_{m'}$, $d^{(E)}_{m''}$, $m'' > m'$ be essential diagonals, $a_{p_{m'}} \in U_{d^{(E)}_{m'}}$, $a_{p_{m''}} \in U_{d^{(E)}_{m''}}$ $\psi^{(2)}_A(d^{(E)}_{m'}) = \psi^{(2)}_A(d^{(E)}_{m''})$. Then for any $s \in \psi^{(2)}_A(d_{m'})$ there exists a number $m''' \geq m''$ and there exists a cell $a_{p_{m'''}} \in U_{d^{(E)}_{m'''}}$ $\phi_A(a_{p_{m'}}) = \phi_A(a_{p_{m'''}}) = s$.*

If $a_1, a_2 \in N^n$ then the distance between a_1 and a_2 will be denoted by $dist(a_1, a_2)$.
Lemma 12 extends Lemma 11 for the case when there are two automata.

Lemma 12. *Let $p_j = a_{p_{j1}} \ldots a_{p_{jm}}$, $m > k$ be a complete path for the automaton A_j, $j = 1, 2$. Let also $d^{(E)}_{m_1}, \ldots, d^{(E)}_{m_u}$, $m_1 < \ldots < m_u < m$ be a set of diagonals such that $\psi^{(2)}_{A_j}(d^{(E)}_{m_t}) = \psi^{(2)}_{A_j}(d^{(E)}_{m_1})$, $t = 1, \ldots, u$, $u < |S|$, $j = 1, 2$. Then there exist numbers m', $m'' \in \{m_1, \ldots, m_u\}$ such that $\phi_{A_j}(a_{p_{m'}}) = \phi_{A_j}(a_{p_{m''}})$, $a_{p_{jm'}} \in U_{d^{(E)}_{m'}}$, $a_{p_{jm''}} \in U_{d^{(E)}_{m''}}$.*

Lemma 13 shows that if

1) for equivalent automata there are two paths accepted by each automaton with states repeating on same essential diagonals;
2) these paths end in the same cell;
then the distance between cells reached on diagonals with repeating states does not change.

Lemma 13. *Let $p_j = a_{p_{j1}} \ldots a_{p_{jm}}$, $m > k$ be an accepted path for the automaton A_j, $j = 1, 2$, $d^{(E)}_{m'}$, $d^{(E)}_{m''}$, $m' < m'' < m$ be essential diagonals, $a_{p_{jm'}} \in U_{d^{(E)}_{m'}}$, $a_{p_{jm''}} \in U_{d^{(E)}_{m''}}$, $\psi^{(2)}_{A_j}(d^{(E)}_{m'}) = \psi^{(2)}_{A_j}(d^{(E)}_{m''})$, $\phi_{A_j}(a_{p_{jm'}}) = \phi_{A_j}(a_{p_{jm''}})$. Then $A_1 \sim A_2$ and $a_{p_{1m}} = a_{p_{2m}} = a_{p_m}$ implies that $dist(a_{p_{1m'}}, a_{p_{2m'}}) = dist(a_{p_{1m''}}, a_{p_{2m''}})$.*

Proof. Assume that $dist(a_{p_{1m''}}, a_{p_{2m''}}) > dist(a_{p_{1m'}}, a_{p_{2m'}})$. Without additional restrictions it can be assumed that the number of essential diagonals $d^{(E)}_{m''}, \ldots, d^{(E)}_{m}$ containing cells $a_{p_{jm''}}, \ldots, a_{p_{jm}}$ is less than $max\{|S_1|, |S_2|\}$.

Due to the made assumption on difference in cells, paths that start from $a_{p_{1m'}}$ and $a_{p_{2m'}}$ which have the same characteristics as the characteristics of the path, are ending in different cells denoted further by $a'_{p_{1m}}$ and $a'_{p_{2m}}$ correspondingly. $\phi_{A_j}(a'_{p_{jm}})$ belongs to the set of final states of the automaton A_j, $j = 1, 2$. Meantime, $\phi_{A_1}(a'_{p_{2m}})$ and $\phi_{A_2}(a'_{p_{1m}})$ do not belong to the set of final states of A_1 and A_2, correspondingly, due to $A_1 \sim A_2$. Otherwise, returning back to the cell a_{p_m} it is evident that then the automaton A_j, $j = 1, 2$, will have two different accepted paths ending in the same cell which contradicts to the assumption that A_j is deterministic.

Lemma 14. $CF_{AP_{A_1}(\omega_1)} = CF_{AP_{A_2}(\omega_2)} \Longleftrightarrow A_1 \sim A_2$.

Proof. First, it will be shown that $CF_{AP_{A_1}(\omega_1)} = CF_{AP_{A_2}(\omega_2)} \Longrightarrow A_1 \sim A_2$.

Assume, that A_1 is not equivalent to A_2 and there are no paths p_1 and p_2 with the same canonic form in the trace words ω_1 and ω_2 that p_1 is accepted by A_1 and p_2 is not accepted by A_2 or, vice versa, p_2 is accepted by A_2 and p_1 is not accepted by A_1.

Without reducing the assumption and due to Lemma 7 there exists a complete path of length m, $m > k$ $p^{(j)} = a^{(j)}_{p_1} \dots a^{(j)}_{p_m}$, $a^{(1)}_{p_m} = a^{(2)}_{p_m} = a_{p_m}$, for the automaton A_j, $j = 1, 2$ such that $\phi_{A_1}(a_{p_m})$ belongs to the set of final states of the automaton A_1 and $\phi_{A_2}(a_{p_m})$ does not belong to the set of final states of the automaton A_2 or $\phi_{A_2}(a_{p_m})$ is not defined.

The case when $\phi_{A_2}(a_{p_m})$ is defined will be considered at first.

As $m > k$ there exist, according to Lemma 8, essential diagonals with numbers m', m'', $m' < m$, $m'' < m$, $m' < m''$ such that $\psi^{(2)}_{A_1}(d^{(E)}_{m'}) = \psi^{(2)}_{A_1}(d^{(E)}_{m''})$ and $\psi^{(2)}_{A_2}(d^{(E)}_{m'}) = \psi^{(2)}_{A_2}(d^{(E)}_{m''})$.

Thus, per Lemma 11, a path $p^{(j)}$, $j = 1, 2$ can be represented in the following way: $p^{(j)} = a^{(j)}_{p_1} \dots a^{(j)}_{p_{m'}} \dots a^{(j)}_{p_{m''}} \dots a_{p_m}$, where $\phi_{A_1}(a^{(1)}_{p_{m'}}) = \phi_{A_1}(a^{(1)}_{p_{m''}})$ and $\phi_{A_2}(a^{(2)}_{p_{m'}}) = \phi_{A_2}(a^{(2)}_{p_{m''}})$. Without reducing the assumption it can be assumed that $p^{(j)}$ are shortest paths for which A_1 and A_2 are not equivalent.

Let χ_j be a characteristics of the subpath $a^{(j)}_{p_{m''}} \dots a_{p_m}$ and $p^{(j)}_1 = a^{(j)}_{p_{m'}} \dots a^{(j)}_{p}$ is a subpath which has the same characteristics χ_j, $j = 1, 2$.

It is evident that such a subpath exists. Due to Lemma 13 $a^{(1)}_p = a^{(2)}_p = a_p$. Denote by $p^{(j)}_2$ the subpath of the path $p^{(j)}$, starting from a_{p_1} and ending with a predecessor of $a^{(j)}_{m'}$. Consider as a new path the following concatenation of those subpaths $p^{(j)}_{new} = p^{(j)}_2 p^{(j)}_1$. Its length is less than the length of the initial path $p^{(j)}$. Meantime, it implies evidently that $\phi_{A_1}(a_p)$ is a final state of A_1, but $\phi_{A_2}(a_p)$ is not a final state of A_2. Meantime, according to Lemmas 12, 13 there is no other complete path for the automaton A_2 to the cell a_p, for which $\phi_{A_2}(a_p)$ belongs to the set of final states. Thus, we obtain that for the diagonal with the number equal to the length of $p^{(j)}_{new}$ there exists a cell for which one of automata has a state different from final states. If the length of $p^{(j)}_{new}$ is still more than k, similar considerations should be done until the obtained paths have a length not exceeding k. Then having these paths we come to a contradiction with the initial assumption of the Lemma 14.

Now the case when $\phi_{A_2}(a_{p_m})$ is not defined is considered.

Similarly to the considerations above one may obtain that $\phi_{A_2}(a_p)$ is also not defined. Thus, it is obtained that there exists a cell at diagonal with the number equal to the length of $p^{(1)}_{new}$ for which ϕ_{A_1} is defined and its value belongs to the set of final states for A_1 and, at the same time, ϕ_{A_2} is not defined.

If the number of the diagonal is still more than k, similar considerations shall be done until the obtained path has a length not exceeding k. Then, having the path and ϕ_2 not defined at the end cell of the path we come to a contradiction with the assumption of Lemma 14.

As $A_1 \sim A_2 \Longrightarrow CF_{AP_{A_1}(\omega_1)} = CF_{AP_{A_2}(\omega_2)}$ is obviously also true, the Lemma is proved.

Theorem 1. *The equivalence problem of deterministic multitape automata is solvable.*

References

1. Bird, M.: The equivalence problem for deterministic two-tape automata (1973)
2. Clifford, A.H., Preston, G.B.: The algebraic theory of semigroups (1961)
3. Godlevskii, A.B., Letichevskii, A.A., Shukuryan, S.K.: Reducibility of program-scheme functional equivalence on a nondegenerate basis of rank unity to the equivalence of automata with multidimensional tapes (1980)
4. Grigorian, H.A., Shoukourian, S.K.: The equivalence problem of multidimensional multitape automata (2008)
5. Harju, T., Karhumäki, J.: The equivalence problem of multitape finite automata (1991)
6. Rabin, M.O., Scott, D.: Finite automata and their decision problems (1959)
7. Shoukourian, A.S.: Equivalence of regular expressions over a partially commutative alphabet (2009)

Primitive Words Are Unavoidable for Context-Free Languages

Peter Leupold[*]

Fachbereich Elektrotechnik/Informatik
Fachgebiet Theoretische Informatik
Universität Kassel, Kassel, Germany
Peter.Leupold@web.de

Abstract. We introduce the concept of unavoidability of languages with respect to a language class; this means that every language of the given class shares at least some word with the unavoidable language. Several examples of such unavoidabilities are presented. The most interesting one is that the set of primitive words is unavoidable for context-free languages that are not linear.

1 The Language of Primitive Words

Primitivity of words is a very fundamental concept, and has played an important role especially in the Theory of Codes [1] and related aspects of Combinatorics on Words [15,16]. Besides this, primitive words have also received much attention in the field of Formal Languages. In 1991 Dömösi et al. for the first time explicitly raised the problem whether the language of all primitive words, Q, is context-free or not [7]. This problem, which is still unsolved, has been the center of attention around which investigations have revolved. So far, it is only known for smaller language classes like the regular and linear ones, as well as the deterministic and the unambiguous context-free ones that they cannot contain the language of all primitive words [11,2,17]. On the other hand, it is straight-forward to see that the language of primitive words is context-sensitive. Only the relation to the class of context-free languages remains unclear and seems very hard to establish.

Thus partial cases of the question to which language class Q belongs have been investigated. For many types of languages those ones which consist only of primitive words have been characterized [5,4,3,19]. On the other hand, also some complementary work on languages that consist exclusively of non-primitive words has been done. Ito and Katsura gave a characterization of those context-free languages that do not contain any primitive word [14], Dömösi et al. presented a hierarchy of languages that consist only of primitive words [6]. This type of work is the starting point for our investigations.

Rather than using the strict classification by considering only languages that consist either exclusively of primitive or non-primitive words, we introduce the notion of *unavoidability*. This allows a finite "contamination" with undesired words, thus it is a

[*] This work was done in part while Peter Leupold was funded as a post-doctoral fellow by the Japanese Society for the Promotion of Science under grant number P07810; the other part was done while he was funded by the Humboldt Foundation with a Return Fellowship.

somewhat fuzzier notion than that of strict membership in a class. In contrast to the most common language classes, it is based on the languages' contents, i.e. on the words they contain, rather than some abstract property. Formalizing this, we say that a language U avoids another language L if they do not share any words. A language U is then said to be unavoidable in a language class $\mathcal{C}$ if it cannot be avoided by any language in $\mathcal{C}$. A stronger property is strong unavoidability, which is given if U shares infinitely many words with every language in $\mathcal{C}$.

We show that this strong property of unavoidability holds for the language of primitive words with respect to several natural language classes such as the regular languages with infinite root and the non-linear context-free languages. The proof of the latter result in parts repeats earlier work of Ito and Katsura [14]; however, the presentation of their proofs remains slightly obscure in several passages, and therefore we prove our results here in full length.

While this work does not directly advance work on the question whether the language of primitive words is context-free, it does show a new way to possibly approach this problem. It is known that Q is not linear and not unambiguous. If one could show that its complement is unavoidable for example for context-free languages that are ambiguous, this would show that Q cannot belong to that class.

2 Preliminaries

We assume the reader to be familiar with fundamental concepts from Formal Language Theory such as alphabet, word, and language, which can be found in many standard textbooks like the one by Harrison [10]. The length of a finite word w is the number of not necessarily distinct symbols it consists of and is written $|w|$. The i-th symbol we denote by $w[i]$. The notation $w[i \dots j]$ is used to refer to the part of a word starting at the i-th position and ending at the j-th position.

Words together with the operation of concatenation form a free monoid, which is usually denoted by Σ^* for an alphabet Σ. Repeated catenation of a word w with itself is denoted by w^i for natural numbers i. This notation is extended to sets such that for a language L we have $L^i := \{w_0 w_1 \cdots w_i : w_0, w_1, \dots, w_i \in L\}$. In contrast to this $L^{(i)} := \{w^i : w \in L\}$. In both cases we use intuitively understandable notations like $L^{(\geq i)}$ for languages like $\bigcup_{j \geq i} L^{(j)}$.

A word u is a *prefix* of w if there exists an $i \leq |w|$ such that $u = w[1 \dots i]$; if $i < |w|$, then the prefix is called *proper*. The set of all prefixes is $\mathsf{pref}(w)$. Suffixes are the corresponding concept reading from the back of the word to the front and they are denoted by suff. A word w has a positive integer k as a *period*, if for all i, j such that $i \equiv j (\text{mod} k)$ we have $w[i] = w[j]$, if both $w[i]$ and $w[j]$ are defined.

One of the main focuses in what follows will be on a special class of words called primitive, which we now define. A word is *primitive*, iff it is not a non-trivial (i.e. with exponent one) power of any word. Thus u is primitive, if $u = v^k$ implies $u = v$ and $k = 1$; this means that λ is not primitive, because, for example, $\lambda^4 = \lambda$. We denote the language of all primitive words by Q. It is a well-known fact that for every non-empty word w there exists a unique primitive word p such that $w \in p^+$; this primitive word is called the *(primitive) root* of w and we will denote it by $\sqrt{w}$. The unique integer i such that $\sqrt{w^i} = w$ is called the *degree* of w. The notion of root is extended to languages in the canonical way such that $\sqrt{L} := \bigcup_{w \in L} \sqrt{w}$.

Two fundamental results from Formal Language Theory will play a central role in several of our argumentations further down. These are the *Pumping Lemmata* for regular and for context-free languages, which we recall here:

Lemma 1. *For every regular language L there exists an integer k_L such that every word $w \in L$ longer than k_L, has a factorization $w = w_1w_2w_3$ such that $w_2 \neq \lambda$, $|w_1w_2| \leq k_L$ and $w_1w_2^*w_3 \subseteq L$.*

Lemma 2. *For every context-free language L there exists an integer k_L such that every word $w \in L$ longer than k_L, has a factorization $w = w_1w_2w_3w_4w_5$ such that $w_2w_4 \neq \lambda$, $|w_2w_3w_4| \leq k_L$ and $w_1w_2^iw_3w_4^iw_5 \in L$ for all $i \geq 0$.*

Both lemmata provide necessary conditions for languages to belong to the respective classes; however, neither condition is sufficient. This means there are non-regular languages that satisfy Lemma 1 and non-context-free languages that satisfy Lemma 2. Finally, we state a weak form of the Theorem of Fine and Wilf, which we will use further down.

Theorem 3. *If two non-empty words p^i and q^j share a prefix of length $|p| + |q|$, then there exists a word r such that $p, q \in r^+$.*

Finally we recall the two concepts of *boundedness* and *slenderness*. A language L is called bounded if there is a finite number of words $w_1, w_2, \ldots, w_i$ such that $L \subseteq w_1^* \cdot w_2^* \cdots w_i^*$. A language L is called slender, if there is an integer k such that $|L \cap \Sigma^n| \leq k$ for all $n \geq 1$; this means L contains at most k words of any given length.

3 Unavoidable Languages

We want to formalize the following intuitive concept: if a language L shares some words with every language from a given class $\mathcal{C}$, then it is unavoidable in $\mathcal{C}$, because parts of it appear in some sense everywhere. Depending on the size of these parts we define also a strong version of unavoidability.

Definition 4. A language $U \subseteq \Sigma^*$ is *unavoidable* in the language class $\mathcal{C}$, iff $U \cap L \neq \emptyset$ for all infinite languages $L \in \mathcal{C}$. U is *strongly unavoidable*, iff $U \cap L$ is infinite for all infinite languages $L \in \mathcal{C}$.

Notice that this concept is different from unavoidable sets or languages as they are used in Combinatorics of Words [15]; there, a set of words U is unavoidable, if there exists an integer k such that every word longer than k must have a word from U (or a morphic image of such a word) as a factor. Thus unavoidability is an absolute property of languages, not one relative to a language class as in our case. A further difference is that we demand that words of U be elements of all languages in $\mathcal{C}$, and not just that they occur as factors.

As a trivial example, Σ^* is strongly unavoidable for all possible language classes over the alphabet Σ. In fact, any language with finite complement is unavoidable for any class of languages, because these languages will have infinite intersections with all infinite languages. Less trivial examples are harder to come by. Two examples can be derived from the Pumping Lemmata 2 and 1.

Example 5. Let L_{sq} be the language of all words that contain a square. From the two Pumping Lemmata we can see that every infinite regular language has a subset of the form $w_1w_2w_2^+w_3$ and that every infinite context-free language has a subset of the form $\{w_1w_2^iw_3w_4^iw_5 : i \geq 2\}$. Both sets contain only words with squares and are thus infinite subsets of L_{sq}. Thus the latter is strongly unavoidable for regular and context-free languages.

Now we proceed to construct a more elaborate example. For this we first state a simple corollary of the Theorem of Lyndon and Schuetzenberger, see for example the book by Shyr [18].

Lemma 6. *Let p and q be two distinct primitive words. Then there are at most two non-primitive words in each of the languages p^*q and pq^*.*

Obviously, the set of primitive words contained in a language cannot be bigger than the language's root. On the other hand, a language with infinite root can consist of only non-primitive words. For example, we have $\sqrt{Q^{(2)}} = Q$ but $Q^{(2)} \cap Q = \emptyset$. For regular languages, however, this cannot be the case, which will provide us with a first example for a class of languages for which primitive words are unavoidable.

Theorem 7. *The language of primitive words is strongly unavoidable for regular languages with infinite root.*

Proof. Let L be a regular language with infinite root, and let k_L be its corresponding constant from the Pumping Lemma for regular languages, Lemma 1. We will give an infinite sequence of primitive words in L. Since L has infinite root, there is a word $w \in L$, whose root r is longer than k_L. If w is primitive, then it is the first word of our sequence. Otherwise there is a factorization $w = r_1r_2r_3r^m$ with $r_1r_2r_3 = r$ and $m \geq 1$ such that $r_1r_2^nr_3r^m \in L$ for all $n \geq 0$ by the Pumping Lemma. Notice that $r_3r^mr_1$ must be a primitive word and it is longer that the root of r_2. Thus by Lemma 6 the language $r_2^nr_3r^mr_1$ contains infinitely many primitive words. Their cyclic permutations of the form $r_1r_2^nr_3r^m$ are primitive, too, and they are in L. Thus L contains infinitely many primitive words.

If w was primitive, then we can take another one of the infinitely many words in L with roots longer than k_L. In this way we either find infinitely many $w \in L$ that are primitive, or we find one that provides us with an infinite subset of L that is primitive. □

For more details on the class of regular languages with infinite root, for example a characterization by a special type of regular expressions, the reader is referred to an article of Horváth et al. [12].

An almost direct consequence of the definitions, which is worth stating explicitly, is the following.

Lemma 8. *For a class of languages $\mathcal{C}$ that is closed under complement, no co-infinite language $L \in \mathcal{C}$ can be unavoidable.*

Proof. If $L \in \mathcal{C}$ and $\mathcal{C}$ is closed under complement, then also the complement $\overline{L}$ is in $\mathcal{C}$. The fact that L is co-infinite means that $\overline{L}$ is infinite. By the definition $L \cap \overline{L} = \emptyset$. □

Earlier we have already seen that co-finite languages are unavoidable for all classes of languages, which means that all examples involving this type of language are somewhat trivial. Now Lemma 8 shows us that for language classes closed under complement at least inside these classes only trivial examples can be found. For the case of Example 5 this tells us that the language of square-free words must be either non-regular or co-finite. Which of these applies actually depends on the alphabet size: for two letters the complement only contains seven words, for more letters the complement is non-context-free.

4 Non-primitive Words in Context-Free Languages

Non-primitive words are those that can be represented in the form p^n with $n \geq 2$. We start out by looking at a subclass of non-primitive words, namely those that can be represented in the form p^n with $n = 2$ for a primitive word p. In the realm of context-free languages they behave quite differently from those with exponents greater than two; the latter we will then investigate further down.

Lemma 9. *Let $w = w_1w_2w_3w_4w_5 \in \Sigma^+$ with $w_2w_4 \neq \lambda$. If $w_1w_2^iw_3w_4^iw_5 \in Q^{(2)}$ for all $i \geq 0$, then w_2 and w_4 are conjugate. Moreover, there is another factorization $w = (fg^kh)^2$ and an integer k such that $w_1w_2^iw_3w_4^iw_5 = (fg^{i+k}h)^2$; here $|f|, |h| \leq |g|$ unless $|w| = |w_1w_5|$.*

Proof. First we show that $|w_2| = |w_4|$. Let us suppose that contrary to this we have $|w_2| > |w_4|$ and $w_1w_2^iw_3w_4^iw_5 \in Q^{(2)}$ for all $i \geq 0$. Because a word's degree is invariant under cyclic permutation, we also have $w_2^iw_3w_4^iw_5w_1 \in Q^{(2)}$. So there is a primitive word p_i such that $w_2^iw_3w_4^iw_5w_1 = p_i^2$ for every i. Since $|w_2| > |w_4|$ the factor w_2^n is longer than $w_3w_4^iw_5w_1$ for large enough i. Thus p_i is a prefix of $w_2^i = \sqrt{w_2}^{i\cdot \mathsf{deg} w_2}$. Since p is primitive, it cannot be a power of $\sqrt{w_2}$, and thus the second factor p in $w_2^iw_3w_4^iw_5w_1 = p_i^2$ starts with a conjugate of $\sqrt{w_2}$. But since $\sqrt{w_2}$ is primitive all of its conjugates are distinct, and since p cannot start with two distinct words of equal length, $|w_2| > |w_4|$ leads to a contradiction. Supposing $|w_2| < |w_4|$ leads to a similar contradiction via $w_4^iw_5w_1w_2^iw_3 \in Q^{(2)}$. Thus we must have $|w_2| = |w_4|$.

Now we show that w_2 and w_4 are conjugate. Let $|w_3| \leq |w_5w_1|$ and as above $w_2^iw_3w_4^iw_5w_1 = p_i^2$. Then for large enough i the first $|w_4|$ letters of the second factor p_i are within the factor w_4^i, which means that they form a conjugate of w_4. On the other hand, w_2 is obviously a prefix of p_i of the same length, and thus is a conjugate of w_4. For $|w_3| > |w_5w_1|$ analogous reasoning works for the words $w_4^iw_5w_1w_2^iw_3$.

Now let us consider the case $i = 0$, i.e. $w_1w_3w_5 \in Q^{(2)}$ and thus $w_3w_5w_1 \in Q^{(2)}$. If $|w_3| < |w_5w_1|$, then w_3 is a factor of w_5w_1, because it is a prefix of the root of $w_3w_5w_1$.

Therefore there exists a word w' such that $w_3w_5w_1 = w_3w'w_3w' = \sqrt{w_3w_5w_1}^2$, and consequently $w_2^iw_3w_4^iw_5w_1 = w_2^iw_3w_4^iw'w_3w'$. Also the circular permutations $w'w_2^iw_3w_4^iw'w_3$ must all have degree two such that $w'w_2^i = w_4^iw'$. Thus w' must be a suffix of w_2^i and a prefix of w_4^i for long enough i.

Thus if w' is longer than w_2, then w_2 is a suffix of w' and w_4 is a prefix of w'; thus $w'w_2^iw_3w_4^iw'w_3 = (w' \cdot w_2^{-1})w_2^{i+1}w_3w_4^{i+1}(w_4^{-1} \cdot w')w_3$. Here the notation $(u^{-1} \cdot v)$ for two words u and v means deleting a prefix u in v, the same for suffixes. Thus $(u^{-1} \cdot uv) = v$. This process of moving w_2 and w_4 from w' into the factors w_2^{i+1} and w_4^{i+1} can be iterated until the factor resulting from w' is shorter than w_2; let us call this factor w''.

If there were k iteration steps, then $w' = w''w_2^k = w_4^kw''$. Therefore the entire word is factorized $w'w_2^iw_3w_4^iw'w_3 = w''w_2^{k+i}w_3w_4^{k+i}w''w_3$. From $w''w_2^k = w_4^kw''$ we see that w'' is a prefix of w_4 and a suffix of w_2. If w'' is shorter than $\frac{|w_2|}{2}$, then $w''w_2^k = w''(w'''w'')^k$ and $w_4^kw'' = (w''w''')^kw''$ for some word w'''. Since $(w''w''')^kw'' = w''(w'''w'')^k$ we get that $w'w_2^iw_3w_4^iw'w_3 = (w''(w'''w'')^{k+i}w_3)^2$. For w'' longer than $\frac{|w_2|}{2}$ an analogous factorization can be found. If w_3 is longer than w_2, then it is longer than w'' and can be reduced by analogous reasoning. In this way we arrive at a factorization $w_1w_2^iw_3w_4^iw_5 = (fg^{n+k}h)^2$ as stated in the lemma. If $|w_3| > |w_5w_1|$ holds, analogous reasoning works. Finally, if $|w_3| = |w_5w_1|$, obviously $w_3 = w_5w_1$, and consequently $w_2 = w_4$. □

From this result we can almost immediately deduce several results concerning context-free subsets of $Q^{(2)}$. As context-free languages these fulfill the Pumping Lemma and thus also the conditions from Lemma 9.

Theorem 10. *All context-free subsets of $Q^{(2)}$ are finite unions of languages of the form $\{(fg^ih)^2 : i \geq 0\}$; thus they are slender and linear.*

Proof. Let L be a context-free subset of $Q^{(2)}$. By the Pumping Lemma 2, for every word $w \in L$ longer than the constant k_L there exists a factorization $z = w_1w_2w_3w_4w_5$ such that $w_1w_2^iw_3w_4^iw_5 \in L$ for all $i \geq 0$. In this case, this implies $w_1w_2^iw_3w_4^iw_5 \in Q^{(2)}$. Thus Lemma 9 applies, and for such a factorization there is an equivalent one of the form $(fg^kh)^2$. The Pumping Lemma also states that $|w_2w_3w_4| \leq k_L$. Thus we have $|gg| \leq k_L$. and by this fact we obtain bounds on the length of both f and h, because by Lemma 9 $|f|, |h| \leq |g|$ unless $|w_3| = |w_5w_1|$. For the case $|w_3| = |w_5w_1|$, on the other hand, the length of f and h is bounded by the constant from the Pumping Lemma, too, because $2|fh| = |w_3| \leq k_L$ and thus $|w_3w_5w_1| \leq 2k_L$. Summarizing, L is a union of only finitely many languages of the form $(fg^kh)^2$.

It is worth noting that there can be finitely many words to which Lemma 9 does not apply. In the beginning we have supposed that a factorization according to the Pumping Lemma can be found. This is not necessarily true for words shorter than the constant k_L. However, there exist only finitely many such words. These are squares and can thus be put in the form $(fg^kh)^2$ for example by setting $f = \sqrt{w}$. Thus nothing is pumped, the resulting language contains only the word w itself.

Finally, observe that all languages of the form $(fg^kh)^2$ are special cases of what Ilie calls *paired loops* [13]. As a finite union of such paired loops L is slender by Ilie's results. Further, since all of the factors f, g, and h are bounded in length, it is relatively easy to construct linear grammars for all of the paired loops, and a finite union of linear languages is again linear. □

Theorem 11. *For a context-free language it is decidable, whether it is a subset of* $Q^{(2)}$.

Proof. We sketch a decision procedure only very roughly. For a given context-free language, the constant k_L from the Pumping Lemma can effectively be computed. Then the finitely many possible words f, g and h such that $|ghfg| < k_L$ can be listed, and in a straight-forward manner the grammars for the corresponding linear languages $(fg^{n+k}h)^2$ can be computed. These are also bounded since they are subsets of the language $f^*g^*(hf)^*g^*h^*$.

Ginsburg and Spanier [9], see also [8], showed that the inclusion problem is decidable for two context-free languages, one of which is bounded. Therefore we can check for every language $(fg^{n+k}h)^2$ whether it is a subset of L. The union of the subsets is still a context-free language and it is also bounded (by their catenation). Therefore by the result of Ginsburg and Spanier it is also decidable whether L is equal to this union. □

After the languages consisting only of squares of primitive words, we now turn our attention to those languages consisting only of higher powers of primitive words. Well-known results like Lemma 6 that pumping in the style of Pumping Lemmata will produce primitive words in most cases. Lemma 9 has provided us with a special case, where the two pumped factors for context-free pumping are identical up to conjugacy. Thus they do the same change in both halves of a word that is a square.

For words of higher degree, two pumping points cannot do this in general. Therefore there cannot be any infinite context-free subsets of $Q^{(k)}$ for $k \geq 3$. If we take the union of all these sets, $Q^{(\geq 3)}$, then simple context-free subsets exist. For example $(ab)^2(ab)^+$ is such a language. Here $\sqrt{(ab)^2(ab)^+} = \{ab\}$, and pumping that adds multiples of ab will increase a word's degree without leading out of the language. This is the only kind of pumping possible for words of degree three or higher without producing also some primitive words.

Theorem 12. *All context-free subsets of* $Q^{(\geq 3)}$ *are finite unions of languages of the form* $p^j(p^i)^+$ *and a finite set* L_F *for primitive words* p *and integers* i *and* j*; thus they are regular and have finite root.*

Proof. Let L be a context-free subset of $Q^{(\geq 3)}$. If L is finite, then it is regular. Otherwise the Pumping Lemma 2 applies to L. This means every word $w \in L$ longer than k_L, has a factorization $w = w_1w_2w_3w_4w_5$ with the conditions of Lemma 2 such that $w_1w_2^iw_3w_4^iw_5 \in L$ for all $i \geq 0$. As a word's degree is invariant under cyclic permutation, this implies $w_2^iw_3w_4^iw_5w_1 \in Q^{(\geq 3)}$ and $w_4^iw_5w_1w_2^iw_3 \in Q^{(\geq 3)}$. Analogous to the proof of Lemma 9 we can show that $|w_2| = |w_4|$ and w_2 and w_4 are conjugate.

For long enough i the root p_i of $w_2^iw_3w_4^iw_5w_1$ will be a prefix of w_2^i, because $p_i \leq \frac{|w_2^iw_3w_4^iw_5w_1|}{3}$. Actually, we have $|w_2p_i| \leq |w_2^i|$ and thus Theorem 3 applies. This means that w_2 and p_i have a common primitive root, which must be p_i, which is primitive. Therefore w_4 has as root a conjugate of p_i. It is rather easy to see that in a word p^j it does not matter where a factor of length $|p|$ (or a multiple thereof) is pumped, the result is always the same. Thus the pumped factor can be moved left and right arbitrarily. By moving the pumped factors in our case to the right we obtain that $w_1 w_2^i w_3 w_4^i w_5 = w[1 \dots |w_1 w_3 w_5|](w[1 \dots 2|w_2|])^{2i}$, which is a factorization in the form stated in this theorem.

To see that we need only finitely many languages of the form $p^j(p^i)^+$ to obtain L, notice that all p_i above are shorter than $|w_2|$, whose length is bounded by the constant k_L from the Pumping Lemma. Thus there are only finitely many different p, and also i is bounded by $|w_2 w_4|$. Thus finitely many languages $p^j(p^i)^+$ will already generate all of p^+ minus a finite set of short words. Every such language has a singleton root, and thus their union has a finite root, too.

Again, as in the proof of Theorem 10 there are only finitely many short words in L to which no pumping can be applied, and they constitute the finite set L_F. □

So the non-primitive words in a context-free language fall basically into two classes according to their degree: two or more than two. We now show that these two classes can be separated by operations that preserve context-freeness, if a given language contains only finitely many primitive words. This will allow us to conclude that Q is unavoidable for non-linear context-free languages.

Theorem 13. *The language of primitive words is strongly unavoidable for $CF \setminus LIN$.*

Proof. Let $L \in CF$ be an infinite context-free language that contains only finitely many primitive words, which form the set P. We first look at the language $L_{\geq 3} := L \cap Q^{(\geq 3)}$. For every word in $L_{\geq 3}$ that is longer than the Pumping Lemma constant k_L for L we have a pumping factorization according to Lemma 2. As in the proof of Theorem 12, the language generated by such a factorization is of the form $p^j(p^i)^+$ and has root p, otherwise infinitely many primitive words would be generated and would be in L. Since the length of p is bounded by k_L, there are only finitely many p, and $L_{\geq 3}$ is a finite union of languages of the form $p^j(p^i)^+$ and of a finite set, again as in the proof of Theorem 12.

This also means that $L_{\geq 3}$ is regular. Since subtracting a regular and a finite set from a context-free language results in a context-free language, $L_2 := L \setminus (L_{\geq 3} \cup P)$ is context-free. By Theorem 10 it is even slender and linear. This, in turn, means that $L = L_2 \cup L_{\geq 3} \cup P$ is linear. Therefore every non-linear context-free language must contain infinitely many primitive words. □

From the proof we see that for a language containing only finitely many primitive words the three parts $L \cap Q$, $L \cap Q^{(2)}$, and $L \cap Q^{(\geq 3)}$ can be separated from each other without exiting the class of context-free languages. These parts are finite, linear and regular, respectively. We can even say more about them, namely that they are bounded.

Corollary 14. *Every context-free language that contains only finitely many primitive words is bounded.*

Proof. $L \cap Q$ if finite and thus bounded. To $L \cap Q^{(2)}$ Theorem 10 applies, and the union of finitely many languages of the form $\{(fg^ih)^2 : i \geq 0\}$ is bounded. Similarly the languages $p^j(p^i)^+$ from Theorem 12 that characterize $L \cap Q^{(\geq 3)}$ are bounded. □

In some sense the above results mean that context-free subsets of $Q^{(\geq 2)}$ cannot be as complex as the most complex, i.e. the non-linear, context-free languages. Theorem 11 also indicates this by stating that it is decidable whether a given context-free language is a subset of $Q^{(\geq 2)}$. Since $L \cap Q^{(\geq 3)}$ is even regular, one might expect that it is also possible to decide whether a given context-free language is a subset of $Q^{(\geq 2)}$.

Theorem 15. *For a context-free language it is decidable, whether it contains infinitely many primitive words.*

Proof. From the factorizations given in Theorems 10 and 12 and the proof of Theorem 13, which states that a context-free language L that contains only finitely many primitive words is slender. As such, this language can be decomposed into finitely many paired loops according to Ilie [13]. Obviously it can be checked for each paired loop, whether it can be represented in one of the simpler forms given in Theorems 10 and 12 or whether it generates a single primitive word. Every loop of a different form will generate infinitely many primitive words, which can also be seen from results similar to Lemma 6.

Thus we can decide the question of whether L contains infinitely many primitive words by first deciding whether it is slender (otherwise the answer is positive), and then checking the form of the paired loops. □

This decidability result parallels a very related one. Horváth showed that for context-free languages it is decidable, whether their primitive roots are finite [11]. So it is decidable, whether there are infinitely many primitive words in a context-free language, and it is also decidable, whether infinitely many primitive words can be reached by the operation of primitive root from a given context-free language.

5 Conclusion

Our investigations so far have not directly contributed towards a solution of the fundamental question whether the set of all primitive words Q is context-free. However, we have established a new relation between primitive words and the class of context-free languages, more exactly a subset of context-free languages. This subset of non-linear languages is actually one area where theoretically the language of primitive words could be situated, after it has been shown that it is not linear. What we have shown is only that the intersection of Q with every non-linear language is infinite. This does not mean that the entire set itself must belong to that class. However, it does imply that the set of all non-primitive words cannot be context-free. This has been shown before via the Pumping Lemma and results like Lemma 6. Looking at unavoidability of the set of non-primitive words might actually lead to a solution to our central question. Let us take a look at the following set of questions.

Open Problem 1. *Is the set of non-primitive words unavoidable for the set of non-regular, non-linear, non-deterministic context-free languages, or for context-free languages with infinite root?*

A positive answer to any of these question would show that the language Q cannot be context-free, because Q obviously avoids the set of non-primitive words. Unfortunately no results in the line of Lemma 6 are known for non-primitive words, and possibly none can be found. Primitivity seems in some sense easier to construct than non-primitivity, especially the bigger the alphabet gets. For example, if we augment an alphabet Σ with an additional letter c, then all words in $\Sigma^* c \Sigma^*$ are primitive. Thus the best hope for results along these lines are for two-letter alphabet.

While the concept of unavoidability of languages introduced here has so far not helped in resolving the question of context-freeness for primitive words, it seems of some theoretical interest in itself. The work presented here includes several non-trivial examples of unavoidability for rather common language classes. In the case of non-linear context-free languages, their definition has no explicit relation at all to primitivity, in contrast to languages with infinite root. Therefore the result is not really expected and might indicate some deeper connection.

References

1. Berstel, J., Perrin, D.: Theory of Codes. Academic Press, Orlando (1985)
2. Dömösi, P., Horváth, G.: The language of primitive words is not regular: Two simple proofs. Bulletin of the EATCS 87, 191–194 (2005)
3. Dömösi, P., Ito, M., Marcus, S.: Marcus contextual languages consisting of primitive words. Discrete Mathematics 308(21), 4877–4881 (2008)
4. Dömösi, P., Martín-Vide, C., Mitrana, V.: Remarks on sublanguages consisting of primitive words of slender regular and context-free languages. In: Karhumäki, J., Maurer, H., Păun, G., Rozenberg, G. (eds.) Theory Is Forever. LNCS, vol. 3113, pp. 60–67. Springer, Heidelberg (2004)
5. Dömösi, P., Hauschildt, D., Horváth, G., Kudlek, M.: Some results on small context-free grammars generating primitive words. Publicationes Mathematicae Debrecen 54, 667–686 (1999)
6. Dömösi, P., Horváth, G., Ito, M.: A small hierarchy of languages consisting of non-primitive words. Publicationes Mathematicae Debrecen 64(3-4), 261–267 (2004)
7. Dömösi, P., Horváth, S., Ito, M.: On the connection between formal languages and primitive words. Analele Univ. din Oradea, Fasc. Mat., 59–67 (1991)
8. Ginsburg, S.: The Mathematical Theory of Context-free Languages. McGraw-Hill, New York (1966)
9. Ginsburg, S., Spanier, E.H.: Bounded ALGOL-like languages. Trans. Am. Math. Soc. 113, 333–368 (1964)
10. Harrison, M.A.: Introduction to Formal Language Theory. Addison-Wesley, Reading (1978)
11. Horváth, S., Ito, M.: Decidable and undecidable problems of primitive words, regular and context-free languages. Journal of Universal Computer Science 5(9), 532–541 (1999)
12. Horváth, S., Leupold, P., Lischke, G.: Roots and powers of regular languages. In: Ito, M., Toyama, M. (eds.) DLT 2002. LNCS, vol. 2450, pp. 220–230. Springer, Heidelberg (2003)
13. Ilie, L.: On a conjecture about slender context-free languages. Theoretical Computer Science 132(1-2), 427–434 (1994)
14. Ito, M., Katsura, M.: Context-free languages consisting of non-primitive words. Int. Journal of Computer Mathematics 40, 157–167 (1991)

15. Lothaire, M.: Combinatorics on Words. Encyclopedia of Mathematics and Its Applications, vol. 17. Addison-Wesley, Reading (1983)
16. Lothaire, M.: Algebraic Combinatorics on Words. In: Encyclopedia of Mathematics and Its Applications, vol. 90. Cambridge University Press, Cambridge (2002)
17. Petersen, H.: On the language of primitive words. Theoretical Computer Science 161, 141–156 (1996)
18. Shyr, H.: Free Monoids and Languages. Hon Min Book Company, Taichung (1991)
19. Shyr, H., Yu, S.: Non-primitive words in the language p^+q^+. Soochow J. Math. 4 (1994)

Modal Nonassociative Lambek Calculus with Assumptions: Complexity and Context-Freeness

Zhe Lin

Institute of Logic and Cognition, Sun Yat-sen University, Guangzhou, China
Faculty of Mathematics and Computer Science Adam Mickiewicza University, Poznàn, Poland
pennyshaq@gmail.com

Abstract. We prove that the consequence relation of the Nonassociative Lambek Calculus with S4-modalities ($\mathrm{NL}_{\mathrm{S4}}$) is polynomial time decidable and categorial grammars based on $\mathrm{NL}_{\mathrm{S4}}$ with finitely many assumptions generate context-free languages. This extends earlier results of Buszkowski [3] for NL and Plummer [16][17] for a weaker version of $\mathrm{NL}_{\mathrm{S4}}$ without assumptions.

1 Introduction and Preliminaries

Nonassociative Lambek Calculus NL is a type logical system for categorial grammars, introduced in Lambek [10] as a nonassociative version of Syntactic Calculus of Lambek [9]. Both systems are regarded now as basic logics for Type-Logical Grammar (categorial grammar). In a sense, NL is a purely substructural logic, since its sequent system admits no structural rules. Syntactic Calculus, now called the Lambek calculus (L), admits one structural rule (Associativity).

Moortgat [12] studies NL with unary modalities $\Diamond, \Box$ (also several pairs $\Diamond_i, \Box_i$) as a system which enables one to use certain structure postulates in a controlled way. For instance, one can admit a restricted permutation $(\Diamond A) \bullet B = B \bullet (\Diamond A)$ or restricted contraction $(\Diamond A) \bullet (\Diamond A) \geq \Diamond A$ instead of their unrestricted forms $A \bullet B$, $A \bullet A \geq A$. This follows the usage of exponentials !, ? in Linear Logic (Girard [5]).

Pentus [14] proves that categorial grammars based on L, (L-grammars), generate precisely all (ε-free) context-free languages. Analogous results for NL-grammars are due to Buszkowski [1] (the product-free NL) and Kandulski [8]. Jäger [7] provides a new proof for NL, which employs a special form of interpolation (of subtrees of the antecedent tree by single formulae).

Buszkowski [3] refines Jäger's interpolation to prove that the consequence relation of NL is polynomial time decidable and categorial grammar, based on NL with finitely many assumptions, generate context-free languages. Buszkowski and Farulewki [4] extend the method to Full NL, i.e. NL with additives $\wedge, \vee$ with distribution (of $\wedge$ respect to $\vee$, and conversely) and finitely many assumptions, proving the context-freeness.

These results are seemingly in conflict with the known facts that the consequence relation for L is undecidable and categorial grammars based on L with assumptions can generate all recursively enumerable languages (Buszkowski [2][3]). Pentus [15] shows that the provability problem for the pure L is NP-complete.

Plummer [16][17] employs Jäger's method with refinements of Buszkowski [3] to prove the context-freeness of categorial grammars, based on NL with modalities $\Diamond$, $\Box$, satisfying the axioms:

$$\mathrm{T}: \quad A \Rightarrow \Diamond A, \qquad 4: \Diamond\Diamond A \Rightarrow \Diamond A.$$

He also claims (without proof) the polynomial time decidability of the resulting system. A key idea of Plummer's work is to replace the above axioms with some corresponding structural rules.

Here we extend Plummer's result for systems with assumptions. Precisely, we consider the system $\mathrm{NL}_{\mathrm{S4}}$ in the sense of Moortgat [12], which admits T, 4, and

$$\mathrm{K}: \qquad \Diamond(A \bullet B) \Rightarrow \Diamond A \bullet \Diamond B,$$

but our results remain valid for the system with 4, T only ($\mathrm{NL}_{\mathrm{S4}}$ in the sense of Plummer [16]). We prove that the consequence relation for $\mathrm{NL}_{\mathrm{S4}}$ with finitely many assumptions is polynomial time decidable and categorial grammars based on it generate precisely the (ε-free) context-free languages.

Let us recall the sequent system of NL. Formulae (types) are formed out of atomic types p, q, r ... by means of three binary operation symbols $\bullet$ (product), $\backslash$ (right residuation), $/$ (left residuation). Formulae are denoted by A, B, C, ... Formula trees (formula-structures) are recursively defined as follows: (i) every formula is a formula-tree, (ii) if Γ, Δ are formula-trees, then $(\Gamma \circ \Delta)$ is a formula-tree. Sequents are of the form $\Gamma \Rightarrow A$ such that Γ is a formula tree and A is a formula. One admits the axioms:

$$(\mathrm{Id}) \quad A \Rightarrow A$$

and the inference rules

$$(\backslash \mathrm{L})\frac{\Delta \Rightarrow A; \quad \Gamma[B] \Rightarrow C}{\Gamma[\Delta \circ (A\backslash B)] \Rightarrow C} \qquad (\backslash \mathrm{R})\frac{A \circ \Gamma \Rightarrow B}{\Gamma \Rightarrow A\backslash B}$$

$$(/\mathrm{L})\frac{\Gamma[A] \Rightarrow C; \quad \Delta \Rightarrow B}{\Gamma[(A/B) \circ \Delta] \Rightarrow C} \qquad (/\mathrm{R})\frac{\Gamma \circ B \Rightarrow A}{\Gamma \Rightarrow A/B}$$

$$(\bullet \mathrm{L})\frac{\Gamma[A \circ B] \Rightarrow C}{\Gamma[A \bullet B] \Rightarrow C} \qquad (\bullet \mathrm{R})\frac{\Gamma \Rightarrow A; \quad \Delta \Rightarrow B}{\Gamma \circ \Delta \Rightarrow A \bullet B}$$

$$(\mathbf{CUT})\frac{\Delta \Rightarrow A; \quad \Gamma[A] \Rightarrow B}{\Gamma[\Delta] \Rightarrow B}$$

The cut-elimination theorem for NL is proved by Lambek [10]. It yields the decidability and the subformula property of NL. However, the cut-elimination theorem does not hold if we affix new non-logical assumptions of the form

$A \Rightarrow B$. In section 2, we introduce a purely syntactic method based on the cut-elimination theorem in some restricted form to prove a subformula property in some extended form, for systems with finitely many non-logical assumptions. This method is new and ensensially different from the model-theoretic method proposed by Buszkowski [3][4].

We describe the formalism of $\mathrm{NL}_{\mathrm{S4}}$. Formulae (types) are formed out of atomic types p, q, r ... by means of three binary operation symbols $\bullet$, $\backslash$, $/$ and two unary operation symbols $\Diamond$, $\Box$. Formula trees (formula-structures) are recursively defined as follow: (i) every formula is a formula-tree, (ii) if Γ, Δ are formula-trees, then $(\Gamma \circ \Delta)$ is a formula-tree, (iii) if Γ is a formula-tree, then $\langle \Gamma \rangle$ is a formula-tree. Sequents are of the form $\Gamma \Rightarrow A$ such that Γ is a formula tree and A is a formula.

The sequent system of $\mathrm{NL}_{\mathrm{S4}}$ is obtained by extending NL with inference rules for the unary modalities and structural rules corresponding to axioms 4, T, K.

The following are sequent rules for the unary modalities:

$$(\Diamond\mathrm{L})\frac{\Gamma[\langle A \rangle] \Rightarrow B}{\Gamma[\Diamond A] \Rightarrow B} \qquad (\Diamond\mathrm{R})\frac{\Gamma \Rightarrow A}{\langle \Gamma \rangle \Rightarrow \Diamond A}$$

$$(\Box\mathrm{L})\frac{\Gamma[A] \Rightarrow B}{\Gamma[\langle \Box A \rangle] \Rightarrow B} \qquad (\Box\mathrm{R})\frac{\langle \Gamma \rangle \Rightarrow A}{\Gamma \Rightarrow \Box A}$$

The following are structural rules corresponding to axioms 4, T, K:

$$(4)\frac{\Gamma[\langle \Delta \rangle] \Rightarrow A}{\Gamma[\langle\langle \Delta \rangle\rangle] \Rightarrow A} \qquad (\mathrm{T})\frac{\Gamma[\langle \Delta \rangle] \Rightarrow A}{\Gamma[\Delta] \Rightarrow A} \qquad (\mathrm{K})\frac{\Gamma[\langle \Delta_1 \rangle \circ \langle \Delta_2 \rangle] \Rightarrow A}{\Gamma[\langle \Delta_1 \circ \Delta_2 \rangle] \Rightarrow A}.$$

Our interest in modal postulates and non-logical assumptions can be motivated in various way. First, in linguistic practice, different phenomena may require different sets of structure postulates. For instance, let us consider an NP (noun phrase) such as "the man who Mary met today ", (an example from Versmissen [18]). In Lambek notation, there is no suitable type assigned to the relative pronoun "who" in this noun phrase. The solution proposed by Morrill [13] is to assign to "who" the single type $(N\backslash N)/(S/\Box NP)$. One also assigns NP/N to "the", N to "man", NP to "Mary", $N\backslash(S/NP)$ to "met", and $S\backslash S$ to "today". The sequent $NP/N \circ N \circ (N\backslash N)/(S/\Box NP) \circ NP \circ NP\backslash(S/NP) \circ S\backslash S \Rightarrow NP$, corresponding to the noun phrase "the man who Mary met today" is derivable in systems enriched with postulate T and $\Box B \bullet A \Rightarrow A \bullet \Box B$. More examples can be found in Morrill [13]. Second, there are many evidences for the usefulness of non-logical assumptions in linguistics. For example, Lambek [11] uses axioms of the form $\pi_i \rightarrow \pi$ to express the inclusion of the class of pronouns in $i - th$ Person in the class of pronouns. Further, in the Lambek calculus we can not transform $S\backslash(S/S)$ (the type of sentence conjunction) to $VP\backslash(VP/VP)$ (the type of verb phrase conjunction), however, we can add the sequent $S\backslash(S/S) \rightarrow VP\backslash(VP/VP)$ as an assumption.

2 $\mathbf{NL_{S4}}$ Enriched with Non-logical Assumptions

We assume that non-logical assumptions are of the form $A \Rightarrow B$. For a set Φ of formulae $A \Rightarrow B$, $\mathrm{NL_{S4}}(\Phi)$ denotes the system of $\mathrm{NL_{S4}}$ with all formulae from Φ as assumptions.

Usually, if we can prove the cut-elimination theorem for a system then we immediately get the subformula property: all formulae in a cut-free derivation are subformulae of the endsequent formulae. (CUT) is a legal rule in $\mathrm{NL_{S4}}(\Phi)$, and the cut-elimination theorem does not hold for $\mathrm{NL_{S4}}(\Phi)$. Hence we can not obtain the standard subformula property for $\mathrm{NL_{S4}}(\Phi)$. Here we consider *the extended subformula property* (see [3]). We introduce a restricted cut rule, Φ-*restricted cut*, where Φ is the set of assumptions.

By Φ-*restricted cut*, we mean the following rules:

$$(\Phi - \mathrm{CUT}) \quad \frac{\Gamma_2 \Rightarrow A \qquad \Gamma_1[B] \Rightarrow C}{\Gamma_1[\Gamma_2] \Rightarrow C}$$

where $A \Rightarrow B$ is an assumption in Φ.

We describe another Genzten style presentation of $\mathrm{NL_{S4}}(\Phi)$, denoted by $\mathrm{NL^r_{S4}}(\Phi)$: Axioms of $\mathrm{NL^r_{S4}}(\Phi)$ are (Id) $A \Rightarrow A$. The inference rules of $\mathrm{NL^r_{S4}}(\Phi)$ are simply the rules of $\mathrm{NL_{S4}}(\Phi)$ together with the Φ-*restricted cut* given above. By $\vdash_S \Gamma \Rightarrow A$, we denote: the sequent $\Gamma \Rightarrow A$ is provable in system S.

Lemma 1. *If $A \Rightarrow B \in \Phi$ then $\vdash_{\mathrm{NL^r_{S4}}(\Phi)} A \Rightarrow B$.*

Proof: Assume $A \Rightarrow B \in \Phi$. We apply Φ-*restricted cut* to axioms $A \Rightarrow A$ and $B \Rightarrow B$ and get $A \Rightarrow B$. Hence $\vdash_{\mathrm{NL^r_{S4}}(\Phi)} A \Rightarrow B$. □

We provide a proof of the cut-elimination theorem for $\mathrm{NL^r_{S4}}(\Phi)$.

Theorem 2. *Every sequent which is provable in* $\mathrm{NL^r_{S4}}(\Phi)$ *can be proved also in* $\mathrm{NL^r_{S4}}(\Phi)$ *without* (CUT).

Proof: We must prove: if both premises of (CUT) are provable in $\mathrm{NL^r_{S4}}(\Phi)$ without (CUT), then the conclusion of (CUT) is provable in $\mathrm{NL^r_{S4}}(\Phi)$ without (CUT). The proof can be arranged as follows.

We apply induction (*i*): on $\mathcal{D}$(A), the complexity of (CUT) formula A, i.e. the total number of occurrences of symbols in A. For each case, we apply induction (*ii*): on $\mathcal{R}$(CUT), the rank of (CUT), i.e. the total number of sequents appearing in the proofs of both premises of (CUT).

Let us consider one case $A = \Diamond A'$. Others can be treated similarly. We write the premises of (CUT) as $\Gamma_2 \Rightarrow A$ and $\Gamma_1[A] \Rightarrow B$ obtained by rules $\mathrm{R_i}$ (i$= 1, 2$), respectively. We switch on induction (*ii*). Three subcases arise.

1. $\Gamma_2 \Rightarrow A$ or $\Gamma_1[A] \Rightarrow B$ is an axiom (Id). If $\Gamma_2 \Rightarrow A$ is a axiom then $\Gamma_2 = A$, and $\Gamma_2[A] \Rightarrow B$ equals $\Gamma_1[\Gamma_2] \Rightarrow B$. If $\Gamma_1[A] \Rightarrow B$ is a axiom then $\Gamma_2 \Rightarrow A$ equals $\Gamma_1[\Gamma_2] \Rightarrow B$.
2. $\mathrm{R}_1 \neq \Diamond\mathrm{R}$ or $\mathrm{R}_2 \neq \Diamond\mathrm{L}$. The thesis follows from the induction hypothesis (*ii*). We show two typical cases. Others can be treated similarly.

2a Let $\Gamma_2 \Rightarrow A$ arise by *Φ-restricted cut* with premises $\Gamma_2 \Rightarrow C$ and $D \Rightarrow A$. We replace

$$\dfrac{\dfrac{\Gamma_2 \Rightarrow C \quad D \Rightarrow A}{\Gamma_2 \Rightarrow A}\,(\Phi - \mathrm{CUT}) \qquad \dfrac{\dots}{\Gamma_1[A] \Rightarrow B}}{\Gamma_1[\Gamma_2] \Rightarrow B}\,(\mathrm{CUT}).$$

where $C \Rightarrow D$ is an assumption, by

$$\dfrac{\dfrac{\dots}{\Gamma_2 \Rightarrow C} \qquad \dfrac{D \Rightarrow A \quad \Gamma[A] \Rightarrow B}{\Gamma_1[D] \Rightarrow B}\,(\mathrm{CUT}')}{\Gamma_1[\Gamma_2] \Rightarrow B}\,(\Phi - \mathrm{CUT})$$

where $C \Rightarrow D$ is an assumption. Clearly $\mathcal{R}(\mathrm{CUT}') < \mathcal{R}(\mathrm{CUT})$. Hence $\Gamma_1[D] \Rightarrow B$ is provable without (CUT), by the hypothesis of induction (*ii*). Then, Φ-restricted cut yields $\Gamma_1[\Gamma_2] \Rightarrow B$. $\Gamma_1[\Gamma_2] \Rightarrow B$

2b Let $\Gamma_1[A] \Rightarrow B$ arise by Φ-*restricted cut*. Similarly, we can first apply (CUT) to $\Gamma \Rightarrow A$ and the premise of $\Gamma_1[A] \Rightarrow C$ which contains cut formula A. After that, we apply Φ-restricted cut to the conclusion of the new (CUT) and the other premise of $\Gamma_1[A] \Rightarrow B$. The thesis follows from the induction hypothesis (*ii*).

3. $\mathrm{R}_1 = \Diamond\mathrm{R}$ and $\mathrm{R}_2 = \Diamond\mathrm{L}$. We replace

$$\dfrac{\dfrac{\Gamma_2' \Rightarrow A'}{\langle\Gamma_2\rangle \Rightarrow \Diamond A'}\,(\Diamond\mathrm{R}) \qquad \dfrac{\Gamma_1[\langle A'\rangle] \Rightarrow B}{\Gamma_1[\Diamond A'] \Rightarrow B}\,(\Diamond\mathrm{L})}{\Gamma_1[\Gamma_2] \Rightarrow B}\,(\mathrm{CUT})$$

where $\langle\Gamma_2'\rangle = \Gamma_2$, by

$$\dfrac{\Gamma_2' \Rightarrow A' \quad \Gamma_1[\langle A'\rangle] \Rightarrow B}{\Gamma_1[\langle\Gamma_2'\rangle] \Rightarrow B}\,(\mathrm{CUT}')$$

where $\langle\Gamma_2'\rangle = \Gamma_2$. Since $\mathcal{D}(\mathrm{A}') < \mathcal{D}(\mathrm{A})$ then $\Gamma_1[\Gamma_2] \Rightarrow B$ is provable in $\mathrm{NL}^{\mathrm{r}}_{\mathrm{S4}}(\Phi)$ without (CUT) by the hypothesis of induction (*i*). □

Let π be a cut free proof in $\mathrm{NL}^{\mathrm{r}}_{\mathrm{S4}}(\Phi)$. By π^+, we mean a proof obtained from π by replacing all occurrences of Φ-*restricted cut* by two applications of (CUT) as follows:

$$\dfrac{\dfrac{\Gamma_2 \Rightarrow A \quad A \Rightarrow B}{\Gamma_2 \Rightarrow B}\,(\mathrm{CUT}) \qquad \dfrac{\dots}{\Gamma_1[B] \Rightarrow C}}{\Gamma_1[\Gamma_2] \Rightarrow C}\,(\mathrm{CUT})$$

where $A \Rightarrow B$ is a nonlogical assumption in Φ. Obviously, π^+ is a proof in $\mathrm{NL}_{\mathrm{S4}}(\Phi)$. Hence we get the following corollary.

Corollary 3. *For any sequent $\Gamma \Rightarrow A$ provable in* $\mathrm{NL}_{\mathrm{S4}}(\Phi)$*, there exists a proof of $\Gamma \Rightarrow A$ such that all formulae appearing in the proof are subformulae of formulae in Φ or $\Gamma \Rightarrow A$.*

Proof: Follows from Lemma 1, Theorem 2, and the construction of π^+ given above. □

Let T be a finite set of formulae, closed under subformulae which contains all formulae appearing in Φ. By a T-sequent, we mean a sequent $\Gamma \Rightarrow A$ such that all formulae appearing in Γ and A belong to T.

Corollary 4. *For any T-sequent $\Gamma \Rightarrow A$ provable in $\mathrm{NL}_{\mathrm{S4}}(\Phi)$, there is a proof of $\Gamma \Rightarrow A$ in $\mathrm{NL}_{\mathrm{S4}}(\Phi)$ such that all sequents appearing in this proof are T-sequents.*

Proof: Immediate from Corollary 3. □

3 Main Results

Here after, we assume that Φ is finite. T denotes a finite set of formulae containing all formulae in Φ and closed under subformulae. Let $T_{\Box} = \{\Box A | A \in T\}$, $T^{\Box} = T \cup T_{\Box}$, $T_{\Diamond} = \{\Diamond A | A \in T^{\Box}\}$ and $T^{\Diamond} = T^{\Box} \cup T_{\Diamond}$. A sequent is said to be *basic* if it is a $T^{\Diamond}$-sequent of the form $A \circ B \Rightarrow C$, $A \Rightarrow B$, or $\langle A \rangle \Rightarrow B$. We describe an effective procedure producing all *basic* sequents provable in $\mathrm{NL}_{\mathrm{S4}}(\Phi)$.

Let S_0 consist of all $T^{\Diamond}$-sequents from Φ, all $T^{\Diamond}$-sequents of the form (Id), and all $T^{\Diamond}$-sequents of the form:

$$\langle A \rangle \Rightarrow \Diamond A, \quad \langle \Diamond A \rangle \Rightarrow \Diamond A, \quad \langle \Box A \rangle \Rightarrow A.$$

$$A \circ (A \backslash B) \Rightarrow B, \quad (A/B) \circ B \Rightarrow A, \quad A \circ B \Rightarrow A \bullet B.$$

Assume S_n has already been defined. S_{n+1} is S_n enriched with all sequents arising by the following rules:

(R1) if $(\langle A \rangle \Rightarrow B) \in S_n$ and $\Diamond A \in T^{\Diamond}$ then $(\Diamond A \Rightarrow B) \in S_{n+1}$,

(R2) if $(\langle A \rangle \Rightarrow B) \in S_n$ and $\Box B \in T^{\Diamond}$ then $(A \Rightarrow \Box B) \in S_{n+1}$,

(R3) if $(\Diamond A \circ \Diamond B \Rightarrow C) \in S_n$ and $C \in T$ then $(A \circ B \Rightarrow \Box C) \in S_{n+1}$,

(R4) if $(\langle A \rangle \Rightarrow B) \in S_n$ then $(A \Rightarrow B) \in S_{n+1}$,

(R5) if $(A \circ B \Rightarrow C) \in S_n$ and $A \bullet B \in T^{\Diamond}$ then $(A \bullet B \Rightarrow C) \in S_{n+1}$,

(R6) if $(A \circ B \Rightarrow C) \in S_n$ and $(A \backslash C) \in T^{\Diamond}$ then $(B \Rightarrow A \backslash C) \in S_{n+1}$,

(R7) if $(A \circ B \Rightarrow C) \in S_n$ and $(C/B) \in T^{\Diamond}$ then $(A \Rightarrow C/B) \in S_{n+1}$,

(R8) if $(A \Rightarrow B) \in S_n$ and $(\langle B \rangle \Rightarrow C) \in S_n$ then $(\langle A \rangle \Rightarrow C) \in S_{n+1}$,

(R9) if $(A \Rightarrow B) \in S_n$ and $(D \circ B \Rightarrow C) \in S_n$ then $(D \circ A \Rightarrow C) \in S_{n+1}$,

(R10) if $(A \Rightarrow B) \in S_n$ and $(B \circ D \Rightarrow C) \in S_n$ then $(A \circ D \Rightarrow C) \in S_{n+1}$,

(R11) if $(\Gamma \Rightarrow B) \in S_n$ and $(B \Rightarrow C) \in S_n$ then $(\Gamma \Rightarrow C) \in S_{n+1}$, Obviously, $S_n \subseteq S_{n+1}$, for all $n \geq 0$. For any $n \geq 0$, S_n is a finite set of *basic* sequents. $S^{T^\Diamond}$ is defined as the union of all S_n. Due to the definition of *basic* sequents, there are only finitely many *basic* sequents. Since $S^{T^\Diamond}$ is a set of *basic* sequents, hence it must be finite. This yields: there exists $k \geq 0$ such that $S_k = S_{k+1}$ and $S^{T^\Diamond} = S_k$. $S^{T^\Diamond}$ is closed under rules (R1)-(R11). The rules (R1), (R2), (R3), (R4), (R5), (R6), (R7) are ($\Diamond$L), ($\Box$R), (K), (T), ($\bullet$L), (\R), (/R) restricted to basic sequents, and (R8)-(R11) in fact describe the closure of basic sequents under (CUT). (R5)-(R7) and (R9)-(R11) are the same as in [3].

Lemma 5. *$S^{T^\Diamond}$ can be constructed in polynomial time.*

Proof: Let n denote the cardinality of $T^\Diamond$. The total number of *basic* sequents of the form $A \Rightarrow B$, $\langle A \rangle \Rightarrow B$, and $A \circ B \Rightarrow C$ are no more than n^2, n^2, and n^3 respectively. Therefore there are at most $m = n^3 + 2 \times n^2$ *basic* sequents. Hence we can construct S_0 in time $O(n^3)$. The construction of S_{n+1} from S_n requires at most $6 \times (m^2 \cdot n) + m^2 + 6 \times m^3$ steps. It follows that the time of this construction of S_{n+1} is $O(m^3)$. Since the least k satisfying $S^{T^\Diamond} = S_k$ does not exceed m. Thus we can construct $S^{T^\Diamond}$ in polynomial time, in time $O(m^4)$. □

By $S(T^\Diamond)$, we denote the system whose axioms are all sequents from $S^{T^\Diamond}$ and whose only inference rule is (CUT). Clearly, every proof in $S(T^\Diamond)$ consists of $T^\Diamond$-sequents. By $\vdash_{S(T^\Diamond)} \Gamma \Rightarrow A$ we denote: $\Gamma \Rightarrow A$ is provable in $S(T^\Diamond)$.

Lemma 6. *Every basic sequent provable in $S(T^\Diamond)$ belongs to $S^{T^\Diamond}$.*

Proof: We proceed by induction on the length of its proof in $S(T^\Diamond)$. For the base case, the claim is trivial. For the inductive case, we assume that s is a *basic* sequent provable in $S(T^\Diamond)$ such that s is obtained from premises s_1 and s_2 by (CUT). Since s is a *basic* sequent, clearly, s_1 and s_2 must be *basic* sequents. By the induction hypothesis, s_1 and s_2 belong to $S(T^\Diamond)$. Hence s belongs to $S^{T^\Diamond}$, by (R8)-(R11). □

We prove two interpolation lemmas for $S(T^\Diamond)$.

Lemma 7. *If $\vdash_{S(T^\Diamond)} \Gamma[\Delta] \Rightarrow A$ then there exists $D \in T^\Diamond$ such that $\vdash_{S(T^\Diamond)} \Delta \Rightarrow D$ and $\vdash_{S(T^\Diamond)} \Gamma[D] \Rightarrow A$.*

Proof: We proceed by induction on the proofs in $S(T^\Diamond)$.

Base case: $\Gamma[\Delta] \Rightarrow A$ belongs to $S^{T\Diamond}$. We consider three subcases. First, if $\Gamma = \Delta = B$ then $D = A$ and the claim stands. Second, if $\Gamma = \langle B \rangle$, $\Delta = B$ *or* $\Delta = \langle B \rangle$ then $D = B$ or $D = A$, respectively. Third, if $\Gamma = B \circ C$, and either $\Delta = B$, or $\Delta = C$, then $D = B$ or $D = C$, respectively.

Inductive case: Assume $\Gamma[\Delta] \Rightarrow A$ is the conclusion of (CUT) whose both premises are $\Delta' \Rightarrow B$ and $\Gamma'[B] \Rightarrow A$ such that $\Gamma[\Delta] = \Gamma'[\Delta']$. Then three cases arise.

1. Δ' is a substructure of Δ. Assume $\Delta = \Delta''[\Delta']$, $\Gamma'[B] = \Gamma[\Delta''[B]]$. Hence there exists $D \in T^{\Diamond}$ satisfying $\vdash_{S(T^{\Diamond})} \Delta''[B] \Rightarrow D$ and $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow A$ by the induction hypothesis. We have $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow D$ from $\vdash_{S(T^{\Diamond})} \Delta' \Rightarrow B$ and $\vdash_{S(T^{\Diamond})} \Delta''[B] \Rightarrow D$, by (CUT).
2. Δ is a substructure of Δ'. Assume $\Delta' = \Delta''[\Delta]$ and $\Gamma[B] = \Gamma'[\Delta''[B]]$. By the induction hypothesis, it is easy to obtain $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow D$ and $\vdash_{S(T^{\Diamond})} \Delta''[D] \Rightarrow B$ for some $D \in T^{\diamond}$, which yields $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow A$ by (CUT).
3. Δ and Δ' do not overlap. Hence $\Gamma'[B]$ must contains Δ. Assume $\Gamma'[B] = \Gamma[B, \Delta]$. By the induction hypothesis, there exists a $D \in T^{\diamond}$ such that $\vdash_{S(T^{\Diamond})} \Gamma'[B, D] \Rightarrow A$ and $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow D$. By (CUT), $\vdash_{S(T^{\Diamond})} \Gamma'[\Delta', D] \Rightarrow A$, which means $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow A$. □

Lemma 8. *If $\vdash_{S(T^{\Diamond})} \Gamma[\langle\Delta\rangle] \Rightarrow A$, then there exists $\Diamond D \in T^{\Diamond}$ such that $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow \Diamond D$ and $\vdash_{S(T^{\Diamond})} \Gamma[\Diamond D] \Rightarrow A$.*

Proof: Assume $\vdash_{S(T^{\Diamond})} \Gamma[\langle\Delta\rangle] \Rightarrow A$. By Lemma 7, there exists $D \in T^{\diamond}$ such that $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow A$ and $\vdash_{S(T^{\Diamond})} \langle\Delta\rangle \Rightarrow D$. Again by Lemma 7, we get $\vdash_{S(T^{\Diamond})} \langle D'\rangle \Rightarrow D$ and $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow D'$, for some $D' \in T^{\Diamond}$. We consider two possibilities.

If $D' \in T^{\Box}$, then $\Diamond D' \in T^{\Diamond}$. We get $\vdash_{S(T^{\Diamond})} \Diamond D' \Rightarrow D$, by Lemma 3.9 and (R1). Since $\langle D'\rangle \Rightarrow \Diamond D'$ and $\langle\Diamond D'\rangle \Rightarrow \Diamond D'$ belong to $S^{T^{\Diamond}}$, we get $\langle\langle D'\rangle\rangle \Rightarrow \Diamond D' \in S^{T^{\Diamond}}$. Hence by two applications of (R4), $D' \Rightarrow \Diamond D' \in S^{T^{\Diamond}}$, which yields $\vdash_{S(T^{\Diamond})} D' \Rightarrow \Diamond D'$. By applying (CUT) to $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow A$ and $\vdash_{S(T^{\Diamond})} \Diamond D' \Rightarrow D$, we get $\vdash_{S(T^{\Diamond})} \Gamma[\Diamond D'] \Rightarrow A$. Again $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow D'$ and $\vdash_{S(T^{\Diamond})} D' \Rightarrow \Diamond D'$, so we get $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow \Diamond D'$. Therefore the claim holds.

If D' does not belong to $T^{\Box}$, then $D' = \Diamond D^*$ for some $D^* \in T^{\Box}$. Hence $\vdash_{S(T^{\Diamond})} \langle\Diamond D^*\rangle \Rightarrow D$ and $\vdash_{S(T^{\Diamond})} \Delta \Rightarrow \Diamond D^*$. Due to Lemma 6, $\langle\Diamond D^*\rangle \Rightarrow D$ belongs to $S^{T^{\Diamond}}$. It yields: $\Diamond D^* \Rightarrow D$ belongs to $S^{(T^{\Diamond})}$, by (R4). Hence $\vdash_{S(T^{\Diamond})} \Diamond D^* \Rightarrow D$. Then, by (CUT), $\vdash_{S(T^{\Diamond})} \Gamma[\Diamond D^*] \Rightarrow A$. Therefore the claim stands. □

For any $T^{\Diamond}$-sequent $\Gamma \Rightarrow A$, by $\Gamma \Rightarrow_{T^{\diamond}} A$ we mean: $\Gamma \Rightarrow A$ has a proof in $\mathrm{NL_{S4}}(\Phi)$ consisting of $T^{\Diamond}$-sequents only.

Lemma 9. *For any $T^{\Diamond}$-sequent $\Gamma \Rightarrow A$, $\Gamma \Rightarrow_{T^{\diamond}} A$ iff $\vdash_{S(T^{\Diamond})} \Gamma \Rightarrow A$.*

Proof: The 'if' part is easy. Notice that all $T^{\Diamond}$-seuqents which are axioms of $\mathrm{NL_{S4}}(\Phi)$ belong to $S^{T^{\Diamond}}$. The 'only if' part is proved by showing that all inference rules of $\mathrm{NL_{S4}}(\Phi)$, restricted to $T^{\Diamond}$-sequents, are admissible in $S(T^{\Diamond})$. The rules (CUT), (\L), (/L) (\R) (/R) (•L) (•R) are settled by Buszkowski[3]. Here we provide full arguments for ($\Diamond$L), ($\Diamond$R), ($\Box$L), ($\Box$R), (4), (T), (K).

1. For $(\Diamond L)$, assume $\vdash_{S(T^{\Diamond})} \Gamma[\langle A\rangle] \Rightarrow B$ and $\Diamond A \in T^{\Diamond}$. By Lemma 7, there exists $D \in T^{\Diamond}$ such that $\vdash_{S(T^{\Diamond})} \Gamma[D] \Rightarrow B$ and $\vdash_{S(T^{\Diamond})} \langle A\rangle \Rightarrow D$. Since $\vdash_{S(T^{\Diamond})} \langle A\rangle \Rightarrow D$ is a basic sequent, then by Lemma 6, $\langle A\rangle \Rightarrow D \in S^{T^{\Diamond}}$. By (R1), we get $\Diamond A \Rightarrow D \in S^{T^{\Diamond}}$, which yields $\vdash_{S(T^{\Diamond})} \Diamond A \Rightarrow D$. Hence $\vdash_{S(T^{\Diamond})} \Gamma[\Diamond A] \Rightarrow B$, by(CUT).

2. For $(\Diamond R)$, assume $\vdash_{S(T^\Diamond)} \varGamma \Rightarrow A$ and $\Diamond A \in T^\Diamond$. Since $\vdash_{S(T^\Diamond)} \langle A \rangle \Rightarrow \Diamond A$, we get $\vdash_{S(T^\Diamond)} \langle \varGamma \rangle \Rightarrow \Diamond A$, by (CUT).
3. For $(\Box L)$, assume $\varGamma[A] \Rightarrow_{S(T^\Diamond)} B$ and $\Box A \in T^\Diamond$. Since $\vdash_{S(T^\Diamond)} \langle \Box A \rangle \Rightarrow A$, we get $\vdash_{S(T^\Diamond)} \varGamma[\langle \Box A \rangle] \Rightarrow B$, by (CUT).
4. For $(\Box R)$, assume $\vdash_{S(T^\Diamond)} \langle \varGamma \rangle \Rightarrow B$ and $\Box B \in T^\Diamond$. By Lemma 7, there exists $D \in T^\Diamond$ such that $\vdash_{S(T^\Diamond)} \langle D \rangle \Rightarrow B$ and $\varGamma \Rightarrow_{S(T^\Diamond)} D$. Then $\langle D \rangle \Rightarrow B \in S^{T^\Diamond}$, by Lemma 6. By (R2), $D \Rightarrow \Box B \in S^{T^\Diamond}$, which yields $D \Rightarrow_{S(T^\Diamond)} \Box B$. Hence we get $\vdash_{S(T^\Diamond)} \varGamma \Rightarrow \Box B$, by (CUT).
5. For (4), assume $\vdash_{S(T^\Diamond)} \varGamma[\langle \varDelta \rangle] \Rightarrow A$. By Lemma 8 there exists $\Diamond D \in T^\Diamond$ such that $\vdash_{S(T^\Diamond)} \varGamma[\Diamond D] \Rightarrow A$ and $\vdash_{S(T^\Diamond)} \varDelta \Rightarrow \Diamond D$. Since $\vdash_{S(T^\Diamond)} \langle \Diamond D \rangle \Rightarrow \Diamond D$, we get $\vdash_{S(T^\Diamond)} \langle\langle \Diamond D \rangle\rangle \Rightarrow \Diamond D$, by (CUT). Hence $\vdash_{S(T^\Diamond)} \varGamma[\langle\langle \varDelta \rangle\rangle] \Rightarrow A$, by two applications of (CUT).
6. For (T), assume $\vdash_{S(T\Diamond)} \varGamma[\langle \varDelta \rangle] \Rightarrow A$. By Lemma 8, there exists $\Diamond D \in T^\Diamond$ such that $\vdash_{S(T\Diamond)} \varGamma[\Diamond D] \Rightarrow A$ and $\vdash_{S(T\Diamond)} \varDelta \Rightarrow \Diamond D$. Clearly, $\vdash_{S(T\Diamond)} \varGamma[\varDelta] \Rightarrow A$, by (CUT).
7. For (K), assume $\vdash_{S(T^\Diamond)} \varGamma[\langle \varDelta_1 \rangle \circ \langle \varDelta_2 \rangle] \Rightarrow A$. By Lemma 7, there exists $D \in T^\Diamond$ such that $\vdash_{S(T^\Diamond)} \varGamma[D] \Rightarrow A$ and $\vdash_{S(T^\Diamond)} \langle \varDelta_1 \rangle \circ \langle \varDelta_2 \rangle \Rightarrow D$. Then, by applying Lemma 7 twice, we get $\vdash_{S(T^\Diamond)} \varDelta_1 \Rightarrow D_1$, $\vdash_{S(T^\Diamond)} \varDelta_2 \Rightarrow D_2$ and $\vdash_{S(T^\Diamond)} \langle D_1 \rangle \circ \langle D_2 \rangle \Rightarrow D$, for some D_1, $D_2 \in T^\Diamond$. By the proof of case 1, we get $\vdash_{S(T^\Diamond)} \Diamond D_1 \circ \Diamond D_2 \Rightarrow D$. We consider three possibilities.

   First, $D \in T$. By Lemma 6, we obtain $\Diamond D_1 \circ \Diamond D_2 \Rightarrow D \in S^{T^\Diamond}$. Hence, by (R3), $\vdash_{S(T^\Diamond)} D_1 \circ D_2 \Rightarrow \Box D$. Since $\vdash_{S(T^\Diamond)} \langle \Box D \rangle \Rightarrow D$, we get $\vdash_{S(T^\Diamond)} \langle D_1 \circ D_2 \rangle \Rightarrow D$, by (CUT). Then, by three applications of (CUT), we get $\vdash_{S(T^\Diamond)} \varGamma[\langle \varDelta_1 \circ \varDelta_2 \rangle] \Rightarrow A$.

   Second, $D \in T^\Box$ but $D \notin T$. Assume $D = \Box D'$, for some $D' \in T$. Since $\vdash_{S(T^\Diamond)} \langle \Box D' \rangle \Rightarrow D'$, by the proof for case 5, we obtain $\vdash_{S(T^\Diamond)} \langle\langle \Box D' \rangle\rangle \Rightarrow D'$. Then, due to the proof for case 4, we get $\vdash_{S(T^\Diamond)} \langle \Box D' \rangle \Rightarrow \Box D'$. Hence $\vdash_{S(T^\Diamond)} \langle D \rangle \Rightarrow D$, which yields $\vdash_{S(T^\Diamond)} \langle \Diamond D_1 \circ \Diamond D_2 \rangle \Rightarrow D$. Since $\langle D_1 \rangle \Rightarrow \Diamond D_1 \in S^{T^\Diamond}$ and $\langle D_2 \rangle \Rightarrow \Diamond D_2 \in S^{T^\Diamond}$, then, by rule (R4), $D_1 \Rightarrow \Diamond D_1 \in S^{T^\Diamond}$ and $D_2 \Rightarrow \Diamond D_2 \in S^{T^\Diamond}$. Hence $\vdash_{S(T^\Diamond)} \langle D_1 \circ D_2 \rangle \Rightarrow D$, by (CUT). We get $\vdash_{S(T^\Diamond)} \varGamma[\langle \varDelta_1 \circ \varDelta_2 \rangle] \Rightarrow A$, by (CUT).

   Third, $D \in T^\Diamond$, but $D \notin T^\Box$. Then, $D = \Diamond D^*$, for some $D^* \in T^\Box$. Since $\vdash_{S(T^\Diamond)} \langle \Diamond D^* \rangle \Rightarrow \Diamond D^*$, clearly, we get $\vdash_{S(T^\Diamond)} \langle D \rangle \Rightarrow D$. Again, $\vdash_{S(T^\Diamond)} \langle D_1 \circ D_2 \rangle \Rightarrow D$. Hence $\vdash_{S(T^\Diamond)} \varGamma[\langle \varDelta_1 \circ \varDelta_2 \rangle] \Rightarrow A$, by several applications of (CUT). $\Box$

We define an operator $\sharp$ on formula structure recursively as follows: (i) $A^\sharp = A$, for any formula A. (ii) $(\varGamma_1 \circ \varGamma_2)^\sharp = \varGamma_1^\sharp \bullet \varGamma_2^\sharp$, for any formula structures $\varGamma_1$ and $\varGamma_2$. (iii) $(\langle \varGamma \rangle)^\sharp = \Diamond(\varGamma)^\sharp$, for any formula structure $\varGamma$.

Now we are ready to prove the main results of this paper.

Theorem 10. *If* Φ *is finite, then* $\mathrm{NL_{S4}}(\Phi)$ *is decidable in polynomial time.*

Proof: Let Φ be a finite set of non-logical assumptions and $\varGamma \Rightarrow A$ be a sequent. Clearly, $\vdash_{\mathrm{NL_{S4}}(\Phi)} \varGamma \Rightarrow A$ can be checked in polynomial time if and only if $\vdash_{\mathrm{NL_{S4}}(\Phi)} \Gamma^\sharp \Rightarrow A$ can be checked in polynomial time. Let n be the number of logical constants and atoms occurring in $\Gamma^\sharp \Rightarrow A$ and in sequents for Φ. The number of subformulae of any formula is equal to the number of logical constants and atoms in it. Hence T can be constructed in time $O(n^2)$, and T contains at most n elements. It yields that we can construct $T^\diamond$ in time $O(n^2)$. Since $T \subseteq T^\diamond$, by Corollary 4, $\vdash_{\mathrm{NL_{S4}}(\Phi)} \Gamma^\sharp \Rightarrow A$ is provable in $\mathrm{NL_{S4}}(\Phi)$ iff $\Gamma^\sharp \Rightarrow_{T^\diamond} A$. By Lemma 10, $\Gamma^\sharp \Rightarrow_{T^\diamond} A$ iff $\vdash_{S(T^\diamond)} \Gamma^\sharp \Rightarrow A$. Since $\Gamma^\sharp \Rightarrow A$ is a basic sequent, we get $\vdash_{S(T^\diamond)} \Gamma^\sharp \Rightarrow A$ iff $\Gamma^\sharp \Rightarrow A \in S^{T^\diamond}$, by Lemma 6. Hence $\Gamma^\sharp \Rightarrow A$ is provable in $\mathrm{NL_{S4}}(\Phi)$ iff $\Gamma^\sharp \Rightarrow A \in S^{T^\diamond}$. Besides, by Lemma 5, $S^{T^\diamond}$ can be constructed in polynomial time. Consequently, $\vdash_{\mathrm{NL_{S4}}(\Phi)} \Gamma^\sharp \Rightarrow A$ can be checked in time polynomial with respect to n. □

An $\mathrm{NL_{S4}}(\Phi)$-grammar over an alphabet Σ is a pair $\langle \mathcal{L}, \mathcal{D}\rangle$, where $\mathcal{L}$, the lexicon, is a finite relation between strings from Σ^+ and formulae of $\mathrm{NL_{S4}}(\Phi)$, and $\mathcal{D} \subseteq \mathcal{F}$ is a finite set of designated formulae (types).

By $s(\Gamma)$, we denote a string obtained from a formula structure Γ by dropping all binary and unary operators, $\circ$ and $\langle\rangle$, respectively, and corresponding parentheses (). A language $L(G)$ generated by a $\mathrm{NL_{S4}}(\Phi)$-grammar $G = \langle \mathcal{L}, \mathcal{D}\rangle$ is defined as a set of strings $a_1 \cdots a_n$, where $a_i \in \Sigma^+$, $1 \le i \le n$, and $n \ge 1$, satisfying the following condition: there exist formulae $A_1, \ldots, A_n$, S, and formulae structure Γ such that for all $1 \le i \le n$ $\langle a_i, A_i\rangle \in \mathcal{L}$, $S \in \mathcal{D}$, and $\vdash_{\mathrm{NL_{S4}}(\Phi)} \Gamma \Rightarrow S$ where $s(\Gamma) = A_1 \cdots A_n$.

Notice that for $\mathrm{NL_{S4}}(\Phi)$-grammars the definition of $L(G)$ can be modified by assuming that Γ does not contain $\langle\rangle$. For if $\Gamma \Rightarrow A$ is provable in $\mathrm{NL_{S4}}(\Phi)$, then $\Gamma' \Rightarrow B$ is provable in $\mathrm{NL_{S4}}(\Phi)$, where Γ' arises from Γ by dropping all $\langle\rangle$.

Theorem 11. *Every language generated by an* $\mathrm{NL_{S4}}(\Phi)$*-grammar is context-free.*

Proof: Let Φ be a finite set of sequents of the form $A \Rightarrow B$, $G_1 = \langle \mathcal{L}, \mathcal{D}\rangle$ be an $\mathrm{NL_{S4}}(\Phi)$-grammar, and T be the set of all subformulae of formulae in $\mathcal{D}$ and all subformulae of formulae appearing in $\mathcal{L}$. We construct $T^\diamond$ as above. Now we construct an equivalent *CFG* (context-free grammar) G_2, in the following way. The terminal elements of G_2 are lexical items of G_1. The non-terminals are all types from $T^\diamond$ and a fresh non-terminal S. Productions are $\{A \to B \mid B \Rightarrow A \in S^{T^\diamond}\} \cup \{A \to B,\, C \mid B \circ C \Rightarrow \in S^{T^\diamond}\} \cup \{A \to v \mid \langle v,\, A\rangle \in L\} \cup \{S \to A \mid A \in D\}$.

If $v_1 \ldots v_m$ is generated by G_1, then there is a $\mathrm{NL_{S4}}(\Phi)$-derivable sequent $\Gamma \Rightarrow B$ where B is a designated type, $s(\Gamma) = A_1 \cdots A_m$, and $\langle v_i,\, A_i\rangle \in \mathcal{L}$ for $1 \le i \le m$. We get $S \to^*_{G_2} B$ by the construction of G_2. Due to Lemma 9 and the construction of G_2, we obtain $S \to^*_{G_2} A_1 \cdots A_m$ which leads to $S \to^*_{G_2} v_1 \ldots v_m$. Hence $v_1 \cdots v_m$ is generated by G_2.

Now suppose $v_1 \cdots v_m$ is generated by G_2, which means $S \to^*_{G_2} v_1 \cdots v_n$. Then, there exists a $B \in \mathcal{D}$ such that $B \to^*_{G_2} A_1 \cdots A_n$ where $\langle v_i,\, A_i\rangle \in \mathcal{L}$, $1 \le i \le m$. Hence, by the construction of G_2, there exists a formula structure Γ such that $s(\Gamma) = A_1 \cdots A_n$ and $\vdash_{S(T^\diamond)} \Gamma \Rightarrow B$. By Lemma 9, $\vdash_{\mathrm{NL_{S4}}(\Phi)} \Gamma \Rightarrow B$. Therefore $v_1 \cdots v_m$ is generated by G_1. □

Obviously, we can easily obtain the same results for systems without K (NL_{S4} in the sense of Plummer [16][17]). The inclusion of the class of ε-free context free languages in the class of $NL_{S4}(\Phi)$-recognizable languages can be easily established. Every context-free language is generated by some NL-grammars (see [8]). Since neither the *lexicon* nor *designated formulae* contain modal operators, by Corollary 4, these NL-grammars can be conceived of as an $NL_{S4}(\Phi)$-grammars, where Φ is empty. Hence $NL_{S4}(\Phi)$-grammars generate exactly the ε-free context-free languages.

4 Variants

In [6], grammars based on $L\Diamond$ enriched with structural rules (K_1), (K_2) are shown to surpass the context-free languages. Plummer [16][17] conjectures that structural rules (K_1), (K_2) extend the generative capacity of $NL\Diamond$. However the situation is different for system with 4 and T. We consider a system $NL_{S4'}$, which admits 4, T, and

$$K_1: \quad \Diamond(A \bullet B) \Rightarrow \Diamond A \bullet B \qquad K_2: \quad \Diamond(A \bullet B) \Rightarrow A \bullet \Diamond B$$

The sequent system of $NL_{S4'}$ is obtained by extending $NL_{S4'}$ without (K) with the following rules corresponding to axioms K_1 K_2

$$(K_1)\ \frac{\Gamma[\langle\Delta_1\rangle \circ \Delta_2] \Rightarrow A}{\Gamma[\langle\Delta_1 \circ \Delta_2\rangle] \Rightarrow A}, \quad (K_2)\ \frac{\Gamma[\Delta_1 \circ \langle\Delta_2\rangle] \Rightarrow A}{\Gamma[\langle\Delta_1 \circ \Delta_2\rangle] \Rightarrow A}$$

The main results in section 3 can be easily extended to $NL_{S4'}(\Phi)$, $NL_{S4'}(\Phi)$ with finitely many non-logical assumptions Φ, utilizing the technique presented in section 3. We outline the proof as follows.

It is easy to prove the analogues of theorem 2 for $NL_{S4'}(\Phi)$. Then we modify the construction of $S(T^\Diamond)$. We replace rule (R3) by the following two rules:

(R3.1) if $\Diamond A \circ B \Rightarrow C \in S_n$ and $C \in T$ then $A \circ B \Rightarrow \Box C \in S_n + 1$,
(R3.2) if $A \circ \Diamond B \Rightarrow C \in S_n$ and $C \in T$ then $A \circ B \Rightarrow \Box C \in S_n + 1$.

The proof of the analogue of Lemma 9 is similar to the proof of Lemma 9 in section 3, except for replacing $\langle\Delta_1\rangle\circ\langle\Delta_2\rangle$ by $\langle\Delta_1\rangle\circ\Delta_2$ or $\Delta_1\circ\langle\Delta_2\rangle$ and $\Diamond D_1\circ\Diamond D_2$ by $\Diamond D_1 \circ D_2$ or $D_1 \circ \Diamond D_2$. The remainder of the proof goes without changes.

5 Conclusion

This article shows that the consequence relation of NL_{S4} or $NL_{S4'}$ are polynomial time decidable, and the categorial grammars based on $NL_{S4}(\Phi)$ or $NL_{S4'}(\Phi)$ generate context-free languages. The basic idea of proof is adapted from Buszkowski [3]. However the proof of the extended subformula property is different. We introduce the Φ-*restricted cut* and prove the cut-elimination theorem for systems enriched with finitely many assumptions.

References

1. Buszkowski, W.: Some decision problems in the theory of syntactic categories. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 28, 539–548 (1982)
2. Buszkowski, W.: Generative Capacity of Nonassociative Lambek Calculus. Bulletin of Polish Academy of Sciences 34, 507–516 (1986)
3. Buszkowski, W.: Lambek Calculus with Nonlogical Axioms. In: Casadio, C., Scott, P.J., Seely, R.A. (eds.) Languages and Grammars Studies in Mathematical Linguistics and Natural Language. Lecture Notes, pp. 77–93. CSLI, Stanfor (2005)
4. Buszkowski, W., Farulewki, M.: Nonassociative Lambek Calculus with Additives and Context-Free Languages. In: Grumberg, O., Kaminski, M., Katz, S., Wintner, S. (eds.) Languages: From Formal to Natural. LNCS, vol. 5533, pp. 45–58. Springer, Heidelberg (2009)
5. Girard, J.Y.: Linear Logic. Theoretical Computer Science 50, 1–102 (1987)
6. Jäger, G.: On the generative capacity of multi-modal categorial grammars. Research on Language and Computation 1, 105–125 (2003)
7. Jäger, G.: Residuation, Structural Rules and Context Freeness. Journal of Logic, Language and Informationn 13, 47–59 (2004)
8. Kandulski, M.: The equivalence of Nonassociative Lambek Categorial Grammars and Context-free Grammars. Zeitschrift für mathematische Logik und Grundlagen der Mathematik 52, 34–41 (1988)
9. Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly 65, 154–170 (1958)
10. Lambek, J.: On the calculus of syntactic types. In: Structure of Language and Its Mathematical Aspects, pp. 168–178. American Mathematical Society, Providence (1961)
11. Lambek, J.: Type Grammars as Pregroups. Grammars 4, 21–39 (2001)
12. Moortgat, M.: Multimodal linguistic inference. Journal of Logic, Language and Information 5, 349–385 (1996)
13. Morrill, G.: Intensionality and boundedness. Linguistics and Philosopliy 13, 699–726 (1990)
14. Pentus, M.: Lambek grammars are context-free. In: Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science, pp. 429–433 (1993)
15. Pentus, M.: Lambek calculus is NP-complete. Theoretical Computer Science 357, 186–201 (2006)
16. Plummer, A.: S4 enriched multimodal categorial grammars are context-free. Theoretical Computer Science 388, 173–180 (2007)
17. Plummer, A.: S4 enriched multimodal categorial grammars are context-free: Corrigendum. Theoretical Computer Science 403, 406–408 (2008)
18. Versmissen, J.: Grammatical Composition: Modes, Models, Modalities. PhD thesis, Universiteit Utrecht (1996)

Hard Counting Problems for Partial Words

Florin Manea[1,2,⋆] and Cătălin Tiseanu[1]

[1] Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, 010014, Bucharest, Romania

[2] Faculty of Computer Science, Otto-von-Guericke-University Magdeburg,
PSF 4120, D-39016 Magdeburg, Germany

{flmanea,ctiseanu}@gmail.com

Abstract. In this paper we approach several decision and counting problems related to partial words, from a computational point of view. First we show that finding a full word that is not compatible with any word from a given list of partial words, all having the same length, is NP-complete; from this we derive that counting the number of words that are compatible with at least one word from a given list of partial words, all having the same length, is #P-complete. We continue by showing that some other related problems are also #P-complete; from these we mention here only two: counting all the distinct full words of a given length compatible with at least one factor of the given partial word, and counting all the distinct squares compatible with at least a factor of a given partial word.

Keywords: Partial Words, NP-completeness, #P Complexity Class, #P-complete Problems, Combinatorics on Words.

1 Introduction

Partial words, a canonical extension of the classical words, are sequences that, besides regular symbols, may have a number of unknown symbols, called holes or wild cards. The study of the combinatorial properties of partial words was initiated by Berstel and Boasson in their paper [2], having as motivation an intriguing practical problem, namely gene comparison, related to the central topics of combinatorics on words. Until now, several such combinatorial properties of the partial words have been investigated: periodicity, conjugacy, freeness and primitivity (see [3] for an extensive survey and further references on such works). Part of these studies consisted in finding efficient algorithms testing if a word and its factors verify a given combinatorial property ([4,5,8,10]).

A research direction that appeared in the study of partial words, related to those mentioned already, consists in the study of a series of problems on identifying and counting factors, which verify some restrictions, of partial words (such as repetitions, primitive factors, etc.). However, in the case of counting problems ([6,7]) one is usually interested in finding the number of all the different

⋆ Florin Manea acknowledges support from the *Alexander von Humboldt Foundation.*

full words, satisfying some specific conditions, that are compatible with factors of a given partial word (for instance, square full words that are compatible with factors of a given partial word, or full words of a fixed length that are compatible with factors of a given partial word, etc.). Until now, most of the results obtained on this topic state mathematical properties of the functions that may express the result of such counting problems. Here we are interested in the computational aspects of these problems: we prove that, in some cases, counting problems for partial words are complete for the class #P, thus hard problems.

The structure of our paper is the following: we start by giving some basic definitions, we continue by proving that a series of problems on partial words are NP-complete, and for each of these problems we show that they have an associated hard counting problem, and, finally, we propose three counting problems, that cannot be associated canonically with NP-complete problems, but are still complete for the class #P.

2 Basic Definitions

A *partial word* of length n over the alphabet A is a partial function $u: \{1,\ldots,n\} \overset{\circ}{\rightarrow} A$. For $i \in \{1,\ldots,n\}$, if $u(i)$ is defined we say that *i belongs to the domain of u* (denoted by $i \in D(u)$), otherwise we say that *i belongs to the set of holes of u* (denoted by $i \in H(u)$). For convenience, finite partial words are seen as words over the extended alphabet $A \cup \{\diamond\}$: a partial word u of length n is depicted as $u = a_1 \ldots a_n$, where $a_i = u(i)$, for $i \in D(u)$, and $a_i = \diamond$, otherwise. In this way, one can easily define the catenation, respectively the equality, of partial words, as the catenation, respectively the equality, of the corresponding words over $A \cup \{\diamond\}$ (see [3] for details); we denote by λ the empty partial word (i.e., the partial word of length 0). If u and v are two partial words of equal length, then *u is said to be contained in v*, $u \subset v$, if all the elements of $D(u)$ are contained in $D(v)$ and $u(i) = v(i)$ for all $i \in D(u)$. Two partial words *u and v are compatible*, denoted $u \uparrow v$, if there exists a partial word w such that $u \subseteq w$ and $v \subseteq w$. We say that *the partial word u is a factor of the partial word w* if there exist partial words x and y such that $w = xuy$. If $w = a_1 \ldots a_n$, we let $w[i..j]$ denote the factor $a_i \ldots a_j$ of w, and by $w[i]$ the symbol a_i.

Let $w \in (A \cup \{\diamond\})^*$ be a partial word; *w is said to be a k-repetition* if $w = x_1 \ldots x_k$ and there exists a non-empty partial word u such that $x_i \subseteq u$ for all $i \in \{1,\ldots,k\}$. Usually, 2-repetitions are called squares; in the case of full words, a square over V is a word having the form xx, with $x \in V^+$. The reader interested in more definitions and results on partial words is referred to [3].

For the definitions regarding the computational complexity notions appearing in this paper, such as different complexity classes, NP-complete problems, polynomial-time reductions and Turing reductions, we refer to [9]. For the definition of the complexity class #P and for some seminal results regarding #P-complete problems we refer to [11]. However, we briefly recall the definition of the basic NP-complete problem CNF-SAT (Satisfiability of Boolean Formulas in conjunctive normal form).

Problem 1. *Given f a boolean formula in conjunctive normal form, with the variables $S = \{x_1, \ldots, x_k\}$, i.e., $f = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each C_i is the disjunction of several literals (variables from S or the negation of these variables), decide if there exists an assignment of the variables from S that makes f true.*

The natural counting problem associated with CNF-SAT, usually denoted by #CNF-SAT, asks how many assignments of the variables from S, that make f true, exist. This problem is #P-complete.

3 Several NP-Complete Problems and the Associated Counting Problems

First we will show that the following problem is NP-complete.

Problem 2. *Given a natural number L, and a list of partial words $S = \{w_1, w_2, \ldots, w_k\}$, $w_i \in (V \cup \{\diamond\})^L$ for all i, decide if there exists a word $v \in V^L$ such that v is not compatible with any of the words in S.*

To do this we will present a polynomial-time reduction from the CNF-SAT problem to this problem.

Proposition 1. *Problem 2 is NP-complete.*

Proof. First, assume we are given a list of partial words $S = \{w_1, w_2, \ldots, w_k\}$, over the alphabet V, each partial word having the same length L. We can easily construct a nondeterministic Turing machine, working in polynomial time, that decides if there exists a word $v \in V^L$, such that v is not compatible with any of the words in S. We simply choose a word from V^L nondeterministically and then check (deterministically) if it is compatible with any of the words in S or not; if it is compatible with such a word we reject the input, otherwise we accept it. Thus, Problem 2 is in NP. In the following we show that the problem is also complete for the class NP.

Let us consider an instance of the CNF-SAT Problem. More precisely, let f be a boolean formula in conjunctive normal form, let L be the number of logical variables which appear in f, and denote these variables by $x_1, x_2, \ldots, x_L$, let n be the number of the clauses of f, and let $C_1, C_2, \ldots, C_n$ be these clauses (each of them being actually the disjunction of several literals). Then, if $f = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_n$, we observe that there exists an assignment of the variables $x_1, \ldots, x_L$ such that f evaluates to 1 if and only if there exists an assignment of the variables $x_1, \ldots, x_L$ such that $\bar{f}$ (the negation of f) evaluates to 0. Note that $\bar{f} = \overline{C}_1 \vee \overline{C}_2 \vee \ldots \vee \overline{C}_n$. We can now associate the following instance of Problem 2 with this instance of the CNF-SAT problem: consider the alphabet $V = \{0, 1\}$, the length of the words L, and construct the list of words $S = \{w_1, w_2, ..., w_n\}$, where w_i, with $i \in \{1, \ldots, n\}$, is defined as follows:

$$\textit{for } j \in \{1, \ldots, L\}, \textit{ let } w_i[j] = \begin{cases} 0, & \textit{if } \overline{x}_j \in \overline{C}_i; \\ 1, & \textit{if } x_j \in \overline{C}_i; \\ \diamond, & \textit{otherwise.} \end{cases}$$

It is clear that a word of length L over V, denoted by v, corresponds to an assignment of the variables $\{x_1, \ldots, x_L\}$, and conversely, in a canonical way: we simply take $x_i = v[i]$, for all $i \in \{1, \ldots, L\}$. Moreover, if such a word v is compatible with a partial word from the list S, say w_j, then the variables assignment defined by v will clearly make $\overline{C_j}$ equal to 1, thus $\overline{f}$ equal to 1. Conversely, an assignment of the variables that makes $\overline{f}$ equal to 1 makes at least one of the clauses $\overline{C_j}$ equal to 1; from this it follows easily that the word corresponding to that assignment is compatible with at least one of the partial words w_j, $j \in \{1, \ldots, n\}$. This shows that deciding if there exists an assignment that makes $\overline{f}$ equal to 1 (and f equal to 0) corresponds to deciding the existence of a full word compatible with at least one of the words of the set S. Clearly, deciding if there exists an assignment that makes f equal to 1 corresponds to deciding if there exists a full word which is not compatible with any of the words of the set S, thus solving Problem 2 for the list S.

Finally, notice that the reduction described above (from an instance of CNF-SAT to an instance of Problem 2) can be easily implemented by a deterministic Turing machine working in polynomial time. Consequently, it follows that Problem 2 is NP-complete. Nevertheless, this reduction clearly establishes a bijection between the solutions of the initial instance of CNF-SAT and the solutions of the instance of Problem 2 we construct; therefore, this reduction is parsimonious. □

Now consider the counting problem associated with Problem 2:

Problem 3. *Given a list of partial words $S = \{w_1, w_2, \ldots, w_k\}$, over the alphabet V, with $|V| \geq 2$, each partial word having the same length L, count the distinct words $v \in V^L$ such that v is compatible with at least one of the words in L.*

From Proposition 1 we can easily derive that this problem is #P-complete.

Proposition 2. *Problem 3 is #P-complete.*

Proof. One can easily modify the nondeterministic Turing machine described in the beginning of the proof of Proposition 1 such that it will have, for a given input, as many accepting paths as the number of words in V^L that are compatible with at least one of the words of the input list. Such a machine would choose a word from V^L nondeterministically and then check (deterministically) if it is compatible with any of the words in S or not; if it is compatible with such a word we accept the input, otherwise we reject it. Since it works in nondeterministic polynomial time, it follows that Problem 3 is in #P. The proof of Proposition 1 shows that #CNF-SAT can be easily reduced, by a Turing reduction (as explained in [11]) to Problem 3; consequently, this problem is #P-complete. □

In the following we address a problem regarding the factors of a fixed length of a given partial word.

Problem 4. *Given a partial word w, over the alphabet V, with $|V| \geq 2$, and a natural number L, with $0 < L \leq |w|$, decide if there exists a word $v \in V^L$ such that v is not compatible with any factor of length L of w.*

We show that this problem is NP-complete, by reducing Problem 2 to it.

Proposition 3. *Problem 4 is NP-complete.*

Proof. Let w, a partial word over the alphabet V and L a natural number with $0 < L \leq |w|$, be an input instance for our problem. It is not hard to construct a nondeterministic Turing machine, working in polynomial time, that decides if there exists a word $v \in V^L$, such that v is not compatible with any of the factors of length L of w. Consequently, Problem 4 is in NP. It remains to show that it is also complete for this class.

In this respect we consider an instance of the Problem 2. Let $S = \{w_1, w_2, \ldots, w_k\}$ be a list of partial words, over the alphabet V, each of them having the same length L. We consider the word $w = w_1\$w_2\$\ldots\$w_k\$\diamond^{L-1}\$\diamond^{L-1}\$$, where $\$ \notin V$. Clearly, this word can be constructed from the aforementioned instance of Problem 2 by a deterministic Turing machine working in polynomial time. We will show that there exists a word $v \in V^L$ which is not compatible with any of the words in S if and only if there exists a word $v \in (V \cup \$)^L$ such that v is not compatible with any factor of length L of w. Once we prove this statement it follows that we have a deterministic polynomial-time reduction from Problem 2 to Problem 4, and, since Problem 2 is NP-complete, it follows that Problem 4 is also NP-complete.

In order to finish the proof, let us analyze which words from $(V \cup \{\$\})^L$ can be compatible with factors of w. First of all, it is clear that all the words of length L that contain a \$ symbol are compatible with a factor of w, namely at least a factor from the suffix $\$\diamond^{L-1}\$\diamond^{L-1}\$$. Then, we notice that every word of length L from V^L that is compatible with a factor of w must be compatible with at least one of the words w_1, w_2, …, w_k; thus, if there exists a word from V^L that is not compatible with any of the words from S, then it is not compatible with any factor of w also, and conversely. This concludes our proof; however, note that the reduction we presented is, again, parsimonious. □

Again, we can consider the counting problem associated with Problem 4:

Problem 5. *Given a partial word w, over the alphabet V, with $|V| \geq 2$, and a natural number L, with $L \leq |w|$, count the distinct words $v \in V^L$ such that v is compatible with a factor of length L of w.*

As in the former case, we can easily derive that this problem is #P-complete.

Proposition 4. *Problem 5 is #P-complete.*

Proof. One can easily construct a nondeterministic Turing machine that accepts the words which are compatible with factors of w, and has as many accepting paths as the number of words in V^L compatible with a factor of the input partial word. Consequently, Problem 5 is in #P. The reduction from the previous proof can be used Turing-reduce Problem 3 to Problem 5; therefore, Problem 5 is also #P-complete. □

Propositions 3 and 4 have many implications. First, one can show the following result, using similar reductions:

Corollary 1. *Consider the following problems:*

(i). Given a partial word w, over the alphabet V, with $|V| \geq 2$, and a natural number L, with $L \leq |w|$, decide if there exists a natural number k, with $0 < k \leq L$, and a word $v \in V^k$, such that v is not compatible with any factor of w.
(ii). Given a partial word w, over the alphabet V, with $|V| \geq 2$, and a natural number L, with $L \leq |w|$, count the words $v \in V^k$, with $0 < k \leq L$, such that v is compatible with any factor of w.

Problem **(i)** *is NP-complete and Problem* **(ii)** *is #P-complete.*

Both Problem 5 and the Problem **(ii)** stated in Corollary 1 are related to the subword complexity of a word, as defined in [1]. The subword complexity of a full word is defined for finite and right infinite words as follows: let V be a finite alphabet and w be a finite or right infinite word over V; the subword complexity of w is the function which assigns to each positive integer, n, the number, $p_w(n)$, of distinct factors of length n of w. One can give a similar definition for partial words ([7]): let V be a finite alphabet and w be a finite or right infinite partial word over V; the subword complexity of w is the function which assigns to each positive integer, n, the number, $p_w(n)$, of distinct full words over V that are compatible with factors of length n of w.

A direct consequence of Proposition 4 is that computing the subword complexity of a finite partial word is #P-complete.

Also, we can consider a class of very simple right infinite partial words: let V be an alphabet, with $|V| \geq 2$, and let $C = \{w\$ \diamond^{L-1} \$ \diamond^{L-1} \$ \ldots \mid w \in (V \cup \diamond)^*, \$ \notin V, 0 < L \leq |w|\}$. Clearly, one can compute the value $p_x(n)$ for every word $x \in C$ and every natural number $n \in \mathbf{N}$. Moreover, each word $x \in C, x = w\$\diamond^{L-1}\$\diamond^{L-1}\$\ldots$ for some $w \in (V \cup \diamond)^*$, can be described succinctly in the following manner: we consider the morphism $\phi : V \cup \{\$, \diamond\} \to (V \cup \{\$, \diamond\})^*$, defined by $\phi(a) = a$, for all $a \in V$, $\phi(\diamond) = \diamond$ and $\phi(\$) = \$ \diamond^{L-1} \$$; clearly $x = \lim_{n\to\infty} \phi^n(w\$)$ and the space needed to represent x in this way is $\mathcal{O}(|w|)$. Now we can consider the problem of computing the subword complexity of the infinite partial words from the class C: "given $x \in C$ and $n \in \mathbf{N}$, compute $p_x(k)$, for all $k \leq n$". However, it is not hard to see that if $n \geq L$ solving this restricted problem implies solving Problem 5 for the word w and the number L. As we have already shown, Problem 5 is #P-complete, thus the problem "given $x \in C$ and $n \in \mathbf{N}$, $n \geq L$, compute $p_x(k)$, for all $k \leq n$" is also $\#P$-hard. Consequently, computing the subword complexity of the infinite partial words from the class C is $\#P$-hard. Further, this shows that computing the subword complexity of a infinite partial words (when it is possible) is a hard counting problem.

Finally, one may be interested in counting all the full words that are compatible with factors of a partial word w, over an alphabet with at least 2 symbols.

We were not able to show neither that this problem can be solved efficiently nor that it is a hard counting problem. However, we conjecture that it is a #P-complete problem, as well. In this respect, the result of Proposition 5 shows that a natural approach would not yield an efficient solution of the problem: one cannot hope to solve it efficiently by counting separately the factors of length k of the partial word w, for all $k \leq |w|$, and summing up the results afterwards.

4 Further Hard Counting Problems for Partial Words

The hard counting problems that we presented in the last Section have a common feature: they can all be associated canonically with (and were actually derived from) hard decision problems. However, as stated in [11], the most interesting hard counting problems are those that do not have a corresponding hard decision problem. Such a problem is, for instance, the open problem mentioned in the end of the previous Section: count all the full words that are compatible with factors of a partial word w, over an alphabet with at least 2 symbols. It is clear that one can efficiently decide if there exists or not a full word, of length less or equal to n, that is not compatible with any factor of the partial word w, where $w \in (V \cup \{\diamond\})^n$. If w does contain a symbol a, other than $\diamond$, on the position i, we construct a word of the same length with w having the symbol b on the position i, where $b \in V \setminus \{a\}$ and this word would not be compatible with any factor of w; otherwise, i.e., w contains only $\diamond$ symbols, we would say that any full word, of length less or equal to n, is compatible with a factor of w. Also, if $n > 0$ then there exists always a word compatible with a factor of w: if w contains a symbol a different from $\diamond$, then a is such a word; otherwise, any word of length 1 is compatible with a factor of w. Thus, the counting problem we mentioned cannot be associated canonically with a hard decision problem.

In the following we present a series of other counting problems that cannot be associated with hard decision problems, but which can be shown to be #P-complete. While the first two are related somehow to the problem of counting all the distinct full words compatible with the factors of a partial word, the last one comes from the area of combinatorics on words, being related to the problem of counting distinct squares in a partial word (see [6]).

The first problem consists in counting all the full words, over a restricted alphabet, that are compatible with the factors of a partial word.

Problem 6. *Given a partial word w, over the alphabet V, with $|V| \geq 3$, and a symbol $\$ \in V$, count the words $v \in (V \setminus \{\$\})^*$, with $0 < |v| \leq |w|$, that are compatible with a factor of w.*

Note that this problem cannot be associated canonically with a NP-complete problem. The existence of a word over $V \setminus \{\$\}$ compatible with a factor of w is easy to settle: if w contains a $\diamond$ symbol or a symbol $a \in V \setminus \{\$\}$ then a is such a word; otherwise, if w contains only \$ symbols, then such a word does not exist. The existence of a word over $V \setminus \{\$\}$ which is not compatible with any factor of w is again easy: if w does contain a symbol $a \in V$ on the position i then we construct a word of the same length with w having the symbol b on the position i, where $b \in V \setminus \{a, \$\}$, and this word would not be compatible with any factor of w; otherwise, i.e., w contains only $\diamond$ symbols, we would say that any full word over $V \setminus \{\$\}$, of length less or equal to $|w|$, is compatible with a factor of w.

However, we show that Problem 6 is a hard counting problem, by giving a Turing reduction from Problem 5.

Proposition 5. *Problem 6 is #P-complete.*

Proof. It is not hard to see that this problem is in #P. Let w, a partial word over the alphabet V, and \$, a symbol from V, be an instance of our problem. We design a nondeterministic polynomial Turing machine that accepts the words $v \in (V \setminus \{\$\})^k$, for $k \leq |w|$, such that v is compatible with a factor of w. The machine chooses a word from $(V \setminus \{\$\})^*$, shorter than w, nondeterministically and checks (deterministically) if it is compatible with a factor of w; if so we accept the input, otherwise we reject it. This machine has as many accepting paths as the number of words $v \in (V \setminus \{\$\})^*$, with $0 < |v| \leq |w|$, compatible with a factor of w. Consequently, Problem 6 is in #P. It remains to show that it is complete for this class.

Assume that there exists a function Solve$(w, \$)$ that can compute efficiently, on a deterministic Turing machine, the solution of Problem 6, having as input data w and the symbol \$. Also, consider an instance of the Problem 3: $S = \{w_1, w_2, \ldots, w_k\}$ is a list of partial words of length L, over the alphabet $V \setminus \{\$\}$ which has at least two symbol. As in the proof of Proposition 4, we construct the word $w = w_1\$w_2\$\ldots\$w_k\$ \diamond^{L-1} \$ \diamond^{L-1} \$$. It is not hard to see that by running the function Solve$(w, \$)$ we will obtain the result $(|V|-1)^{L-1} + N_L$, where N_L is the number of words over $V \setminus \{\$\}$ compatible with at least a word from the list S; moreover, N_L can be efficiently obtained by subtracting $(|V|-1)^{L-1}$ from the result of the function Solve$(w, \$)$. Thus, if there exists an efficient solution of Problem 6 (encoded by a function Solve$(w, \#)$) then we will also have an efficient solution for Problem 3. Since Problem 3 is #P-complete, it follows that Problem 6 is also #P-complete. □

In the following we consider a restricted compatibility relation. Given two partial words u and v over the alphabet V, $|V| \geq 3$, and a symbol $s \in V$, we say that u and v are compatible$-s$ (the words are "compatible minus s"), denoted by $u \uparrow_s v$, if there exists a partial word w such that $u \subseteq w$ and $v \subseteq w$ and for each $i \in H(u) \cup H(v)$ we have $w[i] \neq s$. Intuitively, the idea behind this compatibility relation is that the $\diamond$ symbol is seen as a wild card replacing any symbol from the alphabet, except for s; in the usual case $\diamond$ can replace all the symbols of the alphabet.

In the following we will consider the problem of counting, for a partial word w over V and a symbol $\$ \in V$, all the full words that are compatible-\$ with a factor of w.

Problem 7. *Given a partial word w, over the alphabet V, with $|V| \geq 3$, and a symbol $\$ \in V$, count the words $v \in V^*$, with $0 < |v| \leq |w|$, that are compatible-\$ with a factor of w.*

One can show, similarly to the case of Problem 6, that Problem 7 cannot be associated canonically with a NP-complete problem: it can efficiently decided if there exists a word $v \in V^*$, with $0 < |v| \leq |w|$, compatible-\$ with a factor of w, or if there exists a word shorter than w which is not compatible-\$ with any factor of w. Further we show how Problem 7 is #P-complete.

Proposition 6. *Problem 7 is #P-complete.*

Proof. Again, it is not hard to see that this problem is in #P. We will not go into details on how we can design a nondeterministic polynomial Turing machine having as many accepting paths as the number of words $v \in V^*$, with $0 < |v| \leq |w|$, compatible-\$ with a factor of w; basically, it is the same construction as in the previous proof. It remains to show that Problem 7 is complete for #P.

Assume that there exists a function $\text{Comp}(w, \$)$ that efficiently computes (on a deterministic Turing machine) the solution of Problem 7, having as input data w and the symbol \$. Further, consider an input instance of the Problem 5: w is a partial word over the alphabet $V \setminus \{\$\}$ (note that $|V \setminus \{\$\}| \geq 2$), and L is a natural number with $0 < L \leq |w|$; we should compute the number $X_{w,L}$, of the full words over $V \setminus \{\$\}$ compatible with at least a factor of length L of w.

Consider now the words $w_1 = \diamond^{L-1}\$w$ and $w_2 = \diamond^L\$w$. We briefly analyze the words that are compatible-\$ with factors of these two words:

– All the full words from $(V \setminus \{\$\})^k$, with $k \leq L - 1$, are compatible-\$ with factors of both w_1 and w_2, namely with factors of the form $\diamond^k$.
– A full word of length k, with $k \leq L - 1$, containing \$, is compatible-\$ with a factor of w_1 if and only if it is compatible-\$ with a factor of w_2. These words have the form $x\$y$, with $x \in V^p$, for some $p < k$, and $y \in V^{k-p-1}$ compatible with a prefix of w.
– A full word from $(V \setminus \{\$\})^L$ is compatible-\$ with a factor of w_1 if and only if it is compatible with a factor of w. On the other hand, all the full words from $(V \setminus \{\$\})^L$ are compatible-\$ with factors of w_2.
– A full word of length L, containing \$, is compatible-\$ with a factor of w_1 if and only if it is compatible-\$ with a factor of w_2. These words have the form $x\$y$, with $x \in V^p$, $p < L$, and $y \in V^{L-p-1}$ compatible with a prefix of w.
– A full word from $(V \setminus \{\$\})^k$, with $L + |w| + 1 \geq k > L$, is compatible-\$ with a factor of w_1 if and only if it is compatible with a factor of w. The same holds for the full words from $(V \setminus \{\$\})^k$, with $L + |w| + 1 \geq k > L$, compatible-\$ with factors of w_2.
– A full word of length k, with $L + |w| \geq k > L$, containing \$, is compatible-\$ with a factor of w_1 if and only if it has the form $x\$y$, with $0 \leq |x| \leq L - 1$ and y is a prefix of w. A full word of length k, with $L + |w| \geq k > L$, containing \$, is compatible-\$ with a factor of w_2 if and only if it has the form $x\$y$, with $0 \leq |x| \leq L$ and y is a prefix of w. Therefore, one can compute exactly the difference between the number of words of length k, with $L + |w| \geq k > L$, compatible-\$ with factors of w_2, and the number of such words compatible-\$ with factors of w_1: this difference, which will be denoted in the following by

$N_{k,w}$, equals $L^{|V|-1}|y|_\diamond^{|V|-1}$, if the number of $\diamond$ symbols from y (denoted $|y|_\diamond$) is greater than 0, or $L^{|V|-1}$, otherwise. Clearly, given w one can compute $N_{k,w}$ in polynomial time.
– There are no words of length $L + |w| + 1$ compatible-\$ with factors of w_1, but there are several such words compatible-\$ with the entire w_2. The number of these words is denoted by N_w and equals $L^{|V|-1}$, if w contains no $\diamond$ symbol, or $L^{|V|-1}|w|_\diamond^{|V|-1}$, otherwise. N_w can be computed in polynomial time, starting from w.

From the above considerations it follows that $\mathrm{Comp}(w_2, \$) - \mathrm{Comp}(w_1, \$) = (|V| - 1)^L - X_{w,L} + \sum_{k \in \{L+1,\ldots,L+|w|\}} N_{k,w} + N_w$. Since all the numbers $N_{k,w}$, for $k \in \{L+1, \ldots, L+|w|\}$, can be computed in polynomial time, as well as N_w, and if we assume that the function Comp computes the solutions to Problem 7 in polynomial time, we obtain that $X_{w,L}$ can be computed in polynomial time. Since Problem 5 is #P-complete, it follows that Problem 7 is #P-complete, as well. □

We conclude by proving that counting all the square full words which are compatible with factors of a partial word is also a hard counting problem.

Problem 8. *Given a partial word w, over the alphabet V, with $|V| \geq 2$, count the words $x \in V^*$, with $0 < |x| \leq |w|$ and $x = vv$ for some $v \in V^*$, compatible with a factor of w.*

According to the results in [8] one can identify all the 2-repetitions in a partial word w in $\mathcal{O}(|w|^2)$. Thus one can decide efficiently the existence of a square vv which is compatible with at least one factor of w. Nevertheless, unless w has only $\diamond$ symbols, there exists also a square that is not compatible with any of its factors (the arguments are similar with those used in the case of Problems 6 and 7). Therefore, Problem 8 cannot be associated canonically with a NP-complete problem. However, this problem is also hard for the class #P.

Proposition 7. *Problem 8 is #P-complete.*

Proof. It is straightforward to construct a nondeterministic polynomial Turing machine having as many accepting paths as the number of words $x \in V^*$, with $0 < |x| \leq |w|$ and $x = vv$ for some $v \in V^*$, compatible with a factor of w. Thus Problem 8 is in #P. It remains to show that it is also complete for this class.

We will finish this proof by giving a reduction from a slightly modified version of Problem 3. Assume that there exists a function Squares(w) that computes efficiently (on a deterministic Turing machine) the solution of Problem 8, having as input data the partial word w. Further, consider an input instance of the Problem 3: $S = \{w_1, w_2, \ldots, w_k\}$, with $k > 3$, is a list of partial words of length L, over the alphabet $V = \{0, 1\}$; we are interested in computing all the full words over $V' = V \cup \{\clubsuit, \heartsuit, \perp, \spadesuit\}$, which are compatible with at least one word of the list (this version of Problem 3 can be shown to be $\#P$-complete in the exact same way as in the case of the initial problem).

Starting from the list S we can construct, in deterministic polynomial time, the partial word $w = \heartsuit w_1 \clubsuit \heartsuit w_1 \clubsuit \$^{kL+1} \perp \heartsuit w_2 \clubsuit \heartsuit w_2 \clubsuit \$^{kL+2} \perp^2 \ldots \heartsuit w_k \clubsuit \heartsuit w_k \clubsuit \$^{kL+k} \perp^k \spadesuit^{2k^2L^2} \diamond^{2L+2}$.

Next we analyze what square full words can be compatible with factors of w:

a. All the words $x = vv$ from V'^{2k}, with $k \leq L+1$, are compatible with factors of the form $\diamond^{2k}$ of w. The number of these words is $N_1 = \sum_{k=1,L+1} 6^k$.
b. If v is a word of length L, compatible with one of the words $w_1, w_2, \ldots, w_k$, then $\clubsuit v \heartsuit \clubsuit v \heartsuit$ is a square compatible with a factor of w. The number of such squares equals the number of words over V' compatible with at least a word from the list S, denoted here by $X_{S,V'}$.
c. All the words of the form $\spadesuit^{2r}$, with $k^2L^2 + L + 1 \geq r > L+1$, are squares compatible with factors of w. Their number is, clearly, $N_2 = \lceil (k^2L^2)/2 \rceil$.
d. All the words of the form vv, where $v = \spadesuit^r x$, $L+1 < r + |x| < 2L+2$, $|x| > 0$, and x does not contain only $\spadesuit$ symbols, are squares compatible with factors of w (namely factors of the form $\spadesuit^r \diamond^{|x|}$). The number of such words is $N_3 = \sum_{0<r,0<t,L+1<t+r<2L+2} (5^t + \sum_{1 \leq m \leq t-1} (\binom{m}{t} 5^{t-m}))$.
e. Any other word of length greater than $2L+2$ contained in w is not a square (we will show this a little later). Moreover, the sets of squares described in the previous four claims are each two disjoint.

Clearly, the numbers N_1, N_2, and N_3 can be computed in polynomial time. Also, if the function Square() outputs its value in polynomial time, it follows that one can find the number $X_{S,V'}$ in polynomial time; indeed, $X_{S,V'} =$Square$(w) - N_1 - N_2 - N_3$. But this shows that Problem 8 is harder than Problem 3. Therefore, Problem 8 is #P-complete.

It remains only to show that the first part of the claim **e** above is true. For this, let vv be a square, compatible with a factor of w, other than any of the squares mentioned in the claims a, b, c, d. There exists a factor x_1x_2 of w such that $x_1 \subset v$ and $x_2 \subset v$. Let y be the starting symbol of x_1 and z be the starting symbol of x_2. There are several cases to be analyzed:

1. $y = \heartsuit$ and $z = \heartsuit$. If $x_1 = \heartsuit w_i \clubsuit \heartsuit w_i \clubsuit \ldots \$ \perp^{j-1}$ it follows that $x_2 = \heartsuit w_j \clubsuit \heartsuit w_j \clubsuit \ldots$; but this is impossible due to the fact that the word x_2 contains more \$ symbols, after the second $\clubsuit$, than x_1 contains after the second $\clubsuit$, thus one of the \$ symbols in x_2 would be compatible with a $\perp$ symbol from x_1. If $x_1 = \heartsuit w_i \clubsuit \heartsuit w_i \clubsuit \ldots \$ \perp^{j-1} \heartsuit w_j \clubsuit$ it follows that $x_2 = \heartsuit w_j \clubsuit \$ \ldots$; again, this is impossible due to the fact that the second $\heartsuit$ symbol of x_1 would be compatible with a \$ symbol from x_2. If $x_1 = \heartsuit w_i \clubsuit \$ \ldots \$ \perp^{j-1} \heartsuit w_j \clubsuit$ it follows that $x_2 = \heartsuit w_j \clubsuit \$ \ldots$; this is a contradiction, again, because a \$ symbol of x_2 would be compatible with a $\perp$ symbol of x_1. Finally, if $x_1 = \heartsuit w_i \clubsuit \$ \ldots \$ \perp^{j-1}$ it follows that $x_2 = \heartsuit w_j \clubsuit \heartsuit w_j \clubsuit \$ \ldots$; this is contradiction, because the first \$-symbol from x_1 would be compatible with a $\heartsuit$ symbol from x_2. Clearly, no other case exists.
2. $y = \heartsuit$ and $z = \diamond$. If $x_1 = \heartsuit w_i \clubsuit \heartsuit w_i \clubsuit \ldots \$ \perp^{j-1} \heartsuit u$ it follows that $x_2 = \diamond u' \clubsuit \heartsuit w_j \clubsuit \ldots$, where $u \diamond u' = w_j$; this leads to a contradiction, since the first $\perp$ symbol in x_1 would be compatible with a \$ symbol from x_2. If $x_1 = \heartsuit w_i \clubsuit \heartsuit w_i \clubsuit$

$\ldots \$\perp^{j-1}\heartsuit w_j\clubsuit\heartsuit u$ it follows that $x_2 = \diamond u'\clubsuit\$\ldots$, where $u\diamond u' = w_j$; in this case, the first $\perp$ from x_2 would be compatible with a \$ symbol from x_1, a contradiction. Finally, x_1 cannot contain a $\spadesuit$ since it would imply that x_1 contains all the $\spadesuit$ symbols, and, thus, it would be longer than x_2. This completes the analysis of this case.
3. The cases when $y \in \{0, 1, \clubsuit, \perp, \$\}$ (and all the possible corresponding assignments for z) can be treated similarly to the above. Also, y cannot be $\spadesuit$, since it would imply that vv is one of the words analyzed in the claims a, b, c, d.
4. $y = \diamond$ and $z = \diamond$.

– If $x_1 = \diamond u_1\clubsuit\heartsuit w_i\clubsuit\ldots\$\perp^{j-1}\heartsuit u_2$ it follows that $x_2 = \diamond u_3\clubsuit\heartsuit w_j\clubsuit\ldots$, where there exists u_0 such that $u_0 \diamond u_1 = w_i$ and $u_2 \diamond u_3 = w_j$. If $|u_1| = |u_3|$ we will reach a contradiction because the first $\perp$ symbol of x_1 would be compatible with a \$ symbol of x_2. If $|u_1| < |u_3|$ the second $\clubsuit$ symbol of x_2 would be compatible with a \$ symbol from x_1, again a contradiction. Finally, if $|u_3| < |u_1|$, the second $\clubsuit$ of x_1 would be compatible with a \$ symbol of x_2, also a contradiction.
– If $x_1 = \diamond u_1\clubsuit\heartsuit w_i\clubsuit\ldots\$\perp^{j-1}\heartsuit w_j\clubsuit\heartsuit u_2$ it follows that $x_2 = \diamond u_3\clubsuit\$\ldots$, where there exists u_0 such that $u_0\diamond u_1 = w_i$ and $u_2\diamond u_3 = w_j$. We obtain a contradiction because the second $\clubsuit$ of x_1 would be compatible with a \$ symbol of x_2.
– All the other cases can be treated analogously, and they all lead either to contradiction, either to the case when vv is a word that was already considered in the claims a, b, c, d.

By the analysis performed above it follows that claim **e** is correct. Thus our proof is complete. □

We conjecture that the proof we presented above can be generalized to show that counting all the distinct k-repetitions-full-words ($k \geq 3$), compatible with at least a factor of a given partial word, is also a #P-complete problem. Again, according to [8,10], this would be an example of a hard counting problem that cannot be associated canonically with a NP-complete problem.

References

1. Allouche, J.P., Shallit, J.: Automatic Sequences: Theory, Applications, Generalizations. Cambridge University Press, Cambridge (2003)
2. Berstel, J., Boasson, L.: Partial words and a theorem of Fine and Wilf. Theoretical Computer Science 218, 135–141 (1999)
3. Blanchet-Sadri, F.: Algorithmic Combinatorics on Partial Words. Chapman & Hall/CRC Press (2008)
4. Blanchet-Sadri, F., Anavekar, A.R.: Testing primitivity on partial words. Discrete Applied Mathematics 155(3), 279–287 (2007)
5. Blanchet-Sadri, F., Mercaş, R., Rashin, A., Willett, E.: An answer to a conjecture on overlaps in partial words using periodicity algorithms. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 188–199. Springer, Heidelberg (2009)
6. Blanchet-Sadri, F., Mercaş, R., Scott, G.: Counting distinct squares in partial words. In: Csuhaj-Varju, E., Esik, Z. (eds.) Proceedings of AFL, pp. 122–133 (2008)

7. Blanchet-Sadri, F., Schwartz, J., Stich, S.: Partial subword complexity (submitted 2010)
8. Diaconu, A., Manea, F., Tiseanu, C.: Combinatorial queries and updates on partial words. In: Gębala, M. (ed.) FCT 2009. LNCS, vol. 5699, pp. 96–108. Springer, Heidelberg (2009)
9. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-completeness. W.H. Freeman, New York (1979)
10. Manea, F., Mercaş, R.: Freeness of partial words. Theoretical Computer Science 389(1-2), 265–277 (2007)
11. Valiant, L.: The complexity of computing the permanent. Theoretical Computer Science 8, 189–201 (1979)

Exact Analysis of Horspool's and Sunday's Pattern Matching Algorithms with Probabilistic Arithmetic Automata

Tobias Marschall and Sven Rahmann

Bioinformatics for High-Throughput Technologies,
Algorithm Engineering, Computer Science XI, TU Dortmund, Germany
{tobias.marschall,sven.rahmann}@tu-dortmund.de

Abstract. We define *deterministic arithmetic automata* (DAAs) and connect them to a framework called *probabilistic arithmetic automata* (PAAs) [9]. We use DAAs and PAAs to compute the entire *exact* probability distribution (in contrast to, e.g., asymptotic expectation and variance) of the number X_ℓ^p of text characters accessed by the Horspool or Sunday pattern matching algorithms when matching a fixed pattern p against a random text of length ℓ. The random text model can be quite general, from simple uniform models to higher-order Markov models or hidden Markov models (HMMs). We develop several alternative constructions with different state spaces of the automata, leading to alternative time and space complexities for the computations. To our knowledge, this is the first time that suffix-based pattern matching algorithms are analyzed exactly. We present (perhaps surprising) exemplary results on short patterns and moderate text lengths. Our results easily generalize to any search-window based pattern matching algorithm.

1 Introduction

The basic pattern matching problem is to find all occurrences of a *pattern* string in a (long) *text* string, with few character accesses. Let ℓ be the text length and m be the pattern length. The well-known Knuth-Morris-Pratt algorithm [6] reads each text character exactly once from left to right after preprocessing the pattern and needs a total of $\Theta(\ell+m)$ character accesses. In contrast, the Boyer-Moore [3], Horspool [5] and Sunday [14] algorithms move a length-m search window across the text and first compare its *last* character to the last character of the pattern. This often allows to move the search window by more than one position (at best, by m positions if the last window character does not occur in the pattern at all), for a best case of $\Theta(m+n/m)$ but a worst case of $\Theta(m+mn)$ character accesses. The worst case can be improved to $\Theta(m+n)$, but this makes the code more complicated and is seldom useful in practice. The Horspool algorithm and the variant of Sunday can be seen as modifications of the Boyer-Moore algorithm that are simpler to implement and additionally perform better in practice [11].

A question that has apparently so far not been investigated is about the exact probability distribution of the number of required character accesses X_ℓ^p

when matching a given pattern p against a random text of finite length ℓ (non-asymptotic case), even though related questions have been answered in the literature. For example, [1,2] analyze the expected value of X_ℓ^p. In [8] it is shown that X_ℓ^p is asymptotically normally distributed for i.i.d. texts, and [13] extends this result to Markovian text models.

In contrast to these results, we use a general framework called *probabilistic arithmetic automata* (PAAs), recently introduced at CPM'08 [9], to compute the exact distribution of X_ℓ^p. In our framework, the random text model can be quite general, from very simple i.i.d. uniform models to high-order Markov models or HMMs. The approach is applied exemplarily to the Horspool and Sunday algorithms here, but easily generalizes to all search-window-based pattern matching algorithms.

In the next section, we introduce notation and text models, give a brief review of the Horspool and Sunday algorithms, define *deterministic arithmetic automata* (DAAs), summarize the PAA framework with its generic algorithms, and connect DAAs to PAAs. Sections 3 and 4 present two alternative PAA constructions that allow to compute the entire distribution of X_ℓ^p. In Section 5, we argue that the state space of the PAA construction from Section 3 can be considerably reduced. Exemplary results on the comparison of the Horspool and Sunday algorithms in terms of their running time distributions can be found in Section 6, where we also give a concluding discussion.

2 Preliminaries

Notation. Both pattern and text are over a finite alphabet Σ. The pattern $p = p[0] \dots p[m-1]$ is of length m, a (concrete) text s is of length ℓ. We also consider a stochastic process $S = S_0 S_1 S_2 \dots$ according to a random text model (see next paragraph), whose prefix of length ℓ is of interest to us. Let $\mathbb{P}(s) := \mathbb{P}(S_0 \dots S_{|s|-1} = s)$ be the probability of observing s as the length-$|s|$ prefix of S, and let $\xi^p(s)$ be the number of text character accesses of the Horspool (resp. Sunday) algorithm when matching p against s. Let $X_\ell^p := \xi^p(S_0 \dots S_{\ell-1})$ be the corresponding random variable for random texts of length ℓ. We are interested in the exact probability distribution of X_ℓ^p, which we denote by $\mathcal{L}(X_\ell^p)$.

Text models. A text model $\mathbb{P}$ is the distribution of a stochastic process $(S_i)_{i\in\mathbb{N}}$, where each S_i takes values in Σ. It can be specified by giving the probabilities $\mathbb{P}(S_0 \dots S_{|s|-1} = s)$ for all $s \in \Sigma^*$. We only consider finite-memory models, specifically i.i.d. or Markovian texts. These can by defined by a tuple (Y, φ, y_0), where Y is a finite state space, $y_0 \in Y$ a start state, and $\varphi : Y \times \Sigma \times Y \to [0,1]$ is such that $\varphi(y, a, y')$ is the probability of going from state y to state y' and thereby generating the letter a. Therefore $\sum_{(a,y')\in\Sigma\times Y} \varphi(y, a, y') = 1$ for all $y \in Y$. The model given by (Y, φ, y_0) therefore generates a random text by going from state to state and emitting a character at each transition. We refer to both the tuple (Y, φ, y_0) and the induced probability measure $\mathbb{P}$ as *text models*. Similar text models are used in [7] and there called probability transducers.

For an i.i.d. model, we set $Y = \{\circ\}$ and $\varphi(\circ, a, \circ) = p_a$ for each $a \in \Sigma$, where p_a is the occurrence probability of letter a (and $\circ$ may be interpreted as an empty string or context). For a Markovian text model of order k, the distribution of the next character depends only on the k preceding characters (fewer at the beginning); thus we set $Y := \bigcup_{i=0}^{k} \Sigma^i$. For all contexts $y \in \Sigma^k$ and $y' = y[1 \dots |y|-1]a$, the conditional letter-a follow-up probabilities are given by $\varphi(y, a, y')$. It is clear how to handle the shorter contexts at the beginning. This notion of text models also covers variable order Markov chains as introduced in [12] and HMMs.

Algorithms of Horspool and Sunday. The idea of Horspool is as follows. We maintain a search window w of length $m = |p|$ that initially starts at position 0 in the text s, such that its rightmost character is at position $t = m - 1$. The variable t grows in the course of the algorithm; we always have $w = s[(t-m+1) \dots t]$. First, the rightmost characters of window and pattern are compared; that means, $a := w[m-1] = s[t]$ is compared with $p[m-1]$. If they match, the remaining $m-1$ characters are compared until either the first mismatch is found or an entire match has been verified. This comparison can happen right-to-left, left-to-right, or in an arbitrary order that may depend on p. In our analysis, we focus on the right-to-left case for concreteness, but the modifications for the other cases are trivial. In any case, the rightmost window character a is used to determine how far the window can be shifted for the next iteration: $t \mapsto t + \mathit{shift}^p(a)$. The shift-function ensures that no match can be missed by moving the window such that a becomes aligned to the rightmost a in p (not considering the last position). If a does not occur in p (or only at the last position), it is safe to shift by m positions. Formally,

$$\begin{aligned} \mathit{right}^p(a) &:= \max\left[\{i \in \{0, \dots, m-2\} : p[i] = a\} \cup \{-1\}\right], \\ \mathit{shift}^p(a) &:= (m-1) - \mathit{right}^p(a). \end{aligned}$$

Sunday's algorithm is similar, except that the first character *after* the current window w is used to determine the shift. This leads to the modified definitions

$$\begin{aligned} \mathit{right}^p(a) &:= \max\left[\{i \in \{0, \dots, m-1\} : p[i] = a\} \cup \{-1\}\right], \\ \mathit{shift}^p(a) &:= m - \mathit{right}^p(a). \end{aligned}$$

For both algorithms and fixed p, we write *right* and *shift*. This causes no confusion, as it is clear from the context to which algorithm we are referring.

For concreteness, we state Horspool's algorithm and how we count text character accesses as pseudocode in Algorithm 1. We assume that characters are compared right-to-left. Note that after a shift, even when we know that a now matches its corresponding pattern character, the corresponding position is compared again and counts as a text access. Otherwise the additional bookkeeping would make the algorithm more complicated; this is not worth the effort in practice. The lookup in the *shift*-table does not count as an additional access, since we can remember $\mathit{shift}[a]$ as soon as the last window character has been read.

Algorithm 1. HORSPOOL

```
Input: text $s \in \Sigma^*$, pattern $p \in \Sigma^m$
Output: pair (number $occ$ of occurrences of $p$ in $s$, number $cost$ of accesses to $s$)
 1: pre-compute table $shift[a]$ for all $a \in \Sigma$
 2: $(occ, cost) \leftarrow (0, 0)$
 3: $t \leftarrow m - 1$
 4: while $t < |s|$ do
 5:    $i \leftarrow 0$
 6:    while $i < m$ do
 7:       $cost \leftarrow cost + 1$
 8:       if $s[t-i] \neq p[(m-1)-i]$ then break
 9:       $i \leftarrow i + 1$
10:    if $i = m$ then $occ \leftarrow occ + 1$
11:    $t \leftarrow t + shift[s[t]]$
12: return $(occ, cost)$
```

Arithmetic Automata. Before we review the framework of probabilistic arithmetic automata (PAAs) introduced in [9], which we use to compute the sought distribution $\mathcal{L}(X_\ell^p)$, we first define another class of automata "half way" between deterministic finite automata (DFAs) and PAAs.

Definition 1 (Deterministic Arithmetic Automaton, DAA). *A deterministic arithmetic automaton is a tuple $\mathcal{D} = \big(Q, q_0, \Sigma, \delta, N, n_0, E, (\varepsilon_q)_{q\in Q}, (\theta_q)_{q\in Q}\big)$, where Q is a finite set of states, $q_0 \in Q$ is the start state, Σ is a finite alphabet, $\delta : Q \times \Sigma \to Q$ is called transition function, N is a finite or countable set of values, $n_0 \in N$ is called the start value, E is a finite set of emissions, $\varepsilon_q \in E$ is the emission associated to state q, and $\theta_q : N \times E \to N$ is a binary operation associated to state q.*

Informally, a DAA starts with the state-value pair (q_0, n_0) and reads a sequence of symbols from Σ. Being in state q with value v, upon reading $a \in \Sigma$, the DAA performs a state transition to $q' := \delta(q, a)$ and updates the value to $v' := \theta_{q'}(v, \varepsilon_{q'})$ using the operation and emission of the new state q'. Formally, we define the associated joint transition function

$$\hat{\delta} : (Q \times N) \times \Sigma \to (Q \times N), \qquad \hat{\delta}\big((q, v), a\big) := \big(\delta(q, a)\,,\, \theta_{\delta(q,a)}(v, \varepsilon_{\delta(q,a)})\big).$$

As usual, we extend this definition inductively to Σ^* by $\hat{\delta}\big((q, v), \circ\big) := (q, v)$ for the empty string $\circ$ and $\hat{\delta}\big((q, v), xa\big) := \hat{\delta}\big(\hat{\delta}((q, v), x), a\big)$ for all $x \in \Sigma^*$ and $a \in \Sigma$.

When $\hat{\delta}\big((q_0, n_0), x\big) = (q, v)$ for some $q \in Q$, we say that $\mathcal{D}$ computes value v for input x and define $value_{\mathcal{D}}(x) := v$.

For each state q, the emission ε_q is fixed and could be dropped from the definition of DAAs. In fact, one could also dispense with values and operations entirely and define a DFA over state space $Q \times N$, performing the same operations as a DAA. However, we intentionally include values, operations, and emissions to emphasize the connection to PAAs. The reason is that DAAs become more interesting when we allow the emissions in each state and the transitions between states to be probabilistic instead of deterministic, as follows.

Definition 2 (Probabilistic Arithmetic Automaton, PAA, [9]). *A* probabilistic arithmetic automaton *is a tuple* $\mathcal{P} = \big(Q, q_0, T, N, n_0, E, e = (e_q)_{q \in Q}, \theta = (\theta_q)_{q \in Q}\big)$*, where* Q*,* q_0*,* N*,* n_0*,* E *and* θ *have the same meaning as for a DAA, each* e_q *is a state-specific probability distribution on the emissions* E*, and* T *is a* $Q \times Q$ *stochastic matrix, such that* $T_{q,q'}$ *specifies the probability of a transition from state* q *to state* q'*.*

A PAA induces three stochastic processes: (1) the state process $(Q_t)_{t \in \mathbb{N}}$ *with values in* Q*, (2) the emission process* $(Z_t)_{t \in \mathbb{N}}$ *with values in* E*, and (3) the value process* $(V_t)_{t \in \mathbb{N}}$ *with values in* N *such that* $V_0 :\equiv n_0$ *and* $V_t := \theta_{Q_t}(V_{t-1}, Z_t)$*.*

In [9], PAAs are used to compute pattern occurrence count distributions. Applications in biological sequence analysis include the exact computation of p-values of sequence motifs [10], and the determination of seed sensitivity for pairwise sequence alignment algorithms based on filtering [4], among others. We now restate the PAA recurrences from [9] to compute the state-value distribution after t steps. For the sake of a shorter notation, we define $f_t(q, v) := \mathbb{P}(Q_t = q, V_t = v)$. Since we are generally only interested in the value distribution, note that it can be obtained by marginalization: $\mathbb{P}(V_t = v) = \sum_{y \in Q} f_t(q, v)$.

Lemma 1 (State-value recurrence, [9]). *The state-value distribution is given by* $f_0(q, v) = 1$ *if* $q = q_0$ *and* $v = n_0$*, and* $= 0$ *otherwise. For* $t \geq 0$*,*

$$f_{t+1}(q, v) = \sum_{q' \in Q} \sum_{(v', z) \in \theta_q^{-1}(v)} f_t(q', v') \cdot T(q', q) \cdot e_q(z), \tag{1}$$

where $\theta_q^{-1}(v)$ *denotes the inverse image set of* v *under* θ_q*.*

The recurrence in Lemma 1 resembles the Forward recurrences known from HMMs. While N may be infinite, only a finite set of values N_t can be attained during a finite number t of steps. While always $|N_t| \leq |E|^t$, it can be much smaller, often $O(t)$. To compute the state-value distribution up to step t in this way takes $O(t \cdot |Q|^2 \cdot |N_t| \cdot |E|)$ arithmetic operations. The space requirement is $O(|Q| \cdot |N_t|)$. Using a doubling technique [9], it can alternatively be computed with $O(\log t \cdot |Q|^3 \cdot |N_t|^3)$ arithmetic operations using $O(|Q|^2 \cdot |N_t|^2)$ space.

We now formally state how to convert a DAA into a (restricted) PAA, where each emission distribution is deterministic (assigning probability 1 to a particular value), given a random text model (i.i.d., Markovian, or a HMM).

Lemma 2 (DAA + Text model → PAA). *Let* (Y, φ, y_0) *be a text model and* $\mathcal{D} = \big(Q^{\mathcal{D}}, q_0^{\mathcal{D}}, \Sigma, \delta, N, n_0, E, (\varepsilon_q)_{q \in Q^{\mathcal{D}}}, (\theta_q^{\mathcal{D}})_{q \in Q^{\mathcal{D}}}\big)$ *be a DAA. Define state space* $Q := Q^{\mathcal{D}} \times Y$*, start state* $q_0 := (q_0^{\mathcal{D}}, y_0)$*, (deterministic) emission probability vectors* $e_{(q,y)}$ *by* $e_{(q,y)}(\varepsilon_q) := 1$ *and* $e_{(q,y)}(z) := 0$ *for* $z \neq \varepsilon_q$*, operations* $\theta_{(q,y)} := \theta_q^{\mathcal{D}}$ *for all* $(q, y) \in Q = Q^{\mathcal{D}} \times Y$*, and transition probabilities* $T_{(q,y),(q',y')} :=$

$\sum_{a\in\Sigma:\,\delta(q,a)=q'} \varphi(y,a,y')$. *States with zero probability of being reached from* q_0 *may be omitted from* Q *and* T. *Then* $\mathcal{P} = \big(Q, q_0, T, N, n_0, E, (e_q)_{q\in Q}, (\theta_q)_{q\in Q}\big)$ *is a PAA with* $\mathcal{L}(V_\ell) = \mathcal{L}\big(value_{\mathcal{D}}(S_0 \dots S_{\ell-1})\big)$, *where* S *is a random text according to the text model* (Y, φ, y_0). *For such a PAA, the state-value distribution up to step* t *can be computed with* $O(t \cdot |Q| \cdot |\Sigma| \cdot |N|)$ *operations.*

Proof (Idea). It is obvious that $\mathcal{P}$ defines a PAA. For $\ell = 0$, the proof of $\mathcal{L}(V_\ell) = \mathcal{L}\big(value_{\mathcal{D}}(S_0 \dots S_{\ell-1})\big)$ is obvious. For $\ell > 0$, we inductively show that

$$\sum_{y\in Y} f_\ell\big((q,y),v\big) = \sum_{x\in\Sigma^\ell:\,\hat{\delta}((q_0^{\mathcal{D}},n_0),x)=(q,v)} \mathbb{P}(x) \qquad \forall\, q \in Q^{\mathcal{D}},\ v \in N.$$

This follows from the definition of the PAA transition probabilities.

The complexity follows from Lemma 1, noting that $|Q|^2$ can be replaced by $|Q||\Sigma|$, since each state in the PAA has outdegree at most $|\Sigma|$, and by omitting $|E|$, since emissions are deterministic in each state. □

3 Basic PAA Construction

We first present a straightforward PAA construction to compute $\mathcal{L}(X_\ell^p)$, the distribution of text character accesses during the Horspool algorithm when matching $p \in \Sigma^m$ against a random text of length ℓ. The strategy is to simulate the Horspool algorithm with a DAA that counts both the reached position in the text and the number of character accesses so far. The DAA is then converted into a PAA using Lemma 2.

Consider a substring $w \in \Sigma^m$ of the text, corresponding to the current search window, ending at text position t. Its last character $a := w[m-1] = s[t]$ determines the shift $\mathit{shift}(a)$. For convenience, let us therefore define $\mathit{shift}(w) := \mathit{shift}(w[m-1])$ for the Horspool algorithm. All of w determines the number $\xi_{\llcorner}^p(w)$ of text character accesses necessary to compute the shift and to verify or disprove $w = p$ with the Horspool inner loop (Algorithm 1, lines 6–9). Explicitly, still assuming right-to-left comparison within the window, we have the following per-window access costs.

$$\xi_{\llcorner}^p(w) = \begin{cases} m & \text{if } p = w, \\ 1 + \min\{i : 1 \le i \le m,\ p[m-i] \ne w[m-i]\} & \text{otherwise.} \end{cases}$$

It is easy to modify this function to accommodate different comparison orders; in each case w entirely determines how many characters must be accessed and compared before reaching a decision.

To summarize, when the right end of the search window has reached text position t with c character accesses so far and examines the current window $w = s[t-m+1 \dots t]$, we afterwards obtain a total of $c + \xi_{\llcorner}^p(w)$ character accesses and move to text position $t + \mathit{shift}(w)$. Therefore we can define the following DAA $\mathcal{D} = \big(Q, q_0, \Sigma^{\mathcal{D}}, \delta, N, n_0, E, \varepsilon = (\varepsilon_q)_{q\in Q}, \theta = (\theta_q)_{q\in Q}\big)$ for text length ℓ:

- $Q := \Sigma^m \cup \{q_0\}$, where q_0 is the start state and any other state corresponds to the current window content.
- $N := \{m-1, m, \ldots, \ell-1, \bullet\} \times \{0, \ldots, \ell m\}$, where $n_0 = (m-1, 0)$. Value $v = (t, c)$ corresponds to the current window ending at text position t (with $\bullet$ indicating that the text has ended), having accumulated c text character accesses so far.
- $E := \{1, \ldots, m\} \times \{1, \ldots, m\}$; the deterministic emission in state $q \in \Sigma^m$ is $\varepsilon_q = \big(\mathit{shift}(q), \xi^p_{\llcorner}(q)\big)$.
- The operation θ_q in each state is essentially addition on N with one exception: To freeze the character access count when crossing position $\ell-1$, we use the special values $(\bullet, c) \in N$. Therefore, for all q, we define $\theta_q\big((t,c),(t',c')\big)$ as $(t+t', c+c')$ if $t \neq \bullet$ and $t+t' < \ell$, as $(\bullet, c+c')$ if $t \neq \bullet$ and $t+t' \geq \ell$, and as $(\bullet, c)$ if $t = \bullet$.
- $\Sigma^{\mathcal{D}} := \cup_{i=1}^{m} \Sigma^i$ (Horspool): When leaving state q, exactly $\mathit{shift}(q)$ new text characters enter the window; therefore the DAA must read a block $b \in \Sigma^{\mathit{shift}(q)}$ characters as a single symbol. Accordingly $\delta(q, b) = q[\mathit{shift}(q)..]b$, where $q[i..]$ denotes the suffix of q starting at position i. Note that a text model on $\Sigma^{\mathcal{D}}$ is induced by a text model on Σ. For $b \in \Sigma^{\mathcal{D}} \setminus \Sigma^{\mathit{shift}(q)}$, the transition $\delta(q, b)$ is not defined, which is not a problem, since the according transition probabilities are set to zero in the constructed PAA.

Lemma 3. *The DAA $\mathcal{D}^p$ for pattern p, as defined above, satisfies $\mathit{value}_{\mathcal{D}^p}(s) = \xi^p(s)$ for all $s \in \Sigma^*$.*

Proof. Straightforward by induction on the number of windows used with pattern p on text s. □

Applying Lemma 2, we convert the DAA into a PAA, replacing transition labels by transition probabilities, to obtain $\mathcal{L}\big(\xi^p(s)\big)$ over all $s \in \Sigma^\ell$, that is, $\mathcal{L}(X^p_\ell)$. In summary, this leads to the following theorem.

Theorem 1. *The above DAA/PAA construction allows to compute $\mathcal{L}(X^p_\ell)$ for Horspool with $O(m\,\ell^3\,|\Sigma|^{2m})$ arithmetic operations in $O(m\,\ell^2\,|\Sigma|^m)$ space, assuming that the text model is i.i.d. or a Markov chain of order $\leq m$.*

Proof. We have $|Q| = \Sigma^m$ and $|N| = O(m\ell^2)$. Emissions are deterministic. The claim now follows from Lemma 1. □

For the Sunday algorithm, we need to extend the current search window one position to the right for a length of $m+1$ and to access an additional character to determine the shift; it is straightforward to modify $\mathit{shift}(w)$, $\xi^p_{\llcorner}(w)$, state space Q and emission set E. Theorem 1 holds with m replaced by $m+1$.

4 An Alternative PAA Construction

So far a step of the automaton corresponded to examining a search window. Now, however, we construct a DAA where a step corresponds to processing a single text character, and we process the text from left to right (in contrast to the actual working of Horspool's algorithm). Therefore the state has to remember where the window starts relative to the last read character. We define state space Q and value space N as follows:

- $Q := (\{0, \ldots, m-1\} \times \Sigma) \cup \{q_0\}$, where q_0 is the start state,
- $N := \mathbb{N} \times \{0, \ldots, m-1\}^{m-1}$ with a start value of $n_0 = \big(0, (0, \ldots, 0)\big)$.

The defined spaces are endowed with the following semantics. After n steps, a state of (k, a) and a value of $\big(c, (x_0, \ldots, x_{m-1})\big)$ indicate that we have done c comparisons so far, the last read character was a, and the current position n corresponds to position k in the current window (i.e., the current window starts at position $n-k$ in the text). The next shift is unknown until the last character of the current window has been read (k reaches $m-1$). Therefore, we store information on the comparisons needed for every possible next window using $(x_0, \ldots, x_{m-1})$, such that x_j is defined as the number of characters that will not be accessed when next scanning a window that starts at position $n-k+j$. This DAA does not need emissions; we set $E = \{0\}$ and $e_q(0) = 1$ for all $q \in Q$ and omit emissions by writing $\theta_q(\cdot) := \theta_q(\cdot, 0)$.

The described semantics lead to the following operation for a state $q = (k, a)$: We define two auxiliary functions *update* and *move*. The *update* function updates $\mathbf{x} = (x_0, \ldots, x_{m-1})$ such that the information gained by reading $a \in \Sigma$ is incorporated;

$$\mathit{update}_k : \big(\mathbf{x} = (x_0, \ldots, x_{m-1}), a\big) \mapsto \mathbf{x}' = (x'_0, \ldots, x'_{m-1}), \tag{2}$$

where

$$x'_j = \begin{cases} k - j & \text{if } j < k \text{ and } a \neq p[k-j], \\ x_j & \text{otherwise,} \end{cases}$$

that means, if a mismatch at position k in the window is found, we know that (at least) k characters need (and will) not be compared. The *move* function "shifts" the contents of the vector:

$$\mathit{move} : \big((x_0, \ldots, x_{m-1}), \ell\big) \mapsto (x_\ell, \ldots, x_{m-1}, \underbrace{0, \ldots, 0}_{\ell}) .$$

Now we can define the DAA operations $\theta_{(k,a)}$:

$$\theta_{(k,a)} : (c, \mathbf{x}) \mapsto \begin{cases} \big(c, \mathbf{z}\big) & \text{if } k < m-1 \,, \\ \big(c + m - z_0, \mathit{move}(\mathbf{z}, \mathit{shift}(a))\big) & \text{if } k = m-1 \,, \end{cases} \tag{3}$$

where $\mathbf{z} = (z_0, \ldots, z_{m-1}) := \mathit{update}_k(\mathbf{x}, a)$. The DAA transition function δ is defined as

$$\delta : (q, a) \mapsto \begin{cases} (0, a) & \text{if } q = q_0 \,, \\ (k+1, a) & \text{if } q = (k, a') \text{ and } k < m-1 \,, \\ \big(k - \mathit{shift}(a') + 1, a\big) & \text{if } q = (k, a') \text{ and } k = m-1 \,. \end{cases} \tag{4}$$

By formally proving that the operations and transitions capture the above semantics, we obtain the following theorem, whose proof we omit due to space constraints.

Theorem 2. *Given a pattern p, let $\mathcal{D} = (Q, q_0, \Sigma, \delta, N, n_0, E, \varepsilon, \theta)$ be the DAA constructed for p as detailed above. Then, $\mathcal{D}$ correctly computes the number of accesses to a given text s done by Horspool's algorithm as given in Algorithm 1, that means $\xi^p(s) = c$, where $value_{\mathcal{D}}(s) = (c, \boldsymbol{x})$.*

Theorem 3. *With a PAA constructed from the above DAA using Lemma 2, assuming a Markov text model of at most first order, $\mathcal{L}(X_\ell^p)$ for Horspool's algorithm can be computed with $O(\ell^2 \cdot m^2 \cdot m! \cdot \Sigma^2)$ arithmetic operations.*

Proof. The size of the relevant value set N for text length ℓ can be bounded by ℓm (first component) times $m!$ (second component), since $0 \leq x_j \leq m - j - 1$ Further, we have $|Q| = O(m \cdot |\Sigma|)$ and $|E| = O(1)$. The claim again follows from Lemma 1. □

Sunday's Algorithm. The DAA for Sunday's algorithm is similar to the one for Horspool's algorithm. In Sunday's algorithm, the window can be positioned completely before or completely after the currently processed character. We therefore define $Q := (\{-1, \dots, m\} \times \Sigma) \cup \{q_0\}$. Transition function and operations can be adjusted straightforwardly. Theorem 3 holds in the same way.

5 Reducing the State Space

While the basic construction in Section 3 is straightforward, the fact that $|Q| = |\Sigma|^m$ limits investigations to short patterns and/or small alphabets. Here we show that the Horspool and Sunday algorithm can be simulated with a much smaller state space, for whose size we conjecture a bound. We now adopt the convention that after comparing the last window character, the remaining $m - 1$ characters are compared left-to-right instead of right-to-left, and we only describe details for

	$\|\Sigma\|$			
m	2	3	4	5
2	4 / 4	6 / 9	8 / 16	10 / 25
3	7 / 8	11 / 27	14 / 64	17 / 125
4	14 / 16	24 / 81	28 / 256	32 / 625
5	28 / 32	58 / 243	66 / 1024	71 / 3125
6	56 / 64	156 / 729	166 / 4096	176 / 15625
7	112 / 128	413 / 2187	440 / 16384	452 / 78125
8	224 / 256	1086 / 6561	1158 / 65536	1190 / 390625

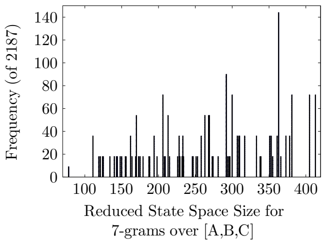


Fig. 1. Left: Maximal size $M(|\Sigma|, m)$ of the reduced state space for different alphabet sizes $|\Sigma|$ and pattern lengths m. Entries are given in the form $M(|\Sigma|, m)$ / $|\Sigma|^m$ (state space size of basic construction). **Right:** Distribution of $|\mathcal{Q}(p)|$ for $p \in \{A, B, C\}^7$. Minimum reduced state space size is 78 (for 9 patterns), maximum is $M(3, 7) = 413$ (for 72 patterns).

Horspool. For this variant the character accesses in window $w \in \Sigma^m$ are counted as $\xi^p_{\llcorner}(w) = 1$ if $p[m-1] \neq w[m-1]$, and $\xi^p_{\llcorner}(w) = \min\{m, 2+\min\{i \geq 0 : p[i] \neq w[i]\}\}$ otherwise. Thus the following information (instead of the entire window content) is sufficient to compute $(\mathit{shift}(w), \xi^p_{\llcorner}(w))$ for the current and future windows:

1. character $a := w[m-1]$ to compute $\mathit{shift}(w) = \mathit{shift}(a)$ and outgoing transition probabilities in a first-order Markov model (for higher-order models, more characters are required).
2. length x of the longest w-prefix that matches a prefix of p to compute $\xi^p_{\llcorner}(w)$.
3. all pairs (σ_i, λ_i) for which $\sigma_i \geq \mathit{shift}(a)$, $\sigma_{i-1} < \sigma_i$, $\lambda_i \geq 1$, and for which $w[\sigma_i \ldots \sigma_i + \lambda_i - 1] = p[0 \ldots \lambda_i - 1]$, that is, all pairs of positions and lengths where a proper prefix of p starts in w beyond $\mathit{shift}(a)$; this is required to ensure that a and x can be obtained after shifting the window once or several times.

Thus we define a mapping $r : \Sigma^m \to \mathcal{Q}$ with $r(w) := (x, \{(\sigma_i, \lambda_i)\}_i, a)$, where we define $\mathcal{Q}(p)$ as the image of Σ^m under r according to the above description. By aggregating transition probabilities, the PAA can be defined solely on $\mathcal{Q}(p)$ without further difficulties for i.i.d. models and first-order Markov models (since each state knows the last character a of its window).

Consider two patterns $p_1 = \texttt{AAAAA}$ and $p_2 = \texttt{ACAGC}$. The sizes of the reduced state space differ: We obtain $\mathcal{Q}(p_1) = 31$ and $\mathcal{Q}(p_2) = 60$ (in comparison to $4^5 = 1024$). Therefore we ask for (an upper bound of) the maximal size $M(|\Sigma|, m) := \max_{p \in \Sigma^m} \mathcal{Q}(p)$.

Figure 1 shows $M(|\Sigma|, m)$ for different values of $|\Sigma|$ and m. We observe that $M(|\Sigma|, m)$ is generally much smaller than $|\Sigma|^m$, especially for large alphabets. The figure also shows the distribution of the size of $\mathcal{Q}(p)$ for all 2187 patterns with $|\Sigma| = 3$ and $m = 7$: $|\mathcal{Q}(p)|$ varies between 78 and 413 (in comparison to 2187), depending on p. We propose the following conjecture, which would be a considerable improvement over the results of Sections 3 and 4.

Conjecture 1. There exists a constant $C \approx 3$ independent of $|\Sigma|$ and m such that $M(|\Sigma|, m) = O(|\Sigma| m \cdot C^m)$. Thus $\mathcal{L}(X^p_\ell)$ can be computed with $O(\ell |\mathcal{Q}|^2 |N|) = O(\ell^3\, m^3\, |\Sigma|^2\, C^{2m})$ operations.

For patterns p that never repeat their first character ("simple" patterns), we may encode a search window w as a vector $b = (b_0, \ldots, b_{m-1}) \in \{0,1,2\}^m$, where $b_i := 1$ if a new prefix of p starts at position i in w, $b_i := 2$ if a currently running prefix does not continue, and $b_i := 0$ otherwise. Storing the last character of w explicitly, this leads to a loose bound of $O(|\Sigma| \cdot 3^m)$ for simple patterns. A difficulty arises with the encoding if $p[0]$ is repeated in p.

6 Results and Discussion

Using PAAs, we have shown how the exact distribution of the number of character accesses for Horspool's and Sunday's algorithms can be computed. The framework is general enough to admit i.i.d. text models and Markovian text models of arbitrary order. We have presented two different PAA constructions

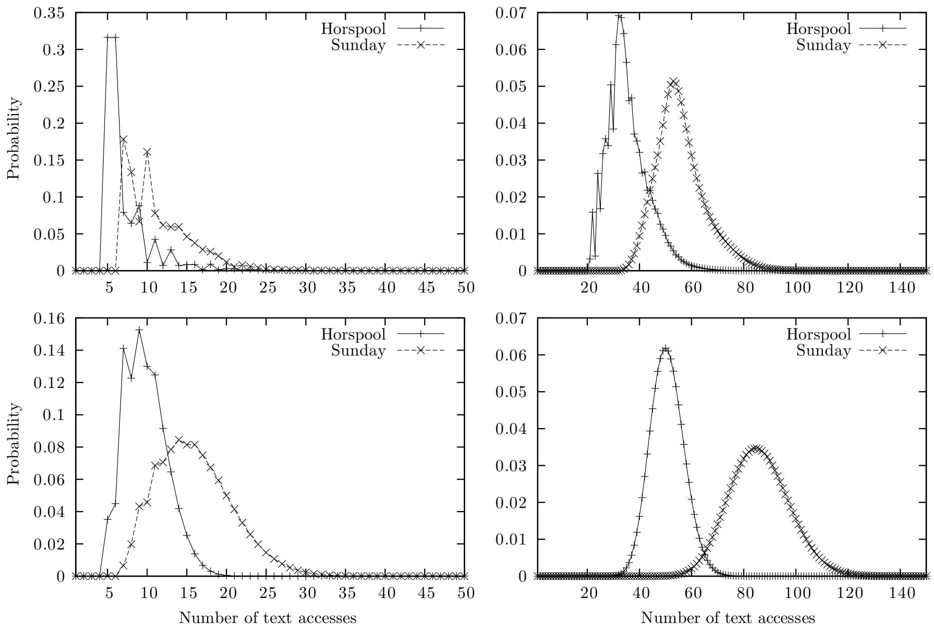


Fig. 2. Exact distribution of the number of text accesses for Horspool's and Sunday's algorithms using a uniform i.i.d. text model over the alphabet $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$. Top vs. bottom row: pattern `AAAAA` vs. `ACAGC`. Left vs. right column: text length 20 vs. 100.

which result in asymptotic runtimes of $O(\ell^3\, m\, |\Sigma|^{2m})$ and $O(\ell^2\, m^2\, m!\, \Sigma^2)$, respectively. The (super)exponential dependency on m allows practical computations only for short patterns. For a pattern length of 5 and text lengths 20 and 100, the calculation took 1.8 seconds and 40.4 seconds, respectively[1]. As shown in Section 5, it may be possible to reduce the state space considerably for the first construction. A tight bound for the size of the state space remains open at present. Also, it is not obvious how to construct the reduced state space and transition probabilities for a given pattern efficiently. For the second construction, it is also true that not all of the $O(\ell m \cdot n!)$ states can be reached for each pattern, and that the space can be possibly reduced.

For long texts the distribution of the number of text accesses converges to a Gaussian distribution [13]. For short texts, however, the distribution is governed by combinatorial effects, as illustrated in Figure 2. Even for text length 100, we observe a non-smooth distribution for `AAAAA` over the alphabet $\{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}$ with a uniform i.i.d. text model. In general, the distribution for the Sunday algorithm is smoother. On average, Sunday uses more character accesses than Horspool: Even though some shifts for Sunday are longer by one character, at least one shift value is not, and Sunday accesses at least two characters per window, while Horspool may access only one.

[1] See `http://www.rahmannlab.de/software` for an implementation in JAVA. The experiments were run on an Intel Core 2 Quad CPU at 2.66GHz.

Using the basic construction of Section 3, any window-based pattern matching algorithm can be analyzed in a similar fashion. We only need to define the corresponding *shift*-function and a cost function $\xi^p_{\text{␣}}(w)$ counting the text accesses for each window content. Particular algorithms to analyze in this way include, for example, Backward Dawg Matching (BDM), its nondeterministic counterpart BNDM or Backward Oracle Matching (BOM), as described in [11]. Even other metrics than text character accesses are possible, for example, just counting the number of windows by defining $\xi^p_{\text{␣}}(w) = 1$ for all $w \in \Sigma^m$. The presented material may therefore be seen as a general guide to the exact analysis of window-based pattern matching algorithms.

References

1. Baeza-Yates, R.A., Gonnet, G.H., Régnier, M.: Analysis of Boyer-Moore-type string searching algorithms. In: SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, pp. 328–343. SIAM, Philadelphia (1990)
2. Baeza-Yates, R.A., Régnier, M.: Average running time of the Boyer-Moore-Horspool algorithm. Theor. Comput. Sci. 92(1), 19–31 (1992)
3. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communications of the ACM 20(10), 762–772 (1977)
4. Herms, I., Rahmann, S.: Computing alignment seed sensitivity with probabilistic arithmetic automata. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 318–329. Springer, Heidelberg (2008)
5. Horspool, R.N.: Practical fast searching in strings. Software-Practice and Experience 10, 501–506 (1980)
6. Knuth, D.E., Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)
7. Kucherov, G., Noé, L., Roytberg, M.: A unifying framework for seed sensitivity and its application to subset seeds. Journal of Bioinformatics and Computational Biology 4(2), 553–569 (2006)
8. Mahmoud, H.M., Smythe, R.T., Régnier, M.: Analysis of Boyer-Moore-Horspool string-matching heuristic. Random Structures and Algorithms 10(1-2), 169–186 (1997)
9. Marschall, T., Rahmann, S.: Probabilistic arithmetic automata and their application to pattern matching statistics. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 95–106. Springer, Heidelberg (2008)
10. Marschall, T., Rahmann, S.: Efficient exact motif discovery. Bioinformatics 25(12), i356–i364 (2009)
11. Navarro, G., Raffinot, M.: Flexible Pattern Matching in Strings. Cambridge University Press, Cambridge (2002)
12. Schulz, M., Weese, D., Rausch, T., Döring, A., Reinert, K., Vingron, M.: Fast and adaptive variable order Markov chain construction. In: Crandall, K.A., Lagergren, J. (eds.) WABI 2008. LNCS (LNBI), vol. 5251, pp. 306–317. Springer, Heidelberg (2008)
13. Smythe, R.T.: The Boyer-Moore-Horspool heuristic with Markovian input. Random Structures and Algorithms 18(2), 153–163 (2001)
14. Sunday, D.M.: A very fast substring search algorithm. Communications of the ACM 33(8), 132–142 (1990)

SA-REPC – Sequence Alignment with Regular Expression Path Constraint

Nimrod Milo, Tamar Pinhas, and Michal Ziv-Ukelson

Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel

Abstract. In this paper, we define a novel variation on the constrained sequence alignment problem, the *Sequence Alignment with Regular Expression Path Constraint* problem, in which the constraint is given in the form of a regular expression. Our definition extends and generalizes the existing definitions of alignment-path constrained sequence alignments to the expressive power of regular expressions. We give a solution for the new variation of the problem and demonstrate its application to integrate microRNA-target interaction patterns into the target prediction computation. Our approach can serve as an efficient filter for more computationally demanding target prediction filtration algorithms. We compare our implementation for the SA-REPC problem, `cAlign`, to other microRNA target prediction algorithms.

1 Motivation

microRNAs are short (19 to 25 nucleotides) RNAs that play a central role in gene regulation [3,20]. microRNAs interact with other RNAs that carry the gene informations later expressed in cells. This interaction, called hybridization, causes suppression of the gene and is based on sequence complementarity (see Fig. 1). microRNA target prediction is a computational approach towards finding microRNA-RNA hybridizations which are likely to occur in nature.

Several difficulties challenge microRNA target prediction in animals today. The detection of microRNA-target hybridization pairs in vivo is complex and, in fact, only a small fraction of computational predictions have actually been validated in the lab. microRNA target prediction methods and effective biological validation techniques contribute significantly towards a growing understanding of microRNA functionality [3]. Initial data indicates that genes which represent approximately half of our gene sets are targeted by microRNAs and approximately 148 conserved microRNAs target 30% of all genes [18], where genes are often regulated by multiple microRNAs [16].

Different characteristics of the microRNA-target hybridization were observed by [3], some of which serve as a basis for current microRNA target prediction algorithms. The majority of the hybridization characteristics relate to *seeds*, which are short segments of the microRNA. These characteristics fall into 5 classes, as follows:

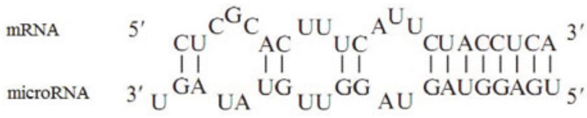

mRNA
microRNA
5′
3′
3′
5′
CU C G C AC UU UC A U U CUACCUCA
U GA UA UG UU GG AU GAUGGAGU

Fig. 1. A typical hybridization between let-7 microRNA and a binding site within the 3' Untranslated Region of the target hbl-1 mRNA sequence in C. elegans [20]. Note that the base C typically pairs with G and the base U pairs with A. The seed, which is the 8 consecutive base pairs, appears on the 5' side of the microRNA.

1. *5'-end dominant seed:* Studies suggest the existence of 6-8 consecutive base pairs in the 5'-end of the microRNA, commonly referred to as the seed of the microRNA [5,2]. The 5' end of the seed is usually unpaired or starts with `U`. Additional characteristics of the dominant seed are that there are no wobble pairs (i.e. a `(G:U)` base pair) and it ends with `A` [18].
2. *3'-end compensatory seed:* Several studies found that although the 5'-end dominant seed is important, there is significant evidence that a 3'-end seed of a microRNA can compensate for a non-perfect 5'-seed base pairing [5,29]. The presence of the 3'-end compensatory seed can replace or supplement the 5'-end seed.
3. *Multiplicity:* microRNAs have been shown to be capable of functioning in a collaborative manner. Studies show that microRNA function may depend on the ability to bind to multiple binding sites on the same 3' Untranslated Region [henceforth: UTR] of the target mRNA [29]. Therefore, it is probable that one microRNA, with a lower level of complementarity, but a multiple-site target, could have the same regulatory effect as another microRNA, exhibiting high complementarity to its single-site target.
4. *Accessibility of target-mRNA binding site:* It has been shown that target site accessibility plays an important role in the formation of microRNA-target hybridization [15].The binding site area in the mRNA should be accessible, allowing the microRNA to bind with it. Also, the secondary structure of the area surrounding the binding site should contain neither stabilizing (e.g., stems) nor destabilizing structural features.
5. *Thermodynamics of microRNA-target hybridization:* Researchers observed that microRNA-target hybridization tends to have low free energy, due to the fact that biomolecular interaction will reach equilibrium when the interaction is thermodynamically stable [27].

Several approaches toward target prediction exist, the dominant ones being sequence complementarity and thermodynamics [21]. While some methods focus on first finding complementarity, followed by thermodynamic analysis, other methods use thermodynamics as the initial indicator of microRNA binding site potential. A score is typically assigned to each predicted target; this score can be useful for target ranking.

Another approach to target prediction uses statistical methods and machine learning to attempt to generalize characteristics of known microRNA-target pairs. The results of these probabilistic models depend on their training data sets. A training data set which is not diverse enough may cause an insignificant feature to become a key search criterion in that probabilistic model (e.g. Hidden Markov Model [henceforth: HMM], neural network or support vector machines [30]). To our knowledge, there are no existing solutions that use probabilistic automata for microRNA target prediction.

Among the classes listed above, we note that classes 4 and 5 are the computational bottlenecks, as they involve the prediction of RNA secondary structure based on thermodynamics, when taking both the hybridization structure, as well as self-folding, into account. The complexity of such computations ranges from $O(nm^2)$ (when restricting the size of interior loops and ignoring self-folding of microRNA and target) [4] and up to $O(nm^5)$ [23], where n denotes the length of the target sequence and m denotes the size of the microRNA. This observation motivated a line of microRNA target prediction applications that use complementarity constraints as a front-end filter and apply the expensive thermodynamic constraint computation only to the surviving candidates. In most cases, complementarity was used to identify potential targets, followed by iterative rounds of filtering based on thermodynamics, binding site structure and conservation. In this work, we propose an expressive and flexible filtration criterion (based on classes 1-3 above) that is more informative than the sequence complementarity criterion.

Our approach is based on the observation that microRNA-target hybridization classes 1-3 can be formulated in a simple and flexible constraint via a regular expression. Thus, below, we define the *Sequence Alignment with Regular Expression Path Constraint* [henceforth: SA-REPC] problem, a new variant of alignment-path constrained sequence alignment and solve it. In addition, we discuss the possibilities of guided scoring schemes and local alignment. We implement our approach in a software package, `cAlign`, and apply it as a fast filter for microRNA target prediction. Our experimental results show that our predictions yield a relatively high sensitivity value compared to existing target prediction tools.

Using our approach has some advantages over other automata-based probabilistic approaches, such as pair HMMs [7]. Commonly used hybridization rules are based on biological experiments, rather than automatic (supervised or unsupervised) learning. If the dogma changes (which happens frequently in the dynamic and relatively new field of microRNA research), we just need to modify the constraint descriptions and not the algorithm. In contrast to learning-based algorithms, such a change does not require a rerun of a heavy learning process. Furthermore, our approach is simple, both in terms of algorithmic complexity and in terms of ease of implementation. The algorithmic complexity is $O(nmt)$ for the Deterministic Finite Automaton [henceforth: DFA] used in this application, where t denotes the number of states in the automaton used to represent the hybridization rules (with $t \sim m$). As far as we know, we are the first to apply such an automata-based approach to microRNA target prediction filtration.

The rest of the paper proceeds as follows: In Section 3, we define the SA-REPC problem. In Section 4, we propose a solution to this problem, implemented as a software package. Then, we describe suggested extensions to the basic algorithm. Finally, in Section 5, we apply the SA-REPC problem to microRNA target prediction.

2 Preliminaries

Sequence alignment is one of the most basic and well-studied problems in the Stringology field [9], with numerous applications in computational biology (see Fig. 2a). Let Σ represent an alphabet and let s denote a *scoring matrix*, where $s[\sigma_1, \sigma_2]$ is the score of the pair $\sigma_1, \sigma_2 \in \Sigma \bigcup \{'-'\}$. The sequence alignment problem is defined as follows:

Definition 1 (The global sequence alignment problem)
Let S_1 and S_2 be two strings over an alphabet Σ and let s be a scoring matrix. A sequence alignment is obtained by inserting $'-'$ symbols into S_1 and S_2, so that the symbols of the resulting strings can be put in one-to-one correspondence with each other. The optimal global sequence alignment is a sequence alignment that has the optimal sum of scores, according to s, over the pairs of symbols that correspond to each other in the alignment.

The use or meaning of the global sequence alignment is derived from the input scoring matrix. For instance, if the purpose of the alignment is to compute the similarity between two sequences then $s(\sigma_1, \sigma_2)$ would carry a relatively good score if σ_1 and σ_2 are the same symbol. If, on the other hand, the objective is to compute the most likely hybridization between two RNA sequences, then $s(\sigma_1, \sigma_2)$ would carry a good score if σ_1 and σ_2 can bind to each other.

The pairwise global sequence alignment problem is classically solved as an optimal path in an alignment grid or an alignment table. There is a one-to-one correspondence between each alignment of two strings and a single path, referred to as an *alignment path*, from the source to the sink nodes in the respective alignment grid.

Numerous studies suggest the application of additional constraints to sequence alignment for the purpose of improved speed or accuracy. The additional constraint reflects a priori knowledge of the alignment and, therefore, narrows the problem search space or guides the search towards a preferred solution. We observe that the types of constraints imposed on sequence alignment appearing in the literature can be grouped into three categories as follows: in the first category, the constraint delineates aligned substrings of each of the aligned strings independently. Several variants in this category exist, ranging from a string pattern constraint [28] to a regular expression constraint [1].

In the second category, the constraint delineates the alignment path, rather than the aligned substrings. This includes simple constraints, such as position anchoring [24] and k-difference alignment [9], and more sophisticated constraints, such as spaced seeds [17].

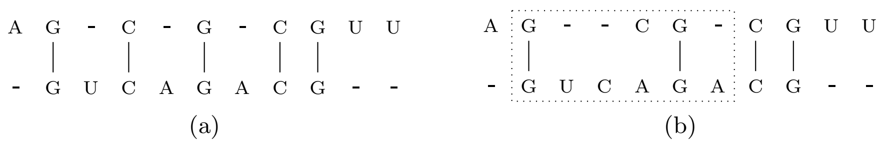


Fig. 2. Examples for global sequence alignment (Def. 1) and SA-REPC (Def. 2) on the two strings $S_1 = AGCGCGUU$ and $S_2 = GUCAGACG$, with a scoring matrix (-1 for mismatch/indel (space), 1 for match). (a) The maximal score of the global alignment is -1. (b) Let R be $10^3 10$, the maximal score of the constrained global alignment is -3.

In the third category, a context-sensitive scoring scheme is used to influence and guide the alignment. This includes position-based scoring functions [14] and pair HMMs, which adapt their scoring scheme via a learning process on a training data set. HMM is a probabilistic automaton, in which both the target state and the output of a transition are determined by probabilities. Durbin, et al. [7] defined the pair HMM which generates an aligned pair of sequences. A common use of pair HMMs is to calculate the most probable alignment of two sequences, according to the probabilities of a given pair HMM.

Among the above categories, the new problem we introduce here, denoted the SA-REPC problem, falls within the alignment-path constrained sequence alignment (second) category, extending it to include constraints in the form of regular expressions.

3 Problem Definition of SA-REPC

We give an extended definition of sequence alignment with alignment-path constraints by requiring part of the alignment to match a given regular expression R. An *alignment alphabet* with respect to a sequence alphabet Σ may be one of the following sets:

1. $\Sigma_r = \{0, 1\}$, where 1 denotes a match and 0 a mismatch.
2. $\Sigma_r = \{i, d, m, s\}$, where i,d,m and s denote the operations insertion, deletion, match and substitution, respectively.
3. $\Sigma_r = \left\{ \begin{matrix}\sigma_1 \\ \sigma_2\end{matrix} \mid \sigma_1, \sigma_2 \in \Sigma \bigcup \{'-'\} \right\} \setminus \left\{ \begin{matrix} '-' \\ '-' \end{matrix} \right\}$, where $\begin{matrix}\sigma_1 \\ \sigma_2\end{matrix}$ represents two aligned symbols σ_1 and σ_2. The symbol $'-'$ cannot be aligned with itself.

We consider an alignment-path constraint to be a regular expression over an alignment alphabet (see Fig. 2b). The SA-REPC problem is defined as follows:

Definition 2 (The global SA-REPC problem)
Let S_1 and S_2 be two strings over an alphabet Σ with lengths n and m, respectively, and let s be a scoring matrix. Let R be a regular expression over an alignment alphabet Σ_r. Find an alignment of S_1 and S_2 such that two conditions hold:

1. *There exists an accepted region in the alignment belonging to* L_R.
2. *The overall score of the alignment, computed according to* s, *is optimal among all such alignments.*

A symbol of the alignment alphabet may fit multiple aligned pairs of symbols from the sequence alphabet. For this reason, we define a function $f : \Sigma \bigcup \{'-'\} \times \Sigma \bigcup \{'-'\} \rightarrow \Sigma_r$, which specifies which alignment alphabet symbol fits which pairs of aligned symbols.

A standard alteration of the above definition yields the *local* SA-REPC problem. In the case of local alignment [26], calculation of the optimal score takes into account part of the alignment of the given input strings.

4 An Algorithm for the SA-REPC Problem

We start by formulating an algorithm for global SA-REPC .
Let $A^R = \langle Q^R, q_0^R, F^R, \delta^R, \Sigma \rangle$ be a non-deterministic, finite automaton [henceforth: NFA] corresponding to R. We define an NFA $A = \langle Q, q_0, F, \delta, \Sigma \rangle$ as follows:

1. $Q = Q^R \bigcup \{q_{\text{init}}, q_{\text{final}}\}$.
2. $q_0 = q_{\text{init}}$.
3. $F = \{q_{\text{final}}\}$.
4. $\delta = \delta^R$ with the following additions
   (a) $\delta(q_{\text{init}}, a) = \{q_{\text{init}}\}, \forall a \in \Sigma$
   (b) $\delta(q_{\text{init}}, \epsilon) = \{q_0^R\}$
   (c) $\delta(q_{\text{final}}, a) = \{q_{\text{final}}\}, \forall a \in \Sigma$
   (d) $\delta(q, \epsilon) = \{q_{\text{final}}\}, \forall q \in F^R$

An example of the NFA construction appears in Fig. 3. ϵ-transitions are then eliminated from A according to [11].

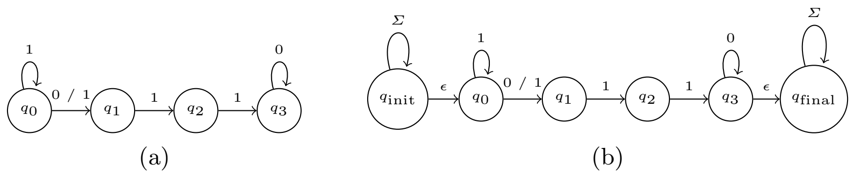


Fig. 3. An example of the NFA construction. (a) An NFA A^R for $1^*(1|0)1^20^*$. (b) The constructed NFA A.

$$\begin{tikzcd} q_0 \arrow[loop above, "1"] \arrow[r, "0\,/\,1"] & q_1 \arrow[r, "1"] & q_2 \arrow[r, "1"] & q_3 \arrow[loop above, "0"] \end{tikzcd}$$

$$\begin{tikzcd} q_{\text{init}} \arrow[loop above, "\Sigma"] \arrow[r, "\epsilon"] & q_0 \arrow[loop above, "1"] \arrow[r, "0\,/\,1"] & q_1 \arrow[r, "1"] & q_2 \arrow[r, "1"] & q_3 \arrow[loop above, "0"] \arrow[r, "\epsilon"] & q_{\text{final}} \arrow[loop above, "\Sigma"] \end{tikzcd}$$

We calculate a dynamic programming table M. Each cell $M[i, j]$ holds $|Q|$ entries. $M[i, j](q)$ holds the optimal alignment score of $S_1[1, i]$ with $S_2[1, j]$, such that there exists a reading in A of the alignment that reached q. If no such alignment exists, then the value of the entry $M[i, j](q)$ is *null*. The entry of q_{init} holds the optimal unconstrained alignment score of $S_1[1, i]$ with $S_2[1, j]$. The entry of q_{final} is the optimal alignment score of $S_1[1, i]$ with $S_2[1, j]$, such that a prefix of the alignment belongs to L_R. The recurrence formula for the problem is as follows[1]:

$$M[0,0](q) = \begin{cases} 0 & \forall q = q_{\text{init}} \\ null & \text{otherwise} \end{cases} \tag{4.1}$$

$$M[i,j](q) = \text{opt} \begin{cases} \text{opt}\,\{M[i-1,j-1](p) + s[S_1[i], S_2[j]] \mid q \in \delta(p, f(S_1[i], S_2[j]))\} \\ \text{opt}\,\{M[i-1,j](p) + s[S_1[i], '-'] \mid q \in \delta(p, f(S_1[i], '-'))\} \\ \text{opt}\,\{M[i,j-1](p) + s['-', S_2[j]] \mid q \in \delta(p, f('-', S_2[j]))\} \end{cases} \tag{4.2}$$

If $i = 0$ (or $j = 0$), the terms above corresponding to $i-1$ (or to $j-1$) are ignored. The value of $M[n,m](q_{\text{final}})$ is the score of the best alignment containing a region belonging to L_R.

The above algorithm can be easily modified for *local* SA-REPC within the same time and space complexities. This is achieved by adding 0 as an additional term to the optimum calculation in Eq. (4.2).

Time and Space Complexity Analysis

Using the recurrence formula yields a run time complexity of $O(mnt^2)$ when the automaton representing R is non-deterministic and $O(mnt)$ when the automaton representing R is deterministic. The algorithm computes mn cells. According to Eq. (4.2) above, in the case of an NFA, the calculation of $M[i,j](q)$ is performed for t states; each is calculated according to at most t values. In the case of a DFA, each state has exactly one outgoing transition with a specific alignment alphabet symbol; therefore, the overall amount of states included in the calculation of $M[i,j]$ is $O(t)$.

The space required in the naïve approach, in which the entire dynamic programming table is kept in memory, is $O(mnt)$. If only the value of the optimal alignment is required, then it suffices to keep $\min\{n,m\}+1$ cells, each having $t+2$ entries. Thus, the space requirement is $O(\min\{n,m\}t)$. The Hirschberg approach [10] can be applied to our algorithm in order to obtain a space complexity of $O(\min\{m,n\}t)$, without sacrificing its run time complexity and allowing the alignment trace to be recovered.

4.1 Scoring Scheme Refinement for Constraint Matching Region

The region of an alignment that matches a regular expression may be scored more precisely by applying, to that segment, a scoring scheme which differs from the scoring matrix that is applied to the rest of the alignment. We suggest taking the regular expression into account when scoring the above region (e.g. the accepted region). Another scoring matrix is introduced, that bares additional knowledge, relevant to the expected behavior of the accepted region.

[1] The optimum value of an empty set is null. Arithmetic operations also yield a null value if one of their arguments is null.

Example 1. Consider the case where the regular expression is

$$R = \binom{A}{A}\left(\binom{A}{T} \mid \binom{A}{C}\right)\binom{-}{G}^{*}\binom{C}{C}$$

and the corresponding alignment substrings are AACC and ATGGGGGGCC. If R is used without modifying the scoring scheme to reflect the expected indels and substitutions, then the overall score would not reflect the quality of the alignment. Moreover, a certain substitution may be preferred over others within the accepted region, deserving a better score in the scoring matrix. In addition, there may be biological motivation to leniently score $\binom{-}{G}$ within the accepted region, because e.g. that gap may represent a region that was extracted during replication and does not significantly effect the similarity of the sequences.

Definition 3 (The score-guided global SA-REPC problem)
Let S_1 and S_2 be two strings over a sequence alphabet Σ with lengths n and m, respectively, and let s_1 and s_2 be scoring matrices. Let R be a regular expression over an alignment alphabet. Find an alignment of S_1 and S_2 such that two conditions hold:

1. *There exists an accepted region in the alignment belonging to L_R.*
2. *The score of the alignment, computed over the entire alignment, is optimal among all such alignments. The score is the sum of the score of the accepted region, according to s_1, and the scores of the remaining regions of the alignment of S_1 and S_2, according to s_2.*

We change the scoring contribution of the accepted region in the global optimal alignment computation by applying s_1 for all aligned symbol pairs that are read by transitions which do not equal the transition from q_{init} or q_{final} to itself. Thus, the score function s applied in Eq. (4.2) is either s_1 or s_2, according to the relevant transition.

5 The Application of SA-REPC to microRNA Target Prediction

In this section, we describe the application of SA-REPC to microRNA target prediction. In the context of microRNA target prediction, the aligned sequences are a microRNA and a 3'UTR, which have lengths approximately $m = 25$ and $n = 2000$ respectively. The RE constraint we are using for this application (below) has approximately $t = 45$ states.

We suggest applying our tool as a filter before applying more expensive algorithms to the surviving candidates. We customized our application to perform semi-local alignment, in order to get the best alignments of the entire microRNA with parts of the 3'UTR. A discussion of the constraint design and our experimental results follow.

5.1 Utilizing Path-constrained Semi-local Alignment for microRNA Target Prediction

In Section 1, we described the characteristics of microRNA-target hybridization (see Fig. 1). We now use these characteristics in the design of a regular expression constraint for target prediction. Many of the features of hybridization of microRNA with mRNA targets can be represented via a regular expression. We show that regular-expression-type alignment-path constraints provide a simple and flexible way to formulate multiple hybridization features in the context of target prediction. We can translate some features from the learning of hybridizations into patterns. The 5' dominant seed can be described by a simple regular expression

$$\left(0 \mid \frac{A}{U}\right)\left(\frac{G}{C} \mid \frac{C}{G} \mid \frac{A}{U} \mid \frac{U}{A}\right)^{6-8} \tag{5.1}$$

We also describe the constraints on the middle section of the hybridization. We do not allow single, non-stacked pairs in the middle of the hybridization, since we assume that they are rare, nor do we allow a gap size greater than 5 for thermodynamic reasons. This section has a limited length, since the length of the microRNA is bounded. Despite this property, we preferred a regular expression that allows unlimited length, by including a Kleene star, in order to get a shorter expression than the one yielded by an enumeration and, thus, to minimize the number of states t, resulting in better performance. The part in the hybridization which is not the seed can be expressed as:

$$\left(0^{0-3}110^{0-2}\right)^{*} \tag{5.2}$$

In the case of a non-perfect 5' seed, there is a 3' compensatory seed. We assume that the 3' compensatory seed contains at least 4 matches and ends with, at most, 2 mismatches [2]. As opposed to the 5' dominant seed description, we allow the 3' compensatory seed to have wobble pairs. The form of the 3' compensatory seed expression is:

$$1^{4-5}0^{0-2} \tag{5.3}$$

An expression for the non-perfect 5' seed, which complements the 3' compensatory seed, can be composed as a union of 10 possibilities. The non-perfect 5' seed contains at least 6 positions, with either 1 or 2 mismatches. Position 1 is a mismatch and positions 2 and 7 must match. Hence, the non-perfect 5' seed has the following form:

$$01\,(0111 \mid 1011 \mid 1101 \mid 1110 \mid 0011 \mid 0101 \mid 0110 \mid 1001 \mid 1010 \mid 1100)\,1 \tag{5.4}$$

5.2 Test Results

We implemented the algorithm in a tool called `cAlign` for microRNA target prediction. `cAlign` is implemented in `Java 1.6` and `Python 2.6`. Our data test set consists of 873 verified binding sites of microRNAs on human 3'UTRs, collected from the `miRecords` database [31]. These verified targets involve 99 microRNAs from

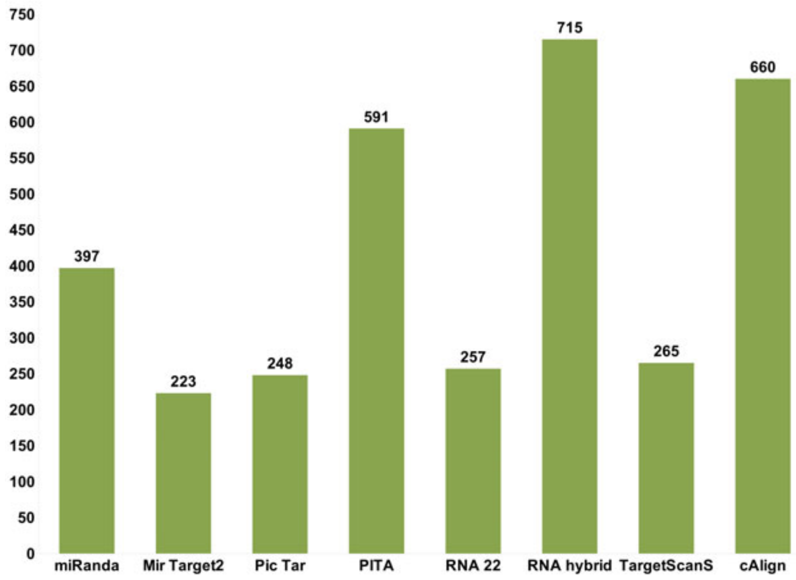


Fig. 4. A comparison between `cAlign` and other target prediction tools [16,15,30,14,25,19,22]. The results for the additional target prediction tools are taken from `miRecords` [31]. Due to space restrictions, additional information regarding the other target prediction tools will be given in the extended version of this paper.

`mirBase` [8] and 640 human genes from `Ensembl v56` [12]. Most genes have several transcripts, which yield a total of 2183 transcripts. We performed constrained sequence alignment, using the regular expression described in Subsection 5.1.

Each microRNA was aligned to the 3'UTR of each gene in a semi-local alignment. For these alignments, we used a scoring matrix correlating to the number of hydrogen bonds in a base pair. This scoring matrix assigns the values – 3 for a `(G:C)` pair, 2 for a `(A:U)` pair, 1.5 for a `(G:U)` pair and -1 otherwise. For each microRNA-3'UTR pair, we calculated the optimal semi-local alignment score and calculated a multiplicity value (i.e. we counted the number of different hybridizations within a small delta from the optimal score). We next randomly generated 1000 3'UTR sequences over the di-nucleotide distribution of the compared 3'UTR using `UShuffle` [13]. We counted the number of random scores which were greater than the hybridization score and calculated their average. A hybridization is statistically meaningful if the number of random sequences with a better score is less than 10% (specificity of 90%) and its score is higher than the average score produced according to its randomly generated sequences. We consider a hybridization to be well-matched if it has a relatively high multiplicity (above average) or if it is statistically meaningful.

Using the above procedure, we identified 660 of the verified targets (75.6%). The comparison of our results to the results of other algorithms on the same data set is shown in Fig. 4. These results show that our predictions on the test data set rank in sensitivity between `PITA` [15] and `RNAHybrid` [25].

6 Conclusions and Open Problems

We extended the alignment-path constrained sequence alignment to handle constraints in the form of regular expressions. We have demonstrated the ease of use of our SA-REPC algorithm in one of many possible applications of this problem, namely as an efficient filter for microRNA target prediction. In future, our approach may be extended to more general language classifications, such as grammars. An interesting open problem might be the application of some of the techniques previously used to obtain sub-quadratic sequence alignment, such as Four Russians [9] and acceleration by compression [6], to reduce the time complexity of SA-REPC .

Acknowledgments

The authors would like to thank Isana Vexler-Lublinsky for her helpful comments regarding microRNA target prediction and Dr. Chen Keasar for a helpful discussion. The work of Nimrod Milo and Michal Ziv-Ukelson was partially supported by the Frankel Center for Computer Science at Ben-Gurion University of the Negev. The authors are very grateful to Ethelea Katzenell (Tamar's mom) for carefully editing the manuscript.

References

1. Arslan, A.: Regular expression constrained sequence alignment. Journal of Discrete Algorithms 5(4), 647–661 (2007)
2. Bartel, D.: MicroRNAs: target recognition and regulatory functions. Cell 136(2), 215–233 (2009)
3. Bentwich, I.: Prediction and validation of microRNAs and their targets. FEBS letters 579(26), 5904–5910 (2005)
4. Bernhart, S., Tafer, H., Mückstein, U., Flamm, C., Stadler, P., Hofacker, I.: Partition function and base pairing probabilities of RNA heterodimers. Algorithms for Molecular Biology 1(1), 3 (2006)
5. Brennecke, J., Stark, A., Russell, R., Cohen, S.: Principles of MicroRNA–Target Recognition. PLoS Biol. 3(3), e85 (2005)
6. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. SIAM Journal on Computing 32, 1654 (2003)
7. Durbin, R., Eddy, S., Krogh, A., Mitchison, G.: Biological sequence analysis. Cambridge Univ. Press, Cambridge (1998)
8. Griffiths-Jones, S., Grocock, R., van Dongen, S., Bateman, A., Enright, A.: miRBase: microRNA sequences, targets and gene nomenclature. Nucleic acids research 34(Database Issue), D140 (2006)
9. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (January 1997)
10. Hirschberg, D.S.: Algorithms for the longest common subsequence problem. J. ACM 24(4), 664–675 (1977)
11. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading (2006)

12. Hubbard, T., Andrews, D., Caccamo, M., Cameron, G., Chen, Y., Clamp, M., Clarke, L., Coates, G., Cox, T., Cunningham, F., et al.: Ensembl 2005. Nucleic Acids Research 33(Database Issue), D447 (2005)
13. Jiang, M., Anderson, J., Gillespie, J., Mayne, M.: uShuffle: A useful tool for shuffling biological sequences while preserving the k-let counts. BMC bioinformatics 9(1), 192 (2008)
14. John, B., Sander, C., Marks, D., et al.: Prediction of human microRNA targets. Methods In Molecular Biology 342, 101 (2006)
15. Kertesz, M., Iovino, N., Unnerstall, U., Gaul, U., Segal, E.: The role of site accessibility in microRNA target recognition. Nature genetics 39(10), 1278–1284 (2007)
16. Krek, A., Grün, D., Poy, M., Wolf, R., Rosenberg, L., Epstein, E., MacMenamin, P., da Piedade, I., Gunsalus, K., Stoffel, M., et al.: Combinatorial microRNA target predictions. Nature genetics 37(5), 495–500 (2005)
17. Kucherov, G., Noé, L., Roytberg, M.: Multiseed lossless filtration. IEEE/ACM Transactions on Computational Biology and Bioinformatics, 51–61 (2005)
18. Lewis, B., Burge, C., Bartel, D.: Conserved seed pairing, often flanked by adenosines, indicates that thousands of human genes are microRNA targets. Cell 120(1), 15–20 (2005)
19. Lewis, B., Shih, I., Jones-Rhoades, M., Bartel, D., Burge, C.: Prediction of mammalian microRNA targets. Cell 115(7), 787–798 (2003)
20. Lin, S., Johnson, S., Abraham, M., Vella, M., Pasquinelli, A., Gamberi, C., Gottlieb, E., Slack, F.: The C. elegans hunchback homolog, hbl-1, controls temporal patterning and is a probable microRNA target. Developmental Cell 4(5), 639–650 (2003)
21. Maziere, P., Enright, A.: Prediction of microRNA targets. Drug discovery today 12(11-12), 452–458 (2007)
22. Miranda, K., Huynh, T., Tay, Y., Ang, Y., Tam, W., Thomson, A., Lim, B., Rigoutsos, I.: A pattern-based method for the identification of MicroRNA binding sites and their corresponding heteroduplexes. Cell 126(6), 1203–1217 (2006)
23. Mückstein, U., Tafer, H., Bernhard, S., Hernandez-Rosales, M., Vogel, J., Stadler, P., Hofacker, I.: Translational control by RNA-RNA interaction: Improved computation of RNA-RNA binding thermodynamics. BioInformatics Research and DevelopmentBIRD 13, 114–127 (2008)
24. Myers, G., Selznick, S., Zhang, Z., Miller, W.: Progressive multiple alignment with constraints. Journal of Computational Biology 3(4), 563–572 (1996)
25. Rehmsmeier, M., Steffen, P., Hochsmann, M., Giegerich, R.: Fast and effective prediction of microRNA/target duplexes. RNA 10(10), 1507–1517 (2004)
26. Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of molecular biology 147(1), 195–197 (1981)
27. Stark, A., Brennecke, J., Russell, R., Cohen, S.: Identification of Drosophila MicroRNA Targets. PLoS Biol. 1(3), e60 (2003)
28. Tang, C., Lu, C., Chang, M., Tsai, Y., Sun, Y., Chao, K., Chang, J., Chiou, Y., Wu, C., Chang, H., et al.: Constrained multiple sequence alignment tool development and its application to RNase family alignment. Journal of Bioinformatics and Computational Biology 1(2), 267–288 (2003)
29. Vella, M., Reinert, K., Slack, F.: Architecture of a validated microRNA: target interaction. Chemistry & Biology 11(12), 1619–1623 (2004)
30. Wang, X., El Naqa, I.: Prediction of both conserved and nonconserved microRNA targets in animals. Bioinformatics 24(3), 325 (2008)
31. Xiao, F., Zuo, Z., Cai, G., Kang, S., Gao, X., Li, T.: miRecords: an integrated resource for microRNA-target interactions. Nucleic Acids Research (2008)

CD-Systems of Stateless Deterministic R(1)-Automata Accept All Rational Trace Languages[⋆]

Benedek Nagy[1] and Friedrich Otto[2]

[1] Department of Computer Science, Faculty of Informatics
University of Debrecen, 4032 Debrecen, Egyetem tér 1., Hungary
nbenedek@inf.unideb.hu
[2] Fachbereich Elektrotechnik/Informatik, Universität Kassel
34109 Kassel, Germany
otto@theory.informatik.uni-kassel.de

Abstract. We study cooperating distributed systems (CD-systems) of restarting automata that are very restricted: they are deterministic, they cannot rewrite, but only delete symbols, they restart immediately after performing a delete operation, they are stateless, and they have a read/write window of size 1 only, that is, these are stateless deterministic R(1)-automata. We relate the class of languages that are accepted by mode = 1 computations of CD-systems of such automata to other well-studied language classes, showing in particular that it only consists of semi-linear languages, and that it includes all rational trace languages.

1 Introduction

Cooperating distributed systems (CD-systems) of restarting automata have been defined in [6], and in [7] various types of deterministic CD-systems of restarting automata have been studied. As expected CD-systems are much more expressive than their component automata themselves. For example, already the marked copy language $L_{\text{copy}} = \{ wcw \mid w \in \{a,b\}^* \}$ is accepted by a CD-system consisting of only two deterministic R-automata, although this language is not even growing context-sensitive, that is, it is not even accepted by any deterministic RRWW-automaton (see, e.g., [8]). On the other hand, stateless restarting automata, that is, restarting automata with only a single state, have been introduced and studied in [4]. In the monotone case and in the deterministic case, they are just as expressive as the corresponding restarting automata with states, provided that auxiliary symbols are available. Without the latter, however, stateless restarting automata are in general much less expressive than their corresponding counterparts with states.

Here we study CD-systems of deterministic restarting automata that are stateless and that have a read/write window of size 1 only. In fact, we concentrate

[⋆] This work was supported by grants from the Balassi Intézet Magyar Ösztöndíj Bizottsága (MÖB) and the Deutsche Akademischer Austauschdienst (DAAD).

on CD-systems of stateless deterministic R-automata with window size 1. The restarting automata of this type are really very restricted, and accordingly their expressive power is very limited. However, combining several such automata into a CD-system yields a device that is suprizingly expressive as we will see. We show that in mode $= 1$ these systems only accept languages with semi-linear Parikh image, including all regular languages, but that they also accept some languages that are not even context-free. In fact, these systems accept all rational trace languages. Accordingly they can also be interpreted as a refinement of the so-called *multiset finite automata* of [2], which accept the commutative closures of all regular languages. Actually we present a syntactic restriction for CD-systems of stateless deterministic R-automata with window size 1 such that the corresponding systems characterize the class of rational trace languages. These systems yield an effective calculus for rational trace languages in that from systems of this form for rational trace languages S_1 and S_2 we can effectively construct systems for the rational trace language $S_1 \cup S_2$, $S_1 \cdot S_2$, and S_1^*.

2 Stateless R-Automata with Constant Window Size

Stateless types of restarting automata were introduced in [4]. Here we are only interested in the most restricted form of them, the *stateless* R*-automaton.* A *stateless* R*-automaton* is a one-tape machine that is described by a 5-tuple $M = (\Sigma, ¢, \$, k, \delta)$, where Σ is a finite alphabet, the symbols $¢, \$ \notin \Sigma$ serve as markers for the left and right border of the work space, respectively, $k \geq 1$ is the size of the *read/write window*, and δ is the *transition relation* that associates a finite set of *transition steps* to each possible content u of the read/write window. There are three types of transition steps: *move-right steps* (MVR), which shift the window one step to the right, combined *rewrite/restart steps*, which delete one or more symbols from the content u of the window, thereby shortening the tape, and place the window over the left end of the tape, and *accept steps* (Accept), which cause the automaton to halt and accept.

A *configuration* of M is described by a pair (α, β), where either $\alpha = \varepsilon$ (the empty word) and $\beta \in \{¢\} \cdot \Sigma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Sigma^*$ and $\beta \in \Sigma^* \cdot \{\$\}$; here $\alpha\beta$ is the current content of the tape, and it is understood that the head scans the first k symbols of β or all of β when $|\beta| \leq k$. A *restarting configuration* is of the form $(\varepsilon, ¢w\$)$, where $w \in \Sigma^*$; to simplify the notation a restarting configuration $(\varepsilon, ¢w\$)$ is usually simply written as $¢w\$$. By $\vdash_M$ we denote the single-step computation relation of M, and $\vdash_M^*$ denotes the reflexive transitive closure of $\vdash_M$.

The automaton M proceeds as follows. Starting from an initial configuration $¢w\$$, the window moves right until a configuration of the form $(¢x, uy\$)$ is reached such that $\delta(u)$ contains a rewrite step that rewrites u to v by deleting some symbols, that is, $v \in \delta(u)$. If this particular transition is now chosen, then the latter configuration is transformed into the restarting configuration $¢xvy\$$. This computation, which is called a *cycle*, is expressed as $w \vdash_M^c xvy$. A computation of M now consists of a finite sequence of cycles that is followed by a tail computation, which consists of a sequence of move-right operations possibly followed by an accept step. An input word $w \in \Sigma^*$ is *accepted* by M, if there is a computation of M which starts with the initial configuration $¢w\$$, and which finishes by executing an accept step. By $L(M)$ we denote the language consisting of all words accepted by M.

A stateless R-automaton is called *deterministic* if $\delta(u)$ contains at most one operation for all possible values of u. The prefix det- is used to denote deterministic types of restarting automata. In [4] it is shown that the class $\mathcal{L}(\mathsf{stl\text{-}det\text{-}mon\text{-}R})$ of languages that are accepted by stateless deterministic R-automata that are in addition monotone properly contains the class REG of all regular languages, but that it is a proper subclass of the class DCFL of all deterministic context-free languages.

Here we are interested in stateless R-automata with a fixed window size. For each positive integer k, we denote by stl-R(k) the class of stateless R-automata that have a read/write window of size k. For stateless deterministic R-automata with window size 1 we introduce the following notions that we will repeatedly use throughout the paper.

Definition 1. *Assume that $M = (\Sigma, ¢, \$, 1, \delta)$ is a stateless deterministic* R(1)-*automaton. Then we can partition the alphabet Σ into four disjoint subalphabets:*

$$\begin{array}{ll} (1.)\ \Sigma_1 = \{\, a \in \Sigma \mid \delta(a) = \mathsf{MVR}\,\}, & (3.)\ \Sigma_3 = \{\, a \in \Sigma \mid \delta(a) = \mathsf{Accept}\,\}, \\ (2.)\ \Sigma_2 = \{\, a \in \Sigma \mid \delta(a) = \varepsilon\,\}, & (4.)\ \Sigma_4 = \{\, a \in \Sigma \mid \delta(a) = \emptyset\,\}. \end{array}$$

Thus, Σ_1 is the set of letters that M just moves across, Σ_2 is the set of letters that M deletes, Σ_3 is the set of letters which cause M to accept, and Σ_4 is the set of letters on which M will get stuck.

Then the following characterization is easily established.

Proposition 1. *Let $M = (\Sigma, ¢, \$, 1, \delta)$ be a stateless deterministic* R(1)-*automaton, and assume that the subalphabets $\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4$ are defined as above. Then the simple language $S(M)$ of words accepted by M in tail computations is characterized as*

$$S(M) = \begin{cases} \Sigma^*, & \text{if } \delta(¢) = \mathsf{Accept}, \\ \Sigma_1^* \cdot \Sigma_3 \cdot \Sigma^*, & \text{if } \delta(¢) = \mathsf{MVR} \text{ and } \delta(\$) \neq \mathsf{Accept}, \\ \Sigma_1^* \cdot ((\Sigma_3 \cdot \Sigma^*) \cup \{\varepsilon\}), & \text{if } \delta(¢) = \mathsf{MVR} \text{ and } \delta(\$) = \mathsf{Accept}, \end{cases}$$

and the language $L(M)$ is characterized as

$$L(M) = \begin{cases} \Sigma^*, & \text{if } \delta(¢) = \mathsf{Accept}, \\ (\Sigma_1 \cup \Sigma_2)^* \cdot \Sigma_3 \cdot \Sigma^*, & \text{if } \delta(¢) = \mathsf{MVR} \text{ and } \delta(\$) \neq \mathsf{Accept}, \\ (\Sigma_1 \cup \Sigma_2)^* \cdot ((\Sigma_3 \cdot \Sigma^*) \cup \{\varepsilon\}), & \text{if } \delta(¢) = \mathsf{MVR} \text{ and } \delta(\$) = \mathsf{Accept}. \end{cases}$$

It is easily seen that a stateless finite-state acceptor with input alphabet Σ accepts a language of the form Σ_0^*, where Σ_0 is a subalphabet of Σ. Thus, it follows that a language L is accepted by a stateless deterministic R(1)-automaton that only accepts on reaching the right delimiter \$, if and only if it is accepted by a stateless finite-state acceptor.

For each $n \geq 1$, the Dyck language $D_n'^*$ over n pairs of brackets (see, e.g., [1]) is accepted by a stateless deterministic $\mathsf{R}(2)$-automaton. Further, it can be shown that these automata are necessarily monotone (see, e.g., [4]), which implies that $\mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(2)) \subseteq \mathsf{DCFL}$. Actually this inclusion is a proper one, as shown by the following result.

Lemma 1. *For each integer $k \geq 1$, the regular language $L_k = \{ (ab^k)^i \mid i \geq 0 \}$ satisfies $L_k \in \mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k+1)) \smallsetminus \mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k))$.*

Thus, the language classes $(\mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k)))_{k \geq 1}$ form an infinite strictly increasing sequence, where for all $k \geq 2$, the class $\mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k))$ is incomparable under inclusion to the class REG of regular languages. In [5] a non-context-free language $L_{\text{expo}}^{(\varphi)}$ is presented that is accepted by a stateless deterministic R-automaton of window size 9. It follows that, for all $k \geq 9$, the class $\mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k))$ is incomparable under inclusion to the class CFL of context-free languages. What is the smallest integer k such that the class $\mathcal{L}(\mathsf{stl\text{-}det\text{-}R}(k))$ contains a non-context-free language?

3 CD-Systems of Stateless Deterministic R-Automata with Window Size 1

Cooperating distributed systems of restarting automata were introduced and studied in [6]. Here we only consider a very restricted version: *cooperating distributed systems of stateless deterministic* R-*automata* (or $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R}$-systems for short in accordance with the notation introduced in [7]). Such a system consists of a finite collection $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ of stateless deterministic R-automata $M_i = (\Sigma, ¢, \$, k, \delta_i)$ $(i \in I)$, *successor relations* $\sigma_i \subseteq I$ $(i \in I)$, and a subset $I_0 \subseteq I$ of *initial indices.* Here it is required that $I_0 \neq \emptyset$, that $\sigma_i \neq \emptyset$ for all $i \in I$, and that $i \notin \sigma_i$ for all $i \in I$. Various modes of operation have been introduced and studied, but here we are only interested in mode $= 1$ computations.

A computation of $\mathcal{M}$ in mode $= 1$ on an input word w proceeds as follows. First an index $i_0 \in I_0$ is chosen nondeterministically. Then the R-automaton M_{i_0} starts the computation with the initial configuration $¢w\$$, and executes a single cycle. Thereafter an index $i_1 \in \sigma_{i_0}$ is chosen nondeterministically, and M_{i_1} continues the computation by executing a single cycle. This continues until, for some $l \geq 0$, the machine M_{i_l} accepts. Should at some stage the chosen machine M_{i_l} be unable to execute a cycle or to accept, then the computation fails. By $L_{=1}(\mathcal{M})$ we denote the language that the $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R}$-system $\mathcal{M}$ accepts in mode $= 1$. It consists of all words $w \in \Sigma^*$ that are accepted by $\mathcal{M}$ in mode $= 1$ as described above. Finally, $\mathcal{L}_{=1}(\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R}(i))$ denotes the class of languages that are accepted by mode $= 1$ computations of $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R}$-systems with window size i. The following example illustrates the expressive power of these systems.

Example 1. The language $L_{\text{copy}} = \{\, wcw \mid w \in \{a,b\}^* \,\}$ is not even growing context-sensitive (see, e.g., [8]). However, $L_{\text{copy}} \in \mathcal{L}_{=1}(\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(2)})$, as it is accepted by the mode $= 1$ computations of the following stl-det-local-CD-R-system $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$. Here $I = \{a, b, -, +\}$, $I_0 = \{a, b, +\}$, $\sigma_a = \{-\} = \sigma_b$, $\sigma_- = \{a, b, +\}$, $\sigma_+ = \{-\}$, and M_a, M_b, M_-, and M_+ are given by the following transition functions:

$$\begin{array}{lllll}
M_a & : (1.)\ \delta_a(\text{¢}a) & = \mathsf{MVR}, & \\
 & \ (2.)\ \delta_a(xy) & = \mathsf{MVR} & \text{for all } x \in \{a,b\} \text{ and } y \in \{a,b,c\}, \\
 & \ (3.)\ \delta_a(ca) & = c, & \\
M_b & : (4.)\ \delta_b(\text{¢}b) & = \mathsf{MVR}, & \\
 & \ (5.)\ \delta_b(xy) & = \mathsf{MVR} & \text{for all } x \in \{a,b\} \text{ and } y \in \{a,b,c\}, \\
 & \ (6.)\ \delta_b(cb) & = c, & \\
M_- & : (7.)\ \delta_-(\text{¢}x) & = \text{¢} & \text{for all } x \in \{a,b\}, \\
M_+ & : (8.)\ \delta_+(\text{¢}c) & = \mathsf{MVR}, & \\
 & \ (9.)\ \delta_+(c\$) & = \mathsf{Accept}. &
\end{array}$$

We now concentrate on the class of CD-systems of stateless deterministic R-automata of window size 1. As shown by Proposition 1 stateless deterministic R-automata of window size 1 can only accept very special regular languages. So it is certainly of interest to investigate the expressive power of CD-systems of restarting automata of this very restricted form.

Proposition 2. *The Dyck language $D_1'^*$ is accepted by a CD-system of stateless deterministic* R*-automata of window size 1 working in mode* $= 1$.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$, where $I = \{a, b, +\}$, $I_0 = \{a, +\}$, $\sigma_a = \{b\}$, $\sigma_b = \{a, +\}$, $\sigma_+ = \{a\}$, and M_a, M_b, and M_+ are the stateless deterministic R-automata of window size 1 that are given by the following transition functions:

$$\begin{array}{llll}
M_a : (1.)\ \delta_a(\text{¢}) = \mathsf{MVR}, & M_b : (3.)\ \delta_b(\text{¢}) = \mathsf{MVR}, & M_+ : (6.)\ \delta_+(\text{¢}) = \mathsf{MVR}, \\
\ (2.)\ \delta_a(a) = \varepsilon, & \ (4.)\ \delta_b(a) = \ \mathsf{MVR}, & \ (7.)\ \delta_+(\$) = \mathsf{Accept}. \\
 & \ (5.)\ \delta_b(b) = \quad \varepsilon, &
\end{array}$$

Let $w \in \{a,b\}^*$ be given as input. The automaton M_+ accepts the empty word and rejects (that is, gets stuck on) all other inputs. If $w \neq \varepsilon$, then the computation starts with M_a. If $w = aw_1$, then M_a simply deletes the first occurrence of a in w, otherwise, it gets stuck. Then M_b takes over, which deletes the first occurrence of the letter b, provided $|w_1|_b > 0$. Now this sequence consisting of two cycles is repeated until either the empty word is reached, and then the computation finishes with M_+ accepting, or until a non-empty word is reached that does not start with the letter a, or that does not contain the letter b, and then the computation gets stuck. It follows that $L_{=1}(\mathcal{M}) = D_1'^*$. □

Also the following result is easily shown.

Proposition 3. *The language $L_{abc} = \{\, w \in \{a,b,c\}^* \mid |w|_a = |w|_b = |w|_c \geq 0 \,\}$ is accepted by a CD-system of stateless deterministic* R*-automata of window size 1 working in mode* $= 1$.

On the other hand, all regular languages are accepted by $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$-systems working in mode $= 1$.

Proposition 4. $\mathsf{REG} \subsetneq \mathcal{L}_{=1}(\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)})$.

Proof. Let $L \subseteq \Sigma^*$ be a regular language, and let $A = (Q, \Sigma, p_0, F, \delta)$ be a complete deterministic finite-state acceptor for L. From A we construct a $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$-system $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ as follows:

– The set of indices is $I = (Q \times \Sigma) \cup (Q' \times \Sigma) \cup \{+\}$, where $Q' = \{ q' \mid q \in Q \}$ is a copy of Q such that $Q \cap Q' = \emptyset$,

– the set of initial indices is $I_0 = \begin{cases} \{ (p_0, a) \mid a \in \Sigma \}, & \text{if } \varepsilon \notin L, \\ \{ (p_0, a) \mid a \in \Sigma \} \cup \{+\}, & \text{if } \varepsilon \in L, \end{cases}$

– the successor relations are defined by

- $\sigma_{(q,a)} = \begin{cases} \{ (\delta(q,a), b) \mid b \in \Sigma \} \cup \{+\}, & \text{if } \delta(q,a) \neq q \text{ and } \delta(q,a) \in F, \\ \{ (\delta(q,a), b) \mid b \in \Sigma \}, & \text{if } \delta(q,a) \neq q \text{ and } \delta(q,a) \notin F, \\ \{ (q', b) \mid b \in \Sigma \} \cup \{+\}, & \text{if } \delta(q,a) = q \text{ and } q \in F, \\ \{ (q', b) \mid b \in \Sigma \}, & \text{if } \delta(q,a) = q \text{ and } q \notin F, \end{cases}$
- $\sigma_{(q',a)} = \begin{cases} \{ (\delta(q,a), b) \mid b \in \Sigma \} \cup \{+\}, & \text{if } \delta(q,a) \in F, \\ \{ (\delta(q,a), b) \mid b \in \Sigma \}, & \text{if } \delta(q,a) \notin F, \end{cases}$
- $\sigma_+ = \{ (p_0, a) \mid a \in \Sigma \}$,

– and the $\mathsf{stl\text{-}det\text{-}R(1)}$-automata $M_{(q,a)}$, $M_{(q',a)}$, and M_+ are defined by the following transition functions:

$$\begin{array}{lll} M_{(q,a)} : \delta_{(q,a)}(\text{¢}) = \mathsf{MVR}, & M_{(q',a)} : \delta_{(q',a)}(\text{¢}) = \mathsf{MVR}, & M_+ : \delta_+(\text{¢}) = \mathsf{MVR}, \\ \qquad\quad \delta_{(q,a)}(a) = \varepsilon, & \qquad\quad \delta_{(q',a)}(a) = \varepsilon, & \qquad \delta_+(\$) = \mathsf{Accept}. \end{array}$$

Then it can be checked easily that the accepting mode $= 1$ computations of $\mathcal{M}$ correspond one-to-one to the accepting computations of the finite-state acceptor A. Thus, $L = L(A) = L_{=1}(\mathcal{M})$ holds. □

Next we introduce a normal form for $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$-systems.

Definition 2. *A* $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$*-system* $\mathcal{M} = ((M_i, \sigma_i)_{i \in i}, I_0)$ *is in* normal form*, if it satisfies the following three conditions for all* $i \in I$*, where* $\Sigma_1^{(i)}, \Sigma_2^{(i)}, \Sigma_3^{(i)}, \Sigma_4^{(i)}$ *is the partitioning of alphabet* Σ *from Definition 1 for the automaton* M_i *:*

$$(1.)\ |\Sigma_2^{(i)}| \leq 1,\ (2.)\ \delta_i(\text{¢}) = \mathsf{MVR} \textit{ and } \Sigma_3^{(i)} = \emptyset,\ (3.)\ \Sigma_2^{(i)} = \emptyset,\ \textit{if } \delta_i(\$) = \mathsf{Accept}.$$

If $\mathcal{M}$ is in normal form, and $\Sigma_i^{(2)} = \emptyset$ and $\delta_i(\$) \neq \mathsf{Accept}$ for some index i, then M_i cannot be used in any accepting computation of $\mathcal{M}$, that is, we could simply drop M_i from $\mathcal{M}$. Hence, we can assume that $\delta_i(\$) = \mathsf{Accept}$ if and only if $\Sigma_i^{(2)} = \emptyset$.

By splitting each component automaton M_i of a $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$-system $\mathcal{M}$ into $|\Sigma_2^{(i)}| + 1$ many parts, $M_i^{(a)}$ for $a \in \Sigma_2^{(i)}$, and $M_i^{(+)}$, where the former is responsible for executing the cycles of M_i in which an occurrence of the letter a is deleted, while the latter takes care of the accepting tail computations of M_i, we can prove the following technical result.

Lemma 2. *From a* stl-det-local-CD-R(1)*-system* $\mathcal{M}$ *one can construct a* stl-det-local-CD-R(1)*-system* $\mathcal{M}'$ *in normal form such that* $L_{=1}(\mathcal{M}') = L_{=1}(\mathcal{M})$.

Our next result implies that $\mathcal{L}_{=1}(\text{stl-det-local-CD-R}(1))$ only contains semi-linear languages, that is, if $L \subseteq \Sigma^*$ belongs to this language class, and if $|\Sigma| = n$, then the Parikh image $\psi(L)$ of L is a semi-linear subset of $\mathbb{N}^n$.

Theorem 1. *Each language* $L \in \mathcal{L}_{=1}(\text{stl-det-local-CD-R}(1))$ *contains a regular sublanguage* E *such that* $\psi(L) = \psi(E)$ *holds. In fact, a finite-state acceptor for* E *can be constructed effectively from a* stl-det-local-CD-R(1)*-system for* L.

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a stl-det-local-CD-R(1)-system over Σ, and let $L = L_{=1}(\mathcal{M})$. By Lemma 2 we can assume that $\mathcal{M}$ is in normal form. From $\mathcal{M}$ we construct a nondeterministic finite-state acceptor (NFA) A over Σ such that the language $L(A)$ is letter-equivalent to L.

For each index $i \in I$, let $M_i = (\Sigma, ¢, \$, 1, \delta_i)$, and let $\Sigma = \Sigma_1^{(i)} \cup \Sigma_2^{(i)} \cup \Sigma_3^{(i)} \cup \Sigma_4^{(i)}$ be the partitioning of Σ associated with M_i (see Definition 1). As $\mathcal{M}$ is in normal form, we see that $\Sigma_3^{(i)} = \emptyset$ and $|\Sigma_2^{(i)}| \leq 1$ for each $i \in I$. Further, we know that $\delta_i(¢) = \text{MVR}$, and $\delta_i(\$) = \text{Accept}$ if and only if $\Sigma_2^{(i)} = \emptyset$.

We define an NFA $A = (Q, \Sigma, q_0, F, \delta_A)$ with ε-transitions as follows:

– The set of states Q and the set of final states F are defined by

$$Q = I \cup \{q_0\} \cup \{\, q_\Delta \mid \Delta \subseteq \Sigma \,\} \text{ and } F = \{\, q_\Delta \mid \Delta \subseteq \Sigma \,\}.$$

– The transition relation δ_A is defined by:

$$\begin{array}{lll} (1)\ \delta_A(q_0, \varepsilon) & = I_0, & \\ (2)\ \delta_A(i, a) & = \sigma_i & \text{for all } i \in I \text{ such that } a \in \Sigma_2^{(i)}, \\ (3)\ \delta_A(i, \varepsilon) & = \{q_{\Sigma_1^{(i)}}\} & \text{for all } i \in I \text{ such that } \delta_i(\$) = \text{Accept}, \\ (4)\ \delta_A(q_\Delta, a) & = \{q_\Delta\} & \text{for all } \Delta \subseteq \Sigma \text{ and } a \in \Delta. \end{array}$$

Obviously $L(A)$ is a regular language over Σ. It remains to prove that $L(A)$ is a sublanguage of $L = L_{=1}(\mathcal{M})$ that is letter-equivalent to L. We first establish the following related technical result.

Claim 1. If $w = w_0 \vdash^c_{M_{i_1}} w_1 \vdash^c_{M_{i_2}} \cdots \vdash^c_{M_{i_s}} w_s \vdash^*_{M_{i_{s+1}}} \text{Accept}$ is a mode $= 1$ computation of $\mathcal{M}$, then there exists a word $z \in \Sigma^*$ such that $i_1 z \vdash^*_A q \in F$ holds, and $\psi(z) = \psi(w)$.

Proof. We proceed by induction on the number s of cycles in the above computation. If $s = 0$, then $w = w_s$ is accepted by M_{i_1} through a tail computation. Thus, $w \in {\Sigma_1^{(i_1)}}^*$ and $\delta_{i_1}(\$) = \text{Accept}$. Hence, A can perform the computation $i_1 w \vdash^{(3)}_A q_{\Sigma_1^{(i_1)}} w \vdash^{(4)^*}_A q_{\Sigma_1^{(i_1)}} \in F$. Thus, A accepts starting from $i_1 w$.

If $w = xay \vdash^c_{M_{i_1}} xy$, then $x \in {\Sigma_1^{(i_1)}}^*$, $a \in \Sigma_2^{(i_1)}$, and $i_2 \in \sigma_{i_1}$. Thus, A can perform the step $i_1 axy \vdash^{(2)}_A i_2 xy$. From the induction hypothesis we see that there exists a word $z_1 \in \Sigma^*$ that is accepted by A starting from the configuation $i_2 z_1$, and that is letter-equivalent to $w_1 = xy$. Hence, the word $z = az_1$ is accepted by A starting from the configuration $i_1 a z_1$, and az_1 is letter-equivalent to axy and therewith to $w = xay$. This completes the proof of Claim 1. $\square$

If $w \in L_{=1}(\mathcal{M})$, then there exists an accepting mode $= 1$ computation of $\mathcal{M}$ of the form $w = w_0 \vdash^c_{M_{i_1}} w_1 \vdash^c_{M_{i_2}} \cdots \vdash^c_{M_{i_s}} w_s \vdash^*_{M_{i_{s+1}}} \mathsf{Accept}$. Then $i_1 \in I_0$, and by Claim 1 there exists a word $z \in \Sigma^*$ such that z is letter-equivalent to w, and A accepts starting from the configuration $i_1 z$. But then $i_1 \in \delta_A(q_0, \varepsilon)$ implies that A accepts starting from the initial configuration $q_0 z$. Thus, for each word $w \in L_{=1}(\mathcal{M})$, there exists a word $z \in L(A)$ such that z and w are letter-equivalent.

To complete the proof we establish the following claim.

Claim 2. If $z \in \Sigma^*$ and $i \in I$ such that A accepts starting from the configuration iz, then $\mathcal{M}$ has an accepting mode $= 1$ computation in which component automaton M_i starts from the initial tape contents $¢z\$$.

Proof. We proceed by induction on the number of steps of group (2) that are applied in the accepting computation of A.

If no such step is applied at all, then the accepting computation of A has the form $iz \vdash^{(3)}_A q_{\Sigma_1^{(i)}} z \vdash^{(4)^*}_A q_{\Sigma_1^{(i)}} \in F$. From the definition of A we see that $\delta_i(\$) = \mathsf{Accept}$, and hence, component automaton M_i will accept starting from the tape contents $¢z\$$.

Now assume that the accepting computation of A is of the form $iz = iav \vdash^{(2)}_A jv \vdash^*_A q_\Delta$, where $a \in \Sigma$, and $\Delta \subseteq \Sigma$. From the definition of A we see that $\delta_i(a) = \varepsilon$, and that $j \in \sigma_i$. Further, from the induction hypothesis we know that $\mathcal{M}$ has an accepting mode $= 1$ computation in which M_j starts from the tape contents $¢v\$$. It follows that there exists an accepting mode $= 1$ computation of $\mathcal{M}$ in which M_i starts with tape contents $¢av\$ = ¢z\$$. $\square$

It follows that each word $z \in L(A)$ belongs to the language $L_{=1}(\mathcal{M})$. Thus, $L(A)$ is indeed a regular sublanguage of L that is letter-equivalent to L. $\square$

As all regular languages have semi-linear Parikh image, this yields the following important result.

Corollary 1. *The language class* $\mathcal{L}_{=1}(\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)})$ *only contains languages for which the Parikh image* $\psi(L)$ *is semi-linear.*

As the deterministic linear language $L = \{ a^n b^n \mid n \geq 0 \}$ does not contain a regular sublanguage that is letter-equivalent to the language itself, we obtain the following non-inclusion result.

Proposition 5. *The language* $L = \{ a^n b^n \mid n \geq 0 \}$ *is not accepted by any* $\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)}$*-system working in mode* $= 1$.

Together with Proposition 3 this gives the following incomparability result.

Corollary 2. *The language class* $\mathcal{L}_{=1}(\mathsf{stl\text{-}det\text{-}local\text{-}CD\text{-}R(1)})$ *is incomparable to the classes* DLIN, LIN, DCFL, *and* CFL *with respect to inclusion.*

4 Rational Trace Languages

A *dependency relation* D is a binary relation on an alphabet Σ that is reflexive and symmetric. Then $I_D = (\Sigma \times \Sigma) \smallsetminus D$ is the corresponding *independence relation*. Obviously, the relation I_D is irreflexive and symmetric. It induces a binary relation $\equiv_D$ on Σ^* that is defined as the smallest congruence relation containing the set of pairs $\{ (ab, ba) \mid (a, b) \in I_D \}$. For $w \in \Sigma^*$, the congruence class of w mod $\equiv_D$ is denoted by $[w]_D$. These congruence classes are called *traces*, and the factor monoid $M(D) = \Sigma^*/\equiv_D$ is a *trace monoid*. By φ_D we denote the morphism $\varphi_D : \Sigma^* \to M(D)$ that is defined by $w \mapsto [w]_D$ for all words $w \in \Sigma^*$.

To simplify the notation in what follows, we introduce the following notions. For $w \in \Sigma^*$, we use $\mathrm{Alph}(w)$ to denote the set of all letters that occur in w. Then the independence relation can be extended from letters to words by defining, for all words $u, v \in \Sigma^*$, $(u, v) \in I_D$ if and only if $\mathrm{Alph}(u) \times \mathrm{Alph}(v) \subseteq I_D$. As $\mathrm{Alph}(\varepsilon) = \emptyset$, we see that $(\varepsilon, w) \in I_D$ for every word $w \in \Sigma^*$.

A subset S of a trace monoid $M(D)$ is called *recognizable* if there exist a finite monoid N, a morphism $\alpha : M(D) \to N$, and a subset P of N such that $S = \alpha^{-1}(P)$ [1]. Accordingly, $S \subseteq M(D)$ is recognizable if and only if the language $\varphi_D^{-1}(S)$ is a regular language over Σ. By $\mathsf{REC}(M(D))$ we denote the set of recognizable subsets of $M(D)$. A subset S of a trace monoid $M(D)$ is called *rational* if it can be obtained from singleton sets by a finite number of unions, products, and star operations [1]. It follows that $S \subseteq M(D)$ is rational if and only if there exists a regular language L over Σ such that $S = \varphi_D(L)$. By $\mathsf{RAT}(M(D))$ we denote the set of rational subsets of $M(D)$. It is known that $\mathsf{REC}(M(D)) \subseteq \mathsf{RAT}(M(D))$ for each trace monoid $M(D)$, and that these two sets are equal if and only if $I_D = \emptyset$ (see, e.g., [3]). Now we come to our main result.

Theorem 2. *Let $M(D)$ be the trace monoid presented by (Σ, D), where Σ is a finite alphabet. Then the language $\varphi_D^{-1}(S)$ is accepted by a* stl-det-local-CD-R(1)-*system working in mode $= 1$ for each rational set of traces $S \subseteq M(D)$.*

Proof. Let S be a rational subset of $M(D)$. Then there exists a regular language L over Σ such that $S = \varphi_D(L)$. Hence, $\varphi_D^{-1}(S) = \bigcup_{u \in L} [u]_D$.

As $L \subseteq \Sigma^*$ is a regular language, there exists a complete deterministic finite-state acceptor $A = (Q, \Sigma, p_0, F, \delta)$ for L. From A we obtain a stl-det-local-CD-R(1)-system $\mathcal{M} = ((M_i, \sigma_i)_{i \in I}, I_0)$ as in the proof of Proposition 4. This system is now modified by redefining the transition functions of the R-automata $M_{(q,a)}$ and $M_{(q',a)}$ ($q \in Q$, $a \in \Sigma$) as follows:

$$\begin{aligned} M_{(q,a)} : \delta_{(q,a)}(\cent) &= \mathsf{MVR}, \\ \delta_{(q,a)}(b) &= \mathsf{MVR} \quad \text{for all } b \in \Sigma \text{ satisfying } (b, a) \in I_D, \\ \delta_{(q,a)}(a) &= \quad \varepsilon, \end{aligned}$$

$$\begin{aligned} M_{(q',a)} : \delta_{(q',a)}(\cent) &= \mathsf{MVR}, \\ \delta_{(q',a)}(b) &= \mathsf{MVR} \quad \text{for all } b \in \Sigma \text{ satisfying } (b, a) \in I_D, \\ \delta_{(q',a)}(a) &= \quad \varepsilon. \end{aligned}$$

It can now be verified that $L_{=1}(\mathcal{M}) = \bigcup_{u\in L}[u]_D = \varphi_D^{-1}(S)$, which completes the proof of Theorem 2. □

Next we present a restricted class of stl-det-local-CD-R(1)-systems that accept exactly the rational trace languages by mode = 1 computations.

Definition 3. *Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a* stl-det-local-CD-R(1)*-system on Σ that is in normal form and that satisfies the following condition:*

$$(*) \qquad \forall i,j \in I : \Sigma_2^{(i)} = \Sigma_2^{(j)} \text{ implies that } \Sigma_1^{(i)} = \Sigma_1^{(j)}.$$

With $\mathcal{M}$ we associate a binary relation $I_{\mathcal{M}} = \bigcup_{i\in I}(\Sigma_1^{(i)} \times \Sigma_2^{(i)})$, that is, $(a,b) \in I_{\mathcal{M}}$ iff there exists a component automaton M_i such that $\delta_i(a) =$ MVR and $\delta_i(b) = \varepsilon$. Further, by $D_{\mathcal{M}}$ we denote the relation $D_{\mathcal{M}} = (\Sigma \times \Sigma) \smallsetminus I_{\mathcal{M}}$.

Observe that the relation $I_{\mathcal{M}}$ defined above is necessarily irreflexive, but that it will in general not be symmetric. For example, consider the system $\mathcal{M}$ from the proof of Proposition 2. It is in normal form, but the corresponding relation $I_{\mathcal{M}} = \{(a,b)\}$ is not symmetric. And indeed, the language $L_{=1}(\mathcal{M})$ is the Dyck language $D_1'^*$, which is not a rational trace language.

Theorem 3. *Let $\mathcal{M}$ be a* stl-det-local-CD-R(1)*-system over Σ satisfying condition $(*)$ above. If the associated relation $I_{\mathcal{M}}$ is symmetric, then $L_{=1}(\mathcal{M})$ is a rational trace language over Σ. In fact, from $\mathcal{M}$ one can construct a finite-state acceptor B over Σ such that $L_{=1}(\mathcal{M}) = \varphi_{D_{\mathcal{M}}}^{-1}(\varphi_{D_{\mathcal{M}}}(L(B)))$.*

Proof. Let $\mathcal{M} = ((M_i, \sigma_i)_{i\in I}, I_0)$ be a stl-det-local-CD-R(1)-system in normal form on Σ that satisfies condition $(*)$. In addition, we assume that the associated relation $I_{\mathcal{M}} = \bigcup_{i\in I}(\Sigma_1^{(i)} \times \Sigma_2^{(i)})$ is symmetric. Then $D_{\mathcal{M}} = (\Sigma \times \Sigma) \smallsetminus I_{\mathcal{M}}$ is a reflexive and symmetric relation, and so it is a dependency relation on Σ with associated independence relation $I_{\mathcal{M}}$. Without loss of generality we may assume that all letters from Σ do actually occur in some words of $L_{=1}(\mathcal{M})$, since otherwise we could simply remove these letters from Σ. Further, we can assume that $\mathcal{M}$ has only a single accepting component automaton M_+, and that M_+ only accepts the empty word. From the properties of $\mathcal{M}$ we obtain the following consequences:

1. As all words $w \in L_{=1}(\mathcal{M})$ are first reduced to the empty word, which is then accepted by the accepting component automaton of $\mathcal{M}$, we see that, for each letter $a \in \Sigma$, there exists a component automaton M_i such that $\Sigma_2^{(i)} = \{a\}$.
2. If $(a,b) \in I_{\mathcal{M}}$, then $a \in \Sigma_1^{(i)}$ for all component automata M_i for which $\Sigma_2^{(i)} = \{b\}$ holds.
3. If $(a,b) \in I_{\mathcal{M}}$, then $(b,a) \in I_{\mathcal{M}}$, too, and hence, $b \in \Sigma_1^{(j)}$ for all component automata M_j for which $\Sigma_2^{(j)} = \{a\}$ holds.

Let $L = L_{=1}(\mathcal{M})$. We claim that L is a rational trace language over the trace monoid defined by $(\Sigma, D_{\mathcal{M}})$. To verify this claim we present a regular language $R \subseteq \Sigma^*$ such that $L = \bigcup_{u\in R}[u]_{D_{\mathcal{M}}}$.

The regular language R will be defined through a nondeterministic finite-state acceptor (with ε-moves) $B = (Q, \Sigma, p_0, p_+, \delta)$. This finite-state acceptor is obtained from $\mathcal{M}$ as follows. Here $I_r = I \setminus \{+\}$ is the subset of I containing all component automata that perform a rewrite operation, $i \in I_r$, and $a \in \Sigma$:

$$\begin{array}{lll}
Q & = \{p_0, p_+\} \cup \{\, q_i \mid i \in I_r \,\}, & \\
\delta(p_0, \varepsilon) & = \{\, q_i \mid i \in I_0 \,\}, & \text{if } + \notin I_0, \\
\delta(p_0, \varepsilon) & = \{\, q_i \mid i \in I_0 \cap I_r \,\} \cup \{p_+\}, & \text{if } + \in I_0, \\
\delta(q_i, a) & = \{\, q_j \mid j \in \sigma_i \,\}, & \text{if } \{a\} = \Sigma_2^{(i)} \text{ and } + \notin \sigma_i, \\
\delta(q_i, a) & = \{\, q_j \mid j \in \sigma_i \cap I_r \,\} \cup \{p_+\}, & \text{if } \{a\} = \Sigma_2^{(i)} \text{ and } + \in \sigma_i, \\
\delta(q, a) & = \emptyset & \text{for all other cases.}
\end{array}$$

Then $R = L(B)$ is a regular language over Σ. From the properties of $I_{\mathcal{M}}$ it can now be shown that $L = \bigcup_{u \in R} [u]_{D_{\mathcal{M}}}$ holds, that is, $L = \varphi_{D_{\mathcal{M}}}^{-1}(\varphi_{D_{\mathcal{M}}}(L(B)))$. □

Observe that the system $\mathcal{M}$ constructed in the proof of Theorem 2 is in normal form, that it satisfies property $(*)$, and that the associated relation $I_{\mathcal{M}}$ coincides with the relation I_D, and hence, it is symmetric. Thus, Theorems 2 and 3 together yield the following characterization.

Corollary 3. *A language $L \subseteq \Sigma^*$ is a rational trace language if and only if there exists a* stl-det-local-CD-R(1)*-system $\mathcal{M}$ in normal form satisfying condition $(*)$ such that the relation $I_{\mathcal{M}}$ is symmetric and $L = L_{=1}(\mathcal{M})$.*

In the proof of Theorem 2 we effectively constructed a stl-det-local-CD-R(1)-system for the rational trace language $\varphi_D^{-1}(\varphi_D(R))$ from a finite-state acceptor for the regular language R. Hence, if $S_1, S_2 \subseteq M(D)$ are rational subsets of the trace monoid $M(D)$, then we can construct finite-state acceptors B_1 and B_2 from stl-det-local-CD-R(1)-systems $\mathcal{M}_1$ for $L_1 = \varphi_D^{-1}(S_1)$ and $\mathcal{M}_2$ for $L_2 = \varphi_D^{-1}(S_2)$ such that $S_1 = \varphi_D(R_1)$ and $S_2 = \varphi_D(R_2)$, where $R_i = L(B_i)$, $i = 1, 2$. It is easily seen that $S_1 \cup S_2 = \varphi_D(R_1 \cup R_2)$, $S_1 \cdot S_2 = \varphi_D(R_1 \cdot R_2)$, and $S_1^* = \varphi_D(R_1^*)$. From B_1 and B_2 we can construct finite-state acceptors for the languages $R_1 \cup R_2$, $R_1 \cdot R_2$, and R_1^*. Thus, Theorem 3 shows that we can construct stl-det-local-CD-R(1)-systems for the languages $\varphi_D^{-1}(S_1 \cup S_2)$, $\varphi_D^{-1}(S_1 \cdot S_2)$, and $\varphi_D^{-1}(S_1^*)$. Hence, the stl-det-local-CD-R(1)-systems of Corollary 3 form an effective calculus for rational trace languages.

5 Concluding Remarks

We have seen that the stl-det-local-CD-R(1)-systems accept a subclass of all semi-linear languages that contains all rational trace languages, but that this subclass is incomparable to the (deterministic) linear languages and context-free languages. However, it remains open whether this language class can be characterized through other, more traditional, means.

Theorem 1 yields an effective construction of a finite-state acceptor B from a stl-det-local-CD-R(1)-system $\mathcal{M}$ such that the language $E = L(B)$ is a subset of the language $L = L_{=1}(\mathcal{M})$ that is letter-equivalent to L. Hence, E is non-empty if and only if L is non-empty, and E is infinite if and only if L is infinite. Thus, the emptiness problem and the finiteness problem are effectively decidable for stl-det-local-CD-R(1)-systems. On the other hand, it is undecidable in general whether a rational trace language is recognizable (see, e.g., [3]). As a rational subset S of a trace monoid $M(D)$ is recognizable if and only if $\varphi_D^{-1}(S)$ is a regular language, it follows from Corollary 3 that it is undecidable in general whether a given stl-det-local-CD-R(1)-system accepts a regular language, that is, the regularity problem is undecidable for these systems.

In the proof of Proposition 4 we have constructed a stl-det-local-CD-R(1)-system from a deterministic finite-state acceptor. This construction is quite inefficient, as we have used $O(|Q| \cdot |\Sigma|)$ many component automata. Is there a more efficient (that is, more succinct) simulation?

Further, the closure properties and algorithmic problems for the language class $\mathcal{L}_{=1}$(stl-det-local-CD-R(1)) are still to be investigated. For example, it is obvious that this class is closed under union, but that it is not closed under intersection with regular languages. What can be said about other operations like, e.g., product, iteration, or reversal? These questions are addressed in a forthcoming paper.

References

1. Berstel, J.: Transductions and Context-free Languages. In: Teubner Studienbücher: Informatik. Teubner, Stuttgart (1979)
2. Csuhaj-Varjú, E., Martín-Vide, C., Mitrana, V.: Multiset automata. In: Calude, C., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Multiset Processing. LNCS, vol. 2235, pp. 69–83. Springer, Heidelberg (2001)
3. Diekert, V., Rozenberg, G.: The Book of Traces. World Scientific, Singapore (1995)
4. Kutrib, M., Messerschmidt, H., Otto, F.: On stateless two-pushdown automata and restarting automata. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) Automata and Formal Languages, AFL 2008, Proc. Computer and Automation Research Institute, pp. 257–268. Hungarian Academy of Sciences (2008)
5. Kutrib, M., Messerschmidt, H., Otto, F.: On stateless two-pushdown automata and restarting automata. Intern. J. Found. Comput. Sci (to appear, 2010); Extended version of [4]
6. Messerschmidt, H., Otto, F.: Cooperating distributed systems of restarting automata. Intern. J. Found. Comput. Sci. 18, 1333–1342 (2007)
7. Messerschmidt, H., Otto, F.: On deterministic CD-systems of restarting automata. Intern. J. Found. Comput. Sci. 20, 185–209 (2009)
8. Otto, F.: Restarting automata. In: Ésik, Z., Martin-Vide, C., Mitrana, V. (eds.) Recent Advances in Formal Languages and Applications. Studies in Computational Intelligence, vol. 25, pp. 269–303. Springer, Berlin (2006)

A Boundary between Universality and Non-universality in Extended Spiking Neural P Systems*

Turlough Neary

Boole Centre for Research in Informatics, University College Cork, Ireland
tneary@cs.nuim.ie

Abstract. We solve the problem of finding the smallest possible universal spiking neural P system with extended rules. We give a universal spiking neural P system with extended rules and only 4 neurons. This is the smallest possible universal system of its kind. We prove this by showing that the set of problems solved by spiking neural P systems with 3 neurons is bounded above by NL, and so there exists no such universal system with 3 neurons (for any reasonable definition of universality). Finally, we show that if we generalise the output technique we can give a universal spiking neural P system with extended rules that has only 3 neurons. This is also the smallest possible universal system of its kind.

1 Introduction

Spiking neural P systems (SN P systems) [1] are quite a new computational model that are a synergy inspired by P systems and spiking neural networks. Here we solve the problem of finding the smallest possible universal spiking neural P system with extended rules; one of the open problems given in [9]. We give a universal extended SN P system that has only only 4 neurons. Following this, we prove that the set of problems solved by spiking neural P systems with 3 neurons is bounded above by NL, and so there exists no such universal system with 3 neurons (for any reasonable definition of universality). Thus, our 4-neuron system is the smallest possible universal extended SN P system. Finally, we show that if we generalise the output technique we can give a universal SN P system with extended rules that has only 3 neurons. This is also the smallest possible universal system of its kind. Table 1 gives the smallest universal extended SN P systems and their respective simulation time and space overheads. For more on the time/space complexity of small universal SNP systems see [4,6].

In their paper containing the 49-neuron system, Păun and Păun [8] state that a significant decrease on the number of neurons of their two universal SN P systems is improbable (also stated in [9]). The dramatic improvement on the

* Turlough Neary is funded by Science Foundation Ireland Research Frontiers Programme grant number 07/RFP/CSMF641. I would also like to thank Damien Woods for his helpful suggestions and comments.

Table 1. Small universal extended SN P systems. The "simulation time/space" column gives the overheads used by each system when simulating a standard single tape Turing machine. † A more generalised output technique is used. ‡ The 18 neuron system is not explicitly given in [5]; it was presented at [3] and is easily derived from the other system in [5]. *The system in [7] does not include the input module. This is not the case for all the other systems in this table. Note that if we remove the input module from the 18- and 12-neuron systems in [5] we get 12- and 9-neuron systems, respectively.

number of neurons	simulation time/space	author
49	exponential	Păun & Păun [8]
41	exponential	Zhang et al. [10]
18	exponential	Neary [5,3]‡
12	double-exponential	Neary [5]
12	exponential	Pan & Zeng [7]*
4	**exponential**	**Section 4**
3	**exponential**	**Section 5†**

size of earlier small universal SN P systems given by Theorem 1 is in part due to the method we use to encode the instructions of the counter machines being simulated. All of the SN P systems given in Table 1 simulate counter machines. The size of previous small universal systems [8,10] were dependant on the number of instructions in the counter machine being simulated. In our systems each unique counter machine instruction is encoded as a unique number of spikes, and thus the size of our SN P systems is independent of the number of counter machine instructions. The technique of encoding the instructions as spikes was first used to construct small universal SN P systems in [5] (see Table 1).

2 SN P Systems

Definition 1 (Spiking neural P system). *A spiking neural P system (SN P system) is a tuple $\Pi = (O, \sigma_1, \sigma_2, \cdots, \sigma_m, syn, in, out)$, where:*

1. *$O = \{s\}$ is the unary alphabet (s is known as a spike),*
2. *$\sigma_1, \sigma_2, \cdots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i), 1 \leqslant i \leqslant m$, where:*
   (a) *$n_i \geqslant 0$ is the initial number of spikes contained in σ_i,*
   (b) *R_i is a finite set of rules of the following two forms:*
      i. *$E/s^b \rightarrow s; d$, where E is a regular expression over s, $b \geqslant 1$ and $d \geqslant 0$,*
      ii. *$s^e \rightarrow \lambda$, where λ is the empty word, $e \geqslant 1$, and for all $E/s^b \rightarrow s; d$ from R_i $s^e \notin L(E)$ where $L(E)$ is the language defined by E,*
3. *$syn \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\}$ is the set of synapses between neurons, where $i \neq j$ for all $(i, j) \in syn$,*
4. *$in, out \in \{\sigma_1, \sigma_2, \cdots, \sigma_m\}$ are the input and output neurons, respectively.*

A firing rule $r = E/s^b \rightarrow s; d$ is applicable in a neuron σ_i if there are $j \geqslant b$ spikes in σ_i and $s^j \in L(E)$ where $L(E)$ is the set of words defined by the regular expression E. If, at time t, rule r is executed then b spikes are removed from the neuron, and at time $t+d$ the neuron fires. When a neuron σ_i fires a spike is sent to each neuron σ_j for every synapse (i,j) in Π. Also, the neuron σ_i remains closed and does not receive spikes until time $t+d$ and no other rule may execute in σ_i until time $t+d+1$. A forgeting rule $r' = s^e \rightarrow \lambda$ is applicable in a neuron σ_i if there are exactly e spikes in σ_i. If r' is executed then e spikes are removed from the neuron. At each timestep t a rule must be applied in each neuron if there is one or more applicable rules. Thus, while the application of rules in each individual neuron is sequential the neurons operate in parallel with each other.

Note from 2b(i) of Definition 1 that there may be two rules of the form $E/s^b \rightarrow s; d$, that are applicable in a single neuron at a given time. If this is the case then the next rule to execute is chosen non-deterministically.

An *extended* SN P system [8] has more general rules of the form $E/s^b \rightarrow s^p; d$, where $b \geqslant p \geqslant 1$. Thus, a synapse in a SN P system with extended rules may transmit more than one spike in a single timestep. The SN P systems we present in this work use rules without delay, and thus in the sequel we write rules as $E/s^b \rightarrow s^p$. Also, if in a rule $E = s^b$ then we write the rule as $s^b \rightarrow s^p$.

Spikes are introduced into the system from the environment by reading in a binary sequence (or word) $w \in \{0,1\}$ via the input neuron σ_1. The sequence w is read from left to right one symbol at each timestep and a spike enters the input neuron from the environment on a given timestep iff the read symbol is 1. The output of a SN P system Π is the time between the first and second timesteps a firing rule is applied in the output neuron.

3 Counter Machines

Definition 2. *A counter machine is a tuple $C = (z, R, c_m, Q, q_1, q_h)$, where z gives the number of counters, R is the set of input counters, c_m is the output counter, $Q = \{q_1, q_2, \cdots, q_h\}$ is the set of instructions, and $q_1, q_h \in Q$ are the initial and halt instructions, respectively.*

Each counter c_j stores a natural number value $y \geqslant 0$. Each instruction q_i is of one of the following two forms:

– $q_i : INC(j)$ increment the value y stored in counter c_j by 1 and move to instruction q_l.
– $q_i : DEC(j)q_k$ if the value y stored in counter c_j is greater than 0 then decrement this value by 1 and move to instruction q_l, otherwise if $y = 0$ move to instruction q_k.

At the beginning of a computation the first instruction executed is q_1. The input to the counter machine is initially stored in the input counters. If the counter machine's control enters instruction q_h, then the computation halts at that timestep. The result of the computation is the value y stored in the output counter c_m when the computation halts.

In earlier work [8], Korec's notion of strong universality was adopted for small SN P systems as follows: if the input to ϕ_x is y then the input to the SN P system

Table 2. Rules for each of the neurons of Π_C. Here $1 \leqslant i < h$, $1 \leqslant l \leqslant h$ and $1 \leqslant k \leqslant h$.

neuron	rules
σ_1	$(s^{12h})^* s^{6h+8}/s^{6h} \rightarrow s^{6h}$, $(s^{12h})^* s^{6h+9}/s^{9h+5} \rightarrow s^{3h+2}$, $(s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}$, if $q_i : INC(1), q_i : DEC(1) \notin \{Q\}$ $(s^{12h})^* s^{6(h+i)+4}/s^{12h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : INC(1) \in \{Q\}$ $(s^{12h})^* s^{54h+6i+4}/s^{36h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : DEC(1) \in \{Q\}$ $s^{42h+6i+4}/s^{24h+6i+4-6k} \rightarrow s^{6k}$, if $q_i : DEC(1) \in \{Q\}$
σ_2	$s^{18h+7}/s^{6h} \rightarrow s^{6h}$, $(s^{12h})^* s^{6h+9}/s^{9h+5} \rightarrow s^{3h+2}$, $(s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}$, if $q_i : INC(2), q_i : DEC(2) \notin \{Q\}$ $(s^{12h})^* s^{6(h+i)+4}/s^{12h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : INC(2) \in \{Q\}$ $(s^{12h})^* s^{54h+6i+4}/s^{36h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : DEC(2) \in \{Q\}$ $s^{42h+6i+4}/s^{24h+6i+4-6k} \rightarrow s^{6k}$, if $q_i : DEC(2) \in \{Q\}$
σ_3	$s^{18h+7}/s^{6h} \rightarrow s^{6h}$, $(s^{12h})^*/s^{12h+1} \rightarrow s^{12h}$, $s^{18h+2}/s^{6h} \rightarrow s^{6h}$, $s^{24h-1} \rightarrow s^{12h}$, $s^{18h+3}/s^{6h+1} \rightarrow s^{6h+1}$, $s^{18h+8}/s^{6h+6} \rightarrow s^{6h+1}$, $(s^{12h})^* s^{36h-1}/s^{12h} \rightarrow s^{6h}$, $(s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}$, if $q_i : INC(3), q_i : DEC(3) \notin \{Q\}$ $(s^{12h})^* s^{6(h+i)+4}/s^{12h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : INC(3) \in \{Q\}$ $(s^{12h})^* s^{54h+6i+4}/s^{36h+6i+4-6l} \rightarrow s^{6l}$, if $q_i : DEC(3) \in \{Q\}$ $s^{42h+6i+10}/s^{24h+6i+4-6k} \rightarrow s^{6k}$, if $q_i : DEC(3) \in \{Q\}$
σ_4	$s^{6h} \rightarrow \lambda$, $s^{6h+1} \rightarrow \lambda$, $s^{12h+2} \rightarrow \lambda$, $s^{6l} \rightarrow \lambda$, $s^{12h} \rightarrow s$

is the sequence $10^{y-1}10^{x-1}1$. As with the SN P systems given in [8,10], the system we give in Theorem 1 satisfies the notion of strong universality adopted from Korec in [8]. However, as we noted in other work [6], it could be considered that Korec's notion [2] of strong universality is somewhat arbitrary and we also pointed out some inconsistency in his notion of weak universality. Hence, in this work we rely on time/space complexity analysis to compare small SN P systems and their encodings (see Table 1).

4 A Small Universal Extended SN P System

Theorem 1. *Let C be a universal counter machine with 3 counters that completes its computation in time t to give the output value x_3 when given the pair of input values (x_1, x_2). Then there is a universal extended SN P system Π_C that simulates the computation of C in time $O(t + x_1 + x_2 + x_3)$ and has only 4 neurons.*

Proof. Let $C = (3, \{c_1, c_2\}, c_3, Q, q_1, q_h)$ where $Q = \{q_1, q_2, \cdots, q_h\}$. Our SN P system Π_C is given by Figure 1 and Table 2. Π_C is deterministic.

Encoding of a configuration of C and reading input into Π_C. A configuration of C is stored as spikes in the neurons of Π_C. The next instruction q_i to be executed is stored in each of the neurons σ_1, σ_2 and σ_3 as $6(h+i)$ spikes. Let

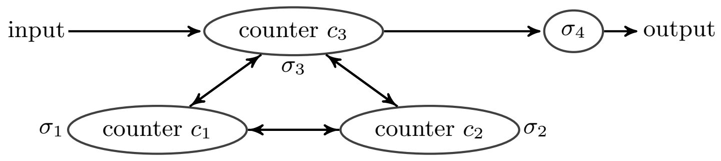


Fig. 1. Universal extended SN P system Π_C. Each oval labeled σ_i is a neuron. An arrow going from neuron σ_i to neuron σ_j illustrates a synapse (i, j).

x_1, x_2 and x_3 be the values stored in counters c_1, c_2 and c_3, respectively. Then the values x_1, x_2 and x_3 are stored as $12h(x_1+1)$, $12h(x_2+1)$ and $12h(x_3+1)$ spikes in neurons σ_1, σ_2 and σ_3, respectively. The input to Π_C is read into the system via the input neuron σ_3 (see Figure 1). If C begins its computation with the values x_1 and x_2 in counters c_1 and c_2, respectively, then the binary sequence $w = 10^{x_1-1}10^{x_2-1}1$ is read in via the input neuron σ_3. Thus, σ_3 receives a single spike from the environment at times t_1, t_{x_1+1} and $t_{x_1+x_2+1}$. We explain how the system is initialised to encode an initial configuration of C by giving the number of spikes in each neuron and the rule that is to be applied in each neuron at time t. Before the computation begins neuron σ_1 contains $6h+7$ spikes, σ_2 contains $18h+7$ spikes, σ_3 contains $18h+6$ spikes and σ_4 contains no spikes. Thus, when σ_3 receives it first spike at time t_1 we have

$$\begin{array}{lll} t_1: & \sigma_1 = 6h+7, & \\ & \sigma_2, \sigma_3 = 18h+7, & s^{18h+7}/s^{6h} \rightarrow s^{6h}. \end{array}$$

where on the left $\sigma_k = z$ gives the number z of spikes in neuron σ_k at time t and on the right is the rule that is to be applied at time t, if there is an applicable rule at that time. Thus, from Figure 1, when we apply the rule $s^{18h+7}/s^{6h} \rightarrow s^{6h}$ in neurons σ_2 and σ_3 at time t_1 we get

$$\begin{array}{lll} t_2: & \sigma_1 = 18h+7, & \\ & \sigma_2, \sigma_3 = 18h+7, & s^{18h+7}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda, \end{array}$$

$$\begin{array}{lll} t_3: & \sigma_1 = 30h+7, & \\ & \sigma_2, \sigma_3 = 18h+7, & s^{18h+7}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda. \end{array}$$

Neurons σ_2 and σ_3 send $12h$ spikes to neuron σ_1 on each timestep between times t_1 and t_{x_1+1}. This gives a total of $12hx_1$ spikes sent to σ_1 during the time interval t_1 to t_{x_1+1}. Thus when σ_3 receives the second spike from the environment we get

$$\begin{aligned} t_{x_1+1}: \quad & \sigma_1 = 12hx_1 + 6h + 7, & \\ & \sigma_2 = 18h + 7, & s^{18h+7}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_3 = 18h + 8, & s^{18h+8}/s^{6h+6} \rightarrow s^{6h+1}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{x_1+2}: \quad & \sigma_1 = 12h(x_1+1) + 6h + 8, & (s^{12h})^* s^{6h+8}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_2 = 18h + 8, & \\ & \sigma_3 = 18h + 2, & s^{18h+2}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_4 = 6h + 1, & s^{6h+1} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{x_1+3}: \quad & \sigma_1 = 12h(x_1+1) + 6h + 8, & (s^{12h})^* s^{6h+8}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_2 = 30h + 8, & \\ & \sigma_3 = 18h + 2, & s^{18h+2}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda. \end{aligned}$$

Neurons σ_1 and σ_3 fire on every timestep between times t_{x_1+2} and $t_{x_1+x_2+2}$ to send a total of $12hx_2$ spikes to σ_2. Thus, when σ_3 receives the last spike from the environment we have

$$\begin{aligned} t_{x_1+x_2+1}: \quad & \sigma_1 = 12h(x_1+1) + 6h + 8, & (s^{12h})^* s^{6h+8}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_2 = 12hx_2 + 6h + 8, & \\ & \sigma_3 = 18h + 3, & s^{18h+3}/s^{6h+1} \rightarrow s^{6h+1}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{x_1+x_2+2}: \quad & \sigma_1 = 12h(x_1+1) + 6h + 9, & (s^{12h})^* s^{6h+9}/s^{9h+5} \rightarrow s^{3h+2}, \\ & \sigma_2 = 12h(x_2+1) + 6h + 9, & (s^{12h})^* s^{6h+9}/s^{9h+5} \rightarrow s^{3h+2} \\ & \sigma_3 = 18h + 2, & s^{18h+2}/s^{6h} \rightarrow s^{6h}, \\ & \sigma_4 = 6h + 1, & s^{6h+1} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{x_1+x_2+3}: \quad & \sigma_1 = 12h(x_1+1) + 6(h+1), \\ & \sigma_2 = 12h(x_2+1) + 6(h+1), \\ & \sigma_3 = 12h + 6(h+1). \end{aligned}$$

At time $t_{x_1+x_2+3}$ neuron σ_1 contains $12h(x_1+1) + 6(h+1)$ spikes, σ_2 contains $12h(x_2+1) + 6(h+1)$ spikes and σ_3 contains $12h + 6(h+1)$ spikes. Thus at time $t_{x_1+x_2+3}$ the SN P system encodes an initial configuration of C.

Π_C simulating $q_i : INC(1)$. Let counters c_1, c_2, and c_3 have values x_1, x_2, and x_3, respectively. Then the simulation of $q_i : INC(1)$ begins at time t_j with

$12h(x_1+1)+6(h+i)$ spikes in σ_1, $12h(x_2+1)+6(h+i)$ spikes in σ_2 and $12h(x_3+1)+6(h+i)$ spikes in σ_3. Thus, at time t_j we have

$$\begin{aligned} t_j: \quad & \sigma_1 = 12h(x_1+1)+6(h+i), \\ & \sigma_2 = 12h(x_2+1)+6(h+i), \quad (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}, \\ & \sigma_3 = 12h(x_3+1)+6(h+i), \quad (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}. \end{aligned}$$

From Figure 1, when we apply the rule $(s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}$ in neurons σ_2 and σ_3 at time t_j we get

$$\begin{aligned} t_{j+1}: \quad & \sigma_1 = 12h(x_1+3)+6(h+i)+4, \quad (s^{12h})^* s^{6(h+i)+4}/s^{12h+6i+4-6l} \rightarrow s^{6l}, \\ & \sigma_2 = 12h(x_2+1)+6h, \\ & \sigma_3 = 12h(x_3+1)+6h, \\ & \sigma_4 = 12h+2, \qquad s^{12h+2} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{j+2}: \quad & \sigma_1 = 12h(x_1+2)+6(h+l), \\ & \sigma_2 = 12h(x_2+1)+6(h+l), \\ & \sigma_3 = 12h(x_3+1)+6(h+l). \end{aligned}$$

At time t_{j+2} the simulation of $q_i : INC(1)$ is complete. Note that an increment on the value x_1 in counter c_1 was simulated by increasing the $12h(x_1+1)$ spikes in σ_1 to $12h(x_1+2)$ spikes. Note also that the encoding $6(h+l)$ of the next instruction q_l has been established in neurons σ_1, σ_2 and σ_3.

Π_C simulating $q_i : DEC(1)q_k$. There are two cases to consider here. Case 1: if counter c_1 has value $x_1 > 0$, then decrement c_1 and move to instruction q_l. Case 2: if counter c_1 has value $x_1 = 0$, then move to instruction q_k. As with the previous example, our simulation begins at time t_j. Thus Case 1 ($x_1 > 0$) gives

$$\begin{aligned} t_j: \quad & \sigma_1 = 12h(x_1+1)+6(h+i), \\ & \sigma_2 = 12h(x_2+1)+6(h+i), \quad (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}, \\ & \sigma_3 = 12h(x_3+1)+6(h+i), \quad (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \rightarrow s^{12h+2}, \end{aligned}$$

$$\begin{aligned} t_{j+1}: \quad & \sigma_1 = 12h(x_1+3)+6(h+i)+4, \quad (s^{12h})^* s^{54h+6i+4}/s^{36h+6i+4-6l} \rightarrow s^{6l}, \\ & \sigma_2 = 12h(x_2+1)+6h, \\ & \sigma_3 = 12h(x_3+1)+6h, \\ & \sigma_4 = 12h+2, \qquad s^{12h+2} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{j+2}: \quad & \sigma_1 = 12hx_1+6(h+l), \\ & \sigma_2 = 12h(x_2+1)+6(h+l), \\ & \sigma_3 = 12h(x_3+1)+6(h+l). \end{aligned}$$

At time t_{j+2} the simulation of $q_i : DEC(1)q_k$ for Case 1 ($x_1 > 0$) is complete. Note that a decrement on the value x_1 in counter c_1 was simulated by decreasing the $12h(x_1+1)$ spikes in σ_1 to $12hx_1$ spikes. Note also that the encoding $6(h+l)$ of the next instruction q_l has been established in neurons σ_1, σ_2 and σ_3. Alternatively, if we have Case 2 ($x_1 = 0$) then we get

$$\begin{aligned}
t_j : \quad & \sigma_1 = 12h + 6(h+i), & & \\
& \sigma_2 = 12h(x_2+1) + 6(h+i), & & (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \to s^{12h+2}, \\
& \sigma_3 = 12h(x_3+1) + 6(h+i), & & (s^{12h})^* s^{6(h+i)}/s^{12h+6i+2} \to s^{12h+2},
\end{aligned}$$

$$\begin{aligned}
t_{j+1} : \quad & \sigma_1 = 42h + 6i + 4, & & s^{42h+6i+4}/s^{24h+6i+4-6k} \to s^{6k}, \\
& \sigma_2 = 12h(x_2+1) + 6h, & & \\
& \sigma_3 = 12h(x_3+1) + 6h, & & \\
& \sigma_4 = 12h + 2, & & s^{12h+2} \to \lambda,
\end{aligned}$$

$$\begin{aligned}
t_{j+2} : \quad & \sigma_1 = 12h + 6(h+k), \\
& \sigma_2 = 12h(x_2+1) + 6(h+k), \\
& \sigma_3 = 12h(x_3+1) + 6(h+k).
\end{aligned}$$

At time t_{j+2} the simulation of $q_i : DEC(1)q_k$ for Case 2 is complete. The encoding $6(h+k)$ of the next instruction q_k has been established in σ_1, σ_2 and σ_3.

Halting. The halt instruction q_h is encoded as $12h$ spikes. Thus if C halts we get

$$\begin{aligned}
t_j : \quad & \sigma_1 = 12h(x_1+2), & & \\
& \sigma_2 = 12h(x_2+2), & & \\
& \sigma_3 = 12h(x_3+2), & & (s^{12h})^*/s^{12h+1} \to s^{12h},
\end{aligned}$$

$$\begin{aligned}
t_{j+1} : \quad & \sigma_1 = 12h(x_1+3), & & \\
& \sigma_2 = 12h(x_2+3), & & \\
& \sigma_3 = 12h(x_3+1) - 1, & & (s^{12h})^* s^{36h-1}/s^{12h} \to s^{6h}, \\
& \sigma_4 = 12h, & & s^{12h} \to s,
\end{aligned}$$

$$\begin{aligned}
t_{j+2} : \quad & \sigma_1 = 12h(x_1+3) + 6h, & & \\
& \sigma_2 = 12h(x_2+3) + 6h, & & \\
& \sigma_3 = 12hx_3 - 1, & & (s^{12h})^* s^{36h-1}/s^{12h} \to s^{6h}, \\
& \sigma_4 = 6h, & & s^{6h} \to \lambda.
\end{aligned}$$

The rule $(s^{12h})^* s^{36h-1}/s^{12h} \rightarrow s^{6h}$ is applied a further $x_3 - 2$ times in σ_3 to give

$$\begin{aligned} t_{j+x_3}: \quad & \sigma_1 = 12h(x_1+3) + 6h(x_3-1), & \\ & \sigma_2 = 12h(x_2+3) + 6h(x_3-1), & \\ & \sigma_3 = 24h-1, & s^{24h-1} \rightarrow s^{12h}, \\ & \sigma_4 = 6h, & s^{6h} \rightarrow \lambda, \end{aligned}$$

$$\begin{aligned} t_{j+x_3+1}: \quad & \sigma_1 = 12h(x_1+4) + 6h(x_3-1), & \\ & \sigma_2 = 12h(x_2+4) + 6h(x_3-1), & \\ & \sigma_4 = 12h, & s^{12h} \rightarrow s. \end{aligned}$$

As usual the output is the time interval between the first and second timesteps when a firing rule is applied in the output neuron. Note from above that the output neuron σ_4 fires for the first time at timestep t_{j+1} and for the second time at timestep t_{j+x_3+1}. Thus, the output of Π_C is x_3 the value of the output counter c_3 when C enters the halt instruction q_h. Note that if $x_3 = 0$ then the rule $s^{24h-1} \rightarrow s^{12h}$ can not be executed as there is only $12h - 1$ spikes in σ_3 at timestep t_{j+1}. Thus if $x_2 = 0$ the output neuron will fire only once.

We have shown how to simulate arbitrary instructions of the form $q_i : INC(1)$ and $q_i : DEC(1)q_k$ that operate on counter c_1. Instructions which operate on counters c_2 and c_3 are simulated in a similar manner. Immediately following the simulation of an instruction Π_C is configured to simulate the next instruction. Each instruction of C is simulated in 2 timesteps. The pair of input values (x_1, x_2) is read into the system in $x_1 + x_2 + 3$ timesteps and sending the output value x_3 out of the system takes $x_3 + 1$ timesteps. Thus, if C completes it computation in time t, then Π_C simulates the computation of C in linear time $O(t + x_1 + x_2 + x_3)$. □

5 Lower Bounds for Small Universal SN P Systems

In this and other works [8,10] on small SN P systems the input neuron only receives a constant number of spikes from the environment and the output neuron fires no more than a constant number of times. Hence, we call the input standard if the input neuron receives no more than x spikes from the environment, where x is a constant independent of the input (i.e. the number of 1s in its input sequence is $< x$). Similarly, we call the output standard if the output neuron fires no more than y times, where y is a constant independent of the input. Here we say a SN P system has generalised input if the input neuron is permitted to receive $\leqslant n$ spikes from the environment where $n \in \mathbb{N}$ is the length of its input sequence.

Theorem 2. *Let Π be any extended SN P system with only 3 neurons, generalised input and standard output. Then there is a non-deterministic Turing machine T_Π that simulates the computation of Π in space $O(\log n)$ where n is the length of the input to Π.*

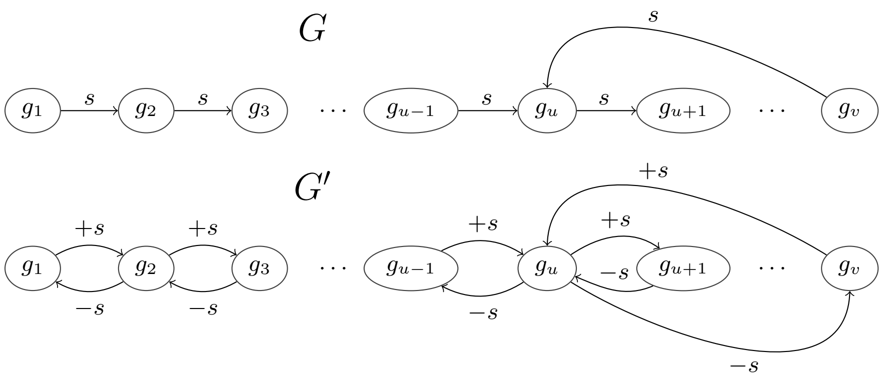


Fig. 2. Finite state machine G decides if there is any rule applicable in a neuron given the number of spikes in the neuron *at a given time* in the computation. Each s represents a spike in the neuron. Machine G' keeps track of the movement of spikes into and out of the neuron and decides whither or not any rule is applicable *at each timestep* in the computation. $+s$ represents a single spike entering the neuron and $-s$ represents a single spike exiting the neuron.

Proof. Let Π be any extended SN P system with generalised input, standard output, and neurons σ_1, σ_2 and σ_3. Also, let y be the maximum number of times the output neuron σ_3 is permitted to fire and let q and r be the maximum value for b and p respectively, for all $E/s^b \rightarrow s^p; d$ in Π.

We begin by explaining how the activity of σ_3 may be simulated using only the states of T_Π (i.e. no workspace is required to simulate σ_3). Recall that the applicability of each rule is determined by a regular expression over a unary alphabet. We can give a single regular expression R that is the union of all the regular expressions for the firing rules of σ_3. This regular expression R determines whither or not there is any applicable rule in σ_3 at each timestep. Figure 2 gives the deterministic finite automata G that accepts $L(R)$ the language generated by R. During a computation we may use G to decide which rules are applicable in σ_3 by passing an s to G each time a spike enters σ_3. However, G may not give the correct result if spikes leave the neuron as it does not record spikes leaving σ_3. Thus, using G we may construct a second machine G' such that G' records the movement of spikes going into and out of the neuron. G' is constructed as follows: G' has all the same states (including accept states) and transitions as G along with an extra set of transitions that record spikes leaving the neuron. This extra set of transitions are given as follows: for each transition on s from a state g_i to a state g_j in G there is a new transition on $-s$ going from state g_j to g_i in G' that records the removal of a spike from σ_3. By recording the dynamic movement of spikes, G' is able to decide which rules are applicable in σ_3 at each timestep during the computation. G' is also given in Figure 2. To simulate the operation of σ_3 we emulate the operation of G' in the states of T_Π. Note that there is a single non-deterministic choice to be made in G'. This choice is at state g_u if a spike is being removed $(-s)$. It would seem that in order to make the correct choice in this situation we need to know the exact number of spikes in σ_3. However, we need only store at most $u+yq$ spikes. The reason for this is that if there are $\geqslant u+yq$ spikes in σ_3, then G' will not enter state g_{u-1} again. To see this, note that σ_3 spikes a maximum of y times using at most q spikes each time, and so once there are $> u+yq$ spikes the number of spikes in σ_3 will be $> u-1$ for the remainder of the computation. Thus, T_Π simulates the activity of σ_3 by simulating the operation of G' and encoding at most $u+yq$ spikes in its states.

In this paragraph we explain the operation of T_Π. Following this, we give an analysis of the space complexity of T_Π. T_Π has 4 tapes including an output tape, which is initially blank, and a read only input tape. The tape head on both the input and output tapes is permitted to only move right. Each of the remaining tapes, tapes 1 and 2 simulate the activity of the neurons σ_1 and σ_2, respectively. These tapes record the number of spikes in σ_1 and σ_2. A timestep of Π is simulated as follows: T_Π scans tapes 1 and 2 to determine if there are any applicable rules in σ_1 and σ_2 at that timestep. The applicability of each neural rule in Π is determined by a regular expression and so a decider for each rule is easily implemented in the states of T_Π. Recall from the previous paragraph that the applicability of the rules in σ_3 is already recorded in the states of T_Π. Also, T_Π is non-deterministic and so if more than one rule is applicable in a neuron T_Π simply chooses the rule to simulate in the same manner as Π. Once T_Π has determined which rules are applicable in each of the three neurons at that timestep it changes the encodings on tapes 1 and 2 to simulate the change in the number of spikes in neurons σ_1 and σ_2 during that timestep. As mentioned in the previous paragraph any change in the number of spikes in σ_3 is recorded in the states of T_Π. The input sequence of Π may be given as binary input to T_Π by placing it on its input tape. Also, if at a given timestep a 1 is read on the input tape then T_Π simulates a spike entering the simulated input neuron. At each simulated timestep, if the output neuron σ_3 spikes then a 1 is place on the output tape, and if σ_3 does not spike a 0 is placed on the output tape. Thus the output of Π is encoded on the output tape when the simulation ends.

In a two neuron system each neuron has at most one out-going synapse and so the number of spikes in the system does not increase over time. Thus, the total number of spikes in neurons σ_1 and σ_2 can only increase when σ_3 fires or a spike is sent into the system from the environment. The input is of length n, and so σ_1 and σ_2 receive a maximum of n spikes from the environment. Neuron σ_3 fires no more than y times sending at most r spikes each time to σ_1 and σ_2. Thus the maximum number of spikes in σ_1 and σ_2 during the computation is $n+2ry$. Using a binary encoding tapes 1 and 2 of T_Π encode the number of spikes in σ_1 and σ_2 using space of $\log_2(n+2ry)$. As mentioned earlier no space is used to simulate σ_3, and thus T_Π simulates Π using space of $O(\log n)$. $\square$

If we remove the restriction that allows the output neuron to fire only a constant number of times then we may construct a universal system with 3 neurons.

Theorem 3. *Let C be a universal counter machine with 3 counters that completes its computation in time t to give the output value x_3 when given the pair of input values (x_1, x_2). Then there is a universal extended SN P system Π'_C with standard input and generalised output that simulates the computation of C in time $O(t + x_1 + x_2 + x_3)$ and has only 3 neurons.*

Proof. A graph of Π'_C is constructed by removing the output neuron σ_4 from the graph in Figure 1 and making σ_3 the new output neuron by adding a synapse to the environment. The rules for Π'_C are given by the first 3 rows of Table 2. The operation of Π'_C is identical to the operation of Π_C from Theorem 1 with the exception of the new output technique. The output of Π'_C is the time interval between the first and second timesteps where exactly $12h$ spikes are sent out of the output neuron σ_3. □

From the last paragraph of the proof of Theorem 2 we get Corollary 1.

Corollary 1. *Let Π be any extended SN P system with only 2 neurons and generalised input and output. Then there is a non-deterministic Turing machine T_Π that simulates the computation of Π in space $O(\log n)$ where n is the length of the input to Π.*

6 Conclusion

Our results show that there is no significant trade-off between the time/space complexity and the number of neurons in universal extended SN P systems; our new systems suffer no exponential slow-down when compared with universal SN P systems with a greater number of neurons (see Table 1). The size of a SN P system could also be measured by the number of neural rules in the system. It would be interesting to explore possible trade-offs between the number of neuron, the number of rules and the time/space complexity of universal SN P systems.

References

1. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae 71(2-3), 279–308 (2006)
2. Korec, I.: Small universal register machines. Theoretical Computer Science 168(2), 267–301 (1996)
3. Neary, T.: Presentation at Computing with Biomolecules (CBM (2008), `http://www.emcc.at/UC2008/Presentations/CBM5.pdf`
4. Neary, T.: On the computational complexity of spiking neural P systems. In: Calude, C.S., Costa, J.F., Freund, R., Oswald, M., Rozenberg, G. (eds.) UC 2008. LNCS, vol. 5204, pp. 189–205. Springer, Heidelberg (2008)
5. Neary, T.: A small universal spiking neural P system. In: Csuhaj-Varjú, E., Freund, R., Oswald, M., Salomma, K. (eds.) International Workshop on Computing with Biomolecules, Vienna, August 2008, pp. 65–74. Austrian Computer Society (2008)
6. Neary, T.: A boundary between universality and non-universality in spiking neural P systems (December 2009), arXiv:0912.0741v1 [cs.CC]

7. Pan, L., Zeng, X.: A note on small universal spiking neural P systems. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A. (eds.) Tenth Workshop on Membrane Computing (WMC10), Curtea de Argeş, Romania, August 2009, pp. 464–475 (2009)
8. Păun, A., Păun, G.: Small universal spiking neural P systems. BioSystems 90(1), 48–60 (2007)
9. Păun, G., Pérez-Jiménez, M.J.: Spiking Neural P Systems. Recent Results, Research Topics. In: Algorithmic Bioprocesses. Natural Computing Series, pp. 273–291. Springer, Heidelberg (2009)
10. Zhang, X., Zeng, X., Pan, L.: Smaller universal spiking neural P systems. Fundamenta Informaticae 87(1), 117–136 (2008)

Using Sums-of-Products for Non-standard Reasoning

Rafael Peñaloza

Theoretical Computer Science
TU Dresden, Germany
penaloza@tcs.inf.tu-dresden.de

Abstract. An important portion of the current research in Description Logics is devoted to the expansion of the reasoning services and the developement of algorithms that can adequatedly perform so-called non-standard reasoning. Applications of non-standard reasoning services cover a wide selection of areas such as access control, agent negotiation, or uncertainty reasoning, to name just a few. In this paper we show that some of these non-standard inferences can be seen as the computation of a sum of products, where "sum" and "product" are the two operators of a bimonoid. We then show how the main ideas of automata-based axiom-pinpointing, combined with weighted model counting, yield a generic method for computing sums-of-products over arbitrary bimonoids.

1 Introduction

Description Logics (DL) [1] is a family of logic-based knowledge representation formalisms, which are employed in various application domains, like natural language processing, configuration, databases, and bio-medical ontologies. One of its most notable successes so far is the adoption of the DL-based language OWL [12] as the standard ontology language for the semantic web. For years, the main interest in the area revolved around the tradeoff between expressivity and the complexity of reasoning. Highly optimized DL reasoning systems have been developed [10,21,22,3,15,13], which can perform standard reasoning (i. e. deciding satisfiability, or subsumption between concepts) within short time bounds, even for realistic applications, where representation requires a very large number of axioms. Although these systems are still being optimized and improved, researchers are slowly turning their attention to the definition and solution of new reasoning problems. Some of these problems, like axiom-pinpointing [7,6], refer to the extraction of more information from an unmodified knowledge base; in the case of axiom pinpointing, the goal is to detect the reason why a consequence follows. Other problems are defined by extending the expressivity of the knowledge base, not by adding new constructors, but rather by giving extended semantics to the axioms. As an example, consider the blend of uncertainty reasoning and DL [9,14], where axioms are apended with a degree of uncertainty.

In this paper we show that some of these new inference problems can be seen as instances of the more general SumProd problem, which consists on computing sums of products of values attached to axioms, based on the sub-ontologies from which a consequence follows. In the following section we introduce some basic notions of DL and general inference relations. We then define the SumProd problem and four of its instances that have been recently studied independently. Finally, we use the ideas of automata-based axiom pinpointing to show a reduction from the SumProd problem to weighted model counting, for which very efficient implementations exist [8]. Due to lack of space, we leave some of the proofs out of this paper.

2 Description Logics and Inference Relations

The common feature of all description logics is the use of *concepts*, that intuitively describe properties of the individuals of the domain, and *roles* that express relations between pairs of individuals. Complex *concept terms* are inductively defined with the help of a set of *constructors*, starting from a set $\mathsf{N_C}$ of *concept names* and a set $\mathsf{N_R}$ of *role names*. What distinguishes one DL from another is the set of constructors used to build concept terms. The most basic constructors are the Boolean ones: *conjunction* $\sqcap$, *disjunction* $\sqcup$, and *negation* $\neg$, and the *existential-* ($\exists$) and *value-restrictions* ($\forall$), whose syntax is shown in the first column of Table 1. The DL that uses only this constructors is called $\mathcal{ALC}$ [19].

We consider two kinds of axioms: *concept definitions* of the form $A \doteq C$, with $A \in \mathsf{N_C}$ and C a concept term, and *general concept inclusions* (GCIs) $C \sqsubseteq D$, where C, D are concept terms. An *acyclic TBox* is a finite set of concept definitions such that every concept name occurs at most once as a left-hand side, and there is no cyclic dependency between the definitions. A *general TBox* is an acyclic TBox extended with a finite set of GCIs. We will refer in general to a *TBox* whenever it is not relevant whether it is an acyclic or a general TBox.

The semantics of $\mathcal{ALC}$ is defined in terms of *interpretations* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where the *domain* $\Delta^\mathcal{I}$ is a non-empty set of individuals, and the interpretation function $\cdot^\mathcal{I}$ maps each concept name $A \in \mathsf{N_C}$ to a subset $A^\mathcal{I}$ of $\Delta^\mathcal{I}$ and each role name $r \in \mathsf{N_R}$ to a binary relation $r^\mathcal{I}$ on $\Delta^\mathcal{I}$. The mapping $\cdot^\mathcal{I}$ can be extended to arbitrary concept terms as shown in the second column of Table 1. An

Table 1. Syntax and semantics of $\mathcal{ALC}$

Syntax	Semantics
$C \sqcap D$	$C^\mathcal{I} \cap D^\mathcal{I}$
$C \sqcup D$	$C^\mathcal{I} \cup D^\mathcal{I}$
$\neg C$	$\Delta^\mathcal{I} \setminus C^\mathcal{I}$
$\exists r.C$	$\{x \in \Delta^\mathcal{I} \mid \exists y \in \Delta^\mathcal{I} : (x,y) \in r^\mathcal{I} \wedge y \in C^\mathcal{I}\}$
$\forall r.C$	$\{x \in \Delta^\mathcal{I} \mid \forall y \in \Delta^\mathcal{I} : (x,y) \in r^\mathcal{I} \Rightarrow y \in C^\mathcal{I}\}$
$A \doteq C$	$A^\mathcal{I} = C^\mathcal{I}$
$C \sqsubseteq D$	$C^\mathcal{I} \subseteq D^\mathcal{I}$

interpretation $\mathcal{I}$ is a *model* of a TBox $\mathcal{T}$ (denoted $\mathcal{I} \models \mathcal{T}$) if, for every axiom in $\mathcal{T}$ the conditions on the semantics column of Table 1 are satisfied.

One of the main decision problems in DL is concept subsumption:[1]

Definition 1. *Let C, D be two concepts and $\mathcal{T}$ a TBox. We say that C is* subsumed *by D w.r.t. $\mathcal{T}$ (denoted as $C \sqsubseteq_{\mathcal{T}} D$) if, for every model $\mathcal{I}$ of $\mathcal{T}$, it holds that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. We say that C is* satisfiable *w.r.t. $\mathcal{T}$ ($\mathcal{T} \models C$) if $C \not\sqsubseteq_{\mathcal{T}} \bot$, where $\bot$ represents any contradictory concept.*

Following [4,5], we will introduce the SumProd problem not for a specific logic and inference problem, but rather in a more general setting. The type of inference problems that we will consider is deciding whether a so-called inference relation holds. To obtain an intuitive understanding of the following definition, just assume that consequences are $\mathcal{ALC}$ concept terms, admissible sets of axioms are $\mathcal{ALC}$ TBoxes, and the inference relation is unsatisfiablility.

Definition 2. *Let $\mathfrak{I}$ and $\mathfrak{T}$ be (possibly infinite) sets of* consequences *and* axioms*, respectively, and let $\mathscr{P}_{admis}(\mathfrak{T}) \subseteq \mathscr{P}_{fin}(\mathfrak{T})$ be a set of finite subsets of $\mathfrak{T}$ such that $\mathcal{T} \in \mathscr{P}_{admis}(\mathfrak{T})$ implies $\mathcal{T}' \in \mathscr{P}_{admis}(\mathfrak{T})$ for all $\mathcal{T}' \subseteq \mathcal{T}$.*

A relation $\vdash$ between $\mathscr{P}_{admis}(\mathfrak{T})$ and $\mathfrak{I}$ is an inference relation *if for every $\mathcal{T} \in \mathscr{P}_{admis}(\mathfrak{T}), \alpha \in \mathfrak{I}, \mathcal{T} \vdash \alpha$ implies $\mathcal{T}' \vdash \alpha$ for all $\mathcal{T}' \in \mathscr{P}_{admis}(\mathfrak{T})$ with $\mathcal{T}' \supseteq \mathcal{T}$.*

The reason why we have introduced the set $\mathscr{P}_{admis}(\mathfrak{T})$ of admissible subsets of $\mathfrak{T}$ (rather than taking all finite subsets of $\mathfrak{T}$) is to allow us to impose additional restrictions on the sets of axioms that must be considered. For instance, acyclic TBoxes are not arbitrary finite sets of concept definitions: in addition, we require that there is no cyclic dependency between axioms, and that every concept name appears at most once as a left-hand side. Clearly, these restrictions satisfy our requirement for admissible sets of axioms. For the rest of this work, we will often call an admissible set of axioms an *ontology*.

The problem of *un*satisfiability of $\mathcal{ALC}$ concepts w.r.t. TBoxes is an inference relation. More formally, let $\mathfrak{I}$ be all $\mathcal{ALC}$ concepts, $\mathfrak{T}$ all GCIs and concept definitions, and $\mathscr{P}_{admis}(\mathfrak{T})$ all TBoxes. The following is an inference relation:

$$\vdash = \{(\mathcal{T}, C) \mid C \text{ is unsatisfiable w.r.t. } \mathcal{T}\}.$$

3 The SumProd Problem

For the SumProd problem we consider that every axiom in an ontology is annotated with a value. These values can be extended to sets of axioms by computing the *product* of the values of axioms in the set. The SumProd problem consists then on computing the *sum* of the values of all subontologies from which a consequence follows. The specific instances of this problem are characterised by the choice of operators for the *sum* and the *product*. To stay as general as possible, we simply assume that there is a *bimonoid* $(M, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, where $\oplus$ is the "sum", with neutral element $\mathbf{0}$, and $\otimes$ is the "product", whose neutral element is $\mathbf{1}$.

[1] For the rest of this paper, we will often refer to *concept terms* simply as *concepts*.

Definition 3. *Let* $(M, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ *be a bimonoid,* $\mathcal{T}$ *an ontology,* α *a consequence with* $\mathcal{T} \vdash \alpha$*, and* $\mathsf{lab}_M : \mathcal{T} \to M$*. The* SumProd problem *is the task of computing*

$$\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_M) := \bigoplus_{\mathcal{S} \subseteq \mathcal{T}, \mathcal{S} \vdash \alpha} \bigotimes_{t \in \mathcal{S}} \mathsf{lab}_M(t).$$

We now present some instances of this problem that have received some attention from research communities in recent years.

3.1 Pinpointing Formula

Suppose that we have a consequence α that follows from an ontology $\mathcal{T}$. The pinpointing formula is a monotone Boolean formula that describes all the subsets of $\mathcal{T}$ from wich a consequence α still follows. More formally, let $\mathsf{lab}_{\mathbb{B}}$ be a mapping that assigns to each axiom t in $\mathcal{T}$ a unique propositional variable. A monotone Boolean formula ϕ is called a *pinpointing formula* for $\mathcal{T}, \alpha$ if for every $\mathcal{S} \subseteq \mathcal{T}$ it holds: $\mathcal{S} \vdash \alpha$ iff $\bigwedge_{t \in \mathcal{S}} \mathsf{lab}_{\mathbb{B}}(t) \models \phi$.[2]

It is easy to see that if we consider as bimonoid the lattice of monotone Boolean formulae over the image of $\mathsf{lab}_{\mathbb{B}}$ (modulo equivalence) $(\mathbb{B}^+, \vee, \wedge, \bot, \top)$, then computing a pinpointing formula is an instance of the SumProd problem; i. e. $\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_{\mathbb{B}})$ is a pinpointing formula for $\mathcal{T}, \alpha$. We will later show that the pinpointing formula is in fact a general solution to the SumProd problem.

3.2 Access Control

In access control we assume that there is a finite lattice $(L, \leq)$ that represents the levels of security in an application. Given an ontology $\mathcal{T}$, each axiom $t \in \mathcal{T}$ is assigned an element $\mathsf{lab}_L(t)$ of L. Basically, $\mathsf{lab}_L(t_1) < \mathsf{lab}_L(t_2)$ means that axiom t_2 is more public than t_1 (which is more private). Additionally, there are some users that are assigned an access level in L; that is, there is a mapping acc from the set of all users to L. The access level of a user u defines a subset of axioms that are visible to this user: $\mathcal{T}_u := \{t \in \mathcal{T} \mid \mathsf{acc}(u) \leq \mathsf{lab}_L(t)\}$.

Let α be a consequence such that $\mathcal{T} \vdash \alpha$. We are interested in finding a so-called boundary. An element $\mu \in L$ is called a *boundary* for $\mathcal{T}, \alpha$ under lab_L if for every user u it follows that $\mathcal{T}_u \vdash \alpha$ iff $\mathsf{acc}(u) \leq \mu$.

It was shown in [2] that $\mathsf{lub}_{\mathcal{S} \subseteq \mathcal{T}, \mathcal{S} \vdash \alpha} \mathsf{glb}_{t \in \mathcal{S}} \mathsf{lab}_L(t)$ is a boundary.[3] Hence, if we consider the lattice L with its lub and glb operators as a bimonoid, we obtain that the computation of a boundary is an instance of the SumProd problem. That is, $\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_L)$ is a boundary for $\mathcal{T}, \alpha$ under lab_L.

3.3 Utility from Preference Formulae

We now leave behind applications where a lattice is used and allow for more general cases of bimonoids. One problem that has started to raise interest is how

[2] A monotone Boolean formula is a propositional formula that contains no negation.

[3] lub and glb denote the least upper bound and the greatest lower bound, respectively.

to compute the utility of a preference set in a negotiation process. We first define the problem of finding the minimal utility value in DL [18,17], and then show that this problem is an instance of the SumProd problem.

Definition 4. *Let $\mathcal{T}$ be a DL ontology. A* preference *is a pair (P, v) where P is a DL concept such that $\mathcal{T} \models P$ and $v \in \mathbb{R}^+$.*

Intuitively, a preference (P, v) tells us how much *value* we assign to the satisfaction of the concept P. If we have a set of preferences $\mathcal{P}$, and are presented with a concept C (called a *proposal*) we would like to be able to know how good this proposal is related to $\mathcal{P}$, in the sense of knowing the total value of the preferences in $\mathcal{P}$ that are compatible with C. Taking the conservative approach, we want to know the *minimal utility value.*

Definition 5. *Let $\mathcal{T}$ be an ontology, C a concept such that $\mathcal{T} \models C$ and $\mathcal{P}$ a set of preferences. The* minimal utility value *for C w.r.t. $\mathcal{P}$ is given by:*

$$\mathsf{MUV}(\mathcal{T}, C, \mathcal{P}) := \min_{\mathcal{I} \models C, \mathcal{I} \models \mathcal{T}} \sum_{(P,v) \in \mathcal{P}, \mathcal{I} \models P} v.$$

Basically, the minimal utility value expresses the least value that we are expected to obtain whenever the proposal C is satisfied. In a negotiation process, we would be confronted with several proposals. We can then compare how worth each of them is w.r.t. our preference set and accept that with the highest MUV.

We now show that the problem of finding the minimal utility value is in fact an instance of SumProd. We consider the bimonoid $(\mathbb{R}^+ \cup \{0, \infty\}, \min, +, \infty, 0)$, which is a semiring, and construct a new ontology $\mathcal{T}' := \mathcal{T} \cup \{\top \sqsubseteq \neg P \mid (P, v) \in \mathcal{P}\}$.[4] The labeling $\mathsf{lab}_{\mathbb{R}} : \mathcal{T}' \to \mathbb{R}^+ \cup \{0, \infty\}$ is defined as follows:

$$\mathsf{lab}_{\mathbb{R}}(t) := \begin{cases} 0 & \text{if } t \in \mathcal{T}, \\ v & \text{if } (P, v) \in \mathcal{P}, t = \top \sqsubseteq \neg P. \end{cases}$$

Finally, given a proposal C, we consider the consequence $\alpha := C \sqsubseteq \bot$.

Theorem 1. *Let $\mathcal{T}$ be an ontology, C a concept such that $\mathcal{T} \models C$ and $\mathcal{P}$ a set of preferences. If $\mathcal{T}, \alpha$ and $\mathsf{lab}_{\mathbb{R}}$ are constructed as above, then, under the bimonoid $(\mathbb{R}^+ \cup \{0, \infty\}, \min, +, \infty, 0)$,*

$$\mathsf{SP}(\mathcal{T}', \alpha, \mathsf{lab}_{\mathbb{R}}) = \mathsf{MUV}(\mathcal{T}, C, \mathcal{P}).$$

3.4 Best Entailment Degree

Another problem that is gaining the interest of the community is the combination of DLs with reasoning under uncertainty and, in particular, with the use of fuzzy operators: *t-norm* $\boxtimes$, *t-conorm* $\boxplus$, *negation* $\boxminus$, and *implication* $\Rrightarrow$. The exact semantics of fuzzy DLs depends on the specific family of fuzzy operators chosen.

[4] $\top$ represents any tautological concept.

The most important of these families are the *Zadeh* [24], the *Łukasiewicz*, the *Product* and the *Gödel* [11] families.

In fuzzy DLs, every axiom in an ontology has an associated *degree of truth*, denoted as a pair $\langle t, n\rangle$, where t is a DL axiom and $n \in [0, 1]$. Intuitively, such a pair denotes that axiom t is true with a degree of at least n. The semantics of fuzzy DLs is defined by means of *fuzzy interpretations*. A fuzzy interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$, called the *domain*, and a *fuzzy interpretation function* that assigns to each concept name $A \in \mathsf{N_C}$ a function $A^{\mathcal{I}} : \Delta^{\mathcal{I}} \to [0, 1]$ and to each role name $r \in \mathsf{N_R}$ a function $r^{\mathcal{I}} : \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \to [0, 1]$. This function is extended to concept terms as follows: $(C_1 \sqcap C_2)^{\mathcal{I}}(x) = C_1(x)^{\mathcal{I}} \boxtimes C_2(x)^{\mathcal{I}}$; $(C_1 \sqcup C_2)^{\mathcal{I}}(x) = C_1(x)^{\mathcal{I}} \boxplus C_2(x)^{\mathcal{I}}$; $(\neg C)^{\mathcal{I}}(x) = \boxminus C^{\mathcal{I}}(x)$; $(\forall r.C)^{\mathcal{I}}(x) = \inf_{y\in\Delta^{\mathcal{I}}} r^{\mathcal{I}}(x, y) \rightrightarrows C^{\mathcal{I}}(y)$; and $(\exists r.C)^{\mathcal{I}}(x) = \sup_{y\in\Delta^{\mathcal{I}}} r^{\mathcal{I}}(x, y) \boxtimes C^{\mathcal{I}}(y)$.

An interpretation $\mathcal{I}$ is a *model* of a fuzzy ontology $\mathcal{T}$ if for every $\langle C \sqsubseteq D, n\rangle \in \mathcal{T}$ it holds that $\inf_{x\in\Delta^{\mathcal{I}}}(C^{\mathcal{I}}(x) \rightrightarrows D^{\mathcal{I}}(x)) \geq n$. A fuzzy GCI $\langle C \sqsubseteq D, n\rangle$ is a *consequence* of a fuzzy ontology $\mathcal{T}$, denoted $\mathcal{T} \vdash \langle C \sqsubseteq D, n\rangle$, if every model of $\mathcal{T}$ is also a model of $\langle C \sqsubseteq D, n\rangle$.[5]

Definition 6. *Let $\mathcal{T}$ be a fuzzy ontology and t a (crisp) GCI. The* best entailment degree *of t w.r.t. $\mathcal{T}$ is*

$$\mathsf{BED}(\mathcal{T}, t) := \sup_{\mathcal{T}\vdash\langle t,n\rangle} n.$$

Briefly, the best entailment degree expresses the best bound that can be given on the fuzzy value at which t follows from the ontology $\mathcal{T}$. This problem is in fact an instance of the SumProd problem over the bimonoid $([0, 1], \max, \boxtimes, 0, 1)$, where $\boxtimes$ is the t-norm being used and the function $\mathsf{lab}_{[0,1]}$ maps every GCI in $\mathcal{T}$ to its associated degree of truth. The following theorem holds for the Łukasiewicz, Product, and Gödel families of fuzzy operators, but not for the Zadeh family.

Theorem 2. *Let $\mathcal{T}$ be an ontology, $\mathsf{lab}_{[0,1]} : \mathcal{T} \to [0, 1]$ the function assigning, to every axiom in $\mathcal{T}$, its associated degree of truth, and t a (crisp) GCI. Then, under the bimonoid* $([0, 1], \max, \boxtimes, 0, 1)$,

$$\mathsf{SP}(\mathcal{T}, t, \mathsf{lab}_{[0,1]}) = \mathsf{BED}(\mathcal{T}, t).$$

4 Solving the SumProd Problem

It was shown in [16] that the pinpointing formula is the most general solution of the SumProd problem over distributive lattices: given an arbitrary distributive lattice M, the (unique) homomorphism from $\mathbb{B}^+$ to M can be used to compute $\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_M)$ from the pinpointing formula for $\mathcal{T}, \alpha$. In fact, the pinpointing formula can be used to solve the SumProd problem over any bimonoid, through *weighted model counting*.

[5] For simplicity, we are restricting ourselves to the case where both, the axioms in the ontology and the consequences, are concept inclusions. For settings dealing with a wider variety of axioms and consequences, see, e.g. [9].

Definition 7. *Let $(M, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a bimonoid, V a set of propositional variables, ψ a Boolean formula over V and* wt *a function that maps every literal corresponding to a variable in V to an element of M.* Weighted model counting *corresponds to the task of computing*

$$\mathsf{WMC}(\psi, \mathsf{wt}) := \bigoplus_{\mathcal{V} \models \psi} \bigotimes_{\ell \in \mathcal{V}} \mathsf{wt}(\ell).$$

Let $\mathcal{T}$ be an ontology and α a consequence such that $\mathcal{T} \vdash \alpha$. If ϕ is a pinpointing formula for $\mathcal{T}, \alpha$, then by definition there is a bijective function $\mathsf{lab}_{\mathbb{B}}$ between $\mathcal{T}$ and a superset of the propositional variables appearing in ϕ. Let now $(M, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ be a bimonoid and $\mathsf{lab}_M : \mathcal{T} \to M$. We construct the function wt as follows: for every positive literal p, we set $\mathsf{wt}(p) = \mathsf{lab}_M(\mathsf{lab}_{\mathbb{B}}^{-1}(p))$, and for every negative literal $\neg p$, we set $\mathsf{wt}(\neg p) = \mathbf{1}$.

Theorem 3. *Let ϕ be a pinpointing formula for $\mathcal{T}, \alpha$ and* wt *built as above. Then* $\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_M) = \mathsf{WMC}(\phi, \mathsf{wt})$.

Proof. Let $\mathcal{V}$ be a valuation and $p_1, \ldots, p_n$ the positive literals appearing in $\mathcal{V}$, and set $\mathcal{S} = \{\mathsf{lab}_{\mathbb{B}}^{-1}(p_i) \mid 1 \leq i \leq n\}$. As ϕ is a pinpointing formula, we have that $\mathcal{S} \vdash \alpha$ iff $\bigwedge_{i=1}^{n} p_i \models \phi$ iff $\mathcal{V} \models \phi$. Additionally, $\bigotimes_{t \in \mathcal{S}} \mathsf{lab}_M(t) = \bigotimes_{i=1}^{n} \mathsf{lab}_M(\mathsf{lab}_{\mathbb{B}}^{-1}(p_i)) = \bigotimes_{i=1}^{n} \mathsf{wt}(p_i) = \bigotimes_{\ell \in \mathcal{V}} \mathsf{wt}(\ell)$. Hence, we have that

$$\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_M) = \bigoplus_{\mathcal{S} \vdash \alpha} \bigotimes_{t \in \mathcal{S}} \mathsf{lab}_M(t) = \bigoplus_{\mathcal{V} \models \phi} \bigotimes_{\ell \in \mathcal{V}} \mathsf{wt}(\ell) = \mathsf{WMC}(\phi, \mathsf{wt}).$$

□

This theorem shows that if one has a pinpointing formula for some ontology $\mathcal{T}$ and consequence α, then one can solve any instance of the SumProd problem related to $\mathcal{T}, \alpha$ through a call to a weighted model counter. It has been shown that the pinpointing formula can be computed by a modified version of the decision algorithm used to verify that $\mathcal{T} \vdash \alpha$. Recently, general approaches that modify tableaux- [4,7] and automata-based [5,6] decision procedures have been developed. However, the formulas obtained by these methods are in general form, with conjunctions and disjunctions nested within each other, while the efficiency of modern weighted model counters relies on having an input formula in CNF.

It is well-known that for every formula ψ it is possible to construct in polynomial time (on the length of ψ) a formula ψ' in CNF such that there is a bijection between the models of ψ and the models of ψ' [23]. The idea consists in introducing new variables that capture complex subformulae of ψ. By setting to $\mathbf{1}$ the weights of all newly added literals, we can ensure that both formulas are also equivalent w.r.t. weighted model counting. Although this does not affect the overall complexity of the method, it introduces an unnecessary step. In fact, it is possible to improve the structure-sharing idea used in [6] to directly obtain a formula that is equivalent (with respect to WMC) to the pinpointing formula.

We first recall the necessary notions for automata-based pinpointing, and then show how these ideas yield a formula in CNF that can be used to solve any instance of SumProd.

4.1 Axiomatic Automata

We consider Büchi automata over trees, whose input alphabet has only one element of arity k. A *Büchi automaton for arity* k is a tuple (Q, Δ, I, F), where Q is a finite set of *states*, $\Delta \subseteq Q^{k+1}$ is the set of transitions, and $I, F \subseteq Q$ are the set of *initial* and *final states*, respectively. A *weighted Büchi automaton* (WBA) over a lattice L is a tuple (Q, in, wt, F), where Q is a finite set of states, $in : Q \to L, wt : Q^{k+1} \to L$, are the *initial* and *transition distribution*, and $F \subseteq Q$.

The reasoning necessary for the computation of the pinpointing formula (and, in general, the SumProd problem) for $\mathcal{T}, \alpha$, needs to know for which subontologies $\mathcal{T}'$ of $\mathcal{T}$, $\mathcal{T}' \vdash \alpha$ holds. Thus, we assume that the automaton $\mathcal{A}_{\mathcal{T},\alpha}$ for deciding $\mathcal{T} \vdash \alpha$ also contains automata for all axiomatized inputs $\mathcal{T}', \alpha$ with $\mathcal{T}' \subseteq \mathcal{T}$,[6] which can be obtained by appropriately restricting the states and transitions of $\mathcal{A}_{\mathcal{T},\alpha}$. To be more precise, let $\mathcal{A} = (Q, \Delta, I, F)$ be a Büchi automaton for arity k, $\mathcal{T}$ an ontology and α a consequence. The functions $\Delta\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q^{k+1})$ and $I\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q)$ are respectively called a *transition restricting function* and an *initial restricting function*. The restricting functions $\Delta\mathsf{res}$ and $I\mathsf{res}$ are extended to sets of axioms $\mathcal{T}' \subseteq \mathcal{T}$ as follows:

$$\Delta\mathsf{res}(\mathcal{T}') := \bigcap_{t \in \mathcal{T}'} \Delta\mathsf{res}(t) \quad \text{and} \quad I\mathsf{res}(\mathcal{T}') := \bigcap_{t \in \mathcal{T}'} I\mathsf{res}(t).$$

For $\mathcal{T}' \subseteq \mathcal{T}$, the $\mathcal{T}'$*-restricted subautomaton of* $\mathcal{A}$ *w.r.t.* $\Delta\mathsf{res}$ *and* $I\mathsf{res}$ is

$$\mathcal{A}_{|\mathcal{T}'} := (Q, \Delta \cap \Delta\mathsf{res}(\mathcal{T}'), I \cap I\mathsf{res}(\mathcal{T}'), F).$$

Definition 8. *Let* $\mathcal{A} = (Q, \Delta, I, F)$ *be a Büchi automaton for arity* k*,* $\mathcal{T}$ *an ontology,* α *a consequence, and* $\Delta\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q^{k+1})$ *and* $I\mathsf{res} : \mathcal{T} \to \mathscr{P}(Q)$ *a transition and an initial restricting function, respectively. We call* $(\mathcal{A}, \Delta\mathsf{res}, I\mathsf{res})$ *an* axiomatic automaton *for* Γ*.*

Given an inference relation $\vdash$*, we say that* $(\mathcal{A}, \Delta\mathsf{res}, I\mathsf{res})$ *is* correct *for* $\mathcal{T}, \alpha$ *w.r.t.* $\vdash$ *if the following holds for every* $\mathcal{T}' \subseteq \mathcal{T}$*:* $\mathcal{T}' \vdash \alpha$ *iff* $\mathcal{A}_{|\mathcal{T}'}$ *does not have a successful run* r *with* $r(\varepsilon) \in I \cap I\mathsf{res}(\mathcal{T}')$*.*

Given a correct axiomatic automaton for $\mathcal{T}, \alpha$, we can decide $\mathcal{T}' \vdash \alpha$ for $\mathcal{T}' \subseteq \mathcal{T}$ through an emptiness test on the automaton $\mathcal{A}_{|\mathcal{T}'}$. Any correct axiomatic automaton can be transformed into a *pinpointing automaton*: a weighted Büchi automaton whose behaviour is a pinpointing formula for the input.

Recall first that in the definition of pinpointing formula we consider a mapping $\mathsf{lab}_{\mathbb{B}}$ assigning a unique propositional variable to each $t \in \mathcal{T}$. The pinpointing automaton takes its weights from the $\mathcal{T}$-Boolean bimonoid $(\mathbb{B}(\mathcal{T}), \wedge, \vee, \top, \bot)$, where $\mathbb{B}(\mathcal{T})$ is the quotient set of all monotone Boolean formulae over $\mathsf{lab}_{\mathbb{B}}(\mathcal{T})$ by the propositional equivalence relation, i.e., two propositionally equivalent formulae correspond to the same element of $\mathbb{B}(\mathcal{T})$. It is easy to see that this bimonoid is in fact a finite distributive lattice, where the partial order is defined as $\phi \leq \psi$ iff $\psi \to \phi$ is valid.[7] Note that this bimonoid is different from the one

[6] Recall that every subset of an admissible set of axioms is also admissible.
[7] More precisely, $\mathbb{B}(\mathcal{T})$ is the free distributive lattice over the generators $\mathsf{lab}_{\mathbb{B}}(\mathcal{T})$.

used in Section 3.1, in that the two operations are exchanged. This is done to follow the construction in [6] and be able to reuse their results.

Definition 9. *Let* $(\mathcal{A}, \Delta\mathsf{res}, I\mathsf{res})$ *be an axiomatic automaton for* $\mathcal{T}, \alpha$*, with* $\mathcal{A} = (Q, \Delta, I, F)$*. The* violating functions $\Delta\mathsf{vio} : Q^{k+1} \to \mathbb{B}(\mathcal{T})$ *and* $I\mathsf{vio} : Q \to \mathbb{B}(\mathcal{T})$ *are*

$$\begin{aligned}\Delta\mathsf{vio}(q_0, q_1, \ldots, q_k) &:= \bigvee_{\{t\in\mathcal{T} \mid (q_0,q_1,\ldots,q_k)\notin\Delta\mathsf{res}(t)\}} \mathsf{lab}(t);\\ I\mathsf{vio}(q) &:= \bigvee_{\{t\in\mathcal{T} \mid q\notin I\mathsf{res}(t)\}} \mathsf{lab}(t).\end{aligned}$$

The pinpointing automaton *induced by* $(\mathcal{A}, \Delta\mathsf{res}, I\mathsf{res})$ *w.r.t.* $\mathcal{T}$ *is the WBA over* $\mathbb{B}^{\mathcal{T}}$ $(\mathcal{A}, \Delta\mathsf{res}, I\mathsf{res})^{\mathsf{pin}} = (Q, in, wt, F)$*, where*

$$in(q) := \begin{cases} I\mathsf{vio}(q) & \text{if } q \in I,\\ \top & \text{otherwise;}\end{cases}$$

$$wt(q_0, q_1, \ldots, q_k) := \begin{cases} \Delta\mathsf{vio}(q_0, q_1, \ldots, q_k) & \text{if } (q_0, q_1, \ldots, q_k) \in \Delta,\\ \top & \text{otherwise.}\end{cases}$$

As shown in [6], the behaviour of the pinpointing automaton yields the pinpointing formula. However, the iterative approach for computing the behaviour of a weighted automaton requires an alternation of the operators $\otimes$ and $\oplus$, and hence, when grounded to the bimonoid $\mathbb{B}(\mathcal{T})$, the formula obtained this way is not in CNF. Furthermore, in order to ensure a polynomially bounded execution time, it was necessary to resort to a compact encoding of the generated formula, using *structure sharing.* Translating this encoding into a CNF formula may result in an exponential blowup.

Fortunately, it is possible to modify the above mentioned iterative approach so that it explicitely exploits the idea of structure sharing by adding new variables during the construction of the formula. The result of this modification is an algorithm that outputs a formula ψ in CNF such that every valuation satisfying the pinpointing formula can be uniquely extended to a valuation satisfying ψ, and conversely, every valuation that satisfies ψ, satisfies also the pinpointing formula. We now show how these changes can be made.[8]

4.2 Computing a CNF Formula

We first briefly recall the iterative method for computing the behaviour of the pinpointing automaton and some of its properties. We later show how it can be used to compute the desired CNF formula.

In the following we assume that we have a pinpointing automaton $\mathcal{A} = (Q, in, wt, F)$. The iterative method defines operators $\mathcal{O}_f, \mathcal{Q} : \mathbb{B}(\mathcal{T})^Q \to \mathbb{B}(\mathcal{T})^Q$,

[8] For the DL $\mathcal{EL}$, our approach reduces to the one in [20].

where $\mathbb{B}(\mathcal{T})^Q$ denotes the set of all mappings from Q to $\mathbb{B}(\mathcal{T})$, and $f \in \mathbb{B}(\mathcal{T})^Q$. The operator $\mathcal{O}_f$ is defined as follows for every $\sigma \in \mathbb{B}(\mathcal{T})^Q$:

$$\mathcal{O}_f(\sigma)(q) = \bigwedge_{(q,q_1,\ldots,q_k)\in Q^{k+1}} \left(wt(q,q_1,\ldots,q_k) \vee \bigvee_{j=1}^{k} \mathsf{step}_f(\sigma)(q_j) \right),$$

where

$$\mathsf{step}_f(\sigma)(q) = \begin{cases} f(q) & \text{if } q \in F \\ \sigma(q) & \text{otherwise.} \end{cases}$$

This operator is monotonic, and hence it makes sense to speak about its least fixpoint (lfp). The operator $\mathcal{Q}$ is based in this lfp: given $\sigma \in \mathbb{B}(\mathcal{T})^Q$,

$$\mathcal{Q}(\sigma) = \mathsf{lfp}(\mathcal{O}_\sigma).$$

The operator $\mathcal{Q}$ is also monotonic, and thus it has a greatest fixpoint (gfp). The following result is a direct consequence of those in [6].

Lemma 1. *Let $\varsigma = \mathsf{gfp}(\mathcal{Q})$. Then $\bigwedge_{q\in Q} in(q) \vee \varsigma(q)$ is a pinpointing formula.*

The results in [6] are in fact stronger, since they also set a bound, depending on the number of states and the number of final states, on the times the operators need to be applied before obtaining the fixpoints.

Lemma 2. *Let $n = |Q|, m = |F|$, and denote as $\widetilde{\top}, \widetilde{\bot}$ the functions that map every state in Q to $\top$ and $\bot$, respectively. The following two results hold:*

$$\mathsf{lfp}(\mathcal{O}_f) = \mathcal{O}_f^{n-m+1}(\widetilde{\top}), \qquad\qquad \mathsf{gfp}(\mathcal{Q}) = \mathcal{Q}^m(\widetilde{\bot}).$$

In order to construct a formula in CNF, we are going to simulate applications of the operators $\mathcal{O}_f$ and $\mathcal{Q}$, introducing new variables that will stand as abbreviations of the formulas constructed at each application. The total number of variables and clauses introduced this way will be polynomially bounded by the size of the automaton, due to Lemma 2.

We introduce the variables $x_{\zeta,q}, y^{\eta}_{\zeta,q}$, and $z^{\eta}_{\zeta,(q,q_1,\ldots,q_k)}$. Intuitively, the variable $x_{\zeta,q}$ is an abbreviation for the formula $\mathcal{Q}^{\zeta}(\widetilde{\bot})(q)$. Likewise, the variable $y^{\eta}_{\zeta,q}$ represents the value of $\mathcal{O}^{\eta}_{\mathcal{Q}^{\zeta}(\widetilde{\bot})}(\widetilde{\top})(q)$. The other variables are used as auxiliary means for keeping the formula in CNF. The formula φ_{CNF} is composed by the following clauses, where $0 \leq \zeta < m, 0 \leq \eta < n - m + 1, q, q_1, \ldots, q_k \in Q$:[9]

$$x_{0,q} \Leftrightarrow \bot, \qquad y^{0}_{\zeta,q} \Leftrightarrow \top,$$

$$x_{\zeta+1,q} \Leftrightarrow y^{n-m+1}_{\zeta,q}, \qquad y^{\eta+1}_{\zeta,q} \Leftrightarrow \bigwedge_{(q,q_1,\ldots,q_k)\in Q^{k+1}} z^{\eta+1}_{\zeta,(q,q_1,\ldots,q_k)},$$

$$z^{\eta+1}_{\zeta,(q,q_1,\ldots,q_k)} \Leftrightarrow wt(q,q_1,\ldots,q_k) \vee \bigvee_{j=1}^{k} \mathsf{choice}^{\eta}_{\zeta}(q_j),$$

[9] For brevity, we use double implications rather than clauses. These implications can easily be transformed in clausal form, thus yielding a CNF formula.

where

$$\mathsf{choice}^{\eta}_{\zeta}(q) = \begin{cases} x_{\zeta,q} & \text{if } q \in F \\ y^{\eta}_{\zeta,q} & \text{otherwise.} \end{cases}$$

Finally, we add for every $q \in Q$ the clause

$$in(q) \vee x_{m,q}.$$

Notice that the new variables are effectively nothing more than abbreviations for longer formulas. The truth value of each of them depends ultimately only on the truth value of the original propositional variables used for defining the function *wt*. The last clauses introduced simply use the definition of pinpointing formula from Lemma [1]. The following result is a direct consequence of Lemmas [1] and [2].

Theorem 4. *Let ϕ be a pinpointing formula and φ_{CNF} the formula in CNF constructed above. Then, every valuation $\mathcal{V}$ satisfying ϕ can be uniquely extended to a valuation $\mathcal{V}'$ satisfying φ_{CNF}. Conversely, every valuation that satisfies φ_{CNF} satisfies also ϕ.*

Corollary 1. *Let φ_{CNF} be constructed as above, and wt built as for Theorem [3], and extended to the new literals by setting $\mathsf{wt}(\ell) = 1$ for all new literal ℓ. Then $\mathsf{SP}(\mathcal{T}, \alpha, \mathsf{lab}_M) = \mathsf{WMC}(\varphi_{\mathsf{CNF}}, \mathsf{wt})$.*

5 Conclusions

We have shown that some of the recently studied non-standard inference problems can be seen as instances of the general SumProd problem. We have also shown that the ideas of automata-based axiom pinpointing can be adapted to reduce the SumProd problem to a weighted model counting problem (with the input formula in CNF).

As future work we would like to find more non-standard inferences that fall into the framework described in this paper, for distinct inference relations, also beyond the realm of DL. Additionally, we would like to empirically test our approach by introducing the formula φ_{CNF} into a state-of-the-art weighted model counter. We want then to compare the execution time to other *ad-hoc* implementations, such as the black-box method for computing the boundary in access control [2] or the algorithm for MUV from [18].

References

1. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
2. Baader, F., Knechtel, M., Peñaloza, R.: A generic approach for large-scale ontological reasoning in the presence of access restrictions to the ontology's axioms. In: Bernstein, A., Karger, D.R., Heath, T., Feigenbaum, L., Maynard, D., Motta, E., Thirunarayan, K. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 49–64. Springer, Heidelberg (2009)
3. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL — A polynomial-time reasoner for life science ontologies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 287–291. Springer, Heidelberg (2006)

4. Baader, F., Peñaloza, R.: Axiom pinpointing in general tableaux. In: Olivetti, N. (ed.) TABLEAUX 2007. LNCS (LNAI), vol. 4548, pp. 11–27. Springer, Heidelberg (2007)
5. Baader, F., Peñaloza, R.: Automata-based axiom pinpointing. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 226–241. Springer, Heidelberg (2008)
6. Baader, F., Peñaloza, R.: Automata-based axiom pinpointing. Journal of Automated Reasoning (2010); Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 226–241. Springer, Heidelberg (2008)
7. Baader, F., Peñaloza, R.: Axiom pinpointing in general tableaux. Journal of Logic and Computation 20(1), 5–34 (2010); Special Issue: Tableaux '07 (2007)
8. Bacchus, F., Dalmao, S., Pitassi, T.: Solving #SAT and Bayesian inference with backtracking search. J. of Art. Intel. Research 34, 391–442 (2009)
9. Bobillo, F., Straccia, U.: Fuzzy description logics with general t-norms and datatypes. Fuzzy Sets and Systems 160(23), 3382–3402 (2009)
10. Haarslev, V., Möller, R.: RACER system description. In: Goré, R.P., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS (LNAI), vol. 2083, p. 701. Springer, Heidelberg (2001)
11. Hájek, P.: Metamathematics of Fuzzy Logic. Kluwer, Dordrecht (2001)
12. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: The making of a web ontology language. J. of Web Sem. 1(1), 7–26 (2003)
13. Kazakov, Y.: Consequence-driven reasoning for Horn SHIQ ontologies. In: Boutilier, C. (ed.) Proc. of IJCAI 2009, Pasadena, California, pp. 2040–2045 (2009)
14. Lukasiewicz, T.: Expressive probabilistic description logics. Artif. Intel. 172(6-7), 852–883 (2008)
15. Motik, B., Shearer, R., Horrocks, I.: Optimized reasoning in description logics using hypertableaux. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 67–83. Springer, Heidelberg (2007)
16. Peñaloza, R.: Reasoning with weighted ontologies. In: Grau, B.C., Horrocks, I., Motik, B., Sattler, U. (eds.) Proc. of DL '09. CEUR-WS, vol. 477 (2009)
17. Ragone, A., Noia, T.D., Donini, F.M., Sciascio, E.D., Wellman, M.P.: Computing utility from weighted description logic preference formulas. In: Baldoni, M., van Riemsdijk, M.B. (eds.) DALT 2009. LNCS, vol. 5948, pp. 158–173. Springer, Heidelberg (2010)
18. Ragone, A., Noia, T.D., Donini, F.M., Sciascio, E.D., Wellman, M.P.: Weighted description logics preference formulas for multiattribute negotiation. In: Godo, L., Pugliese, A. (eds.) SUM 2009. LNCS, vol. 5785, pp. 193–205. Springer, Heidelberg (2009)
19. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intel. 48(1), 1–26 (1991)
20. Sebastiani, R., Vescovi, M.: Axiom pinpointing in lightweight description logics via Horn-SAT encoding and conflict analysis. In: Schmidt, R.A. (ed.) Automated Deduction – CADE-22. LNCS, vol. 5663, pp. 84–99. Springer, Heidelberg (2009)
21. Sirin, E., Parsia, B.: Pellet: An OWL DL reasoner. In: Proc. of DL '04, pp. 212–213 (2004)
22. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: System description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
23. Tseitin, G.S.: On the complexity of derivations in the propositional calculus. In: Studies in Mathematics and Mathematical Logic, Part II (1968)
24. Zadeh, L.A.: Fuzzy sets. Information and Control 8(3), 338–353 (1965)

Restarting Automata with Structured Output and Functional Generative Description

Martin Plátek, František Mráz, and Markéta Lopatková

Charles University in Prague, Czech Republic
martin.platek@mff.cuni.cz, frantisek.mraz@mff.cuni.cz,
lopatkova@ufal.mff.cuni.cz

Abstract. Restarting automata were introduced for modeling linguistically motivated analysis by reduction. In this paper we enhance these automata with a structured output in the form of a tree. Enhanced restarting automata can serve as a formal framework for the Functional Generative Description. In this framework, a natural language is described at four layers. Working simultaneously with all these layers, tectogrammatical dependency structures that represent the meaning of the sentence are derived.

1 Introduction

Functional Generative Description (FGD) is a stratificational dependency based descriptive system for the Czech language, which has been in development since 1960s, see esp. [9].

Stratification approaches split language description into layers, each layer providing complete description of sentence and having its own vocabulary and syntax. Here we focus on two layers of FGD, the layer of wordforms (w-layer) and the most abstract layer of meaning (called tectogrammatical layer in FGD, t-layer). At the w-layer, sentence is represented as a simple string of words and punctuation. The t-layer comprises language meaning; the core concepts of this layer are dependency, valency, and topic-focus articulation, see esp. [9].

FGD as a *dependency based approach* describes meaning of a sentence as (tectogrammatical) dependency tree – (meaningful) words are represented as nodes of the tree (each node being a complex unit capturing the lexical meaning, important morphological and syntactic characteristics); relations among words are represented by oriented edges between nodes of the tree [2].

We attempt to provide a formal model for natural language processing based on an elementary method of analysis by reduction. The *analysis by reduction* (AR henceforth, see [4,5], here Section 1.1), serves for describing syntactic structures of natural languages (and particularly languages with free word order). The framework of restarting automata meets the basic requirements set for natural language description stated in [5], i.e., (i) distinguishing the set of well-formed sentences (henceforth L_w) and the set of meaning structures (TR for tectogrammatical representation) and (ii) setting TSH relation comprising mutual relation between these two sets, relation of synonymy and ambiguity (homonymy).

In [5], FGD is modeled as a formal string to string translation using a suitable model of restarting automata. Here we enrich this class of restarting automata with structured output; that is, we define a new type of restarting transducers – *enhanced simple restarting automata* that output dependency trees derived from AR (Section 2). However, such a model is still too general. We formulate several additional restrictions that should be put on an enhanced simple restarting automaton $\mathsf{M}_{\mathsf{FGD}}$ (the last subsection in Section 3). As a result, we get a formal automaton, which helps linguists in further developing the theoretical model of FGD for the Czech language while it allows us to study its formal properties from the mathematical point of view. Section 3.1 summarizes the current model and proposes directions for further research.

This work does not focus on mathematical statements about restarting automata and dependency grammars. Restarting automata have been intensively studied, see e.g. [7,8]. Articles studying mathematical properties of enhanced restarting automata are to be expected in a near future. They will combine the methodologies of restarting automata and dependency grammars (see e.g. [1,2]).

1.1 Basic Principles of Analysis by Reduction

Analysis by reduction makes it possible to define formal dependency relations between particular sentence members. Roughly speaking, (i) a certain word depends on (modifies) another word from the sentence if it is possible to reduce this modifying word (and obtain a simplified correct sentence), and (ii) two words can be removed in an arbitrary order if and only if they are mutually independent. So whereas the basic operation in constituent-based approaches is the decomposition of a sentence into continuous parts representing simplified structures (phrases), in the (dependency) analysis by reduction it is possible to determine dependency relations between individual lexical words leaving aside their word order.

AR is based on a stepwise simplification of an analyzed sentence (enriched with metalanguage information from all layers of FGD); in each step, at least one word / symbol of the input string is deleted, which may lead to rewriting some other symbol. The sentence is simplified until so called *core predicative structure* is reached (typically formed by a sentence predicate and its valency complementations).

When simplifying input sentence, it is necessary to apply certain elementary principles assuring adequate analysis, primarily the principle of correctness preserving and the principle of completeness with respect to valency structure [4]. Moreover, each step of AR must be minimal from a certain point of view – any potential reduction step concerning less symbols in the sentence would violate the principle of completeness (it would lead to an incomplete sentence); it implies that only items fulfilling valency slots of a single governing word are processed in a single reduction step.

The basic principles of AR are exemplified on several reduction steps for particular Czech sentence (Example 1); they illustrate how the sentence is simplified and how fragments of its dependency tree are built (see [5] for more details).

Eng.	*w*-layer	*m*-layer	*a*-layer	*t*-layer
Our	*Našeho*	*můj*.PSMS4	Atr	
Karel	*Karla*	*Karel*.NNMS4	Obj	
				[*my*].t_ACT
(we) plan	*plánujeme*	*plánovat*.VB-P-	Pred	*plánovat*.f_PRED.VF1
				Karel.c_PAT
				my.f_APP
				[*my*].t_ACT
to send	*poslat*	*poslat*.Vf- - -	Obj	*poslat*.f_PAT.VF2
for	*na*	*na*.RR- - 4	AuxP	
next	*příští*	*příští*.AA4IS	Atr	
year	*rok*	*rok*.NNIS4	Adv	*rok*.f_THL
				příští.f_RSTR
to	*do*	*do*.RR- - 2	AuxP	
England	*Anglie*	*Anglie*.NNFS2	Adv	*Anglie*.f_DIR3
.	.	..Z: - - -	AuxK	

Fig. 1. Representation of the sample sentence from Example 1 at four layers of FGD (simplified)

Example 1
Našeho Karla plánujeme poslat na příští rok do Anglie. ([9], p. 241, modified)
our - Karel - (we) plan - to sent - for - next - year - to - England
Eng. 'It's our Karel whom we plan to send for the next year to England.'

Figure 1 presents a simplified representation of the sample sentence at four layers of FGD. Each column captures one layer of language description (*w*-, *m*-, *a*- and *t*-layer, respectively, see Section 1). Rows capture information related to particular words and punctuation marks (one or more rows for an individual word / punctuation, depending on its surface and deep word order, see [5]).

In the first reduction steps of AR, either word *našeho* 'our' or *příští* 'next' may be reduced (and the whole rows representing these words) in an arbitrary order – both simplified sentences are grammatically correct and they are complete. It implies that these two words are mutually independent and each of them modifies some word in the respective simplified sentence (see [4] for details).

Based on surface syntactic and morphological categories, pronoun *našeho* 'our' and adjective *příští* 'next' are analyzed as depending on proper name *Karla* 'Karel' and noun *rok* 'year', respectively. These reductions are reflected at *t*-layer as edges between respective nodes, namely edge '*my*.f_APP $\longrightarrow$ *Karel*.c_PAT' and edge '*příští*.f_RSTR $\longrightarrow$ *rok*.f_THL' (Fig. 2). When reducing simple dependents as *našeho* 'our' and *příští* 'next', only delete operation is required. The same is true if a word w can be reduced together with all its valency complementations in one step without a loss of completeness. On the other hand, if word w fills valency requirements of some other word (and thus cannot be simply reduced without a loss of completeness), rewriting operation is used for indicating the completeness of the original sentence (the reduced word w is rewritten by its syntactic category; such a structure is interpreted as a complete structure).

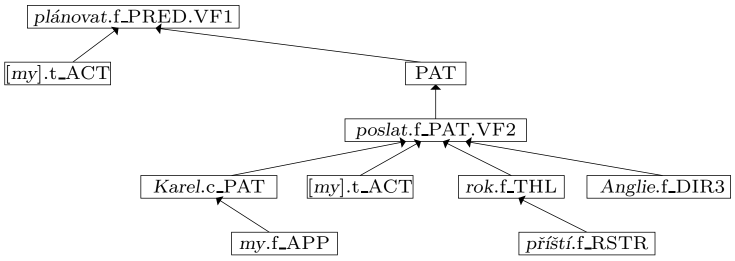


Fig. 2. Tectogrammatical tree for example sentence (1)

We can continue in a similar way and build tectogrammatical tree on the basis of individual reduction steps; the final tectogrammatical tree is in Figure 2.

2 Restarting Automata as a Formal Framework for FGD

Simple restarting automaton – *t*-sRL-automaton. An sRL-*automaton* M is a (in general) nondeterministic machine with a finite-state control Q, a finite working alphabet Γ, and a head (window of size 1) that works on a flexible tape delimited by the left sentinel ¢ and the right sentinel \$ ($¢, \$ \notin \Gamma$). For an input $w \in \Gamma^*$, the initial tape inscription is $¢w\$$. To process this input, M starts in its initial state q_0 with its window over the left end of the tape, scanning the left sentinel ¢. According to its transition relation, M performs *move-right* and *move-left steps*, which shift the window one position to the right or left, respectively, thereby possibly changing the state of M, *rewriting steps*, which rewrite the content of the window without a further move and change the state, and *delete steps*, which delete the content of the window (thus shorten the tape), change the state, and shift the window to the right neighbor of the deleted symbol. Of course, neither the left sentinel ¢ nor the right sentinel \$ must be deleted. At the right end of the tape M either halts and accepts, or it halts and rejects, or it *restarts*, that is, it places its window over the left end of the tape and reenters the initial state. It is required that before the first restart step and also between any two restart steps, M executes at least one delete operation.

A *configuration* of M is a string $\alpha q \beta$ where $q \in Q$, and either $\alpha = \lambda$ and $\beta \in \{¢\} \cdot \Gamma^* \cdot \{\$\}$ or $\alpha \in \{¢\} \cdot \Gamma^*$ and $\beta \in \Gamma^* \cdot \{\$\}$; here q represents the current state, $\alpha\beta$ is the current content of the tape, and it is understood that the window contains the first symbol of β. A *restarting configuration* is of the form $q_0¢w\$$, where $w \in \Gamma^*$.

A *cycle* starts in a restarting configuration, the window is moved along the tape by performing some operations until a restart operation is performed. If after a restart no further restart operation is performed, each finite computation necessarily finishes in a halting configuration – such a subcomputation is called a *tail*. We assume that no delete and rewrite operation is executed in a tail computation.

We use the notation $u \vdash^{c}_{M} v$ to denote a cycle of M that begins with the restarting configuration $q_0¢u\$$ and ends with the restarting configuration $q_0¢v\$$; the relation $\vdash^{c^*}_{M}$ is the reflexive and transitive closure of $\vdash^{c}_{M}$.

An input $w \in \Gamma^*$ is *accepted* by M, if there is an accepting computation which starts with the restarting configuration $q_0¢w\$$. By $L_C(M)$ we denote the language consisting of all words accepted by M; we say that M *accepts the characteristic language* $L_C(M)$. By $S(M)$ we denote the *simple language of* M, which consists of all words that M accepts by tail computations. Observe that, for each $w \in \Gamma^*$, we have $w \in L_C(M)$ if and only if $w \vdash^{c^*}_{M} v$ holds for some word $v \in S(M)$.

A t-sRL-*automaton* ($t \geq 1$) is an sRL-automaton which uses at most t delete operations in a cycle, and for each $v \in S(M)$ it holds $|v| \leq t$.

Remark 1. Let us note, that t-sRL-automata are two-way nondeterministic automata which can check the whole sentence prior to any changes in a cycle. It resembles a linguist, who can read the whole sentence first and reduce the sentence in a correct way afterward. We choose nondeterministic model in order to obtain various sequences of possible reductions.

Similarly as in [7], we can describe a t-sRL-automaton by (meta-)instructions of the following two forms. A *restarting instruction* over Γ is defined as:

$$I_{\mathrm{R}} = (¢ \cdot E_0, a_1 \to b_1, E_1, a_2 \to b_2, E_2, \ldots, E_{s-1}, a_s \to b_s, E_s \cdot \$),$$

where $s \in \{1, \ldots t\}$, $b_1, \ldots, b_s \in \{\lambda\} \cup \Gamma$, λ denotes the empty string, $E_0, E_1, \ldots,$ E_s are regular languages over Γ (often represented by regular expressions), and $a_1, a_2, \ldots, a_s \in \Gamma$ correspond to symbols that are processed during the corresponding cycle of M; either all of them are deleted or one of them is rewritten and the rest is deleted. When trying to execute I_{R} starting from a configuration $q_0¢w\$$, M gets stuck (and so reject), if w does not admit a factorization of the form $w = v_0a_1v_1a_2 \ldots v_{s-1}a_sv_s$ such that $v_i \in E_i$ for all $i = 0, \ldots, s$. On the other hand, if w admits factorizations of this form, then one of them is chosen nondeterministically, and $q_0¢w\$$ is transformed onto $q_0¢v_0b_1v_1b_2 \ldots v_{s-1}b_sv_s\$$.

Tails of accepting computations are described by *accepting instructions* over Γ of the form:

$$I_{\mathrm{A}} = (¢ \cdot E \cdot \$, \mathsf{Accept}),$$

where E is a finite language over Γ. In a configuration $¢z\$$, a tail computation controlled by I_{A} finishes by accepting if $z \in E$, otherwise the computation halts with rejection.

Further we refer to a t-sRL-automaton as to a tuple $M = (\Gamma, ¢, \$, R(M),$ $A(M))$, where Γ is a characteristic vocabulary (alphabet), $¢$ and $\$$ are sentinels not belonging to Γ, $R(M)$ is a finite set of restarting instructions over Γ and $A(M)$ is a finite set of accepting instructions over Γ.

The following property of restarting automata has a crucial role for modeling the analysis by reduction.

(Correctness Preserving Property). A t-sRL-automaton M is *correctness preserving* if $u \in L_C(M)$ and $u \vdash_M^{c^*} v$ imply that $v \in L_C(M)$.

It is known that all deterministic sRL-automata are correctness preserving. On the other hand, it is easy to design a nondeterministic sRL-automaton which is not correctness preserving (see [7]).

Enhanced *t*-sRL-automata. Enhanced t-sRL-automaton M_o is an extension of a t-sRL-automaton; it is described by instructions enhanced with trees. These trees are used to assign a tree structure to items (cells of the tape containing symbols) and consequently to assign an output to each accepting computation. Enhanced instruction can attach a tree to any item containing symbol which is not deleted during the corresponding cycle.

We will use tree structures denoted as DR-trees (Delete-Rewrite trees); DR-tree is a rooted ordered tree with edges oriented from its leaves to its root. Nodes of the trees are (some of) items deleted or rewritten during individual cycles. Each item has its horizontal position, which preserves left-to-right ordering of the input word. Vertical position of a node corresponds to the number of rewritings on the item(s) with the same horizontal position. The root of such a tree is one of the nodes of the tree which remain on the tape after an accepting tail. The edges of such trees are of the following two types:

– an *oblique* edge connects a marked item to some other marked item;
– a *vertical edge* connects a rewritten item to the original item. The horizontal position of the rewritten item is the same as the horizontal position of the original item. The vertical position of the rewritten item is by one higher than that of the original item. We can consider rewriting as creating a new item (cell) containing a new symbol and placing it in the same position of the tape as the original item.

Formally, a DR-tree $T = (V, H)$ is a rooted tree consisting of a finite set of nodes V and a finite set of oriented edges $H \subseteq V \times V$.

1. *A node* $u \in V$ is a triple $u = [i, j, a]$, where i, j are integers and a is a symbol from an alphabet Γ. Index i represents horizontal position of u. Index j represents vertical position of u and equals to the number of rewritings performed till a appeared in the corresponding cell of the tape (it is 0 when the cell was not rewritten).
2. Each *edge* $h = ([i_u, j_u, a], [i_v, j_v, b]) \in H$ is either oblique or vertical. We say that h is:
   – an *oblique edge*: if $i_u \neq i_v$;
   – a *vertical (rewriting) edge*: if $i_u = i_v$ and $j_v = j_u + 1$.

DR-trees are used in enhanced instructions for storing structural information contained in the deleted or rewritten parts of an input sentence. This information is combined with the DR-structures from accepting instructions into resulting trees.

Enhanced restarting instructions. Two kinds of restarting instructions are distinguished: instructions that rewrite one symbol and delete some other symbols, and instructions that delete symbols only.

A. An enhanced restarting instruction which rewrites one symbol is of the form:

$$I_{\mathrm{R}} = (\text{¢} \cdot E_0, [a_1 \to \lambda]_1, E_1, [a_2 \to \lambda]_2, E_2, \ldots, E_{r-1}, [a_r \to b]_r,$$
$$E_r, [a_{r+1} \to \lambda]_{r+1}, \ldots E_{s-1}, [a_s \to \lambda]_s, E_s \cdot \$, T_{\mathrm{R}}),$$

where $I = (\text{¢} \cdot E_0, a_1 \to \lambda, E_1, a_2 \to \lambda, \ldots, E_{r-1}, a_r \to b, E_r, a_{r+1} \to \lambda, \ldots, a_s \to \lambda, E_s \cdot \$)$ is a restarting instruction of an t-sRL-automaton. This instruction rewrites exactly one symbol a_r onto b and deletes symbols a_j, for $j = 1, \ldots, r-1, r+1, \ldots, s$. $T_{\mathrm{R}} = (V, H)$ is a DR-tree with $V \subseteq \{[i, 0, a_i] \mid 1 \leq i \leq s\} \cup \{[r, 1, b]\}$, root $[r, 1, b]$, oblique edges between nodes $[1, 0, a_1], \ldots, [s, 0, a_s]$ and one rewriting edge $([r, 0, a_r], [r, 1, b])$.

B. An enhanced restarting instruction which deletes only is of the form:

$$I_{\mathrm{D}} = (\text{¢} \cdot E_0, [a_1 \to \lambda]_1, E_1, [a_2 \to \lambda]_2, E_2 \ldots, a_{r-1}, E_{r-1}, [a_r]_r,$$
$$E_r, [a_{r+1} \to \lambda]_{r+1}, \ldots, [a_{s+1} \to \lambda]_{s+1}, E_s \cdot \$, T_{\mathrm{D}}),$$

where $I = (\text{¢} \cdot E_0, a_1 \to \lambda, E_1, a_2 \to \lambda, E_2, \ldots, E_{r-1} \cdot a_r \cdot E_r, a_{r+1} \to \lambda, \ldots, a_s \to \lambda, E_s \cdot \$)$ is a restarting instruction of an t-sRL-automaton. This instruction deletes symbols a_j, for $j = 1, \ldots, r-1, r+1, \ldots, s$. The symbol a_r is not deleted and actually denotes the item which will become the root of the DR-tree T_{D}. Thus $T_{\mathrm{D}} = (V, H)$, where $V \subseteq \{[i, 0, a_i] \mid 1 \leq i \leq s\}$, H contains oblique edges only and the root of T_{D} is $[r, 0, a_r]$.

Example 2. Let us consider the language $L_{abc} = \{a^n b^n c^n \mid n > 0\}$. During one reduction of a word (sentence) $a^n b^n c^n$, for some $n \geq 2$, we can delete the last symbol a, the symbol b preceding the last b will be considered for the root of DR-tree T_{D}, the last symbol b as well as the first symbol c will be deleted. The DR-tree T_{D} represents dependencies between deleted symbols of the word; it contributes to an incrementally built resulting tree. Hence the automaton has a single restarting instruction $I_{\mathrm{D}} = (\text{¢} \cdot a^*, [a \to \lambda]_1, b^*, [b]_2, \lambda, [b \to \lambda]_3, \lambda, [c \to \lambda]_4, c^* \cdot \$, T_{\mathrm{D}})$, where $T_{\mathrm{D}} = (\{[2,0,b], [3,0,b], [4,0,c]\}, \{([4,0,c], [3,0,b]), ([3,0,b], [2,0,b])\})$ is depicted in Fig. 3 on the left. The DR-tree T_{D} indicates that the deleted symbol b depends on the non-deleted symbol b, that the deleted c depends on the deleted b, and further that the also deleted symbol a is not included into the dependency structure at all. Several trees different from T_{D} could be used, too. E.g., $T_1' = (\{[2,0,b], [3,0,b], [4,0,c]\}, \{([3,0,b], [2,0,b]), ([4,0,c], [2,0,b])\})$ (Fig. 3 middle); this type of a DR-tree we will be later ruled-out.

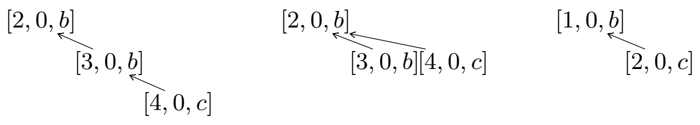


Fig. 3. DR-trees T_{D} (left), T_1' (middle) and T_{A} (right)

Enhanced accepting instructions. Enhanced accepting instructions cannot delete symbols, they can mark some symbols (items) of the tape and combine them into a resulting tree. They are of the form:

$$I_{\mathrm{A}} = (¢ \cdot E_0, [a_1]_1, E_1, [a_2]_2, E_2 \ldots, [a_s]_s, E_s \cdot \$, T_{\mathrm{A}}, \mathsf{Accept}),$$

where $I = (¢ \cdot E_0 \cdot a_1 \cdot E_1 \cdot a_2 \cdot E_2 \ldots a_s \cdot E_s \cdot \$, \mathsf{Accept})$ is an accepting instruction of an t-sRL-automaton. An t-sRL-automaton cannot rewrite during a tail computation, hence the tree $T_{\mathrm{A}} = (V, H)$ is a rooted DR-tree without rewriting edges. $V = \{[i, 0, a_i] \mid 1 \leq i \leq s\}$, H contains oblique edges only and the root of T_{A} is an arbitrary node from V.

Example 3. For the language $L_{abc} = \{a^n b^n c^n | \ n > 0\}$ from Example 2 we will need an accepting instruction, too. We can use the following

$$I_{\mathrm{A}} = (¢ \cdot a, [b]_1, \lambda, [c]_2, \$, \ T_{\mathrm{A}}, \mathsf{Accept}),$$

where $T_{\mathrm{A}} = (\{[1, 0, b], [2, 0, c]\}, \{([2, 0, c], [1, 0, b])\})$ (Fig. 3 right). Using this instruction, the item containing the last symbol b will become the root of the created tree and the last symbol c will become his descendant.

Computations of enhanced t-sRL-automata – combining DR-trees. Informally, each enhanced restarting instruction attaches a DR-tree to some item of the tape containing a non-deleted symbol. When an item with attached tree is used as a node of another tree in a subsequent reduction, it preserves the former descendant nodes. In this way bigger and bigger trees can be build.

An accepting computation of an enhanced t-sRL-automaton starts in an initial configuration with the input sentence (word) written on its tape. Automaton M performs several cycles according to its restarting instructions and the computation finishes by an accepting tail. Of course, in each cycle the items of the tape can contain several non-connected trees. The final accepting instruction produces the resulting DR-tree. The set of all DR-trees which can be obtained as a result of some accepting computations of an enhanced t-sRL-automaton M are called the TR-language of M and we denote it as $\mathsf{TR}(M)$.

Remark 2. Note that when we apply an enhanced instruction, the first elements of the nodes (which are small integers in DR-trees in the instructions of M) of resulting trees are always replaced by original positions of the corresponding items in the input word.

In step k of a computation of an enhanced t-sRL-automaton, we can record the number of rewritings applied on each non-deleted symbol from the beginning of the computation. To each symbol a_i forming the word $w = a_1 a_2 ... a_n$ on the tape after performing step k, we can assign a triple (h_i, v_i, a_i) containing its position h_i in the original (input) word and the number v_i of rewritings applied to the original symbol located at the position h_i. We denote the sequence $t_k = [h_1, v_1, a_1][h_2, v_2, a_2] \ldots [h_n, v_n, a_n]$ as an *extended tape content at step k*. Obviously, $1 \leq h_1 < h_2 < \ldots < h_n$ and $v_i \geq 0$, for all $i = 1, \ldots, n$. For each extended tape content $t' = [h'_1, v'_1, a'_1][h'_2, v'_2, a'_2] \ldots [h'_m, v'_m, a'_m]$, we have

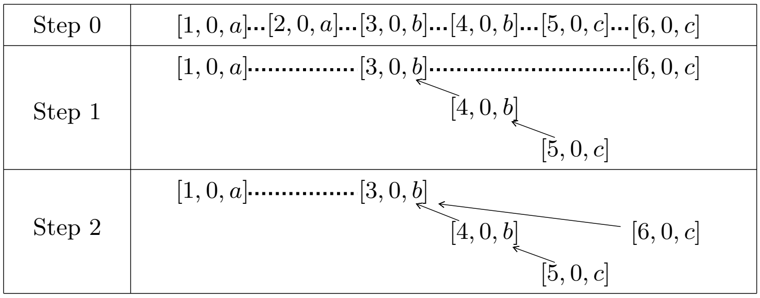

Step 0
[1, 0, a] [2, 0, a] [3, 0, b] [4, 0, b] [5, 0, c] [6, 0, c]
Step 1
[1, 0, a] [3, 0, b] [6, 0, c]
[4, 0, b]
[5, 0, c]
Step 2
[1, 0, a] [3, 0, b]
[4, 0, b] [6, 0, c]
[5, 0, c]

Fig. 4. A sample computation

a unique string of symbols $a'_1 \dots a'_m$. In particular, if w is an input word, i.e. prior to any reduction, we define the initial extended tape content as $\mathsf{Sp}(w) = [1, 0, a_1][2, 0, a_2] \dots [n, 0, a_n]$.

Example 4. Let $M = (\{a, b, c\}, ¢, \$, R(M), A(M))$ be an enhanced sRL-automaton with $R(M) = \{I_\mathrm{D}\}$, where I_D is the restarting instruction from Example 2, $A(M) = \{I_\mathrm{A}\}$, where I_A is the accepting instruction from Example 2. Then a computation of M on input word (sentence) *aabbcc* will produce the DR-tree depicted in the second step in Fig. 4. The root of the tree $[3, 0, b]$ corresponds to the third input symbol. The node $[5, 0, c]$ denotes the last but one symbol c which is dependent on the fourth symbol (b).

3 Representation of FGD by Enhanced *t*-sRL-Automaton

Our ultimate goal is to model FGD. In this section we introduce an enhanced t-sRL-automaton M_FGD which makes it possible to describe relations between characteristic language, analysis by reduction, tectogrammatical dependency structures, characteristic relation and individual language layers that create the basis of (the formal framework for) FGD.

Let Σ, Γ be alphabets and $\Sigma \subseteq \Gamma$. In the following $\mathsf{Pr}^\Sigma(z)$, where $z \in \Gamma$, denotes the projection from Γ^* onto Σ^*, that is, Pr^Σ is the morphism defined by $a \mapsto a$ (for $a \in \Sigma$) and $A \mapsto \lambda$ (for $A \in \Gamma \smallsetminus \Sigma$).

$\mathsf{M}_\mathsf{FGD} = (\Gamma, ¢, \$, R, A)$ is an enhanced nondeterministic t-sRL-automaton, which is *correctness preserving*. Its characteristic alphabet (vocabulary) consists of four parts $\Gamma = \Sigma_w \cup \Sigma_m \cup \Sigma_a \cup \Sigma_t$, which correspond to the respective layers of FGD (Sect. 1.1). M_FGD can rewrite only symbols from Σ_t and such symbols can be rewritten onto symbols from Σ_t only. Recall that the symbols from individual layers can be quite complex. E.g., '*plánujeme*' is a symbol from the alphabet (vocabulary) Σ_w, '*plánovat*.VB-P-' is a symbol from Σ_m, 'Pred' is a symbol from Σ_a and '*plánovat*.f_PRED.VF1' is a symbol from Σ_t (see Fig. 1).

A language of layer $\ell \in \{w, m, s, t\}$ *accepted by* M_FGD (denoted as $L_\ell(\mathsf{M}_\mathsf{FGD}) = \mathsf{Pr}^{\Sigma_\ell}(L_C(\mathsf{M}_\mathsf{FGD})))$ is the set of all sentences (strings) obtained from $L_C(\mathsf{M}_\mathsf{FGD})$

by removing all the symbols not belonging to the alphabet Σ_ℓ. In particular, $L_w(\mathsf{M_{FGD}})$ represents the set of all correct sentences defined by $\mathsf{M_{FGD}}$.

The characteristic language $L_C(\mathsf{M_{FGD}})$ contains input sequences (over Σ_w) interleaved with language information of the form of symbols from $\Sigma_m \cup \Sigma_a \cup \Sigma_t$. The language of correct Czech sentences is $L_w = \mathsf{Pr}^{\Sigma_w}(L_C(\mathsf{M_{FGD}}))$.

During an accepting computation, the automaton $\mathsf{M_{FGD}}$ collects exactly all the items with tectogrammatical symbols (symbols from Σ_t) into nodes of a resulting DR-tree. Remember that the TR-language of $\mathsf{M_{FGD}}$ represents the set of meanings described by FGD.

Let $z \in L_C(\mathsf{M_{FGD}})$ and let $\mathsf{TR}(z, \mathsf{M_{FGD}})$ denote the set of all DR-trees from $\mathsf{TR}(\mathsf{M_{FGD}})$ resulting from accepting computations of $\mathsf{M_{FGD}}$ on z. It is possible to formulate detailed requirements put on $\mathsf{M_{FGD}}$ in order to obtain *single* tree for each such input z (i.e. $|\mathsf{TR}(z, \mathsf{M_{FGD}})| = 1$, see Appendix).

Remark 3. Let us note that $L_t(\mathsf{M_{FGD}})$ is designed as a deterministic context-free language. Readers familiar with restarting automata can see that $L_C(\mathsf{M_{FGD}})$ is a deterministic context-sensitive language and $L_w(\mathsf{M_{FGD}})$ is a context-sensitive language.

Now we can define TR-*characteristic relation* $\mathsf{TSH}(\mathsf{M_{FGD}})$ of the automaton $\mathsf{M_{FGD}}$, which is the relation between sentence on the (w-layer) and its meaning (t-layer).

$$\mathsf{TSH}(\mathsf{M_{FGD}}) = \{(u,t) \mid \exists y \in L_C(\mathsf{M_{FGD}}), u = \mathsf{Pr}^{\Sigma_w}(y) \text{ and } t \in \mathsf{TR}(y, \mathsf{M_{FGD}})\}.$$

Remark 4. TR-characteristic relation represents important relations in the description of natural language – the relations of synonymy and ambiguity (homonymy). From the characteristic relation, the significant notions of analysis and synthesis can be derived.

For each $t \in \mathsf{TR}(\mathsf{M_{FGD}})$ we introduce TSH-*synthesis* using $\mathsf{M_{FGD}}$ as the set of wordforms u which are in TSH relation with the DR-tree t. Formally:

$$\textit{synt}\text{-}\mathsf{TSH}(\mathsf{M_{FGD}}, t) = \{u \mid (u,t) \in \mathsf{TSH}(\mathsf{M_{FGD}})\} \ .$$

Altogether, TSH-synthesis links the tectogrammatical representation, i.e. DR-tree t from $\mathsf{TR}(\mathsf{M_{FGD}})$ to all corresponding sentences u belonging to $L_w(\mathsf{M_{FGD}})$. This notion makes it possible to study synonymy and its degree.

Finally we introduce a notion dual to the TSH-synthesis – the notion of TSH-*analysis of a string u using* $\mathsf{M_{FGD}}$:

$$\textit{anal}\text{-}\mathsf{TSH}(\mathsf{M_{FGD}}, u) = \{t \mid (u,t) \in \mathsf{TSH}(\mathsf{M_{FGD}})\} \ .$$

For a given sentence u, TSH-analysis returns all its possible tectogrammatical representations t from $\mathsf{TR}(\mathsf{M_{FGD}})$. Hence it models ambiguity of particular surface sentences. This notion represents a formal definition of complete syntactic-semantic analysis using $\mathsf{M_{FGD}}$.

Reduction steps and DR-structures. $\mathsf{M_{FGD}}$ is a restricted instance of an enhanced t-sRL-automaton.

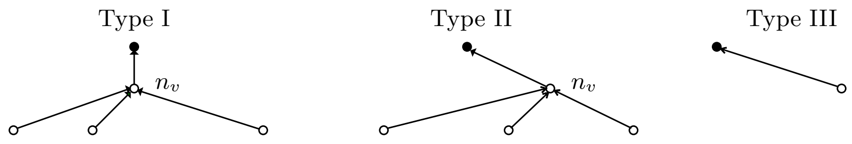


Fig. 5. Types of DR-trees used in $\mathsf{M}_{\mathsf{FGD}}$

A. We distinguish three types of restarting instructions used by $\mathsf{M}_{\mathsf{FGD}}$ (Fig. 5):

Type I instructions are rewriting instructions that process valency. Their corresponding trees are of the height 2 with a single inner node n_v representing a rewritten word v and one or more leaves representing valency complementations of v. The inner node n_v is connected to the root by a vertical edge, the leaves are attached to the inner node by oblique edges only. The rewriting (vertical) edge solves an incompleteness that would arise if v is simply deleted: v is rewritten by its syntactic category; the resulting structure is interpreted as complete with respect to the t-layer.

Type II instructions are deleting instructions that also process a word together with its valency complementations. Their corresponding trees are of the height 2 with a single inner node n_v representing a word v and leaves representing valency complementations of v. The inner node n_v is connected to the root by an oblique edge, the leaves are attached to n_v by oblique edges only. This type of instructions is used if v together with its complementations can be simply deleted without a loss of completeness at the t-layer.

Type III instructions are deleting instructions that process free modifications. The corresponding DR-trees have a root and only one leaf connected to the root by an oblique edge. These instructions are used to process simple reductions which delete a single dependent word.

B. Accepting instructions contain only DR-trees with all edges pointing to the root that corresponds to a governing node of a sentence. The root will become the root of the resulting tree of the whole computation.

C. Each resulting tree $T \in \mathsf{TR}(\mathsf{M}_{\mathsf{FGD}})$ is *projective* (with respect to its descendants); i.e., for each node n of the tree T all its descendants constitute a contiguous segment in the horizontal ordering of nodes of the tree T.

3.1 Concluding Remarks

In this paper, we pursue our studies of a formal model for natural language (Czech in particular) presented in [4,5]. We extend the presentation of the methodology of FGD so that it outputs neither lists of words nor lists of symbols but (tectogrammatical) DR-trees. Such DR-trees can express valencies and simple dependencies in Czech sentences. The model presented in this article captures synonymy and ambiguity as a relation between (surface) sentences and their DR-trees. In this way, we can describe the relation between analysis by reduction and dependency structures at the tectogrammatical layer in detail.

Moreover, we outline a formalization of the basic methodology of FGD in terms of automata theory (see Appendix). This is the main point of our paper as this methodology was presented till now in a way usual in traditional linguistics – that is, quite informally using numerous linguistic examples only.

We envisage that the proposed methodology is not FGD-specific and that similar approach can be used to obtain a formal frame for other language descriptions, as e.g. those presented in [6] and [3]. We plan to focus on them in the near future.

Appendix. Formal definition of enhanced computations of *t*-sRL-automata as well as more details on the principles of FGD can be found in the Appendix posted at the webpage `http://ufal.mff.cuni.cz/~lopatkova/lata10/app.pdf`.

Acknowledgment. The paper reports on the research supported by the grants of GAČR P202/10/1333 and P103/10/0783 and by the grant of MŠMT ČR MSM0021620838.

References

1. Gramatovici, R., Martín-Vide, C.: Sorted Dependency Insertion Grammars. Theor. Comput. Sci. 354(1), 142–152 (2006)
2. Holan, T., Kuboň, V., Oliva, K., Plátek, M.: Two Useful Measures of Word Order Complexity. In: Polguére, A., Kahane, S. (eds.) Proceedings of the Workshop Processing of Dependency-Based Grammars, Montréal, Quebeck, pp. 21–28 (1998)
3. Kunze, J.: Abhängigkeitsgrammatik. Studia Grammatica, vol. XII. Akademie Verlag, Berlin (1975)
4. Lopatková, M., Plátek, M., Kuboň, V.: Modeling Syntax of Free Word-Order Languages: Dependency Analysis by Reduction. In: Matoušek, V., Mautner, P., Pavelka, T. (eds.) TSD 2005. LNCS (LNAI), vol. 3658, pp. 140–147. Springer, Heidelberg (2005)
5. Lopatková, M., Plátek, M., Sgall, P.: Towards a Formal Model for Functional Generative Description: Analysis by Reduction and Restarting Automata. The Prague Bulletin of Mathematical Linguistics 87, 7–26 (2007)
6. Mel'čuk, I.A.: Dependency Syntax: Theory and Practice. State University of New York Press, Albany (1988)
7. Messerschmidt, H., Mráz, F., Otto, F., Plátek, M.: Correctness Preservation and Complexity of Simple RL-automata. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 162–172. Springer, Heidelberg (2006)
8. Otto, F.: Restarting Automata and Their Relation to the Chomsky Hierarchy. In: Ésik, Z., Fülöp, Z. (eds.) DLT 2003. LNCS, vol. 2710, pp. 55–74. Springer, Heidelberg (2003)
9. Sgall, P., Hajičová, E., Panevová, J.: The Meaning of the Sentence in Its Semantic and Pragmatic Aspects. Reidel, Dordrecht (1986)

A Randomized Numerical Aligner (rNA)

Alberto Policriti[1,2], Alexandru I. Tomescu[1,3], and Francesco Vezzi[1,2]

[1] Dipartimento di Matematica e Informatica,
Università di Udine, Via delle Scienze, 206, 33100 Udine, Italy
{policriti,alexandru.tomescu,francesco.vezzi}@dimi.uniud.it
[2] Istituto di Genomica Applicata (IGA),
Via J.Linussio, 51, 33100 Udine, Italy
[3] Faculty of Mathematics and Computer Science,
University of Bucharest, Str. Academiei, 14, 010014 Bucharest, Romania

Abstract. With the advent of new sequencing technologies able to produce an enormous quantity of short genomic sequences, new tools able to search for them inside a references sequence genome have emerged. Because of chemical reading errors or of the variability between organisms, one is interested in finding not only exact occurrences, but also occurrences with up to k mismatches. The contribution of this paper is twofold. On one hand, we present a generalization of the classical Rabin-Karp string matching algorithm to solve the k-mismatch problem, with average complexity $\mathcal{O}(n+m)$. On the other hand, we show how to employ this idea in conjunction with an index over the text, allowing to search a pattern, with up to k mismatches, in time proportional to its length. This novel tool—rNA (randomized Numerical Aligner)—outperforms available tools like SOAP2, BWA, and BOWTIE, processing up to 10 times more patterns per second on texts of (practically) significant lengths.

1 Introduction

One of the main applications of string matching is computational biology. A DNA sequence can be seen as a string over the alphabet $\Sigma = \{A, C, G, T\}$. Given a reference genome sequence, we are interested in searching (*aligning*) different sequences (*reads*) of various lengths. Reads are produced by sequencing machines able to read stretches of the DNA of a given organism. When aligning such reads against another DNA sequence, we must take care of errors due to the sequencer and of intrinsic errors do to the variability between organisms. For these reasons, all the programs that align reads against a reference sequence have to deal with mismatches [2,13]. String matching can be divided into two main areas: exact string matching and approximate string matching.

The most famous exact string matching algorithms are [14,4,12], requiring time $\Theta(n + m)$ (n text and m pattern lengths, respectively). When a great number of patterns must be searched, it is convenient to build an index over the text, such as a Suffix Tree, e.g. [25], or a Suffix Array [21]. The most popular tool for searching inside DNA strings [2] is based on the construction of an index. For a complete review on indexing algorithms refer to [6].

Approximate string matching at distance k under the *edit distance* is called the *k-difference problem*, while under the *Hamming distance*, it is called the *k-mismatch problem*. A naive algorithm for the k-difference problem is based on dynamic programming and it has a running time $\mathcal{O}(nm)$. Several efforts where made to improve this result. In [1] the k-mismatch problem is solved in time $\mathcal{O}(n\sqrt{m \log m})$, while [15,16] introduced a method running in time $\mathcal{O}(nk)$. The algorithm of [7] attains the same complexity $\mathcal{O}(nk)$. A more recent paper [24] proposed a variation of FAAST [20] that has average running time $\mathcal{O}(n(\log m + k)/m)$. The asymptotic running time was improved in [3] to $\mathcal{O}(n\sqrt{k \log k})$, by a method based on counting and filtering, the suffix tree with kangaroo hooping, and fast Fourier transforms, which may ultimately lead to a more sophisticated implementation.

The first algorithm that solved the k-mismatch problem with the construction of an index is [11]. The first solution with the query time depending only on k and m was proposed in [25] using Suffix Trees. More recently [10], the k-difference problem has been solved in time $\mathcal{O}(|\Sigma|^k m^k \max(k, \log n))$ where Σ is the alphabet, using compressed Suffix Arrays [8].

In many practical applications, we are interested in finding the *best* occurrence of the pattern, with at most k mismatches (the *best k-mismatch problem*—to be introduced in Section 3). Recently, a flurry of papers presenting new indexing algorithms to solve this problem appeared [18,19,17]. All these algorithms aim to search inside a reference sequence the myriad of reads that are produced by new sequencing technologies[1]. For example, the Illumina Genome Analyzer is able to produce 22 Giga bases of high quality output in a single experiment. Each read has length 100 bases, and in the close future it will reach 150 bases. Tools like SOAP2 [19] are able to align this large set of reads in a very short time, thanks to advanced indices and heuristics, that can, however, reduce accuracy.

In this paper we will focus on the *best k-mismatch problem*. First we will show how we can generalize the on-line algorithm of Rabin and Karp [12] to solve the k-mismatch problem, with an average complexity of $\mathcal{O}(n + m)$. In order to solve the best k-mismatch problem, we will then employ this idea in conjunction with an index over the text, allowing to search a pattern in time proportional to its length. We call our tool a randomized Numerical Aligner (**rNA**)[2]. Even though we do not sacrifice accuracy, the experimental results show that our algorithm has significantly better performances than the most used aligners for short reads, like [18,19,17].

Problem definition and notations. Let $\Sigma = \{0, 1, \ldots, b-1\}$ be an alphabet of $b \geqslant 2$ characters, and let $c, d \in \Sigma$. Define $\text{neq}(c, d) = 1$ if $c \neq d$, and 0 otherwise. Let $X = X[0]X[1]\ldots X[n-1]$ and $Y = Y[0]Y[1]\ldots Y[n-1]$ be two strings over the alphabet Σ. The *Hamming distance* between X and Y is defined as $d_H(X, Y) =_{\text{def}} \sum_{i=0}^{n-1} \text{neq}(X[i], Y[i])$. Given numbers $0 < m \leqslant n$ and $0 \leqslant s \leqslant n - m$, we denote by $X_{(s)}$ the string $X_{(s)} =_{\text{def}} X[s]X[s+1]\ldots X[s+m-1]$. We denote the numerical radix-b representation of the string X of length n by

[1] `www.illumina.com`, `www.solid.com`, `www.appliedbiosystems.com`

[2] Source code available upon request.

$x =_{\mathrm{def}} b^{n-1}X[0] + b^{n-2}X[1] + \dots bX[n-2] + X[n-1]$. Given a positive integer q, the number $\hat{x}$ stands for $x \bmod q$, and is called the *fingerprint* of the string X. The *k-mismatch problem* is defined as follows:

IN: Text $T = T[0]T[1]\dots T[n-1]$, pattern $P = P[0]P[1]\dots P[m-1]$, over the alphabet Σ, and a natural number $k < m$.
OUT: All pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$.

For such a pair $\langle s, d_H(P, T_{(s)})\rangle$, we say that P occurs (with mismatches) with shift s in T. If $d_H(P, T_{(s)}) = 0$, we say that $T_{(s)}$ is an exact occurrence of P.

2 An On-Line Algorithm for String Matching with *k* Mismatches

One of the simplest exact string matching algorithms, that also performs well in practice, is the Rabin-Karp randomized algorithm [12]. For every $s = 0 \dots n-m$, the algorithm encodes P and any $T_{(s)}$ by the radix-b numbers p and $t_{(s)}$, respectively, such that expensive string comparisons are replaced by constant-time numerical comparisons. As usually m is larger than the length of a processor word, instead of storing p and $t_{(s)}$, one keeps the values $\hat{p} = p \bmod q$ and $\hat{t}_{(s)} = t_{(s)} \bmod q$. As an indication that P *may* occur with shift s in T, it now tests whether $\hat{p} = \hat{t}_{(s)}$, and if so, it proceeds to a character-by-character comparison of P and $T_{(s)}$. Randomly choosing q to be a prime number in the interval $[2, mn^2]$, the test $\hat{p} = \hat{t}_{(s)}$ produces few false positives [12] (i.e., it gives a positive answer in the case when $P \neq T_{(s)}$). Moreover, as $\hat{t}_{(s+1)}$ can be computed from $\hat{t}_{(s)}$ in constant time, the overall expected time complexity is $\mathcal{O}(n+m)$.

The Rabin-Karp method has already been employed in [22] to solve the k-mismatch problem. That approach is based on generating all the $\sum_{i=0}^{k} \binom{m}{i}(b-1)^i$ strings obtained from P with at most k mismatches. In this paper we will instead make use of some algebraic properties of the Hamming distance under the modulo operation. In this way, we can replace 'generation' by 'verification', and we can reduce the exponential blow-up on m, to an exponential blow-up on the length w of a processor word.

We will retain the advantageous features of the Rabin-Karp algorithm, like encoding strings by a radix-b number, and storing values modulo an appropriate number q. The only point where a change is needed is in the heuristic check whether the pattern occurs with shift s (i.e., in the test $\hat{p} = \hat{t}_{(s)}$). In what follows, we will seek an answer to these questions:

1. If $d_H(P, T_{(s)}) \leqslant k$, then what *fast* test on the available data (e.g., $\hat{p}$, $\hat{t}_{(s)}$) can we use to detect such a situation?
2. How can we guarantee that this test produces *few* false positives, and what is the probability of such an event?

We note that when $k = 0$, then $\hat{p} = \hat{t}_{(s)}$ is equivalent to $(\hat{p} - \hat{t}_{(s)}) \bmod q = 0$. With this clue in mind, we still compute $(\hat{p} - \hat{t}_{(s)}) \bmod q$, but we will try to characterize the set $\mathcal{Z}(k,q) \subseteq \{0,\ldots,q-1\}$, such that whenever $d_H(P,T_{(s)}) \leqslant k$, then $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ holds. More formally, the set $\mathcal{Z}(k,q)$ is defined as follows.

Definition 1. *Given $m > 0$, $0 < k < m$ and $q > 0$, define $\mathcal{Z}(k,q)$*[3] *to be the set*

$$\mathcal{Z}(k,q) =_{\text{def}} \{(x-y) \bmod q \mid X,Y \in \Sigma^m,\ d_H(X,Y) \leqslant k\}.$$

The algebraic difference between the numerical representations of two strings at a given Hamming distance is characterized in Lemma 1.

Lemma 1. *Given two strings X and Y of the same length m, for any $0 < k < m$ we have $d_H(X,Y) = k$* if and only if

$$x - y \in \{(-1)^{u_1} t_1 b^{i_1} + \ldots + (-1)^{u_k} t_k b^{i_k} : i_1 > \ldots > i_k \in \{0,\ldots,m-1\},\\ u_1,\ldots,u_k \in \{0,1\},\ t_1,\ldots,t_k \in \{1,\ldots,b-1\}\}.$$

Plainly, from Lemma 1, $\mathcal{Z}(k,q)$ can be expressed as

$$\begin{aligned}\mathcal{Z}(k,q) = \{&\big((-1)^{u_1} t_1 b^{i_1} + \ldots + (-1)^{u_j} t_j b^{i_j}\big) \bmod q : 0 < j \leqslant k,\\ &i_1 > \ldots > i_j \in \{0,\ldots,m-1\},\\ &u_1,\ldots,u_j \in \{0,1\},\ t_1,\ldots,t_j \in \{1,\ldots,b-1\}\} \cup \{0\}.\end{aligned}$$

An upper bound for the cardinality of $\mathcal{Z}(k,q)$ is $\min\{q, \sum_{j=0}^{k} \binom{m}{j} (2(b-1))^j\}$, as for each $0 \leqslant j \leqslant k$, there are $\binom{m}{j}$ ways to choose j pairwise distinct $i_1,\ldots,i_j$, and $(2(b-1))^j$ ways to choose $u_1,\ldots,u_j$ and $t_1,\ldots,t_j$.

In order for the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ to give few false positives, the size of $\mathcal{Z}(k,q)$ must be *small*, which, working modulo an arbitrary number q, may not be true. The main idea of our approach is to choose $q = b^w - 1$, where $w < m$ is a natural number large enough, according to a few complexity considerations[4]. The following lemma shows that the choice $q = b^w - 1$ makes $\mathcal{Z}(k,q)$ have a small cardinality.

Lemma 2. *Given $w < m$,*

$$\begin{aligned}\mathcal{Z}(k,b^w-1) = \{&\big((-1)^{u_1} t_1 b^{i_1} + \ldots + (-1)^{u_j} t_j b^{i_j}\big) \bmod (b^w-1) : 0 < j \leqslant k,\\ &i_1 > \ldots > i_j \in \{0,\ldots,w-1\},\\ &u_1,\ldots,u_j \in \{0,1\},\ t_1,\ldots,t_j \in \{1,\ldots,b-1\}\} \cup \{0\}.\end{aligned}$$

[3] We will sometimes refer to the elements of $\mathcal{Z}(k,q)$ as *witnesses*, as they testify that two strings *can* be at Hamming distance at most k.

[4] Notice that, arithmetic modulo numbers of the form $2^w - 1$ (called Mersenne numbers) is used in various applications, like digital systems based on residue number system, or cryptography, therefore, efficient VLSI circuit architectures for addition and multiplication modulo $2^w - 1$ have been proposed over the years (see, e.g., the discussion in [26], and the references therein). Notice also that, in general, the usage of q of the form $2^w - 1$ is *not* suggested when exact search is performed.

Proof. See the extended version of this paper [23].

Hence, $|\mathcal{Z}(k, b^w - 1)|$ is at most $\sum_{j=0}^{k} \binom{w}{j}(2(b-1))^j$, as for each $0 \leqslant j \leqslant k$, there are $\binom{w}{j}$ ways to choose j pairwise distinct $i_1, \dots, i_j$, and $(2(b-1))^j$ ways to choose $u_1, \dots, u_j$ and $t_1, \dots, t_j$.

Onwards, we suppose to work modulo $q = b^w - 1$, without explicitly mentioning it. The generalized algorithm (shown as Algorithm 1) works in a similar manner as the Rabin-Karp algorithm [12]. It starts by setting $q = b^w - 1$, $s = 0$, and by computing $\hat{p} = p \bmod q$ and $\hat{t}_{(0)} = t_{(0)} \bmod q$, using Horner's rule and bringing into play the linearity of the modulo operation. Then, for each $0 \leqslant s \leqslant n - m$ it checks whether $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k, q)$. If yes, it performs a character-by-character comparison of P and $T_{(s)}$. When incrementing s, the value $\hat{t}_{(s)}$ can be computed in constant time, as follows. For all $0 \leqslant s < n - m$, we have $\hat{t}_{(s+1)} = b \cdot (t_{(s)} - b^{m-1} T[s]) + T[s+m]$. Working modulo q, this equation becomes $\hat{t}_{(s+1)} = \big(b \cdot (\hat{t}_{(s)} - (b^{m-1} \bmod q) T[s]) + T[s+m]\big) \bmod q$. If we let $h =_{\text{def}} b^{m-1} \bmod q = b^{(m-1) \bmod w}$, we get $\hat{t}_{(s+1)} = \big(b \cdot (\hat{t}_{(s)} - h \cdot T[s]) + T[s+m]\big) \bmod q$.

Algorithm 1. String matching with k mismatches

```
Input: $T = T[0]T[1]\dots T[n-1]$, $P = P[0]P[1]\dots P[m-1]$, both over the
       alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number of mismatches $k$ $(0 \leqslant k < m)$
       and word length $w$.
Output: All pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n - m$ and $d_H(P, T_{(s)}) \leqslant k$.
1  $q \leftarrow b^w - 1$; $h \leftarrow b^{m-1 \bmod w}$;
2  $\mathcal{Z} \leftarrow$ GENERATEZ$(k, q)$; SOLUTION $\leftarrow \emptyset$;
3  $\hat{p} \leftarrow \hat{t} \leftarrow 0$;
4  for $i \leftarrow 0$ to $m - 1$ do
5      $\hat{p} \leftarrow (b \cdot \hat{p} + P[i]) \bmod q$;
6      $\hat{t} \leftarrow (b \cdot \hat{t} + T[i]) \bmod q$;
7  if $(\hat{p} - \hat{t}) \bmod q \in \mathcal{Z}$ then
8      if $d_H(P, T_{(0)}) \leqslant k$ then
9          SOLUTION $\leftarrow$ SOLUTION $\cup \{\langle 0, d_H(P, T_{(0)})\rangle\}$;
10 for $s \leftarrow 1$ to $n - m$ do
11     $\hat{t} \leftarrow (b \cdot (\hat{t} - h \cdot T[s-1]) + T[s+m-1]) \bmod q$;
12     if $(\hat{p} - \hat{t}) \bmod q \in \mathcal{Z}$ then
13         if $d_H(P, T_{(s)}) \leqslant k$ then
14             SOLUTION $\leftarrow$ SOLUTION $\cup \{\langle s, d_H(P, T_{(s)})\rangle\}$;
15 return SOLUTION;
```

In Algorithm 1 we assume that procedure GENERATEZ(k, q) generates the set $\mathcal{Z}(k, q)$. In order to evaluate the expected complexity of the string matching phase of Algorithm 1, we follow the formalism of [5, Ch. 32.2]. We have to take into account the time $c(q)$ the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}$ on lines 7 and 12 takes, and the number of false positives produced by it. If we denote by $p(q)$ the probability that at a specific shift $0 \leqslant s \leqslant n-m$ this test will produce a false positive, we can estimate the number of false positives as $n \cdot p(q)$. Considering ν to be the number of occurrences of P in T with at most k mismatches, the expected complexity is

$$\mathcal{O}(n \cdot c(q) + (m \cdot \nu + m \cdot n \cdot p(q)).$$

In many applications ν is small (i.e., $\mathcal{O}(1)$) and if we choose q such that $n \cdot p(q) \leqslant 1$, then the expected complexity becomes $\mathcal{O}(n \cdot c(q) + m)$. The only values of $t_{(s)}$ for which $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$, but $d_H(P, T_{(s)}) > k$ are of the form $p+z+j \cdot q$, where $z \in \mathcal{Z}(k,q)$ and $0 \leqslant j \leqslant \lfloor b^m/q \rfloor$. As we have at most $\lfloor b^m/q \rfloor |\mathcal{Z}(k,q)|$ such values, and there are at most b^m possible values for $t_{(s)}$, the probability that at a specific shift s, the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ produces a false positive is $p(q) \leqslant \frac{|\mathcal{Z}(k,q)|}{q}$, under the assumption that the operation $\bmod\,(b^w - 1)$ uniformly distributes numbers in the interval $[0 \ldots q-1]$ (for example when $b^w - 1$ is a prime number).

Therefore, to attain the desired time complexity, one has to choose $q = b^w - 1$ such that $b \cdot q$ fits into a processor word and such that $q \geqslant n|\mathcal{Z}(k,q)|$. Working on a 32-bit processor, with strings over the alphabet $\{0,1,2,3\}$, limits w to 15, therefore, if n or k are large enough, a flurry of false positives are due to appear. If we use a 64-bit architecture, w is limited by 31, and hence the number of false positives drastically decreases. These numbers are computed in Table 1.

Table 1. The average number of false positives returned by the heuristic test $(\hat{p} - \hat{t}_s) \bmod q \in \mathcal{Z}(k, 4^w - 1)$, when $|\Sigma| = 4$, n =4,000,000,000, and $w = 15$ (32-bit architecture) and $w = 31$ (64-bit architecture)

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
false positives on 32 bits	3.73	339	13079	279959	3662224	30549760
false positives on 64 bits	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	2.4

We choose to implement the test $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$ by generating before-hand the set $\mathcal{Z}(k,q)$, in time $\mathcal{O}(|\mathcal{Z}(k,q)|)$. The data structure storing it can be an ordered array, with search complexity $\mathcal{O}(\log|\mathcal{Z}(k,q)|)$. A data structure more appropriate for unsuccessful queries, as we expect most of them to be, is a trie, with worst case search time $\mathcal{O}(w)$. However, due to better memory locality, a hash table with collisions resolved by chaining is preferred. Under the assumption of simple uniform hashing and using $\mathcal{O}(\alpha)$ memory, the average search complexity becomes $\mathcal{O}(1 + |\mathcal{Z}(k,q)|/\alpha)$.

If one agrees to use an additional amount $\mathcal{O}(q)$ of memory, then $\mathcal{Z}(k,q)$ can be simply stored as a direct-address table $\mathcal{Z}[0 \ldots q-1]$, where $\mathcal{Z}[z] = 1$ iff $z \in \mathcal{Z}(k,q)$. Therefore, we have the following:

Theorem 1. *Algorithm 1 solves the k-mismatch problem; if $q = b^w - 1 \geqslant |\mathcal{Z}(k,q)|$, and if the set $\mathcal{Z}(k,q)$ is stored as a direct-address table of size $\mathcal{O}(q)$, its expected search complexity is $\mathcal{O}(n + m + |\mathcal{Z}(k,q)|)$. If $n \gg m$ then the expected search time is $\mathcal{O}(n)$.*

3 A Randomized Numerical String Aligner with k Mismatches

A string aligner is given a text T, of length n, and a collection $\mathcal{P}$ of patterns, and is required to find all occurrences of P in T with at most k mismatches, for all $P \in \mathcal{P}$. In many applications, like in the search of a set of sequences (reads) inside a reference sequence genome, we are interested in finding the *best* occurrence of a string with at most k mismatches, instead of find all its occurrences with less than k mismatches. This problem (referred to in what follows as the *best k-mismatch problem*) can be reformulated in the following way:

IN: Text $T = T[0]T[1]\dots T[n-1]$, a collection $\mathcal{P}$ of patterns of length m, all over the alphabet Σ, and a natural number $k < m$.

OUT: For every $P \in \mathcal{P}$, all pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$ and $d_H(P, T_{(s)}) \leqslant k$, such that for all $0 \leqslant s' \leqslant n-m$ we have $d_H(P, T_{(s)}) \leqslant d_H(P, T_{(s')})$.

A naive approach is to iteratively apply Algorithm 1 for each $P \in \mathcal{P}$, with an overall average time complexity of $\mathcal{O}((n+m)|\mathcal{P}|)$. However, we choose to build in time $\Theta(n)$ and space $\Theta(n)$ the index[5]

$$\mathcal{T} = \{\langle \hat{t}_{(s)}, s\rangle : 0 \leqslant s \leqslant n-m\}.$$

The shifts s in T which *may* be exact occurrences of a $P \in \mathcal{P}$ correspond to those pairs $\langle \hat{p}, s\rangle \in \mathcal{T}$. The set $\mathcal{T}$ can be stored is a way similar to a hash by chaining. We use an array indexed by numbers from 0 to $q-1$, having, for all $0 \leqslant r \leqslant q-1$, $\mathcal{T}[r] = \{s_1, \dots, s_l\}$ iff for all $1 \leqslant i \leqslant l$, $\hat{t}_{(s_i)} = r$.

Algorithm 1 can be adapted to use the index $\mathcal{T}$, by reverting from 'verification' back to 'generation'. For every $P \in \mathcal{P}$, we are interested in finding all the shifts s in T which *may* be occurrences of P with at most k mismatches. They correspond to those pairs $\langle \hat{t}_{(s)}, s\rangle \in \mathcal{T}$ such that $(\hat{p} - \hat{t}_{(s)}) \bmod q \in \mathcal{Z}(k,q)$. Using the linearity of the modulo operation, we thus iteratively search in $\mathcal{T}$ all numbers $(\hat{p} - z) \bmod q$, for every $z \in \mathcal{Z}(k,q)$. For all shifts s such that $\langle (\hat{p} - z) \bmod q, s\rangle \in \mathcal{T}$, we check that indeed $d_H(P, T_{(s)}) \leqslant k$. The average complexity of a search of a pattern is thus $\mathcal{O}(m + |\mathcal{Z}(k,q)|)$, amounting to a total complexity of $\mathcal{O}(n + (m + |\mathcal{Z}(k,q)|)|\mathcal{P}|)$.

However, the bigger w is, the lower the probability of a false positive is, but the bigger $|\mathcal{Z}(k,q)|$ gets, and vice-versa. We can remediate this problem by a rather standard use (in this field) of the pigeonhole principle.

Given a pattern $P = P[0]P[1]\dots P[m-1]$ and a positive integer $1 \leqslant t \leqslant m$, for every $0 \leqslant i < t$, we denote by $P_t(i)$ its substring $P[i\lfloor m/t \rfloor]\dots P[(i+1)\lfloor m/t \rfloor - 1]$ and call it the ith *block* of P. Note that the t blocks of P do not overlap, a crucial property for the following lemma to hold.

Lemma 3. *Let T be a text, $P = P[0]P[1]\dots P[m-1]$ be a pattern, and t be a positive integer, $1 \leqslant t \leqslant m$. If P occurs in T with at most k mismatches, then there is at least one block $P_t(i)$ of P that occurs in T with at most $\lfloor k/t \rfloor$ mismatches.*

[5] For a discussion on different indexing strategies, refer to [9].

Algorithm 2. The randomized Numerical Aligner (**rNA**) with k mismatches

Input: Text $T = T[0]T[1]\dots T[n-1]$, a collection $\mathcal{P}$ of patterns of length m, all over the alphabet $\Sigma = \{0, 1, \dots, b-1\}$, number k of mismatches ($0 \leqslant k < m$), the number t of blocks in which the patterns get divided ($1 \leqslant t \leqslant k+1$), and word length w.

Output: For all $P \in \mathcal{P}$, all pairs $\langle s, d_H(P, T_{(s)})\rangle$, where $0 \leqslant s \leqslant n-m$, $d_H(P, T_{(s)}) \leqslant k$ and for all $0 \leqslant s' \leqslant n-m$, it holds that $d_H(P, T_{(s)}) \leqslant d_H(P, T_{(s')})$.

```
procedure SearchPattern($P$)
    for $i \leftarrow 0$ to $t-1$ do    //compute the fingerprints of all the blocks of $P$
        $\hat{p}_t(i) \leftarrow 0$;
        for $j \leftarrow i \cdot l$ to $(i+1) \cdot l - 1$ do
            $\hat{p}_t(i) \leftarrow (b \cdot \hat{p}_t(i) + P[j]) \bmod q$;
    Solution $\leftarrow \emptyset$;
    $best\_k \leftarrow k$;    //the smallest distance at which to search onwards
    $exact\_occurrence \leftarrow false$;
    $j \leftarrow 0$;
    while $j < |\mathcal{Z}(\lfloor best\_k/t \rfloor, w)|$ do    //for every witness $\mathcal{Z}[j]$
        $i \leftarrow 0$;
        while $i \leqslant t-1$ and $(\neg exact\_occurrence)$ do    //for every block $i$
            forall $s \in indexT[(\hat{p}(i) - \mathcal{Z}[j]) \bmod q]$ do
                if $s - i \cdot l \geqslant 0$ and $d_H(P, T_{(s-i \cdot l)}) \leqslant best\_k$ then
                    if $d_H(P, T_{(s-i \cdot l)}) < best\_k$ then
                        $best\_k \leftarrow d_H(P, T_{(s-i \cdot l)})$;
                        Solution $\leftarrow \emptyset$;
                    Solution $\leftarrow$ Solution $\cup \{\langle s - i \cdot l, best\_k \rangle\}$;
            if $best\_k = 0$ then $exact\_occurrence \leftarrow true$;
            $j \leftarrow j + 1$;
    print Solution;
end

$q \leftarrow b^w - 1$;
$l \leftarrow \lfloor m/t \rfloor$;    // the length of each of the $t$ blocks of $P$
$indexT \leftarrow$ PreProcessText$(T, b, l, q)$;
$\mathcal{Z} \leftarrow$ GenerateZ$(k, q)$;
forall $P \in \mathcal{P}$ do
    SearchPattern($P$);
```

Accordingly, instead of searching for an entire pattern P with at most k mismatches, we can perform t searches for all of the blocks of P, each with at most $\lfloor k/t \rfloor$ mismatches. Each occurrence of a block $P_t(i)$ $(0 \leqslant i < t)$ of P in T, with shift s, is an indication that P may occur in T with shift $s - i\lfloor m/t \rfloor$. As we are interested in finding the best occurrences of P in T, we will keep the smallest number of mismatches at which an occurrence of P has been found so far in a variable *best_k*. In this way, each block of the pattern is searched with at most $\lfloor best\_k/t \rfloor$ mismatches. The pseudo-code is given as Algorithm 2.

Procedure PREPROCESSTEXT(T, b, l, q) builds the index $\mathcal{T}$ that allows the search of strings of length $l = \lfloor m/t \rfloor$. Procedure GENERATEZ(k, q) returns an array containing the elements of the set $\mathcal{Z}(k, q)$, ordered as follows: for all $0 \leqslant i < k$, the elements of $\mathcal{Z}(i, q)$ are placed before the elements of $\mathcal{Z}(i+1, q) \setminus \mathcal{Z}(i, q)$.

The procedure SEARCHPATTERN(P) divides the pattern in t blocks, each of length $l = \lfloor m/t \rfloor$. For each block $P_t(i)$ $(0 \leqslant i < t)$, its fingerprint $\hat{p}_t(i)$ is computed employing Horner's rule and the linearity of the modulo operation (lines 2 – 5). The variable *best_k* stores the smallest distance at which an occurrence of P has been found so far, while *exact_occurrence* indicates whether an *exact* occurrence has been found in the text.

For each index j $(0 \leqslant j < |\mathcal{Z}(\lfloor best\_k/t \rfloor, q)|)$, we iteratively search in the text every *block* $P_t(i)$ $(0 \leqslant i < t)$, with at most $\lfloor best\_k/t \rfloor$ mismatches (line 12). Every such shift s where the block i may occur is an indication that the *pattern* may occur at shift $s - i \cdot l$ with at most *best_k* mismatches (if, of course, $s - i \cdot l \geqslant 0$).

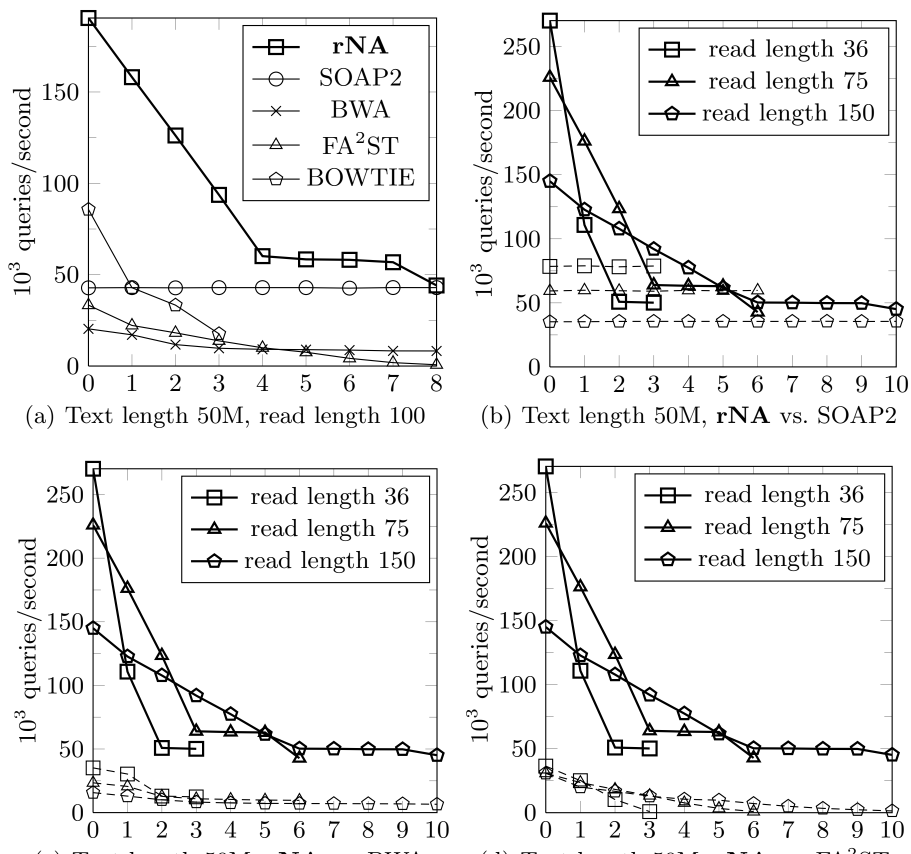
If this is indeed the case (line 14), we have to check whether the current occurrence is at distance strictly smaller than *best_k* (line 15). If so, the variable *best_k* is updated with the current distance, and all the shifts s stored so far in the set SOLUTION are discarded. Anyhow, the current shift s together with *best_k* are added to SOLUTION. In other words, at every step of the computation, the set SOLUTION stores occurrences only at distance *best_k*.

Lastly, in line 19 we implement the following trick: if the pattern occurs in an exact manner in the text, then the first block does as well. Since this block will indicate all exact occurrences, searching the remaining blocks of P brings no additional information. Therefore, we set *exact_occurrence* to true, stopping the search (this is true because *best_k* was changed to 0, hence the loop in line 10 is no longer executed).

4 Experimental Results

The main field where string alignment tools are used is bioinformatics. Such tools are used in order to search inside a reference sequence genome a sets of sequences (reads) generated by a sequencer, an instrument able to read DNA. Over the last years, a new generation of sequencers has emerged (see, e.g., `www.illumina.com`). Compared with previous sequencers (so called Sanger sequencers), the main features of these novel machines are the enormous quantity of data generated and the short length of the sequences that they give in output. For example, a single Illumina experiment can produce 22 Giga bases of output grouped in reads of length 100 (their length is expected to grow to 150 bases in the near future).

For these reasons, many of the currently available aligners sometimes sacrifice correctness over speed, by skipping a small number of occurrences of a read.



(a) Text length 50M, read length 100
(b) Text length 50M, **rNA** vs. SOAP2
(c) Text length 50M, **rNA** vs. BWA
(d) Text length 50M, **rNA** vs. FA^{2}ST

Fig. 1. The number of errors is represented on the X-axis, while the Y-axis indicates the number of reads processed per second. Figure 1(a) compares the read throughput of the 5 algorithms when aligning reads of length 100 against a reference sequence of length 50M. Figures 1(b) - 1(d) compares the performances of **rNA** (continuous lines) with SOAP2, BWA, and FA^{2}ST (dashed lines) when varying the read length (36, 75, 150) and the number of errors.

For example, SOAP2 [19] searches only the first $l \leqslant m$ characters of the read in the reference text with at most $v \leqslant 2$ mismatches, and for each such occurrence, it checks that the entire pattern occurs with at most k mismatches (both parameters l and v are chosen by the user).

Our tool was compared against SOAP2 [19], BWA [18], BOWTIE [17], and FA^{2}ST (C. Del Fabbro, PhD thesis, in preparation, 2009). The last tool implements Suffix Arrays and relies on the idea behind Lemma 3, where t, the number of blocks in which the pattern gets divided, is always chosen to be $k+1$. Like **rNA**, it is the only aligner, to our knowledge, that solves the best k-mismatch problem in an accurate manner.

The tests where performed over a machine running Linux 2.6.24, on two quad-core Intel Xeon 3Ghz processors with 32GB of RAM. Even if the algorithm we proposed is easily parallelizable, all the experiments were run using only a single CPU. The algorithm was written in C++ and compiled with the GNU gcc 4.2 compiler with the options `-O3 -static-libgcc`. The test data was constructed by extracting from the grapevine genome 4 sequences of sizes 50K, 500K, 5M, 50M, to be used as references. From each such reference, we extracted 400.000 reads of length m ($m \in \{36, 50, 75, 100, 150\}$), with an average error rate of 1%. These assumptions are similar to the technical specifications of the Illumina sequencer. For all the possible combinations of tool, text length and read length, several experiments where done varying the input parameters and only the best result was considered.

Because of length constraints, we summarize the most significant results in Figure 1. Figure 1(a) compares the 5 tools on reference length 50M and query length 100. When allowing less than 4 mismatches, **rNA** greatly outperforms all other tools. Instead, if this numbers increases, the only tool that achieves comparable results is SOAP2 (whose performances tend to be constant). However, in real applications we are interested in alignments that have similarities higher than 95%, implying that it is not biologically relevant to search a read of length 100 with more than 5 mismatches. Figure 1(b) illustrates a complete comparison between **rNA** and SOAP2 on a reference of length 50M bases. When the ratio between the number of mismatches and the length of the read is low, **rNA** is significantly faster than SOAP2. In particular, for read length 150 and at most 10 mismatches, **rNA** is always better than SOAP2.

References

1. Abrahamson, K.: Generalized string matching. SIAM Journal on Computing 16(6), 1039–1051 (1987)
2. Altschul, S.F., Madden, T.L., Schäffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research 25(17), 3389–3402 (1997)
3. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. Journal of Algorithms 50, 257–275 (2004)
4. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Commun. ACM 20(10), 762–772 (1977)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press, McGraw-Hill Book Company (2001)
6. Ferragina, P.: String algorithms and data structures. CoRR abs/0801.2378 (2008)
7. Galil, Z., Giancarlo, R.: Improved string matching with k mismatches. SIGACT News 17(4), 52–54 (1986)
8. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. 35(2), 378–407 (2005)
9. Horner, D.S., Pavesi, G., Castrignano, T., De Meo, P.D., Liuni, S., Sammeth, M., Picardi, E., Pesole, G.: Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing. Brief. Bioinform., bbp046+ (2009)

10. Huynh, T.N.D., Hon, W.K., Lam, T.W., Sung, W.K.: Approximate string matching using compressed suffix arrays. Theor. Comput. Sci. 352(1), 240–249 (2006)
11. Jokinen, P., Ukkonen, E.: Two algorithms for approximate string matching in static texts. In: Proc. 2nd Ann. Symp. on Mathematical Foundations of Computer Science, vol. 520, pp. 240–248 (1991)
12. Karp, R., Rabin, M.: Efficient randomized pattern-matching algorithms. IBM J. Res. Develop. 31(2), 249–260 (1987)
13. Kent, W.J.: BLAT—The BLAST-like Alignment Tool. Genome research 12(4), 656–664 (2002)
14. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)
15. Landau, G.M., Vishkin, U.: Efficient string matching in the presence of errors. In: Proceedings of the 26th IEEE Symposium on Foundations of Computer Science, pp. 126–136 (1985)
16. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theoretical Computer Science 43, 239–249 (1986)
17. Langmead, B., Trapnell, C., Pop, M., Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. Genome Biology 10(3), R25 (2009)
18. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics 25(14), 1754–1760 (2009)
19. Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K., Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics 25(15), 1966–1967 (2009)
20. Liu, Z., Chen, X., Borneman, J., Jiang, T.: A fast algorithm for approximate string matching on gene sequences. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM 2005. LNCS, vol. 3537, pp. 79–90. Springer, Heidelberg (2005)
21. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: SODA '90: Proc. 1st Ann. ACM-SIAM Symp. on Discrete Algorithms, pp. 319–327. Society for Industrial and Applied Mathematics, Philadelphia (1990)
22. Muth, R., Manber, U.: Approximate multiple string search. In: Proc. 7th Ann. Symp. on Combinatorial Pattern Matching, Laguna Beach, CA, pp. 75–86 (1996)
23. Policriti, A., Tomescu, A.I., Vezzi, F.: A Randomized Numerical Aligner (rNA) (2010), http://sole.dimi.uniud.it/~alexandru.tomescu/files/rNA-ext.pdf
24. Salmela, L., Tarhio, J., Kalsi, P.: Approximate Boyer-Moore string matching for small alphabets. Algorithmica (to appear)
25. Ukkonen, E.: Approximate string matching over suffix trees. In: Proc. 4th Ann. Symp. on Combinatorial Pattern Matching, pp. 228–242 (1993)
26. Zimmermann, R.: Efficient VLSI Implementation of Modulo ($2^n \pm 1$) Addition and Multiplication. In: IEEE Symposium on Computer Arithmetic, pp. 158–167. IEEE Computer Society, Los Alamitos (1999)

Language-Based Comparison of Petri Nets with Black Tokens, Pure Names and Ordered Data

Fernando Rosa-Velardo[1,⋆] and Giorgio Delzanno[2]

[1] Universidad Complutense de Madrid, Spain
fernandorosa@sip.ucm.es
[2] Università di Genova, Italy
giorgio@disi.unige.it

Abstract. We apply language theory to compare the expressive power of models that extend Petri nets with features like colored tokens and/or whole place operations. Specifically, we consider extensions of Petri nets with transfer and reset operations defined for black indistinguishable tokens (Affine Well-Structured Nets), extensions in which tokens carry pure names dynamically generated with special ν-transitions (ν-APN), and extensions in which tokens carry data taken from a linearly ordered domain (Data nets and CMRS). These models are well-structured transitions systems. In order to compare these models we consider the families of languages they recognize, using coverability as accepting condition. With this criterion, we prove that ν-APNs are in between AWNs and Data Nets/CMRS. Moreover, we prove that the family of languages recognized by ν-APNs satisfies a good number of closure properties, being a semi-full AFL. These results extend the currently known classification of the expressive power of well-structured transition systems with new closure properties and new relations between extensions of Petri nets.

1 Introduction

Dynamic name generation has been thoroughly studied in the last decade, mainly in the field of security and mobility [9]. Paradigmatic examples of nominal calculi are the π-calculus and the Ambient Calculus [9]. In previous works we have studied a very simple extension of Petri Nets, that we called ν-APNs [14,13]. Tokens in ν-APNs are pure names, that can be created fresh, moved along the net and used to restrict the firing of transitions with name matching. Since any fresh name can be created, we identify markings up to renaming of their names.

The paper [10] proves that reachability is undecidable for ν-APNs. However, ν-APNs belong to the class of (strictly) Well Structured Transition Systems (WSTS) [14]. This means that the problems of boundedness (whether the set of reachable states is finite) and coverability (whether a marking which is *greater* than a given one is reachable) are both decidable.

⋆ Author partially supported by the Spanish projects DESAFIOS10 TIN2009-14599-C03-01, WEST TIN2006-15578-C02-01 and PROMETIDOS S2009/TIC-1465.

In this paper we compare ν-APNs with other models that are also WSTS. Among these models, we highlight *Affine Well-structured Nets* (AWN) [5], a well-structured extension of Petri nets in which whole-place operations (as transfers and resets) are allowed; Data nets [11], an extension of AWNs in which tokens are no longer indistinguishable, but taken from a linearly ordered domain; and CMRS [1], a fragment of Data nets without whole-place operations. All above mentioned models are well-structured transition systems in which the reachability problem is undecidable.

To compare the expressive power of different models, it comes natural to study the class of languages generated by associating labels to transitions: a finite firing sequence defines a word. The standard notion of acceptance is based on reachability of a configuration. However, since reachability is undecidable for these models, the class of languages they recognize is the class of recursively enumerable languages. Therefore, we need finer grain criteria to distinguish Petri net extended with whole place operations and colored tokens. More specifically, we consider well-structured languages, in which the acceptance condition is defined using coverability of a given configuration.

In [3] such comparison is done for Petri nets (PN), AWNs, and Data Nets, and the following is proved:

$$L(\text{PN}) \subset L(\text{AWN}) \subset L(\text{Data nets})$$

Moreover, the authors proved that Data nets are equivalent (they generate the same family of languages) to the so called Petri Data nets, Data nets for which no whole-place operation is allowed, and equivalent to CMRS. We plan to put ν-APNs in that picture, by studying the family of languages recognized by them. More precisely, we study the families recognized by ν-APNs and also by $\nu_{\neq}$-APNs, a variation of ν-APNs allowing to check for inequality of names. In particular, we prove that both are in between AWNs and Data nets.

The rest of the paper is organized as follows: Section 2 defines some basic concepts that we use throughout the paper. In Section 3 we study the languages recognized by ν-APNs and $\nu_{\neq}$-APNs with different accepting conditions, and prove some closure properties for them. Section 4 compares the languages recognized by AWNs and ν-APNs. Section 5 compares $\nu_{\neq}$-APNs and Data Nets. Finally, Section 6 presents our conclusions and future work.

2 Preliminaries

Languages, AFLs. Given a (finite) alphabet Σ, any $w = a_1 \cdots a_n$ with $n \geq 0$ and $a_i \in \Sigma$, for all i, is a (finite) word on Σ. We denote by Σ^* the set of words on Σ. If $n = 0$ then w is the empty word, which is denoted by ϵ. The length of w is $|w| = n$. A language on Σ is a set of words on Σ. If we denote by $\cdot$ the word concatenation, then $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1,\ w_2 \in L_2\}$ is the concatenation of L_1 and L_2. If we denote by L^i the language $L \overset{i}{\cdots} L$, the iteration L^+ of the language L is $\bigcup_{i>0} L^i$. A function $h : \Sigma^* \to \Sigma^*$ is an homomorphism if $h(w_1 \cdot w_2) = h(w_1) \cdot h(w_2)$. Given an homomorphism h and a language L, we can define $h(L) = \{h(w) \mid w \in L\}$ and $h^{-1}(L) = \{w \mid h(w) \in L\}$.

A *semi-full abstract family of languages* (semi-full AFL) [8] is a family of languages closed under union, intersection with regular languages, homomorphism and inverse homomorphism. A semi-full AFL is a *full AFL* if it is closed under concatenation and iteration.

wqos. A *quasi order* $\leq$ is a reflexive and transitive binary relation on a set X. A quasi order $\leq$ is well founded if there are no infinite strictly decreasing sequences, and it is decidable if for every $a, b \in X$ we can effectively decide if $a \leq b$. A well founded quasi order is simply said well (*wqo*) [6], if for every infinite sequence $a_0, a_1, \ldots$ there are i and j with $i < j$ such that $a_i \leq a_j$. Equivalently, an order is a wqo if every sequence has an increasing subsequence.

WSTS, WSL. A transition system is a pair $N = (X, \rightarrow)$ with set of states X and transition relation $\rightarrow \subseteq X \times X$. We denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. A *Well Structured Transition System* (WSTS) is a tuple $N = (X, \rightarrow, \leq)$, where $(X, \rightarrow)$ is a transition system, and $(X, \leq)$ is a decidable wqo, satisfying the following monotonicity condition[1]: $M_1 \leq M_2$ and $M_1 \rightarrow M_1'$ implies the existence of M_2' such that $M_2 \rightarrow M_2'$ and $M_1' \leq M_2'$.

In the classic theory of Petri net languages [12] three types of labelling functions are considered: injective, ϵ-free and arbitrary. In this work we concentrate on arbitrary labelling functions (generally having better closure properties). We consider a fixed finite alphabet Σ and a special symbol $\epsilon \notin \Sigma$, and we assume that ϵ is such that $\epsilon \cdot w = w \cdot \epsilon = w$. A labelled WSTS S is a WSTS $S = (X, \rightarrow, \leq)$, where the transition relation is partitioned into $\rightarrow = \bigcup_{a \in \Sigma \cup \{\epsilon\}} \xrightarrow{a}$. For a word $w \in \Sigma^*$, we write $M \xrightarrow{w} M'$ if M' can be reached from M and the concatenation of the labels of the transitions used is the word w. Moreover, four acceptance conditions can be considered: reachability, coverability, deadlock and no condition.

Definition 1. *Given a labelled WSTS S and two states s_0 and s_f, we define:*[2]

- $L^L(S) = \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s_f\}$,
- $L^G(S) = \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s,\ s \geq s_f\}$,
- $L^T(S) = \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s,\ s \not\rightarrow\}$,
- $L^P(S) = \{w \in \Sigma^* \mid s_0 \xrightarrow{w} s\}$,

Notice that conditions T and P do not make use of the final state s_f. For any of the models $\mathbf{M}$ we consider in this paper, we denote by $L^{\mathcal{R}}(\mathbf{M})$ the class of languages $\{L^{\mathcal{R}}(S) \mid S \in \mathbf{M}\}$, with $\mathcal{R} \in \{L, G, T, P\}$. A Well Structured Language (WSL) is any language accepted by a WSTS, with G as accepting condition [7]. In [7] the following *pumping lemma* is proved.

Lemma 1 (Lemma 6 (pg. 262) [7]). *Let L be a WSL and $(w_k)_{k=1}^{\infty} \subseteq L$ with $w_k = B_k \cdot E_k$ for every $k \geq 1$. Then, there exist $i < j$ such that $B_j \cdot E_i \in L$.*

[1] Less restrictive monotonicy notions are considered in [6].

[2] We use the classical notation for Petri Net Languages in [12].

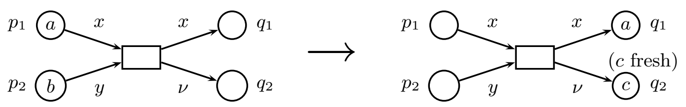


Fig. 1. A simple ν-APN and the firing of its only transition

Multisets. Given an arbitrary set A, we denote by $\mathcal{MS}(A)$ the set of finite multisets of A, that is, the mappings $m : A \to \mathbb{N}$. When needed, we identify each set with the multiset defined by its characteristic function, and use set notation for multisets when convenient. We denote by $S(m)$ the support of m, that is, the set $\{a \in A \mid m(a) > 0\}$ and by $|m| = \sum\limits_{a \in S(m)} m(a)$ the cardinality of m. We denote by $m_1 + m_2$, $m_1 \subseteq m_2$ and $m_1 - m_2$ the multiset addition, inclusion, and substraction, respectively. If $f : A \to B$ is an injection and $m \in \mathcal{MS}(A)$ then we can define $f(m) \in \mathcal{MS}(B)$ by $f(m)(b) = m(a)$ if $f(a) = b$ for some a, and $f(m)(b) = 0$, otherwise.

3 Nets in Which Tokens Carry Pure Names

In this section we study the class of languages generated by an extension of Petri nets with pure names, called ν-APN [13].

***ν*-APNs.** The class of ν-APNs is an extension of Petri Nets in which tokens are not indistinguishable, but pure names, that can only be compared by the equality predicate. We consider a set *Id* of names, a set *Var* of variables and a disjoint set Υ of special variables.

A ν-APN is a tuple $N = (P, T, F)$, where P and T are finite disjoint sets of elements called places and transitions, respectively,

$$F : (P \times T) \cup (T \times P) \to \mathcal{MS}(\mathit{Var})$$

is such that for every $t \in T$, $\mathit{post}(t) \setminus \Upsilon \subseteq \mathit{pre}(t)$ and $\mathit{pre}(t) \cap \Upsilon = \emptyset$, where

– $\mathit{pre}(t) = \bigcup\limits_{p \in P} S(F(p, t))$,
– $\mathit{post}(t) = \bigcup\limits_{p \in P} S(F(t, p))$ and
– $\mathit{Var}(t) = \mathit{pre}(t) \cup \mathit{post}(t)$.

The mapping F labels every pair (p, t) and (t, p) with a multiset of variables. These variables specify how tokens flow from preconditions to postconditions. Only variables in ν can appear in some postarc without appearing in some prearc. Variables in Υ can only be instantiated to names that do not occur in the current marking, so that they formalize fresh name creation. We are assuming that these variables only appear in post-arcs, that is, labelling pairs of the form (t, p). A marking M of a ν-APN assigns to each place a multiset of names. We denote by $S(M)$ the set of names that occur in some place according to marking M, that is, $S(M) = \bigcup_{p \in P} S(M(p))$. A transitions t can be fired with respect to a mode $\sigma : \mathit{Var}(t) \to \mathit{Id}$

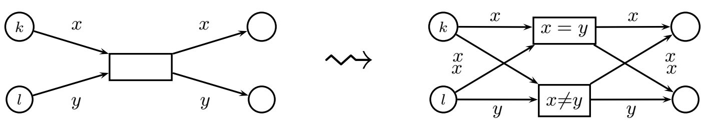


Fig. 2. Simulation of ν-APN (left) by means of a $\nu_{\neq}$-APN (right)

that instantiates each variable to an identifier so that $\sigma(\nu_1) \neq \sigma(\nu_2)$ for each different $\nu_1, \nu_2 \in \Upsilon$. We use $\sigma, \sigma', \sigma_1 \ldots$ to range over modes. A transition t is enabled with mode σ for a marking M if for all $p \in P$, $\sigma(F(p,t)) \subseteq M(p)$ and $\sigma(\nu) \notin S(M)$ for all $\nu \in \Upsilon$. The reached state after the firing of t with mode σ is the marking $M'(p) = (M(p) - \sigma(F(p,t))) + \sigma(F(t,p))$ for all $p \in P$.

We write $M \xrightarrow{t(\sigma)} M'$, $M \rightarrow M'$ and $M \xrightarrow{\tau} M'$ with $\tau = t_1(\sigma_1) \cdots t_n(\sigma_n)$, saying that τ is a transition sequence, with their obvious meanings.

Figure 1 depicts a simple ν-APN with a single transition. When fired, it moves one token from p_1 to q_1 (because of variable x labelling both arcs), removes a token from p_2 (variable y does not appear in any outgoing arc) and a new name is created in q_2 (because of variable ν). In this example, had the token in p_2 carried an a instead of a b, the transition could also have been fired (reaching the same marking), since modes can instantiate different variables with the same name. In other words, in ν-APNs we cannot check for inequality. We consider a variation of ν-APNs, that we call $\nu_{\neq}$-APNs, in which we can check for inequality, which can be simply formalized by taking modes to be injections.

We define $M_1 \sqsubseteq M_2$ if there is an injection $\iota : S(M_1) \rightarrow S(M_2)$ such that $\iota(M_1(p)) \subseteq M_2(p)$, for all $p \in P$. We take $\equiv$ as $\sqsubseteq \cap \sqsupseteq$ and identify markings up to $\equiv$. The relation $\sqsubseteq$ is a wqo and the transition system generated by ν-APNs and $\nu_{\neq}$-APNs are WSTS with that order [14].

ν-APN Languages. In this section we study $L^{\mathcal{R}}(\nu\text{-APN})$ and $L^{\mathcal{R}}(\nu_{\neq}\text{-APN})$ for $\mathcal{R} \in \{L, G, T, P\}$, that is, the families of languages recognized by ν-APNs and $\nu_{\neq}$-APNs, with reachability, coverability, termination and no-condition, as accepting conditions. We can immediately obtain the following basic results.

Proposition 1. *$L^L(\nu\text{-}APN)$, $L^L(\nu_{\neq}\text{-}APN)$ and $L^T(\nu_{\neq}\text{-}APN)$ are the class of recursively enumerable languages.*

Proposition 2. *The following relations among languages hold:*

1. $L^P(\nu\text{-}APN) \subset L^G(\nu\text{-}APN) \subset L^L(\nu\text{-}APN)$
2. $L^P(\nu_{\neq}\text{-}APN) \subset L^G(\nu_{\neq}\text{-}APN) \subset L^L(\nu_{\neq}\text{-}APN)$

Proof. The proof is the same for ν-APNs and $\nu_{\neq}$-APNs. For the first inclusions it is enough to consider the empty marking as acceptance. To see that they are strict, notice that languages in L^P are always prefix-closed, and it is trivial to devise non prefix-closed languages in $L^G(\nu\text{-APN})$ and $L^G(\nu_{\neq}\text{-APN})$. The second inclusions follow from the previous proposition. Moreover, they are strict because there are recursively enumerable languages that are not WSL, such as $\{a^n b^n \mid n > 0\}$, which can be easily seen using the pumping lemma for WSL.

In general, for all the languages considered, being able to check inequality gives us at least the same expressive power:

Proposition 3. *$L^{\mathcal{R}}(\nu\text{-}APN) \subseteq L^{\mathcal{R}}(\nu_{\neq}\text{-}APN)$ for $\mathcal{R} \in \{L, G, T, P\}$*

Proof. We have to simulate a ν-APN by means of a $\nu_{\neq}$-APN. For a transition t and a partition $\mathcal{X} = X_1 \sqcup \cdots \sqcup X_k$ of $Var(t)$, we choose k variables $x_1, \ldots, x_k$ so that $x_i \in X_i$. Then, for each t and each partition of $Var(t)$, we consider a transition $t_{\mathcal{X}}$ (with the same label as t) and an arc from p to $t_{\mathcal{X}}$ labelled with x_i iff $F(p,t) \in X_i$, and analogously for arcs (t,p) (see Fig. 2).

The families of languages $L^G(\nu\text{-APN})$ and $L^G(\nu_{\neq}\text{-APN})$ satisfy a good number of closure properties, which are summarized in the following result.

Proposition 4. *$L^G(\nu\text{-}APN)$ and $L^G(\nu_{\neq}\text{-}APN)$ are semi-full AFLs closed under concatenation and intersection.*

Therefore, the families of languages recognized by ν-APNs and $\nu_{\neq}$-APNs, with coverability as accepting condition, are semi-full AFLs, but we do not know if they are also full AFLs, since we have not proved whether they are closed under iteration. However, we can prove the following.

Proposition 5. *If $L \in L^G(Petri\ nets)$ then $L^+ \in L^G(\nu\text{-}APN)$.*

Proof. We represent each of the executions of the Petri net by a different identifier. Then we add a place that contains the identifier that represents the current execution, and a transition that can be fired when the final marking is covered (matching the current identifier) and creating the initial marking with a fresh identifier. The final marking is that with the same tokens as indicated by the final marking of the Petri net, which in turn must be the same token in the new place.

Thanks to this result it is straightforward to see that, for instance, the language $\{a^{n_1}b^{m_1} \ldots a^{n_k}b^{m_k} \mid n_i \geq m_i \ for \ i : 1, \ldots, k\}$ is in $L^G(\nu\text{-APN})$. It would be interesting to see what happens with iteration for $L^G(\nu\text{-APN})$. We conjecture that for an arbitrary $L \in L^G(\nu\text{-APN})$, L^+ is not necessarily in $L^G(\nu\text{-APN})$. The intuitive reasoning is the same as for $L^G(\text{Petri nets})$, namely the fact that by means of coverability we cannot distinguish between different "executions" within the same net (we cannot throw away arbitrary garbage).

To conclude this section, let us see that if we forbid name matching in ν-APNs, then its expressive power boils down to that of Petri nets, since we are not considering whole-place operations. We simply call $\nu_{=}$-APNs the subclass of ν-APNs where for each transition t, variables in pre-arcs appear at most once, that is, such that $\sum_{p \in P} F(p,t)(x) \leq 1$.

The intuitive idea is that, without matching, the specific nature of named tokens, that is, the identifiers carried by tokens, does not play any role in the firing of transitions. Therefore, we could flatten the given $\nu_{=}$-APN to the Petri Net with the same places, transitions and flow relation (by removing variables in arcs). This would be enough if we were considering T or P as terminating

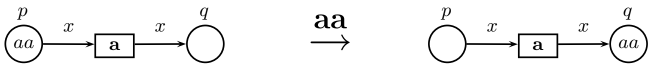


Fig. 3. $\nu_=$-APN with final marking $M(p) = \emptyset$ and $M(q) = \{a, b\}$

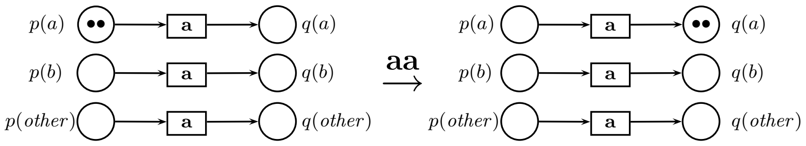


Fig. 4. Simulation of the $\nu_=$-APN in Fig. 3 by means of a Petri Net net with final marking $M^*(q(a)) = M^*(q(b)) = 1$ and $M^*(p(a)) = M^*(p(b)) = M^*(p(other)) = M^*(q(other)) = 0$

conditions, but this is not the case for G. To see it, it is enough to consider the net depicted in Fig. 3, using $M(p) = \emptyset$ and $M(q) = \{a, b\}$ as final marking. That net can fire its only transition twice, reaching a marking with the identifier a twice in place q, which does not cover M. Therefore, it generates the empty language, though the sketched construction would generate the language $\{\mathbf{aa}\}$. In other words, the terminating condition does allow us to retrieve some information about the involved tokens, even though that information was not relevant in the enabling and firing of transitions. However, that information is finite (about tokens in the initial and the final marking), so that we can control it with some special places (see Fig. 4).

Proposition 6. *$L^{\mathcal{R}}$(Petri Nets) $= L^{\mathcal{R}}$($\nu_=$-APN) for $\mathcal{R} \in \{L, G, T, P\}$.*

4 Pure Names vs. Black Tokens

In this section we compare ν-APNs with AWNs, a well structured extension of Petri Nets that allows whole-place operations An AWN N is given by a set of n places and a set of transitions. Each transition comes equipped with two n-vectors, F_t and H_t, and a $n \times n$-matrix G_t over $\mathbb{N}$. A marking M of an AWN must specify how many (black) tokens are there in each place, so that it is also an n-vector. Then, a transition t can fire whenever $F_t \leq M$, and the reached marking after the firing is $M' = (M - F_t) \cdot G_t + H_t$. The matrices G_t are responsible for the whole place operations. For instance, if the i-th column of G_t is null, then G_t resets the i-th place, that is, it empties its content. If G_t is the identity matrix for all t, then N is an ordinary Petri net.

Now let us see that we can simulate the whole place operations allowed by AWNs thanks to the name creation mechanism in ν-APNs. We obtain lossy simulations of the AWNs, in which some whole-place operations can loose some tokens. However, we know [3] that a lossy version of any WSTS produces the same language, that is, that if N' is a lossy version of N then $L^G(N) = L^G(N')$.

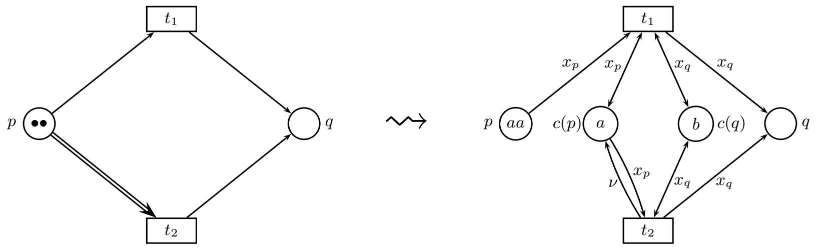


Fig. 5. Simulation of Reset Nets by means of ν-APNs

Proposition 7. *$L^G(AWN) \subseteq L^G(\nu\text{-}APN)$.*

Proof (sketch). The idea is to have for each place p another place $c(p)$ which at all times contains a single identifier token, which is the current valid token of p. All transitions use only valid tokens (matching their values with the values in the places $c(p)$). Transitions that reset a place p cause the replacement of the current valid token, by means of the ν variable (see Fig. 5). This has the effect of leaving some garbage tokens in the place that should have been reset, but these tokens cannot interfere with the execution of the net because of the previous comments. Notice that the simulation of resets is not lossy.

A transfer from p to q is simulated as follows:

1. For each token in p matching the one in $c(p)$, remove it, and add a token in q matching the one in $c(q)$,
2. Replace the token in $c(p)$ by a fresh token.

Notice that the second step can be performed even if the first one can still be done, that is, when there are still tokens in p matching the one in $c(p)$. In that case, the simulation is missing tokens. Moreover, the previous simulation must be done in a transactional way, so that after step 1 has been followed once, no other transition can fire until step 2 has happened, which can be achieved thanks to some new "control places".

5 Pure Names vs. Ordered Data

Now we compare ν-APNs with two extensions of Petri nets in which tokens carry data taken from an ordered domain, namely Data nets [11] and CMRS [1].

Data Nets. Data nets [11] are an extension of AWN in which tokens are colored with data taken from an infinite domain D equipped with a linear and dense ordering $\prec$. A data net consists of a finite set of places P and of a finite set of transitions. A data net marking s is a multiset of tokens that carry data in D. Formally, a marking s is a finite sequence of vectors in $\mathbb{N}^P \setminus \{\mathbf{0}\}$, where $\mathbf{0}$ is the vector that contains only 0's. Each index i in the sequence s corresponds to some $d_i \in D$ such that $i \leq j$ if and only if $d_i \prec d_j$. For each $p \in P$, $s(i)(p)$ is the number of tokens with data d_i in place p.

First of all, a data net transition t has an associated arity α_t (a natural number greater than zero). The arity $\alpha_t = k$ is used to non-deterministically select k distinct data $d_1 \prec \ldots \prec d_k$ from the current configuration s. Some of the selected data may not occur in s (they are fresh). This choice induces a finite and ordered partitioning $R(\alpha_t)$ of the data in s. A transition t operates on the regions in the partitioning $R(\alpha_t)$ in three steps defined resp. by three matrices $F_t, H_t \in \mathbb{N}^{R(\alpha_t)\times P}$, and $G_t \in \mathbb{N}^{R(\alpha_t)\times P\times R(\alpha_t)\times P}$. As in AWN, F_t is responsible for the removal of tokens, G_t performs whole-place operations and H_t is responsible for the addition of tokens. The tokens involved in the firing are not just those carrying the k selected data, but potentially every token present in the marking, though those belonging to the same region are treated uniformly.

As proved in [11], data nets are well-structured, so that their languages are in the class WSL. Though they can perform very general whole-place operations, from the point of view of the languages they accept, whole-place operations do not make any difference [3]. For that reason, we present CMRS, a more manageable formalism equivalent to Data nets from the language point of view.

CMRS. We assume a set $\mathbb{V}$ of variables which range over $\mathbb{N}$, and a set $\mathbb{P}$ of unary predicate symbols. In CMRS we write multisets as lists, so $[1, 5, 5, 1, 1]$ represents a multiset with three occurrences of 1 and two occurrences of 5; $[\,]$ represents the empty multiset. We use the relations and operations such as $\subset$ (inclusion), $+$ (union), and $-$ (difference) on multisets. For a set $V \subseteq \mathbb{V}$, a *valuation* Val of V is a mapping from V to $\mathbb{N}$. A *condition* is a finite conjunction of *gap order* formulas of the forms: $x <_c y$, $x \leq y$, $x = y$, $x < c$, $x > c$, $x = c$, where $x, y \in \mathbb{V}$ and $c \in \mathbb{N}$. Here $x <_c y$ stands for $x + c < y$. We often use $x < y$ instead of $x <_0 y$. Sometimes, we treat a condition ψ as a set, and write e.g. $(x <_c y) \in \psi$ to indicate that $x <_c y$ is one of the conjuncts in ψ. We use *true* to indicate an empty set of conditions. A *term* is of the form $p(x)$ where $p \in \mathbb{P}$ and $x \in \mathbb{V}$. A *ground term* is of the form $p(c)$ where $p \in \mathbb{P}$ and $c \in \mathbb{N}$. We sometimes say that a predicate symbol is *nullary* to mean that its parameter is not relevant.

A *constrained multiset rewriting system (CMRS)* $\mathcal{S}$ consists of a finite set of *rules* each of the form $L \rightsquigarrow R : \psi$, where L and R are multisets of terms, and ψ is a condition. We assume that ψ is consistent (otherwise, the rule is never enabled). For a valuation Val, we use $\mathit{Val}(\psi)$ to denote the result of substituting each variable x in ψ by $\mathit{Val}(x)$. We use $\mathit{Val} \models \psi$ to denote that $\mathit{Val}(\psi)$ evaluates to *true*. For a multiset T of terms we define $\mathit{Val}(T)$ as the multiset of ground terms obtained from T by replacing each variable x by $\mathit{Val}(x)$. A *configuration* is a multiset of ground terms. Each rule $\rho = L \rightsquigarrow R : \psi \in \mathcal{S}$ defines a relation between configurations. More precisely, $\gamma \xrightarrow{\rho} \gamma'$ if and only if there is a valuation Val s.t. the following conditions are satisfied: (i) $\mathit{Val} \models \psi$, (ii) $\gamma \geq \mathit{Val}(L)$, and (iii) $\gamma' = \gamma - \mathit{Val}(L) + \mathit{Val}(R)$.

In [3] it is proved that $L^G(\mathit{Data\ nets}) = L^G(\mathit{CMRS})$ (i.e. they have the same power w.r.t. well structured languages). The following property then follows

Proposition 8. $L^G(\nu_{\neq}\text{-}\mathit{APN}) \subseteq L^G(\mathit{CMRS}/\mathit{Datanets})$

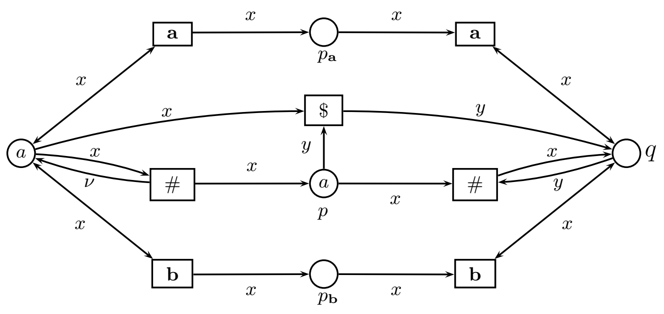


Fig. 6. ν-APN that recognizes L_Σ with $\Sigma = \{a, b\}$

Proof. We have to simulate a $\nu_{\neq}$-APNs N by means of a CMRS N^*. Wlog, for the sake of readability, we assume that each transition can only create at most one name, by means of a variable $\nu \in \Upsilon$. We add a special predicate a to identify the future new identifier. For each t we have the rule

$$\sum_{F(p,t)=x} [p(x)] + [a(\nu)] \to \sum_{F(t,p)=x} [p(x)] + [a(\nu')] : \nu' > \nu$$

Thus, at each time, the new *place* a contains an upper bound of all the names that appear in the current marking. Whenever a fresh name is created in N, the simulating CMRS uses the name in a instead (which is known to be different from any other name), and replaces it by a greater one, so that it is still an upper bound. Notice that in N^* we are recording the order in which the different identifiers have been created, though we do not record such order in N. Therefore, in the final marking we have to say the order in which the different identifiers have been created (this is forced by the CMRS syntax) but any such order is good, so that we must consider all of those orders. For each m_i that represents the final marking in some order we add a rule r_i (labelled by ϵ) that converts m_i, into the final marking, that can consists only of a new predicate *final*.

Corollary 1. $L^G(\mathit{AWN}) \subseteq L^G(\nu\text{-}\mathit{APN}) \subseteq L^G(\nu_{\neq}\text{-}\mathit{APN}) \subseteq L^G(\mathit{Data\ Nets})$

Since $L^G(\text{AWN}) \subseteq L^G(\text{Data Nets})$, we know that at least one of the inclusions in the previous result is strict. This is the case for the first one.

Proposition 9. $L^G(\mathit{AWN}) \subset L^G(\nu\text{-}\mathit{APN})$.

Proof. The language $L_\Sigma = \{w_1\# \ldots \#w_n\$v_1\# \ldots \#v_k \mid w_i, v_i \in \Sigma^*,$ where $h : \{1, \ldots, k\} \to \{1, \ldots, n\}$ is an injection s.t. v_i included in $w_{h(i)}$ (as multisets)$\}$ belongs to $L^G(\nu\text{-APN})$, assuming $\#, \$ \notin \Sigma$. For instance, in Fig. 6 we show the net that recognize words on the alphabet $\{a, b\}$, when it has final marking M_f with a token in q and empty elsewhere. Each name is used to represent one of the words w_i. If a name a represents w, then the number of tokens carrying a in $p_{\mathbf{a}}$ equals the number of $\mathbf{a}$'s in w (analogously for $\mathbf{b}$). Everytime a new w is started (by firing the transition labelled by $\#$), a fresh name is used. The place p

contains the set of names used along the computation. This information is used in the second phase (after the firing of the transition labelled by &).

Now let $\Sigma_0 = \{a, b, 0, 1\}$ and $L_0 = L_{\Sigma_0}$. We prove that there is no AWN that recognizes L_0. The proof is per absurdum. Suppose there exists a AWN N that recognizes L_0 with initial marking M_{init} and accepting marking M_f. Assume that M has places $p_1, \dots, p_n$.

Let $S = \{w_1\# \dots \# w_k \mid w_i \in \{a, b, 0, 1\}^*,\ i : 1, \dots, k,\ k \geq 0\}$. Furthermore, let $v_1\# \dots \# v_n \leq^* w_1\# \dots \# w_m$ if there exists $h : \overline{n} \to \overline{m}$ s.t. $v_i \subseteq w_{h(i)}$ for $i : 1, \dots, n$. We first notice that, for any $s \in S$, $s\$s \in L_0$. Under our hypothesis, we have then that, for each $s \in S$, there is a marking M_s such that $M_{init} \overset{s\$}{\Longrightarrow} M_s \overset{s}{\Longrightarrow} M$ and $M_f \subseteq M$. Consider the sequences $s_0, s_1, s_2, \dots$ and $M_{s_0}, M_{s_1}, M_{s_2}, \dots$ of words and markings defined as follows:

- $s_0 := b\#b \dots \#b$ such that s_0 has n occurrences of b;
- If $M_{s_i} = (m_1, \dots, m_n)$ then $s_{i+1} := a^{m_1}p_1\#\, a^{m_2}p_2\# \cdots \#\, a^{m_n}p_n$, for $i = 0, 1, \dots$, where $p_1, \dots, p_n$ are unary encodings of the positions $1, \dots, n$ over n bits, i.e., $p_1 = 10\dots0$, $p_2 = 110\dots0$, $\dots$, $p_n = 111\dots1$.

Since b occurs only in s_0 $s_0 \not\leq^* s_i$ for all $i > 0$. Furthermore, for any $i < j$, $M_{s_i} \subseteq M_{s_j}$ iff $s_{i+1} \leq^* s_{j+1}$. This holds because s_{i+1} and s_{j+1} have both $n-1$ occurrences of the separator $\#$ and because any injection needed in the definition of $\leq^*$ is forced to preserve positions (their unary representation) in our encoding of markings. Since marking inclusion is a well-quasi ordering, there exist i, j such that $i < j$ and $M_{s_i} \subseteq M_{s_j}$. Now let j be the smallest natural number satisfying this property. Then, we have that $M_{s_{i-1}} \not\subseteq M_{s_{j-1}}$ and $s_i \not\leq^* s_j$ for $i > 0$. Furthermore, since by definition $s_0 \not\leq^* s_j$, we have that $s_i \not\leq^* s_j$ for any $i \geq 0$. Since $M_{s_i} \subseteq M_{s_j}$, by monotonicity of AWNs, we have that $M_{s_i} \overset{s_i}{\Longrightarrow} M$ with $M_f \subseteq M$ implies that $M_{s_j} \overset{s_i}{\Longrightarrow} M'$ with $M_f \subseteq M \subseteq M'$. Hence, we obtain $M_{init} \overset{s_j\$s_i}{\Longrightarrow} M'$ and $s_j\$s_i \in L^G(N)$ which is in contradiction with the hypothesis that $L^G(N) = L_0$. □

Furthermore, we conjecture that ν-APN are strictly less expressive than CMRS and Data nets. Strict inclusion seems closely related to the $+$-closure of their corresponding languages. Indeed, while we conjecture that ν-APN are not closed under $+$ (we only know that the subclass of Petri net languages is not closed under $+$ [7]) the following property holds.

Proposition 10. *If $L \in L^G(CMRS)$, then $L^+ \in L^G(CMRS)$.*

6 Conclusions and Open Problems

The study of the expressive power of computation models in between Petri nets and Turing machines, and in particular of the class of well-structured transition systems, is a challenging research problem with several open questions. In this paper we have extended the classification of well-structured transition systems studied in [11,2,3] by comparing infinite-state models like *Affine Well-structured Nets*

(AWN) [5], Data nets [11], and CMRS [1] with ν-APN, an extension of Petri nets in which tokens are pure names [13]. We extend the results on [3] obtaining

$$L(\text{PN}) = L(\nu_{=}\text{-APN}) \subset L(\text{AWN}) \subset L(\nu\text{-APN}) \subseteq L(\nu_{\neq}\text{-APN}) \subseteq L(\text{Data nets})$$

In [3] the authors prove that considering whole-place operations when data is ordered does not have any effect, ie, $L(\text{Data nets}) = L(\text{Petri Data Nets})$. Though we do not show it by lack of space, this is also true in the case of pure names.

Concerning open problems, we conjecture that ν-APNs are strictly less expressive than CMRS and Data nets and that ν-APN and lossy FIFO channel systems [4] define incomparable classes of L^G-languages. For the latter conjecture, we know how to prove that ν-APN languages are not included in the languages of Lossy FIFO channel systems. However, the proof of the other direction remains to be proved.

References

1. Abdulla, P., Delzanno, G.: On the coverability problem for constrained multiset rewriting. In: AVIS, an ETAPS workshop (2006)
2. Abdulla, P.A., Delzanno, G., Begin, L.V.: Comparing the expressive power of well-structured transition systems. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 99–114. Springer, Heidelberg (2007)
3. Abdulla, P.A., Delzanno, G., Begin, L.V.: A language-based comparison of extensions of petri nets with and without whole-place operations. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 71–82. Springer, Heidelberg (2009)
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. Finkel, A., McKenzie, P., Picaronny, C.: A well-structured framework for analysing petri net extensions. Inf. Comput. 195(1-2), 1–29 (2004)
6. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
7. Geeraerts, G., Raskin, J.F., Begin, L.V.: Well-structured languages. Acta Inf. 44(3-4), 249–288 (2007)
8. Ginsburg, S.: Algebraic and Automata-Theoretic Properties of Formal Languages. Elsevier Science Inc., New York (1975)
9. Gordon, A.D.: Notes on nominal calculi for security and mobility. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, pp. 262–330. Springer, Heidelberg (2001)
10. Kummer, O.: Undecidability in object-oriented petri nets. Petri Net Newsletter 59, 18–23 (2000)
11. Lazic, R., Newcomb, T., Ouaknine, J., Roscoe, A.W., Worrell, J.: Nets with tokens which carry data. Fundam. Inform. 88(3), 251–274 (2008)
12. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs (1981)
13. Rosa-Velardo, F., de Frutos-Escrig, D.: Name creation vs. replication in petri net systems. Fundam. Inform. 88(3), 329–356 (2008)
14. Rosa-Velardo, F., de Frutos-Escrig, D., Alonso, O.M.: On the expressiveness of mobile synchronizing petri nets. Electr. Notes Theor. Comput. Sci. 180(1), 77–94 (2007)

Verifying Complex Continuous Real-Time Systems with Coinductive CLP(R)

Neda Saeedloei and Gopal Gupta

Department of Computer Science,
University of Texas at Dallas,
Richardson, TX 75080
{nxs048000,gupta}@utdallas.edu

Abstract. Timed automata have been used as a powerful formalism for specifying, designing, and analyzing real-time systems. We consider the generalization of timed automata to Pushdown Timed Automata (PTA). We show how PTA can be elegantly modeled via logic programming extended with *co-induction* and *constraints over reals*. We propose a general framework based on constraint logic programming and co-induction for modeling/verifying real-time systems. We use this framework to develop an elegant solution to the *generalized railroad crossing problem* of Lynch and Heitmeyer. Interesting properties of the system can be verified merely by posing appropriate queries.

1 Introduction

Design, specification, implementation and verification of real-time systems is an important area of research as real-time systems are ubiquitous. Timed automata is a popular approach to designing, specifying and verifying real-time systems [1,2]. Timed automata can also provide foundational basis for Cyber-Physical Systems (CPS) [11] that are currently receiving a lot of attention. Timed automata are ω-automata [14] extended with clocks. Transitions from one state to another are not only made on the alphabet symbols of the language but also on constraints imposed on clocks (e.g., at least 2 units of time must have elapsed). Timed automata are suitable for specifying a large class of real-time systems; however, they suffer from the same limitations that any automaton suffers, in that they can recognize only timed regular languages. This restriction to regular languages renders them unsuitable for many complex, useful applications where the language involved may not be regular. To overcome this problem, timed automata have been extended to pushdown timed automata [5]. A pushdown timed automaton recognizes a sequence of timed words, where a timed word is a symbol from the alphabet of the language the automaton accepts, paired with the time-stamp indicating the time that symbol was seen. The sequence of timed words in a string accepted by a pushdown timed automaton must obey the rules of syntax laid down by the underlying untimed pushdown automaton, while the time-stamps must obey the timing constraints imposed on the times at which the symbols appear.

The work in [7] showed how timed automata can be modeled via constraint logic programming over reals (or CLP(R)) and their properties verified. However, [7] does not model the ω-automata naturally. Later, [6] showed how co-induction can be introduced in logic programming (or LP) to naturally model ω-automata and verify their properties. In this paper we extend the efforts of [7] and [6] to show how co-induction and CLP(R) can also be used to elegantly model PTA. We propose a general framework based on constraint logic programming and co-induction for modeling/verifying real-time systems (including CPS). The formalism that are used in this framework are timed automata and PTA which can be computationally modeled by combination of coinductive logic programming (or Co-LP) and CLP(R). We show how a coinductive CLP(R) rendering of a PTA can be used to verify safety and liveness properties of a system. We illustrate the effectiveness of our approach by showing how the well-known *generalized railroad crossing (GRC) problem* [8] can be naturally modeled, and how its various safety and utility properties can be easily verified. Our approach based on coinductive CLP(R) for handling the GRC is considerably more elegant and simpler than other approaches proposed such as in [12].

The rest of the paper is organized as follows. We present an overview of timed automata. Next, we consider PTA and timed grammars and show how they can be modeled via coinductive CLP(R). Note that the formulation of PTA is our own, though they were first introduced in [5]. We illustrate our method of modeling and verifying PTA to model the *generalized railroad crossing (GRC) problem.* The elegant modeling and verification of timed automata and PTA, and their application to naturally modeling/verifying complex real-time systems and CPS is the main contribution of this paper. We assume that the reader is familiar with *Constraint Logic Programming* over reals *(CLP(R))* [9], as well as with *Coinductive Logic Programming (Co-LP)* [6,13].

2 Timed Automata

A *timed automaton* [2] is a tuple $M = \langle \Sigma, Q, Q_0, C, E, F \rangle$, where

- Σ is a finite alphabet;
- Q is the (*finite*) set of states;
- $Q_0 \subseteq Q$ is the set of initial states;
- C is a finite set of *clocks*;
- $E \subseteq Q \times Q_0 \times \Sigma \times 2^C \times \phi(C)$ gives the set of transitions. An edge $\langle q, q\prime, a, \lambda, \delta \rangle$ represents a transition from state q to state $q\prime$ on input symbol a. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and δ is a clock constraint over C;
- F is a subset of 2^Q.

A run r, denoted by $(\bar{q}, \bar{\nu})$, of a timed automaton over a timed word (σ, t) is an infinite sequence of the form

$$r : \langle q_0, \nu_0 \rangle \xrightarrow[t_1]{\sigma_1} \langle q_1, \nu_1 \rangle \xrightarrow[t_2]{\sigma_2} \langle q_2, \nu_2 \rangle \xrightarrow[t_3]{\sigma_3} \dots$$

with $q_i \in Q$ and $\nu_i \in [C \mapsto \mathbf{R}^+]$, for all $i \geq 0$, satisfying two requirements:

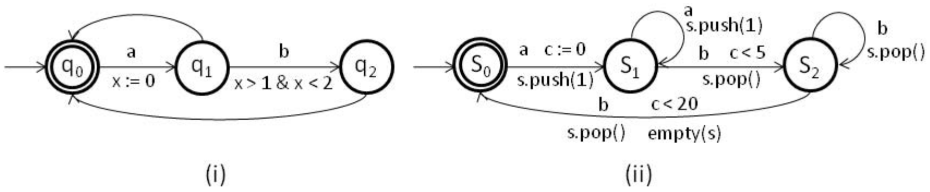


Fig. 1. (i) A Timed Automaton, (ii) A Pushdown Timed Automaton

– $q_0 \in Q_0$, and $\nu_0(x) = 0$ for all clocks $x \in C$.
– for all $i \geq 1$, there is an edge in E of the form $\langle q_{i-1}, q_i, \sigma_i, \lambda_i, \delta_i \rangle$ such that $(\nu_{i-1} + t_i - t_{i-1})$ satisfies δ_i and ν_i equals $[\lambda_i \mapsto 0](\nu_{i-1} + t_i - t_{i-1})$.

The set *inf(r)* consists of $q \in Q$ such that $q = q_i$ for infinitely many $i \geq 0$.

Different notions of acceptance have been proposed. A run $r = (\bar{q}, \bar{\nu})$ of a timed *Büchi* automaton over a timed word (σ, t) is called an *accepting run* iff F is a singleton, and $inf(r) \cap F \neq \emptyset$. A run $r = (\bar{q}, \bar{\nu})$ of a timed *Müller* automaton over a timed word (σ, t) is called an *accepting run* iff $inf(r) \in F$.

Figure 1 (i) shows a simple Büchi timed automaton, with clock x and final state $F = \{q_0\}$. It describes a system in which signals are recognized. Each signal a can (but need not) be followed by a b signal, with the constraint that the b signal must arrive at least one time unit and at most two time units after a.

3 Pushdown Timed Automata (PTA)

PTA extend timed automata with a stack in exactly the same manner that pushdown automata extend finite automata. Thus, a pushdown timed automaton is obtained from a timed automaton by adding:

– ϵ (empty string) to the input alphabet Σ;
– a stack alphabet $\Gamma_\epsilon = \Gamma \cup \epsilon$, where Γ is a set of symbols disjoint from Σ;
– a stack represented by Γ_ϵ^*;
– $PD : E \mapsto \Gamma \times \Gamma_\epsilon$ assigns a pair (a, γ) with $a \in \Gamma$ and $\gamma \in \Gamma_\epsilon$, called a *stack operation*, to each transition in E. A stack operation (a, γ) replaces the top symbol a of the stack with a string (possibly empty) in Γ_ϵ.

Acceptance conditions for an infinite string for PTA are similar to those for timed automata but, additionally, the stack must be empty in every final state.

Pushdown timed automata have been introduced earlier [5]. Our aim in this paper is to show how PTA can be modeled and their properties verified with coinductive CLP(R) with the same ease as that for timed automata [7].

In many cases real-time systems that are naturally modeled as PTA can be modeled as timed automata by imposing restrictions (such as limiting the size of the string, i.e., limiting the number of allowable events), but, our experience indicates that such a timed automaton will have an enormous number of states, and thus would be unwieldy and time consuming to specify. Proving its safety and liveness properties will also be quite cumbersome simply due to the large size of the automaton.

As an example of a pushdown timed automaton, consider a language in which sequences of a's are followed by sequences of an equal number of b's (each such string has at least two a's and at least two b's). For each pair of equinumerous sequences of a's and b's, the first b symbol must appear within 5 units of time from the first a symbol and the final b symbol must appear within 20 units of time from the first a symbol. The grammar annotated with clock constraints is shown below. Note that c is a clock; clock expressions are written within braces.

$$
\begin{aligned}
&S \rightarrow R\ \ S \\
&R \rightarrow a\ \ \{c := 0\}\ \ T\ \ b\ \ \{c < 20\} \\
&T \rightarrow a\ \ T\ \ b \\
&T \rightarrow a\ \ b\ \ \{c < 5\}
\end{aligned}
$$

Note also that the first rule is coinductive [13] (i.e., a recursive rule with no base case) and accepts infinite strings. Thus, the above grammar is an ω-grammar. The pushdown timed automaton realizing this timed grammar is shown in Figure 1 (ii). In this figure, s_0 is the final state, c is the clock, and s is the stack. Actions $s.push(1)$ and $s.pop()$, respectively push 1 onto the stack and pop the stack (this automaton accepts empty string also; we allow this for simplicity of presentation). The requirement that the stack be empty ensures that only strings with equal numbers of a's and b's are accepted. Note that the wall clock time keeps advancing at the normal uniform rate, as the automaton makes transitions.

4 Modeling PTA with Coinductive CLP(R)

To model and reason about PTA and timed grammars we should be able to handle the fact that: (i) the underlying language is context free, not regular, (ii) accepted strings are infinite, and (iii) clock constraints are posed over continuously flowing time. All three aspects can be elegantly handled within LP. Thus, grammars (and automata) can be naturally modeled via LP; Specifically, the definite clause grammar (DCG) facility of Prolog allows one to obtain a parser for context-free grammars or even context-sensitive grammars with a minimal amount of work. By extending LP with co-induction, one can develop language processors that recognize infinite strings. DCGs extended with co-induction can act as recognizers for ω-pushdown automata and ω-grammars. Further, incorporation of co-induction and CLP(R) into the DCG allows modeling of time aspects of the system. Once a timed system is modeled as a coinductive CLP(R), it can be used to (i) verify if a particular timed-string will be accepted or not; and, (ii) systematically generate all possible timed strings that can be accepted. The LP realization of the system based on co-induction and CLP(R) can also be used to verify system properties by posing appropriate queries.

The general method of converting PTA to coinductive CLP(R) programs is not included here (but shown in `http://www.utdallas.edu/~nxs048000/pta.pdf`). The method takes the description of a pushdown timed automaton and generates a coinductive constraint logic program over reals. To illustrate, we describe the logic programming rendering of the pushdown timed automaton shown in Figure 1 (ii) in section 3. The generated logic program models the pushdown timed automaton as a collection of transition rules (one rule per transition in the PTA), where each rule is extended with stack actions as well as clock constraints. The first three arguments of the `trans/8` predicate are self-explanatory. The fourth argument represents the wall clock time. The pair of arguments, `Ti` and `To`, represent the clock `c` of the timed automaton. In fact, a pair of arguments have to be added for each clock that is used in the automaton (in the current example there is only one clock). The first argument of this pair is used to remember the last wall clock time this clock was reset, while the second one is used to pass on this clock's value to the next transition. The last two arguments represent the stack actions.

The coinductive `driver/6` rule realizes the automaton, calling the `trans/8` rule repeatedly (notice the absence of a base case). The CLP(R) constraints are enclosed within curly braces, as is the convention in most Prolog systems. The constraint `Ta > T` advances the time on the wall clock after every transition. The driver generates the timed trace of events as output.

```
trans(s0, a, s1, T, Ti, To, _,    [1]) :- {To = T}.
trans(s1, a, s1, T, Ti, To, C, [1| C]) :- {To = Ti}.
trans(s1, b, s2, T, Ti, To, [1| C], C) :- {T - Ti < 5,  To = Ti}.
trans(s2, b, s2, T, Ti, To, [1| C], C) :- {To = Ti}.
trans(s2, b, s0, T, Ti, To, [1| C], C) :- {T - Ti < 20, To = Ti}.

:- coinductive(driver/6).
driver([ X | R], Si, T, Ti, C1, [ (X, T) | S]) :-
            trans(Si, X, So, T, Ti, To, C1, C2),
            {Ta > T}, driver(R, So, Ta, To, C2, S).
```

[(a,0), (a,2), (b,4), (b,16), (a,20), (a,22), (a,23), (b,24), (b,36), (b,37), ...] is an example of a timed string accepted by `driver/6`. Note that the predicate `driver/6`'s coinductive termination will depend only on the first two arguments (input symbol seen and the current state, respectively), i.e., the wall-clock time and other arguments will be ignored to check if the `driver/6` predicate is cyclical. In the Co-LP system we have used for our work[1], one can declare the arguments w.r.t. which a predicate should behave coinductively. Only those arguments will be employed by the system for determining coinductive termination for that predicate. Note that for truly coinductive termination, the constraints induced in a given cycle in the PTA should also be taken into account. Thus, one must ensure that if a cycle `P` is part of an accepting string, then the constraints generated in one traversal of cycle `P` must be entailed by those generated in the next traversal of `P`. This is indeed the case for practical timed systems, where all clocks involved are reset in every accepting cycle. Due to this resetting, the same constraints are repeated in every such cycle and therefore coinductive termination is justified w.r.t. constraints also.

[1] The interpreter for Co-LP that we have used is based on YAP, and can be found in `http://www.utdallas.edu/~nxs048000/co-lp.yap`

Given this program one can pose queries to it to check if a timed string satisfies the timing constraint. Alternatively, one can generate possible (cyclical) legal timed strings. Finally, one can verify properties of this timed language (e.g., checking the simple property that all the a's are generated within 5 units of time, in any timed string that is accepted).

Next we show how our coinductive CLP(R) realization of PTA can be used to model the generalized railroad crossing problem. We have verified the safety and utility properties, as well as other interesting properties of the system, by posing appropriate queries to the program. We believe that a LP based approach to solving the GRC problem is the simplest and most elegant.

5 The Generalized Railroad Crossing (GRC)

The GRC problem has been proposed [8] as a benchmark problem in order to compare the formal methods that have been invented for specifying, designing, and analyzing real-time systems. The formal statement of the GRC problem, taken directly from [8], is as follows.

> The system to be developed operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R, i.e., $I \subseteq R$ (R is the region from where a train passes a sensor until it exits the crossing). A set of trains travel through R on multiple tracks in both directions. A sensor system determines when each train enters and exits region R. To describe the system formally, we define a gate function $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up. We also define a set $\{\lambda_i\}$ of *occupancy intervals*, where each occupancy interval is a time interval during which one or more trains are in I. The ith occupancy interval is presented as $\lambda_i = [\tau_i, \nu_i]$, where τ_i is the time of the ith entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since τ_i that no train is in the crossing (i.e., the train that entered at τ_i has exited as have any trains that entered the crossing after τ_i). Given two constants ξ_1 and ξ_2, $\xi_1 > 0$, $\xi_2 > 0$, the problem is to develop a system to operate the crossing gate that satisfies the following two properties:
>
> **Safety Property:** $t \in \cup_i \lambda_i \Rightarrow g(t) = 0$
> **Utility Property:** $t \notin \cup_i [\tau_i - \xi_1, \nu_i + \xi_2] \Rightarrow g(t) = 90$

Some positive real-valued constants are defined by the GRC as follows:

- ϵ_1, and ϵ_2, a lower bound and an upper bound on the time from when a train enters R until it reaches I respectively.
- γ_{down}, an upper bound on the time to lower the gate completely.
- γ_{up}, an upper bound on the time to raise the gate completely.
- ξ_1, an upper bound on the time from the start of lowering the gate until some train is in I.

– ξ_2, an upper bound on the time since the last train leaves I until the gate is up (unless the raising is interrupted by another train getting close to I).
– β, an arbitrarily small constant used for some technical race conditions.
– δ, the minimum useful time for the gate to be up.

Some restrictions are placed on the values of the various constants as follows:

1. $\epsilon_1 \leq \epsilon_2$.
2. $\epsilon_1 > \gamma_{down}$. (The time from when a train arrives until it reaches the crossing is sufficiently large to allow the gate to be lowered.)
3. $\xi_1 \geq \gamma_{down} + \beta + \epsilon_2 - \epsilon_1$. (The time allowed between the start of lowering the gate and some train reaching I is sufficient to allow the gate to be lowered in time for the fastest train, and then to accommodate the slowest train.)
4. $\xi_2 \geq \gamma_{up}$. (The time allowed for raising the gate is sufficient.)

6 Modeling the GRC with Coinductive CLP(R)

The GRC problem consists of several tracks and an unspecified number of trains traveling in both directions (it is theoretically possible for the gate to never go up once it goes down, if an infinite number of trains arrive one after the other within close enough interval). There is a gate at the railroad crossing that should be operated in a way that guarantees the *safety* and *utility* properties. The safety property stipulates that the gate must be down while one or more trains are in the crossing. The utility property states that the gate must be up when there is no train in the crossing. Our task is to develop (specify and prove correct) the system to control the gate at the crossing.

To model the GRC, the number of tracks has to be given as an input to the system so that any number of tracks can be handled. The system is composed of: the gate automaton that controls the gate, the controller automaton that acts as an overall controller, and a track automaton per track that models the behavior of trains traveling through each track (Figure 2). In Figure 2, for ease of understanding, we assign wall clock time (`T`) to clock variables when they are reset, since this is how clocks are realized in our implementation.

Each track is modeled as a timed automaton which works in parallel with other track automata. When an event takes place in a specified track, only that track responds to this event and all automata for other tracks remain in their current states. Track automaton has five states (Figure 2, (i)) and takes actions based on three events: *(i)* `approach` indicates a train approaching the crossing; *(ii)* `in` indicates the train being in the crossing; and, *(iii)* `exit` indicates that the train has left the crossing. The track automaton assumes that there cannot be two trains at the same time in the crossing area in each track. In other words trains travel in a safe distance from each other. The range of sensors is such that the `approach` signal of only at most one approaching train is registered for each track. Therefore, on receiving a new `approach` signal from a train on a given track, the system will respond to it assuming that there is no other train in that track or that any trains on that track will exit the crossing area before the new

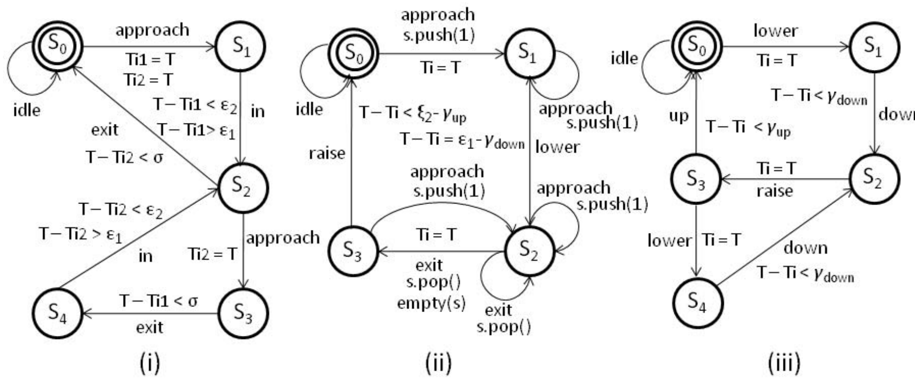


Fig. 2. (i) Track Automaton, (ii) Controller Automaton, (iii) Gate Automaton

train would arrive there (while in state `s2` the track automaton can receive a new approach). There could be a situation in which multiple trains travel one after the other (at a safe distance from each other) in one track. In this situation, the system will work properly in the sense that the first train will enter the crossing area, the second train will enter the crossing area after exiting the first train, the third train will take the place of the second train and so on. Therefore the gate remains down until the last train exits the crossing area. Note that the gate crossing system is not responsible for ensuring safe distance between trains, its task is to ensure the safety and utility of the crossing.

The gate automaton is modeled as a timed automaton with five states (Figure 2, (iii)) which takes actions based on four different events: *(i)* `lower` indicates starting of lowering the gate; *(ii)* `down` indicates the gate being down; *(iii)* `raise` indicates starting of raising the gate; and, *(iv)* `up` indicates the gate being up.

The controller automaton is modeled as a pushdown timed automaton with four states and a stack, `s` (Figure 2, (ii)). This automaton must keep track of trains currently in the system (i.e., those trains whose approach signal has been received): it has to ensure that the number of `approach` events is identical to those of `exit` events. Timed automata are not appropriate for specifying the controller, for two reasons: *(i)* we do not know the number of `approach` events in advance, so we cannot design one general timed automaton that works for an arbitrary number of tracks and trains. In other words, we would have to have different controller automata for different numbers of tracks. *(ii)* The automaton would become too complicated as the number of tracks increases. More tracks means more states and transitions, therefore a more complicated automaton. *Use of a stack in pushdown timed automaton eliminates the need for new extra states and transitions as the number of* `approach` *signals and* `tracks` *increase.*

The controller automaton must respond to four events: `approach`, `lower`, `exit`, and `raise` described above. On receiving an `approach` signal at state `s0` the controller clock will be reset. This will ensure lowering of the gate before the train gets into the crossing area. The controller clock will not get reset if the approach signal is received while in other states. The stack in pushdown timed automaton is used to keep track of the number of trains in the system. On receiving an approach signal the controller pushes the symbol "1" onto the stack and on receiving the exit signal, it pops a "1" from the stack (implemented as a counter[2]). When the stack is empty, the controller sends the raise signal to the gate as the last train has left the system and it is safe to raise the gate. A transition is activated by a pair (event, state of counter) and triggers an action on the counter. Testable states of the counter are "$= 0$" and "$\neq 0$", and counter actions are increment and decrement. The automaton may ignore the state of the counter and act on the input signal. For example on receiving an approach signal at any state, the automaton increments the counter regardless of its current state. The automaton can act based on both the input signal, and the counter state. On receiving the exit signal in state s2, the automaton checks the state of the counter. If the counter is equal to zero, the controller will go to state s3 and reset its clock; this will ensure that a raise signal will be sent to the gate automaton within $\xi_2 - \gamma_{up}$ units of time after the controller clock is reset. If the counter is not equal to zero, the controller remains in state s2.

For modeling the GRC problem we set $\epsilon_1 = 2$, $\epsilon_2 = 3$, $\gamma_{down} = 1$, $\gamma_{up} = 2$, $\xi_1 = 2$, $\xi_2 = 3$. Note that these values are taken directly from [2]. A real-time system designer can choose other values for these parameters. The GRC does not put any restrictions on how long a train can take to pass the gate crossing (theoretically speaking, a train can even stop at the gate and stay there indefinitely). To disallow such behaviors, we put an upper bound on the maximum time a train should take to exit the crossing. We introduce a constant σ, which is the maximum time in which the exit signal should appear since the approach signal was seen. For GRC, $\sigma = \infty$. Following [2] we set $\sigma = 5$. The behavior of the track automaton is specified by the following CLP(R) rules.

```
track(Trk, s0, approach, s1, T, Ti1, Ti2, To1, To2) :- {To1=T, To2=T}.
track(Trk, s1, in,       s2, T, Ti1, Ti2, To1, To2) :-
                             {T - Ti1 > 2, T - Ti1 < 3, To1=Ti1, To2=Ti2}.
track(Trk, s2, approach, s3, T, Ti1, Ti2, To1, To2) :- {To1=Ti1, To2=T}.
track(Trk, s3, exit,     s4, T, Ti1, Ti2, To1, To2) :-
                             {T - Ti1 < 5, To1=Ti1, To2=Ti2}.
track(Trk, s4, in,       s2, T, Ti1, Ti2, To1, To2) :-
                             {T - Ti2 > 2, T - Ti2 < 3, To1=Ti1, To2=Ti2}.
track(Trk, s2, exit,     s0, T, Ti1, Ti2, To1, To2) :-
                             {T - Ti2 < 5, To1=Ti1, To2=Ti2}.
track(_,    X, lower,     X, T, Ti1, Ti2, Ti1, Ti2).
track(_,    X, down,      X, T, Ti1, Ti2, Ti1, Ti2).
track(_,    X, raise,     X, T, Ti1, Ti2, Ti1, Ti2).
track(_,    X, up,        X, T, Ti1, Ti2, Ti1, Ti2).
```

The first argument of the track predicate is the track number. The second argument is current state of the track. The third argument is one of the events triggering an action explained above. The fourth argument is the new state that

[2] The stack can be realized in many ways, in this case the stack is simply implemented as a counter. In the example described in section 4 we implemented it as a list.

results. T represents the wall clock time. As we mentioned before there could be two trains at the same time in one track, i.e., one train in the crossing area and another one approaching the crossing. Therefore two clocks in track automaton are needed to handle this situation. Ti1 and Ti2 are used to remember the last wall clock time these clocks were reset, while To1 and To2 are used to pass on these clock's values to the next transition.

The behavior of the gate and controller can be specified similarly. The coinductive driver/8 predicate for GRC composes three automata, gate, track, and controller (in a similar manner as driver/6 in section 4) and generates its output as a list of $(X, Track, T)$ triples where X is an event, Track is the track number in which the event happened, and T is the time that event occurred. The set of CLP(R) rules for gate and controller along with the driver/8 predicate is shown in http://www.utdallas.edu/~nxs048000/pta.pdf.

Given the logic programming definitions of controller, track, and gate automata and the driver routine, one can check if a given sequence of timed events is legal or not, i.e., use the logic program as a simulator. One can also generate a sample sequence of timed events accepted by the system.

7 Verifying Safety and Utility Properties

Properties of interests are verified as follows: given a property Q to be verified, we specify its negation as a logic program. Let's call this predicate notQ. If the property Q holds, the query notQ will fail w.r.t. the logic program that models the system. If the query notQ succeeds, the answer provides a counter example to why the property Q does not hold.

To prove the safety property, we define the unsafe/1 predicate which looks for an accepting string that contains an in signal a little after an up signal and with no intervening down signal, i.e., we look for any possibility that a train is in the crossing area before the gate goes down, with the gate being up initially. The unsafe predicate is parameterized on the number of tracks in the system.

```
unsafe(N) :-
  driver(s0, s0, 0, 0, 0, X, N, R),
  append(C, [(in, _, _)| D], R),
  append(A, [(up, _, _)| B], C),
  not_member((down, _, _), B).
```

```
unutilized(N) :-
  driver(s0, s0, 0, 0, 0, X, N, R),
  append(A, [(down, _, _)| B], R),
  find_first_up(B, C),
  not_member((in, _, _), C).
```

The call to this predicate for any values for N fails which proves the safety of our system. Note that the not_member/2 predicate takes an element, X, and a list, L, and succeeds if X is not a member of L and fails otherwise.

Similarly we can check the utility property using the unutilized/1 predicate defined above. As mentioned before, the utility property stipulates that the gate must be up when there is no train in the crossing area. The unutilized/1 predicate looks for possibility of situations in which the gate is down without any train being in the crossing area. The find_first_up/2 predicate returns all the signals after down up to up signal. If a call to the unutilized/1 predicate fails we know that the utility property is satisfied.

Table 1. Safety and utility verification times

Number of tracks	safety	utility
1	0.006	0.006
2	0.065	0.072
3	0.6	0.587
4	5.666	5.634
5	60.013	60.430
6	426.300	453.544

Note that as the number of tracks in the system increases, the number of combinations in the system increases leading to an increase in the size of the state space. The execution time for verifying safety and utility properties therefore also increases. To keep this execution time down, and not explore irrelevant state space, we fold the negated properties into the `driver` predicate itself. Use of logic programming is again helpful here, where the negated property can be called before the call to the `driver` predicate in both `unsafe/1` and `unutilized/1`. Appropriate delay declarations must be included to ensure that the calls corresponding to the negated property are invoked only when appropriate bindings have been established by the driver. Table 1 shows the verification time in seconds for safety and utility properties for a system with up to 6 tracks (an average of five run time is taken). Our results show that our LP-based method is a practical method to verify complex real-time systems.

Other interesting properties of the system can also be verified using appropriate queries. For example one can compute the minimum time distance between two consecutive trains (i.e., two consecutive `approach` signals) in one track through a call to a simple predicate similar to `unsafe` and `unutilized`.

8 Conclusions and Related Work

Automata based real-time formalisms such as timed automata [2,8] and timed transition systems have been proposed to model and analyze a wide range of real-time systems. Jaffar [10] builds on the work of [7] and translates timed automata to a CLP program and uses it for proving assertions. All these papers do not consider PTA, they limit themselves to timed automata.

PTA with dense clocks were considered by Dang [5] and used to give a decidable characterization of the binary reachability of PTA. However, the treatment in that work is largely theoretical, there is not much focus on how to efficiently realize PTA. In contrast, we are more interested in elegantly modeling and analyzing PTA applied to complex applications such as the GRC.

Few solutions have been proposed for GRC problem. The most notable is that of Puchol [12], which is based on the ESTEREL programming language. In Puchol's work, time is discretized and thus is not faithful to the original problem. In contrast, our solution treats time as continuous. Verifying safety properties of the system in Puchol's approach is extremely complex: this complexity is such that one cannot be sure if the verification process itself is trustworthy. In our approach, in contrast, safety properties as well as other properties can be verified easily by posing simple queries.

UPPAAL has been also proposed as a toolbox for verification of real-time systems [4]. In fact, a model for a Train-Gate example is distributed with UPPAAL. UPPAAL is based on a timed automata formalism and was not designed to handle PTA directly. However, UPPAAL allows arbitrary C-code to be executed during transitions. The inclusion of this facility can be used to model PTA, however, the user has to be careful since arbitrary, low level C-code may not be amenable to verification. In contrast, in our approach, the modeling of all operations of PTA is done directly at the higher level of logic programming.

Recently, cyber-physical systems have received considerable attention [11]. Pushdown hybrid automata (of which pushdown timed automaton is an instance) constitute the foundation for CPS, and therefore, interest in this area has been significantly renewed [3].

To conclude, a combination of constraint over reals, co-induction, and the language processing capabilities of logic programming provides an expressive, and easy-to-use formalism for modeling and analyzing complex real-time systems and CPS. In fact, our framework is a general framework that can be applied to different complex systems to handle not only the time but also other continuous quantities. The LP based approach is simpler and more elegant than other approaches that have been proposed for this purpose.

References

1. Alur, R., Dill, D.L.: Automata for modeling real-time systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
2. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126, 183–235 (1994)
3. Baker, T.P. (ed.): Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, December 1-4 (2009); Baker, T.P. (ed.): IEEE Real-Time Systems Symposium. IEEE Computer Society, Los Alamitos (2009)
4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: SFM, pp. 200–236 (2004)
5. Dang, Z.: Binary reachability analysis of pushdown timed automata with dense clocks. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 506–518. Springer, Heidelberg (2001)
6. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007)
7. Gupta, G., Pontelli, E.: A constraint-based approach for specification and verification of real-time systems. In: IEEE Real-Time Systems Symp., pp. 230–239 (1997)
8. Heitmeyer, C.L., Lynch, N.A.: The generalized railroad crossing: A case study in formal verification of real-time systems. In: IEEE RTSS, pp. 120–131 (1994)

9. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. 19/20, 503–581 (1994)
10. Jaffar, J., Santosa, A.E., Voicu, R.: A CLP proof method for timed automata. In: RTSS, pp. 175–186 (2004)
11. Lee, E.A.: Cyber physical systems: Design challenges. In: ISORC (May 2008)
12. Puchol, C.: A solution to the generalized railroad crossing problem in Esterel. Technical report, Dep. of Comp. Science, The University of Texas at Austin (1995)
13. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-logic programming: Extending logic programming with coinduction. In: ICALP, pp. 472–483 (2007)
14. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science. Formal Models and Sematics (B), vol. B, pp. 133–192. MIT Press, Cambridge (1990)

Incremental Building in Peptide Computing to Solve Hamiltonian Path Problem

Muthiah Sakthi Balan and Parameswaran Seshan

E-Comm Research Lab
Infosys Technologies Limited
Bangalore - 560100, India
{sakthi_muthiah,parameswaran_seshan}@infosys.com

Abstract. To solve intractable problems using biomolecules the model requires exponential number of the same. As a proof of concept this model is acceptable, but when it comes to realization of the model the number of biomolecules needed should be drastically reduced by some techniques. In this work we address this issue for peptide computing – we propose a method called incremental building to reduce the number of peptides needed to work with for solving large combinatorial problems. We explain this model for solving the Hamiltonian path problem, analyze the space and time complexity for the same, and also discuss this method from the perspective of molecular computing as a whole and study its implications.

1 Introduction

One of the primary objectives to look for new computational models is to solve difficult combinatorial problems, called intractable problems, at an exponentially faster rate by using the massive parallelism inherent in bio-computing models. Many models such as DNA computing [1], P-systems [11] and peptide computing [9] have been proposed to solve some of the difficult problems.

In most of the bio-computing models the system works by exploring all possible ways by performing exhaustive search in parallel and by finding out the correct solution from the pool of possible solutions after performing a constant number of bio-steps. Since these models can have several copies of bio-molecules or cells, it has the flexibility to explore all possible combinations simultaneously. This helps the model to arrive at the solution, if any, at a rate exponentially faster than silicon computer models.

Peptide computing is a computing model that considers the interaction between peptides and antibodies as a computational operation and builds on the massive parallelism present in it to solve various NP-complete problems in an exponentially faster way. For solving hard combinatorial problems this computing model works as follows:

1. Formation of peptide sequences where each one represents a possible solution for the problem. For example, to solve the satisfiability problem in [9],

peptides are so formed, that each sequence represents one possible truth value for any Boolean formula over n variables. Hence, we form 2^n different peptide sequences. Note that the formation of peptide sequences does not depend on the specific input.

2. Antibodies are chosen to eliminate those sequences that do not represent the solution for the given problem. After all the elimination methods are done, if we are left with some peptide sequences then the answer will be *yes* to the decision problem, otherwise the answer will be *no*.

It is to be noted that in DNA computing, peptide computing, or P-system, we explore all possible solutions of the problem, which is already exponential in terms of its input size, and, for that to be carried out the system needs exponential number of biomolecules or cells. Even if we assume that the exponential number of molecules can be synthesized artificially, the sum of the molecular weights of the biomolecules involved in the process, would be too big to be handled in a wetlab [7].

When we examine the literature, we see that it is not only that exponential number of different bio-molecules or cells is required, but also each bio-molecule or cell should have multiplicities. When we mention about sequences, we are always talking about multiple number of the same bio-molecules or cells. Thus, a multi-set is involved in these models. We need non-determinism and parallelism in the model to compute hard problems quite efficiently.

Recently, it has been shown that a bacterial computer can be successfully used to solve the Hamiltonian Path Problem for a graph [3], by encoding gene segments to represent the graph and allowing a large number of E-coli bacteria to take those segments inside of themselves to automatically form permutations of genes. Since a large number of such bacteria is involved and given that they can self-reproduce, this gene combination activity happens in each bacterium in parallel with other bacteria, giving us benefits of parallelized processing. Some such random permutations of the DNA segments result in gene sequences that represent Hamiltonian paths in the graph - the bacteria that eventually gets such sequences inside them appear as emitting a distinct detectable fluorescence. This computing model also uses exhaustive search.

Another important point to mention in bio-computing in general is, the formation of specific bio-molecules is a pre-processing step. So the model requires that all bio-molecules, that are exponential in number, are available in order to start the processing. Therefore, when we look from the top level this computing model consists of three primary steps (see Figure 1): (i) pre-processing (can also be called as encoding) (ii) processing, and (iii) finding output, if any, and decoding it.

In this paper we first address the above issues with respect to peptide computing. The main issue here is the need for exponential number of peptide sequences even before the processing takes place. We address this problem by proposing a method called incremental building. In this method, the peptide sequences are not formed a priori unlike the older models in peptide computing, but only when it is required. We start with some part of the peptide sequence and build

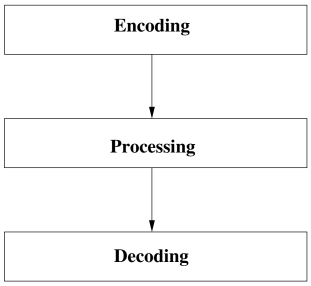


Fig. 1. Bio-computing Model

it incrementally as and when it is required. For this to happen unlike making pre-processing and processing as two separate steps, we do them in turns, i.e., alternatively. We first do a part of encoding in peptide sequences and then process them to see if they will lead to a solution, if yes then we will continue with them, or else we reject them. In the next step, with the selected peptide sequences from the first step, we add/append more peptide sequences and do the processing again to select only those peptide sequences that might lead to a solution. This sequence of encoding and processing is continued for a finite number of steps usually in linear number of steps with respect to the input size of the problem. This we call as the incremental building model. We also feel that this method can be extended to other bio-computing models and this will greatly reduce the number of bio-molecules needed for processing – this amounts to doing pre-processing and processing in turn for a finite number of steps and then decoding (see Figure 2).

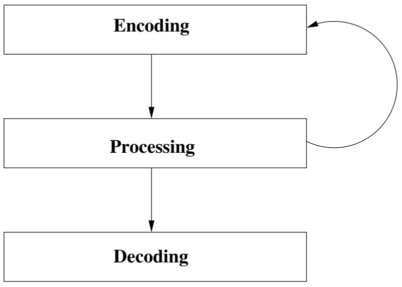


Fig. 2. Incremental Building Model

In this incremental approach the system builds the solution incrementally in an automated fashion unlike the method presented in [2] wherein, we need to form specific peptides to represent a possible solution. Even though the building of sequences that represent possible solution (Hamiltonian paths) in the

Adleman's experiment [1] and bacterial computing [3] occurs in an automated fashion, it is not done in an incremental way. Hence they suffer from explosion in number of molecules needed for processing. Incremental way provides an option of continuing with some molecules or to reject some molecules from further processing. This option is not present in the older models.

In the next section, we briefly look into the peptide computing model and the way it solves the Hamiltonian path problem which is presented in [2]. Since our paper is mainly based on the HPP we present briefly the model presented in [2]. In Section 3, we propose our new method called *incremental building* to reduce the number of molecules needed in the pre-processing and processing steps in order to solve Hamiltonian path problem. In the penultimate section we discuss the feasibility of our model with respect to present day techniques in bio-chemistry. Our paper concludes with the Section 5.

2 Background and Preliminaries

In this paper, a sequence will always mean a peptide sequence unless otherwise stated. Sequences are denoted in small-case letters, like for example p_1, p_2 and so on and antibodies are denoted by upper-case letters like A, B and so on. If A is an antibody we denote the affinity of antibody as $aff(A)$. If p_1 and p_2 are two sequences then p_1p_2 denotes the concatenation of two sequences p_1 and p_2.

2.1 A Brief Look into the Existing Models

In this section, we examine the pre-processing and processing steps in the existing bio-computing models closely.

As mentioned earlier in bio-computing models that are used for solving combinatorial problems involves three steps: first step is a pre-processing step or encoding step, then secondly, the processing step and lastly, the decoding step.

The pre-processing step consists of the preparation of bio-molecules of specific sequences or structures which altogether constitute the solution space of the given problem. The processing step explores all possible molecules constructed in the pre-processing step in search of specific molecules that represent the solution for the problem, if there is any solution for it. The main part of the processing step is the elimination of bio-molecules that does not represent the solution for the given problem. The output step is the detection part to see if there are any molecules left in the pool after all the elimination steps. Since we are concerned in this paper in reducing the number of biomolecules needed, we concentrate only on the first two steps.

When we solve an instance of a hard combinatorial problem using bio-molecular computing, all possible solutions of the problem have to be represented. And for this to happen there is a need for a pre-processing step where specific biomolecules are prepared and put in a pool to start the processing step. Since the number of possible solutions is exponential, the number of biomolecules will also be exponential. And all the more, we also need many copies of the biomolecules. This all adds to a huge number of biomolecules and makes the existing methods almost impossible, even when the input size is slightly larger, to implement in a wetlab [7].

Once we have this pool of exponential number of biomolecules then using molecular biology techniques the processing starts by eliminating those biomolecules which do not represent exact solutions for the problem. Hence, we note here that once the processing starts it is assumed that all the biomolecules representing all possible solutions are present. This means that we have all the biomolecules a priori before starting the processing. Another issue to be noted here is in most of the cases almost all, except very few, biomolecules are going to be eliminated when processing.

In the next two subsections we briefly describe two things – first, the peptide computing model and second, the method for solving Hamiltonian path problem that is presented in [2].

2.2 Peptide Computing

Peptide computing was first proposed by H. Hug and R. Schuler in [9] where they presented a method to solve Satisfiability problem [6] using the natural interactions between peptides and antibodies.

Peptide is a short sequences of proteins consisting of sequence of amino acids attached by peptide bonds. A peptide consists of recognition sites called *epitopes* for the antibodies to bind on it. A peptide can contain more than one epitopes for the same or different antibodies and for each antibody which attach to a specific epitope there is a binding power associated with it called as *affinity*. If the sites for two or more antibodies overlap in the given peptide, then the antibody with more affinity always gets the higher priority to bind to its epitope.

The process of binding of antibodies to specific sites and removal of antibodies by another antibody that binds to a site with a higher affinity[1] resembles the action a Turing machine where at some particular state the head reads/(re)writes a symbol according to some specified rules. This exemplifies the fact that there is some computation taking place when peptides and antibodies interacts. Another fact is that since the process works with several copies of peptides and antibodies it has massive parallelism and non-determinism in it. These facts provides the flexibility to solve some intractable problems efficiently.

2.3 Solving Hamiltonian Path Problem – Previous Method

Let $G = (V, E)$ be a directed graph. Let $V(G) = \{v_1, v_2, \cdots, v_n\}$ be the vertex set of the graph and $E(G) = \{e_{ij} \mid v_j \text{ is adjacent to } v_i\}$ be the edge set. Without loss of generality we take v_1 as the source vertex and v_n as the end vertex. Our problem is to test whether there exists a Hamiltonian path between v_1 and v_n. We assume that m is the number of edges in the graph G.

[1] This is basically an immune reaction.

First step is the pre-processing step that involves formation of peptides and antibodies.

For each vertices $v_1, v_2, \ldots, v_n$ we choose an epitope $e_1, e_2, \ldots, e_n$ and these epitopes are arranged in the peptide sequence in such a way that they represent a possible Hamiltonian path on a graph over n vertices. For example, one such specific peptide sequence P will be

$$P = e_1 e_2 e_2 e_3 e_3 \ldots e_{n-1} e_{n-1} e_n$$

Also see Fig. 3, for one such peptide sequence over 8 vertices.

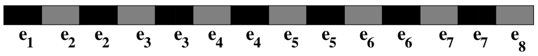


Fig. 3. Peptide sequence over 8 vertices

Likewise if we permute the set of epitopes $\{e_2, e_3, \ldots, e_{n-1}\}$ with e_1 and e_n as two extremes then we get $(n-2)!$ peptide sequences. Note that this pre-processing step does not depend on the graph G.

In the next step of the pre-processing step we form three set of antibodies. The first set of antibodies that we call as A antibodies are formed for each edge present in the graph G. These antibodies binds to the subsequence $ep_i ep_j$ in the peptide sequence if there exists an edge $v_i v_j$. The second set of antibodies called B antibodies binds to all subsequences $ep_i ep_j$ where $v_i v_j$ is not an edge in the graph G. The third set of antibodies C consists of only one labelled or colored antibody that binds to the whole of peptide sequences. If they bind to some peptide sequence then this can be seen by the emitted fluorescence as a result of this binding. The affinity of the antibodies are defined as follows:

$$aff(A) < aff(C) < aff(B).$$

The algorithm is:

Algorithm 1

```
1. Take all the peptide sequences formed in an aqueous solution.
2. Add A set of antibodies to the collection.
3. Add B set of antibodies to the collection.
4. Add C set of antibodies to the collection.
5. If fluorescence is detected then there exists a Hamilton path in the graph G,
   otherwise there exists no such path. If the peptides are bound to an addressed
   chip the solution can be immediately read.
```

It should be easy to note that even a single presence of a B antibody binding to a subsequence in a peptide sequence denotes that the path it corresponds is not a valid path in the given graph G. Hence the valid paths are the ones where the corresponding peptide sequences that contain only A antibodies binding to it. When we add the C set of antibodies since C has more affinity than A, C will remove the antibodies A and binds to that whole peptide sequence. With respect to antibodies B, C has lesser affinity, so C can not bind to those peptide sequences where there are one or more antibodies B binding to the sequences. Hence C selects all the Hamiltonian paths of the graph G, if at all it is present there.

A quick analysis will show the following resource complexity – (i) number of peptides is $(n-2)!$, (ii) length of peptides is $\mathcal{O}(n)$, and (iii) number of antibodies is $\mathcal{O}(n^2)$.

Hence we note that this model requires $(n-2)!$ peptide sequences before the start of the processing, because we assume that any of the $(n-2)!$ sequences might lead to a solution without even considering the input of the graph G.

It should also be clear to see that there are three separate steps – one encoding step, next the processing step and the last one the decoding step. For more details on this model please refer [2].

In the next section we describe how we build the peptide sequences only when it is required and reduce much of the elimination methods which in turn reduces the number of peptide sequences needed on the whole.

3 Incremental Building of Peptide Sequences

In this section we present our proposed model called incremental building of peptide sequences for solving Hamiltonian path problem.

As mentioned earlier, our main motivation here is to reduce the number of peptide sequences. Unlike the previous methods where we first assumed every possible solution might be a solution for the given problem and start the elimination process, in this method we build the required solution in an incremental way.

Let us suppose the problem statement for Hamiltonian path problem is given as in the previous section (see Section 2.3).

First we explain the pre-processing step in the following.

Pre-Processing Step

For each vertices $v_1, v_2, \ldots, v_n$ we take an unique peptide sequence. Let us denote these peptide sequences as $p_1, p_2, \ldots, p_n$. Each peptide sequence p_i for $2 \leq i \leq n-1$ is of the form (see Figure 4):

$$p_i = pre_i x_i suf_i$$

where suf_i and pre_i denote the suffix and prefix part of the sequence p_i and x_i is a random sequence. p_1 and p_n are represented as $x_1 suf_1$ and $pre_n x_n$ respectively.

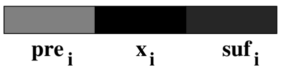


Fig. 4. Peptide sequence p_i

Hence the sequence p_1 is devoid of the prefix part and the sequence p_n the suffix part.

In the next step, we form antibodies. For each edge $e = v_i v_j, 1 \leq i, j \leq n$ and $i \neq j$, in the graph G we form antibodies A_{ij}. The epitope of A_{ij} is defined as $suf_i pre_j$. Let us denote this set of antibodies as $\mathcal{A}$.

We also form another set of labelled antibodies denoted as $\mathcal{L}$. The set $\mathcal{L}$ consists of antibodies L_i for each $v_i \in V(G)$. The antibody L_i binds to the subsequence x_i of the peptide sequence p_i.

Incremental Building

In this step we always have two sets of peptide sequences one called as *source*, denoted as S, and the other one called as *target*, denoted as T. First, the set S consists of p_1 and the set T will always be the set of all peptide sequences $\{p_2, p_3, \ldots, p_n\}$.

In the sequel we describe the steps. Please refer Algorithm 2 for step-wise description. A note on the implementation of these steps is presented in the Section 4.

First we take the source set $\{p_1\}$ in an aqueous solution. We add the target set T to it followed by the set of antibodies A. The epitope is split across two peptide sequences. This enables that, when antibodies attach to their epitopes it actually creates a link between the two peptide sequences p_1 and p_i. To be more clear, an antibody A_{1i} binds to its epitopes $suf_1 pre_i$ and creates a link between two separate peptide sequences p_1 and p_i thus forming a bigger peptide sequence consisting $p_1 p_i$. It is noted that the link between two sequences happens only when there is an edge e_{1i} in the given graph G (see Figure 5).

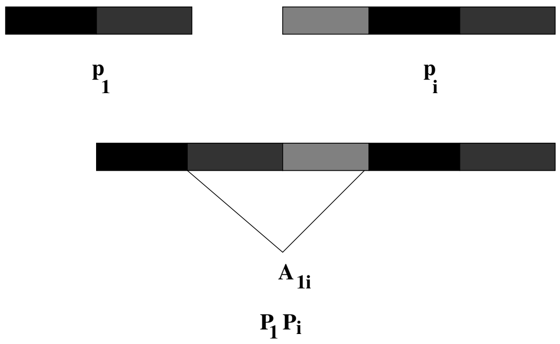


Fig. 5. Two peptide sequences p_1 and p_i linked together by the antibody A_{1i}

After this linking happens, all the antibodies A_{ij} and peptide sequences p_k that are simply floating around are filtered out. Therefore we are left with only those sequences of the form $p_1 p_j$ $(2 \leq j \leq n)$ where $e_{ij} \in E(G)$ – this set of sequences is denoted by $S^{(1)}$. After the end of this step, which we call as the first iteration, the set $S^{(1)}$ becomes the source set.

In the first step of the next iteration we add the target set T and the set of antibodies A to the aqueous solution containing $S^{(1)}$. This facilitates further elongation of the peptide sequences since antibodies A_{ij} will again link one peptide sequence from $S^{(1)}$, say $p^1 = p_1 p_i$, with a peptide sequence from T, say p_j, provided $e_{ij} \in E(G)$. At the end of this step we filter out all the floating antibodies A_{ij} and peptides p_k.

After repeating these steps $n-1$ times, if we end up with a sequence of length $n-1$ then it shows that there exists a Hamiltonian path for the graph G. If not, then there is no such path in the graph G.

When we do these iterations we might end with a sequence like $p_1 p_{i_1} p_{i_2} p_{i_3} p_{i_1}$ where $2 \leq i_1, i_2, i_3 \leq n$. We should not allow this sequence to be elongated at the next step because this has repetition of a vertex, namely v_{i_1}. To take care of these situations, we use the set of antibodies $\mathcal{L}$. The set $\mathcal{L}$ consists of antibodies L_i for each $v_i \in V(G)$. The antibody L_i binds to the subsequence x_i of the peptide sequence p_i. At the end of each iteration we add the set of antibodies $\mathcal{L}$ to the source set. If any sequence has two labelled antibodies, that can be inferred with a detection of two fluorescence emitting antibodies, then it is filtered out. One way of doing this is, each labelled antibody can be given a color (this is normally done in experiments in bio-chemistry) and through the emitted fluorescence we can find out the duplication of the vertices.

Moreover, again, at the end of i^{th} $(1 \leq i \leq n-1)$ iteration we check if the length of the peptide sequences are of length i or not. If they are not of length i then they are also filtered out. Since at the end of i^{th} iteration if the length of the peptide sequence is not of length i then it shows that it can not be further elongated and it will not lead to a Hamiltonian path for the graph G.

The complete algorithm is given below:

Algorithm 2

```
1. Take the source set of peptide sequences $S$ as the set $\{p_1\}$ in an aqueous solution;
2. Set the counter $i = 1$;
3. Add the set of peptide sequences $T$ to the solution;
4. Add the set of antibodies $\mathcal{A}$ to the solution;
5. Filter-out all the peptide sequences $p_j$ and antibodies $A_{ij}$ that are simply floating around the solution;
6. Add the set of antibodies $\mathcal{L}$ to the solution;
7. Filter-out all the sequences having a labelled antibody $L_i$ binding to two or more epitopes;
8. Filter-out all the sequences of length less than $i$;
9. If $i < n-1$ then $i = i+1$ and go to step 3 with $S$ as the set of remaining peptide sequences in the aqueous solution;
10. If there exists a peptide sequence in the solution then there exists a Hamiltonian path or else there is no such path in the graph.
```

Discussion on the Incremental Building Model

We analyze the amount of peptides and antibodies needed in the proposed incremental model in the sequel.

If we assume the number of vertices in the given graph G as n and the number of edges in G as m then the amount of peptides needed is $O(n)$. The amount of antibodies needed in the set A is $O(m)$ which is $O(n^2)$. And the amount of antibodies needed in the set L is $O(n)$. Hence the total amount of antibodies needed is of the order of n^2.

In the following we discuss the main differences between the previous model [2], that we call as the old model, and the incremental model.

1. In the old model we needed $(n-2)!$ peptide sequences but here we need only n peptide sequences that can be linked using antibodies to form larger sequences.
2. In the old model we had only constant number of bio-steps but here the number of bio-steps needed is of the order of n, the number of vertices.
3. In the old model we need to form specific peptide sequences to denote all possible paths of any graph over n number of vertices. In this incremental model the algorithm itself builds the paths automatically using antibody as a link.

4 Remarks

This paper is mostly based on the assumption that two peptide sequences can be combined together using an antibody that acts a linker. In this section we ask the following question and survey some work that has been done in this context.

Is it possible for two amino acid sequences to form a single linear sequence upon binding to a target protein?

Any two peptides can bind to a target protein in a vectorial fashion. In order to combine two separate amino acid sequences in to a single one upon binding to a target protein, those two peptides should follow certain conditions:

1. Both peptides should come together in close proximity to each other upon binding to a target protein.
2. Binding sites for these peptides should be distinct with no overlap.
3. Binding can be head-tail, tail-head or tail-tail fashion (N and C-terminal of a peptide denotes, head and tail respectively).
4. The binding of one peptide may or may not affect the binding of the other (often refereed as allosteric vs independent binding).
5. The conformation of the peptide may or may not change upon binding.
6. The binding constant between the peptides and the target protein should be in nano-molar range, so that both the peptides would be in bound-form for an extended period of time.
7. Finally, the interface of these bound peptides may create a unique key that might fit perfectly into the lock of the target protein, implying the symbiotic nature of stability of individual peptides.

The antigen binding region of the antibody is composed of hetero-dimer of Heavy chain (VH) and Light chain (VL) regions. Specificity of the antibody is attributed to the complementarity determining region (CDRs) in Heavy and light chains. It has been shown that the in vitro recombination of heavy chain and light chain binds to the antigen, albeit with low-affinity [8]. Close proximity of these two different polypeptide chains have been confirmed by the addition of linker between the two [10]. The affinity was drastically improved upon stabilizing both VH and VL.

First breakthrough in antibody design also featured linking the two different polypeptide chains with a short linker of 6 amino acids [4]. Also, the length of the linker between the polypeptide affects the stability of the complex [5]. Finally, a universal Linker theory [12] has also been proposed in order to create an antibody that can bind to two different epitopes of an antigen. The theory suggests that linking the polypeptide fragment A and B via a flexible linker drastically improves the binding to the target protein. The length of the linker depends upon the distance between the non-overlapping epitopes. It suggests that the any two polypeptides can arrange into a single polypeptide sequence with certain reservation.

5 Conclusion

We proposed a new method called incremental building for peptide computing wherein instead of building all the peptide sequences in the pre-processing step we build it incrementally only after checking if it will lead to a possible solution or not. We saw that this method reduces the number of peptide sequences required in the pre-processing step. This method, we feel, can be applied to other computing models using bio-molecules. This will have a good impact on reducing the number of bio-molecules needed for the processing since the building of further bio-molecules, like extending peptide sequences, is done only after checking the feasibility of getting to a solution. Another interesting question will be – what set of problems this method can solve.

Acknowledgments

The authors wish to thank Dr. Saravanakumar Narayanan from the Wagner Lab, Harvard Medical School for his inputs on peptide-antibody interactions.

References

1. Adleman, L.: Molecular computation of solutions to combinatorial problems. Science 266, 1021–1024 (1994)
2. Balan, M.S., Krithivasan, K., Sivasubramanyam, Y.: Peptide computing: Universality and computing. In: Jonoska, N., Seeman, N.C. (eds.) DNA 2001. LNCS, vol. 2340, pp. 290–299. Springer, Heidelberg (2002)
3. Baumgardner, J., Acker, K., Adefuye, O., Crowley, S., DeLoache, W., Dickson, J., Heard, L., Martens, A., Morton, N., Ritter, M., Shoecraft, A., Treece, J., Unzicker, M., Valencia, A., Waters, M., Malcolm, A., Heyer, L., Poet, J., Eckdahl, T.: Solving a hamiltonian path problem with a bacterial computer. Journal of Biological Engineering 3(1), 11 (2009)

4. Bird, R.E., Hardman, K.D., Jacobson, J.W., Johnson, S., Kaufman, B.M., Lee, S.M., Lee, T., Pope, S.H., Riordan, G.S., Whitlow, M.: Single-chain antigen-binding proteins. Science 242(4877), 423–426 (1988)
5. Blenner, M., Banta, S.: Characterization of the 4D5Flu single-chain antibody with a stimulus-responsive elastin-like peptide linker: A potential reporter of peptide linker conformation. Protein Science 17, 527–536 (2008)
6. Garey, M.R., Johnson, D.S.: Computers and Intractability A Guide to the Theory of NP-Completeness. W.H.Freeman and Company, New York (1979)
7. Hartmanis, J.: On the Weight of Computation. Bulletin of the EATCS 55, 136–138 (1995)
8. Hudson, N., Mudgett-Hunter, M., Panka, D., Margolies, M.: Immunoglobulin chain recombination among antidigoxin antibodies by hybridoma-hybridoma fusion. J. Immunol. 139, 2715–2723 (1987)
9. Hug, H., Schuler, R.: Strategies for the developement of a peptide computer. Bioinformatics 17, 364–368 (2001)
10. Huston, J., Levinson, D., Mudgett-Hunter, M., Tai, M.S., Novotny, J., Margolies, M., Ridge, R., Bruccoleri, R., Haber, E., Crea, R., Oppermann, H.: Protein engineering of antibody binding sites: Recovery of specific activity in an anti-digoxin single-chain Fv analogue produced in Escherichia coli. Proc. Natl. Acad. Sci. 85, 5879–5883 (1988)
11. Păun, G.: Computing with membranes–A variant: P systems with polarized membranes. Intern. J. of Foundations of Computer Science 11(1), 167–182 (2000)
12. Zhou, H.X.: Quantitative account of the enhanced affinity of two linked scFvs specific for different epitopes on the same antigen. J. Mol. Biol. 329, 1–8 (2003)

Variable Automata over Infinite Alphabets

Orna Grumberg[1], Orna Kupferman[2], and Sarai Sheinvald[2]

[1] Department of Computer Science, The Technion, Haifa 32000, Israel
[2] School of Computer Science and Engineering, Hebrew University, Jerusalem 91904, Israel

Abstract. Automated reasoning about systems with infinite domains requires an extension of regular automata to *infinite alphabets*. Existing formalisms of such automata cope with the infiniteness of the alphabet by adding to the automaton a set of registers or pebbles, or by attributing the alphabet by labels from an auxiliary finite alphabet that is read by an intermediate transducer. These formalisms involve a complicated mechanism on top of the transition function of automata over finite alphabets and are therefore difficult to understand and to work with.

We introduce and study *variable finite automata over infinite alphabets* (VFA). VFA form a natural and simple extension of regular (and ω-regular) automata, in which the alphabet consists of letters as well as variables that range over the infinite alphabet domain. Thus, VFAs have the same structure as regular automata, only that some of the transitions are labeled by variables. We compare VFA with existing formalisms, and study their closure properties and classical decision problems. We consider the settings of both finite and infinite words. In addition, we identify and study the deterministic fragment of VFA. We show that while this fragment is sufficiently strong to express many interesting properties, it is closed under union, intersection, and complementation, and its nonemptiness and containment problems are decidable. Finally, we describe a determinization process for a determinizable subset of VFA.

1 Introduction

Automata-based formal methods are successfully applied in automated reasoning about systems. When the systems are finite-state, their behaviors and specifications can be modeled by finite automata. When the systems are infinite-state, reasoning is undecidable, and research is focused on identifying decidable special cases (e.g., pushdown systems) and on developing heuristics (e.g., abstraction) for coping with the general case.

One type of infinite-state systems, motivating this work, are systems in which the control is finite and the source of infinity is data. This includes, for example, software with integer parameters [3], datalog systems with infinite data domain [15,4], and XML documents, whose leaves are typically associated with data values from some infinite domain [7,5]. Lifting automata-based methods to the setting of such systems requires the introduction of automata with *infinite alphabets*.[1]

The transition function of a nondeterministic automaton over finite alphabets (NFA) maps a state q and a letter σ to a set of states the automaton may move to when it is in

[1] Different approaches for automatically reasoning about such systems are based on extensions of first-order logic [2] and linear temporal logics [8].

state q and the letter in the input is σ. When the alphabet of the automaton is infinite, specifying all transitions is impossible, and a new formalism is needed in order to represent them in a finite manner. Existing formalisms of automata with infinite alphabets fulfill this task by augmenting the automaton by *registers* or *pebbles*, or by attributing the alphabet by labels from an auxilary finite alphabet that is read by an intermediate transducer. We elaborate of the existing formalisms below.

A register automaton [13] has a finite set of registers, each of which may contain a letter from the infinite alphabet. The transitions of a register automaton compare the letter in the input with the content of the registers, and may also store the input letter in a register. Several variants of this model have been studied. For example, [10] forces the content of the registers to be different, [12] adds alternation and two-wayness, and [9] allows the registers to change their content nondeterministically during the run.

A pebble automaton [12] places pebbles on the input word in a stack-like manner. The transitions of a pebble automaton compare the letter in the input with the letters in positions marked by the pebbles. Several variants of this model have been studied. For example, [12] studies alternating and two-way pebble automata, and [14] introduces top-view weak pebble automata.

The newest formalism is *data automata* [2,1]. For an infinite alphabet Σ, a data automaton runs on *data words*, which are words over the alphabet $\Sigma \times F$, where F is a finite auxilary alphabet. Intuitively, the finite alphabet is accessed directly, while the infinite alphabet can only be tested for equality, and is used for inducing an equivalence relation on the set of positions. Technically, a data automaton consists of two components. The first is a letter-to-letter transducer that runs on the projection of the input word on F and generates words over yet another alphabet Γ. The second is a regular automaton that runs on subwords (determined by the equivalence classes) of the word generated by the transducer.

The quality of a formalism is measured by its simplicity, expressive power, compositionality, and computability. In *simplicity*, we refer to the effort required in order to understand a given automaton, work with it, and implement it. In *compositionality*, we refer to closure under the basic operations of union, intersection, and complementation. In *computability*, we refer to the decidability and complexity of classical problems like nonemptiness, membership, universality, and containment.

The formalisms of register, pebble, and data automata all fail hard the simplicity criterion. Augmenting NFAs with registers or pebbles requires a substantial modification of the transition function. The need to maintain the registers and pebbles makes the automata hard to understand and work with. Unfortunately, most researchers in the formal-method community are not familiar with register and pebble automata. Indeed, even the definition of the basic notion of a run of such automata cannot simply rely on the familiar definition of a run of an NFA, and involves the notions of configurations, successive configurations, and so on, with no possible shortcuts.

Data automata do not come to the rescue. The need to accept several subwords per input word and to go through an intermediate alphabet and transducer makes them very complex. Even trivial languages such as a^* require extra letters and checks in order to be recognized. Simplicity is less crucial in the process of automatic algorithms, and indeed, data automata have been succesfully used for the decidability of two-variable first order logic on words with data - a formalism that is very useful in XML reasoning [2,1]. For the purpose of specification and design, and for developing new algorithms and applications, simplicity is crucial. A simpler, friendlier formalism is needed.

Data and register automata and most of their variants fail the compositionality and computability criteria too. Data automata and register automata are not closed under complementation, apart from specific fragments of register automata that limit the number of registers [8]. Their universality and containment problems are undecidable [12]. Pebble automata and most of their variants fail the computability criterion, as apart from weaker models [14], their nonemptiness, universality, and containment problems are undecidable. Nonemptiness of data and register automata is decidable, but is far more complex than the easy reachability-based nonemptiness algorithm for NFAs.

We introduce and study a new formalism for recognizing languages over infinite alphabets. Our formalism, *variable finite automata* (VFA), forms a natural and simple extension of NFAs. We also identify and study a fragment of VFA that fulfills the simplicity, compositionality, and computability criteria, and is still sufficiently expressive to specify many interesting properties. Intuitively, a VFA is an NFA some of whose letters are variables ranging over the infinite alphabet. The tight connection with NFAs enables us to apply much of the constructions and algorithms known for them.

More formally, a VFA is a pair $\mathcal{A} = \langle \Sigma, A \rangle$, where Σ is an infinite alphabet and A is an NFA, referred to as the *pattern automaton* of $\mathcal{A}$. The alphabet of A consists of *constant letters* – a finite subset of Σ, a set of *bounded variables*, and a single *free variable*. The language of $\mathcal{A}$ consists of words in Σ^* that are formed by assigning letters in Σ to the occurrences of variables in words in the language of A. Each bounded variable is assigned a different letter (also different from the constant letters), thus all occurrences of a particular bounded variable must be assigned the same letter. This allows describing words in Σ^* in which some letter is repeated. The free variable may be assigned different letters in every occurrence, different from the constant letters and from letters assigned to the bounded variables. This allows describing words in which every letter may appear. For example, consider a VFA $\mathcal{A} = \langle \mathbb{N}, A \rangle$, where A has a bounded variable x and its free variable is y. if the language of A is $(x+y)^* \cdot x \cdot (x+y)^* \cdot x \cdot (x+y)^*$, then the language of $\mathcal{A}$ consists of all words over $\mathbb{N}$ in which at least some letter occurs at least twice.

We prove that VFAs are closed under union and intersection. The constructions we present use the union and product constructions for NFAs in their basis, but some pirouettes are needed in order to solve conflicts between different assignments to the variables of the underlying automata. Such pirouettes are helpless for the problem of complementation, and we prove that VFAs are not closed under complementation. We study the classical decision problems for VFAs. We show that a VFA is nonempty iff its pattern automaton is nonempty. Thus, the nonemptiness problem is NL-complete, and is not more complex than the one for NFAs. We also show that the membership problem is NP-complete. Thus, while the problem is more complex than the one for NFAs, it is still decidable. The universality and containment problems, however, are undecidable.

We then define and study *deterministic VFA* (DVFA), a fragment of VFA in which there exists exactly one run on every word. Unlike the case of DFAs, determinism is not a syntactic property. Indeed, since the variables are not pre-assigned, there may be several runs on a word even when the pattern automaton is deterministic. However, a syntactic definition does exist and deciding whether a given VFA is deterministic is NL-complete. We introduce an *unwinding operator* for VFAs. In an unwinded VFA, each state is labeled by the variables that have been read, and therefore assigned, in paths leading to the state. Using the unwinding operator, we can define DVFAs for the union and intersection of DVFAs. Moreover, the closure under complementation of DVFAs is immediate, and it enables us to solve the universality and containment problems for DVFAs. Thus, DVFAs suggest an expressive formalism that fulfills the three criteria.

We study further properties of DVFA. As bad news, we show that the problem of determinizing a given VFA (or concluding that no equivalent DVFA exists) is undecidable. As good news, we show that all VFAs with no free variable have an equivalent DVFA, and present a determinization process for VFAs of this kind. The advantages of DVFA make us optimistic about the extensions of algorithms that involve DFAs, like symbolic formal verification and synthesis, to the setting of infinite alphabets.

We demonstrate the robustness of our formalism by showing that its extension to the setting of ω-regular words is straightforward. In Section 5, we introduce and study *variable Büchi automata* (VBAs), whose pattern automata are nondeterministic Büchi automata on infinite words [6]. VBAs are useful for specifying languages of infinite words over infinite alphabets, and in particular, specifications of systems with variables ranging over infinite domains. We show that the known relation between NFAs and nondeterministic Büchi automata extends to a relation between VFAs and VBAs. This enables us to easily lift the properties and decision procedures we presented for VFA to the setting of VBAs.

2 Variable Automata over Infinite Alphabets

A nondeterministic finite automaton (NFA) is a tuple $A = \langle \Gamma, Q, Q_0, \delta, F\rangle$, where Γ is a finite alphabet, Q is a finite set of *states*, $Q_0 \subseteq Q$ is a set of *initial states*, $\delta : Q \times \Gamma \to 2^Q$ is a *transition function*, and $F \subseteq Q$ is a set of *accepting states*. If there exists q' such that $q' \in \delta(q, a)$, we say that *a exits q*. A *run of A* on $w = \sigma_1\sigma_2 \ldots \sigma_n$ in Γ^* is a sequence of states $r = r_0, r_1, \ldots, r_n$ such that $r_0 \in Q_0$ and for every $1 \leq i \leq n$ it holds that $r_i \in \delta(r_{i-1}, \sigma_i)$. If $r_n \in F$ then r is *accepting*. Note that a run may not exist. If a run does exist, we say that w is *read along A*. The language of A, denoted $L(A)$, is the set of words on which there exists an accepting run of A.

Before defining variable automata with infinite alphabets, let us explain the idea behind them. Consider the NFA A_1 over the finite alphabet $\{x, y\}$ appearing in Figure 1. It is easy to see that $L(A_1) = x \cdot y^* \cdot x$. Consider the language $L' = \{i_1 \cdot i_2 \cdots i_k : k \geq 2, i_1 = i_k,$ and $i_j \neq i_1$ for all $1 < j < k\}$ over the alphabet $\mathbb{N}$; that is, L' contains exactly all words in which the first letter is equal to the last letter, and is different from all other letters. Since $\mathbb{N}$ is infinite, an NFW for it needs infinitely many states and transitions. The idea behind variable automata is to label the transitions of the NFA by both letters from the infinite alphabet and variables that can take values from it. For example, if we refer to x as a bounded variable whose value is fixed once assigned, and refer to y as a free variable, which can take changing values, different from the value assigned to x, then the NFA A_1, when viewed as a variable automaton over $\mathbb{N}$, recognizes the language L'. Also, if we want to remove the restriction about the letters in the middle being different from the first letter, thus consider $L'' = \{i_1 \cdot i_2 \cdots i_k : k \geq 2 \text{ and } i_1 = i_k\}$, we can label the self loop in A_1 by both x and y.

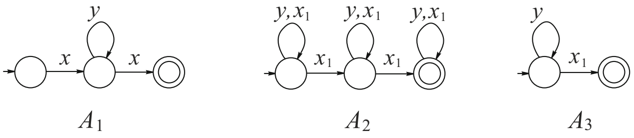


Fig. 1. The pattern automata A_1, A_2, and A_3 for the VFAs $\mathcal{A}_1$, $\mathcal{A}_2$, and $\mathcal{A}_3$

We now define *variable finite automata* (VFAs) formally. A VFA is a pair $\mathcal{A} = \langle \Sigma, A \rangle$, where Σ is an infinite alphabet and A is an NFA, to which we refer as the *pattern automaton of* $\mathcal{A}$. The (finite) alphabet of A is $\Gamma_A = \Sigma_A \cup X \cup \{y\}$, where $\Sigma_A \subset \Sigma$ is a finite set of *constant letters*, X is a finite set of *bounded variables* and y is a *free variable*. The variables in $X \cup \{y\}$ range over $\Sigma \setminus \Sigma_A$.

Consider a word $v = v_1 v_2 \ldots v_n \in \Gamma_A^*$ read along A, and another word $w = w_1 w_2 \ldots w_n \in \Sigma^*$. We say that w is a *legal instance of* v *in* $\mathcal{A}$ if

– $v_i = w_i$ for every $v_i \in \Sigma_A$,
– For $v_i, v_j \in X$, it holds that $w_i = w_j$ iff $v_i = v_j$, and $w_i, w_j \notin \Sigma_A$ and
– For $v_i = y$ and $v_j \neq y$, it holds that $w_i \neq w_j$.

Intuitively, a legal instance of v leaves all occurrences of $v_i \in \Sigma_A$ unchanged, associates every occurrence of $v_j \in X$ with the same unique letter, not in Σ_A, and associates every occurrence of y freely with letters from $\Sigma \setminus \Sigma_A$, different from these associated with X variables.

We say that a word $v \in \Gamma_A^*$ is a *witnessing pattern* for a word $w \in \Sigma^*$ if w is a legal instance of v. Note that v may be the witnessing pattern for infinitely many words in Σ^*, and that a word in Σ^* may have several witnessing patterns (or have none). Given a word $w \in \Sigma^*$, a *run of* $\mathcal{A}$ *on* w is a run of A on a witnessing pattern for w. The language of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words in Σ^* for which there exists a witnessing pattern in $L(A)$.

Example 1. Let $\mathcal{A}_2 = \langle \Sigma, A_2 \rangle$ where A_2 is the automaton appearing in Figure 1. Then, $L(\mathcal{A}_2)$ is the language of all words in Σ^* in which some letter appears at least twice. By deleting the x_1 labels from the self loops in A_2, we get the language of all words in which some letter appears exactly twice.

Example 2. Let $\mathcal{A}_3 = \langle \Sigma, A_3 \rangle$ where A_3 is the NFA appearing in Figure 1. Then $L(\mathcal{A}_3)$ is the language of all words in Σ^* in which the last letter is different from all the other letters.

Comparison with Other Formalisms. In terms of expressive power, VFAs are incomparable with FMAs – the register automata of [10], but can be simulated by NFMA [9], which extend FMAs with nondeterministic updates of the registers. Intuitively, the variables of a VFA are analogous to registers, but while a register can change its content during the run, a bounded variable cannot change the value assigned to it. VFAs are also incomparable with data automata [2], yet a VFA with no constant letters can be simulated by a data automaton. Intuitively, the transducers of data automata can be used in order to check that the restrictions imposed by the pattern automaton apply.

In the full version, we elaborate more on the relation with the existing formalisms. As detailed there, the examples showing the expressiveness superiority of the existing formalisms are tightly related to their complexity, and we do not find them appealing in practice. For example, it is not surprising that a formalism for which the emptiness problem can be checked in NL (in Theorem 3 we show that emptiness of a VFA can be reduced to emptiness of its pattern automaton) cannot recognize the language of all words in which all letters are different. Data automata can recognize this language since their notion of acceptance involves several runs, on different subwords of the word. Of course, for some applications such an ability is important. VFAs, however, are sufficiently strong to specify many natural properties, and for many applications, we rather give up the expressiveness superiority of the other formalisms for a simple and computationally easy formalism.

3 Properties of VFAs

This section studies closure properties of VFAs and the decidability and complexity of basic problems. We show that VFAs are closed under union and intersection, but are not closed under complementation. In the computability front, we show that while the emptiness problem for VFAs is not harder than the one for NFAs, the membership problem is harder, yet decidable, whereas the universality and containment problems are undecidable.

Theorem 1. *VFAs are closed under union and intersection.*

Consider two VFAs $\mathcal{A}_1 = \langle \Sigma, A_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, A_2 \rangle$ with $A_1 = \langle \Sigma_1 \cup X_1 \cup \{y_1\}, Q_1, Q_0^1, \delta_1, F_1 \rangle$ and $A_2 = \langle \Sigma_2 \cup X_2 \cup \{y_2\}, Q_2, Q_0^2, \delta_2, F_2 \rangle$.

We start with the union construction. The standard construction for NFAs, which guesses whether to follow $\mathcal{A}_1$ or $\mathcal{A}_2$, does not work for VFAs. To see why, note that the range of the variables in a standard union construction would be $\Sigma \setminus (\Sigma_1 \cup \Sigma_2)$. Accordingly, words in $L(\mathcal{A}_1)$ in which variables are assigned values in Σ_2 may be missed, and dually for $L(\mathcal{A}_2)$. We solve this problem by defining the union of $\mathcal{A}_1$ and $\mathcal{A}_2$ as a union of several copies of the underlying VFAs. In each copy, a subset of the variables is taken care of, and transitions labeled by variables from the set are labeled by constants of the other VFA.

We proceed to an intersection construction. Recall that in the product construction for NFAs A_1 and A_2, the state space is $Q_1 \times Q_2$, and $\langle q_1', q_2' \rangle \in \delta(\langle q_1, q_2 \rangle, a)$ iff $q_1' \in \delta_1(q_1, a)$ and $q_2' \in \delta_2(q_2, a)$. Since A_1 and A_2 are pattern automata of VFAs, the letter a may be a variable. Accordingly, there are cases in which it should be possible to intersect two differently labeled transitions: intersecting two transitions with different bounded variables, meaning they get the same assignment in $\mathcal{A}_1$ and in $\mathcal{A}_2$; intersecting a variable with a letter σ, meaning the variable is assigned σ; and intersecting the free variable y with a bounded variable x or with a letter σ, meaning the assignment to y in this transition agrees with the assignment of x or with σ. Accordingly, we would like to define δ such that for $z \in \Sigma_1 \cup \Sigma_2 \cup X \cup \{y\}$, we have that $\langle q_1', q_2' \rangle \in \delta(\langle q_1, q_2 \rangle, z)$ iff there exist $z_1 \in \Sigma_1 \cup X_1 \cup \{y_1\}$ and $z_2 \in \Sigma_2 \cup X_2 \cup \{y_2\}$ such that $q_1' \in \delta_1(q_1, z_1)$ and $q_2' \in \delta_2(q_2, z_2)$ and such that z_1 and z_2 can be matched according to the cases described above. Formally, we do this by taking several copies of the product construction of the pattern automata, each associated with a relation H that matches the variables and constant letters of $\mathcal{A}_1$ with the variables and constant letters of $\mathcal{A}_2$.

Theorem 2. *VFAs are not closed under complementation.*

Proof: Consider the VFA $\mathcal{A}_2$ of Example 1. Recall that $L(\mathcal{A}_2)$ contains exactly all words in Σ^* in which some letter appears at least twice. The complement $\tilde{L}$ of $L(\mathcal{A}_2)$ then contains exactly all words all of whose letters are different. It can be shown that a VFA that recognizes $\tilde{L}$ needs an unbounded number of variables, and therefore does not exist. $\square$

We now turn to study the decidability and complexity of the emptiness, membership and universality problems for VFAs. Checking nonemptiness of existing formalisms is complex and even undecidable. The fact that a bounded variable keeps its value along the run makes the nonemptiness checking of VFAs very simple. In fact, a VFA is nonempty iff its pattern automaton is nonempty. Beyond the straightforward algorithm this induces, it shows that the VFA formalism is indeed very close to the simple formalism of NFAs.

Theorem 3. *The nonemptiness problem for VFA is NL-complete.*

Theorem 4. *The membership problem for VFA is NP-complete.*

The algorithms for the universality and containment problems for the finite-alphabet case rely on the closure of NFAs under complementation, which does not hold for VFAs. Similarly to [12], for register automata, the undecidability of the universality problem for VFA is proved by a reduction from Post's Correspondence Problem. Since we can easily define a universal VFA, undecidability of the containment problem follows too.

Theorem 5. *The universality and containment problems for VFAs are undecidable.*

4 Deterministic VFA

In this section we define deterministic VFA and study their properties. We show that deterministic VFA are simple, expressive, and are closed under all Boolean operations. In addition, the nonemptiness, membership, universality, and containment problems are all decidable for them.

Recall that an NFA is deterministic if $|Q_0| = 1$ and for all $q \in Q$ and $\sigma \in \Sigma$, we have $|\delta(q, \sigma)| \leq 1$. Indeed, these syntactic conditions guarantee that the automaton has at most one run on each input word. To see that such a syntactic characterization does not exist for VFA, consider the VFA $\mathcal{A}$ appearing in Figure 2. Its pattern automaton is deterministic, but the word a has two different runs in $\mathcal{A}$: one in which x_1 is assigned a, and one in which x_2 is assigned a. Thus, there is a need to define deterministic VFAs in a non-syntactic manner.

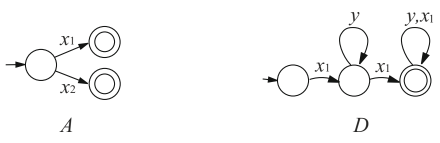


Fig. 2. A nondeterministic VFA whose pattern automaton is deterministic, and a DVFA that accepts all words in which the first letter is repeated at least twice

Definition 1. *A VFA $\mathcal{A} = \langle \Sigma, A \rangle$ is* deterministic *(DVFA, for short), if for every word $w \in \Sigma^*$, there exists exactly one run of $\mathcal{A}$ on w.*

Example 3. Consider the VFA $\mathcal{D} = \langle \Sigma, D \rangle$, where D is the DFA appearing in Figure 2. The language of $\mathcal{D}$ is the set of all words over Σ in which the first letter is repeated at least twice. To see that it is deterministic, consider a word $w = w_1 w_2 \ldots w_n$ in Σ^*. A witnessing pattern for w is over x_1 and y. Since only x_1 exits the initial state, then x_1 must be assigned w_1, and all other occurrences of other letters must be assigned to y. Therefore, every word that has a witnessing pattern has a single witnessing pattern. Since D is deterministic, every witnessing pattern has a single run in D. It is easy to see that every word in Σ^* can be read along $\mathcal{D}$. It follows that $\mathcal{D}$ is deterministic.

Although for VFA, unlike NFA, the definition of determinization is semantic, an equivalent syntactic definition does exist, as we show below.

Theorem 6. *Deciding whether VFA is deterministic is NL-complete.*

Proof: We start with the upper bound. Consider a VFA $\mathcal{A} = \langle \Sigma, A \rangle$ with variables $X \cup \{y\}$ and an initial state q_{in}. We claim that $\mathcal{A}$ is not deterministic iff one of the following holds.

- A is nondeterministic, or
- there exists a reachable state s such that there exist two bounded variables x and x' that exit s, and a path from q_{in} that reaches s and does not contain x and x', or
- there exists a bounded variable x such that both x and y exit s, and a path from q_{in} that reaches s but does not contain x, or
- there exists a reachable state s such that there exists a constant letter that does not exit s, or a variable that appears along a path from q_{in} to s that does not exit s.

Intuitively, the first three conditions check that each word $w \in \Sigma^*$ has at most one run in $\mathcal{A}$. Then, the last condition checks that w has at least one run. In order to implement the above check in NL, we guess the condition that is violated, and check that it is indeed violated. Since NL is closed under complementation, we are done. The lower bound can be shown by a reduction from the reachability problem. $\square$

Note that Theorem 6 refers to the problem of deciding whether a given VFA is deterministic and not whether it has an equivalent DVFA. As we show in the sequel, the latter problem is much harder.

We now turn to study the closure properties of DVFAs. Note that closure under union and intersection does not follow from Theorem 1, as here we want to end up with a DVFA and not with a VFA. In order to study the closure properties, we introduce an *unwinding operator* for VFAs. Given a VFA over Σ with a pattern automaton $A = \langle \Sigma_A \cup X \cup \{y\}, Q, Q_0, \delta, F \rangle$, the *unwinding* of $\mathcal{A}$ is the VFA $\mathcal{U} = \langle \Sigma, U \rangle$, with $U = \langle \Sigma_A \cup X \cup \{y\}, Q \times 2^X, \langle Q_0, \emptyset \rangle, \rho, F \times 2^X \rangle$, where ρ is defined, for every $\langle q, \theta \rangle \in Q \times 2^X$ and $z \in \Sigma_A \cup X \cup \{y\}$ as follows.

$$\rho(\langle q, \theta \rangle, z) = \begin{cases} \delta(q, z) \times \{\theta \cup \{z\}\} & z \in X \\ \delta(q, z) \times \{\theta\} & z \in \Sigma_A \cup \{y\} \end{cases} \tag{1}$$

Intuitively, the states in $\mathcal{U}$ keep track of the set of bounded variables that have been assigned along the paths from the initial state. A run of $\mathcal{A}$ corresponds to a run of $\mathcal{U}$ in which every state is augmented with the set of bounded variables that have appeared earlier in the run. Also, a run of $\mathcal{U}$ corresponds to a run of $\mathcal{A}$ along which the assignments have been accumulated. Therefore, we have that a VFA is equivalent to its unwinding.

We start with union and intersection. The constructions have the construction for DFAs in their basis, applied to the unwinding of the DVFA.

Theorem 7. *DVFA are closed under union and intersection.*

Proof: The constructions for union and intersection both rely on the unwinding of the DVFAs. Since there is a one-to-one correspondence between runs of a VFA and its unwinding, a VFA is deterministic iff its unwinding is deterministic. Let $\mathcal{U}_1$ and $\mathcal{U}_2$ be the unwindings of two DVFAs with pattern automata U_1 and U_2, respectively.

Consider two states q_1 and q_2 in U_1 and U_2, respectively. The intersection construction is based on the product construction of U_1 and U_2. Each state in the unwinding introduces at most one new bounded variable (Theorem 6). If new bounded variables x_1 and x_2 exit q_1 and q_2 respectively, the construction matches x_1 and x_2 together to form a new bounded variable. Similarly for a y_1 transition and a new bounded variable x_2. Transitions labeled by y_1 and y_2 are matched together to form a y transition. The states of the intersection construction keep track of the matchings of bounded variables.

The union of $\mathcal{U}_1$ and $\mathcal{U}_2$ is constructed on top of the intersection construction. Intuitively, a run on the union construction continues along both DVFAs as long as possible. Once it cannot continue along $\mathcal{U}_1$ (w.l.o.g.), it continues along a copy of $\mathcal{U}_2$. As in the proof of Theorem 1, several such copies are taken, in which constants of $\mathcal{U}_1$ are assigned to variables of $\mathcal{U}_2$. $\square$

The fact that a DVFA has exactly one run on each input word makes its complementation easy: one only has to complement the pattern automaton. Formally, we have the following.

Theorem 8. *Given a DVFA $\mathcal{A} = \langle \Sigma, A \rangle$ with a set of states Q and a set of accepting states F, let $\tilde{A}$ be the pattern automaton obtained from A by defining its set of accepting states to be $Q \setminus F$, and let $\tilde{\mathcal{A}} = \langle \Sigma, \tilde{A} \rangle$. Then, $L(\tilde{\mathcal{A}}) = \Sigma^* \setminus L(\mathcal{A})$.*

We now study the computability of the DVFA model. We first study the problems of nonemptiness and membership. As argued in the proof of Theorem 3, a VFA is empty iff its pattern automaton is empty. Since the nonemptiness problem in NL-complete also for DFAs, the NL-complete complexity there applies also for DVFAs. For the membership problem, determinism makes the problem easier.

Theorem 9. *The membership problem for DVFA is in PTIME.*

We note that the question of whether the membership problem is PTIME-hard, or in NL is still open, and we suspect that it is very difficult, as it has the same flavor of the long-standing open problem of the complexity of one-path LTL model checking [11]. We now turn to study the universality and containment problems and show that they are decidable.

Theorem 10. *The universality problem for DVFA is NL-complete.*

This result follows from the NL-completeness of the emptiness problem, and from the fact that complementation only involves a dualization of the acceptance condition. Since DVFA are closed under complementation and instersection, the containment problem is also decidable. In fact, we have the following.

Theorem 11. *The containment problem for DVFA is in co-NP.*

4.1 Determinization

In this section we show that not all VFAs have an equivalent DVFA, and the problem of determinizing a given VFA (or concluding that no equivalent DVFA exists) is undecidable. As good news, we point to a fragment of VFAs that can always be determinized.

One evidence that not all VFAs have an equivalent DVFA is the fact that while DVFA are closed under complementation, VFA are not. As a specific example, which also demonstrates the weakness of DVFA, consider the VFA $\mathcal{A}_2$ of Example 1. In the proof of Theorem 2, we showed that there is no VFA for the complement of $\mathcal{A}_2$. Since DVFAs are closed under complementation, it follows that there is also no DVFA equivalent to $\mathcal{A}_2$.

Theorem 12. *The problem of determinizing a given VFA (or concluding that no equivalent DVFA exists) is undecidable.*

Proof: Assume by way of contradiction that there is a Turing Machine M that, given a VFA, returns an equivalent DVFA or returns that no such DVFA exists. We construct from M a Turing machine M' that decides the universality problem for VFA, which, according to Theorem 5, is undecidable.

The machine M' proceeds as follows. Given a VFA $\mathcal{A}$, it runs M on $\mathcal{A}$. If M returns that $\mathcal{A}$ does not have an equivalent DVFA, then M' returns that $\mathcal{A}$ is not universal. Otherwise, M' returns a DVFA $\mathcal{A}'$ equivalent to $\mathcal{A}$. By Theorem 10, M' can then check $\mathcal{A}'$ for universality. $\square$

However, it turns out that VFA have an expressive determinizable fragment.

Definition 2. *A VFA is* syntactically determinizable *if it has no y transitions.*

For example, consider the syntactically determinizable VFA $\mathcal{A} = \langle \{a, \ldots, z\}^*, A \rangle$, appearing in Figure 3. The VFA $\mathcal{A}$ accepts all words of the form

$$\texttt{url=www.}x_1\texttt{.com;email=}z\texttt{@}x_1\texttt{.com} \text{ or}$$
$$\texttt{url=www.}x_2\texttt{.}t\texttt{.com;email=}z\texttt{@}x_2\texttt{.}t\texttt{.com},$$

where x_1, x_2, t, and z are words over the alphabet $\{a, \ldots, z\}$. Thus, $\mathcal{A}$ makes sure that the domain of the url agrees with that of the email, and it nondeterministically branches to allow both domain of the form x`.com` and of the form x`.`t`.com`.

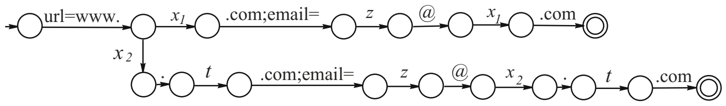


Fig. 3. A syntactically determinizable VFA

Theorem 13. *A syntactically determinizable VFA has an equivalent DVFA.*

The full details of the construction are given in the full paper. Here, we show the result of applying the algorithm on the VFA described in Figure 3. For clarity, we do not include in the figure the transition to the rejecting sinks.

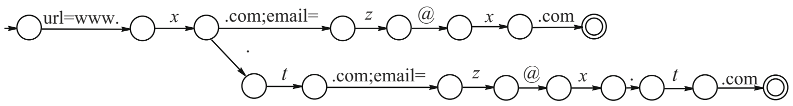


Fig. 4. The DVFA equivalent to the VFA from Figure 3

5 Variable Büchi Automata

In [6], Büchi extended NFAs to nondeterministic Büchi automata, which run on infinite words. The similarity between VFAs and NFAs enables us to extend VFAs to nondeterministic variable Büchi automata (VBA, for short). Formally, a VBA is $\mathcal{A} = \langle \Sigma, A \rangle$, where A is a nondeterministic Büchi automaton (NBA). Thus, a run of the pattern automaton A is accepting iff it visits the set of accepting states infinitely often. Similar straightforward extensions can be described for additional acceptance conditions for infinite words. As we specify below, the properties and decision procedures for VFAs generalize to VBA in the expected way, demonstrating the robustness of the VFA formalism.

We start with closure properties. The union construction for VBA is identical to the union construction for VFA. The intersection construction for NBAs involves two copies of the product automaton. Recall that the intersection construction for VFAs involves several copies of the product automaton. Combining the two constructions, we construct the intersection of two VBAs by taking two copies of these several copies. Therefore, we have the following.

Theorem 14. *VBA and DVBA are closed under union and intersection.*

As with VFAs, VBAs are not closed under complementation. Recall that a DVFA can be complemented by complementing its pattern automaton. Since deterministic Büchi automata are not closed under complementation, so are DVBA. Like deterministic Büchi automata, a DVFA can be complemented to a VBA, by translating its pattern automaton to a complementing NBA.

Theorem 15. *VBAs and DVBAs are not closed under complementation. A DVBA can be complemented to a VBA.*

As for the various decision problems, the complexities and reductions of VFAs all apply, with minor modifications.

Theorem 16.
- *The nonemptiness problem for VBA and DVBA is NL-complete.*
- *The membership problem for VBA is NP-complete and for DVBA is in PTIME.*
- *The containment problem for VBA is undecidable and for DVBA is in co-NP.*
- *Deciding whether a given VBA is a DVBA is NL-complete.*

References

1. Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L.: Two-variable logic on data trees and xml reasoning. J. ACM 56(3), 1–48 (2009)
2. Bojanczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: LICS, pp. 7–16. IEEE Computer Society, Los Alamitos (2006)
3. Bouajjani, A., Habermehl, P., Mayr, R.: Automatic verification of recursive procedures with one integer parameter. Theoretical Computer Science 295, 85–106 (2003)
4. Bouajjani, A., Habermehl, P., Jurski, Y., Sighireanu, M.: Rewriting systems with data. In: Csuhaj-Varjú, E., Ésik, Z. (eds.) FCT 2007. LNCS, vol. 4639, pp. 1–22. Springer, Heidelberg (2007)
5. Brambilla, M., Ceri, S., Comai, S., Fraternali, P., Manolescu, I.: Specification and design of workflow-driven hypertexts. J. Web Eng. 1(2), 163–182 (2003)
6. Büchi, J.: On a decision method in restricted second order arithmetic. In: Int. Congress on Logic, Method, and Philosophy of Science, pp. 1–12. Stanford University Press, Stanford (1962)
7. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Designing Data-Intensive Web Applications. Morgan Kaufmann Publishers Inc., San Francisco (2002)
8. Demri, S., Lazic, R., Nowak, D.: On the freeze quantifier in constraint ltl: Decidability and complexity. Information and Computation 7, 2–24 (2007)
9. Kaminski, M., Zeitlin, D.: Extending finite-memory automata with non-deterministic reassignment. In: Csuhaj-Varjú, E., Ézik, Z. (eds.) AFL, pp. 195–207 (2008)
10. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329–363 (1994)
11. Markey, N., Schnoebelen, P.: Model checking a path. In: Amadio, R.M., Lugiez, D. (eds.) CONCUR 2003. LNCS, vol. 2761, pp. 251–265. Springer, Heidelberg (2003)
12. Neven, F., Schwentick, T., Vianu, V.: Towards regular languages over infinite alphabets. In: Sgall, J., Pultr, A., Kolman, P. (eds.) MFCS 2001. LNCS, vol. 2136, pp. 560–572. Springer, Heidelberg (2001)
13. Shemesh, Y., Francez, N.: Finite-state unification automata and relational languages. Information and Computation 114, 192–213 (1994)
14. Tan, T.: Pebble Automata for Data Languages: Separation, Decidability, and Undecidability. PhD thesis, Technion - Computer Science Department (2009)
15. Vianu, V.: Automatic verification of database-driven systems: a new frontier. In: ICDT 2009, pp. 1–13. ACM, New York (2009)

Some Minimality Results on Biresidual and Biseparable Automata

Hellis Tamm

Institute of Cybernetics, Tallinn University of Technology,
Akadeemia tee 21, 12618 Tallinn, Estonia
hellis@cs.ioc.ee

Abstract. Residual finite state automata (RFSA) are a subclass of nondeterministic finite automata (NFA) with the property that every state of an RFSA defines a residual language of the language accepted by the RFSA. Recently, a notion of a biresidual automaton (biRFSA) – an RFSA such that its reversal automaton is also an RFSA – was introduced by Latteux, Roos, and Terlutte, who also showed that a subclass of biRFSAs called biseparable automata consists of unique state-minimal NFAs for their languages. In this paper, we present some new minimality results concerning biRFSAs and biseparable automata. We consider two lower bound methods for the number of states of NFAs – the fooling set and the extended fooling set technique – and present two results related to these methods. First, we show that the lower bound provided by the fooling set technique is tight for and only for biseparable automata. And second, we prove that the lower bound provided by the extended fooling set technique is tight for any language accepted by a biRFSA. Also, as a third result of this paper, we show that any reversible canonical biRFSA is a transition-minimal ϵ-NFA. To prove this result, the theory of transition-minimal ϵ-NFAs by S. John is extended.

1 Introduction

In automata theory, it is well known that while there is a unique minimal deterministic finite automaton (DFA) for every regular language, in many cases there exists more than one minimal nondeterministic automaton (NFA) accepting a given language. A subclass of NFAs, called *residual finite state automata* (RFSA) which has a property that is similar to the uniqueness of the minimal DFA, was introduced in [2]. An RFSA is an automaton in which every state defines a residual language of the language accepted by the automaton. There is a unique RFSA called the canonical RFSA for a given language that is a state-minimal RFSA.

Recently, a notion of a biresidual automaton (biRFSA) – an RFSA such that its reversal automaton is also an RFSA – was introduced by Latteux, Roos, and Terlutte in [8,9]. They studied minimality issues of biRFSAs, and among other things, they showed that a subclass of biRFSAs called *biseparable* automata consists of unique state-minimal NFAs for their languages. Since *bideterministic* automata are a strict subclass of biseparable automata then this result was a generalization of the uniqueness result of state minimality of bideterministic automata presented in [11]. Also, learnability of biRFSA languages has been studied in [7].

In this paper, we present some new minimality results concerning biRFSAs and biseparable automata. In the first part of the paper we consider two lower bound methods for the number of states of NFAs – the fooling set and the extended fooling set technique – and present two results related to these methods. It must be noted that the lower bounds obtained by using these methods are not always tight. Therefore, it is of interest to find classes of automata for which the lower bounds are tight.

Our first result is related to a much-cited work of Glaister and Shallit [3] presenting the fooling set technique. This method has been used recently, for example, in [5] to obtain nondeterministic state complexity results for prefix-free regular languages. In the current paper, we show that the lower bound provided by the fooling set technique is tight for and only for biseparable automata. Since any biseparable automaton is a unique state-minimal NFA for its language, it is implied that if the lower bound obtained this way is tight then the language has a unique minimal NFA. Second, we consider the extended fooling set technique introduced by Birget [1]. This technique is a little bit more general, and sometimes one may obtain better lower bounds by using this technique. We show that the lower bound provided by the extended fooling set technique is tight for any language accepted by a biRFSA. Both lower bound techniques have been investigated recently in [4].

Our third result concerns transition minimality of reversible biRFSAs. Although biRFSAs in general do not have the property of transition minimality, we will show that any reversible canonical biRFSA is transition-minimal in the class of ϵ-NFAs. From this result, transition minimality among NFAs easily follows. To prove our result, the theory for finding transition-minimal ϵ-NFAs provided by John [6] will be extended.

In [10], a proof of transition minimality of bideterministic automata among ϵ-NFAs which used the theory of transition-minimal ϵ-NFAs by John [6] was presented. However, there appears to be a flaw in the proof presented in [10]. Since bideterministic automata are a strict subclass of reversible canonical biRFSAs, the transition minimality result presented in this paper will confirm and improve that result of [10].

The paper is organized as follows. The basic definitions of automata will be given in Section 2. In Section 3, we will present the notions and some properties of RFSAs, biRFSAs and biseparable automata. In Section 4, we will recall the lower bound techniques for the number of states of NFAs proposed in [3] and [1] and present our results on biseparable automata and biRFSA languages related to these methods. In Section 5, we will present and extend the ideas of the theory of transition-minimal ϵ-NFAs of [6], and in Section 6 we apply that extended theory to show that reversible canonical biRFSAs are transition-minimal among ϵ-NFAs, and thus also among NFAs.

2 Definitions

A nondeterministic finite automaton with ϵ-transitions (ϵ-NFA) A is presented by $A = (Q, \Sigma, E, I, F)$ where Q is a finite set of states, Σ is an input alphabet, $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ is a set of transitions with ϵ being the empty string, $I \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of final states. Let $p \in Q$, $a \in \Sigma$, and $c \in \Sigma \cup \{\epsilon\}$. Let $p \cdot \epsilon$ denote the ϵ-closure of p, that is, a subset of Q consisting of p and all such states which can be reached from p by a path consisting of only ϵ-transitions. Let $p \cdot a$ denote the set $\{q \in Q \mid$ there are $p', q' \in Q$ such that $p' \in p \cdot \epsilon$, $(p', a, q') \in E$ and $q \in q' \cdot \epsilon\}$. We extend this definition in the following way: for all $P \subseteq Q$ and $x \in \Sigma^*$, $P \cdot c = \bigcup_{p \in P} p \cdot c$, $P \cdot ax = (P \cdot a) \cdot x$.

A nondeterministic finite automaton (NFA) is an ϵ-NFA which has no ϵ-transitions. An NFA is called a deterministic finite automaton (DFA) if it has a single initial state and if for any state $q \in Q$ and any $a \in \Sigma$ there is at most one out-transition of q with the label a. An NFA such that for any state $q \in Q$ and any label $a \in \Sigma$ there is at most one in-transition and one out-transition involving the state q and label a is called *reversible*.

The *reversal* of an automaton A is the automaton $A^R = (Q, \Sigma, E^R, F, I)$ where for each $p, q \in Q$ and $c \in \Sigma \cup \{\epsilon\}$, $(p, c, q) \in E^R$ if and only if $(q, c, p) \in E$. An automaton A is called *bideterministic* if both A and its reversal automaton A^R are deterministic.

A word $x \in \Sigma^*$ is *accepted* by A if and only if $I \cdot x \cap F \neq \emptyset$. The set of all words accepted by A is the *language* of A denoted by $L(A)$. Two automata are *equivalent* if they accept the same language. Let $q \in Q$. The set $L_L(A, q) = \{x \in \Sigma^* \mid q \in I \cdot x\}$ is the *left language* of q and the set $L_R(A, q) = \{x \in \Sigma^* \mid q \cdot x \cap F \neq \emptyset\}$ is the *right language* of q.

Let $L \subseteq \Sigma^*$ be a language. A language $L' \subseteq \Sigma^*$ is a *residual* of L if there is a word $u \in \Sigma^*$ such that $L' = \{v \in \Sigma^* \mid uv \in L\}$, denoted by $L' = u^{-1}L$. Sometimes, a residual is also called a *right residual.* [1] A language $L' \subseteq \Sigma^*$ is a *left residual* of L if there is a word $v \in \Sigma^*$ such that $L' = \{u \in \Sigma^* \mid uv \in L\}$, denoted by $L' = Lv^{-1}$.

A state q of A is *useful* if it is on some path from an initial state to a final state of A. Any such path is called an *accepting path* of A. An automaton is *trim* if all of its states are useful.

3 RFSA, BiRFSA and Biseparable Automata

It is known that given any trim DFA A, for any state q of A, the right language of q, $L_R(A, q)$, is a residual of $L(A)$. However, this property does not hold for NFAs in general. Therefore, the notion of a residual finite state automaton was introduced in [2]:

Definition 1. *An automaton $A = (Q, \Sigma, E, I, F)$ is a* residual finite state automaton *(RFSA) if for every state $q \in Q$, $L_R(A, q)$ is a residual of $L(A)$.*

[1] Or, sometimes the terms *quotient* and *left quotient* are used instead.

Definition 2. *A residual of a language L is* prime *if it is non-empty and if it cannot be obtained as a union of other residuals of L.*

Definition 3. *The canonical RFSA A of a regular language L is the automaton $A = (Q, \Sigma, E, I, F)$ where Q is the set of prime residuals of L, Σ is an input alphabet, I is the set of prime residuals of L which are included in L, F is the set of prime residuals of L containing the empty word, and for every prime residual S of L and for all $a \in \Sigma$, $S \cdot a = \{S' \mid aS' \subseteq S,\ S' \text{ is a prime residual of } L\}$.*

Proposition 1. ([2]). *For every state q of the canonical RFSA A of a regular language L there exists a word $u_q \in L_L(A, q)$ such that $L_R(A, q) = u_q^{-1}L$.*

Proposition 2. ([2, Theorem 2]). *The canonical RFSA of a regular language L is the unique RFSA that has a maximum number of transitions among the set of RFSA which have a minimum number of states.*

In [8,9], Latteux, Roos, and Terlutte introduced the notion of a biRFSA and discussed its properties. In the following we will present some of the results they obtained.

Definition 4. *An automaton A is a biRFSA if both A and its reversal automaton A^R are RFSA. A language is a biRFSA language if there exists a biRFSA accepting that language.*

Proposition 3. ([9, Proposition 3.1]). *A recognizable language is a biRFSA language if and only if its canonical RFSA is a biRFSA.*

In general, the canonical RFSA is not necessarily a minimal NFA. However, the following proposition holds:

Proposition 4. ([9, Proposition 4.1]). *The canonical RFSA of a biRFSA language L is a state-minimal NFA for L.*

Definition 5. *A trim NFA $A = (Q, \Sigma, E, I, F)$ is called* separable *if for every state $q \in Q$ there is some $u \in \Sigma^*$ such that $I \cdot u = \{q\}$. A is* biseparable *if both A and A^R are separable.*

Clearly, any separable automaton is an RFSA, but the converse is not true.

Proposition 5. ([9, Proposition 5.1]). *Any biseparable NFA is a canonical biRFSA.*

Proposition 6. ([9, Proposition 5.4]). *Any biseparable NFA is uniquely state-minimal.*

A result similar to Proposition 6 was presented in [11] for bideterministic automata. It is easy to see that any bideterministic automaton is biseparable by observing that any DFA satisfies the separability condition. Since biseparable automata include bideterministic automata as a proper subclass, the statement in Proposition 6 was a generalization of the result in [11]. However, it must be noted that there exist unique minimal NFAs which are not biseparable either.

4 Lower Bound Techniques for the Size of NFAs

In this section we consider two simple lower bound methods for the number of states of NFAs: the fooling set technique by Glaister and Shallit [3], and the extended fooling set technique introduced by Birget [1]. These methods can be presented, respectively, as the first and the second case of the following theorem:

Theorem 1. *Let $L \subseteq \Sigma^*$ be a regular language, and suppose there exists a set of pairs $P = \{(x_i, w_i) \mid 1 \leq i \leq n\}$ such that either*

1) (a) $x_i w_i \in L$ for $1 \leq i \leq n$, (b) $x_j w_i \notin L$ for $1 \leq i, j \leq n$, and $i \neq j$,

or

2) (a) $x_i w_i \in L$ for $1 \leq i \leq n$, (b) $x_j w_i \notin L$ or $x_i w_j \notin L$ for $1 \leq i, j \leq n$, and $i \neq j$,

holds. Then any NFA accepting L has at least n states.

The set P satisfying the conditions (a) and (b) in case 1 of Theorem 1 is called a *fooling set* of L, in case 2 it is called an *extended fooling set* of L. One can easily see that, actually, the fooling set technique is a special case of the extended fooling set technique. In fact, the latter one can provide better lower bounds for some languages.

However, neither of these techniques provides necessarily a tight bound. As an example of a language for which the lower bound obtained by applying the fooling set technique is tight, [3] gives the language $A_k = \{w \in (0+1)^k \mid w = w^R\}$ consisting of binary palindromes of length k. It is easy to see that this language is accepted by a bideterministic automaton.

In the following, we will show that the lower bound provided by the fooling set technique can be tight if and only if L is accepted by a biseparable automaton. Obviously, to get the tight lower bound, the fooling set P has to be of maximum size. Then the size of P is equal to the size of a state-minimal NFA.

Theorem 2. *Let $L \subseteq \Sigma^*$ be a regular language, and let n be the maximum integer such that there exists a set of pairs $P = \{(x_i, w_i) \mid 1 \leq i \leq n\}$ with*

(a) $x_i w_i \in L$ for $1 \leq i \leq n$;

(b) $x_j w_i \notin L$ for $1 \leq i, j \leq n$, and $i \neq j$.

Then any NFA accepting L has n states if and only if it is biseparable.

Proof. Let the assumptions of the theorem about L and P hold, and let A be an automaton that accepts L. It is clear that for every pair $(x_i, w_i) \in P$, $i = 1, ..., n$, there exists at least one state p of A such that $x_i \in L_L(A, p)$ and $w_i \in L_R(A, p)$. On the other hand, it is not difficult to see that for every state q of A there is at most one pair $(x_i, w_i) \in P$, $i \in \{1, ..., n\}$, such that $x_i \in L_L(A, q)$ and $w_i \in L_R(A, q)$. Indeed, if there were two pairs $(x_j, w_j) \in P$ and $(x_k, w_k) \in P$, $j, k \in \{1, ..., n\}, j \neq k$, such that $x_j, x_k \in L_L(A, q)$ and $w_j, w_k \in L_R(A, q)$ then $x_j w_k \in L$ which is in contradiction with the condition (b).

To prove the necessity part of the theorem, let us suppose that A has n states. Since there are n pairs in P, we can see by the reasoning above that there exists a one-to-one correspondence between the state set of A and the set P such that for every state q_i of A there is exactly one pair $(x_i, w_i) \in P$, $i \in \{1, ..., n\}$, with $x_i \in L_L(A, q_i)$ and $w_i \in L_R(A, q_i)$. Given any two states q_k and q_l of A, $q_k \neq q_l$, we know that $x_k \in L_L(A, q_k)$ but $x_k \notin L_L(A, q_l)$ since if $x_k \in L_L(A, q_l)$ then $x_k w_l \in L$ which is in contradiction with (b). Thus, for every state q_i there exists a word $x_i \in L_L(A, q_i)$ such that there is no state $q_j \neq q_i$ with $x_i \in L_L(A, q_j)$. This is equivalent to saying that A is separable. Also, we know that $w_k \in L_R(A, q_k)$ but $w_k \notin L_R(A, q_l)$ since if $w_k \in L_R(A, q_l)$ then $x_l w_k \in L$ which is in contradiction with (b). Thus, for every state q_i there exists a word $w_i \in L_R(A, q_i)$ such that there is no state $q_j \neq q_i$ with $w_i \in L_R(A, q_j)$. This is equivalent to saying that A^R is separable. We can conclude that A is biseparable.

To prove the sufficiency direction, let us assume that A is biseparable. Then for each state q of A there exists x_q such that $x_q \in L_L(A, q)$ but there is no state $q' \neq q$ such that $x_q \in L_L(A, q')$. Also, for each state q of A there exists w_q such that $w_q \in L_R(A, q)$ but there is no state $q' \neq q$ such that $w_q \in L_R(A, q')$. We claim that we can form the set P by taking its elements to be all pairs (x_q, w_q), $q \in Q$, and that $|P| = |Q|$. Indeed, it is clear that for every $q, q' \in Q$, $x_q w_q \in L$ and $x_{q'} w_q \notin L$, so the conditions (a) and (b) hold. Since for every $q, q' \in Q$, $x_q \neq x_{q'}$ and $w_q \neq w_{q'}$, then $|P| = |Q|$. □

Since any biseparable automaton is a unique minimal NFA for its language, we can state the following corollary:

Corollary 1. *Any NFA with n states accepting a language that has a fooling set of size n is a unique minimal NFA for that language.*

Next, we will show that the extended fooling set technique provides a tight lower bound for all biRFSA languages. However, it can be noted here that there are languages other than those accepted by a biRFSA for which the lower bound obtained by this technique is still tight.

Theorem 3. *Let $L \subseteq \Sigma^*$ be a biRFSA language, and let n be the maximum integer such that there exists a set of pairs $P = \{(x_i, w_i) \mid 1 \leq i \leq n\}$ with*
(a) $x_i w_i \in L$ for $1 \leq i \leq n$;
(b) $x_j w_i \notin L$ or $x_i w_j \notin L$ for $1 \leq i, j \leq n$, and $i \neq j$.
Then a state-minimal NFA accepting L has n states.

Proof. Let $A = (Q, \Sigma, E, I, F)$ be a state-minimal biRFSA accepting L. Then, for every state q of A, $L_R(A, q)$ is a right residual of L, and $L_L(A, q)$ is a left residual of L. That is, for every $q \in Q$ there exist two words x_q and w_q such that $L_R(A, q) = \{v \in \Sigma^* \mid x_q v \in L\}$ and $L_L(A, q) = \{u \in \Sigma^* \mid u w_q \in L\}$.

Next, we will show that if P is taken to be the set of all pairs (x_q, w_q), $q \in Q$, then the conditions (a) and (b) hold, and also $|P| = |Q|$. Indeed, it is clear that for every $q \in Q$, $x_q w_q \in L$. Now, suppose that for some $q, q' \in Q$ such that $q \neq q'$, both $x_q w_{q'} \in L$ and $x_{q'} w_q \in L$ hold. From $x_q w_{q'} \in L$ we get that $x_q \in L_L(A, q')$, and since $L_R(A, q) = \{v \in \Sigma^* \mid x_q v \in L\}$, we get that $L_R(A, q') \subseteq L_R(A, q)$. Similarly, from $x_{q'} w_q \in L$ we obtain $L_R(A, q) \subseteq L_R(A, q')$.

But then, $L_R(A,q) = L_R(A,q')$ which is not possible since A is a minimal NFA. Thus, it must be that either $x_q w_{q'} \notin L$ or $x_{q'} w_q \notin L$ is true. Also, it is easy to see that $|P| = |Q|$ since otherwise there would be a pair of states $q, q' \in Q$, $q \neq q'$, such that $x_q = x_{q'}$ and $w_q = w_{q'}$, implying again that A is not a minimal NFA. □

5 Transition-Minimal ϵ-NFAs

John [6] has developed a theory to reduce the number of transitions of ϵ-NFAs. In this chapter, we present the main ideas from this theory and we extend this theory to prove our result in Section 6.

Let us consider an ϵ-NFA $A = (Q, \Sigma, E, I, F)$ where the transition relation E is partitioned into subrelations E_Σ and E_ϵ such that $E_\Sigma = \{(p,a,q) \mid (p,a,q) \in E, a \in \Sigma\}$ and $E_\epsilon = \{(p,\epsilon,q) \mid (p,\epsilon,q) \in E\}$. Let $t_0 \notin E$ be a new special transition and let $E_0 = E_\Sigma \cup \{t_0\}$. Let the source and target states of a transition t be denoted as *source*(t) and *target*(t). The follow-relation $\longrightarrow$ is defined on $E_0 \times E_0$ as follows:

Definition 6. *For $s, t \in E_\Sigma$:*

$$\begin{aligned}
s \longrightarrow t \ &:\Leftrightarrow\ \textit{target}(s)\ E_\epsilon^*\ \textit{source}(t)\\
t_0 \longrightarrow t \ &:\Leftrightarrow\ \textit{there is an initial state } q \in I \textit{ with } q\ E_\epsilon^*\ \textit{source}(t)\\
s \longrightarrow t_0 \ &:\Leftrightarrow\ \textit{there is a final state } q \in F \textit{ with } \textit{target}(s)\ E_\epsilon^*\ q
\end{aligned}$$

A path $\eta = \eta_1 \cdots \eta_m$ is a sequence of $m \geq 0$ transitions with $\eta_1, \eta_m \in E_0$ and $\eta_i \in E_\Sigma$, for $i = 2, ..., m-1$, connected by the follow-relation. The transitions $\eta_i \in E_\Sigma$ where $i \in \{1,..,m\}$ are labeled by $l(\eta_i) \in \Sigma$. Let $l(t_0) = l(\epsilon) = \epsilon$. Then, the string yielded by the path η is defined to be $l(\eta) = l(\eta_1) \cdots l(\eta_m)$.

Definition 7. *Let A be an ϵ-NFA. Then $L(A) = \{w \in \Sigma^* \mid$ there is a path $\eta = \eta_1 \cdots \eta_m$ with $l(\eta) = w$ and $\eta_1 = \eta_m = t_0\}$. The automaton A is* unambiguous *if and only if for each $w \in L(A)$ there is exactly one path $\eta = \eta_1 \cdots \eta_m$ with $l(\eta) = w$ and $\eta_1 = \eta_m = t_0$.*[2]

Definition 8. *Let $t \in E_\Sigma$ with $l(t) = a$. The* future *of t is the set $\varphi(t) = \{w \in \Sigma^* \mid$ there is a path $\eta = \eta_1 \cdots \eta_m$ with $l(\eta) = w$, $\eta_1 = t$, and $\eta_m = t_0\}$. The* past *of t is the set $\pi(t) = \{w \in \Sigma^* \mid$ there is a path $\eta = \eta_1 \cdots \eta_m$ with $l(\eta) = w$, $\eta_1 = t_0$, and $\eta_m = t\}$. Also, the* strict future *of t is the set $\hat{\varphi}(t) = \{w \in \Sigma^* \mid aw \in \varphi(t)\}$ and the* strict past *of t is the set $\hat{\pi}(t) = \{w \in \Sigma^* \mid wa \in \pi(t)\}$.*

Lemma 1. (John [6, Lemma 1]). *Let $v, w \in \Sigma^*$ and $a \in \Sigma$. Then $vaw \in L(A)$ if and only if there exists a transition t such that $va \in \pi(t)$ and $aw \in \varphi(t)$.*

[2] John uses here a definition of unambiguity that allows multiple paths for passing through ϵ-transitions.

Definition 9. *Let $L \subseteq \Sigma^*$ be a regular language and let $U, V \subseteq \Sigma^*$, $a \in \Sigma$. We call (U, a, V) a* slice *of L if and only if $U \neq \emptyset$ and $V \neq \emptyset$ and $UaV \subseteq L$. A* slicing *of L is a set of slices of L. Let S be the set of all slices of L. We define a partial order on S by considering $(U_1, a, V_1) \leq (U_2, a, V_2)$ if and only if $U_1 \subseteq U_2$ and $V_1 \subseteq V_2$. We define $S_{max} \subseteq S$, the set of* maximal slices *of L, by $S_{max} := \{(U, a, V) \in S \mid$ there is no $(U', a, V') \in S$ with $(U, a, V) < (U', a, V')\}$.*

Definition 10. *Let $t \in E_\Sigma$. We define the* transition slice *of t to be the slice $(U_t, l(t), V_t)$ of $L(A)$ where $U_t = \hat{\pi}(t)$ and $V_t = \hat{\varphi}(t)$.*

Definition 11. *Assume $t_0 \notin S$ and $S_0 := S \cup \{t_0\}$. The follow-relation $\longrightarrow \subseteq S_0 \times S_0$ is defined for all slices (U_1, a, V_1) and $(U_2, b, V_2) \in S$:*

$$\begin{array}{ccccl} (U_1, a, V_1) & \longrightarrow & (U_2, b, V_2) & :\Leftrightarrow & U_1 a \subseteq U_2 \text{ and } bV_2 \subseteq V_1 \\ t_0 & \longrightarrow & (U_2, b, V_2) & :\Leftrightarrow & \epsilon \in U_2 \\ (U_1, a, V_1) & \longrightarrow & t_0 & :\Leftrightarrow & \epsilon \in V_1 \\ t_0 & \longrightarrow & t_0 & :\Leftrightarrow & \epsilon \in L \end{array}$$

Let $S' \subseteq S$ be a finite slicing of L. In order to read an automaton $A_{S'}$ out of S', each slice from S' is transformed into a transition of $A_{S'}$, and these transitions are connected via states and ϵ-transitions according to the follow-relation. John [6] shows that if the ϵ-NFA $A_{S_{max}}$ induced by the slicing S_{max} is unambiguous then this automaton has the minimum number of non-ϵ-transitions.

Lemma 2. (John [6]). *$L(A_{S_{max}}) = L$.*

Theorem 4. (John [6, Theorem 2]). *The three following statements are equivalent for languages $L \subseteq \Sigma^*$ if the slicing S_{max} of L induces an unambiguous ϵ-NFA $A_{S_{max}}$:*

- *L is accepted by an ϵ-NFA (Q, Σ, E, I, F)*
- *$L = L(A_{S'})$ for some finite slicing $S' \subseteq S$*
- *S_{max} is finite*

Furthermore, $|S_{max}| \leq |S'| \leq |E_\Sigma|$.

Corollary 2. (John [6, Corollary 3]). *An unambiguous ϵ-NFA $A_{S_{max}}$ has the minimum number of non-ϵ-transitions.*

Next, we will present an extension of the above theory of transition-minimal ϵ-NFAs that we will use to prove our result in Section 6. We will consider the case where a subset of maximal slicing consists of slices which have a property that we call *distinctiveness*. We will prove that if an ϵ-NFA induced by this kind of subset of maximal slices accepts L then this automaton has the minimum number of non-ϵ-transitions. Also, we will show that if an ϵ-NFA induced by the maximal slicing is unambiguous then all maximal slices are distinctive.

Definition 12. *Let $x = (U_x, a, V_x)$ be a slice of a language L where $U_x, V_x \subseteq \Sigma^*$ and $a \in \Sigma$. We say that x is* distinctive *if there exist strings $u_x \in U_x$ and $v_x \in V_x$ such that for any maximal slice $y = (U_y, a, V_y)$ where $y \in S_{max}$ and $y \neq x$, $u_x \notin U_y$ or $v_x \notin V_y$ holds.*

Proposition 7. *Any distinctive slice is maximal.*

Proof. Let $x = (U_x, a, V_x)$ be a distinctive slice of a language L. If we suppose that x is not maximal then there has to be some maximal slice $y = (U_y, a, V_y), y \neq x$ such that $U_x \subseteq U_y$ and $V_x \subseteq V_y$. Thus, for all $u_x \in U_x$ and $v_x \in V_x$, both $u_x \in U_y$ and $v_x \in V_y$ hold, implying that x is not distinctive. Thus, x must be maximal. □

Proposition 8. *Let $S'_{max} \subseteq S_{max}$ be a set of distinctive slices and let the ϵ-NFA induced by the slicing S'_{max} be denoted by $A_{S'_{max}}$. If $L(A_{S'_{max}}) = L$ then $A_{S'_{max}}$ has the minimum number of non-ϵ-transitions.*

Proof. Let the assumptions of the proposition hold. Let A be an ϵ-NFA accepting L. We claim that for every $x = (U_x, a, V_x)$ where $x \in S'_{max}$, there exists a transition t of A with its transition slice $x_t = (U_t, l(t), V_t)$ such that $l(t) = a$ and $x_t \leq x$ but for any $y = (U_y, a, V_y)$ where $y \in S_{max}$ and $y \neq x$, $x_t \leq y$ does not hold. Indeed, if $x \in S'_{max}$ then, since x is distinctive there exist some strings $u_x \in U_x$ and $v_x \in V_x$ such that for any $y \in S_{max}$ where $y \neq x$, $u_x \notin U_y$ or $v_x \notin V_y$ holds. Since $u_x a v_x \in L$ then by Lemma 1 there is a transition t such that $u_x \in U_t$ and $v_x \in V_t$. Since every slice is an element of a linearly ordered set of slices, then for x_t there must exist at least one maximal slice $z \in S_{max}$ such that $x_t \leq z$. Now, if $x_t \leq y$ for any $y \in S_{max}$, $y \neq x$ then it is implied that $u_x \in U_y$ and $v_x \in V_y$ which cannot be true. So, x can and must be the only maximal slice such that $x_t \leq x$. Thus, our claim as stated above holds. This implies that the number of slices in S'_{max} cannot exceed the number of non-ϵ-transitions of A. That is, the number of non-ϵ-transitions of $A_{S'_{max}}$ is less or equal to the number of non-ϵ-transitions of A. Thus, if $L(A_{S'_{max}}) = L$ then $A_{S'_{max}}$ has the minimum number of non-ϵ-transitions among all ϵ-NFAs accepting L. □

Proposition 9. *If $A_{S_{max}}$ is unambiguous then all maximal slices are distinctive.*

Proof. Let us suppose that there is a maximal slice $x = (U_x, a, V_x)$ that is not distinctive. Then for all $u_x \in U_x$ and $v_x \in V_x$ there is a maximal slice $y = (U_y, a, V_y)$, $x \neq y$, with $u_x \in U_y$ and $v_x \in V_y$. This implies that for each word $u_x a v_x$ where $u_x \in U_x$ and $v_x \in V_x$ there are at least two accepting paths in $A_{S_{max}}$ going through the transitions corresponding to x and y, respectively. Thus, $A_{S_{max}}$ is ambiguous. We conclude that if $A_{S_{max}}$ is unambiguous then every maximal slice must be distinctive. □

Remark 1. The statement in Corollary 2 can also be obtained by applying Propositions 9 and 8, and Lemma 2.

The following example presents a case of a language where not all maximal slices are distinctive, however, there exists a set of distinctive slices such that Proposition 8 can be applied.

Example 1. Let us consider the language $L = a^*bc^* + d^*bf^*$. A minimal NFA A accepting this language is presented in Figure 1. The set of maximal slices of L is

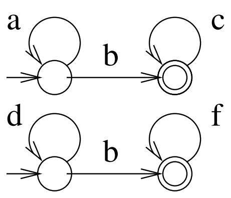


Fig. 1. Minimal NFA A of the language $a^*bc^* + d^*bf^*$

given by $S_{max} = \{(a^*, a, a^*bc^*), (a^*, b, c^*), (a^*bc^*, c, c^*), (d^*, d, d^*bf^*), (d^*, b, f^*), (d^*bf^*, f, f^*), (\epsilon, b, c^* \cup f^*), (a^* \cup d^*, b, \epsilon)\}$. In this set, the first six slices are distinctive but the last two are not. By Proposition 9, we know that $A_{S_{max}}$ is ambiguous. However, there is a set of distinctive slices $S'_{max} = \{(a^*, a, a^*bc^*), (a^*, b, c^*), (a^*bc^*, c, c^*), (d^*, d, d^*bf^*), (d^*, b, f^*), (d^*bf^*, f, f^*)\}$ consisting of all transition slices of A. By Lemma 3 in Section 6, $L(A_{S'_{max}}) = L$. In fact, A is a reversible canonical biRFSA (more precisely, it is biseparable) that will be shown to be transition-minimal among all ϵ-NFAs accepting L in Section 6.

6 Transition Minimality of Reversible BiRFSAs

Although a canonical biRFSA is a minimal NFA with respect to the number of states, it is not necessarily minimal with respect to the number of transitions. In fact, by Proposition 2, it has a maximum number of transitions among the set of RFSAs which have a minimum number of states.

In this section, we will consider the case where the canonical biRFSA is reversible. We will show that every reversible canonical biRFSA is a transition-minimal ϵ-NFA and thus also a transition-minimal NFA. It is implied that a reversible biseparable automaton is transition-minimal as well. But first we will prove the following lemma:

Lemma 3. *Let A be an ϵ-NFA and let S_A be the set of all transition slices of A. Let A_{S_A} be the ϵ-NFA induced by the slicing S_A. Then $L(A_{S_A}) = L(A)$.*

Proof. Let the assumptions of the lemma hold. First, we will show that any word accepted by A is also accepted by A_{S_A}. Let $w = w_1...w_n$, $n \geq 1$, $w \in L(A)$. Then there is a sequence of transitions $t_1, ..., t_n$ of A with $l(t_i) = w_i$, $i = 1, ..., n$, that forms a path in A accepting w. Let $x_1, ..., x_n$ be the corresponding transition slices with $x_i = (U_{x_i}, l(t_i), V_{x_i})$ where $U_{x_i} = L_L(A, source(t_i))$ and $V_{x_i} = L_R(A, target(t_i))$. Let us define the follow-relation $\longrightarrow$ for these slices according to Definition 11. Clearly, $\epsilon \in U_{x_1}$, and therefore $t_0 \longrightarrow x_1$. Also, for $i \in \{1, ..., n-1\}$, $U_{x_i} l(t_i) \subseteq U_{x_{i+1}}$ and $l(t_{i+1})V_{x_{i+1}} \subseteq V_{x_i}$, therefore $x_i \longrightarrow x_{i+1}$. Finally, since $\epsilon \in V_{x_n}$ then $x_n \longrightarrow t_0$. Since A_{S_A} was formed by transforming every transition slice of A into a transition of A_{S_A}, and these transitions were connected via states and ϵ-transitions according to the follow-relation, then there is an accepting path $t_0x_1...x_nt_0$ in A_{S_A} reading the word $w_1...w_n$. Thus $w \in$

$L(A_{S_A})$. In case $\epsilon \in L(A)$ then the follow-relation also includes $t_0 \longrightarrow t_0$, and thus $\epsilon \in L(A_{S_A})$.

Similarly, it can be shown that any word accepted by A_{S_A} is also accepted by A. □

Proposition 10. *Every transition slice of a reversible canonical biRFSA is distinctive.*

Proof. Let A be a reversible canonical biRFSA of a language L and let t be a transition of A with its transition slice (U_t, a, V_t). Let $p = source(t)$ and $q = target(t)$. Since A is a canonical biRFSA then it is a canonical RFSA, and by Proposition 1, there is a string $u_t \in L_L(A, p)$ such that $L_R(A, p) = u_t^{-1}L$. Similarly, there is a string $v_t \in L_R(A, q)$ such that $L_L(A, q) = Lv_t^{-1}$. Then $u_t \in U_t$ and $v_t \in V_t$. We claim that for any maximal slice (U, a, V) of L where $(U, a, V) \neq (U_t, a, V_t)$, $u_t \notin U$ or $v_t \notin V$ holds. Let us suppose that the claim is not true, that is, there is a maximal slice (U, a, V) with $(U, a, V) \neq (U_t, a, V_t)$ such that $u_t \in U$ and $v_t \in V$. Since $UaV \subseteq L$, $u_t \in U$ and $u_t^{-1}L = L_R(A, p)$ then $aV \subseteq L_R(A, p)$. Since A is reversible then t is the only out-transition with the label a from the state p. Therefore, $V \subseteq L_R(A, q)$. Similarly, we will get $U \subseteq L_L(A, p)$. Since $L_L(A, p) = U_t$ and $L_R(A, q) = V_t$ then we will have $U \subseteq U_t$ and $V \subseteq V_t$ which imply $(U, a, V) \leq (U_t, a, V_t)$. Since (U, a, V) is a maximal slice, it has to be that $U = U_t$ and $V = V_t$, that is, $(U, a, V) = (U_t, a, V_t)$. We have obtained a contradiction. Thus, the above claim holds implying that the slice (U_t, a, V_t) is distinctive. □

Proposition 11. *Let A be a reversible canonical biRFSA and let t_1 and t_2 be two transitions of A, $t_1 \neq t_2$ with the same label $l(t_1) = l(t_2) = a$. Let the corresponding transition slices be (U_1, a, V_1) and (U_2, a, V_2). Then $U_1 \neq U_2$ and $V_1 \neq V_2$.*

Proof. Let the assumptions of the proposition hold. Let $q_i = target(t_i)$, $i = 1, 2$. Since A is reversible then $q_1 \neq q_2$. Now, if we suppose that $V_1 = V_2$ then $L_R(A, q_1) = L_R(A, q_2)$, that is, for q_1 and q_2 the same residuals correspond. However, by Definition 3 this cannot be true for a canonical RFSA, so $V_1 \neq V_2$ must hold. Similarly, we will get that $U_1 \neq U_2$. □

Theorem 5. *A reversible canonical biRFSA has the minimum number of transitions among all ϵ-NFAs accepting the same language.*

Proof. Let A be a reversible canonical biRFSA. By Proposition 10, every transition slice of A is distinctive and so, by Proposition 7, every transition slice of A is also maximal. Let us take S'_{max} to be the set of all transition slices of A: $S'_{max} := \{(U_t, l(t), V_t) \mid t \in E\}$. By Proposition 11, there are no such pairs of transitions of A that would have the same transition slice, therefore $|S'_{max}| = |E|$. The set S'_{max} is used to form the ϵ-NFA $A_{S'_{max}}$ by converting every slice from S'_{max} into a transition of $A_{S'_{max}}$ and connecting these transitions by ϵ-transitions according to the follow-relation of Definition 11. By Lemma 3, $L(A_{S'_{max}}) = L(A)$. By Proposition 8, $A_{S'_{max}}$ has the minimum number of non-ϵ-transitions. Since the number of non-ϵ-transitions of $A_{S'_{max}}$ is equal to the number of transitions of A, and there are no ϵ-transitions in A, we conclude that A is transition-minimal among all ϵ-NFAs accepting the given language. □

Since the class of ϵ-NFAs is more general than the class of NFAs, the following corollary can be made:

Corollary 3. *A reversible canonical biRFSA is a transition-minimal NFA.*

Also, since a biseparable automaton is a canonical biRFSA (Proposition 5) then the following statement holds:

Corollary 4. *A reversible biseparable automaton is a transition-minimal NFA.*

As an anonymous referee pointed out, often, reversibility is studied as a property of languages. A regular language is considered to be reversible if there exists a reversible NFA accepting that language. It must be noted that there exist reversible languages whose canonical RFSA is not reversible. Admittedly, being reversible is a great constraint for a canonical RFSA.

Acknowledgements. This research was supported by the Estonian Center of Excellence in Computer Science, EXCS, financed by the European Regional Development Fund, and by the Estonian Science Foundation grant 7520.

References

1. Birget, J.C.: Intersection and union of regular languages and state complexity. Information Processing Letters 43, 185–190 (1992)
2. Denis, F., Lemay, A., Terlutte, A.: Residual finite state automata. In: Ferreira, A., Reichel, H. (eds.) STACS 2001. LNCS, vol. 2010, pp. 144–157. Springer, Heidelberg (2001)
3. Glaister, I., Shallit, J.: A lower bound technique for the size of nondeterministic finite automata. Information Processing Letters 59, 75–77 (1996)
4. Gruber, H., Holzer, M.: Finding lower bounds for nondeterministic state complexity is hard. In: Ibarra, O.H., Dang, Z. (eds.) DLT 2006. LNCS, vol. 4036, pp. 363–374. Springer, Heidelberg (2006)
5. Han, Y.S., Salomaa, K., Wood, D.: Nondeterministic state complexity of basic operations for prefix-free regular languages. Fundam. Inform. 90, 93–106 (2009)
6. John, S.: Minimal unambiguous ϵ-NFA. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 190–201. Springer, Heidelberg (2005)
7. Latteux, M., Lemay, A., Roos, Y., Terlutte, A.: Identification of biRFSA languages. Theoretical Computer Science 356, 212–223 (2006)
8. Latteux, M., Roos, Y., Terlutte, A.: BiRFSA languages and minimal NFAs. Technical Report GRAPPA-0205, GRAPPA (2005)
9. Latteux, M., Roos, Y., Terlutte, A.: Minimal NFA and biRFSA languages. RAIRO - Theoretical Informatics and Applications 43(2), 221–237 (2009)
10. Tamm, H.: On transition minimality of bideterministic automata. International Journal of Foundations of Computer Science 19(3), 677–690 (2008)
11. Tamm, H., Ukkonen, E.: Bideterministic automata and minimal representations of regular languages. Theoretical Computer Science 328, 135–149 (2004)

Extending Stochastic Context-Free Grammars for an Application in Bioinformatics

Frank Weinberg and Markus E. Nebel

University of Kaiserslautern, Department of Computer Sciences,
Gottlieb-Daimler-Straße,
D-67663 Kaiserslautern, Germany

Abstract. We extend stochastic context-free grammars such that the probability of applying a production can depend on the length of the subword that is generated from the application and show that existing algorithms for training and determining the most probable parse tree can easily be adapted to the extended model without losses in performance. Furthermore we show that the extended model is suited to improve the quality of predictions of RNA secondary structures.

The extended model may also be applied to other fields where SCFGs are used like natural language processing. Additionally some interesting questions in the field of formal languages arise from it.

1 Introduction

Single-stranded RNA molecules consist of a sequence of nucleotides connected by phosphordiester bonds. Nucleotides only differ by the bases involved, them being adenine, cytosine, guanine and uracil. The sequence of bases is called the *primary structure* of the molecule and is typically denoted as a word over the alphabet $\{A, C, G, U\}$. Additionally pairs of the bases can form hydrogen bonds[1] thus folding the molecule to a complex three-dimensional layout called the *tertiary structure*.

As determining the tertiary structure is computationally complex it has proven convenient to first search for the *secondary structure*, for which only a subset of the hydrogen bonds is considered, so that the molecule can be modeled as a planar graph. Additionally so-called pseudoknots are eliminated, that is, there is no subsequence "first base of (hydrogen) bond 1 ... first base of (hydrogen) bond 2 ... second base of bond 1 ... second base of bond 2" when traversing along the primary structure. See Figure 1 for an example of a secondary structure.

When abstracting from the primary structure, secondary structures are often denoted as words over the alphabet $\Sigma = \{(, |,)\}$, where a corresponding pair of parentheses represents a pair of bases connected by a hydrogen bond, while a

[1] Typically adenine pairs with uracil and guanine pairs with cytosine. Other pairs are less stable and hence less common.

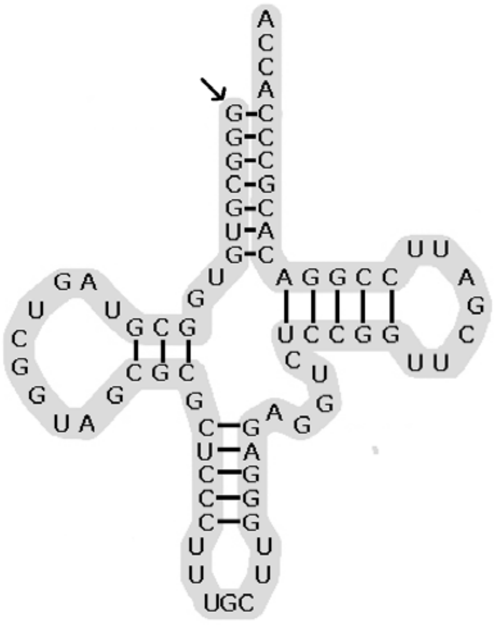


Fig. 1. Example of an RNA secondary structure. Letters represent bases, the light gray band marks the phosphordiester bonds, short edges mark hydrogen bonds.

| stands for an unpaired base. For example when starting transscription at the marked end the structure from Figure 1 would be denoted by the word

$$(((((((||(((||||||||||)))|(((((|||||||)))))|||||(((((|||||||)))))))))))).$$

One common method for computing the secondary structure is to determine the structure with minimum free energy. This was first done by Nussinov et al. who, based on the observation that a high number of paired bases corresponds to a low free energy, used dynamic programming to find the structure with the highest number of paired bases in cubic time ([10]).

While the energy models used today are much more sophisticated, taking into account e.g. the types of bases involved in a pair, the types of bases located next to them, etc., the dynamic programming scheme has remained the same (eg. [16]).

A different approach is based on probabilistic modeling. An (ambiguous) stochastic context-free grammar that generates the primary structures is chosen such that the derivation trees for a given primary structure uniquely correspond to the possible secondary structures. The probabilities of this grammar are then trained either from molecules with known secondary structures or by expectation maximization.

After this training the probabilities on the derivations as induced by the trained probabilities will model the likelihood of the corresponding secondary structures, assuming the training data was representative and the grammar is actually capable of modeling this distribution. Thus the most probable secondary structure (derivation tree) is computed as prediction ([7]).

One aspect these models as well as most other models for predicting secondary structures do not consider is that in vivo the molecules are created sequentially and the folding takes place during this creation. It has already been shown that this *co-transcriptional folding* has an effect on the resulting secondary structures (eg. [1], [8]).

This observation makes it plausible that the probability of two bases being paired depends on how far these bases are apart in the primary structure. Following this train of thought we propose an extension to the concept of SCFGs such that the probabilities of the productions depend on the length of the generated substructure (subword). We will present this extension formally in Section 2 along with the modifications necessary to allow existing algorithms for training and prediction to cope with the extended model without significant losses in performance.

We have compared the prediction quality of the modified model with the conventional one for different grammars and sets of RNA. The results, presented in detail in Section 3, show that taking the lengths into account is an improvement in most cases and an especially big improvement for very simple grammars.

2 Formal Definitions

2.1 RNA Molecules

In order to be able to apply context-free grammars we have to model primary and secondary structures as formal languages. For reasons of convenience we will in the succession use the structures and their formal language representation synonimously.

Definition 1. *([14])*
A RNA primary structure *is a word over the alphabet* $\{a, c, g, u\}$*.*
A RNA secondary structure *is a word* w *over the alphabet* $\{(, |,)\}$ *satisfying*

- $|w|_(= |w|_)$,
- *for each prefix* p *of* w*:* $|p|_(\geq |p|_)$ *and*
- w *does not contain the substring* $()$*.*[2]

2.2 Stochastic Context-Free Grammars

We assume the reader is familiar with basic terms of context-free grammars. An introduction can be found in ([6]).

[2] This substring would correspond to an extremely sharp bend in the molecule which is physically impossible.

Definition 2. *([9])*
A stochastic context-free grammar (SCFG) *is a 5-tuple $G = (I, T, R, S, P)$, where I (resp. T) is an alphabet (finite set) of intermediate (resp. terminal) symbols (I and T are disjoint), $S \in I$ is a distinguished intermediate symbol called* axiom, *$R \subset I \times (I \cup T)^*$ is a finite set of production rules and $P : R \to [0, 1]$ is a mapping such that each rule $f \in R$ is equipped with a probability $P(f)$. In the sequel we will write $A \to \alpha$ instead of $(A, \alpha) \in R$. The probabilities are chosen in such a way that for all $A \in I : \sum_{f \in R,\ Q(f)=A} P(f) = 1$ holds, where $Q(f)$ denotes the* premise *of the production f, that is, the first component A of a production rule $(A, \alpha) \in R$.*

Words are generated as for usual context-free grammars, the product of the probabilities of the used production rules provides the probabilities of a parse tree Δ, the sum of the probabilities of all possible full-parse trees of a word w provides the probability of w.

Note 1. The grammar does not necessarily induce a probalitity distribution on the language as the probabilities of the words do not add up to 1 if the probability of infinite derivations is nonzero. However, if the probabilities are acquired using maximum likelihood training – as is done in this paper – they are guaranteed to give a probability distribution on the language ([2]).

As stated in the introduction we want to include the length of the generated subword in the rule probabilities in order to model that the probability of bases being paired depends on how far they are apart in the primary structure. This can be achieved with only a slight alteration to Definition 2:

Definition 3. *A* length-dependent stochastic context-free grammar (LSCFG) *is defined as a SCFG with the following exceptions:*

- *$P : R \times \mathbb{N} \to [0, 1]$ now takes a second argument (length of subword generated).*
- *The constraint on the probabilities changes to:*
  $\forall A \in I\, \forall n \in \mathbb{N} : \sum_{f \in R,\ Q(f)=A} P(f, n) \in \{0, 1\}$.
- *When the use of a rule $f = (A, \alpha) \in R$ in the generation of a word leads to the subword $v \in T^*$, i.e. $v = \alpha$ or v is derived from α later on, the probability of this application of f is $P(f, |v|)$.*

Note 2. Allowing $\sum_{f \in R,\ Q(f)=A} P(f, n)$ to be 0 makes dealing with cases easier where there is no (sub)word of a given length that can be derived from a given intermediate symbol (e.g. $n = 0$ in an ϵ-free grammar).

This definition allows for the productions to be equipped with arbitrary probability functions as long as for any given pair (premise, length) they represent a probability distribution or are all 0. In the present paper we will however confine ourselves with grouping the lengths together in several intervals. This allows for the probabilities to be stored as a vector and be retrieved in the algorithms without further computation. It is however, as we will see, still powerful enough to yield a significant improvement over non-length-dependent SCFGs with respect to the quality of the prediction.

Note 3. Since for bottom up parsing/training algorithms like CYK or the inside and outside algorithm (see e.g. [4]) the length of the generated subword is determined by the position in the dynamic programming matrix, these algorithms will need no change other than adding the length as an additional parameter for probability lookup[3] in order to use them for LSCFGs.

For probabilistic Earley parsing ([13]) we have to regard that the length of the generated subword will only be known in the completion step. Thus for LSCFGs we have to multiply in the rule probability in this step instead of the predictor step, as is usally done.

In both cases the changes do not influence the run-time significantly.

In order to train the grammars we can make use of the fact that we know the secondary structures for our training data. The derivation corresponding to the correct secondary structure thus can be used to determine the relative frequency of each production among all productions with the same source[4]. As shown in [11] these relative frequencies are a maximum likelihood estimator for the probabilities that lead to the generation of the training data.

2.3 Determining the Most Probable Derivation

In order to find the most probable derivation for a given primary structure we decided to employ a probabilistic Earley parser, since it allows to use the grammars unmodified while the commonly used CYK algorithm requires the grammars to be transformed into Chomsky normal form.

A (non-probabilistic) Earley parser operates on lists of items (also called dotted productions), representing partial derivations. We will write $(i : {}_jX \to \alpha.\beta)$ if itemlist i, $0 \leq i \leq |w|$, w the parsed word, contains the item ${}_jX \to \alpha.\beta$ with semantics: There is a (leftmost) derivation $S \underset{lm}{\overset{*}{\Rightarrow}} w_{1,j}X\gamma$, a production $X \to \alpha\beta$ and a derivation $\alpha \overset{*}{\Rightarrow} w_{j+1,i}$. The item is considered to represent the partial derivation $X \Rightarrow \alpha\beta \overset{*}{\Rightarrow} w_{j+1,i}\beta$.

The parser is initialized with $(0 : {}_0S' \to .S)$, S' a new symbol, S the axiom of the grammar, $S' \to .S$ a new production. Then the transitive closure with respect to the following operations is computed:

$$\begin{aligned}
\text{Scanner: } & \text{If } \exists\ (i : {}_kX \to \beta_1.a\beta_2) \text{ and } w_{i+1} = a, \\
& \text{add } (i+1 : {}_kX \to \beta_1 a.\beta_2). \\
\text{Predictor: } & \text{If } \exists\ (i : {}_kX \to \beta_1.A\beta_2) \text{ and } A \to \alpha \in R, \\
& \text{add } (i : {}_iA \to .\alpha). \\
\text{Completer: } & \text{If } \exists\ (i : {}_jY \to \nu.) \text{ and } (j : {}_kX \to \beta_1.Y\beta_2), \\
& \text{add } (i : {}_kX \to \beta_1 Y.\beta_2).
\end{aligned}$$

[3] Of course the re-estimation step of the inside-outside algorithm has to be done seperately for each length interval, each time considering the respective parts of the matrix.

[4] In the length-dependent case these relative frequencies have to be determined seperately for each length interval.

Intuitively the scanner advances the point past terminal symbols if they match the corresponding symbol of the parsed word, the predictor adds all the productions that might yield a valid extension of the following (nonterminal) symbol and the completer advances the point, if one actually did.

We then have $w \in \mathcal{L}(G) \Leftrightarrow \exists\ (|w| :\ {}_0S' \rightarrow S.)$.

If we want to determine the most probable derivation of a word with respect to a SCFG (either length-dependent or not) we need to keep track of the probabilities of partial derivations. This can simply be done by adding them to the items as an additional parameter.

The initialisation then adds $(0 :\ {}_0S' \rightarrow .S, 1)$ and the operations change as follows:

$$\begin{aligned}
\text{Scanner: }&\text{If } \exists\ (i :\ {}_kX \rightarrow \beta_1.a\beta_2, \gamma) \text{ and } w_{i+1} = a,\\
&\text{add } (i+1 :\ {}_kX \rightarrow \beta_1 a.\beta_2, \gamma).\\
\text{Predictor: }&\text{If } \exists\ (i :\ {}_kX \rightarrow \beta_1.A\beta_2, \gamma) \text{ and } A \rightarrow \alpha \in R,\\
&\text{add } (i :\ {}_iA \rightarrow .\alpha, 1).\\
\text{Completer: }&\text{If } \exists\ (i :\ {}_jY \rightarrow \nu., \gamma_1),\ \exists\ (j :\ {}_kX \rightarrow \beta_1.Y\beta_2, \gamma_2) \text{ and}\\
&\quad \nexists\ (i :\ {}_kX \rightarrow \beta_1 Y.\beta_2, \gamma)\\
&\quad \text{where } \gamma > \gamma' := \gamma_1 \cdot \gamma_2 \cdot P(Y \rightarrow \nu, i-j),\\
&\text{add } (i :\ {}_kX \rightarrow \beta_1 Y.\beta_2, \gamma').
\end{aligned}$$

The modifications of scanner and predictor are straightforward. Since choosing the most probable sequence of steps for each partial derivation will lead to the most probable derivation overall, the completer maximises the overall probability by choosing the most probable alternative, whenever there are multiple possibilities for generating a subword.

If $w \in \mathcal{L}(G)$ we will find $(|w| :\ {}_0S' \rightarrow S., \gamma)$, where γ is the probability of the most probable derivation of w.

Note 4. For a more detailed introduction of probabilistic Earley parsing as well as a proof of correctness and hints on efficient implementation see ([13]).

Differing from ([13]) we multiply in the rule probabilities $P(f, i-j)$ during completion instead of prediction for the reasons mentioned in Note 3.

3 Application

In order to see if adding length-dependency actually improves the quality of the predictions from stochastic context-free grammars we used length-dependent and traditional versions of four different grammars to predict two sets of RNA molecules for which the correct secondary structure is already known. Both sets were split into a training set which was used to train the grammars and a benchmark set for which secondary structure were predicted using the trained grammars. We then compared these predicted structures to the structures from the database, computing two commonly used criteria to measure the quality:

- **Sensitivity:** The relative frequency of correctly predicted pairs among pairs that appear in the correct structure.

– **Specificity:** The relative frequency of correctly predicted pairs among pairs that have been predicted.

Both frequencies were computed over the complete set (instead of calculating individual scores for each molecule and taking the average of these).

3.1 Data

In [3] Dowell and Eddy compared the prediction quality of several different grammars as well as some commonly used programs that predict RNA secondary structures by minimizing free energy. We decided to use the same data so our results are directly comparable to theirs.

Their training set consists of 139 each large and small subunit rRNA's, the benchmark dataset contains 225 RNase P's, 81 SRPs and 97 tmRNA's. Both sets are available from http://selab.janelia.org/software/conus/ . Since it contains different types of RNA we will refer to this set as the mixed set for the remainder of this article.

Additionally we wanted to see if length-dependent prediction can further improve the prediction quality for tRNA which is already predicted well by conventional SCFGs.

In order to do so we took the tRNA database from [12], filtered out all sequences with unidentified bases and split the remaining data into a training set of 1285 sequences and a benchmark set of 1284 sequences.

3.2 Grammars

We used 4 different grammars for our experiments:

$$\text{G1: } S \to bS \mid aS\hat{a}S \mid \epsilon$$

$$\begin{aligned}\text{G2: } S &\to LS \mid L \\ L &\to aF\hat{a} \mid b \\ F &\to aF\hat{a} \mid LS\end{aligned}$$

$$\begin{aligned}\text{G3: } S &\to SP \mid Sb \mid b \mid P \\ P &\to aP\hat{a} \mid aR\hat{a} \\ R &\to Tb \mid TP \\ T &\to Tb \mid TP \mid b \mid P\end{aligned}$$

$$\begin{aligned}\text{G4: } S &\to SAC \mid C \\ C &\to Cb \mid \epsilon \\ A &\to aL\hat{a} \\ L &\to aL\hat{a} \mid M \mid I \mid bH \mid aL\hat{a}Bb \mid bBaL\hat{a} \\ B &\to Bb \mid \epsilon \\ H &\to Hb \mid \epsilon \\ I &\to bJaL\hat{a}Kb \\ J &\to Jb \mid \epsilon \\ K &\to Kb \mid \epsilon \\ M &\to UaL\hat{a}UaL\hat{a}N \\ N &\to UaL\hat{a}N \mid U \\ U &\to Ub \mid \epsilon\end{aligned}$$

Since the words generated by the grammars are the primary structures and the derivation trees correspond to the possible secondary structures, terminal symbols are generated in two ways. Either as a single unpaired base, denoted in the grammars by b or as a pair of bases, denoted by a and $\hat{a}$.

In order to keep the grammar sizes at bay and to remain consistent with [3] we seperated transition and emission probabilities by leaving b in the grammar as a nonterminal and replacing all occurences of $aL\hat{a}$ in G4 with P, adding rules that generate all (combinations of) terminal symbols from these nonterminals.

G1 and G2 have been taken from [3], G1 being the simplest grammar in the comparison and G2, which originates from [7], being the grammar which achieved the best results. G4 has been taken from [9]. It models the decomposition that is used for minimum free energy prediction.

As we stated in Section 2 we implemented length-dependency such that we grouped the lengths into intervals, the rule probabilities changing only from one interval to the other but not within them.

Since the influence a change in length has on the probabilities most likely depends on the relative change rather than the absolute one, we decided to make the intervals longer as the subwords considered get longer. This also helps to keep the estimated probabilities accurate since naturally any training set will contain fewer datapoints per length as the length gets longer.

After doing a few quick tests which indicated that the above assumptions are reasonable we decided on the intervals $[0; 5]$, $[6; 10]$, $[11; 20]$, $[21; 30]$, $[31; 40]$, $[41; 50]$, $[51; 60]$, $[61; 100]$, $[101; 150]$, $[151; 200]$, $[201; 250]$, $[251; 300]$, $[301; 350]$, $[351; 400]$, $[401; 500]$ and $[501; \infty]$[5], which we used for the length-dependent version of all four grammars.

3.3 Observations and Dicussion

We did the training and prediction using the length-dependent Earley-parser from [5]. The results of the benchmarks are listed in Table 1.

Table 1. Grammar performance, given as sensitivity % (specificity %) rounded to full percent

Grammar	**Mixed Set** without	**Mixed Set** with	**tRNA Set** without	**tRNA Set** with
	lengths		lengths	
G1	3 (4)	17 (13)	7 (8)	46 (49)
G2	47 (45)	45 (40)	81 (81)	84 (88)
G3	41 (49)	46 (41)	79 (82)	93 (93)
G4	39 (47)	44 (54)	80 (84)	88 (91)

With the exception of G2 and partially G3 on the mixed dataset the length dependent grammars yielded better predictions than their conventional counterparts.

[5] Since all structures in the benchmark sets are shorter than 500 bases the probabilities of the last interval did not influence the predictions.

The improvement for the simple grammar G1 is especially noticeable, reaching a factor of 6 for the tRNA set. However the values still rank significantly below those of the other grammars.

For G2, G3 and G4 with the mixed set we observe that with one exception none of the values exceeds 50%. As the same was also observed in [3] when the authors tried to enhance prediction quality by modelling stacking correlations with their grammars, it seems possible that this is a natural limit for the prediction quality that can be achieved on this set of data with the approach of determining the most likely parse in a SCFG.

This assumption is further supported by the fact that on the tRNA set the prediction quality could significantly be improved by taking lengths into account. Note that except for G2 sensitivity the number of missing resp. mispredicted pairs was reduced by at least 1/3.

The quality of the predictions of G3 with lengths on tRNA is illustrated not only by the 93% sensiivity and specificity but also by the fact that it predicted 712 (55%) of the molecules completely correct which is twice as many as the 200–400 correct structures the other grammars achieved (except for G1 which got no structure completely correct).

3.4 Runtime

The considerations in Note 3 lead to the assumption that both versions should take about the same time on a given grammar. This was mostly confirmed during our experiments, however the length-dependent version consistently was a few percent slower.

Concerning the different grammars predictions using G1 were faster than those for G2 by a factor of ~ 1.5. Between G2 and G3 the factor was ~ 2 and between G3 and G4 it was ~ 6.5.

4 Conclusion

We introduced an extension to the concept of stochastic context-free grammars that allows the probabilities of the productions to depend on the length of the generated subword.

Furthermore we showed that existing algorithms that work on stochastic context-free grammars like training algorithms or determining the most likely parse can easily be adapted to the new concept without significantly affecting their run-time or memory consumption.

Using the length-dependent SCFGs to predict the secondary structure of RNA molecules we found them to outperform their classical counterparts in most of the cases.

However our test of 2 versions each of 4 different grammars on 2 different sets of data did not yield a clear recommendation, which grammar to use. While G1 can be ruled out for its low prediction quality and G4 is much slower than G2 or G3 without significantly improving the quality of the predictions, the remaining 2 grammars rank close in both categories.

On the mixed set G2 without lengths yielded the best results, while also being the fastest to process, but on the tRNA set G3 with lengths clearly outperformed all of the other grammars.

4.1 Possible Other Applications

While our extension to the concept of stochastic context-free grammars stemmed from one specific application it is not application specific. The concepts and methods presented in Section 2 can immediately be applied to any other application where SCFGs are used as a model, e.g. natural language processing. From our limited insight in that field it appears possible that length-dependent grammars can successfully be applied there as well as for RNA secondary structure prediction.

4.2 Further Research

In addition to the applications, extending the concept of context-free grammars also gives rise to interesting questions in the field of formal languages. The most obvious of these questions is, if adding in length-dependencies changes the class of languages that can be generated. We have already been able to show this[6], leading us to the follow-up question what the properties of this new class of languages are. First results from this direction of research have been presented in [15].

Acknowledgements. We would like to thank the anonymous referees for their helpful suggestions.

References

1. Boyle, J., Robillard, G.T., Kim, S.: Sequential folding of transfer RNA. a nuclear magnetic resonance study of successively longer tRNA fragments with a common 5' end. J. Mol. Biol. 139, 601–625 (1980)
2. Chi, T., Geman, S.: Estimation of probabilistic context-free grammars. Computational Linguistics 24(2), 299–305 (1998)
3. Dowell, R.D., Eddy, S.R.: Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. BMC Bioinformatics 5, 71 (2004)
4. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological sequence analysis. Cambridge University Press, Cambridge (1998)
5. Furbach, F.: Earley parsing for length dependent grammars. Bachelor thesis, TU Kaiserslautern (2009)

[6] For a simple example take the grammar with the productions $S \to A,\ A \to aA \mid \epsilon$, with the first production restricted to lengths which are perfect squares. It generates $\{a^{n^2} | n \in \mathbb{N}\}$ which is not context-free.

6. Harrison, M.A.: Introduction to Formal Language Theory. Addison-Wesley, Reading (1978)
7. Knudsen, B., Hein, J.: RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. Bioinformatics 15, 446–454 (1999)
8. Meyer, I., Miklos, I.: Co-transcriptional folding is encoded within RNA genes. BMC Molecular Biology 5(1), 10 (2004)
9. Nebel, M.E.: On a statistical filter for RNA secondary structures. Technical report, Frankfurter Informatik-Berichte (May 2002)
10. Nussinov, R., Pieczenik, G., Griggs, R., Kleitmann, D.J.: Algorithms for loop matchings. SIAM Journal of Applied Mathematics 35, 68–82 (1978)
11. Prescher, D.: A tutorial on the expectation-maximization algorithm including maximum-likelihood estimation and em training of probabilistic context-free grammars (2003), http://staff.science.uva.nl/~prescher/papers/bib/2003em.prescher.pdf
12. Sprinzl, M., Vassilenko, K.S., Emmerich, J., Bauer, F.: Compilation of tRNA sequences and sequences of tRNA genes (December 20, 1999), http://www.uni-bayreuth.de/departments/biochemie/trna/
13. Stolcke, A.: An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. Computational Linguistics 21(2), 165–201 (1995)
14. Viennot, G., de Chaumont, M.: Enumeration of RNA Secondary Structures by Complexity. In: Mathematics in Biology and Medicine: Proceedings of an International Conference Held in Bari, Italy, July 18-22, 1983 (1985)
15. Weinberg, F.: Position-and-length-dependent context-free grammars. In: Theorietag Automaten und Formale Sprachen (2009)
16. Zuker, M., Stiegler, P.: Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. Nucleic Acids Res. 9, 133–148 (1981)

Chomsky-Schützenberger-Type Characterization of Multiple Context-Free Languages

Ryo Yoshinaka[1,⋆], Yuichi Kaji[2], and Hiroyuki Seki[2]

[1] Graduate School of Information Science and Technology, Hokkaido University
`ry@ist.hokudai.ac.jp`

[2] Graduate School of Information Science, Nara Institute of Science and Technology
`{kaji,seki}@is.naist.jp`

Abstract. It is a well-known theorem by Chomsky and Schützenberger (1963) that every context-free language can be represented as a homomorphic image of the intersection of a Dyck language and a regular language. This paper gives a Chomsky-Schützenberger-type characterization for multiple context-free languages, which are a natural extension of context-free languages, with introducing the notion of multiple Dyck languages, which are also a generalization of Dyck languages.

1 Introduction

A multiple context-free grammar (mcfg) is a natural extension of a context-free grammar (cfg). A nonterminal symbol in an mcfg derives tuples of strings by synchronized parallel derivation. The direct derivation relation of an mcfg is defined by a function over tuples of strings (of terminal symbols) such that each component of the function value is defined by a concatenation of some components of arguments and constant strings of terminal symbols with a linearity condition on components of arguments. Let us call such a function *linear regular*. The language generated by an mcfg is called a multiple context-free language (mcfl).

The generative power of mcfgs is properly larger than cfgs and properly smaller than context-sensitive grammars (csgs). There are several computational models that have the same generative power as mcfgs, e.g., string version of linear context-free rewriting systems, finite copying tree-to-string transducers, string generating context-free hypergraph grammars and local unordered scattered context grammars (see [2,6] for the discussion of these equivalences). Mcfgs share many properties with cfgs such as closure properties. There are other grammatical formalisms of which generative power is between cfgs and csgs such as indexed grammars. In contrast to indexed grammars, the membership problem for an mcfl is solvable in polynomial time in the length of an input string and

⋆ Supported in part by Grant-in-Aid for Young Scientists (B-20700124) and a grant from the Global COE Program "Center for Next-Generation Information Technology based on Knowledge Discovery and Knowledge Federation" from the Ministry of Education, Culture, Sports, Science and Technology of Japan.

each mcfl is semilinear. These properties are due to the synchronized parallel derivation realized by linear regular functions. Generally, each component of a tuple of strings appearing in the derivation is not adjacent with one another in the resultant string of terminal symbols. However, these components always share synchronized structure of derivation. To capture this property of mcfls, we will introduce *multiple Dyck languages* and show a theorem that is an extension of the representation theorem of cfls. It is well-known that any cfl can be represented as a homomorphic image of the intersection of a regular language and a Dyck language (Chomsky-Schützenberger theorem). A Dyck language is the set of well-nested parentheses (brackets). A multiple Dyck language is the set of 'well-nested tuples of parentheses.' The main theorem of this paper is that for a given mcfl L, there exists a multiple Dyck language D, a regular language R and a homomorphism h such that $L = h(D \cap R)$. As is the same with cfls, this representation theorem for mcfls can be easily lifted to the generator theorem.

The main results of this paper were partially published in and are partially based on Chapter 4 of [3].

2 Preliminaries

For an alphabet Σ, Σ^* denotes the set of all strings over Σ and $(\Sigma^*)^m$ denotes the set of all m-tuples of strings over Σ. The empty string is denoted by ε.

Context-Free Grammars. A *context-free grammar (cfg)* is a tuple $G = \langle \Sigma, N, P, S \rangle$, where Σ is a finite set of *terminal symbols*, N is a finite set of *nonterminal symbols*, $P \subseteq N \times (\Sigma \cup N)^*$ is a finite set of *rules*, which are denoted by $A \to \alpha$ for $A \in N$ and $\alpha \in (\Sigma \cup N)^*$, and $S \in N$ is called *the start symbol.* Elements of $P \cap (N \times \Sigma^*)$ are called *terminating rules.* $\Rightarrow_G$ and $\Rightarrow_G^*$ denote derivations of one step and any steps (including zero-step), respectively. The language generated by a cfg G, which is called a *context-free language (cfl)*, is the set $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G^* w \}$. If $P \subseteq N \times (\Sigma^* \cup \Sigma^* N)$, G is called a *right-linear grammar* and $L(G)$ is called a *regular language.*

Let $\overline{\Sigma}$ denote an alphabet disjoint from Σ that admits a bijection $\overline{(\cdot)}$ from Σ to $\overline{\Sigma}$. The *Dyck grammar* over $\Sigma \cup \overline{\Sigma}$ is the cfg that has S as its unique nonterminal symbol and whose rules are $S \to \varepsilon$ and $S \to SaS\bar{a}$ for all $a \in \Sigma$. The language generated by a Dyck grammar is called a *Dyck language.* A string on $\Sigma \cup \overline{\Sigma}$ is *well-bracketed* if it is an element of the Dyck language. An occurrence of $a \in \Sigma$ and an occurrence of $\bar{a} \in \overline{\Sigma}$ in a well-bracketed string are *corresponding* if they are derived at the same derivation step. Note that the Dyck grammar is unambiguous. According to the custom, we call elements of $\Sigma \cup \overline{\Sigma}$ *parentheses.*

Chomsky and Schützenberger [1] gave a characterization of cfls by Dyck languages.

Theorem 1. *A language L over Σ is context-free if and only if there are an alphabet Δ, a homomorphism $h : (\Delta \cup \overline{\Delta})^* \to \Sigma^*$ and a regular language R over $\Delta \cup \overline{\Delta}$ such that $L = h(D \cap R)$ where D is the Dyck language over $\Delta \cup \overline{\Delta}$.*

Theorem 1 can be stated in an even stronger (for the 'only if' direction) form:

Theorem 2. *For a given alphabet Δ, there are an alphabet Δ and a homomorphism $h : (\Delta \cup \overline{\Delta})^* \to \Sigma^*$ such that for any language L over Σ, L is context-free if and only if there is a regular language R such that $L = h(D \cap R)$ where D is the Dyck language over $\Delta \cup \overline{\Delta}$.*

Multiple Context-Free Grammars. We assume a countably infinite set X of *variables*. A function from $(\Sigma^*)^{m_1} \times \cdots \times (\Sigma^*)^{m_n}$ to $(\Sigma^*)^m$ is said to be *linear regular*, if there are $t_1, \ldots, t_m \in (\Sigma \cup \{\, x_{i,j} \in X \mid 1 \le i \le n,\ 1 \le j \le m_i \,\})^*$ such that each variable $x_{i,j}$ occurs at most once in $t_1 \ldots t_m$ and for any $\boldsymbol{w}_i = \langle w_{i,1}, \ldots, w_{i,m_i} \rangle \in (\Sigma^*)^{m_i}$ with $1 \le i \le n$, it holds that

$$f(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n) = \langle v_1, \ldots, v_m \rangle$$

where each v_k for $k = 1, \ldots, m$ is obtained from t_k by substituting $w_{i,j}$ for $x_{i,j}$ for all i and j. We simply write $f(\langle x_{1,1}, \ldots, x_{1,m_1} \rangle, \ldots, \langle x_{n,1}, \ldots, x_{n,m_n} \rangle) = \langle t_1, \ldots, t_m \rangle$ to denote the definition of f. For example, both $f(\langle x_1, x_2 \rangle) = \langle ax_1b, cx_2d \rangle$ and $g(\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle) = \langle x_1y_1, y_2x_2 \rangle$ are linear regular functions where $x_1, x_2, y_1, y_2 \in X$ and $a, b, c, d \in \Sigma$, while $h(\langle x_1 \rangle) = \langle x_1x_1 \rangle$ is not, since x_1 appears twice in the right-hand side. f is said to be *nonerasing*, if every variable in the left-hand side of the definition of f appears in the right-hand side. f is *terminal-free*, if the right-hand side of its definition contains no symbols from Σ.

An alphabet N is said to be *indexed* when we assume a function dim that assigns positive integers to symbols in N.

A *multiple context-free grammar (mcfg)* is a tuple $G = \langle \Sigma, N, F, P, S \rangle$, where

- Σ is an (unindexed) alphabet whose elements are called *terminal symbols*,
- N is an indexed alphabet whose elements are called *nonterminal symbols*,
- F is a finite set of linear regular functions,
- P is a finite set of *rules* of the form $A \to f(B_1, \ldots, B_n)$ where $A, B_1, \ldots, B_n \in N$ and $f : (\Sigma^*)^{\dim(B_1)} \times \cdots \times (\Sigma^*)^{\dim(B_n)} \to (\Sigma^*)^{\dim(A)} \in F$,
- $S \in N$ is called *the start symbol* whose dimension is 1.

For a rule $\pi = A \to f(B_1, \ldots, B_n)$, the *head* and the *body* of π refer to A and $f(B_1, \ldots, B_n)$, respectively, and the *rank* of π is defined to be $\mathrm{rank}(\pi) = n$. If $\mathrm{rank}(\pi) = 0$ and $f() = \boldsymbol{w}$, we simply write $A \to \boldsymbol{w}$ for π with suppressing f. If f is terminal-free, π is also said to be *terminal-free.*

For each $A \in N$, $L_G(A)$ is recursively defined as the smallest set of $\dim(A)$-tuples of strings satisfying that if $A \to f(B_1, \ldots, B_n) \in P$ and $\boldsymbol{w}_i \in L_G(B_i)$ for $i = 1, \ldots, n$, then $f(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n) \in L_G(A)$. The *language $L(G)$ generated by G* is the set $\{\, w \in \Sigma^* \mid \langle w \rangle \in L_G(S) \,\}$. $L(G)$ is called a *multiple context-free language (mcfl).* Two grammars G and G' are *equivalent* if $L(G) = L(G')$.

Example 1. Let G_1 be the mcfg $\langle \Sigma_1, N_1, F_1, P_1, S \rangle$ such that $\Sigma_1 = \{a, b, c, d\}$, $N_1 = \{S, A, B\}$ with $\dim(S) = 1$, $\dim(A) = \dim(B) = 2$, F_1 consists of e, f, g and the constant functions appearing in the body of rules in P_1 below where $e(\langle x_1, x_2 \rangle, \langle y_1, y_2 \rangle) = \langle x_1y_1x_2y_2 \rangle$, $f(\langle x_1, x_2 \rangle) = \langle ax_1, bx_2 \rangle$, $g(\langle x_1, x_2 \rangle) =$

$\langle cx_1, dx_2 \rangle$, and $P_1 = \{S \to e(A, B),\ A \to f(A),\ A \to \langle a, b \rangle,\ B \to g(B),\ B \to \langle c, d \rangle\}$. Let us call the rules in P_1 $\pi_1, \pi_2, \ldots, \pi_5$ in the order written above. For example, $\langle a, b \rangle \in L_{G_1}(A)$ by π_3, $\langle aa, bb \rangle \in L_{G_1}(A)$ by π_2, $\langle c, d \rangle \in L_{G_1}(B)$ by π_5 and $\langle aacbbd \rangle \in L_{G_1}(S)$ by π_1. We have $L(G_1) = \{\, a^m c^n b^m d^n \mid m, n \geq 0 \,\}$.

For a nonterminal symbol A of an mcfg G, a series of rule application steps to obtain a tuple of strings of terminal symbols $\boldsymbol{w} \in L_G(A)$ is called a *derivation* of $\boldsymbol{w}$ in G.

By q-MCFG(r) we denote the collection of mcfgs G such that $\dim(A) \leq q$ for all $A \in N$ and $\mathrm{rank}(\pi) \leq r$ for all $\pi \in P$. q-MCFL(r) is the class of mcfls generated by grammars in q-MCFG(r).

G is said to be *nonerasing*, if all $f \in F$ are nonerasing. It is known that every $G \in q$-MCFG(r) has an equivalent nonerasing grammar in q-MCFG(r) [8]. Grammars from 1-MCFG(r) are identified with cfgs.

Proposition 1 (Seki et al. [8] and Rambow and Satta [6]). *For $q \geq 1$, q-MCFL(r) $\subsetneq$ ($q+1$)-MCFL(r). For $q \geq 2$, $r \geq 1$, q-MCFL(r) $\subsetneq$ q-MCFL($r+1$) except for* 2-MCFL(2) = 2-MCFL(3). *For $q \geq 1$, $r \geq 3$ and $1 \leq k \leq r-2$, q-MCFL(r) $\subseteq$ ($k+1$)q-MCFL($r-k$).*

Proposition 2 (Rambow and Satta [6]). *Each family q-MCFL(r) for $r \geq 2$ is a substitution closed full AFL. That is, they are closed under homomorphism, inverse homomorphism, intersection with regular languages, union, concatenation, the Kleene plus and substitution.*

Proposition 3 (Seki et al. [8]). *Let $G \in q$-MCFG(r) be given. It is decidable in $O(|w|^{q(r+1)})$ time whether $w \in L(G)$ for any $w \in \Sigma^*$.*

Multiple Dyck Languages. Let q and r be fixed. We define the notion of the multiple Dyck language in q-MCFL(r) on an indexed alphabet, where we assume that the maximum dimension of elements of the indexed alphabet does not exceed r. For an indexed alphabet Δ, let

$$\widehat{\Delta} = \{\, a^{[i]}, \bar{a}^{[i]} \mid a \in \Delta,\ 1 \leq i \leq \dim(a) \,\}.$$

Definition 1. The *multiple Dyck grammar D_Δ* on an indexed alphabet Δ is the mcfg that has nonterminal symbols S_m with $\dim(S_m) = m$ for $m \leq q$, among which the start symbol is S_1, and that has rules of the following three types:

1. all the possible terminal-free rules allowed in q-MCFG(r);
2. rules of the form $S_m \to f_a(S_m)$ where $f_a(\langle x_1, \ldots, x_m \rangle) = \langle a^{[1]} x_1 \bar{a}^{[1]}, \ldots, a^{[m]} x_m \bar{a}^{[m]} \rangle$ for $a \in \Delta$ with $\dim(a) = m$;
3. rules of the form $S_m \to f(S_m)$ with $f(\langle x_1, \ldots, x_m \rangle) = \langle t_1, \ldots, t_m \rangle$ where each t_i is either x_i, $x_i a^{[1]} \bar{a}^{[1]}$ or $a^{[1]} \bar{a}^{[1]} x_i$ for some $a \in \Delta$ of dimension 1.

The language $L(D_\Delta)$ is called the *multiple Dyck language* over $\Delta \cup \overline{\Delta}$.

We note that rules of type 3 are redundant if $r > 1$. If $q = 1$ and $r > 1$, $L(D_\Delta)$ is indeed the (context-free) Dyck language over $\Delta \cup \overline{\Delta}$.

Every element of tuples in $L_{D_\Delta}(S_m)$ is well-bracketed. Moreover pairs of corresponding parentheses in a string from $L(D_\Delta)$ are partitioned into groups each of which consists of exactly $\langle a^{[1]}, \bar{a}^{[1]} \rangle, \dots, \langle a^{[\dim(a)]}, \bar{a}^{[\dim(a)]} \rangle$ for some $a \in \Delta$. If some member of such a group A is inside some member of a group B, then all members from A are inside some member of B. For example, $b^{[1]} a^{[1]} \bar{a}^{[1]} a^{[2]} \bar{a}^{[2]} \bar{b}^{[1]} b^{[2]} a^{[3]} \bar{a}^{[3]} \bar{b}^{[2]}$ is allowed, while $b^{[1]} a^{[1]} \bar{a}^{[1]} \bar{b}^{[1]} a^{[2]} \bar{a}^{[2]} b^{[2]} a^{[3]} \bar{a}^{[3]} \bar{b}^{[2]}$ is not, where $\dim(a) = 3$ and $\dim(b) = 2$. The way of combining parentheses is restricted by available means in q-MCFG(r).

3 Theorem

This section discusses Chomsky-Schützenberger type characterization of mcfls.

3.1 Informal Example of Construction

We first review an idea of the proof of Theorem 1 by using a simple example. Let G_0 be the cfg $\langle \Sigma_0, N_0, P_0, S \rangle$ where $\Sigma_0 = \{a, b, c\}$, $N_0 = \{S, A, B\}$, $P_0 = \{S \to aA,\ A \to bAB,\ A \to a,\ B \to c\}$. We call the four rules π_1, π_2, π_3 and π_4 in the order as written above. Let $\Delta = \{ [\![_{\pi_1,1}, [\![_{\pi_2,1}, [\![_{\pi_2,2}, [\![_a, [\![_b, [\![_c \}$ and let us write $]\!]_x$ to denote $\overline{[\![_x}$ for each $[\![_x \in \Delta$. Also let $h : (\Delta \cup \overline{\Delta})^* \to \Sigma_0^*$ be the homomorphism defined by $h([\![_x) = x$ for $x \in \Sigma_0$ and $h(z) = \varepsilon$ for other $z \in \Delta \cup \overline{\Delta}$. Figure 1 shows an example of a derivation tree (called t_0) in G_0. Intuitively, $[\![_{\pi,i}$ and $]\!]_{\pi,i}$ mean the left end and the right end of a derivation

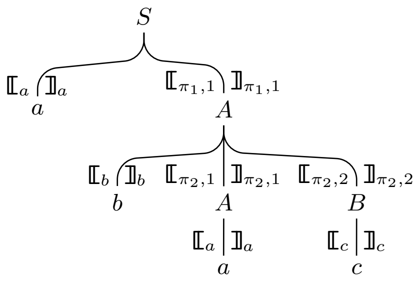

S
$[\![_a$ $]\!]_a$
a
$[\![_{\pi_1,1}$ $]\!]_{\pi_1,1}$
A
$[\![_b$ $]\!]_b$
b
$[\![_{\pi_2,1}$ $]\!]_{\pi_2,1}$
A
$[\![_{\pi_2,2}$ $]\!]_{\pi_2,2}$
B
$[\![_a$ $]\!]_a$
a
$[\![_c$ $]\!]_c$
c

Fig. 1. A derivation tree in G_0

starting from the i-th nonterminal symbol in the body of the rule π. For $x \in \Sigma_0$, a pair $[\![_x$ and $]\!]_x$ denotes x. In the figure, paired symbols in $\Delta \cup \overline{\Delta}$ are placed on the left-side and the right-side of each edge. For a tree t, let $\alpha(t)$ denote the string over $\Delta \cup \overline{\Delta}$ obtained by concatenating these labels in the depth-first left-to-right order. For example,

$$\alpha(t_0) = [\![_a\ ]\!]_a\ [\![_{\pi_1,1}\ [\![_b\ ]\!]_b\ [\![_{\pi_2,1}\ [\![_a\ ]\!]_a\ ]\!]_{\pi_2,1}\ [\![_{\pi_2,2}\ [\![_c\ ]\!]_c\ ]\!]_{\pi_2,2}\ ]\!]_{\pi_1,1}$$

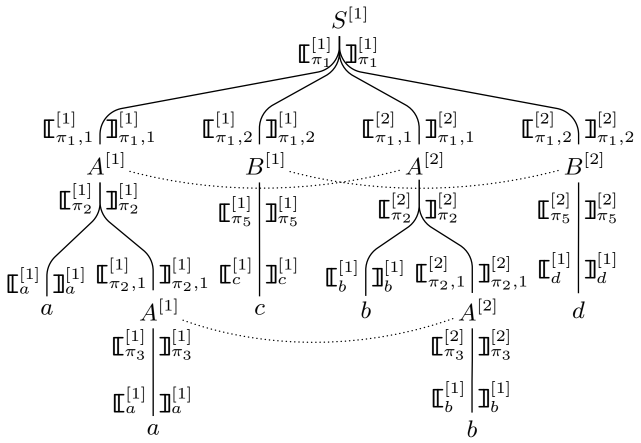


Fig. 2. A derived tree in G_1

for t_0 in the figure. For a tree t, let yield(t) denote the string obtained by concatenating the labels of leaf nodes of t from left to right. Then, yield$(t) = h(\alpha(t))$ for a derivation tree t in G_0 and $L(G_0) = h(\{\, \alpha(t) \mid t \text{ is a derivation tree in } G_0 \,\})$. Therefore, what we should do is to construct a right-linear grammar G_{R_0} such that $L(G_{R_0}) \cap D = \{\, \alpha(t) \mid t \text{ is a derivation tree in } G_0 \,\}$ in this particular example where D is the Dyck language over $\Delta \cup \overline{\Delta}$. G_{R_0} can be defined by considering the finite-state tree traversal that emits $[\![_x$ and $]\!]_x$ when it visits $x \in \Sigma_0$, emits $[\![_{\pi,i}$ when it visits the i-th nonterminal symbol in the body of π, and emits $]\!]_{\pi,i}$ when it returns from that nonterminal symbol. Note that nonterminal symbols in N_0 are used as 'finite states' (nonterminal symbols of G_{R_0}) when the traversal goes down while a new nonterminal symbol T is used when it goes up.

$$\begin{array}{lll}
S \to [\![_a \;]\!]_a \; [\![_{\pi_1,1} \; A & T \to \;]\!]_{\pi_1,1} \; T & \\
A \to [\![_b \;]\!]_b \; [\![_{\pi_2,1} \; A & T \to \;]\!]_{\pi_2,1} \; [\![_{\pi_2,2} \; B & T \to \;]\!]_{\pi_2,2} \; T \\
A \to [\![_a \;]\!]_a \; T \quad B \to [\![_c \;]\!]_c \; T & & \\
T \to \varepsilon & &
\end{array}$$

A similar idea can be applied to mcfg. Let G_1 be the mcfg from Example 1. Figure 2 shows a tree that illustrates the derivation in the example. (This kind of tree is called a *derived tree* in mcfg. Here we use derived tree without formal definition since derived tree is not needed in the formal proofs in the rest of this paper.) In the figure, $A^{[j]}$ $(j = 1, 2)$ denotes a (hypothetical) nonterminal symbol that derives the j-th component w_j of $\langle w_1, w_2 \rangle \in L_{G_1}(A)$. $S^{[1]}$, $B^{[1]}$ and $B^{[2]}$ are used in the same purpose. A horizontal arc between $A^{[1]}$ and $A^{[2]}$ means that these two nodes together represent an instance of A in the derivation. Let

$$\Gamma = \{\pi_1, \pi_2, \ldots, \pi_5, \langle \pi_1, 1\rangle, \langle \pi_1, 2\rangle, \langle \pi_2, 1\rangle, \langle \pi_4, 1\rangle, a, b, c, d\}.$$

The symbol $\langle \pi, i \rangle$ $(1 \le i \le \text{rank}(\pi))$ corresponds to the i-th nonterminal symbol in the body of the rule π. For example, $\langle \pi_1, 1\rangle$ and $\langle \pi_1, 2\rangle$ correspond to A

and B, respectively. For each $\pi \in P_1$, define $\dim(\pi)$ to be the dimension of the head of π. For each $\pi \in P_1$ and i $(1 \le i \le \mathrm{rank}(\pi))$, define $\dim(\langle \pi, i \rangle)$ to be the dimension of the i-th nonterminal symbol in the body of π. For each $x \in \Sigma_1$, define $\dim(x) = 1$. Thus, $\dim(\pi_1) = \dim(a) = \cdots = \dim(d) = 1$ and $\dim(x) = 2$ for other $x \in \Gamma$. We abbreviate symbols in $\widehat{\Gamma}$ as $\llbracket^{[1]}_{\pi_1}$, $\rrbracket^{[1]}_{\pi_1}$, $\llbracket^{[1]}_{\pi_2}$, $\rrbracket^{[1]}_{\pi_2}$, $\llbracket^{[2]}_{\pi_2}$, $\rrbracket^{[2]}_{\pi_2}$, ..., $\llbracket^{[1]}_{\pi_1,1}$, $\rrbracket^{[1]}_{\pi_1,1}$, $\llbracket^{[2]}_{\pi_1,1}$, $\rrbracket^{[2]}_{\pi_1,1}$, ... Similarly to cfg's case, $\llbracket^{[j]}_{\pi,i}$ (rsp. $\rrbracket^{[j]}_{\pi,i}$) denotes the left (rsp. right) end of a derivation for the j-th component of the i-th nonterminal symbol in the body of π. For the mcfg G_1, we also have $L(G_1) = h(\{\, \alpha(t) \mid t \text{ is a `derived tree' in } G_1 \,\})$ where $h$ is defined similarly to cfg's case. Hence, it suffices to give a right-linear grammar $G_{R_1}$ such that $L(G_{R_1}) \cap L(D_\Gamma) = \{\, \alpha(t) \mid t \text{ is a `derived tree' in } G_1 \,\}$. The construction of G_{R_1} is a little cumbersome but not difficult:

$$
\begin{array}{lll}
S^{[1]} \to \llbracket^{[1]}_{\pi_1} \llbracket^{[1]}_{\pi_1,1} A^{[1]} & T \to \rrbracket^{[1]}_{\pi_1,1} \llbracket^{[1]}_{\pi_1,2} B^{[1]} & T \to \rrbracket^{[1]}_{\pi_1,2} \llbracket^{[2]}_{\pi_1,1} A^{[2]} \\
T \to \rrbracket^{[2]}_{\pi_1,1} \llbracket^{[2]}_{\pi_1,2} B^{[2]} & T \to \rrbracket^{[2]}_{\pi_1,2} \rrbracket^{[1]}_{\pi_1} T & \\
A^{[1]} \to \llbracket^{[1]}_{\pi_2} \llbracket^{[1]}_{a} \rrbracket^{[1]}_{a} \llbracket^{[1]}_{\pi_2,1} A^{[1]} & T \to \rrbracket^{[1]}_{\pi_2,1} \rrbracket^{[1]}_{\pi_2} T & \\
A^{[2]} \to \llbracket^{[2]}_{\pi_2} \llbracket^{[1]}_{b} \rrbracket^{[1]}_{b} \llbracket^{[2]}_{\pi_2,1} A^{[2]} & T \to \rrbracket^{[2]}_{\pi_2,1} \rrbracket^{[2]}_{\pi_2} T & \\
A^{[1]} \to \llbracket^{[1]}_{\pi_3} \llbracket^{[1]}_{a} \rrbracket^{[1]}_{a} \rrbracket^{[1]}_{\pi_3} T & A^{[2]} \to \llbracket^{[2]}_{\pi_3} \llbracket^{[1]}_{b} \rrbracket^{[1]}_{b} \rrbracket^{[2]}_{\pi_3} T & \\
B^{[1]} \to \llbracket^{[1]}_{\pi_4} \llbracket^{[1]}_{c} \rrbracket^{[1]}_{c} \llbracket^{[1]}_{\pi_4,1} B^{[1]} & T \to \rrbracket^{[1]}_{\pi_4,1} \rrbracket^{[1]}_{\pi_4} T & \\
B^{[2]} \to \llbracket^{[2]}_{\pi_4} \llbracket^{[1]}_{d} \rrbracket^{[1]}_{d} \llbracket^{[2]}_{\pi_4,1} B^{[2]} & T \to \rrbracket^{[2]}_{\pi_4,1} \rrbracket^{[2]}_{\pi_4} T & \\
B^{[1]} \to \llbracket^{[1]}_{\pi_5} \llbracket^{[1]}_{c} \rrbracket^{[1]}_{c} \rrbracket^{[1]}_{\pi_5} T & B^{[2]} \to \llbracket^{[2]}_{\pi_5} \llbracket^{[1]}_{d} \rrbracket^{[1]}_{d} \rrbracket^{[2]}_{\pi_5} T & \\
T \to \varepsilon \;. & &
\end{array}
$$

3.2 Formal Construction

Let us arbitrarily fix positive integers q and r. We now give our Chomsky-Schützenberger type characterization for q-MCFL(r). Without loss of generality, we may assume that any $G \in q$-MCFG(r) satisfies the following conditions:

- G is nonerasing;
- if G has a rule $A \to f(B_1, \ldots, B_n)$ and $1 \le i < j \le n$, then $B_i \ne B_j$.

Indeed every mcfg in q-MCFG(r) has an equivalent one in q-MCFG(r) with this property.

Let $G = \langle \Sigma, N, F, P, S \rangle \in q$-MCFG$(r)$ be given. Our goal is to find an indexed alphabet Δ, a right-linear grammar R over $\widehat{\Delta}$, and a homomorphism $h : \widehat{\Delta}^* \to \Sigma^*$ such that $L(G) = h(L(D_\Delta) \cap L(R))$.

Let

$$\Delta = \{\, \llbracket_a \mid a \in \Sigma \,\} \cup \{\, \llbracket_\pi \mid \pi \in P \,\} \cup \{\, \llbracket_{\pi,i} \mid 1 \le i \le \mathrm{rank}(\pi),\ \pi \in P \,\}$$

where $\dim(\llbracket_a) = 1$ for $a \in \Sigma$, $\dim(\llbracket_\pi) = \dim(A)$ and $\dim(\llbracket_{\pi,i}) = \dim(B_i)$ if $\pi \in P$ is of the form $A \to f(B_1, \ldots, B_n)$. Hereafter we write $\rrbracket_*$ instead of $\overline{\llbracket_*}$ for each $\overline{\llbracket_*} \in \overline{\Delta}$. By $\widetilde{(\cdot)}$ we denote the homomorphism from Σ^* to $\widehat{\Delta}^*$ such that $\widetilde{a} = \llbracket^{[1]}_a \rrbracket^{[1]}_a$.

The nonterminal symbols of the right-linear grammar R is

$$\{\, T \,\} \cup \{\, A^{[k]} \mid A \in N,\ 1 \le k \le \dim(A) \,\}$$

and the start symbol is $S^{[1]}$. The rules of R are given as follows. Suppose that G has a rule π of the form $A \to f(B_1, \dots, B_n)$ and f is represented as

$$\begin{aligned} & f(\langle x_{1,1}, \dots, x_{1,m_1} \rangle, \dots, \langle x_{n,1}, \dots, x_{n,m_n} \rangle) = \langle t_1, \dots, t_m \rangle \\ & \quad \text{where } t_k = u_{k,0} x_{i_{k1}, j_{k1}} u_{k,1} \dots x_{i_{kp_k}, j_{kp_k}} u_{k,p_k} \text{ with } u_{k,0}, \dots, u_{k,p_k} \in \Sigma^* \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } k = 1, \dots, m. \end{aligned}$$

For each $k = 1, \dots, m$, if $p_k = 0$, then R has the rule

$$A^{[k]} \to \llbracket_{\pi}^{[k]} \, \widetilde{u}_{k,0} \, \rrbracket_{\pi}^{[k]} \, T$$

and otherwise, R has the following $p_k + 1$ rules:

$$\begin{aligned} A^{[k]} &\to \llbracket_{\pi}^{[k]} \, \widetilde{u}_{k,0} \, \llbracket_{\pi, i_{k1}}^{[j_{k1}]} \, B_{i_{k1}}^{[j_{k1}]}, \\ T &\to \rrbracket_{\pi, i_{k(l-1)}}^{[j_{k(l-1)}]} \, \widetilde{u}_{k,l-1} \, \llbracket_{\pi, i_{kl}}^{[j_{kl}]} \, B_{i_{kl}}^{[j_{kl}]} \text{ for } 1 < l \le p_k, \\ T &\to \rrbracket_{\pi, i_{kp_k}}^{[j_{kp_k}]} \, \widetilde{u}_{k,p_k} \, \rrbracket_{\pi}^{[k]} \, T. \end{aligned}$$

Moreover R has

$$T \to \varepsilon,$$

which is the unique terminating rule of R.

We define the homomorphism $h : \widehat{\Delta}^* \to \Sigma^*$ so that for $z \in \widehat{\Delta}$,

$$h(z) = \begin{cases} a & \text{if } z = \llbracket_a^{[1]} \text{ for some } a \in \Sigma; \\ \varepsilon & \text{otherwise.} \end{cases}$$

3.3 Correctness

Lemma 1. *$L(G) \subseteq h(L(R) \cap L(D_\Delta))$.*

Proof. By induction we show that if $\langle w_1, \dots, w_m \rangle \in L_G(A)$, then there are $v_1, \dots, v_m \in \widehat{\Delta}^*$ such that $\langle v_1, \dots, v_m \rangle \in L_{D_\Delta}(S_m)$ and $A^{[k]} \Rightarrow_R^* v_k$ and $h(v_k) = w_k$ for each $k = 1, \dots, m$.

Suppose that $\langle w_1, \dots, w_m \rangle \in L_G(A)$ due to $\pi = A \to f(B_1, \dots, B_n) \in P$ and $\langle w_{i,1}, \dots, w_{i,m_i} \rangle \in L_G(B_i)$ for $i = 1, \dots, n$ where $f(\langle w_{1,1}, \dots, w_{1,m_1} \rangle, \dots, \langle w_{n,1}, \dots, w_{n,m_n} \rangle) = \langle w_1, \dots, w_m \rangle$. Note that the case of $n = 0$ provides the basis of the induction.

The induction hypothesis says that for each $i = 1, \dots, n$ we have $v_{i,1}, \dots, v_{i,m_i} \in \widehat{\Delta}^*$ such that $\langle v_{i,1}, \dots, v_{i,m_i} \rangle \in L_{D_\Delta}(S_{m_i})$, $h(v_{i,j}) = w_{i,j}$ and $B_i^{[j]} \Rightarrow_R^* v_{i,j}$

for $j = 1, \ldots, m_i$, where we have $B_i^{[j]} \Rightarrow_R^* v_{i,j} T \Rightarrow_R v_{i,j}$ because $T \to \varepsilon$ is the unique terminating rule of R. Let us represent f as

$$f(\langle x_{1,1}, \ldots, x_{1,m_1}\rangle, \ldots, \langle x_{n,1}, \ldots, x_{n,m_n}\rangle) = \langle t_1, \ldots, t_m\rangle$$
$$\text{where } t_k = u_{k,0} x_{i_{k1},j_{k1}} u_{k,1} \ldots x_{i_{kp_k},j_{kp_k}} u_{k,p_k} \text{ with } u_{k,0}, \ldots, u_{k,p_k} \in \Sigma^* \quad \text{for } k = 1, \ldots, m.$$

We define v_k by

$$v_k = \llbracket_\pi^{[k]} \widetilde{u}_{k,0} \llbracket_{\pi,i_{k1}}^{[j_{k1}]} v_{i_{k1},j_{k1}} \rrbracket_{\pi,i_{k1}}^{[j_{k1}]} \widetilde{u}_{k,1} \ldots \llbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]} v_{i_{kp_k},j_{kp_k}} \rrbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]} \widetilde{u}_{k,p_k} \rrbracket_\pi^{[k]} . \quad (1)$$

It is easy to see that for each k, $h(v_k) = w_k$ and $A^{[k]} \Rightarrow_R^* v_k$ by $B_i^{[j]} \Rightarrow_R^* v_{i,j} T$.

Hence it is enough to show that $\langle v_1, \ldots, v_m\rangle \in L_{D_\Delta}(S_m)$. Let

$$v'_{i_{kl},j_{kl}} = \llbracket_{\pi,i_{kl}}^{[j_{kl}]} v_{i_{kl},j_{kl}} \rrbracket_{\pi,i_{kl}}^{[j_{kl}]} \widetilde{u}_{k,l}, \quad (2)$$
$$v'_k = v'_{i_{k1},j_{k1}} \ldots v'_{i_{kp_k},j_{kp_k}} \quad (3)$$

for $l = 1, \ldots, p_k$ and $k = 1, \ldots, m$. By (1), (2), (3),

$$v_k = \llbracket_\pi^{[k]} \widetilde{u}_{k,0} v'_k \rrbracket_\pi^{[k]} . \quad (4)$$

Applying appropriate rules of type 2 and type 3 of Definition 1 to

$$\langle v_{i,1}, \ldots, v_{i,m_i}\rangle \in L_{D_\Delta}(S_{m_i}),$$

for $i = 1, \ldots, n$, we have

$$\langle v'_{i,1}, \ldots, v'_{i,m_i}\rangle \in L_{D_\Delta}(S_{m_i})$$

by (2). Applying to those the rule $S_m \to f'(S_{m_1}, \ldots, S_{m_n})$ of type 1 where f' is obtained by removing all the occurrences of terminal symbols in the definition of f, we get $\langle v'_1, \ldots, v'_m\rangle \in L_{D_\Delta}(S_m)$ by (3). By (4), appropriate rules of type 3 and type 2 provide

$$\langle v_1, \ldots, v_m\rangle \in L_{D_\Delta}(S_m).$$ □

Lemma 2. *Suppose that $A^{[k]} \Rightarrow_R^* w$ and w is well-bracketed. Then there is a rule $\pi \in P$ such that the head of π is A and the outermost parentheses of w are just $\llbracket_\pi^{[k]}$, $\rrbracket_\pi^{[k]}$.*

Lemma 3. *$h(L(R) \cap L(D_\Delta)) \subseteq L(G)$.*

Proof. We show by induction that whenever $\langle w_1, \ldots, w_m\rangle \in L_{D_\Delta}(S_m)$ and $A^{[k]} \Rightarrow_R^* w_k$ for $k = 1, \ldots, m$ where $m = \dim(A)$, we have $\langle h(w_1), \ldots, h(w_m)\rangle \in L_G(A)$.

Let us consider the derivation of w_k in R. By Lemma 2, each w_k has the form $w_k = \llbracket_{\pi_k}^{[k]} w'_k \rrbracket_{\pi_k}^{[k]}$ for some rule $\pi_k \in P$ and $w'_k \in \widehat{\Delta}^*$. The outermost parentheses of $\langle w_1, \ldots, w_m\rangle$ are exactly $\llbracket_{\pi_1}^{[1]}, \rrbracket_{\pi_1}^{[1]}, \ldots, \llbracket_{\pi_m}^{[m]}, \rrbracket_{\pi_m}^{[m]}$ and thus $\langle w_1, \ldots, w_m\rangle \in$

$L_{D_\Delta}(S_m)$ implies that $\pi_1 = \pi_2 = \cdots = \pi_m$. We may hereafter omit the subscript of π_k as π. Let π be $A \to f(B_1, \ldots, B_n)$ and f represented as

$$\begin{aligned} &f(\langle x_{1,1}, \ldots, x_{1,m_1}\rangle, \ldots, \langle x_{n,1}, \ldots, x_{n,m_n}\rangle) = \langle t_1, \ldots, t_m\rangle \\ &\quad \text{where } t_k = u_{k,0} x_{i_{k1},j_{k1}} u_{k,1} \ldots x_{i_{kp_k},j_{kp_k}} u_{k,p_k} \text{ with } u_{k,0}, \ldots, u_{k,p_k} \in \Sigma^* \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{for } k = 1, \ldots, m. \end{aligned} \tag{5}$$

If $p_k = 0$, the only rule of R that derives $\llbracket_\pi^{[k]}$ is $A^{[k]} \to \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \rrbracket_\pi^{[k]}\, T$ and thus $w_k = \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \rrbracket_\pi^{[k]}$. If $p_k \geq 1$, we have

$$A^{[k]} \underset{R}{\Rightarrow} \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, B_{i_{k1}}^{[j_{k1}]} \overset{*}{\underset{R}{\Rightarrow}} w_k.$$

Corresponding to the occurrence of $\llbracket_{\pi,i_{k1}}^{[j_{k1}]}$, $\rrbracket_{\pi,i_{k1}}^{[j_{k1}]}$ must occur in w_k. The only rule that provides $\rrbracket_{\pi,i_{k1}}^{[j_{k1}]}$ is $T \to \rrbracket_{\pi,i_{k1}}^{[j_{k1}]}\, \widetilde{u}_{k,1}\, \llbracket_{\pi,i_{k2}}^{[j_{k2}]}\, B_{i_{k2}}^{[j_{k2}]}$ unless $p_k = 1$. Thus

$$\begin{aligned} A^{[k]} &\underset{R}{\Rightarrow} \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, B_{i_{k1}}^{[j_{k1}]} \overset{*}{\underset{R}{\Rightarrow}} \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, v_{k,1} T \\ &\underset{R}{\Rightarrow} \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, v_{k,1}\, \rrbracket_{\pi,i_{k1}}^{[j_{k1}]}\, \widetilde{u}_{k,1}\, \llbracket_{\pi,i_{k2}}^{[j_{k2}]}\, B_{i_{k2}}^{[j_{k2}]} \overset{*}{\underset{R}{\Rightarrow}} w_k. \end{aligned}$$

for some $v_{k,1}$, which must be well-bracketed. Then we need $\rrbracket_{\pi,i_{k2}}^{[j_{k2}]}$ corresponding to the occurrence of $\llbracket_{\pi,i_{k2}}^{[j_{k2}]}$. Repeatedly applying this discussion, we finally get

$$A^{[k]} \overset{*}{\underset{R}{\Rightarrow}} \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, v_{k,1}\, \rrbracket_{\pi,i_{k1}}^{[j_{k1}]}\, \widetilde{u}_{k,1} \ldots \llbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]}\, v_{k,p_k}\, \rrbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]}\, \widetilde{u}_{k,p_k}\, \rrbracket_\pi^{[k]}\, T \overset{*}{\underset{R}{\Rightarrow}} w_k.$$

This holds for any $p_k \geq 1$. By Lemma 2

$$w_k = \llbracket_\pi^{[k]}\, \widetilde{u}_{k,0}\, \llbracket_{\pi,i_{k1}}^{[j_{k1}]}\, v_{k,1}\, \rrbracket_{\pi,i_{k1}}^{[j_{k1}]}\, \widetilde{u}_{k,1} \ldots \llbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]}\, v_{k,p_k}\, \rrbracket_{\pi,i_{kp_k}}^{[j_{kp_k}]}\, \widetilde{u}_{k,p_k}\, \rrbracket_\pi^{[k]}.$$

Let $w_{i,j} = v_{k,l}$ if $x_{i,j}$ occurs as the l-th variable in t_k, i.e., $w_{i_{kl},j_{kl}} = v_{k,l}$. We note that $B_i^{[j]} \Rightarrow_R^* w_{i,j} T \Rightarrow_R w_{i,j}$ and

$$h(w_k) = u_{k,0} h(w_{i_{k1},j_{k1}}) u_{k,1} \ldots h(w_{i_{kp_k},j_{kp_k}}) u_{k,p_k}. \tag{6}$$

Applying Lemma 2 to each $w_{i,j}$, which must be well-bracketed, we have $w_{i,j} = \llbracket_{\rho_{i,j}}^{[j]}\, w'_{i,j}\, \rrbracket_{\rho_{i,j}}^{[j]}$ for some rule $\rho_{i,j}$ of G. Here the third outermost parentheses of $\langle w_1, \ldots, w_m\rangle$ consist of exactly $\sum_{1 \leq i \leq n} m_i$ pairs

$$\langle\, \llbracket_{\rho_{i,1}}^{[1]}, \rrbracket_{\rho_{i,1}}^{[1]} \,\rangle, \ldots, \langle\, \llbracket_{\rho_{i,m_i}}^{[m_i]}, \rrbracket_{\rho_{i,m_i}}^{[m_i]} \,\rangle \text{ for } i = 1, \ldots, n.$$

By $\langle w_1, \ldots, w_m\rangle \in L_{D_\Delta}(S_m)$, for each $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$, all of

$$\langle\, \llbracket_{\rho_{i,j}}^{[1]}, \rrbracket_{\rho_{i,j}}^{[1]} \,\rangle, \ldots, \langle\, \llbracket_{\rho_{i,j}}^{[m_i]}, \rrbracket_{\rho_{i,j}}^{[m_i]} \,\rangle$$

must occur as third outermost parentheses in $\langle w_1, \dots, w_m \rangle$. Recall that the head of the rule $\rho_{i,j}$ is B_i and that $B_i = B_{i'}$ implies $i = i'$. Hence $i \neq i'$ implies $\rho_{i,j} \neq \rho_{i',j'}$ for any j and j'. Therefore for any i, j, it holds that

$$\langle \llbracket^{[1]}_{\rho_{i,1}}, \rrbracket^{[1]}_{\rho_{i,1}} \rangle = \langle \llbracket^{[1]}_{\rho_{i,j}}, \rrbracket^{[1]}_{\rho_{i,j}} \rangle, \dots, \langle \llbracket^{[m_i]}_{\rho_{i,m_i}}, \rrbracket^{[m_i]}_{\rho_{i,m_i}} \rangle = \langle \llbracket^{[m_i]}_{\rho_{i,j}}, \rrbracket^{[m_i]}_{\rho_{i,j}} \rangle$$

and we have ρ_i such that $\rho_i = \rho_{i,1} = \cdots = \rho_{i,m_i}$. In the dervation of $\langle w_1, \dots, w_m \rangle \in L_{D_\Delta}(S_m)$, at some point the rule $S_{m_i} \to f_{\llbracket_{\rho_i}}(S_{m_i})$ of type 2, where $f_{\llbracket_{\rho_i}}(\langle x_1, \dots, x_m \rangle) = \langle \llbracket^{[1]}_{\rho_i} x_1 \rrbracket^{[1]}_{\rho_i}, \dots, \llbracket^{[m_i]}_{\rho_i} x_m \rrbracket^{[m_i]}_{\rho_i} \rangle$, must be applied to $\langle w_{i,1}, \dots, w_{i,m_i} \rangle \in L_{D_\Delta}(S_{m_i})$. By the induction hypothesis, we have $\langle h(w_{i,1}), \dots, h(w_{i,m_i}) \rangle \in L_G(B_i)$ for $i = 1, \dots, n$. Applying the rule π to those tuples, we obtain by (5) and (6)

$$\begin{aligned} &f(\langle h(w_{1,1}), \dots, h(w_{1,m_1}) \rangle, \dots, \langle h(w_{n,1}), \dots, h(w_{n,m_n}) \rangle) \\ &\qquad = \langle h(w_1), \dots, h(w_m) \rangle \in L_G(A). \end{aligned}$$ □

Theorem 3. *A language L is in q-MCFL(r) if and only if there are a multiple Dyck language $D \in q$-MCFL(r), a regular language R and a homomorphism h such that*

$$L = h(D \cap R).$$

Proof. By Lemmas 1 and 3 and Proposition 2. □

3.4 Generator Theorem

It is easy to get the stronger Chomsky-Schützenberger-type characterization for q-MCFL(r) by the standard technique.

Let

$$\Delta' = \{ \llbracket_a \mid a \in \Sigma \} \cup \{ \llbracket_m, [_m \mid 1 \leq m \leq q \}$$

where $\dim(a) = 1$ and $\dim(\llbracket_m) = \dim([_m) = m$ and $h' : \widehat{\Delta'}^* \to \Sigma^*$ be the homomorphism mapping each $\llbracket_a$ to a for $a \in \Sigma$ and other symbols to the empty string.

For a given mcfg $G \in q$-MCFG(r), let Δ and R be the indexed alphabet and the right-linear grammar from Section 3.2, respectively. Let us enumerate all the elements of dimension m in $\Delta \setminus \{ \llbracket_a \mid a \in \Sigma \}$ and denote them by $\llbracket_{m,1}, \dots, \llbracket_{m,k_m}$ for each m. We then define a right-linear grammar R' from R by replacing $\llbracket^{[j]}_{m,i}$ with $\llbracket^{[j]}_m \underbrace{[^{[j]}_m \dots [^{[j]}_m}_{i\text{-times}} \llbracket^{[j]}_m$ and $\rrbracket^{[j]}_{m,i}$ with $\rrbracket^{[j]}_m \underbrace{]^{[j]}_m \dots]^{[j]}_m}_{i\text{-times}} \rrbracket^{[j]}_m$.

We have

$$L(G) = h'(L(D_{\Delta'}) \cap L(R')).$$

Corollary 1. *There are a multiple Dyck language $D \in q$-MCFL(r) and a homomorphism h such that a language L is in q-MCFL(r) if and only if there is a regular language R such that $L = h(D \cap R)$.*

4 Conclusion

This paper introduced multiple Dyck languages and then proved a Chomsky-Schützenberger-type representation theorem for each class q-MCFL(r) as well as the generator theorem. The literature (e.g. [7,4]) has proposed other parameters such as *degree* and *well-nestedness* that give further classifications of mcfls. Theorem 3 and Corollary 1 hold for those subclasses as well by accordingly modifying the definition of rules of type 1 of multiple Dyck grammars in Definition 1.

Logical characterizations for several classes of languages have been obtained in the literature. For example, the class of regular languages coincides with the class of languages that are definable in monadic second-order logic (see [9]). Also, the class of cfls is exactly the class of languages definable in an existential second-order logic where the second-order variable ranges only over matching predicates [5]. A *matching* predicate M is a binary predicate over the set of positions of symbols in a given string such that each position belongs to at most one pair (i, j) satisfying $M(i, j)$ and M is not crossing ($(i, j) \in M$, $(k, l) \in M$ and $i < k < j$ imply $i < l < j$). Intuitively, $M(i, j)$ means that the symbols occurring at the positions i and j form a pair of a left parenthesis and its corresponding right one. This suggests us to extend a matching predicate to a $2r$-ary predicate M_r to express r pairs of left and right parentheses in $\widehat{\Delta}$ of Section 2.3. It is left as future study to give a logic that characterizes mcfls by using these extended matching predicates.

References

1. Chomsky, N., Schützenberger, M.P.: The algebraic theory of context-free languages. In: Braffort, P., Hirschberg, D. (eds.) Computer Programming and Formal Systems, pp. 118–161. North Holland, Amsterdam (1963)
2. Engelfriet, J.: Context-free graph grammars. In: Handbook of formal languages, vol. 3, Springer, Heidelberg (1997)
3. Kaji, Y.: Universal recognition problems and a representation theorem using dyck-type languages for multiple context-free grammars. Bachelor's thesis, Osaka University (1991)
4. Kanazawa, M.: The pumping lemma for well-nested multiple context-free languages. In: Diekert, V., Nowotka, D. (eds.) Developments in Language Theory. LNCS, vol. 5583, pp. 312–325. Springer, Heidelberg (2009)
5. 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)
6. Rambow, O., Satta, G.: Independent parallelism in finite copying parallel rewriting systems. Theoretical Computer Science 223(1-2), 87–120 (1999)
7. Seki, H., Kato, Y.: On the generative power of multiple context-free grammars and macro grammars. IEICE Transactions 91-D(2), 209–221 (2008)
8. Seki, H., Matsumura, T., Fujii, M., Kasami, T.: On multiple context-free grammars. Theoretical Computer Science 88(2), 191–229 (1991)
9. Thomas, W.: Languages, automata, and logic. In: Handbook of formal languages, vol. 3. Springer, Heidelberg (1997)

Complexity of Guided Insertion-Deletion in RNA-Editing

Hans Zantema[1,2]

[1] Department of Computer Science, TU Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
H.Zantema@tue.nl

[2] Institute for Computing and Information Sciences, Radboud University
Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract. Inspired by RNA-editing, we study an elementary formalism of string replacement based on insertion and deletion using a fixed finite set of *guides*, and in which only occurrences of a single symbol are inserted or deleted. While in this replacement mechanism computation lengths are at most exponential by construction, we show that this exponential upper bound is tight. Moreover, we show that both the class of regular languages and the class of context-free languages are not closed under this replacement mechanism.

1 Introduction

DNA molecules and RNA molecules can be seen as strings over the alphabet $\{C, G, A, T\}$ or $\{C, G, A, U\}$, containing all genotype information of organisms. Replication of this information is one of the most basic mechanisms in life: strings are exactly copied by the well-known mechanisms of DNA-replication and RNA-translation. However, there are also biological mechanisms not yielding an exact copy of a string, but a slight modification. A particular class of such mechanisms is called *RNA-editing*. Inspired by RNA-editing, but abstracting from biological details, in [6] the computational power of insertion-deletion systems is studied. In [6] an insertion step is the replacement of a string uv by $u\alpha v$ for u, α, v taken from a particular finite set of triples u, α, v. Similarly, a deletion step replaces $u\alpha v$ by uv for another finite set of triples u, α, v. In [5] the restriction is considered where u and v are both empty. An extension based on *guides* has been given in [2]. All of these approaches claim full computational power, that is, they generate all recursively enumerable languages.

However, in many RNA-editing mechanisms occurring in nature, only very restricted instances of these formats apply. Often only the particular symbol U is inserted and deleted, instead of arbitrary strings α, see for instance [1]. Inspired by this restriction, in this paper we study a variant in which only instances of one particular symbol 0 are added or removed, not even depending on the surrounding strings u, v. Instead there is a set G of *guides* being a fixed finite set of strings. Depending on G, a deletion step is the replacement of a substring s by a guide $g \in G$, where g is obtained from s by removing one or more occurrences of 0. Similarly, an insertion step is the replacement of a substring s by $g \in G$, where g is obtained from s by adding one or more occurrences of 0. So, both by insertion steps and deletion steps, substrings s are replaced by guides from the given set G, where in doing so only occurrences of 0 are either added or removed. In order to avoid unbounded growth of strings by insertion steps, we require all guides to start and end in elements distinct from 0.

Although our mechanism is clearly an over-simplification of real RNA-editing, it covers a natural and basic ingredient of RNA-editing which we consider to be worthwhile to study in separation. We emphasize the contrast with DNA-computing: where DNA-computing tries to exploit an extreme form of parallelism based on the high number of DNA molecules, we study the behavior of a specific kind of sequential computation. On the other hand, similar insertion-deletion mechanisms occur as primitives of DNA-computing, see e.g. [4].

For a fixed set of guides, starting from any initial string, there is an upper bound on the size of the strings that can be reached from this initial string. As a consequence, the number of these reachable strings is bounded by a function that is exponential in the size of the initial string. So if a string is reachable from the initial string, it is also reachable in a number of steps which is at most exponential in the size of the initial string. A main topic of this paper is the investigation whether this upper bound is tight. It is. This is quite surprising: since substrings are only replaced by guides from a given finite set, at a first glance one may expect that after a small number of steps all substrings are replaced by guides and further computation by insertion or deletion steps will have no effect any more. In fact, we succeed in proving this exponential lower bound for the alphabet $\{0, 1\}$ and a set G of guides consisting of 12 elements.

As another main result we prove that both the class of regular languages and the class of context-free languages are not closed under this insertion-deletion mechanism. More precisely, although the set of strings reachable from a single string is finite and hence regular, we give an example of a regular language L for which we prove that the language of all strings reachable from strings in L is not regular, and a similar example for the class of context-free languages. For both cases the alphabet is $\{0, 1\}$ and the set G consists of only four guides.

Both these main results are obtained by a construction in which particular string rewriting is mimicked by the insertion-deletion mechanism. For the exponential lower bound we use a length preserving string rewriting system inspired by counting in binary numbers, therefore requiring an exponential number of steps to reach the number $2^n - 1$ represented by n ones from the initial number 0 represented by n zeros. For proving that the classes of languages ar not closed under our replacement mechanism we take the string rewrite systems $21 \rightarrow 12$, essentially performing bubble-sort, and for which the strings reachable from $(12)^*$ form a non-regular language and the strings reachable from $\{(12)^k 3^k\}$ form a non-context-free language.

Although the focus of this paper is on our guide-oriented mechanism, we also give a sketch of how the results apply to a natural instance of the insertion-deletion mechanism from [6], namely where particular strings of the shape uv

are replaced by $u0v$ or conversely, in which u, v are not allowed to start or end in 0.

This paper is organized as follows. In Section 2 we give the basic definitions and prove the exponential upper bound. In Section 3 we present the theory relating string rewriting to the insertion-deletion replacement mechanism, being the preparation of our main results. In Section 4 we present our results on the classes of regular languages and of context-free languages. In Section 5 we prove the exponential lower bound. In Section 6 we sketch how our results apply for the alternative insertion-deletion mechanism. We conclude in Section 7.

2 Preliminaries

Let Σ be a finite alphabet with $0 \in \Sigma$. Let Σ^* be the set of finite strings over Σ. For $u \in \Sigma^*$ write $|u|$ for the size of u, that is, the number of elements.

Definition 1. *The relation $<$ is defined to be the smallest transitive relation on Σ^* satisfying $uv < u0v$ for all $u, v \in \Sigma^*$.*

A string $g \in \Sigma^*$ is called a guide *if g is non-empty and both the first and the last element of g is not 0.*

For a finite set G of guides and $u, v \in \Sigma^*$, we write $u \to_G v$ if either

– *$u = xsy$ and $v = xgy$ for $x, s, y \in \Sigma^*$, $g \in G$, $g < s$ (a* delete step*), or*
– *$u = xsy$ and $v = xgy$ for $x, s, y \in \Sigma^*$, $g \in G$, $s < g$ (an* insert step*).*

So $u < v$ means that u can be obtained from v by removing one or more 0's. Note that by $\to_G$-steps the replacements are always towards guides: a substring s is replaced by $g \in G$ in which only 0's are either deleted or inserted.

In order to provide some intuition for guided insertion and deletion we consider $G = \{1101, 1011\}$. Then we have $111 \to_G 1101$ and $111 \to_G 1011$, both by insert steps, but both from 1101 and 1011 no $\to_G$-step is possible. Starting from 10111 we can do an insert step to 101011, while from 101011 we can do a delete step back to 10111. Repeating this we can go on forever. However, from 10111 we can also do an insert step to 101101, from which no further step is possible.

The reason why we disallow guides to start or end in 0 is that we do not want unbounded growth of strings by insert steps. For instance, if we would allow $10 \in G$, this would cause unbounded creation of zeros in the sequence $1 \to_G 10 \to_G 100 \to_G 1000 \to_G 10000 \to_G \cdots$ of insert steps, not being in the flavor of RNA-editing.

For $n \in \mathbf{N}$ and $\to$ any relation on strings we write $u \to^n v$ if there exist strings $u_0, u_1, \ldots, u_n$ such that $u_0 = u$, $u_n = v$, and $u_i \to u_{i+1}$ for all $i = 0, 1, \ldots, n-1$. So $u \to_G^n v$ means that v can be obtained from u in n $\to_G$-steps. We write $u \to_G^* v$ if there exists $n \geq 0$ such that $u \to_G^n v$.

Our first theorem states that the minimal number of steps in $u \to_G^* v$ is at most exponential in the size of u.

Theorem 1 (upper bound). *For every finite set G of guides there is an exponential function $f : \mathbf{N} \to \mathbf{N}$ such that if $u, v \in \Sigma^*$ satisfy $u \to_G^* v$, then $u \to_G^k v$ for some $k \leq f(|u|)$.*

Proof. Let m be the highest number of consecutive zeros occurring in strings from G. Let $n = |u|$. Let $u \to_G^* v$. Then the only difference between u and v is in the sizes of groups of consecutive zeros. Every such group in v is either equal to the corresponding group in u if it was not changed by $u \to_G^* v$, or it is equal to a group of consecutive zeros occurring in a string from G. Since there are at most n such groups, we obtain

$$\#\{v \mid u \to_G^* v\} \leq (m+2)^n.$$

Then in a shortest path $u \to_G^* v$ there will be no duplicate occurrences of the same string, so $u \to_G^k v$ with $k \leq (m+2)^n$. □

3 Relation to String Rewriting

One of the goals of this paper is to prove that the exponential upper bound of Theorem 1 is tight: to choose G, u_m, v_m for which $u_m \to_G^* v_m$, but for which we can show that $u_m \to_G^k v_m$ is only possible for k being exponential in $|u_m|$. Another goal is to prove that classes of languages are not closed under $\to_G^*$. Both goals are achieved for $\Sigma = \{0,1\}$ and are based on a common approach for mimicking a particular kind of string rewriting by $\to_G$. In this section we introduce how to encode strings over another signature Δ into strings over $\Sigma = \{0,1\}$ and investigate how rewriting in Δ^* relates to $\to_G$-steps over Σ^* for a particular choice of G. The results are the basic lemmas as we need them for both of the goals of this paper.

Let $\Sigma = \{0,1\}$. Let $n \geq 2$. Let $\Delta = \{1,2,\ldots,n\}$. For $a,b \in \Delta \cup \{\bot\}$ we write $[a,b]$ for the string consisting of $2n+1$ ones, and at most two zeros according the following rules:

- if $a \neq \bot$ then there is one zero between the a-th and $a+1$-th one;
- if $b \neq \bot$ then there is one zero between the $b+n$-th and $b+n+1$-th one;
- there is no zero between any two other consecutive ones.

For instance, for $n = 4$ we have $[1,3] = 10111111011$ and $[\bot,2] = 1111110111$.

For $k \in \mathbf{N}$, $k \geq 1$, a *k-string encoding* is defined inductively as follows:

- If $a,b \in \Delta \cup \{\bot\}$, not both being $\bot$, then $[a,b]$ is a 1-string encoding.
- If $[a,b]$ is a 1-string encoding and w is a k-string encoding, then $[a,b]00w$ is a $k+1$-string encoding.

So a k-string encoding consists of k ingredients of the shape $[a,b]$, separated by $k-1$ copies of 00.

For $a,b \in \Delta \cup \{\bot\}$ we write $a \sqsubseteq b \iff a = \bot \vee a = b$.

Lemma 1. *Let G consist only of 2-string encodings, and let u be a k-string encoding satisfying $u \to_G v$. Then v is a k-string encoding too. Moreover, one can write*

$$u = x[a_1,a_2]00[a_3,a_4]y, \quad v = x[b_1,b_2]00[b_3,b_4]y,$$

for $[b_1,b_2]00[b_3,b_4] \in G$ and either $a_i \sqsubseteq b_i$ for $i = 1,2,3,4$ or $b_i \sqsubseteq a_i$ for $i = 1,2,3,4$.

Proof. If $u = v$ then the lemma is trivial; assume $u \neq v$. Let $g = [b_1, b_2]00[b_3, b_4]$ be the element of G used for the step $u \to_G v$, so $u = x'sy'$ and $v = x'gy'$ with either $s < g$ or $g < s$. Then both s and g contain exactly $4n+2$ ones. In g there is exactly one group of two consecutive zeros: between the $2n+1$-th and $2n+2$-th one. Since s is part of the k-string encoding u with $4n+2$ ones, it also contains a group of two consecutive zeros, say between its p-th and $p+1$-th one. If $p \neq 2n+1$ then in s we have two consecutive zeros between its p-th and $p+1$-th one, and at most one zero between its $2n+1$-th and $2n+2$-th one. For g it is the other way around. This contradicts the assumption that either $s < g$ or $g < s$. Hence $p = 2n+1$. Hence we can write $s = [a_1, a_2]00[a_3, a_4]$. Now choosing $x = x'$ and $y = y'$ gives the desired result: observe that if $[a_1, a_2]00[a_3, a_4] < [b_1, b_2]00[b_3, b_4]$ then $a_i \sqsubseteq b_i$ for $i = 1, 2, 3, 4$. □

We will consider string rewriting over Δ with respect to string rewrite rules of the shape $ab \to cd$ for $a, b, c, d \in \Delta$. More precisely, we have a rewrite system R of such rules, and we write $s \to_R t$ for $s, t \in \Delta^*$ if and only if we can write $s = xaby$ and $t = xcdy$ for some $x, y \in \Delta^*$ and some rule $ab \to cd$ in R. For such a rewrite system R we define G_R to consist of the following four 2-string encodings:

$$\begin{gathered}[a, a]00[b, b]\\ [\bot, a]00[\bot, b]\\ [c, a]00[d, b]\\ [c, \bot]00[d, \bot]\end{gathered}$$

for every rewrite rule $ab \to cd$ in R.

In order to relate $\to_R$-rewriting to $\to_{G_R}$ computations on string encodings we give an encoding encod from Δ^* to string encodings. This is inductively defined by

$$\mathsf{encod}(i) = [i, \bot], \quad \mathsf{encod}(is) = [i, \bot]00\mathsf{encod}(s),$$

for $i \in \Delta$ and $s \in \Delta^*$.

First we show that every $\to_R$-step can be mimicked by 4 $\to_{G_R}$-steps after encoding.

Lemma 2. *Let $s, s' \in \Delta^*$ satisfy $s \to_R s'$. Then*

$$\mathsf{encod}(s) \to^4_{G_R} \mathsf{encod}(s').$$

Proof. We can write $s = xaby$ and $s' = xcdy$ for a rule $ab \to cd$ in R. Then we have

$$\begin{aligned}\mathsf{encod}(s) &= \mathsf{encod}(x)00[a, \bot]00[b, \bot]00\mathsf{encod}(y)\\ &\to_G \mathsf{encod}(x)00[a, a]00[b, b]00\mathsf{encod}(y)\\ &\to_G \mathsf{encod}(x)00[\bot, a]00[\bot, b]00\mathsf{encod}(y)\\ &\to_G \mathsf{encod}(x)00[c, a]00[d, b]00\mathsf{encod}(y)\\ &\to_G \mathsf{encod}(x)00[c, \bot]00[d, \bot]00\mathsf{encod}(y)\\ &= \mathsf{encod}(s').\end{aligned}$$

□

Next we will prove the converse of this lemma: all $\to_{G_R}$-computations on string encodings can be mimicked by $\to_R$-steps. In order to do so we need a technical lemma in which we require that R is both left-non-overlapping and right-non-overlapping, that is, if $ab \to cd$ and $a'b' \to c'd'$ are rules of R then $b \neq a'$ (left-non-overlapping) and $c \neq d'$ (right-non-overlapping).

Lemma 3. *Let R be left-non-overlapping and right-non-overlapping. Let $s \in \Delta^*$, $|s| = k$, and let*

$$u = [a_1, b_1]00[a_2, b_2]00 \cdots 00[a_k, b_k]$$

be a k-string encoding satisfying $\mathsf{encod}(s) \to^*_{G_R} u$. *Let i satisfy $\perp \neq a_i \neq b_i \neq \perp$. Then either $i > 1$ and $b_{i-1}b_i \to a_{i-1}a_i$ is a rule of R, or $i < k$ and $b_i b_{i+1} \to a_i a_{i+1}$ is a rule of R.*

Proof. Initially in $\mathsf{encod}(s)$ no pattern $\perp \neq a_i \neq b_i \neq \perp$ occurs since $b_i = \perp$ for all i. Using Lemma 1, the only way the pattern $\perp \neq a_i \neq b_i \neq \perp$ can be introduced is by an insert step with respect to the element $[c, a]00[d, b]$ of G_R, for $ab \to cd$ being a rule of R. Then by definition the property holds. We will prove that the property remains in applying $\to_{G_R}$ as long as $\perp \neq a_i \neq b_i \neq \perp$. We distinguish the two cases:

– $i > 1$ and $b_{i-1}b_i \to a_{i-1}a_i$ is a rule of R.
  Using Lemma 1, by keeping $\perp \neq a_i \neq b_i \neq \perp$ the only way the property can be disturbed is by applying $\to_{G_R}$ on the position $[a_{i-2}, b_{i-2}]00[a_{i-1}, b_{i-1}]$ where a_{i-1} or b_{i-1} is replaced by $\perp$.
  If a_{i-1} is replaced by $\perp$ then this is by a delete step with respect to the element $[\perp, a]00[\perp, b]$ of G for a rule $ab \to cd$ of R and $b = b_{i-1}$. But then both $ab \to cd$ and $bb_i \to a_{i-1}a_i$ are rules of R, contradicting R is left-non-overlapping.
  If b_{i-1} is replaced by $\perp$ then this is by a delete step with respect to the element $[c, \perp]00[d, \perp]$ of G for a rule $ab \to cd$ of R and $d = a_{i-1}$. But then both $ab \to cd$ and $b_{i-1}b_i \to da_i$ are rules of R, contradicting R is right-non-overlapping.
– $i < k$ and $b_i b_{i+1} \to a_i a_{i+1}$ is a rule of R.
  This case follows by symmetry.

□

In the rest of our analysis the direction of $\to_R$ does not play a role; for $s, t \in \Delta^*$ we write $s \leftrightarrow_R t$ if and only if either $s \to_R t$ or $t \to_R s$.

We define a decoding map decod from string encodings to Δ^*. For a 1-string encoding $[a, b]$ we define $\mathsf{decod}([a, b]) = a$ if $a > 0$, otherwise $\mathsf{decod}([a, b]) = b$. Remember that in a 1-string encoding $[a, b]$ not both a and b are 0.

For $k > 1$ and a k-string encoding $[a, b]00u$ where u is a $k-1$-string encoding we inductively define

$$\mathsf{decod}([a, b]00u) = \mathsf{decod}([a, b])\mathsf{decod}(u).$$

Lemma 4. *Let R be left-non-overlapping and right-non-overlapping. Let $s \in \Delta^*$, $|s| = k$, and let u, v be k-string encodings satisfying $\mathsf{encod}(s) \to^*_{G_R} u \to_{G_R} v$. Then either $\mathsf{decod}(u) = \mathsf{decod}(v)$ or $\mathsf{decod}(u) \leftrightarrow_R \mathsf{decod}(v)$.*

Proof. We distinguish all four possibilities of $u \to_{G_R} v$, for a rule $ab \to cd$ of R. Due to Lemma 1, steps in $\mathsf{encod}(s) \to^*_{G_R} u \to_{G_R} v$ with respect to $[\bot, a]00[\bot, b]$ and $[c, \bot]00[d, \bot]$ are delete steps, and steps with respect to $[a, a]00[b, b]$ and $[c, a]00[d, b]$ are insert steps.

- $u \to_{G_R} v$ is an insert step with respect to $[a, a]00[b, b] \in G_R$. In this case $\mathsf{decod}(u) = \mathsf{decod}(v)$.
- $u \to_{G_R} v$ is a delete step with respect to $[\bot, a]00[\bot, b] \in G_R$. So $u = \cdots[p, a]00[q, b]\cdots$ and $v = \cdots[\bot, a]00[\bot, b]\cdots$; since $u \neq v$ we conclude that p and q are not both $\bot$. Then using left-non-overlappingness and Lemma 3 we conclude that $ab \to pq$ is a rule of R. Since $\mathsf{decod}([p, a]00[q, b]) = pq$ and $\mathsf{decod}([\bot, a]00[\bot, b]) = ab$, we conclude $\mathsf{decod}(v) \to_R \mathsf{decod}(u)$.
- $u \to_{G_R} v$ is an insert step with respect to $[c, a]00[d, b] \in G_R$. Then $u = \cdots[p, a]00[q, b]\cdots$ and $v = \cdots[c, a]00[d, b]\cdots$ for $p = \bot$ or $q = \bot$. Using left-non-overlappingness and Lemma 3 we conclude that $p = \bot$ and $q \neq \bot$ is not possible, and similar for $p \neq \bot$ and $q = \bot$. So $p = q = \bot$. Since $\mathsf{decod}([\bot, a]00[\bot, b]) = ab$ and $\mathsf{decod}([c, a]00[d, b]) = cd$ and $ab \to cd$ is a rule of R, we conclude $\mathsf{decod}(u) \to_R \mathsf{decod}(v)$.
- $u \to_{G_R} v$ is a delete step with respect to $[c, \bot]00[d, \bot] \in G_R$. In this case $\mathsf{decod}(u) = \mathsf{decod}(v)$.

□

Lemma 5. *Let R be left-non-overlapping and right-non-overlapping. Let $s, s' \in \Delta^*$ satisfy $\mathsf{encod}(s) \to^k_{G_R} \mathsf{encod}(s')$. Then $s \leftrightarrow^m_R s'$ for some $m \leq k$.*

Proof. Immediate from Lemma 4 and $\mathsf{decod}(\mathsf{encod}(s)) = s$ for every $s \in \Delta^*$. □

By an example we show that non-overlappingness is essential in Lemma 5. Let R consist of the two rules $12 \to 33$, $21 \to 33$, and let $s = 121$ and $s' = 333$. Then

$$\mathsf{encod}(s) = [1, \bot]00[2, \bot]00[1, \bot] \to_{G_R} [1, 1]00[2, 2]00[1, \bot] \to_{G_R}$$

$$[1, 1]00[2, 2]00[1, 1] \to_{G_R} [\bot, 1]00[\bot, 2]00[1, 1] \to_{G_R} [\bot, 1]00[\bot, 2]00[\bot, 1] \to_{G_R}$$

$$[3, 1]00[3, 2]00[\bot, 1] \to_{G_R} [3, 1]00[3, 2]00[3, 1] \to_{G_R}$$

$$[3, \bot]00[3, \bot]00[3, 1] \to_{G_R} [3, \bot]00[3, \bot]00[3, \bot] = \mathsf{encod}(s'),$$

while $s \leftrightarrow^m_R s'$ does not hold for any m.

Now we finished all preparations for the main results.

4 Formal Language Properties

In this section we prove that neither the class of regular languages, nor the class of context-free languages is closed under $\to^*_G$. For basics on these classes of languages we refer to standard text books like [3].

For a regular expression r we write $L(r)$ for the language generated by r.

Theorem 2. *The class of regular languages is not closed under $\rightarrow_G^*$, that is, there exists a regular language L over $\Sigma = \{0,1\}$ and a finite set G of guides such that*

$$\{v \in \Sigma^* \mid \exists u \in L : u \rightarrow_G^* v\}$$

is not regular.

Proof. Choose $n = \#\Delta = 2$ and let R consist of the single rule $21 \rightarrow 12$. Choose

$$L = L([1,\perp]00[2,\perp]00)^*[1,\perp]00[2,\perp]) = \{\mathsf{encod}(s) \mid s \in L((12)^+)\}.$$

Since R describes bubble-sort we obtain $(12)^k \rightarrow_R^* 1^k2^k$ for $k > 0$, so by Lemma 2 we obtain $\mathsf{encod}((12)^k) \rightarrow_{G_R}^* \mathsf{encod}(1^k2^k)$. Conversely, from Lemma 5 we obtain that if $\mathsf{encod}((12)^k) \rightarrow_{G_R}^* \mathsf{encod}(s')$ for $s' \in L(1^+2^+)$, then $(12)^k \leftrightarrow_R^* s'$, from which we conclude $s' = 1^k2^k$, since R preserves both the number of 1's and the number of 2's. Combining this yields

$$\{v \in \Sigma^* \mid \exists u \in L : u \rightarrow_{G_R}^* v\} \cap \{\mathsf{encod}(s) \mid s \in L(1^+2^+)\}$$

$$= \quad \{\mathsf{encod}(1^k2^k) \mid k > 0\}.$$

Since $\{\mathsf{encod}(s) \mid s \in L(1^+2^+)\} = L(([1,\perp]00)^*[1,\perp]00[2,\perp](00[2,\perp])^*)$ is regular, $\{\mathsf{encod}(1^k2^k) \mid k \in \mathbf{N}\}$ is not regular, being a straightforward application of the pumping lemma ([3], Theorem 4.1), and regularity is closed under intersection ([3], Theorem 4.8), we conclude that $\{v \in \Sigma^* \mid \exists u \in L : u \rightarrow_{G_R}^* v\}$ is not regular. □

The set $G = G_R$ of guides in Theorem 2 consists of the four strings

`1 101 101 00 101 101 1`	$= [2,2]00[1,1]$
`1 1 1 101 00 1 1 101 1`	$= [\perp,2]00[\perp,1]$
`101 1 101 00 1 10101 1`	$= [1,2]00[2,1]$
`101 1 1 1 00 1 101 1 1`	$= [1,\perp]00[2,\perp]$

Here spaces are only added for readability.

The following theorem is very similar, but then for context-free languages.

Theorem 3. *The class of context-free languages is not closed under $\rightarrow_G^*$, that is, there exists a context-free language L over $\Sigma = \{0,1\}$ and a finite set G of guides such that*

$$\{v \in \Sigma^* \mid \exists u \in L : u \rightarrow_G^* v\}$$

is not context-free.

Proof. Choose $n = \#\Delta = 3$ and let R consist of the single rule $21 \rightarrow 12$. Choose $L = \{\mathsf{encod}((12)^k3^k) \mid k > 0\}$ which is context-free as being generated by $S ::= \mathsf{encod}(123) \mid \mathsf{encod}(12)00S00\mathsf{encod}(3)$. Since $(12)^k3^k \rightarrow_R^* 1^k2^k3^k$, by Lemma 2 we obtain $\mathsf{encod}((12)^k3^k) \rightarrow_{G_R}^* \mathsf{encod}(1^k2^k3^k)$. Conversely, from Lemma 5 we obtain that if $\mathsf{encod}((12)^k3^k) \rightarrow_{G_R}^* \mathsf{encod}(s')$ for $s' \in L(1^+2^+3^+)$, then $(12)^k 3^k \leftrightarrow_R^* s'$, from which we conclude $s' = 1^k 2^k 3^k$, since R preserves the frequency of all separate numbers. Combining this yields

$$\{v \in \Sigma^* \mid \exists u \in L : u \to_{G_R}^* v\} \cap \{\mathsf{encod}(s) \mid s \in L(1^+2^+3^+)\}$$
$$= \{\mathsf{encod}(1^k 2^k 3^k) \mid k > 0\}.$$

Since $\{\mathsf{encod}(s) \mid s \in L(1^+2^+3^+)\}$ is regular, $\{\mathsf{encod}(1^k 2^k 3^k) \mid k \in \mathbf{N}\}$ is not context-free, being a straightforward application of the pumping lemma ([3], Theorem 7.18), and the intersection of a regular and a context-free language is context-free ([3], Theorem 7.27), we conclude that $\{v \in \Sigma^* \mid \exists u \in L : u \to_{G_R}^* v\}$ is not context-free. □

5 The Exponential Lower Bound

In this section we prove an exponential lower bound for the number of $\to_G$-steps. For doing so we need a string rewrite system R meeting our format with exponential reduction length. We choose R to consist of the following three rewrite rules over $\Delta = \{1, 2, 3, 4\}$:

$$\begin{array}{ll} 21 \to 41, & (1) \\ 34 \to 42, & (2) \\ 24 \to 32. & (3) \end{array}$$

Observe that R is both left-non-overlapping and right-non-overlapping. This system was inspired by the system $0e \to 1e$, $1e \to c0e$, $0c \to 1$, $1c \to c0$ describing binary counting from $0^n e$ representing zero to $1^n e$ representing $2^n - 1$. Here e marks the end of the binary number, and c represents the carry. The rules $0e \to 1e$, $1e \to c0e$ describe adding one to the binary number; the other rules are for processing the carry. Since at every step at most one is added to the value, and in the computation from $0^n e$ to $1^n e$ the value increases by $2^n - 1$, this requires an exponential number of steps. This system was changed to R in order to have all left hand sides and right hand sides of rules of size 2, and meet the non-overlappingness requirement. In the R encoding 1 serves as the end symbol, and 2 and 4 represent the bits. The symbol 3 behaves slightly different from the carry, but still forces reduction lengths to be exponential. The following lemma describes how $2^m 1$ rewrites to $4^m 1$ with the required exponential lower bound.

Lemma 6. *For every $m > 0$ there is a reduction $2^m 1 \to_R^* 4^m 1$, but if $2^m 1 \leftrightarrow_R^k 4^m 1$ then $k \geq 2^m - 1$.*

Proof. We prove existence by induction on m. For $m = 1$ the claim is immediate from rule (1). For the induction step we get

$$\begin{array}{rll} 2^m 1 & \to_R^* 24^{m-1}1 & \text{(induction hypothesis)} \\ & \to_R^* 3^{m-1}21 & \text{(rule (3), } m-1 \text{ times)} \\ & \to_R 3^{m-1}41 & \text{(rule (1))} \\ & \to_R^* 42^{m-1}1 & \text{(rule (2), } m-1 \text{ times)} \\ & \to_R^* 4^m 1 & \text{(induction hypothesis).} \end{array}$$

For the lower bound on the number of steps define the weight W on strings as follows:

$$W(i) = i, \;\; W(si) = 2W(s) + i$$

for strings s and $i = 1, 2, 3, 4$. Observe that by applying rule (2) the weight of a string remains the same, since $2*3+4 = 2*4+2$. Similarly, by applying rule (3) the weight of a string remains the same too, since $2*2+4 = 2*3+2$. Observe that in converting 2^m1 by the shape of the rules the symbol 1 only occurs as the last symbol of the string, by which rule (1) only applies at the rightmost position. Hence in such conversions the weight of the string increases by exactly 4 by applying rule (1).

Let $f(m)$ be the number of (1)-steps in the above R-reduction from 2^m1 to 4^m1. Inspecting this reduction easily yields $f(m) = 2^m - 1$. So $W(4^m1) = W(2^m1) + 4*(2^m - 1)$. Hence every $\leftrightarrow_R$-conversion from 2^m1 to 4^m1 needs at least $2^m - 1$ steps of rule (1). □

Now we arrive at the main theorem.

Theorem 4 (lower bound). *For $\Sigma = \{0, 1\}$ there exists a set $G \subseteq \Sigma^*$ of 12 strings, and $u_m, v_m \in \Sigma^*$ for $m = 1, 2, 3, \ldots$, such that $|u_{m-1}| < |u_m| = |v_m|$ and $u_m \to_G^* v_m$ for every m, but if $u_m \to_G^k v_m$ then k is at least exponential in $|u_m| = |v_m|$.*

Proof. Choose $u_m = \mathsf{encod}(2^m1)$, $v_m = \mathsf{encod}(4^m1)$. Choose $G = G_R$ for R as defined above. Combining Lemma 2 and Lemma 6 we obtain

$$u_m \;=\; \mathsf{encod}(2^m1) \;\to_G^*\; \mathsf{encod}(4^m1) = v_m.$$

For the lower bound part assume $\mathsf{encod}(2^m1) = u_m \to_G^k v_m = \mathsf{encod}(4^m1)$ for any m. Then by Lemma 5 we conclude $2^m1 \leftrightarrow_R^p 4^m1$ for $p \leq k$. Then by Lemma 6 we conclude $k \geq p \geq 2^m - 1$. Note that $|u_m| = |v_m| = 12m + 10$, so k is indeed at least exponential in $|u_m| = |v_m|$. □

The set $G = G_R$ of guides in Theorem 4 consists of the 12 strings

1 101 1 1 101 1 1 00 101 1 1 101 1 1 1	$= [2,2]00[1,1]$
1 1 1 1 1 101 1 1 00 1 1 1 1 101 1 1 1	$= [\perp,2]00[\perp,1]$
1 1 1 101 101 1 1 00 101 1 1 101 1 1 1	$= [4,2]00[1,1]$
1 1 1 101 1 1 1 1 00 101 1 1 1 1 1 1 1	$= [4,\perp]00[1,\perp]$
1 1 101 1 1 101 1 00 1 1 1 101 1 1 101	$= [3,3]00[4,4]$
1 1 1 1 1 1 101 1 00 1 1 1 1 1 1 1 101	$= [\perp,3]00[\perp,4]$
1 1 1 101 1 101 1 00 1 101 1 1 1 1 101	$= [4,3]00[2,4]$
1 1 1 101 1 1 1 1 00 1 101 1 1 1 1 1 1	$= [4,\perp]00[2,\perp]$
1 101 1 1 101 1 1 00 1 1 1 101 1 1 101	$= [2,2]00[4,4]$
1 1 1 1 1 101 1 1 00 1 1 1 1 1 1 1 101	$= [\perp,2]00[\perp,4]$
1 1 101 1 101 1 1 00 1 101 1 1 1 1 101	$= [3,2]00[2,4]$
1 1 101 1 1 1 1 1 00 1 101 1 1 1 1 1 1	$= [3,\perp]00[2,\perp]$

Here spaces are only added for readability.

6 Alternative Format

In this section we sketch how our results apply to another format: the natural restriction of the insertion-deletion mechanism from [6] to the case where only 0s are inserted or deleted.

Observe that in all our results we had $G = G_R$ for a particular string rewriting systems R, and that in Lemma 2 the used $\to_G$-steps all satisfy the following extra conditions:

– Every $g \in G$ is either only used for delete steps, or only for insert steps.
– For delete steps exactly two zeros are removed, on positions that are fixed for every $g \in G$.
– For insert steps exactly two zeros are inserted, on positions that are fixed for every $g \in G$.
– In every $g \in G$ the pattern 00 occurs exactly once, in between the two positions to be inserted or deleted.

So every insert step $\to_G$ for $g \in G$ is an application of a rewrite rule of the shape $uv00wx \to u0v00w0x$, in which $g = u0v00w0x$. Similarly, every delete step $\to_G$ for $g \in G$ is an application of a rewrite rule of the shape $u0v00w0x \to uv00wx$, in which $g = uv00wx$. Here all strings u, v, w, x start and end by 1, and do not contain the pattern 00. In its turn, rules of this shape can be mimicked by rules in which by every step only a single 0 is inserted or deleted. This is done as follows: a rule of the shape $uv00wx \to u0v00w0x$ is replaced by the four rules

$$\begin{aligned} uv00wx &\to uv000wx \\ uv000wx &\to u0v000wx \\ u0v000wx &\to u0v000w0x \\ u0v000w0x &\to u0v00w0x, \end{aligned}$$

and similar for rules of the shape $u0v00w0x \to uv00wx$: take the same rules in opposite direction. Let S_R be the resulting set of rewrite rules. By construction we have for $s, s' \in \Delta^*$: if $\mathsf{encod}(s) \to_{G_R}^m \mathsf{encod}(s')$ then $\mathsf{encod}(s) \to_{S_R}^{4m} \mathsf{encod}(s')$. Conversely one can prove:

Lemma 7. *Let R be left-non-overlapping and right-non-overlapping. Let $s, s' \in \Delta^*$ satisfy $\mathsf{encod}(s) \to_{S_R}^k \mathsf{encod}(s')$. Then $\mathsf{encod}(s) \to_{G_R}^m \mathsf{encod}(s')$ for some $m \leq k/4$.*

So our basic lemmas, and hence also Theorems 2, 3 and 4 all hold for $\to_{G_R}$ replaced by $\to_{S_R}$. Note that all rules of S_R are of the shape $uv \to u0v$ or $u0v \to uv$, for none of the u, v starting or ending in 0. So in this way all our results apply for the insertion-deletion mechanism from [6], where strings of the shape uv are replaced by $u\alpha v$ or conversely, for the restricted case of α always being 0 and u, v are not allowed to start or end in 0. As this format easily implies an exponential upper bound of computation lengths similar to Theorem 1, also in this format this bound is tight.

7 Conclusions

In the literature [4,6,5,2] several insertion-deletion mechanisms have been described, often inspired by RNA-editing. All of these have full computational power, more precisely, they generate all recursively enumerable languages. All of these mechanism allow to insert or delete strings of arbitrary shape. In biological RNA-editing mechanisms, however, not arbitrary strings are deleted or inserted, but often only occurrences of a single symbol. This paper restricts to a natural format in which only occurrences of such a single symbol 0 are inserted or deleted. By nature then the computational power is of a much lower level, for instance, in a given setting the number of distinct strings that can be generated typically is bounded by an exponential function. Results in this paper indicate that within these limitations the complexity is as rich as it could be: we gave instances for which we proved that corresponding computation lengths are at least exponential, and for which the classes of regular and context-free languages are not closed.

Acknowledgments. We are grateful to Dragan Bosnacki and Erik de Vink for fruitful discussions.

References

1. Alfonzo, J., Thiemann, O., Simpson, L.: The mechanism of U insertion/deletion RNA editing in kinetoplastid mitochondria. Nucleic Acids Research 25(19), 3751–3759 (1997)
2. Biegler, F., Burrell, M.J., Daley, M.: Regulated RNA rewriting: Modelling RNA editing with guided insertion. Theoretical Computer Science 387(2), 103–112 (2007)
3. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation. Addison-Wesley, Reading (2001)
4. Kari, L., Thierrin, G.: Contextual insertion/deletions and computability. Information and Computation 131, 47–61 (1996)
5. Margenstern, M., Paun, G., Rogozhin, Y., Verlan, S.: Context-free insertion-deletion systems. Theoretical Computer Science 330, 339–348 (2005)
6. Takahara, A., Yokomori, T.: On the computational power of insertion-deletion systems. Natural Computing 2(4), 321–336 (2003)

Author Index