THEORIE-
TAG 2011

21. Theorietag

Automaten und Formale Sprachen

Allrode (Harz), 27. – 29. September 2011
Tagungsband

Jürgen Dassow, Bianca Truthe (Hrsg.)

OTTO VON GUERICKE
UNIVERSITÄT
MAGDEBURG

INF

FAKULTÄT FÜR
INFORMATIK

Titel:
21. Theorietag „Automaten und Formale Sprachen“,
Allrode (Harz), 27. – 29.9.2011, Tagungsband

Veröffentlicht durch die
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Universitätsplatz 2
39106 Magdeburg

Herausgegeben von
Jürgen Dassow und Bianca Truthe, 2011

Vorwort

Der Theorietag ist die Jahrestagung der Fachgruppe *Automaten und Formale Sprachen* der Gesellschaft für Informatik. Er wurde vor 20 Jahren in Magdeburg ins Leben gerufen. Seit dem Jahre 1996 wird der Theorietag von einem eintägigen Workshop mit eingeladenen Vorträgen begleitet. Im Laufe des Theorietags findet auch die jährliche Fachgruppensitzung statt.

Die bisherigen Theorietage wurden in Magdeburg (1991), in Kiel (1992), auf Schloß Dagstuhl (1993), in Herrsching bei München (1994 und 2003), auf Schloß Rauischholzhausen (1995), in Cunnersdorf in der Sächsischen Schweiz (1996), in Barnstorf bei Bremen (1997), in Riveris bei Trier (1998), in Schauenburg-Elmshagen bei Kassel (1999), in Wien (2000 und 2006), in Wendgräben bei Magdeburg (2001), in der Lutherstadt Wittenberg (2002 und 2009), in Caputh bei Potsdam (2004), in Lauterbad bei Freudenstadt (2005), in Leipzig (2007), in Wettenberg-Launsbach bei Gießen (2008) und in Baunatal bei Kassel (2010) ausgerichtet.

Der diesjährige Theorietag wird nach 1991 und 2001 wieder von der Arbeitsgruppe „Formale Sprachen und Automaten“ der Otto-von-Guericke-Universität Magdeburg organisiert. Er findet mit dem vorangehenden Workshop vom 27. bis 29. September 2011 in Allrode im Harz statt. Auf dem Workshop tragen

- Frank Drewes aus Umeå,
- Manfred Droste aus Leipzig,
- Florin Manea aus Bukarest,
- Sergey Verlan aus Paris und
- Georg Zetzsche aus Kaiserslautern

vor. Auf dem Programm des Theorietags stehen 21 weitere Vorträge. Der vorliegende Tagungsband enthält Kurzfassungen aller 26 Beiträge. Die Teilnehmer kommen aus sieben verschiedenen Ländern: Frankreich, Österreich, Rumänien, Schweden, Tschechien, dem Vereinigten Königreich und Deutschland.

Wir danken der Gesellschaft für Informatik, der Otto-von-Guericke-Universität Magdeburg und der METOP GmbH für die Unterstützung dieses Theorietags. Wir wünschen allen Teilnehmern eine interessante und anregende Tagung sowie einen angenehmen Aufenthalt im Harz.

Magdeburg, im September 2011 Jürgen Dassow und Bianca Truthe

Inhalt

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 5 – 7

Varianten eines Algorithmus zum Lernen von Baumreihen

Frank Drewes

Institutionen för datavetenskap, Umeå universitet (Schweden)
drewes@cs.umu.se

Auf dem Gebiet der grammatischen Inferenz ist das Ziel traditionell, aus einer gegebenen Informationsquelle über eine Sprache eine geeignete formelle Definition dieser Sprache abzuleiten (zu „lernen“), also einen entsprechenden Automaten oder eine Grammatik zu konstruieren. Während es traditionell um das Lernen von Wortsprachen geht, werde ich mich in diesem Vortrag auf das Lernen von Baumreihen konzentrieren, und insbesondere auf die Ergebnisse des Artikels [3].

Zur Erinnerung: Ein (kommutativer) Semiring ist eine Menge $\mathbb{S}$ mit einer Addition, einer Multiplikation und Elementen 0 und 1, sodass $(\mathbb{S},+,0)$ und $(\mathbb{S},\cdot,1)$ kommutative Monoide sind und Multiplikation distributiv über Addition ist (sowohl von links als auch von rechts).

Eine Baumreihe ist nun eine Abbildung $\psi\colon \mathrm{T}_\Sigma \to \mathbb{S}$ der Menge T_Σ aller Bäume über einer gegebenen Signatur Σ (also einem Alphabet mit Stelligkeiten) in einen Semiring $\mathbb{S}$. Der Wert $\psi(t)$ eines Baumes t wird auch sein Koeffizient genannt. Das Konzept verallgemeinert in natürlicher Weise das der Sprache, denn wenn wir als $\mathbb{S}$ den booleschen Semiring wählen, ist ψ gerade die charakteristische Funktion einer Sprache (in diesem Fall einer Baumsprache). Baumreihen werden durch gewichtete Baumautomaten erkannt. In einem solchen Automaten ist jeder Transition ein Gewicht in $\mathbb{S}\setminus\{0\}$ zugeordnet. Zusätzlich hat jeder Zustand ein sog. Endgewicht in $\mathbb{S}$, was den Begriff des Endzustandes verallgemeinert. Das Gewicht einer Berechnung auf einem Eingabebaum t ist das Produkt der Gewichte der angewendeten Transitionen, multipliziert mit dem Endgewicht des an der Wurzel erreichten Zustandes. Das Gewicht von t ist die Summe der Gewichte aller Berechnungen auf t. Eine sehr gute und umfassende Einführung in die vielfältigen Ergebnisse der Theorie der gewichteten Baumautomaten findet sich in [4].

Im Vortrag werden in erster Linie deterministische gewichtete Baumautomaten von Interesse sein. Bei diesen entfällt also das Bilden der Summe der Gewichte aller möglichen Berechnungen, da es für jeden Eingabebaum t maximal eine Berechnung gibt, sodass vom Semiring $\mathbb{S}$ nur das kommutative Monoid $(\mathbb{S},\cdot,1)$ von Interesse ist. Die formale Definition gewichteter Baumautomaten sieht damit wie folgt aus:

Definition 0.1 *Ein deterministischer gewichteter Baumautomat ist ein Tupel $A = (\Sigma, Q, \delta, \lambda)$, bestehend aus*

- *der Eingabesignatur Σ,*

- *der endlichen Menge Q von Zuständen, welche als Symbole der Stelligkeit* 0 *angesehen werden,*
- *der partiellen Transitionsfunktion $\delta\colon \Sigma(Q) \to Q \times (\mathbb{S} \setminus \{0\})$* [1] *und*
- *der Zuordnung $\lambda\colon Q \to \mathbb{S}$ von Endgewichten zu Zuständen.*

Die Arbeitsweise eines solchen Automaten ist wie folgt definiert. Die Transitionsfunktion δ wird rekursiv zu einer Funktion $\hat{\delta}\colon \mathrm{T}_\Sigma \to Q \times (\mathbb{S} \setminus \{0\})$ fortgesetzt. Sei dazu $f[t_1,\dots,t_k] \in \mathrm{T}_\Sigma$ (d.h. f hat die Stelligkeit k) und $t_1,\dots,t_k \in \mathrm{T}_\Sigma$. Wenn sowohl alle $\hat{\delta}(t_i) = (q_i,a_i)$ für $i = 1,\dots,k$ als auch $\delta(f[q_1,\dots,q_k]) = (q,a)$ definiert sind, setzen wir $\hat{\delta}(f[t_1,\dots,t_k]) = (q, a \cdot \prod_{i=1}^{k} a_i)$. Andernfalls ist $\hat{\delta}(f[t_1,\dots,t_k])$ undefiniert. Die von A berechnete (oder erkannte) Baumreihe ist damit wie folgt gegeben:

$$A(t) = \begin{cases} a \cdot \lambda(q) & \text{falls } \hat{\delta}(t) = (q,a) \text{ definiert ist} \\ 0 & \text{sonst.} \end{cases}$$

Das Ziel ist also, einen solchen gewichteten Baumautomaten (algorithmisch) zu erlernen. Dazu gibt es eine Reihe unterschiedlicher Modelle, die zum einen die zur Verfügung stehende Information und zum anderen das genaue Ziel definieren. Im Vortrag konzentriere ich mich im Wesentlichen auf Angluins *Minimal-Adequate-Teacher*-Modell (MAT).[2] Bei diesem ist das Ziel, eine deterministisch erkennbare Baumreihe effizient von einem „Lehrer“ zu erlernen, also möglichst in polynomieller Zeit einen Automaten zu konstruieren, der die gewünschte Baumreihe berechnet. Der Lehrer ist ein Orakel, das folgende Typen von Fragen beantwortet:

(1) Gegeben einen Baum t, welchen Koeffizienten hat t?
(2) Gegeben einen deterministischen gewichteten Baumautomaten A, ist A der gesuchte Automat? Falls nicht, liefere ein Gegenbeispiel, also einen Baum t, sodass $A(t)$ nicht der korrekte Koeffizient ist.

Angluins Originalarbeit [1] zeigt, wie in diesem Modell unter Ausnutzung des Myhill-Nerode-Theorems in polynomieller Zeit eine reguläre Sprache L erlernt werden kann. Die Grundstruktur des Algorithmus ist dabei recht einfach und läuft darauf hinaus, iterativ die Äquivalenzklassen der Myhill-Nerode-Äquivalenzrelation zu ermitteln. Beginnend mit einem trivialen Automaten werden dem Lehrer wiederholt Automaten A vorgeschlagen. Die erhaltenen Gegenbeispiele werden unter Zuhilfenahme von Fragen des Typs (1) benutzt, um mehr und mehr Äquivalenzklassen zu unterscheiden und den Automaten zu verfeinern, bis schließlich der gesuchte Automat gefunden ist. Die dazu benötigten, aus den Antworten des Lehrers extrahierten Informationen werden dabei in einer zweidimensionalen Tabelle gespeichert. Deren Zeilen und Spalten sind mit Zeichenketten u,v indiziert, wobei die Zelle (u,v) das (boolesche) Ergebnis des Tests $vu \in L$ enthält. Sind die Zeilen für u und u' ungleich, so bedeutet dies, dass beide unterschiedlichen Myhill-Nerode-Äquivalenzklassen angehören.

Der Originalalgorithmus wurde in der Literatur zunächst auf reguläre Baumsprachen und dann auf erkennbare Baumreihen erweitert, auch dort die Existenz eines Myhill-Nerode-Theorems [2] ausnutzend. Dazu wird von dem Monoid $\mathbb{S}$ zusätzlich verlangt, dass 0 absorbierend ist und alle anderen Elemente von $\mathbb{S}$ ein Inverses haben. Die Zeilen der Tabelle werden nun

[1] Hierbei bezeichnet $\Sigma(Q)$ die Menge aller Bäume $f[q_1,\dots,q_k]$ bestehend aus einem $f \in \Sigma$ der Stelligkeit k und Zuständen $q_1,\dots,q_k \in Q$.

[2] Im Originalmodell geht es um das Erlernen einer regulären Sprache, aber die Erweiterung des Modells auf den Fall der deterministisch erkennbaren Baumreihen macht keine Schwierigkeiten.

mit Bäumen t und die Spalten mit Kontexten c indiziert.[3] Die so adressierte Zelle enthält den Wert $\psi(ct)$. Sind die Zeilen zweier Bäume s, t keine Vielfachen voneinander, so gehören s und t unterschiedlichen Äquivalenzklassen an.

Will man Baumsprachen oder Baumreihen erlernen, so erlangen verschiedene im Zeichenkettenfall untergeordnete Effizienzgesichtspunkte Wichtigkeit. So kann man den Algorithmus z. B. durch die Anwendung einer Technik namens *Contradiction Backtracking* deutlich effizienter machen. Außerdem kann die Tabelle durch eine partielle Tabelle oder eine baumartige Struktur ersetzt werden, um das Füllen und Abfragen irrelevanter Zellen zu vermeiden. Solche Varianten des Algorithmus verlangen allerdings jedes Mal einen erneuten Korrektheitsbeweis, der im Grunde keine neuen Erkenntnisse zutage fördert, da er nur die bereits bekannten Argumente leicht an die neue Situation angepasst wiederholt. Um dies zu vermeiden, wurde in [3] ein abstrakter Datentyp vorgeschlagen, der von der konkreten Realisierung abstrahiert, und es wurde gezeigt, dass jede korrekte Implementierung dieses Datentyps zu einem korrekten Lernalgorithmus führt. Somit reduziert sich ein Korrektheitsbeweis auf den Nachweis, dass sich die wenigen zu implementierenden Operationen des Datentypen wie gefordert verhalten (im Sinne von garantierten Vor- und geforderten Nachbedingungen). Zusätzlich zum beweistechnischen Vorteil ergibt sich der übliche praktische Vorteil, dass unterschiedliche Realisierungen schnell und einfach auf der Basis einer bereits vorhandenen Implementierung des abstrakten Teiles erhalten werden können.

Eine interessante Frage ist, ob auch andere (Klassen von) Lernalgorithmen in ähnlich vorteilhafter Weise auf abstrakte Datentypen reduziert werden können, dessen Realisierungen dann unterschiedliche, aber notwendigerweise korrekte konkrete Lernalgorithmen ergeben. Zusätzlich zu den bereits genannten Vorteilen würde dies eine vertiefte Einsicht darin liefern, welche Eigenschaften für die Korrektheit wirklich erforderlich sind und auf welche verzichtet werden kann.

Literatur

[1] D. ANGLUIN, Learning regular sets from queries and counterexamples. *Information and Computation* **75** (1987), 87–106.

[2] B. BORCHARDT, The Myhill-Nerode theorem for recognizable tree series. In: Z. ÉSIK, Z. FÜLÖP (eds.), *Proceedings of the 7th International Conference on Developments in Language Theory (DLT'03)*. LNCS 2710, Springer-Verlag Berlin, 2003, 146–158.

[3] F. DREWES, J. HÖGBERG, A. MALETTI, MAT learners for tree series – an abstract data type and two realizations. *Acta Informatica* **48** (2011), 165–189.

[4] Z. FÜLÖP, H. VOGLER, Weighted tree automata and tree transducers. In: W. KUICH, M. DROSTE, H. VOGLER (eds.), *Handbook of Weighted Automata*. Springer-Verlag Berlin, 2009, 313–403.

[3] Ein Kontext ist ein Baum c mit genau einem Vorkommen eines Blattes $\square$, ein Platzhalter, an dessen Stelle ein anderer Baum t eingesetzt werden kann. Wir verwenden für Letzteres die Schreibweise ct.

Weighted Automata and Quantitative Logics

Manfred Droste

Institute of Computer Science, Leipzig University
04009 Leipzig
`droste@informatik.uni-leipzig.de`

Abstract

In automata theory, a classical result of Büchi-Elgot-Trakhtenbrot states that the recognizable languages are precisely the ones definable by sentences of monadic second order logic. We will present a generalization of this result to the context of weighted automata. A weighted automaton is a classical non-deterministic automaton in which each transition carries a weight describing e.g. the resources used for its execution, the length of time needed, or its reliability. The behavior (language) of such a weighted automaton is a function associating to each word the weight of its execution. We develop syntax and semantics of a quantitative logic; the semantics counts 'how often' a formula is true.

Our main results show that if the weights are taken either in an arbitrary semiring or in an arbitrary bounded lattice, then the behaviors of weighted automata are precisely the functions definable by sentences of our quantitative logic. The methods also apply to recent quantitative automata model of Henzinger et al. where weights of paths are determined, e.g., as the average of the weights of the path's transitions. Büchi's result follows by considering the classical Boolean algebra {0,1}.

Joint work with Paul Gastin (ENS Cachan), Heiko Vogler (TU Dresden), resp. Ingmar Meinecke (Leipzig).

Turing Machines Deciding According to the Shortest Computations and Deciding Networks of Evolutionary Processors

Florin Manea

Faculty of Mathematics and Computer Science, University of Bucharest,
Str. Academiei 14, RO-010014 Bucharest, Romania
flmanea@fmi.unibuc.ro

Abstract

In this paper we propose, and analyze from the computational complexity point of view, a new variant of nondeterministic Turing machines. Such a machine accepts a given input word if and only if one of its shortest possible computations on that word is accepting; on the other hand, the machine rejects the input word when all the shortest computations performed by the machine on that word are rejecting. Our main results are two new characterizations of $\mathbb{P}^{\mathbb{NP}[\log]}$ and $\mathbb{P}^{\mathbb{NP}}$ in terms of the time complexity classes defined for such machines. These results can be applied to obtain a surprising characterization of $\mathbb{P}^{\mathbb{NP}[\log]}$ as the class of languages that can be decided in polynomial time by ANEPs.

Keywords: Computational Complexity, Turing Machine, Oracle Turing Machine, Shortest Computations, Deciding Networks of Evolutionary Processors.

The reader is referred to [1, 2, 6] for the basic definitions regarding Turing machines, oracle Turing machines, complexity classes and complete problems. In the following we present just the intuition behind these concepts, as a more detailed presentation would exceed the purpose of this paper.

A k-tape Turing machine is a construct $M = (Q, V, U, q_o, acc, rej, B, \delta)$, where Q is a finite set of states, q_0 is the initial state, *acc* and *rej* are the accepting state, respectively the rejecting state, U is the working alphabet, B is the blank-symbol, V is the input alphabet and $\delta : (Q \setminus \{acc, rej\}) \times U^k \to 2^{(Q \times (U \setminus \{B\})^k \times \{L,R\}^k)}$ is the transition function (that defines the moves of the machine). An instantaneous description (ID for short) of a Turing machine is a word that encodes the state of the machine and the contents of the tapes (actually, the finite strings of non-blank symbols that exist on each tape), and the position of the tape heads, at a given moment of the computation. An ID is said to be final if the state encoded in it is the accepting or the rejecting state. A computation of a Turing machine on a given word can be described as a sequence of IDs: each ID is transformed into the next one by simulating a move of the machine. If the computation is finite then the associated sequence is also finite and it ends with a final ID; a computation is said to be an accepting (respectively, rejecting) one, if and only if the final ID encodes the accepting state (respectively, rejecting state). All the possible computations of a nondeterministic machine on a given word can be described as a (potentially infinite) tree of IDs: each ID is transformed into its sons by simulating the possible moves of the machine; this tree is called computations-tree. A word is accepted by a Turing machine if there exists an accepting computation of the machine on that word; it is rejected if all the computations are rejecting. A language is accepted (decided) by a Turing machine if all its words are accepted by the Turing machine, and no other words are accepted by that machine (respectively, all the other words are rejected by that machine). The class of languages accepted by Turing machines is denoted by *RE* (and called the class of recursively enumerable languages), while the class of languages decided by Turing machines is denoted by *REC* (and called the class of recursive languages).

The time complexity (or length) of a finite computation on a given word is the height of the computations-tree of the machine on the word. A language is said to be decided in polynomial time if there exists a Turing M machine and a polynomial f such that the time complexity of a computation of M on each word of length n is less than $f(n)$, and M accepts exactly the given language. The class of languages decided by deterministic Turing machines in polynomial time is denoted $\mathbb{P}$ and the class of languages decided by nondeterministic Turing machines in polynomial time is denoted $\mathbb{NP}$.

A Turing machine with oracle A, where A is a language over the working alphabet of the machine, is a regular Turing machine that has a special tape (the oracle tape) and a special state (the query state). The oracle tape is just as any other tape of the machine, but, every time the machine enters the query state, a move of the machine consists in checking if the word found on the oracle tape is in A or not, and returning the answer. We denote by $\mathbb{P}^{\mathbb{NP}}$ the class of languages decided by deterministic Turing machines, that work in polynomial time, with oracles from $\mathbb{NP}$. We denote by $\mathbb{P}^{\mathbb{NP}[\log]}$ the class of languages decided by deterministic Turing machines, that work in polynomial time, with oracles from $\mathbb{NP}$, and which can enter the query state at most $\mathcal{O}(\log n)$ times in a computation on a input word of length n.

Further we propose a modification of the way Turing machines decide an input word. Then we propose a series of results on the computational power of these machines and the computational complexity classes defined by them.

Definition 1 *Let M be a Turing machine and w be a word over the input alphabet of M. We say that w* is accepted by M with respect to shortest computations *if there exists at least one finite possible computation of M on w, and one of the shortest computations of M on w is accepting; w* is rejected by M w. r. t. shortest computations *if there exists at least one finite computation of M on w, and all the shortest computations of M on w are rejecting. We denote by $L_{sc}(M)$* the language accepted by M w. r. t. shortest computations*, i.e., the set of all words accepted by M, w. r. t. shortest computations. We say that the language $L_{sc}(M)$ is decided by M w. r. t. shortest computations if all the words not accepted by M, w. r. t. shortest computations, are rejected w. r. t. shortest computations.*

The following remark shows that the computational power of the newly defined machines coincides with that of classic Turing machines.

Remark 1 *The class of languages accepted by Turing machines w. r. t. shortest computations equals RE, while the class of languages decided by Turing machines w. r. t. shortest computations equals REC.*

Next we define a computational complexity measure for the Turing machines that decide w. r. t. shortest computations.

Definition 2 *Let M be a Turing machine, and w be a word over the input alphabet of M.* The time complexity of the computation of M on w, measured w. r. t. shortest computations, *is the length of the shortest possible computation of M on w. A language L is said to be decided in polynomial time w. r. t. shortest computations if there exists a Turing M machine and a polynomial f such that the time complexity of a computation of M on each word of length n, measured w. r. t. shortest computations, is less than $f(n)$, and $L_{sc}(M) = L$. We denote by PTime$_{sc}$* the class of languages decided by Turing machines in polynomial time w. r. t. shortest computations.

The main result of this section is the following:

Theorem 1 *PTime$_{sc}$* $= \mathbb{P}^{\mathbb{NP}[\log]}$.

Previously, we have proposed an decision mechanism of Turing machines that basically consisted in identifying the shortest computations of a machine on an input word, and checking if one of these computations is an accepting one, or not. Now we analyze how the properties of the model are changed if we order the computations of a machine and the decision is made according to the first shortest computation, in the defined order.

Let $M = (Q,V,U,q_0,\mathit{acc},\mathit{rej},B,\delta)$ be a t-tape Turing machine and assume that $\delta(q, a_1,\ldots, a_t)$ is a totally ordered set, for all $a_i \in U$, $i \in \{1,\ldots,t\}$, and $q \in Q$; we call such a machine an *ordered Turing machine*. Let w be a word over the input alphabet of M. Assume s_1 and s_2 are two (potentially infinite) sequences describing two possible computations of M on w. We say that s_1 is lexicographically smaller than s_2 if s_1 has fewer moves than s_2, or they have the same number of steps (potentially infinite), the first k IDs of the two computations coincide and the transition that transforms the kth ID of s_1 into the $k+1$th ID of s_1 is smaller than the transition that transforms the kth ID of s_2 into the $k+1$th ID of s_2, with respect to the predefined order of the transitions. It is not hard to see that this is a total order on the computations of M on w. Therefore, given a finite set of computations of M on w one can define the lexicographically first computation of the set as that one which is lexicographically smaller than all the others.

Definition 3 *Let M be an ordered Turing machine, and w be a word over the input alphabet of M. We say that w* is accepted by M with respect to the lexicographically first computation *if there exists at least one finite possible computation of M on w, and the lexicographically first computation of M on w is accepting;* w is rejected by M w. r. t. the lexicographically first computation *if the lexicographically first computation of M on w is rejecting. We denote by $L_{lex}(M)$* the language accepted by M w. r. t. the lexicographically first computation. *We say that the language $L_{lex}(M)$ is decided by M w. r. t. the lexicographically first computation if all the words not contained in $L_{lex}(M)$ are rejected by M.*

As in the case of Turing machines that decide w. r. t. shortest computations, the class of languages accepted by Turing machines w. r. t. the lexicographically first computation equals *RE*, while the class of languages decided by Turing machines w. r. t. the lexicographically first computation equals *REC*. The time complexity of the computations of Turing machines that decide w. r. t. the lexicographically first computation is defined exactly as in the case of machines that decide w. r. t. shortest computations. We denote by *PTime$_{lex}$ the class of languages decided by Turing machines in polynomial time w. r. t. the lexicographically first computation.* In this context, we are able to show the following theorem.

Theorem 2 *PTime*$_{lex} = \mathbb{P}^{\mathbb{NP}}$.

Remark 2 *Note that the proof of Theorem 1 shows that* $\mathbb{P}^{\mathbb{NP}[\log]}$ *can be also characterized as the class of languages that can be decided in polynomial time w. r. t. shortest computations by nondeterministic Turing machines whose shortest computations are either all accepting or all rejecting. On the other hand, in the proof of Theorem 2, the machine that we construct to solve w. r. t. the lexicographically first computation the* TSP_{odd} *problem may have both accepting and rejecting shortest computations on the same input. This shows that* $\mathbb{P}^{\mathbb{NP}[\log]} = \mathbb{P}^{\mathbb{NP}}$ *if and only if all the languages in* $\mathbb{P}^{\mathbb{NP}}$ *can be decided w. r. t. shortest computations by nondeterministic Turing machines whose shortest computations on a given input are either all accepting or all rejecting.*

For full proofs of the above results see [3].

The accepting networks of evolutionary processors (ANEPs, for short) are a bio-inspired computational mode, introduced in [5]. In [4] we discuss the usage of such networks as deciding devices, and we define time-complexity classes for them. The main result that we report in that paper is the following.

Theorem 3
i. *For every ANEP* Γ*, deciding a language* L *and working in polynomial time* $P(n)$*, there exists a nondeterministic polynomial single-tape Turing machine* M*, deciding* L *w. r. t. shortest computations;* M *can be constructed such that it makes at most* $P^2(n)$ *steps in a halting computation on a string of length* n*.*
ii. *For every nondeterministic polynomial single-tape Turing machine* M*, deciding a language* L *w. r. t. shortest computations, there exists a complete ANEP* Γ *deciding the same language* L*. Moreover,* Γ *can be constructed such that Time*$_\Gamma(n) \in \mathcal{O}(P(n))$*, provided that* M *makes at most* $P(n)$ *steps in a halting computation on a string of length* n*.*

Therefore, one can show that:

Theorem 4 *PTime*$_{ANEP} = \mathbb{P}^{\mathbb{NP}[\log]}$.

References

[1] J. HARTMANIS, R. E. STEARNS, On the Computational Complexity of Algorithms. *Trans. Amer. Math. Soc.* **117** (1965), 533–546.

[2] J. E. HOPCROFT, J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[3] F. MANEA, Deciding according to the shortest computations. In: *CiE 2011*. LNCS 6735, Springer, 2011, 191–200.

[4] F. MANEA, Deciding Networks of Evolutionary Processors. In: *DLT 2011*. LNCS 6795, Springer, 2011, 337–349.

[5] M. MARGENSTERN, V. MITRANA, M. J. PÉREZ-JIMÉNEZ, Accepting Hybrid Networks of Evolutionary Processors. In: *Proc. DNA 2004*. LNCS 3384, Springer, 2004, 235–246.

[6] C. M. PAPADIMITRIOU, *Computational complexity*. Addison-Wesley, 1994.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 15 – 16

An Overview of Insertion-Deletion Systems

Sergey Verlan

LACL, Départment Informatique, Université Paris Est
61, av. gén. de Gaulle, F-94010 Créteil, France

In general form, an insertion operation means adding a substring to a given string in a specified (left and right) context, while a deletion operation means removing a substring of a given string being in a specified (left and right) context. An insertion or deletion rule is defined by a triple (u,x,v) meaning that x can be inserted between u and v or deleted if it is between u and v. Thus, an insertion corresponds to the rewriting rule $uv \to uxv$ and a deletion corresponds to the rewriting rule $uxv \to uv$. A finite set of insertion-deletion rules, together with a set of axioms provide a language generating device: starting from the set of initial strings and iterating insertion or deletion operations as defined by the given rules one gets a language. The size of the alphabet, the number of axioms, the size of contexts and of the inserted or deleted string are natural descriptional complexity measures for insertion-deletion systems.

The idea of insertion of one string into another was firstly considered with a linguistic motivation and it permits to describe many interesting linguistic properties like ambiguity and duplication. Another inspiration for these operations comes from the generalization of Kleene's operations of concatenation and closure. The operation of concatenation would produce a string x_1x_2y from two strings x_1x_2 and y. By allowing the concatenation to happen anywhere in the string and not only at its right extremity a string x_1yx_2 can be produced, i.e., y is inserted into x_1x_2. The third inspiration for insertion and deletion operations comes, from the field of molecular biology as they correspond to a mismatched annealing of DNA sequences. They are also present in the evolution processes under the form of point mutations as well as in RNA editing. This biological motivation of insertion-deletion operations lead to their study in the framework of molecular computing.

Insertion-deletion systems are quite powerful, leading to characterizations of recursively enumerable languages. The proof of such characterizations is usually done by showing that the corresponding class of insertion-deletion systems can simulate an arbitrary Chomsky grammar. However, the obtained constructions are quite complicated and specific to the used class of systems. We present a new method of such computational completeness proofs, which relies on a direct simulation of one class of insertion-deletion systems by another. The obtained method is quite generic and it was used to prove the computational completeness of many classes of insertion-deletion systems, the proof being similar for all cases.

An important result in the area is the proof that context-free insertion-deletion systems are computationally complete. This provides a new characterization of recursively enumerable languages, where every such language can be represented as a reflexive and transitive closure of the insertion-deletion of two finite languages.

In the talk we present the result of a systematical investigation of classes of insertion-deletion systems with respect to the size of contexts and inserted/deleted strings. This investigation shows that there are classes that are not computationally complete, some of them being decidable. In these cases it is possible to consider regulated variants of insertion-deletion systems (e.g. graph-controlled, programmed, ordered or matrix). The computational power of the corresponding variants strictly increases permitting to generate all recursively enumerable languages. Such regulated insertion-deletion systems can also be useful to describe biological structures like hairpin, stem and loop, dumbbell, pseudoknot, and cloverleaf as well as natural language constructs.

Recent Results on Erasing in Regulated Rewriting

Georg Zetzsche

TU Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
zetzsche@cs.uni-kl.de

Abstract

For each grammar model with regulated rewriting, it is an important question whether erasing productions add to its expressivity. In some cases, however, this has been a long-standing open problem. In recent years, several results have been obtained that clarified the generative capacity of erasing productions in some grammar models with classical types of regulated rewriting. The aim of this talk is to give an overview of these results.

1. Introduction

Most grammar models come in two flavors: with or without erasing productions, i.e. productions that have the empty word on their right hand side. The variant with erasing productions has desirable closure properties and often allows for convenient descriptions of formal languages. On the other hand, the non-erasing variant has the property that a given word can only be derived from sentential forms that are no longer than the word itself. Thus, in order to solve the membership problem, one only has to consider a finite set of sentential forms that could lead to the word at hand. This often implies decidability or even a linear bound on the space complexity (and thus the context-sensitivity of the languages) when considering the membership problem.

For these reasons, a grammar model in which the erasing and the non-erasing variants coincide in their generative capacity combines nice closure properties with relatively efficient means of analysis. Therefore, whether these two variants differ in their expressive power is an interesting problem for each model.

Despite these facts, for many models, this has been (and in some cases still is) a longstanding open problem. For example, matrix grammars[1] have been introduced by Abraham as early as 1965 and it is still unknown (see Open Problems 1.2.2 in [2]) whether erasing productions increase their expressivity[2]. Moreover, this question remains unanswered for forbidding random context grammars. In addition, until recently, the problem was open for permitting random context grammars, which have been introduced in 1984 by Kelemen (see Open Problems 1.3.1 in [2]).

[1]There are two versions of matrix grammars: with or without appearance checking. Here, we mean those without appearance checking.

[2]In [3, Theorem 2.1, p. 106], it is claimed that erasing productions cannot be avoided. However, none of the given references contains a proof for this claim and in [1], the problem is again denoted as open.

Grammar model	*Status*
Context-free grammars	$=$
Forbidding random context gramm.	?
Indexed grammars	$=$
Matrix gramm. with app. checking	$\neq$
Matrix grammars	?
Ordered grammars	?
Periodically time-variant grammars	?
Permitting random context gramm.	$=$
Petri net controlled grammars	$=$
Programmed grammars	$\neq$
Progr. gramm. with empty fail. fields	?
Regularly controlled grammars	?
Random context grammars	$\neq$
Scattered context grammars	$\neq$
Unordered scattered context gramm.	?
Unordered matrix grammars	?
Unordered vector grammars	$=$
Valence gramm. over comm. monoids	$=$
Vector grammars	$=$

Table 1: Generative capacity of erasing productions

However, in recent years, several results have shed light on these questions. Specifically, it has been shown that in valence grammars over commutative monoids [4], permitting random context grammars [6], Petri net controlled grammars [8, 7], and vector grammars [7], erasing productions can be avoided. The results on Petri net controlled grammars and vector grammars are consequences of a sufficient condition for erasing productions to be avoidable [7]. In addition, for matrix grammars, a partial result and reformulations have been presented [8, 6, 7].

This talk gives an overview of these results. A more comprehensive account of results on erasing productions as well as further references can be found in [5]. For an introduction to the topic of regulated rewriting, we refer the reader to [2] and [1].

Table 1 presents the status of the problem of erasing productions for most classical types of regulated rewriting. Here, the symbol '$=$' means that in the corresponding grammar model, erasing productions can be avoided. Moreover, '$\neq$' is assigned to those models where erasing productions are necessary to generate all languages. Finally, '?' means that (to the author's knowledge) it is an open problem whether in this model, erasing productions increase the generative capacity.

2. Grammars with Control Languages

Some grammar models are defined by adding unique labels to the productions of context-free grammars and then letting languages over these labels specify the valid derivations. Thus, a (labeled) context-free grammar together with a language over the labels constitutes a *grammar with a control language*. Restricting the control languages to a certain class of languages therefore yields a class of grammars. Roughly speaking, $\mathsf{CF}^\lambda(\mathcal{C})$ and $\mathsf{CF}(\mathcal{C})$ denote the class of languages generated by grammars controlled by languages from the class $\mathcal{C}$ with and without erasing productions, respectively. For any language class $\mathcal{C}$, we write $\mathsf{S}(\mathcal{C})$ for the closure of $\mathcal{C}$ with respect to letter substitutions, i.e., substitutions that map each letter to a (finite) set of letters. Furthermore, $\mathsf{D}(\mathcal{C})$ denotes the closure of $\mathcal{C}$ with respect to shuffling with Dyck languages over one pair of parentheses. The first theorem provides an upper bound for $\mathsf{CF}^\lambda(\mathcal{C})$ in terms of non-erasing controlled grammars and a transformation of the class of control languages.

Theorem 2.1 ([7]) *For any language class $\mathcal{C}$, we have* $\mathsf{CF}^\lambda(\mathcal{C}) \subseteq \mathsf{CF}(\mathsf{S}(\mathsf{D}(\mathcal{C})))$.

Clearly, this implies that erasing productions can be avoided provided that the class of control languages is closed against letter substitutions and shuffling with Dyck languages (over one pair of parentheses).

Corollary 2.2 ([7]) *Let $\mathcal{C}$ be closed under letter substitutions and shuffling with Dyck languages. Then,* $\mathsf{CF}^\lambda(\mathcal{C}) = \mathsf{CF}(\mathcal{C})$.

Petri net controlled grammars are grammars with a Petri net language (defined using reachability of a final marking) as its control language. The language classes generated by these grammars are denoted by PN^λ and PN (for grammars with and without erasing, respectively). Since the class of Petri net languages is well-known to be closed under letter substitution as well as shuffling with Dyck languages, we obtain the following.

Theorem 2.3 ([8, 7]) $\mathsf{PN}^\lambda = \mathsf{PN}$.

Vector grammars are grammars controlled by languages of the form $V^{\sqcup\!\sqcup}$, in which V is a finite set of words and $V^{\sqcup\!\sqcup}$ denotes the iterated shuffle of V. The language classes generated by these grammars are denoted by V^λ and V. Let $L \sqcup\!\sqcup K$ be the shuffle of the languages L and K. Then, we have $V^{\sqcup\!\sqcup} \sqcup\!\sqcup D = (V \cup \{ab\})^{\sqcup\!\sqcup}$ if D denotes the Dyck language with a as the opening and b as the closing parenthesis. Furthermore, if α is a letter substitution, then $\alpha(V^{\sqcup\!\sqcup}) = \alpha(V)^{\sqcup\!\sqcup}$. Thus, the following is also a consequence of Corollary 2.2.

Theorem 2.4 ([7]) $\mathsf{V}^\lambda = \mathsf{V}$.

A *valence grammar over a monoid* M is a grammar controlled by a language of the form $\varphi^{-1}(1)$, where $\varphi : \Gamma^* \to M$ is a monoid homomorphism. The language classes generated by these grammars are denoted by $\mathsf{VAL}^\lambda(M)$ and $\mathsf{VAL}(M)$. Fernau and Stiebe have shown in [4] that valence grammars over commutative monoids admit Chomsky and Greibach normal forms. This means in particular that erasing productions can be eliminated.

Theorem 2.5 ([4]) $\mathsf{VAL}^\lambda(M) = \mathsf{VAL}(M)$ *for any commutative monoid* M.

A *matrix grammar* is a grammar controlled by a language of the form M^*, in which M is a finite set of words. The language classes generated by these grammars are denoted by MAT^λ and MAT. As mentioned above, the question of whether each matrix grammar has a non-erasing equivalent remains open. However, there has been a partial result and reformulations of the problem. The partial result interprets the problem as the question of whether arbitrary homomorphic images of languages in MAT are again contained in MAT. The weaker statement of the partial result then restricts the kind of languages as well as the kind of homomorphisms applied to them. Let $\mathcal{L}_0$ be the class of Petri net languages (with non-empty transition labels). It is a well-known fact that $\mathcal{L}_0 \subseteq \mathsf{MAT}^\lambda$. A homomorphism $h : \Sigma^* \to \Gamma^*$ is called a *linear erasing* on a language L if there is a constant k such that $|w| \leq k \cdot |h(w)|$ for every $w \in L$. $\mathcal{H}^{\mathrm{lin}}(\mathcal{L}_0)$ is the class of all languages $h(L)$ where L is in $\mathcal{L}_0$ and h is a linear erasing on L.

Theorem 2.6 ([8]) $\mathcal{H}^{\mathrm{lin}}(\mathcal{L}_0) \subseteq \mathsf{MAT}$.

For reformulations of the strictness problem of $\mathsf{MAT} \subseteq \mathsf{MAT}^\lambda$, see [8] and [7].

Conjecture 2.7 ([6]) $\mathsf{MAT}^\lambda = \mathsf{MAT}$.

3. Random Context Grammars

Random context grammars extend context-free grammars by equipping each production with two sets of nonterminals: the *forbidding* and the *permitting context*. Then, a production can only be applied if, in the sentential form, each of the symbols in its permitting context appears and none of the symbols in its forbidding context occurs. The subtype of *permitting random context grammars* restricts the forbidding context to be empty. The corresponding language classes are denoted by pRC^λ and pRC.

Theorem 3.1 ([6]) $\mathsf{pRC}^\lambda = \mathsf{pRC}$.

Another open question concerns whether the inclusion $\mathsf{pRC} \subseteq \mathsf{MAT}$ is strict (see Open Problems 1.2.2 in [2]). Since in [6], a language has been presented that seems to lie in $\mathsf{MAT} \setminus \mathsf{pRC}$, the following conjecture has been made. Theorem 3.3 then clearly follows from Theorem 3.1.

Conjecture 3.2 ([6]) $\mathsf{pRC} \neq \mathsf{MAT}$.

Theorem 3.3 ([6]) *Of the conjectures 2.7 and 3.2, at least one holds.*

References

[1] J. Dassow, Grammars with regulated rewriting. In: C. Martín-Vide, V. Mitrana, G. Păun (eds.), *Formal Languages and Applications*. Springer-Verlag, Berlin, 2004, 249–273.

[2] J. Dassow, G. Păun, *Regulated rewriting in formal language theory*. Springer-Verlag, Berlin, 1989.

[3] J. Dassow, G. Păun, A. Salomaa, Grammars with Controlled Derivations. In: G. Rozenberg, A. Salomaa (eds.), *Handbook of Formal Languages, Volume 2*. Springer-Verlag, Berlin, 1997, 101–154.

[4] H. Fernau, R. Stiebe, Sequential grammars and automata with valences. *Theoretical Computer Science* **276** (2002), 377–405.

[5] G. Zetzsche, *On Erasing Productions in Grammars with Regulated Rewriting*. Diploma thesis, Universität Hamburg, 2010.

[6] G. Zetzsche, On Erasing Productions in Random Context Grammars. In: S. A. et al. (ed.), *37th International Colloquium on Automata, Languages, and Programming, ICALP 2010, Bordeaux, France, July 5–10, 2010. Proceedings*. Lecture Notes in Computer Science 6199, Springer-Verlag, Berlin, Heidelberg, New York, 2010, 175–186.

[7] G. Zetzsche, A Sufficient Condition for Erasing Productions to Be Avoidable. In: *Developments in Language Theory. 15th International Conference, DLT 2011, Milan, Italy, July 19-22, 2011. Proceedings*. Lecture Notes in Computer Science 6795, Springer, 2011, 452–463.

[8] G. Zetzsche, Toward Understanding the Generative Capacity of Erasing Rules in Matrix Grammars. *International Journal of Foundations of Computer Science* **22** (2011), 411–426.

SAT-basiertes Minimieren von Büchi-Automaten

Stephan Barth Martin Hofmann

Institut für Informatik, Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München
barths@cip.ifi.lmu.de hofmann@ifi.lmu.de

Zusammenfassung

Wir beschreiben eine SAT-basierte Minimierung für nicht-deterministische Büchi-Automaten (NBA). Für einen gegebenen NBA A findet das Verfahren einen äquivalenten NBA A_{min} mit der minimal notwendigen Zahl an Zuständen. Dies erfolgt indem sukzessiv Automaten berechnet werden, die A annähern, in dem Sinne, dass sie bestimmte positive Beispiele akzeptieren und negative Beispiele ablehnen. Im Laufe des Verfahrens werden diese Beispielmengen schrittweise vergrößert. Wir verwenden einen SAT-Solver um zu den Beispielmengen einen passenden NBA zu finden. Wir verwenden Ramsey-basierte Komplementierung um die Äquivalenz der Kandidaten mit A zu überprüfen. Gilt die Äquivalenz nicht, so erzeugt der Gleichheitstest neue positive oder negative Beispiele. Wir haben das Verfahren an nichttrivialen, im Sinne von schwierig von Hand zu minimierenden, Beispielen getestet. Der Flaschenhals des Verfahrens ist die Komplementierung.

1. Einleitung

1.1. Definition von Büchi-Automaten

Büchi-Automaten sind eine Möglichkeit ω-reguläre Sprachen zu beschreiben, das sind Sprachen unendlich langer Wörter. Formal bestrachtet sind Büchi-Automaten ein Tupel (Q,Σ,q_0,F,δ), wobei Q eine endliche Menge ist, Σ ein endliches Alphabet, $q_0 \in Q$ der Startzustand, $F \subset Q$ die Menge der Endzustände und $\delta : Q \times \Sigma \to 2^Q$ die Übergangsfunktion ist.
Ein (unendlich langes) Wort wird vom Automaten akzeptiert, falls es einen Lauf gibt, der immer wieder bei einem Endzustand vorbeikommt.

1.2. Komplexität

Während das Minimierungsproblem für endliche deterministische Automaten noch in P liegt, liegt das Minimierungsproblem für deterministische Büchi-Automaten (da sie NP-schwer ist [4] und in NP liegt [2]).
Im Falle nichtdeterministische Automaten ist das Minimierungsproblem allerdings, da es für endliche Automaten bereits PSPACE-vollständig ist (sogar schon das Berechnen der Größe des minimalen Automatens ist PSPACE-vollständig [3] Seite 27, Satz 3), im Falle von Büchi-Automaten nicht komplizierter, es ist ebenfalls PSPACE-vollständig.

2. Konzept

2.1. Annäherung durch gute und böse Wörter

Wir berechnen zu einem NBA A (Zustandszahl n, Alphabet Σ) Annäherungen A', die in dem Sinne A annähern, dass sie bestimmte positive Beispiele akzeptieren (im folgenden gute Wörter bezeichnet) und negative Beispiele ablehnen (im folgenden böse Wörter bezeichnet).

Die kleinste Annäherung an A hat höchstens n Zustände, da $A' = A$ bereits eine Annäherung an A mit Zustandszahl n ist, unabhängig davon, welche Listen an guten und bösen Wörter gewählt wurden. Nun gibt es aber über dem Alphabet Σ aber nur endlich viele Automaten mit höchstens n Zuständen (für jede der endlich vielen Zustandszahlen gibt es nur endliche viele Automaten, da es nur endlich viele Variationen der Endzustände und endlich viele mögliche Übergangsfunktionen gibt).

Fängt man nun mit einer beliebigen Liste an Wörtern an, so kann man sich zu dieser Liste an Wörtern den kleinsten Automaten A' suchen. Mit diesem trifft man auf einen von zwei möglichen Fällen

- Der Automat stimmt mit A überein. Dann ist man fertig, man hat den minimalen mit A übereinstimmenden Automaten gefunden, da dieser Automat der kleinste ist, der die Beispielwörter korrekt klassifiziert.
- Hat man jedoch einen Automaten, der bei dieser Liste mit A übereinstimmt, allerdings nicht zu A äquivalent ist, so kann man effektiv ein Gegenbeispiel angeben, bei welchem beiden Automaten nicht übereinstimmen. Mit diesem Gegenbeispiel kann man nun die Beispielliste erweitern und so ausschließen daß diese falsche Annäherung an A nochmals gefunden wird. Da es allerdings, wie anfangs in diesem Abschnitt gesagt, nur endlich viele Büchi-Automaten mit höchstens n Zuständen gibt, so wird dieser Fall nur endlich oft auftreten.

Bei diesem Verfahren ist es ausreichend Schleifenwörter (Wörter w der Gestalt: Es gibt $a, b \in \Sigma^*$ mit $w = ab^\omega$) in der Wortliste zu verwenden, da die Menge der Worte, bei denen der Automat A mit dem gefundenen Kandidaten A' nicht übereinstimmt regulär ist und damit (sofern sie nicht leer ist) mindestens ein Schleifenwort enthält.

2.2. Finden von Annäherungen mit SAT

Zu gegebenen Listen guter und böser Wörter und einer Zustandszahl m kann man nun mithilfe eines SAT-Solvers (wir verwenden Minisat [1]) einen Automaten konstruieren, oder die Information erhalten, daß es einen solchen nicht gibt. Iteriert man dieses Verfahren von $m = 1$ ab für ansteigende m, so erhält man den kleinsten Automaten, der bei den Beispielwörtern passt.

Dazu definiert man Variablen f_i (Zustand i ist ein Endzustand) und $t_{i,j,\alpha}$ (man kann mit Buchstabe α von Zustand i nach Zustand j kommen), die den Automaten beschreiben und über eine Reihe von Hilfsvariablen schließlich Variablen $z_{a,b}$ (das Wort ab^ω wird vom Automaten akzeptiert). Nimmt man diese die Hilfsvariablen definierende Ausdrücke zusammen, sowie $z_{a,b}$ für alle guten Wörter $w = ab^\omega$ und $\neg z_{a,b}$ für alle bösen Wörter $w = ab^\omega$, so erhält man einen SAT-Ausdruck, dessen Lösung den gesuchten Automaten beschreibt, oder eben nicht lösbar ist, wenn die Zustandszahl zu klein gewählt ist.

3. Implementation

3.1. Pseudocode

Der Minimierer ist in der Programmiersprache ocaml implementiert, als SAT-Solver wird das Programm Minisat[1] verwendet. Der Programmfluß ist im Wesentlichen der folgende:

```
A = zu minimierender Automat;
negA = Komplement A;
G = B = []; (* Die Listen guter und böser Wörter. *)
n = 1;
Schleifenanfang
  versuche A’ = SAT-Solver (SAT-Ausdruck G B n)
    Fehlschlag -> (* gibt keinen Kandidaten mit n Zuständen. *)
      n = n + 1;
      falls n = |A| gibt A zurück;
      Zurück zum Schleifenanfang;
    Erfolg -> (* A’ ist nun ein Kandidat für den minimierten Automaten. *)
      xB = Schnitt A’ negA;
      falls xB nichtleer
        push B (einschleifenwortaus xB); (* neues böses Beispiel. *)
        Zurück zum Schleifenanfang;
      sonst
        negA’ = Komplement A’;
        xG = Schnitt A negA’;
        falls xG nichtleer
          push G (einschleifenwortaus xG); (* neues gutes Beispiel. *)
          Zurück zum Schleifenanfang;
        sonst
          Gib A’ zurück.
```

Folgende Erweiterungen sind gegenüber dem reinen Konzept hier noch eingeflossen:

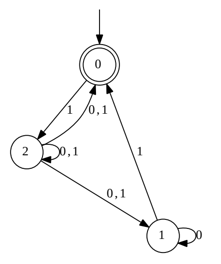
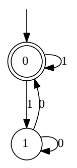
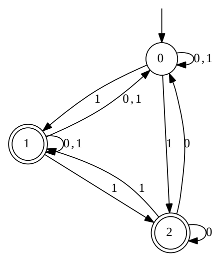
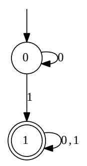
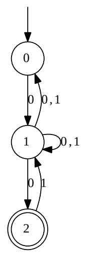
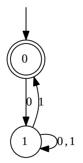
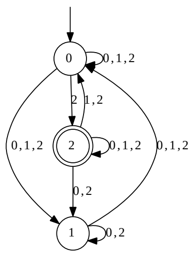
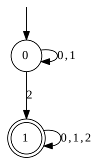
- Für die erweiterten Listen guter und böser Wörter wird nicht jeweils von 1 nach dem minimalen Automaten gesucht, sondern man belässt n bei dem Wert, den es bei kürzeren Wortlisten hatte, da eine Erhöhung der Beispielworte unmöglich zu einer kleineren minimalen Lösung führen kann.

- Gleichheitstest wurde implementiert durch Leerheitstests auf den Schnitten $\neg A \cap A'$ und $A \cap \neg A'$. Ferner sucht man erstmal $\neg A \cap A'$ nach einem Beispielwort ab und sucht nicht gleich die ganze symmetrische Differenz ab, da man dann viele gefundene Kandidaten nicht komplementieren muß. Dies steigert die Effizienz, da der Flaschenhals eben die Komplementierung ist.

- Wenn man Automaten der Größe n sucht, so wird gleich der Originalautomat zurückgegeben, was gültig ist, da man hier bereits weiß, dass es keine kleineren äquivalenten Automaten geben kann. Dieser trivial überprüfbare Fall ist sonst mitunter für einen merklichen Anteil der verwendeten Rechenzeit verantwortlich.

3.2. Berechnungsbeispiele

Zur Veranschaulichung wird hier noch eine Auswahl von zufällig erzeugten Büchi-Automaten gegeben, sowie die von dem Programm gefundenen äquivalenten minimalen Automaten.

Originalautomat	Minimierter Automat	Originalautomat	Minimierter Automat
			
			

Erwähnenswert ist, dass man in keinem der Fälle den minimierten Automaten durch reines Vereinigen und Weglassen von Zuständen erhalten hätte.

4. Fazit

Durch seinen allgemein gehaltenen Ansatz kann man mit diesem Verfahren prinzipiell jeden NBA minimieren. Für viele Praxisanwendungen läuft das Verfahren zwar zu langsam, wie es bei einem PSPACE-vollständigem Problem zu erwarten war, dennoch können eine Reihe von nicht-trivialen NBAs damit in vertretbarer Zeit minimiert werden.

Die Komplementierung ist momentan eindeutig der Flaschenhals (über 90% des Programmlaufes steckt in der Komplementierung), daher sollten Optimierungen primär hier ansetzen.

Literatur

[1] N. EÉN, N. SÖRENSSON, Minisat. http://minisat.se/. Zugriff 13.August 2011.

[2] R. EHLERS, Minimising Deterministic Büchi Automata Precisely Using SAT Solving. In: O. STRICHMAN, S. SZEIDER (eds.), *13th Thirteenth International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*. LNCS 6175, Springer-Verlag, 2010, 326–332.

[3] G. GRAMLICH, *Über die algorithmische Komplexität regulärer Sprachen*. Dissertation, Johann Wolfgang Goethe-Universität, Frankfurt am Main, 2007.

[4] S. SCHEWE, *Minimisation of Deterministic Parity and Büchi Automata and Relative Minimisation of Deterministic Finite Automata*. Technical report, University of Liverpool, 2010.

Learning Limited Context Restarting Automata by Genetic Algorithms

Stanislav Basovník František Mráz

Charles University, Faculty of Mathematics and Physics
Department of Computer Science, Malostranské nám. 25
CZ-118 00 Praha 1, Czech Republic
sbasovnik@gmail.com, mraz@ksvi.ms.mff.cuni.cz

Abstract

We propose a genetic algorithm for learning restricted variants of restarting automata from positive and negative samples. Experiments comparing the proposed genetic algorithm to algorithms RPNI and LARS on sample languages indicate that the new algorithm is able to infer a target language even from a small set of samples.

1. Introduction

Restarting automata [6] were introduced as a model for linguistically motivated method of checking correctness of sentences of a natural language by stepwise simplification of the input sentence while preserving its (non)correctness.

A restarting automaton can be represented as a finite set of meta-instructions defining possible reductions of a current word. In a general restarting automaton, each reduction consists in rewriting a short subword by even shorter word. The possibility to apply a meta-instruction is controlled by the content of the whole tape to the left and to the right from the place of rewriting. The left and right context must belong to given regular languages which are comprised by the meta-instruction as regular constraints.

Several variants of restarting automata were studied in numerous papers [6]. In this paper we propose more restricted variant of restarting automata for which the possibility to apply a meta-instruction is controlled by a fixed finite size context around the rewritten subword – restarting automata with limited context (lc-R-automata). We propose a special version of a genetic algorithm to learn lc-R-automata from both positive and negative samples. Due to their simpler definition, the learned lc-R-automata are much easier to interpret by humans than general restarting automata.

In Section 2 we introduce lc-R-automata and their several more restricted variants. Section 3 presents the proposed genetic algorithm for learning lc-R-automata from positive and negative samples. Then we compare the proposed algorithm with well-known learning algorithms – RPNI [5] and LARS [3].

This work was partially supported by the Grant Agency of the Czech Republic under Grant-No. P103/10/0783 and P202/10/1333.

2. Definitions and Notations

Let λ denote the empty word and $|w|$ denote the *length* of the word w.

Definition 2.1 *A* limited context restarting automaton, lc-R-*automaton for short, is a system* $M = (\Sigma, \Gamma, I)$*, where* Σ *is an input alphabet,* Γ *is a working alphabet containing* Σ*, and* I *is a finite set of* meta-instructions *of the following form* $(\ell \mid x \to y \mid r)$*, where* $x, y \in \Gamma^*$ *such that* $|x| > |y|$*,* $\ell \in \{\lambda, ¢\} \cdot \Sigma^*$ *and* $r \in \Sigma^* \cdot \{\lambda, \$\}$.

An lc-R-*automaton* $M = (\Sigma, \Gamma, I)$ *induces a reduction relation* $\vdash_M$ *as follows: for each* $u, v \in \Gamma^*$*,* $u \vdash_M v$ *if there exist words* $u_1, u_2 \in \Gamma^*$ *and a meta-instruction* $(\ell \mid x \to y \mid r)$ *in* I *such that* $u = u_1 x u_2$*,* $v = u_1 y u_2$*,* ℓ *is a suffix of* $¢u_1$ *and* r *is a prefix of* $u_2\$$*. Let* $\vdash_M^*$ *denote the reflexive and transitive closure of* $\vdash_M$*. The language accepted by the* lc-R-*automaton* M *is* $L(M) = \{w \in \Sigma^* \mid w \vdash^* \lambda\}$.

An lc-R-automaton M accepts exactly the set of input words which can be reduced to λ. Obviously, λ is in $L(M)$, for each lc-R-automaton M.

Example 2.2 *Let* $M = (\{a,b\}, \{a,b\}, I)$*, where* $I = \{(a \mid abb \to \lambda \mid b), (¢ \mid abb \to \lambda \mid \$)\}$*, be an* lc-R-*automaton. Then* $aaabbbbbb \vdash_M^c aabbbb \vdash_M^c abb \vdash_M^c \lambda$ *and the word* a^3b^6 *belongs to* $L(M)$*. It is easy to see that* $L(M) = \{a^n b^{2n} \mid n \geq 0\}$.

We consider several restricted variants of lc-R-automata. We say, that R is of type :

$\mathcal{R}_0$ if I is arbitrary finite set of rules without any restriction.

$\mathcal{R}_1$ if I contains only rules of the following two forms $(u|x \to \lambda|v)$ and $(u|x \to a|v)$, where $a \in \Gamma$, $u \in \{\lambda, ¢\} \cdot \Gamma^*$, $v \in \Gamma^* \cdot \{\lambda, \$\}$ and $x \in \Gamma^*$.

$\mathcal{R}_2$ if I contains only rules of the following two forms $(u|x \to \lambda|v)$ or $(u|x \to a|v)$, where $a \in \Gamma$, $u \in \{\lambda, ¢\}$ and $v \in \{\lambda, \$\}$ and $x \in \Gamma^*$.

$\mathcal{R}_3$ if I contains only rules of the following two forms $(u|x \to \lambda|\$)$ or $(u|x \to a|\$)$, where $a \in \Gamma$, $u \in \{\lambda, ¢\}$ and $x \in \Gamma^*$.

Basovník in [1] studied the power of lc-R-automata. lc-R-automata of type $\mathcal{R}_0$ recognize exactly the class of growing context-sensitive languages. lc-R-automata of type $\mathcal{R}_2$ recognize exactly the class of context-free languages and lc-R-automata of type $\mathcal{R}_3$ recognize exactly the class of regular languages.

The problem of inferring an lc-R-automaton for a target language $L \subseteq \Sigma^*$ consists in learning its set of rewriting meta-instructions. In this work, lc-R-automata are inferred from an input set of positive and negative samples $\langle S_+, S_- \rangle$, where $S_+ \subseteq L$, $S_- \subseteq \Sigma^* \setminus L$, using genetic algorithm [4]. Every individual in a population is a set of rewriting rules. Its fitness is

$$F = 100 \left(1 - \frac{1}{2} \left(\frac{|E_+| - \frac{b}{3}}{|S_+|} + \frac{|E_-|}{|S_-|} \right) \right) \quad ,$$

where E_+ is a set of rejected positive samples, E_- is a set of accepted negative samples and b is a bonus for partially reduced rejected positive samples:

$$b = \sum_{s \in E_+} \frac{|s| - |s'|}{|s|} \quad ,$$

where s' is the shortest word, that can be created from s using rewriting rules of the evaluated individual.

The genetic algorithm uses three operators:

1. Selection: a tournament selection with elitism – 3 fittest individuals are copied into the new generation without any change; in a tournament two individuals are randomly selected and with probability 0.75 the individual with higher fitness is used for constructing the new generation.
2. Crossover: a random shuffle of rules of two individuals – if some individual is left without rules, a random rule is added to it.
3. Mutation: one of the following changes in an individual is made:
   - with probability 0.1 a randomly chosen rule is deleted,
   - with probability 0.35 a random rule is added,
   - with probability 0.55 a randomly chosen rule is edited:
     - with probability 0.1 a random symbol is deleted,
     - with probability 0.45 a new symbol is randomly inserted,
     - with probability 0.45 a randomly chosen symbol is rewritten.

Each initial population contains individuals with a single rule of the form $(¢, w, \$)$, where w is one of the positive training samples of minimal length.

3. Results

We have compared learning of lc-R-automata using genetic algorithms with two well-known methods for grammatical inference – regular positive and negative inference (RPNI; [5]) for inferring regular languages and learning algorithm for rewriting systems (LARS; [3]) for inferring languages represented by string rewriting systems, because restarting automata can be interpreted as a regulated string rewriting systems, too. The algorithms were compared on two sets of languages: 15 regular languages used by Dupont in [2] and the following 7 context-free languages: $\{a^nb^n \mid n \geq 0\}$, $\{a^ncb^n \mid n \geq 0\}$, $\{a^nb^{2n} \mid n \geq 0\}$, $\{ww^R \mid w \in \{a,b\}^*\}$, $\{w \mid w = w^R, w \in \{a,b\}^*\}$, the Dyck language of matching parentheses (and), and another Dyck language of matching parentheses of two types (,), [,].

After thorough experiments, we fixed parameters of the genetic algorithm: population size 200, epoch count 400, mutation rate 0.2, crossover rate 0.6, no auxiliary symbols were allowed (working alphabet coincided with the input alphabet). For each tested language L, we generated two sets S_+, S_- of positive and negative samples of size $N = |S_+| = |S_-|$.

Positive samples were generated by breadth-first search of derivations according to a fixed grammar for L, where i-th generated word was inserted into S_+ with a probability $\frac{i}{10 \cdot N} + 0.5$, until we obtained N positive samples. While negative samples were generated randomly.

For each tested language there were generated 100 sets of positive and negative samples. Each set was randomly split into (i) training samples containing t_n positive and t_n negative samples and (ii) testing samples containing t_t positive and t_t negative samples for the learned language ($N = t_n + t_t$). In all our experiments $t_t = 50$.

Table 1: Mean values of the fitness function on test sets achieved by the tested algorithms together with their standard deviations.

	regular languages		context-free languages
	$t_n = 100$	$t_n = 10$	$t_n = 10$
RPNI	99.6 ± 1.2	84.9 ± 18.4	75.3 ± 18.6
LARS	97.3 ± 9.1	80.8 ± 19.8	72.3 ± 20.6
lc-R	98.2 ± 4.5	91.8 ± 10.6	92.6 ± 12.3

Learning of restarting automata was performed for all restriction types from $\mathcal{R}_0$ to $\mathcal{R}_3$ and the best result was used. In the first experiment we compared all three methods on large training sets of samples ($t_n = 100$) of regular languages. The best method was RPNI, but the learning restarting automata had similar result and for some languages it was even better than RPNI algorithm. During the second and the third experiment the learning methods were tested on a small training sets of samples ($t_n = 10$) from the regular languages and from the context-free languages, respectively. The learning of restarting automata was significantly the best method in these experiments. We can say, that the developed learning method is well-generalizing, particularly for non-regular languages.

Beside general nondeterministic lc-R-automata we have proposed their deterministic version in which the meta-instructions are applied in fixed order and on the leftmost position only. Such automata performed similarly as the nondeterministic lc-R-automata but much faster.

References

[1] S. BASOVNÍK, *Learning Restricted Restarting Automata using Genetic Algorithm*. Master thesis, Charles University, Faculty of Mathematics and Physics, Prague, 2010.

[2] P. DUPONT, Regular grammatical inference from positive and negative samples by genetic search: The GIG method. In: *Proc. 2nd International Colloquium on Grammatical Inference – ICGI '94*. LNAI 862, Springer-Verlag, 1994, 236–245.

[3] R. EYRAUD, C. DE LA HIGUERA, J.-C. JANODET, Representing Languages by Learnable Rewriting Systems. In: G. PALIOURAS, Y. SAKAKIBARA (eds.), *ICGI 2004*. LNAI 3264, Springer, Berlin, 2004, 139–150.

[4] D. E. GOLDBERG, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989.

[5] J. ONCINA, P. GARCÍA, Inferring regular languages in polynomial update time. In: N. P. DE LA BLANCA, A. SANFELIU, E. VIDAL (eds.), *Pattern Recognition and Image Analysis*. Machine Perception and Artificial Intelligence 1, World Scientific, Singapore, 1992, 49–61.

[6] F. OTTO, Restarting automata. In: Z. ÉSIK, C. MARTÍN-VIDE, V. MITRANA (eds.), *Recent Advances in Formal Languages and Applications*. Studies in Computational Intelligence 25, Springer, Berlin, 2006, 269–303.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 29 – 32

Netze evolutionärer Prozessoren mit subregulären Filtern

Jürgen Dassow Florin Manea[A] Bianca Truthe

Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
PSF 4120, 39016 Magdeburg
`{dassow,manea,truthe}@iws.cs.uni-magdeburg.de`

Zusammenfassung

In dieser Arbeit entwickeln wir eine Hierarchie von Klassen von Sprachen, die von Netzen evolutionärer Prozessoren erzeugt werden, bei denen die Filter zu speziellen Klassen von regulären Mengen gehören. Wir zeigen, dass mit Filtern aus den Klassen der geordneten, nicht-zählenden, potenzseparierenden, zirkulären, suffixabgeschlossenen, vereinigungsfreien, definiten und kombinatorischen Sprachen die gleichen Sprachen erzeugt werden können wie mit beliebigen regulären Filtern und dass so jede rekursiv-aufzählbare Sprache erzeugt werden kann. Beim Verwenden von Filtern, die nur endliche Sprachen sind, werden nur reguläre Sprachen erzeugt aber nicht jede reguläre Sprache ist erzeugbar. Mit Filtern, die Monoide, nilpotente Sprachen oder kommutative reguläre Sprachen sind, erhalten wir eine andere Sprachklasse; sie enthält nicht-kontextfreie Sprachen aber nicht alle reguläre Sprachen.

1. Definitionen

Wir gehen davon aus, dass der Leser mit grundlegenden Konzepten der Theorie formaler Sprachen vertraut ist (siehe z. B. [4]). Wir geben hier einige Begriffe und Notationen, die in dieser Arbeit verwendet werden.

Es sei V ein Alphabet. Mit V^* bezeichnen wir die Menge aller Wörter über V (einschließlich des Leerwortes λ).

Eine Regelgrammatik ist ein Quadrupel $G = (N, T, P, S)$, wobei N eine endliche Menge von Nichtterminalen ist, T eine endliche und zu N disjunkte Menge von Terminalen ist, P eine endliche Menge von Ableitungsregeln geschrieben in der Form $\alpha \to \beta$ mit $\alpha \in (N \cup T)^* \setminus T^*$ und $\beta \in (N \cup T)^*$ ist sowie $S \in N$ das Startwort (Axiom) ist.

Mit *REG*, *CF* und *RE* bezeichnen wir die Klassen regulärer, kontextfreier bzw. rekursiv-aufzählbarer Sprachen.

Diese Arbeit wurde auf der LATA 2011 vorgestellt.

[A] Auch: Faculty of Mathematics and Computer Science, University of Bucharest, Str. Academiei 14, RO-010014 Bukarest, Rumänien (`flmanea@fmi.unibuc.ro`). Florin Manea wurde von der *Alexander-von-Humboldt-Stiftung* gefördert.

Zu einer Sprache L über einem Alphabet V setzen wir

$$\begin{aligned}\mathit{Comm}(L) &= \{a_{i_1}\dots a_{i_n} \mid a_1\dots a_n \in L,\ n\geq 1,\ \{i_1,i_2,\dots,i_n\}=\{1,2,\dots,n\}\},\\ \mathit{Circ}(L) &= \{vu \mid uv\in L,\ u,v\in V^*\},\\ \mathit{Suf}(L) &= \{v \mid uv \in L,\ u,v\in V^*\}\end{aligned}$$

Wir betrachten die folgenden Einschränkungen regulärer Sprachen. Es seien L eine Sprache und $V = \mathit{alph}(L)$ das Minimalalphabet von L. Wir sagen, die Sprache L ist

- *kombinatorisch* genau dann, wenn sie in der Form $L = V^*A$ für eine Teilmenge $A \subseteq V$ darstellbar ist,
- *definit* genau dann, wenn sie in der Form $L = A\cup V^*B$ mit endlichen Teilmengen A und B von V^* darstellbar ist,
- *nilpotent* genau dann, wenn sie endlich oder die komplementäre Sprache $V^*\setminus L$ endlich ist,
- *kommutativ* genau dann, wenn $L = \mathit{Comm}(L)$ gilt,
- *zirkulär* genau dann, wenn $L = Circ(L)$gilt,
- *suffixabgeschlossen* genau dann, wenn zu jedem Wort $xy \in L$ für Wörter $x,y\in V^*$ auch $y\in L$ gilt (oder $\mathit{Suf}(L) = L$ gilt),
- *nichtzählend* (oder sternfrei) genau dann, wenn es eine Zahl $k\geq 1$ so gibt, dass für beliebige Wörter $x,y,z\in V^*$ das Wort xy^kz genau dann in L liegt, wenn das Wort $xy^{k+1}z$ in L liegt,
- *potenzseparierend* genau dann, wenn zu jedem Wort $x\in V^*$ eine natürliche Zahl $m\geq 1$ so existiert, dass entweder $J_x^m\cap L=\emptyset$ oder $J_x^m\subseteq L$ gilt mit $J_x^m = \{x^n \mid n\geq m\}$,
- *geordnet* genau dann, wenn L von einem endlichen Automaten $\mathcal{A} = (Z,V,\delta,z_0,F)$ akzeptiert wird, wobei $(Z,\preceq)$ eine geordnete Menge ist und für jeden Buchstaben $a\in V$ die Beziehung $z\preceq z'$ die Beziehung $\delta(z,a)\preceq\delta(z',a)$ impliziert,
- *vereinigungsfrei* genau dann, wenn L durch einen regulären Ausdruck beschreibbar ist, der nur aus Konkatenation und Kleene-Abschluss gebildet wird.

Mit *COMB*, *DEF*, *NIL*, *COMM*, *CIRC*, *SUF*, *NC*, *PS*, *ORD* und *UF* bezeichnen wir die Klassen der kombinatorischen, definiten, nilpotenten, regulären kommutativen, regulären zirkulären, regulären suffixabgeschlossenen, regulären nichtzählenden, regulären potenzseparierenden, geordneten bzw. vereinigungsfreien Sprachen. Außerdem bezeichne *MON* die Menge aller Sprachen der Form V^*, wobei V ein Alphabet ist (wir nennen diese Sprachen monoidal). Es sei

$$\mathcal{G} = \{\mathit{FIN},\mathit{MON},\mathit{COMB},\mathit{DEF},\mathit{NIL},\mathit{COMM},\mathit{CIRC},\mathit{SUF},\mathit{NC},\mathit{PS},\mathit{ORD},\mathit{UF}\}.$$

Die Klassen aus $\mathcal{G}$ wurden z. B. in [3] und [5] untersucht.

Wir nennen eine Ableitungsregel $\alpha\to\beta$

- eine Substitutionsregel, falls $|\alpha| = |\beta| = 1$ gilt, und
- eine löschende Regel, falls $|\alpha| = 1$ und $\beta=\lambda$ gelten.

Aus einem Wort v wird ein Wort w abgeleitet, geschrieben als $v\Longrightarrow w$, wenn zwei Wörter x,y und eine Regel $\alpha\to\beta$ so existieren, dass $v = x\alpha y$ und $w = x\beta y$ gelten.

Einfügen wird als Gegenstück zum Löschen angesehen. Wir schreiben $\lambda\to a$, wobei a ein Buchstabe ist. Mittels einer einfügenden Regel $\lambda\to a$ wird aus einem Wort w jedes Wort w_1aw_2 mit $w = w_1w_2$ erzeugt.

Wir definieren nun Netze evolutionärer Prozessoren (kurz NEPs).

Definition 1.1 *Es sei X eine Familie von regulären Sprachen.*

(i) *Ein Netz evolutionärer Prozessoren (der Größe n) mit Filtern aus der Menge X ist ein Tupel*

$$\mathcal{N} = (V, N_1, N_2, \dots, N_n, E, j),$$

wobei
- *V ein endliches Alphabet ist,*
- *$N_i = (M_i, A_i, I_i, O_i)$ für $1 \le i \le n$ ein evolutionärer Prozessor ist, bei dem*
  - *M_i eine Menge von Regeln eines gewissen Typs ist: $M_i \subseteq \{a \to b \mid a, b \in V\}$ oder $M_i \subseteq \{a \to \lambda \mid a \in V\}$ oder $M_i \subseteq \{\lambda \to b \mid b \in V\}$,*
  - *A_i eine endliche Teilmenge von V^* ist,*
  - *I_i und O_i Sprachen über dem Alphabet V aus der Menge X sind,*
- *E eine Teilmenge von $\{1, 2, \dots, n\} \times \{1, 2, \dots, n\}$ ist und*
- *j eine natürliche Zahl mit $1 \le j \le n$ ist.*

(ii) *Eine Konfiguration C von $\mathcal{N}$ ist ein n-Tupel $C = (C(1), C(2), \dots, C(n))$, wobei $C(i)$ für $1 \le i \le n$ eine Teilmenge von V^* ist.*

(iii) *Es seien $C = (C(1), C(2), \dots, C(n))$ und $C' = (C'(1), C'(2), \dots, C'(n))$ zwei Konfigurationen von $\mathcal{N}$. Wir sagen, dass aus der Konfiguration C die Konfiguration C' in einem*
- *Evolutionsschritt abgeleitet wird (geschrieben als $C \Longrightarrow C'$), falls $C'(i)$ für alle Zahlen i mit $1 \le i \le n$ aus allen Wörtern $w \in C(i)$, auf die keine Regel aus M_i anwendbar ist, und aus allen Wörtern w, zu denen es ein Wort $v \in C(i)$ und eine Regel $p \in M_i$ derart gibt, dass $v \Longrightarrow_p w$ gilt, besteht, und in einem*
- *Kommunikationsschritt abgeleitet wird (geschrieben als $C \vdash C'$), falls für $1 \le i \le n$*

$$C'(i) = (C(i) \setminus O_i) \cup \bigcup_{(k,i) \in E} (C(k) \cap O_k \cap I_i)$$

  gilt.

Die Berechnung eines evolutionären Netzes $\mathcal{N}$ ist jene Folge von Konfigurationen $C_t = (C_t(1), C_t(2), \dots, C_t(n))$ mit $t \ge 0$ derart, dass
- *$C_0 = (A_1, A_2, \dots, A_n)$ gilt,*
- *aus C_{2t} die Konfiguration C_{2t+1} in einem Evolutionsschritt abgeleitet wird (für $t \ge 0$) und*
- *aus C_{2t+1} die Konfiguration C_{2t+2} in einem Kommunikationsschritt abgeleitet wird (für $t \ge 0$).*

(iv) *Die von einem Netz $\mathcal{N}$ erzeugte Sprache $L(\mathcal{N})$ is definiert als*

$$L(\mathcal{N}) = \bigcup_{t \ge 0} C_t(j),$$

wobei $C_t = (C_t(1), C_t(2), \dots, C_t(n))$, $t \ge 0$ die Berechnung von $\mathcal{N}$ ist.

Zu einer Klasse $X \subseteq$ *REG* bezeichnen wir die Klasse der Sprachen, die von Netzen evolutionärer Prozessoren erzeugt werden, bei denen alle Filter zur Klasse X gehören, mit $\mathcal{E}(X)$.

Es ist offensichtlich, dass für zwei Teilklassen X und Y von *REG* mit $X \subseteq Y$ auch die Inklusion $\mathcal{E}(X) \subseteq \mathcal{E}(Y)$ gilt.

Es ist bekannt, dass $\mathcal{E}(\mathit{REG}) = \mathit{RE}$ gilt (siehe z. B. [1]).

2. Ergebnisse

Die erhaltenen Resultate sind in folgendem Diagramm dargestellt.

$$\begin{tikzcd}
 & & \begin{array}{c} RE = \mathcal{E}(REG) = \mathcal{E}(PS) = \mathcal{E}(NC) \\ = \mathcal{E}(ORD) = \mathcal{E}(SUF) = \mathcal{E}(CIRC) \\ = \mathcal{E}(DEF) = \mathcal{E}(COMB) = \mathcal{E}(UF) \end{array} & & \\
CF \arrow[urr] & & & & \\
REG \arrow[u] & & & & \begin{array}{c} \mathcal{E}(MON) = \mathcal{E}(NIL) \\ = \mathcal{E}(COMM) \end{array} \arrow[uull] \\
\mathcal{E}(FIN) \arrow[u] \arrow[urrrr] & & NIL \arrow[ull] \arrow[urr] & & COMM \arrow[ullll] \arrow[u] \\
FIN \arrow[u] \arrow[urr] & & & & MON \arrow[ull] \arrow[ullll] \arrow[u]
\end{tikzcd}$$

Eine gerichtete Kante von einer Sprachklasse X zu einer Sprachklasse Y steht für die echte Inklusion $X \subset Y$.

Die in dieser Arbeit betrachteten Klassen subregulärer Sprachen sind über kombinatorische oder algebraische Eigenschaften der Sprachen definiert. In [2] wurden Unterklassen von *REG* betrachtet, die über Beschreibungskomplexität definiert sind. Es sei REG_n die Menge der regulären Sprachen die von deterministischen endlichen Automaten akzeptiert werden. Dann gilt

$$REG_1 \subset REG_2 \subset REG_3 \subset \cdots \subset REG_n \subset \cdots \subset REG.$$

Da die Klasse $\mathcal{E}(MON)$ nicht-semilineare Sprachen enthält, erhalten wir nach [2] die Beziehungen

$$\mathcal{E}(REG_1) \subset \mathcal{E}(MON) \subset \mathcal{E}(REG_2) = \mathcal{E}(REG_3) = \cdots = RE$$

und aus [2] außerdem die Unvergleichbarkeit von $\mathcal{E}(REG_1)$ mit *REG* und *CF*.

Literatur

[1] J. CASTELLANOS, C. MARTÍN-VIDE, V. MITRANA, J. M. SEMPERE, Networks of Evolutionary Processors. *Acta Informatica* **39** (2003) 6–7, 517–529.

[2] J. DASSOW, B. TRUTHE, On networks of evolutionary processors with filters accepted by two-state-automata. *Fundamenta Informaticae* To appear.

[3] I. M. HAVEL, The theory of regular events II. *Kybernetika* **5** (1969) 6, 520–544.

[4] G. ROZENBERG, A. SALOMAA, *Handbook of Formal Languages*. Springer-Verlag, Berlin, 1997.

[5] B. WIEDEMANN, *Vergleich der Leistungsfähigkeit endlicher determinierter Automaten*. Diplomarbeit, Universität Rostock, 1978.

Über CD Grammatiksysteme mit Valenzregeln

Henning Fernau$^{(A)}$ Ralf Stiebe$^{(B)}$

$^{(A)}$Abteilung Informatik, Fachbereich IV, Universität Trier
fernau@uni-trier.de

$^{(B)}$Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg
stiebe@iws.cs.uni-magdeburg.de

Zusammenfassung

In dieser Arbeit untersuchen wir die Mächtigkeit von CDGS mit Valenzregeln.

1. Einleitung

Grammatiken mit Valenzregeln und kooperierende verteilte Grammatiksysteme (CDGS) wurden in den Arbeiten [2, 7] im Abstand von etwa 10 Jahren eingeführt. Eine Kombination dieser Mechanismen gestattet die Betrachtung einer Vielzahl neuer Sprachfamilien; dieser Aufgabe widmeten wir uns in [5]. Hier wollen wir nur einen Abriss der Ergebnisse skizzieren. Für ein CDGS G mit Valenzregeln über dem Monoid $\mathbf{M} = (M, \circ, e)$ legen wir fest:

$$\begin{aligned}
L_f^{\text{int}}(G) := \{\, w \in T^* \mid (S,e) \Rightarrow_{i_1}^{f} (\alpha_1, e) \Rightarrow_{i_2}^{f} \ldots \Rightarrow_{i_{m-1}}^{f} (\alpha_{m-1}, e) \Rightarrow_{i_m}^{f} (\alpha_m, e) \\
\text{mit } m \geq 1,\ 1 \leq i_j \leq n, 1 \leq j \leq m, \alpha_m = w \,\}. \\
L_f^{\text{ext}}(G) := \{\, w \in T^* \mid (S,e) \Rightarrow_{i_1}^{f} (\alpha_1, x_1) \Rightarrow_{i_2}^{f} \ldots \Rightarrow_{i_{m-1}}^{f} (\alpha_{m-1}, x_{m-1}) \Rightarrow_{i_m}^{f} \\
(\alpha_m = w, x_m = e) \text{ mit } m \geq 1,\ 1 \leq i_j \leq n, x_j \in M, 1 \leq j \leq m \,\}.
\end{aligned}$$

Hierbei ist f einer der bekannten CDGS-Modi. Die entsprechenden Sprachfamilien notieren wir mit $\mathcal{L}_{\text{int}}(\text{CD}, X, f, \mathbf{M})$ bzw. $\mathcal{L}_{\text{ext}}(\text{CD}, X, f, \mathbf{M})$, wobei X den Kernregeltyp angibt (regulär, kontextfrei). Allgemein können wir festhalten:

Lemma 1.1 *Für $X \in \{\text{REG}, \text{CF}\}$, $Y \in \{\text{int}, \text{ext}\}$, $k \in \mathbb{N}$ und Monoide $\mathbf{M} = (M, \circ, e)$ gilt:*

$$\mathcal{L}_Y(\text{CD}, X, *, \mathbf{M}) \subseteq \mathcal{L}_Y(\text{CD}, X, \geq k, \mathbf{M}).$$

Dem mit den genannten Mechanismen vertrauten Leser dürfte ein Beispiel am ehesten helfen, unsere Notationen zu verstehen.

Beispiel 1.2 *Betrachte das System $G = (\{S, A, B\}, \{a, b, c\}, \{P_1, P_2, P_3, P_4\})$, wobei*

$$\begin{aligned}
P_1 &= \{(S \to S, +1), (S \to AcB, -1)\}, \\
P_2 &= \{(A \to aAa, +1), (B \to aBa, -1)\}, \\
P_3 &= \{(A \to bAb, +1), (B \to bBb, -1)\}, \text{ sowie} \\
P_4 &= \{(A \to c, +1), (B \to c, -1)\}.
\end{aligned}$$

Als den Valenzen zugrunde liegendes Monoid betrachten wir hier $(\mathbb{Z},+)$. Für die im $=2$-Modus mit Zwischentests erzeugte Sprache gilt:

$$L^{\text{int}}_{=2}(G) = \{wcw^R cwcw^R \mid w \in \{a,b\}^*\}.$$

Die durch G mit Schlusstests erzeugte Sprache ist wesentlich größer.

2. CDGS mit Zwischentests

Der reguläre Fall wurde bereits von Sorina Vicolov(-Dumitrescu) [8] unter der Bezeichnung CDGS *mit Registern* untersucht. Wir fassen unsere Ergebnisse mit ihren zusammen:

Satz 2.1 *Für beliebige Monoide* $\mathbf{M} = (M,\circ,e)$ *gilt:*
– $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbf{M})$ *ist eine volle abstrakte Familie von Sprachen.*
– *Ist* $\mathbf{M}$ *von E endlich erzeugt, so ist* $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbf{M})$ *gerade die von $I(E)$ erzeugte volle abstrakte Familie von Sprachen.*

Hierbei fassen wir E als Alphabet auf und $I(E)$ sei die Menge aller $w \in E^*$, welche zur Identität e evaluieren, wenn wir die Konkatenation als Monoidoperation $\circ$ interpretieren.

Folgerung 2.2 • *Für endliche Monoide* $\mathbf{M}$ *gilt:* $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbf{M}) = \mathcal{L}(\text{REG})$.

• $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbb{Z}^n)$ *ist eine volle AFL, erzeugt durch:*

$$L_n = \{w \in \{a_1,\dots,a_n,b_1\dots,b_n\}^* \mid |w|_{a_i} = |w|_{b_i}, 1 \le i \le n\}.$$

Insbesondere ist $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbb{Z})$ *eine echte Teilklasse der Einzählersprachen [1].*

• *Für die freie Gruppe F_k mit $k \ge 2$ Erzeugern gilt:*
$\mathcal{L}_{int}(\text{CD},\text{REG},*,F_k) = \mathcal{L}(\text{REG},F_k) = \mathcal{L}(\text{CF})$ *[6].*

Satz 2.3 $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbb{Z}^n)$ *und* $\mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbb{Q}_+)$ *sind nicht schnittabgeschlossen.*

Beweis. Sei $L = \{a^{n+1}ba^nb \mid n \ge 0\}$. Mit L sind $L_1 = L^*$ und $L_2 = a^*bL^*b$ in $\mathcal{L}(\text{CD},\text{REG},*,\mathbb{Z})$, aber die Längenmenge $\{\frac{n(n+1)}{2} \mid n \ge 1\}$ von $L_1 \cap L_2 = \{a^nba^{n-1}b\cdots abb \mid n \ge 1\}$ ist nicht semilinear, also gilt $L_1 \cap L_2 \notin \mathcal{L}_{\text{int}}(\text{CD},\text{REG},*,\mathbb{Q}_+)$. □

Satz 2.4 *Für $f \in \{t,\ge k\}$ und* $\mathbf{M} = (M,\circ,e)$ *gilt:* $\mathcal{L}_{int}(\text{CD},\text{REG},f,\mathbf{M}) = \mathcal{L}_{int}(\text{CD},\text{REG},*,\mathbf{M})$.

Satz 2.5 *Für $f \in \{=k,\le k\}$ und* $\mathbf{M} = (M,\circ,e)$ *gilt:* $\mathcal{L}_{int}(\text{CD},\text{REG},f,\mathbf{M}) = \mathcal{L}(\text{REG})$.

Satz 2.6 *Für jedes Monoid* $\mathbf{M}$ *und alle $k \ge 2$ erhalten wir:*

$$\mathcal{L}_{int}(\text{CD},\text{CF},*,\mathbf{M}) \subseteq \mathcal{L}_{int}(\text{CD},\text{CF},\ge k,\mathbf{M}) \subseteq \mathcal{L}_{int}(\text{CD},\text{CF},*,\mathbf{M}\times\mathbb{Z}).$$

Beweis. Konstruktionsskizze für die zweite Inklusion: Zu einem $\mathbf{M}$-Valenz CDGS $G = (N,T,S,P_1,\dots,P_k)$ sei $\mathbf{M}\times\mathbb{Z}$-Valenz CDGS $G' = (N',T,S',P'_0,P'_1,\dots,P'_k)$: $N' = N \cup \{S',X_1,\dots,X_k\}$, $P'_0 = \{(S' \to SX_i,(e,0)) \mid 1 \le i \le k\}$, $P'_i = \{(A \to \alpha,(m,1)) \mid (A \to \alpha,m) \in P_i\} \cup \{(X_i \to X_j,(e,-k)) \mid 1 \le j \le k \wedge i \ne j\} \cup \{(X_i \to X_i,(e,-1)),(X_i \to \lambda,(e,-k))\}$, für $1 \le i \le k$. □

Folgerung 2.7 *Für $k \geq 2$ gilt:* $\mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, \geq k, \mathbb{Z}) \supseteq \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$.

Satz 2.8 *Für $k \geq 2$ und $f \in \{\leq k, = k\}$ können wir zeigen:* $\mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Z}) = \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$.

Lemma 2.9 *Für bel. Monoide* $\mathbf{M}$ *gilt:* $\mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, = 1, \mathbf{M}) = \mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, \leq 1, \mathbf{M}) = \mathcal{L}(\mathrm{CF})$.

Satz 2.10 $\mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, t, \mathbb{Q}_+) = \mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, t, \mathbb{Z}^n) = \mathcal{L}_{int}(\mathrm{CD}, \mathrm{CF}, t, \mathbb{Z}) = \mathcal{L}(\mathrm{RE})$ *mit* $n \geq 1$.

Der Beweis erfolgt über ET0L-Systeme mit Valenztests [4].

3. CDGS mit Schlusstests

Valenztransduktionen wurden in [4] eingeführt, rationale Transduktionen verallgemeinernd.

Satz 3.1 *Für alle Modi f ist jede Sprache $L \in \mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Z}^k)$ das Bild einer Sprache $L' \in \mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f)$ unter einer $\mathbb{Z}^k$-Valenztransduktion.*

Satz 3.2 *Für $k \geq 2$, $n \geq 1$ und $f \in \{\geq k, = k\}$ erhalten wir:*
$\mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Q}_+) = \mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Z}^n) = \mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Z}) = \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$.

Folgerung 3.3 *Für $k \geq 2$ und $f \in \{\geq k, = k\}$ ist $\mathcal{L}(\mathrm{MAT}, \mathrm{CF})$ der Abschluss von $\mathcal{L}(\mathrm{CD}, \mathrm{CF}, f)$ unter $\mathbb{Z}$-Valenztransduktionen. Daher enthält $\mathcal{L}(\mathrm{CD}, \mathrm{CF}, f)$ Sprachen, die nicht semilinear sind.*

Satz 3.4 *Für beliebige Monoide* $\mathbf{M}$ *und Modi f kennzeichnet $\mathcal{L}_{ext}(\mathrm{CD}, \mathrm{REG}, f, \mathbf{M})$ die regulären* $\mathbf{M}$*-Valenzsprachen.*

Satz 3.5 *Für $k \geq 1$, $n \geq 1$ und $f \in \{\leq k, *, =1\}$ kennzeichnet $\mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Z}^n)$ die kontextfreien $\mathbb{Z}^n$-Valenzsprachen.*

Folgerung 3.6 *Für $k \geq 1$ und $f \in \{\leq k, *, =1\}$ gilt:* $\mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, f, \mathbb{Q}_+) = \mathcal{L}(\mathrm{UV}, \mathrm{CF})$.

ET0L-Systeme mit Valenzen wurden in [4] definiert.

Satz 3.7 *Für $n \geq 1$ gilt: $\mathcal{L}_{ext}(\mathrm{CD}, \mathrm{CF}, t, \mathbb{Z}^n)$ kennzeichnet die von ET0L-Systemen mit $\mathbb{Z}^n$-Valenzen erzeugten Sprachen.*

4. Ausblick und Zusammenfassung

Bekannt ([3, Lemma 14]): $\mathcal{L}_{\mathrm{int}}(\mathrm{CD}, \mathrm{CF}, *, \mathbb{Z}) \supseteq \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$. Ob diese Inklusion echt ist oder auch $\mathcal{L}_{\mathrm{int}}(\mathrm{CD}, \mathrm{CF}, *, \mathbb{Z}^n) \subseteq \mathcal{L}_{\mathrm{int}}(\mathrm{CD}, \mathrm{CF}, *, \mathbb{Z}^{n+1})$, bleibt unklar; Satz 2.6 mag hilfreich sein. Folgerung 3.3 mag einen Hinweis für die Frage liefern, ob die Inklusionen $\bigcup_{k \in \mathbb{N}} \mathcal{L}(\mathrm{CD}, \mathrm{CF}, = k) \subseteq \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$ bzw. $\bigcup_{k \in \mathbb{N}} \mathcal{L}(\mathrm{CD}, \mathrm{CF}, \geq k) \subseteq \mathcal{L}(\mathrm{MAT}, \mathrm{CF})$ echt sind oder nicht. Umgekehrt ergeben sich zahlreiche neue Fragestellungen, z.B.:

– Bei den Zwischentests sind die Moden $\geq k$ nicht voll klassifiziert.
– Fragen der Beschreibungskomplexität wurden überhaupt nicht betrachtet.
– Ebensowenig wurden hybride CDGS mit Valenzen untersucht.

Für $\mathbb{Z}$-Valenzen fassen wir zusammen; auch dabei werden offene Fragen deutlich:

Satz 4.1 *Für $k, k', k'', k''' \geq 2$ gilt:*

$$\begin{aligned}
\mathcal{L}(\mathrm{CF}) &= \mathcal{L}_{int}(\mathrm{CD},\mathrm{CF},=1,\mathbb{Z})\\
&\subsetneq \mathcal{L}_{ext}(\mathrm{CD},\mathrm{CF},=1,\mathbb{Z}) = \mathcal{L}_{ext}(\mathrm{CD},\mathrm{CF},\leq k,\mathbb{Z})\\
&\subsetneq \mathcal{L}(\mathrm{MAT},\mathrm{CF}) = \mathcal{L}_{int}(\mathrm{CD},\mathrm{CF},\leq k',\mathbb{Z}) = \mathcal{L}_{ext}(\mathrm{CD},\mathrm{CF},\geq k'',\mathbb{Z})\\
&\subseteq \mathcal{L}_{int}(\mathrm{CD},\mathrm{CF},\geq k''',\mathbb{Z})\\
&\subseteq \mathcal{L}_{int}(\mathrm{CD},\mathrm{CF},t,\mathbb{Z}) = \mathcal{L}(\mathrm{RE}).
\end{aligned}$$

In [5] haben wir außerdem den Effekt von Valenzen bei CDGS studiert, welche nicht einzelnen Regeln, sondern ganzen Komponenten zugeordnet werden. Dieses Konzept entspricht in natürlicher Weise den in [4] betrachteten Tafelvalenzen. Erstaunlicherweise übertragen sich die Ergebnisse überhaupt nicht leicht. Insbesondere bleibt unklar, ob CDGS mit **M**-Komponentenvalenzen im t-Modus die Sprachklasse kennzeichnen, welche durch ET0L-Systeme mit **M**-Tafelvalenzen beschrieben wird.

Literatur

[1] L. BOASSON, An Iteration Theorem for One-Counter Languages. In: *STOC*. ACM, 1971, 116–120.

[2] E. CSUHAJ-VARJÚ, J. DASSOW, On cooperating/distributed grammar systems. *J. Inf. Process. Cybern. EIK (formerly Elektron. Inf.verarb. Kybern.)* **26** (1990) 1/2, 49–63.

[3] J. DASSOW, G. PĂUN, Cooperating/distributed grammar systems with registers. *Foundations of Control Engineering* **15** (1990) 1, 1–38.

[4] H. FERNAU, R. STIEBE, Valences in Lindenmayer systems. *Fundamenta Informaticae* **45** (2001), 329–358.

[5] H. FERNAU, R. STIEBE, On the expressive power of valences in cooperating distributed grammar systems. In: J. KELEMEN, A. KELEMENOVÁ (eds.), *Computation, Cooperation, and Life*. LNCS 6610, Springer, 2011, 90–106.

[6] V. MITRANA, R. STIEBE, Extended Finite Automata over Groups. *Discrete Applied Mathematics* **108** (2001), 287–300.

[7] G. PĂUN, A new generative device: valence grammars. *Rev. Roumaine Math. Pures Appl.* **XXV** (1980) 6, 911–924.

[8] S. VICOLOV, Cooperating/distributed grammar systems with registers: the regular case. *Computers and Artificial Intelligence* **12** (1993) 1, 89–98.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 37 – 41

A General Framework for Regulated Rewriting

Rudolf Freund[A]

[A]Fakultät für Informatik, Technische Universität Wien
Favoritenstr. 9–11, A-1040 Wien, Austria
rudi@emcc.at

Abstract

The monograph on regulated rewriting by Jürgen Dassow and Gheorghe Păun [2] gave a first comprehensive overview on the basic concepts of regulated rewriting, especially for the string case. As it turned out later, many of the mechanisms considered there for guiding the application of productions/rules can also be applied to other objects than strings, e.g., to n-dimensional arrays or graphs or multisets. For comparing the generating power of grammars working in the sequential derivation mode, we introduce a general model for sequential grammars, with regulating mechanisms only based on the applicability of rules, without any reference to the underlying objects the rules are working on, and establish some general results not depending on the type of the underlying grammars.

1. A General Model for Sequential Grammars

For the basic notions and concepts of formal language theory the reader is referred to the monographs and handbooks in this area [2].

A *(sequential) grammar* G is a construct $(O, O_T, w, P, \Longrightarrow_G)$ where O is a set of *objects*, $O_T \subseteq O$ is a set of *terminal objects*, $w \in O$ is the *axiom (start object)*, P is a finite set of *rules*, and $\Longrightarrow_G \subseteq O \times O$ is the *derivation relation* of G. We assume that each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_G$ fulfilling at least the following conditions: (i) for each object $x \in O$, $(x, y) \in \Longrightarrow_p$ for only finitely many objects $y \in O$; (ii) there exists a finitely described mechanism as, for example, a Turing machine, which, given an object $x \in O$, computes all objects $y \in O$ such that $(x, y) \in \Longrightarrow_p$. A rule $p \in P$ is called *applicable* to an object $x \in O$ if and only if there exists at least one object $y \in O$ such that $(x, y) \in \Longrightarrow_p$; we also write $x \Longrightarrow_p y$. The derivation relation $\Longrightarrow_G$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_G = \cup_{p \in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_G$ is denoted by $\overset{*}{\Longrightarrow}_G$.

In the following we shall consider different types of grammars depending on the components of G (where the set of objects O is infinite, e.g., V^*, the set of strings over the alphabet V), especially with respect to different types of rules (e.g., context-free string rules). Some specific conditions on the elements of G, especially on the rules in P, may define a special

[A]The results presented in this paper originate from the author's stay at the Otto-von-Guericke-Universitä in Magdeburg around twenty years ago, based on many interesting and fruitful discussions with Jürgen Dassow as well as Gheorghe Păun and later on, among others, especially also with Henning Fernau and Marion Oswald. A more detailed presentation and further results can be found in [3].

type X of grammars which then will be called *grammars of type* X. The *language generated by* G is the set of all terminal objects (we also assume $v \in O_T$ to be decidable for every $v \in O$) derivable from the axiom, i.e., $L(G) = \left\{ v \in O_T \mid w \overset{*}{\Longrightarrow}_G v \right\}$. The family of languages generated by grammars of type X is denoted by $\mathcal{L}(X)$. A type X of grammars is called a *type with unit rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G' = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'}\right)$ of type X such that $\Longrightarrow_G \ \subseteq \ \Longrightarrow_{G'}$ and (i) $P^{(+)} = \left\{ p^{(+)} \mid p \in P \right\}$, (ii) for all $x \in O$, $p^{(+)}$ is applicable to x if and only if p is applicable to x, and (iii) for all $x \in O$, if $p^{(+)}$ is applicable to x, the application of $p^{(+)}$ to x yields x back again; X is called a *type with trap rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G' = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'}\right)$ of type X such that $\Longrightarrow_G \ \subseteq \ \Longrightarrow_{G'}$ and (i) $P^{(-)} = \left\{ p^{(-)} \mid p \in P \right\}$, (ii) for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and (iii) for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

Instead of the common notation $G_S = (N, T, w, P)$, in the general framework as defined above, a *string grammar* G_S is represented as $\left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$, where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, $w \in (N \cup T)^+$, P is a finite set of *rules* of the form $u \to v$ with $u \in V^*$ (for generating grammars, $u \in V^+$) and $v \in V^*$ (for accepting grammars, $v \in V^+$), with $V := N \cup T$; the derivation relation for $u \to v \in P$ is defined by $xuy \Longrightarrow_{u \to v} xvy$ for all $x, y \in V^*$, thus yielding the well-known derivation relation $\Longrightarrow_{G_S}$ for the string grammar G_S.

As special types of string grammars we consider string grammars with arbitrary rules, context-free rules of the form $A \to v$ with $A \in N$ and $v \in V^*$, and (right-)regular rules of the form $A \to v$ with $A \in N$ and $v \in TN \cup \{\lambda\}$. The corresponding types of grammars are denoted by ARB, CF, and REG, thus yielding the families of languages $\mathcal{L}(ARB)$, i.e., the family of recursively enumerable languages, as well as $\mathcal{L}(CF)$, and $\mathcal{L}(REG)$, i.e., the families of context-free, and regular languages, respectively.

Observe that the types ARB and CF are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where $F \notin T$ is a new symbol – the trap symbol), whereas the type REG is not a type with unit rules. Therefore, we also consider the type REG' with regular rules of the most general form $A \to v$ with $A \in N$ and $v \in T^*N \cup T^*$ and axioms $w \in (N \cup T)^*$, and denote the corresponding family of languages by $\mathcal{L}(REG')$.

A *multiset grammar [1]* G_m is of the form $\left((N \cup T)^o, T^o, w, P, \Longrightarrow_{G_m}\right)$ where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, w is a non-empty multiset over V, $V := N \cup T$, and P is a (finite) set of multiset rules yielding a derivation relation $\Longrightarrow_{G_m}$ on the multisets over V; the application of the rule $u \to v$ to a multiset x has the effect of replacing the multiset u contained in x by the multiset v. For the multiset grammar G_m we also write $(N, T, w, P, \Longrightarrow_{G_m})$.

As special types of multiset grammars, which became of new interest in the emerging field of P systems [5], we consider multiset grammars with *arbitrary* rules, *context-free* rules of the form $A \to v$ with $A \in N$ and $v \in V^o$, and *regular* rules of the form $A \to v$ with $A \in N$ and $v \in T^oN \cup T^o$; the corresponding types X of multiset grammars are denoted by $mARB$, mCF, and $mREG$, thus yielding the families of multiset languages $\mathcal{L}(X)$. Observe that all these types $mARB$, mCF, and $mREG$ are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where F is a new symbol – the trap symbol). Even with arbitrary multiset rules, it is not possible to get $Ps(\mathcal{L}(ARB))$ [4]:

$$\begin{aligned} Ps(\mathcal{L}(REG)) &= \mathcal{L}(mREG) = \mathcal{L}(mCF) = Ps(\mathcal{L}(CF)) \\ &\subsetneqq \mathcal{L}(mARB) \subsetneqq Ps(\mathcal{L}(ARB)). \end{aligned}$$

Regulating mechanisms

A *graph-controlled grammar* (with appearance checking) of type X is a construct $G_{GC} = \left(G, g, H_i, H_f, \Longrightarrow_{GC}\right)$ where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type,X; $g = (H, E, K)$ is a labeled graph where H is the set of node labels identifying the nodes of the graph in a one-to-one manner, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g, $H_i \subseteq H$ is the set of initial labels, and $H_f \subseteq H$ is the set of final labels. The derivation relation $\Longrightarrow_{GC}$ is defined based on $\Longrightarrow_G$ and the control graph g as follows: For any $i, j \in H$ and any $u, v \in O$, $(u, i) \Longrightarrow_{GC} (v, j)$ if and only if either (i) $u \Longrightarrow_p v$ by some rule $p \in K(i)$ and $(i, Y, j) \in E$ *(success case)*, or (ii) $u = v$, no $p \in K(i)$ is applicable to u, and $(i, N, j) \in E$ *(failure case)*. The language generated by G_{GC} is defined by $L(G_{GC}) = \left\{ v \in O_T \mid (w, i) \Longrightarrow_{G_{GC}}^* (v, j),\ i \in H_i, j \in H_f \right\}$. If $H_i = H_f = H$, then G_{GC} is called a *programmed grammar*. The families of languages generated by graph-controlled and programmed grammars of type X are denoted by $\mathcal{L}(X\text{-}GC_{ac})$ and $\mathcal{L}(X\text{-}P_{ac})$, respectively. If the set E contains no edges of the form (i, N, j), then the graph-controlled grammar is said to be *without appearance checking*; the corresponding families of languages are denoted by $\mathcal{L}(X\text{-}GC)$ and $\mathcal{L}(X\text{-}P)$, respectively. The notion *without appearance checking* (*ac* for short) comes from the fact that in the original definition the appearance of the non-terminal symbol on the left-hand side of a context-free rule was checked; in our general model, the notion *without applicability checking* would even be more adequate.

A *matrix grammar* of type X is a construct $G_M = \left(G, M, F, \Longrightarrow_{G_M}\right)$ where the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, M is a finite set of sequences of the form $(p_1, \ldots, p_n)$, $n \geq 1$, of rules in P, and $F \subseteq P$. For $w, z \in O$ we write $w \Longrightarrow_{G_M} z$ if there are a matrix $(p_1, \ldots, p_n)$ in M and objects $w_i \in O$, $1 \leq i \leq n+1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either (i) $w_i \Longrightarrow_G w_{i+1}$ or (ii) $w_i = w_{i+1}$, p_i is not applicable to w_i, and $p_i \in F$. $L(G_M) = \left\{ v \in O_T \mid w \Longrightarrow_{G_M}^* v \right\}$ is the language generated by G_M. The family of languages generated by matrix grammars of type X is denoted by $\mathcal{L}(X\text{-}MAT_{ac})$. If the set F is empty, then the grammar is said to be *without appearance checking*; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}MAT)$.

A *random-context grammar* G_{RC} *of type* X is a construct $\left(G, P', \Longrightarrow_{G_{RC}}\right)$ where the grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is of type X, P' is a set of rules of the form (q, R, Q) where $q \in P$, $R \cup Q \subseteq P$, $\Longrightarrow_{G_{RC}}$is the derivation relation assigned to G_{RC} such that for any $x, y \in O$, $x \Longrightarrow_{G_{RC}} y$ if and only if for some rule $(q, R, Q) \in P'$, $x \Longrightarrow_q y$ and, moreover, all rules from R are applicable to x as well as no rule from Q is applicable to x. A random-context grammar $G_{RC} = \left(G, P', \Longrightarrow_{G_{RC}}\right)$ of type X is called a *grammar with permitting contexts of type* X if for all rules (q, R, Q) in P' we have $Q = \emptyset$, i.e., we only check for the applicability of the rules in R. G_{RC} is called a *grammar with forbidden contexts of type* X if for all rules (q, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q. The language generated by G_{RC} is $L(G_{RC}) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_{RC}} v\right\}$. The families of languages generated by random context grammars, grammars with permitting contexts, and grammars with forbidden contexts of type X are denoted by $\mathcal{L}(X\text{-}RC)$, $\mathcal{L}(X\text{-}pC)$, and $\mathcal{L}(X\text{-}fC)$, respectively.

An *ordered grammar* G_O *of type* X is a construct $\left(G, <, \Longrightarrow_{G_O}\right)$ where the underlying greammar $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X and $<$ is a partial order relation on the rules from P. The derivation relation assigned to G_O is $\Longrightarrow_{G_O}$ where for any $x, y \in O$, $x \Longrightarrow_{G_O} y$ if and only if for some rule $q \in P$ $x \Longrightarrow_q y$ and, moreover, no rule p from P with $p > q$ is applicable to x. $L(G_O) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_O} v\right\}$ is the language generated by G_O. The family of languages generated by ordered grammars of type X is denoted by $\mathcal{L}(X\text{-}O)$.

2. Results

Based on the definitions given in the preceding section, the following relations between the specific regulating mechanisms can be shown, most of them even for arbitrary types X:

Theorem 2.1 *The following inclusions hold*

— *for any arbitrary type* X,
- - *for any type* X *with unit rules,*
... *for any type* X *with unit rules and trap rules:*

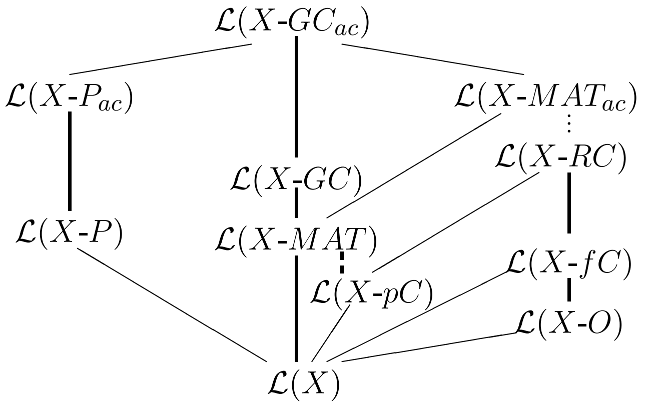

$\mathcal{L}(X\text{-}GC_{ac})$
$\mathcal{L}(X\text{-}P_{ac})$
$\mathcal{L}(X\text{-}MAT_{ac})$
$\mathcal{L}(X\text{-}GC)$
$\mathcal{L}(X\text{-}RC)$
$\mathcal{L}(X\text{-}P)$
$\mathcal{L}(X\text{-}MAT)$
$\mathcal{L}(X\text{-}fC)$
$\mathcal{L}(X\text{-}pC)$
$\mathcal{L}(X\text{-}O)$
$\mathcal{L}(X)$

Figure 1: Hierarchy of control mechanisms for grammars of type X.

Special results for strings

The regulating mechanisms considered in this paper cannot add any generating power to regular grammars, i.e., for any $Y \in \{O, pC, fC, RC, MAT, MAT_{ac}, P, P_{ac}, GC, GC_{ac}\}$, we have

$\mathcal{L}(REG\text{-}Y) = \mathcal{L}(REG)$, as any non-terminal string derivable from the start symbol in a regulated regular grammar contains exactly one non-terminal symbol that is able to store all necessary informations needed to guide the derivation. Yet if we allow the axiom to be an arbitrary string, then the situation changes completely: for example, take the even non-context-free language $L = \{a^n b^n c^n \mid n \geq 0\}$; then $L \in \mathcal{L}(REG'\text{-}Y)$.

Moreover, it is well-known [2] that $\mathcal{L}(CF\text{-}RC) = \mathcal{L}(ARB)$. As CF is a type with unit rules and trap rules, according to Theorem 2.1 we immediately infer $\mathcal{L}(CF\text{-}Y) = \mathcal{L}(ARB)$ for $Y \in \{RC, MAT_{ac}, GC_{ac}\}$ without needing any specific further proofs.

Special Results for Multisets

As in the case of multisets the structural information contained in the sequence of symbols cannot be used, arbitrary multiset rules are not sufficient for obtaining all sets in $Ps(\mathcal{L}(ARB))$. Yet we can easily show that using a partial order relation on the rules is sufficient to obtain computational completeness; in connection with the general results obtained above, we obtain the following:

Theorem 2.2 *For any* $Y \in \{O, fC, MAT_{ac}, GC_{ac}\}$, $\mathcal{L}(mARB\text{-}Y) = Ps(\mathcal{L}(ARB))$.

References

[1] M. CAVALIERE, R. FREUND, M. OSWALD, , D. SBURLAN, Multiset random context grammars, checkers, and transducers. *Theoretical Computer Science* **372 (2-3)** (2007), 136–151.

[2] J. DASSOW, GH. PĂUN, *Regulated Rewriting in Formal Language Theory*. Springer, Berlin, 1989.

[3] R. FREUND, M. KOGLER, M. OSWALD, A general framework for regulated rewriting based an the applicability of rules. In: J. KELEMEN, A. KELEMENOVÁ (eds.), *Computation, Cooperation, and Life. Essays Dedicated to Gheorghe Păun on the Occasion of His 60th Birthday*. LNCS Festschrift 6610, Springer, 2011, 35–53.

[4] M. KUDLEK, C. MARTÍN-VIDE, GH. PĂUN, Toward a formal macroset theory. In: C. S. CALUDE, GH. PĂUN, G. ROZENBERG, A. SALOMAA (eds.), *Multiset Processing – Mathematical, Computer Science and Molecular Computing Points of View*. LNCS 2235, Springer, 2001, 123–134.

[5] GH. PĂUN, G. ROZENBERG, A. SALOMAA, *The Oxford Handbook of Membrane Computing*. Oxford University Press, Oxford (U.K.), 2010.

Entscheidungsprobleme für erweiterte reguläre Ausdrücke

Dominik D. Freydenberger

Institut für Informatik, Goethe-Universität, Frankfurt am Main
`freydenberger@em.uni-frankfurt.de`

Zusammenfassung

Die meisten modernen Implementierungen von regulären Ausdrücken gestatten die Verwendung von Variablen (oder Rückreferenzen). Die daraus entstehenden erweiterten regulären Ausdrücke (die in der Literatur auch als praktische reguläre Ausdrücke, rewbr oder regex bezeichnet werden) können nichtreguläre Sprachen erzeugen, wie zum Beispiel die Sprache aller Wörter der Form ww.

Die vorliegende Arbeit zeigt, dass erweiterte reguläre Ausdrücke weder in Bezug auf ihre Länge, noch in Bezug auf ihre Variablenzahl, berechenbar minimiert werden können, und dass der relative Größenunterschied zwischen erweiterten und „klassischen“ regulären Ausdrücken durch keine rekursive Funktion beschränkt ist. Außerdem wird die Unentscheidbarkeit verschiedener Probleme für erweiterte reguläre Ausdrücke bewiesen. Überraschenderweise gilt all dies sogar dann, wenn die Ausdrücke nur eine einzige Variable verwenden.

Eine ausführlichere Darstellung der Inhalte dieses Artikels findet sich in den Arbeiten [4, 5].

1. Einleitung

Reguläre Ausdrücke zählen zu den am weitesten verbreiteten Beschreibungsmechanismen und werden sowohl in der theoretischen, als auch in der praktischen Informatik auf verschiedenste Arten angewendet. Allerdings haben sich im Lauf der Jahrzehnte in Theorie und Anwendung zwei unterschiedliche Interpretationen dieses Konzepts entwickelt.

Während die Theorie weitestgehend der klassischen Definition folgt und reguläre Ausdrücke betrachtet, die exakt die Klasse der regulären Sprachen beschreiben, erlauben die meisten modernen Implementierungen von regulären Ausdrücken die Spezifikation von Wiederholungen mittels *Variablen* (auch *Rückreferenzen* genannt). Die daraus resultierenden *erweiterten regulären Ausdrücke* können durch diese Wiederholungen auch nichtreguläre Sprachen beschreiben.

Beispielsweise erzeugt der erweiterte reguläre Ausdruck $((\mathtt{a} \mid \mathtt{b})^*)\%x\,x$ die nichtreguläre Sprache $\{ww \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$. Hierbei kann der Teilausdruck $((\mathtt{a} \mid \mathtt{b})^*)\%x$ ein beliebiges Wort $w \in \{\mathtt{a},\mathtt{b}\}^*$ erzeugen, während gleichzeitig dieses Wort w der Variablen x als Wert zugewiesen wird. Weitere Vorkommen von x erzeugen wiederum exakt das gleiche Wort w.

Andererseits führt die Verwendung von Variablen nicht zwangsläufig zu Nichtregularität; beispielsweise erzeugen (für $n \geq 1$) Ausdrücke der Form

$$\alpha_n := (\underbrace{(\mathtt{a} \mid \mathtt{b}) \ldots (\mathtt{a} \mid \mathtt{b})}_{n \text{ mal } (\mathtt{a} \mid \mathtt{b})})\%x\,x$$

die endliche (und daher reguläre Sprache) aller Wörter $ww \in \{\mathtt{a}, \mathtt{b}\}^*$, die die Länge $2n$ haben. Hierbei fällt auf, dass *klassische reguläre Ausdrücke* (d. h. Ausdrücke, die keine Variablen enthalten) für diese Sprachen exponentiell länger sind als die Ausdrücke α_n.

Dieser Artikel befasst sich hauptsächlich mit den folgenden zwei Fragen: Erstens, können erweiterte reguläre Ausdrücke – in Hinsicht auf ihre Länge oder auf ihre Variablenzahl – effektiv minimiert werden? Und zweitens, um wie viel kompakter ist die Beschreibung von regulären Sprachen durch erweiterte reguläre Ausdrücke im Vergleich zu klassischen regulären Ausdrücken?

Da die in der Praxis verwendeten erweiterten regulären Ausdrücke oft nur ungenau definiert sind (meist nur im Sinne einer „Definition durch Implementierung“) ist es für eine theoretische Betrachtung dieser Modelle notwendig, eine theoretisch greifbarere Definition zu finden. Hierfür wurde eine Vielzahl unterschiedlicher Mechanismen vorgeschlagen, unter anderem Aho [1], Câmpeanu et al. [3] und Bordihn et al. [2]. Aus Platzgründen verwenden wir im vorliegenden Artikel nur eine informelle Definition durch Beispiele; die hier vorgestellte Syntax und die beabsichtigte Semantik stammen aus [1]. Dennoch lassen sich die vorgestellten Resultate auf alle dem Autor bekannten verwandten Modelle übertragen, insbesondere auf die aus [3] und [2].

2. Resultate

Wir bezeichnen die Menge aller regulären Ausdrücke in denen höchstens k Variablen vorkommen mit RegEx(k). Folglich ist RegEx(0) die Menge der klassischen regulären Ausdrücke. Hauptgegenstand der Untersuchung sind die folgenden Probleme für RegEx(k):

Allspracheneigenschaft: Gegeben $\alpha \in$ RegEx(k), ist $L(\alpha) = \Sigma^*$?

Koendlichkeit: Gegeben $\alpha \in$ RegEx(k), ist $\Sigma^* \setminus L(\alpha)$ endlich?

$RegEx(l)$**-ität:** Gegeben $\alpha \in$ RegEx(k), existiert ein $\beta \in$ RegEx(l) mit $L(\alpha) = L(\beta)$?

Für den Fall $l = 0$ sprechen wir anstelle von RegEx(0)-ität von *Regularität*.

Theorem 2.1 *Für* RegEx(1) *ist Allspracheneigenschaft nicht semientscheidbar; Koendlichkeit und Regularität sind weder semientscheidbar, noch kosemientscheidbar.*

Beweis. (Skizze) Im Beweis von Theorem 2.1 werden Probleme zum Definitionsbereich von Turingmaschinen auf die in drei Probleme für RegEx(1) reduziert. Dazu wird jede zu untersuchende Turingmaschine $\mathcal{M}$ zuerst in eine sogenannte *erweiterte Turingmaschine* $\mathcal{X}$ konvertiert. Diese erweiterteren Turingmaschinen verwenden das zweielementige Bandalphabet $\{0, 1\}$ und verfügen zusätzlich zu den üblichen Instruktionen (Halten und schreibende Kopfbewegungen) über die Instruktion CHECK$_\mathrm{R}$. Wird jene ausgeführt, so überprüft die Maschine die komplette (unendlich lange) rechte Seite des Bandes. Findet sich dort kein Vorkommen von 1 (also ausschließlich 0), so geht die Maschine in eine Folgezustand über. Ansonsten geht sie in eine (nichtakzeptierende) Endlosschleife.

Wir definieren nun eine Menge VALC$(\mathcal{X}) \subseteq \{0,\#\}^*$ von *gültigen Berechnungen* von $\mathcal{X}$ wie im Folgenden beschrieben. Eine *Konfiguration* $C = (q_i, t_L, a, t_R)$ von $\mathcal{X}$ wird durch folgende Bestandteile charakterisiert:

1. den momentanen Zustand q_i,
2. den Inhalt des Bandes an der Kopfposition a,
3. die linke Bandseite t_L,
4. die rechte Bandseite t_R.

Um C in einem Wort $e(C)$ zu kodieren, interpretieren wir die beiden Bandseiten t_L und t_R als Binärzahlen $b(t_L)$ und $b(t_R)$, wobei jeweils die Bandzelle, die dem Kopf am nächsten ist, dem Bit mit dem niedrigsten Wert entspricht. Durch diese Kodierung lassen sich alle Bandoperationen mittels der Grundrechenarten beschreiben.

Die Kodierung $e(C)$ der Konfiguration $C = (q_i, t_L, a, t_R)$ definieren wir als

$$e(C) := 00^{b(t_L)}\#00^{b(t_R)}\#00^{a}\#0^{i}.$$

Eine Folge $\mathcal{C} = C_1, \ldots, C_n$ von Konfigurationen kodieren wir als

$$e(\mathcal{C})) := \#\#e(C_1)\#\# \cdots \#\#e(C_n)\#\#.$$

Die Menge VALC$(\mathcal{X})$ definieren wir nun als die Menge aller $e(\mathcal{C})$ für die $\mathcal{C}$ eine Folge von Konfigurationen ist, so dass die erste bzw. letzte Konfiguration der Folge eine Startkonfiguration bzw. akzeptierende Konfiguration von $\mathcal{X}$ ist, und jeweils zwei aufeinanderfolgende Konfigurationen dem korrekten Ablauf von $\mathcal{X}$ entsprechen.

Schließlich definieren wir INVALC$(\mathcal{X}) := \{0,\#\}^* \setminus$ VALC$(\mathcal{X})$. Nun lässt sich ein Ausdruck $\alpha \in$ RegEx(1) konstruieren, so dass $L(\alpha) =$ INVALC$(\mathcal{X})$. Insbesondere ist

1. $L(\alpha) = \{0,\#\}^*$ genau dann, wenn $\mathcal{X}$ (und somit $\mathcal{M}$) keine Eingabe akzeptiert, und
2. $L(\alpha)$ ist regulär genau dann, wenn $L(\alpha)$ koendlich ist, was genau dann der Fall ist, wenn $\mathcal{X}$ (und somit $\mathcal{M}$) endlich viele Eingaben akzeptiert.

Anschaulich betrachtet beschreibt α alle möglichen Fehler, die Wörter in INVALC$(\mathcal{X})$ aufweisen können. Die meisten dieser Fehler lassen sich mittels klassischer regulärer Ausdrücke, also ohne die Verwendung von Variablen, modellieren. Variablen werden nur für die Beschreibung Fehler auf einer der beiden Bandseiten benötigt (wenn also beim Übergang der Wert $b(t_L)$ oder $b(t_R)$ der Folgekonfiguration zu groß oder zu klein ist).

Da der Inhalt einer Bandseite der Folgekonfiguration nur vom Inhalt der selben Seite und dem Inhalt der Kopfzelle abhängt, können Beschreibungen von Fehlern der linken Seite die Inhalte der rechten Seite ignorieren (und umgekehrt). Außerdem beginnen alle Beschreibungen von Bandfehlern mit dem gleichen Teilausdruck, nämlich $(0 \mid \#)^*\#(0^*)\%x$. Daher lässt sich dieser Ausdruck „ausmultiplizieren“, so dass in der Tat eine einzige Variable ausreicht.

Die Grade der Unentscheidbarkeit für die genannten Probleme für RegEx(1) folgen nun unmittelbar aus den Graden der Unentscheidbarkeit für die vergleichbaren Probleme für Turingmaschinen. □

Aus der Unentscheidbarkeit der Allspracheneigenschaft folgt unmittelbar, dass die Länge von Ausdrücken aus RegEx(1) nicht berechenbar minimiert werden kann. Die Unentscheidbarkeit der Regularität zeigt dies für die Minimierbarkeit der Variablenzahl. Da Regularität nicht einmal kosemientscheidbar ist, folgt mittels der Konstruktion von Hartmanis [6] (siehe auch Kutrib [7]) die Existenz nichtrekursiver Tradeoffs zwischen RegEx(1) und RegEx(0).

Als praktische Konsequenz lässt sich feststellen, dass selbst die Verwendung einer einzigen Variable zwar deutlich kompaktere Ausdrücke erlauben kann (was sich natürlich auch positiv auf die Geschwindigkeit Abgleichens von Wörtern mit Ausdrücken auswirkt), dass aber andererseits dieser Vorteil aufgrund der zu überwindenden Nichtberechenbarkeiten nicht generell nutzbar ist.

Literatur

[1] A. AHO, Algorithms for Finding Patterns in Strings. In: J. VAN LEEUWEN (ed.), *Handbook of Theoretical Computer Science*. A. chapter 5, Elsevier, Amsterdam, 1990, 255–300.

[2] H. BORDIHN, J. DASSOW, M. HOLZER, Extending Regular Expressions with Homomorphic Replacements. *RAIRO Theoretical Informatics and Applications* **44** (2010) 2, 229–255.

[3] C. CÂMPEANU, K. SALOMAA, S. YU, A formal study of practical regular expressions. *International Journal of Foundations of Computer Science* **14** (2003), 1007–1018.

[4] D. FREYDENBERGER, Extended Regular Expressions: Succinctness and Decidability. In: *Proc. 28th International Symposium on Theoretical Aspects of Computer Science, STACS 2011*. LIPIcs 9, 2011, 507–518.

[5] D. FREYDENBERGER, *Inclusion of Pattern Languages and Related Problems*. Ph.D. thesis, Fachbereich Informatik und Mathematik, Goethe-University Frankfurt am Main, 2011. Logos Verlag, Berlin.

[6] J. HARTMANIS, On Gödel speed-up and succinctness of language representations. *Theoretical Computer Science* **26** (1983) 3, 335–342.

[7] M. KUTRIB, The phenomenon of non-recursive trade-offs. *International Journal of Foundations of Computer Science* **16** (2005) 5, 957–973.

Unäre Operatoren in Regulären Ausdrücken

Stefan Gulan

Universität Trier, FB IV – Informatik
54286 Trier
gulan@uni-trier.de

Zusammenfassung

Zwei Normalformen für reguläre Ausdrücke sind die Stern-Normalform [1] und die darauf aufbauende starke Stern-Normalform [2]. Beide Normalformen basieren auf der Entfernung bestimmter Redundanzen aus einem Ausdruck. Der Ausdruck wird dabei möglicherweise verküzt, zumindest ist die Normalform eines regulären Ausdrucks nicht länger als der Ausdruck selbst. Beide Normalformen sind in Linearzeit konstruierbar, weswegen sie sich zur effizienten Verkürzung regulärer Ausdrücke anbieten. Die starke Stern-Normalform wird hier unter Erhalt der erwähnten Eigenschaften auf reguläre Ausdrücke mit dem Operator für positive Iteration erweitert.

1. Grundlegende Notation

Die Menge der regulären Ausdrücke über dem Alphabet Σ wird RE_Σ notiert und genügt nachstehender EBNF:

$$\mathrm{RE}_\Sigma ::= \mathrm{RE}_\Sigma \cdot \mathrm{RE}_\Sigma \mid \mathrm{RE}_\Sigma + \mathrm{RE}_\Sigma \mid \mathrm{RE}_\Sigma^* \mid \mathrm{RE}_\Sigma^\times \mid \mathrm{RE}_\Sigma^? \mid \Sigma.$$

Hier bezeichnet $r^\times$ die positive Iteration des Ausdrucks r. Eine Summe $s_1 + s_2 + \cdots + s_n$ wird auch als $\sum_{1\leq i\leq n} s_i$ notiert. Die von einem Ausdruck r beschriebene Sprache ist $L(r)$. Ausdrücke, die die gleiche Sprache beschreiben heissen *äquivalent* und wir schreiben $r_1 \equiv r_2$ falls r_1 und r_2 äquivalent sind. Das leere Wort über einem beliebigen Alphabet wird als λ notiert. Ein λ-*Ausdruck* ist ein Ausdruck r mit $\lambda \in L(r)$. Man beachte, dass die Sprachen $\emptyset$ und $\{\lambda\}$ nicht durch Ausdrücke in obiger Form beschreibbar sind. Jede andere reguläre Sprache ist beschreibbar.

2. Schwache Unäre Normalform

Zunächst wird ein Ersetzungssystem angegeben, welches λ-Ausdrücke von gewissen Redundanzen befreit. Dazu löschen und erzeugen die Ersetzungen Vorkommen der unären Operatoren $^?$, * und $^\times$.

Eine *Marke* gibt an, wie sich ein Ausdruck zur Menge der λ-Ausdrücke verhält oder ob der Operator $^\times$ an bestimmten Stellen im Ausdruck steht. Marken werden in Postfix-Notation an Teilausdrücke gesetzt. Es wird zwischen *Aufwärts*- und *Abwärtsmarken* unterschieden. Aufwärtsmarken sind $\upharpoonright_\lambda$, $\upharpoonright_{\not\lambda}$, $\upharpoonright_?$ und $\upharpoonright_\times$, Abwärtsmarke ist $\downharpoonleft_\lambda$. Die intuitive Bedeutung der Marken ist wie folgt:

- $r\upharpoonleft_\lambda$ bedeutet, dass r ein λ-Ausdruck ist
- $r\upharpoonleft_{\not\lambda}$ bedeutet, dass r kein λ-Ausdruck ist
- $r\upharpoonleft_?$ bedeutet, dass r kein λ-Ausdruck ist, jedoch aus einem λ-Ausdruck r' durch Löschen von Vorkommen des Operators $^?$ hervorgeht.
- $r\upharpoonleft_\times$ bedeutet, dass r kein λ-Ausdruck ist und eine positive Iteration enthält.
- $r\downharpoonleft_\lambda$ bedeutet, dass bestimmte Vorkommen von $^\times$ in r durch * ersetzt und bestimmte Vorkommen von $^?$ gelöscht werden dürfen.

Marken „wandern“ vermöge der Ersetzungen durch einen Ausdruck bzw. dessen Syntaxbaum. Eine Aufwärtsmarke wandert dabei zur Wurzel, während eine Abwärtsmarke zu den Blättern wandert. Das Ersetzungssystem enthält für jeden Operator und jede Kombination von Marken an dessen Operanden eine Regel. Im folgenden ist für symmetrische Operandenpaare bei binären Operatoren jeweils nur ein Fall aufgeführt.

- Produkt

$$\begin{array}{cccc}
r\upharpoonleft_{\not\lambda} s\upharpoonleft_{\not\lambda} \mapsto (rs)\upharpoonleft_{\not\lambda} & r\upharpoonleft_{\not\lambda} s\upharpoonleft_\lambda \mapsto (rs)\upharpoonleft_{\not\lambda} & r\upharpoonleft_{\not\lambda} s\upharpoonleft_? \mapsto (rs^?)\upharpoonleft_{\not\lambda} & r\upharpoonleft_{\not\lambda} s\upharpoonleft_\times \mapsto (rs)\upharpoonleft_{\not\lambda} \\
 & r\upharpoonleft_\lambda s\upharpoonleft_\lambda \mapsto (rs)\upharpoonleft_\lambda & r\upharpoonleft_\lambda s\upharpoonleft_? \mapsto (rs^?)\upharpoonleft_\lambda & r\upharpoonleft_\lambda s\upharpoonleft_\times \mapsto (rs)\upharpoonleft_{\not\lambda} \\
 & & r\upharpoonleft_? s\upharpoonleft_? \mapsto (r^?s^?)\upharpoonleft_\lambda & r\upharpoonleft_? s\upharpoonleft_\times \mapsto (r^?s)\upharpoonleft_{\not\lambda} \\
 & & & r\upharpoonleft_\times s\upharpoonleft_\times \mapsto (rs)\upharpoonleft_{\not\lambda}
\end{array}$$

- Summe

$$\begin{array}{cccc}
r\upharpoonleft_{\not\lambda} + s\upharpoonleft_{\not\lambda} \mapsto (r+s)\upharpoonleft_{\not\lambda} & r\upharpoonleft_{\not\lambda} + s\upharpoonleft_\lambda \mapsto (r+s)\upharpoonleft_\lambda & r\upharpoonleft_{\not\lambda} + s\upharpoonleft_? \mapsto (r+s)\upharpoonleft_? & r\upharpoonleft_{\not\lambda} + s\upharpoonleft_\times \mapsto (r+s)\upharpoonleft_\times \\
 & r\upharpoonleft_\lambda + s\upharpoonleft_\lambda \mapsto (r+s)\upharpoonleft_\lambda & r\upharpoonleft_\lambda + s\upharpoonleft_? \mapsto (r+s)\upharpoonleft_? & r\upharpoonleft_\lambda + s\upharpoonleft_\times \mapsto (r+s\downharpoonleft_\lambda)\upharpoonleft_\lambda \\
 & & r\upharpoonleft_? + s\upharpoonleft_? \mapsto (r+s)\upharpoonleft_? & r\upharpoonleft_? + s\upharpoonleft_\times \mapsto (r+s\downharpoonleft_\lambda)\upharpoonleft_\lambda \\
 & & & r\upharpoonleft_\times + s\upharpoonleft_\times \mapsto (r+s)\upharpoonleft_\times
\end{array}$$

- Iteration

$$r\upharpoonleft_{\not\lambda}^* \mapsto r^*\upharpoonleft_\lambda \qquad r\upharpoonleft_\lambda^* \mapsto r^*\upharpoonleft_\lambda \qquad r\upharpoonleft_?^* \mapsto r^*\upharpoonleft_\lambda \qquad r\upharpoonleft_\times^? \mapsto r\downharpoonleft_\lambda\upharpoonleft_\lambda$$

- Positive Iteration

$$r\upharpoonleft_{\not\lambda}^\times \mapsto r^\times\upharpoonleft_\times \qquad r\upharpoonleft_\lambda^\times \mapsto r^*\upharpoonleft_\lambda \qquad r\upharpoonleft_?^\times \mapsto r^*\upharpoonleft_\lambda \qquad r\upharpoonleft_\times^\times \mapsto r^\times\upharpoonleft_\times$$

- Option

$$r\upharpoonleft_{\not\lambda}^? \mapsto r\upharpoonleft_? \qquad r\upharpoonleft_\lambda^? \mapsto r\upharpoonleft_\lambda \qquad r\upharpoonleft_?^? \mapsto r\upharpoonleft_? \qquad r\upharpoonleft_\times^? \mapsto r\downharpoonleft_\lambda$$

Die Ersetzungsregeln für die Abwärtsmarke $\downharpoonleft_\lambda$ sind $a\downharpoonleft_\lambda \mapsto a$ für $a \in \Sigma$, sowie

$$(rs)\downharpoonleft_\lambda \mapsto rs, \quad (r+s)\downharpoonleft_\lambda \mapsto r\downharpoonleft_\lambda + s\downharpoonleft_\lambda, \quad r^*\downharpoonleft_\lambda \mapsto r^* \quad r^\times\downharpoonleft_\lambda \mapsto r^* \quad \text{und} \quad r^?\downharpoonleft_\lambda \mapsto r.$$

Zu einem Ausdruck r wird der *literalmarkierte Ausdruck* $r\upharpoonleft$ gebildet, indem jedes Alphabetsymbol mit $\upharpoonleft_{\not\lambda}$ markiert wird. Das angegebene Ersetzungssystem terminiert auf markierten Ausdrücken in eindeutigen Normalformen. Eine Normalform besteht in einem regulären Ausdruck mit einer exakt einer Marke, die den gesamten Ausdruck markiert. Lässt man diese Marke weg, ergibt sich die *schwache unäre Normalform* von r, die $r^\div$ geschrieben wird.

Lemma 2.1 *Für jeden Ausdruck r gilt $r^\div \equiv r$ und $r^{\div\div} = r^\div$.*

Beispiel 2.2 *Es sei $a,b \in \Sigma$. Jeder der Ausdrücke $a+b^{?\times}$, $(a+b^{\times})^{?}$ und $a^{?}+b^{\times}$ hat die schwache unäre Normalform $a+b^{*}$.*

Ein Ausdruck ist *in* schwacher unärer Normalform, falls $r = r^{\div}$ gilt. Im Folgenden ist eine äquivalente Charakterisierung von Ausdrücken in Normalform gegeben. Diese zeigt die Redundanzen auf, die bei Überführung eines Ausdrucks in Normalform entfernt werden.

Satz 2.3 *Der Ausdruck r ist in schwacher unärer Normalform, $r = r^{\div}$, genau dann wenn r den folgenden Bedingungen genügt:*

1. *Ist $s^{?}$ oder $s^{\times}$ ein Teilausdruck von r, so gilt $\lambda \notin L(s)$*
2. *Ist $s^{?}$ oder s^{*} ein Teilausdruck von r mit $s = \sum_{1\le i\le n} s_i$, so gilt $s_i \neq t^{\times}$ für $1 \le i \le n$*
3. *Ist $s = \sum_{1\le i\le n} s_i$ ein Teilausdruck von r, so gilt*
   (a) *$s_j = t^{?}$ impliziert $\lambda \notin L(s_k)$ und $s_k \neq u^{\times}$ für $1 \le k \le n$ und $k \neq j$*
   (b) *$s_j = t^{\times}$ impliziert $\lambda \notin L(s)$*

3. Unäre Normalform

Definition 3.1 *Die Operatoren $^{\circ}$, $^{\diamond}$ und $^{\bullet}$ sind auf* RE_{Σ} *wie folgt definiert:*

- $[\cdot]^{\circ}$: $a^{\circ} := a$, $r^{?\circ} = r^{\circ}$, $r^{\times\circ} := r^{\circ}$, $r^{*\circ} := r^{\circ}$, $(r+s)^{\circ} := r^{\circ}+s^{\circ}$, *und*

$$(rs)^{\circ} = \begin{cases} r^{\circ}+s^{\circ}, & \textit{falls } \lambda \in L(rs) \\ rs, & \textit{sonst.} \end{cases}$$

- $[\cdot]^{\diamond}$: $a^{\diamond} := a$, $(rs)^{\diamond} := rs$, $(r+s)^{\diamond} := r^{\diamond}+s^{\diamond}$, $r^{?\diamond} = r^{?}$, $r^{\times\diamond} := r^{\diamond}$, $r^{*\diamond} := r^{*}$
- $[\cdot]^{\bullet}$: $a^{\bullet} := a$, $(rs)^{\bullet} := r^{\bullet}s^{\bullet}$, $(r+s)^{\bullet} := r^{\bullet}+s^{\bullet}$, $r^{?\bullet} = r^{\bullet ?}$, $r^{\times\bullet} := r^{\circ\bullet\times}$, $r^{*\bullet} := r^{\circ\bullet *}$

Die Iterations-Normalform *von r ist $r^{\bullet}$. Die* unäre Normalform *von r ist $r^{\circledast} := r^{\div\bullet}$*

Beispiel 3.2

1. *Der Ausdruck $r_1 = (a^{\times}+b)^{\times}$ is in schwacher unärer Normalform. Die Iterations-Normalform und die unäre Normalform von r_1 sind $r_1{}^{\bullet} = (a+b)^{\times}$ sowie $r_1{}^{\circledast} = (a+b)^{\times}$.*
2. *Der Ausdruck $r_2 = (a+b^{*})^{\times}$ ist in Iterations-Normalform. Die schwache unäre Normalform und die unäre Normalform von r_2 sind $r_2{}^{\div} = (a+b^{*})^{*}$ sowie $r_2{}^{\circledast} = (a+b)^{*}$.*

Lemma 3.3 *Für jeden Ausdruck r gilt $r^{\circledast} \equiv r$ und $r^{\circledast\circledast} = r^{\circledast}$.*

Satz 3.4 *Der Ausdruck r ist in unärer Normalform, $r = r^{\circledast}$, genau dann wenn r die folgenden Bedingungen erfüllt:*

1. *Ist $s^{?}$, $s^{\times}$ oder s^{*} ein Teilausdruck von r, so folgt $\lambda \notin L(s)$*

2. *Ist* $s^?$, $s^\times$ *oder* s^* *ein Teilausdruck von* r *mit* $s = \sum_{1 \leq i \leq n} s_i$, *so gilt* $s_i \neq t^\times$ *für* $1 \leq i \leq n$
3. *Ist* $s = \sum_{1 \leq i \leq n} s_i$ *ein Teilausdruck von* r, *so gilt*
   (a) $s_j = t^?$ *impliziert* $\lambda \notin L(s_k)$ *und* $s_k \neq u^\times$ *für* $1 \leq k \leq n$ *und* $k \neq j$
   (b) $s_j = t^\times$ *impliziert* $\lambda \notin L(s)$

Die *alphabetische Weite* eines Ausdrucks $r \in \mathrm{RE}_\Sigma$ ist die Anzahl an Zeichenvorkommen aus Σ in r, sie wird $|r|_\Sigma$ geschrieben. Die Anzahl an Vorkommen der Operatoren $^?$, * und $^\times$ in r wird *unäre Weite* von r genannt und als $|r|_\omega$ notiert.

Lemma 3.1 *Für jeden Ausdruck* r *gilt* $|r^{\circledast}|_\Sigma = |r|_\Sigma$, $|r^{\circledast}|_\omega \leq |r|_\omega$ *und* $|r^{\circledast}|_\omega \leq |r^{\circledast}|_\Sigma$.

Literatur

[1] A. BRÜGGEMANN-KLEIN, Regular expressions into finite automata. *Theoretical Computer Science* **120** (1993), 197–312.

[2] H. GRUBER, S. GULAN, Simplifying Regular Expressions. A Quantitative Perspective. In: *LATA 2010*. Number 6031 in LNCS, Springer, 2010, 285–296.

Beschreibungskomplexität kontextfreier Sprachen bezüglich der AFL-Operationen

Ronny Harbich

Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
harbich@iws.cs.uni-magdeburg.de

Zusammenfassung

Die Beschreibungskomplexität Prod(L) einer kontextfreien Sprache L ist die minimale Anzahl von Regeln, die eine kontextfreie Grammatik benötigt, um L zu erzeugen. Für eine Operation τ auf kontextfreie Sprachen werden kontextfreie Sprachen $L_1, \ldots, L_n$ beliebiger Beschreibungskomplexität so gesucht, dass Prod$(\tau(L_1, \ldots, L_n))$ eine beliebige natürliche Zahl annimmt. Als τ wurden die AFL-Operationen *Vereinigung, Konkatenation, Kleene-Abschluss, Homomorphismus, inverser Homomorphismus* und *Schnitt mit regulären Sprachen* gewählt.

1. Einleitung

Eine kontextfreie Grammatik G erzeugt eine kontextfreie Sprache L. Die Grammatik G besteht aus Regeln. Deren Anzahl, die Beschreibungskomplexität, drücken wir durch Prod(G) aus. Die Beschreibungskomplexität Prod(L) einer kontextfreien Sprache L ist Prod(G'), wenn G' eine kontextfreie Grammatik mit minimaler Regelanzahl ist, die L erzeugt. Wenden wir eine Operation, wie beispielsweise Vereinigung, auf zwei kontextfreie Sprachen L_1 und L_2 an, möchte man wissen, wie groß die Beschreibungskomplexität von $L_1 \cup L_2$, also Prod$(L_1 \cup L_2)$, gilt. Wir gehen hier allerdings der Frage nach, ob für gegebene natürliche Zahlen n_1, n_2 und k kontextfreie Sprachen L_1 und L_2 mit Prod$(L_1) = n_1$ und Prod$(L_2) = n_2$ so existieren, dass Prod$(L_1 \cup L_2) = k$ ist. Wir wollen also wissen, ob es Sprachen beliebiger Beschreibungskomplexität gibt, die nach der Vereinigung eine beliebige Beschreibungskomplexität annehmen. Sinnvoll ist derlei Untersuchung, wenn wir uns vorstellen, dass kontextfreie Grammatiken abgespeichert werden und Speicherplatz eine wesentliche Rolle spielt. So wäre es sinnvoll, die Grammatiken G_1 und G_2 für die Sprachen L_1 bzw. L_2 zu speichern anstatt $G_\cup$ für $L_1 \cup L_2$, falls Prod(G_1) + Prod(G_2) < Prod$(G_\cup)$ (umgekehrt gilt das gleiche).

Dassow und Stiebe sind in [1] der gleichen Problemstellung nachgegangen, nur dass hier nicht die Anzahl der Regeln, sondern die Anzahl der Nichtterminal als Komplexitätsmaß verwendet wurde. Weiterhin haben sie neben der Vereinigung die übrigen AFL[1]-Operationen Konkatenation, Kleene-Abschluss, Homomorphismus, inverser Homomorphismus und Schnitt mit regulären Sprache betrachtet. Ursprünglich geht die Problemstellung auf bezüglich der Anzahl der Zustände minimale endliche Automaten, die reguläre Sprachen akzeptieren, zurück. Wir verweisen auf die Arbeiten [3] und [4].

[1]Abstract Family of Languages

Für eine *kontextfreie Grammatik* $G = (N,T,P,S)$ (N ist das Nichtterminal- und T das Terminal-Alphabet, P ist eine endliche Menge von Regeln der Form $A \to w$ für $A \in N$ und $w \in (N \cup T)^*$, $S \in N$) bezeichnen wir die Anzahl der Regeln von G mit $\mathrm{Prod}(G) = |P|$ und die Anzahl der Symbole von G mit $\mathrm{Symb}(G) = \sum_{A \to w \in P} |w| + 2$ ($|w|$ ist Anzahl der Zeichen in w). Die *Beschreibungskomplexität* einer kontextfreien Sprache L ist

$$\mathrm{K}(L) = \min\{\mathrm{K}(G) \,|\, G \text{ ist kontextfreie Grammatik und } \mathrm{L}(G) = L\}$$

für $\mathrm{K} \in \{\mathrm{Prod}, \mathrm{Symb}\}$ ($\mathrm{L}(G)$ ist die *Sprache von* G). In dieser Arbeit untersuchen wir die folgende Fragestellung: Es seien eine n-stellige AFL-Operation τ für $n \in \mathbb{N}$ und $m_1, \ldots, m_n, k \in \mathbb{N}_0$ gegeben. Gibt es kontextfreie Sprachen L_i, $\mathrm{K}(L_i) = m_i$ für $1 \leq i \leq n$ so, dass $\tau(L_1, \ldots, L_n) = k$ gilt?

2. Anzahl der Regeln unter Vereinigung

Wir geben zunächst einige Sprachen und Sätze an, die wir später zur Beantwortung der Fragestellung benötigen.

Das nachstehende Lemma lässt sich basierend auf Satz 6.3 in [2] beweisen.

Lemma 2.1 *Es seien T ein Alphabet, $a \in T$ und $L_{n,m} = \left\{a^{2^i} \,\middle|\, n \leq i \leq m\right\}$ für $n,m \in \mathbb{N}_0$, $n \leq m$. Dann gilt* $\mathrm{Prod}(L_{n,m}) = m - n + 1$.

Lemma 2.2 *Es seien T ein Alphabet, $a \in T$ und $L_{p,q} = \left\{a^{iq} \,\middle|\, i \geq p\right\}$ für $p \in \mathbb{N}_0$ sowie $q \in \mathbb{N}$. Dann gilt* $\mathrm{Prod}(L_{p,q}) = 2$*. Außerdem existiert für jede kontextfreie Grammatik $G_{p,q}$ mit* $\mathrm{L}(G_{p,q}) = L_{p,q}$ *und* $\mathrm{Prod}(G_{p,q}) = \mathrm{Prod}(L_{p,q})$ *die Ableitung $S \Longrightarrow^* xSy$ für gewisse $x, y \in \{a\}^*$, $xy \neq \lambda$, wobei S das einzige Nichtterminal in $G_{p,q}$ ist.*

Beweis. Für die kontextfreie Grammatik

$$G_{p,q} = (\{S\}, \{a\}, P_{p,q}, S) \text{ mit } P_{p,q} = \{S \to a^q S, S \to a^{pq}\}$$

gilt $\mathrm{L}(G_{p,q}) = L_{p,q}$ und folglich $\mathrm{Prod}(L_{p,q}) \leq \mathrm{Prod}(G_{p,q}) = 2$. Wie wir uns leicht klar machen, kann eine unendliche Sprache mit nicht weniger als 2 Regeln erzeugt werden. Daher gilt insgesamt $\mathrm{Prod}(L_{p,q}) = 2$.

Wenn es mehr als nur ein Nichtterminal S in einer kontextfreien Grammatik $G_{p,q}$ mit $\mathrm{L}(G_{p,q}) = L_{p,q}$ und $\mathrm{Prod}(G_{p,q}) = \mathrm{Prod}(L_{p,q})$ gäbe, gäbe es zwangsweise mehr als 2 Regeln. Da $|L_{p,q}| = |\mathbb{N}|$ gilt, folgt dann $S \Longrightarrow^* xSy$ für gewisse $x, y \in \{a\}^*$, $xy \neq \lambda$, für $G_{p,q}$. □

Der kommende Satz zeigt, dass sich die Vereinigung zweier Sprachen nicht einfacher Beschreiben lässt als die Summe der Beschreibung der einzelnen Sprachen, sofern die Sprachen keine gemeinsamen Buchstaben besitzen und das Leerwort λ nicht enthalten.

Satz 2.3 *Es seien L_1 und L_2 zwei kontextfreie Sprachen, für die*

$$\mathit{Alph}(L_1) \cap \mathit{Alph}(L_2) = \emptyset \textit{ und } \lambda \notin L_1, \lambda \notin L_2$$

gelten. Außerdem seien $i \in \{1,2\}$ sowie $G_i = (N_i, T_i, P_i, S_i)$ eine kontextfreie Grammatik mit $\mathrm{L}(G_i) = L_i$, $\mathrm{Prod}(G_i) = \mathrm{Prod}(L_i)$, *für die ohne Beschränkung der Allgemeinheit $N_1 \cap N_2 = \emptyset$*

und $S \notin N_1 \cup N_2$ gelten. Des Weiteren soll es für G_i im Falle von $|L_i| = |\mathbb{N}|$, keine kontextfreie Grammatik $G_\# = (N_\#, T_\#, P_\#, S_\#)$ mit $\mathrm{L}(G_\#) = \mathrm{L}(G_i)$, $\mathrm{Prod}(G_\#) = \mathrm{Prod}(G_i)$ so geben, dass $S_\# \Longrightarrow^*_{G_\#} x_\# S_\# y_\#$ für alle $x_\#, y_\# \in T_\#^*$, $x_\# y_\# \neq \lambda$, nicht existiert, wenn $S_i \Longrightarrow^*_{G_i} x_i S_i y_i$ für gewisse $x_i, y_i \in T_i^*$, $x_i y_i \neq \lambda$, existiert.

Unter diesen Voraussetzungen gibt es eine kontextfreie Grammatik G mit $\mathrm{L}(G) = L_1 \cup L_2$ und $\mathrm{Prod}(G) = \mathrm{Prod}(L_1 \cup L_2)$ sowie nachstehenden Eigenschaften: Für $x_i, y_i \in T_i^*$, $x_i y_i \neq \lambda$, $i \in \{1, 2\}$ gelten

- $\mathrm{Prod}(G) = \mathrm{Prod}(G_1) + \mathrm{Prod}(G_2)$, *falls $S_1 \Longrightarrow^*_{G_1} x_1 S_1 y_1$ und $S_2 \Longrightarrow^*_{G_2} x_2 S_2 y_2$ nicht gelten oder falls $\mathrm{L}(G_1) = \emptyset$ oder $\mathrm{L}(G_2) = \emptyset$ ist,*
- $\mathrm{Prod}(G) = \mathrm{Prod}(G_1) + \mathrm{Prod}(G_2) + 2$, *falls $S_1 \Longrightarrow^*_{G_1} x_1 S_1 y_1$ und $S_2 \Longrightarrow^*_{G_2} x_2 S_2 y_2$ gelten,*
- $\mathrm{Prod}(G) = \mathrm{Prod}(G_1) + \mathrm{Prod}(G_2) + 1$, *für $j \in \{1, 2\}$ und $k \in \{1, 2\} \setminus \{j\}$, falls $S_j \Longrightarrow^*_{G_j} x_j S_j y_j$ nicht gilt und $S_k \Longrightarrow^*_{G_k} x_k S_k y_k$ gilt sowie $\mathrm{L}(G_j) \neq \emptyset$ ist.*

Das folgende Lemma liefert konkrete kontextfreie Sprachen für die Vereinigung sowie deren Beschreibungskomplexitäten. Es deckt einen großen Teil der natürlichen Zahlen ab.

Lemma 2.4 *Für $n \geq 7$, $4 \leq m \leq n$ und $6 \leq k \leq n$ sowie*

$$n' = n - 3, m' = m - 3, k' = k - 6,$$
$$L_{n,k} = \left\{ a^{2^i} \,\middle|\, 1 \leq i \leq k' \right\} \cup \left\{ b^{2^i} \,\middle|\, k' + 1 \leq i \leq n' \right\} \cup \{c\}^+,$$
$$L_m = \{b\}^+ \cup \left\{ c^{2^i} \,\middle|\, 1 \leq i \leq m' \right\}$$

gelten $\mathrm{Prod}(L_{n,k}) = n$, $\mathrm{Prod}(L_m) = m$ *und* $\mathrm{Prod}(L_{n,k} \cup L_m) = k$.

Beweis. Wir zerlegen die Sprache $L_{n,k} = A_k \cup B_{n,k} \cup C$, indem wir $A_k = \left\{ a^{2^i} \,\middle|\, 1 \leq i \leq k' \right\}$, $B_{n,k} = \left\{ b^{2^i} \,\middle|\, k' + 1 \leq i \leq n' \right\}$ und $C = \{c\}^+$ setzen. Nach Lemma 2.1 erhalten wir $\mathrm{Prod}(A_k) = k'$ und $\mathrm{Prod}(B_{n,k}) = n' - k'$ sowie $\mathrm{Prod}(C) = 2$ nach Lemma 2.2. Durch zweimalige Anwendung des Satzes 2.3 ergibt sich dann

$$\begin{aligned} \mathrm{Prod}(L_{n,k}) &= \mathrm{Prod}(A_k) + \mathrm{Prod}(B_{n,k}) + \mathrm{Prod}(C) + 1 \\ &= k' + n' - k' + 3 = k - 6 + n - 3 - (k - 6) + 3 = n. \end{aligned}$$

Analog errechnen wir für L_m die Regelanzahl $\mathrm{Prod}(L_m) = m$.

Die Vereinigung der Sprachen $L_{n,k}$ und L_m liefert

$$L_{n,k} \cup L_m = \left\{ a^{2^i} \,\middle|\, 1 \leq i \leq k' \right\} \cup \{b\}^+ \cup \{c\}^+$$

Durch Verwendung derselben Lemmata und Sätze wie im vorherigen Absatz ergibt sich

$$\begin{aligned} \mathrm{Prod}(L_{n,k} \cup L_m) &= \mathrm{Prod}\left(\left\{ a^{2^i} \,\middle|\, 1 \leq i \leq k' \right\} \right) + \mathrm{Prod}(\{b\}^+) + 1 + \mathrm{Prod}(\{c\}^+) + 1 \\ &= k' + 3 + 3 = k - 6 + 6 = k. \end{aligned}$$

□

Satz 2.5 *Die nachstehenden Tabellen zeigen an, ob zwei kontextfreie Sprachen* L_1 *und* L_2 *mit* $\mathrm{Prod}(L_1) = n_1$, $\mathrm{Prod}(L_2) = n_2$ *für* $n_1, n_2 \in \mathbb{N}_0$ *so existieren (Häkchen), dass* $\mathrm{Prod}(L_1 \cup L_2) = k$ *für* $k \in \mathbb{N}_0$ *gilt, oder nicht (Kreuz):*

n	m	k	Möglich
≥ 2	$2 \leq m \leq n$	$n+m+2$	✓
2	1	5	✗
≥ 2	$1 \leq m \leq n$	$n+m+1$	✓
1	1	$3 \leq k \leq 4$	✗
≥ 0	$0 \leq m \leq n$	$n \leq k \leq n+m$	✓

n	m	k	Möglich
≥ 7	$4 \leq m \leq n$	$6 \leq k \leq n$	✓
≥ 4	2	$4 \leq k \leq n$	✓
≥ 0	0	$\neq n$	✗
≥ 2	≥ 0	1	✗
≥ 1	≥ 0	0	✗

Für Für $n \geq 7$, $4 \leq m \leq n$ und $6 \leq k \leq n$ haben wir in Lemma 2.4 kontextfreie Sprachen angegeben. Die Tabelle ist nicht vollständig, da noch nicht für alle Fälle Sprachen gefunden werden konnten oder bewiesen werden konnte, dass es keine gibt.

3. Ergebnisse

Bezüglich der Anzahl der Regeln konnte die Beschreibungskomplexität für Homomorphismus, inverser Homomorphismus und Schnitt mit regulären Sprachen vollständig aufgedeckt werden. Im Falle der Vereinigung (siehe Satz 2.5) und des Kleene-Abschlusses ist dies noch nicht vollständig bekannt (wenige Fälle fehlen) und für die Konkatenation ist noch kein Ergebnis verfügbar.

Bezüglich der Anzahl der Symbole konnte die Beschreibungskomplexität für Homomorphismus und Schnitt mit regulärer Sprache vollständig ermittelt werden. Noch unvollständig sind hingegen Vereinigung und inverser Homomorphismus. Die Beschreibungskomplexität unter Konkatenation und Kleene-Abschluss sind zurzeit völlig offen.

Literatur

[1] J. DASSOW, R. STIEBE, Nonterminal Complexity of Some Operations on Context-Free Languages. *Fundamenta Informaticae* **83** (2008) 1-2, 35 – 49.

[2] J. GRUSKA, Some Classifications of Context-Free Languages. *Information and Control* **14** (1969), 152 – 179.

[3] S. YU, State Complexity of Regular Languages. *Journal of Automata, Languages and Combinatorics* **6** (2000), 221 – 234.

[4] S. YU, State complexity of finite and infinite regular languages. *Journal of Automata, Languages and Combinatorics* **76** (2002), 142 – 152.

Chop Operations and Expressions: Descriptional Complexity Considerations

Markus Holzer Sebastian Jakobi

Institut für Informatik, Justus-Liebig-Universität Giessen,
Arndtstr. 2, 35392 Giessen
{holzer,jakobi}@informatik.uni-giessen.de

Abstract

The chop or fusion operation was recently introduced in [S. A. BABU, P. K. PANDYA: Chop Expressions and Discrete Duration Calculus. *Modern Applications of Automata Theory,* World Scientific, 2010], where a characterization of regular languages in terms of chop expressions was shown. Simply speaking, the chop or fusion of two words is a concatenation were the touching letters are coalesced, if both letters are equal; otherwise the operation is undefined. We investigate the descriptional complexity of the chop operation and its iteration for deterministic and nondeterministic finite automata as well as for regular expressions. Moreover, we also consider the conversion problem between finite automata, regular expressions, and chop expressions.

1. Introduction

Recently, an alternative characterization of regular languages in terms of so called *chop expressions* (CEs) was introduced in [1]. These expressions are inspired by logic, to be more precise by the duration calculus, which is an interval logic for real time systems. Simply speaking a CE is nothing but a RE, where the operations of concatenation and Kleene star are replaced by the chop operation and its iterated version. The chop operation can be seen as a generalization of concatenation that allows one to verify whether the end of the first word overlaps by a single letter with the beginning of the second word. It is defined in the following way: For $u, v \in \Sigma^*$ let

$$u \odot v = \begin{cases} u'av' & \text{if } u = u'a \text{ and } v = av', \text{ for } u', v' \in \Sigma^* \text{ and } a \in \Sigma, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

which naturally can be extended to languages. Further, the language operations chop star $\otimes$ and chop plus $\oplus$ can be defined similarly to Kleene star and plus, where the 0th iteration of the chop operation is defined to be the set Σ in contrast to the 0th iteration of the ordinary concatenation.

We concentrate on two topics: (i) the operation problem for the chop operation and its iteration, and (ii) on conversion problems between automata, REs, and CEs. For the operation problem we obtain tight bounds in the exact number of states for DFAs and NFAs, which will be discussed in Section 2. Finally, Section 3. presents our results on the complexity of CEs.

This is an extended abstract of: M. Holzer, S. Jakobi. *Chop Operations and Expressions: Descriptional Complexity Considerations.* In G. Mauri, A. Leporati (eds.): Proceedings of the 15th International Conference on Developments in Language Theory, number 6795 in LNCS, pages 264–275, Milan, Italy, July 2011. Springer.

2. State Complexity of Chop Operations

We investigate the state complexity of the operations $\odot$, $\otimes$, and $\oplus$. Since $\odot$ is some sort of concatenation, the results in this section are very similar to those for concatenation and its iteration. Let us briefly recall the known results from the literature for comparison reasons. For DFAs with m and n states, respectively, tight bounds of $m \cdot 2^n - t \cdot 2^{n-1}$ for concatenation, where t is the number of accepting states of the "left" automaton, and $2^{n-1} + 2^{n-2}$ for Kleene star and Kleene plus in the exact number of states for DFAs were shown in [9]. For NFAs the situation is slightly different as proven in [7], where tight bounds of $m+n$ states for concatenation, $n+1$ for Kleene star, and n for Kleene plus can be found for NFAs. For the chop operation variants, we obtain a similar scenario since these operations will be cheap for NFAs but costly for DFAs. On the other hand, some of the tight bounds for the considered operations will depend on the size of the alphabet. Table 1 summarizes these results. More on the state complexity of chop operations, in particular on unary and finite languages, can be found in [6].

Operation	NFA	DFA
$\odot$	$m+n$	$m \cdot 2^n - t \cdot 2^{n-\min(\|\Sigma\|,n)} + 1$
$\cdot$	$m+n$	$m \cdot 2^n - t \cdot 2^{n-1}$
$\otimes$	$n+2$	$2^n - 1 + \min(\|\Sigma\|, n)$
$*$	$n+1$	$2^{n-1} + 2^{n-2}$
$\oplus$	$n+1$	2^n
$+$	n	$2^{n-1} + 2^{n-2}$

Table 1: Nondeterministic and deterministic state complexity of the chop operation and its iterations compared to the state complexity of concatenation and iteration. Here m is the number of states of the "left" automaton, and t is the number of its accepting states.

3. Complexity of Chop Expressions

We consider the descriptional complexity of conversions from CEs to finite automata or REs, and *vice versa*. Briefly recall what is known for REs and finite automata. The simplest way to transform REs into automata is Thompson's construction [8], which yields an NFA with a linear number of states, and simple examples show that this bound is asymptotically tight. For the backward conversion from finite automata to REs, most classical approaches are based on the same underlying algorithmic idea of *state elimination* [2], which produces REs of alphabetic width at most $2^{O(n)}$, where n is the number of states of the finite automaton over an alphabet polynomial in n. Recent research effort [4, 5] resulted in a tight lower bound of $2^{\Omega(n)}$ for this conversion, even for binary alphabets.

Now let us turn to the descriptional complexity of CEs. Here it it turns out that chop expressions are exponentially more succinct than regular expressions, while still having a linear descriptional complexity bound for the conversion to finite automata. Our results on upper and lower bounds for conversion problems for CEs are summarized in Table 2, where the polynomial upper bound for converting REs to CEs is from [1]. In contrast to this, we can show an exponential lower bound of $2^{\Omega(n^{1/2})}$ alphabetic width for the backwards conversion, i.e. from

Convert ... to ...	NFA	RE	CE
NFA	–	$2^{\Theta(n)}$, for $\lvert\Sigma\rvert \geq 2$	$2^{O(n)}$, for $\lvert\Sigma\rvert = n^{O(1)}$
RE	$\Theta(n)$	–	$\lvert\Sigma\rvert \cdot O(n^2)$
CE	$\Theta(n)$	$2^{\Omega(n^{1/2})}$, for $\lvert\Sigma\rvert = O(n)$ $2^{O(n)}$, for $\lvert\Sigma\rvert = n^{O(1)}$	–

Table 2: Upper and lower bounds for conversions between (non)deterministic finite automata (NFA), regular expressions (RE), and chop expressions (CE). For automata the size is the number of states, while for expressions the measure of alphabetic width is used (appropriately generalized to chop expressions). For comparison we have listed also the known conversions between NFAs and REs.

CEs to REs, if the size of the alphabet is linear compared to the length of the CE. This is done by using a result from [3] on the star height of a specific language and a result from [5] relating the star height of regular languages with their alphabetic width. On the other hand we can show that for a constant size alphabet, a single application of the chop star operation on REs results only in a polynomial increase of size of $2^{O(|\Sigma|)} \cdot n^4$ alphabetic width.

References

[1] S. A. BABU, P. K. PANDYA, Chop Expressions and Discrete Duration Calculus. In: D. D'SOUZA, P. SHANKAR (eds.), *Modern Applications of Automata Theory*. IISc research Monographs Series 2, World Scientific, 2010.

[2] J. A. BRZOZOWSKI, E. J. MCCLUSKEY, Signal flow graph techniques for sequential circuit state diagrams. *IEEE Trans. Comput.* **C-12** (1963) 2, 67–76.

[3] R. S. COHEN, Star Height of Certain Families of Regular Events. *Journal of Computer and System Sciences* **4** (1970) 3, 281–297.

[4] W. GELADE, F. NEVEN, Succinctness of Complement and Intersection of Regular Expressions. In: S. ALBERS, P. WEIL (eds.), *Proceedings of the* 25*th International Symposium on Theoretical Aspects of Compter Science*. LIPIcs 1, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Bordeaux, France, 2008, 325–336.

[5] H. GRUBER, M. HOLZER, Finite Automata, Digraph Connectivity, and Regular Expression Size. In: L. ACETO, I. DAMGAARD, L. A. GOLDBERG, M. M. HALLDÓRSSON, A. INGÓLFSDÓTTIR, I. WALKUWIEWICZ (eds.), *Proceedings of the* 35*th International Colloquium on Automata, Languages and Propgramming*. Number 5126 in LNCS, Springer, Reykjavik, Iceland, 2008, 39–50.

[6] M. HOLZER, S. JAKOBI, *State complexity of chop operations on unary and finite languages*. In preparation, 2011.

[7] M. HOLZER, M. KUTRIB, Nondeterministic Descriptional Complexity of Regular Languages. *Internat. J. Found. Comput. Sci.* **14** (2003) 6, 1087–1102.

[8] K. THOMPSON, Regular Expression Search Algorithm. *Com. ACM* **11** (1968) 6, 419–422.

[9] S. YU, State complexity of regular languages. *J. Autom., Lang. Comb.* **6** (2001), 221–234.

Nondeterministic State Complexity of Star-Free Languages

Markus Holzer Martin Kutrib Katja Meckel

Institut für Informatik, Justus-Liebig-Universität Giessen,
Arndtstr. 2, 35392 Giessen
{holzer,kutrib,meckel}@informatik.uni-giessen.de

The operation problem on a language family is the question of cost of operations on languages from this family with respect to their representations. More than a decade ago the operation problem for regular languages represented by deterministic finite automata (DFAs) as studied in [21, 22] renewed the interest in descriptional complexity issues of finite automata in general. It is well known that given some n-state NFA one can always construct a language equivalent DFA with at most 2^n states [19] and, therefore, NFAs can offer exponential savings in space compared with DFAs. In fact, later it was shown independently in [15, 17, 18] that this exponential upper bound is best possible. Recently, in [1] it was shown that this exponential tight bound for the determinization of NFAs also holds when restricting the NFAs to accept only subregular language families such as star languages [2], (two-sided) comet languages [3], star-free languages [16], prefix-closed languages, etc. On the other hand, there are also subregular language families known, where this exponential bound is not met. Prominent examples are the family of unary regular languages [7, 8] and the family of finite languages [20]. The significant different behavior with respect to the relative succinctness of NFAs compared to DFAs is also reflected in the operation problem for these devices. The operation problem for NFAs was first investigated in [11]. It turned out that in most cases when an operation is cheap for DFAs it is costly for NFAs and *vice versa*. All these results are for general regular languages. So, the question arises what happens to these bounds if the operation problem is restricted to subregular language families.

In fact, for some subregular language families this question was recently studied in the literature [4, 5, 6, 9, 10, 12, 13, 14] mostly for DFAs. An example for a subregular language family whose DFA operation problems meet the general bounds for most operations is the family of star-free languages [6], while prefix-, infix-, and suffix-closed languages [5], bifix-, factor-, and subword-free languages [4] show a diverse behavior mostly not reaching the general bounds. For a few language families, in particular prefix- and suffix-free regular languages, also the operation problem for NFAs was considered [9, 10, 12, 14], but for the exhaustively studied family of star-free languages it is still open. The family of star-free (or regular non-counting) languages is an important subfamily of the regular languages, which can be obtained from the elementary languages $\{a\}$, for $a \in \Sigma$, and the empty set $\emptyset$ by applying the Boolean operations

This is an extended abstract of: M. Holzer, M. Kutrib, K. Meckel. *Nondeterministic State Complexity of Star-Free Languages.* In B. Bouchou-Markhoff, P. Caron, J.-M. Champarnaud, D. Maurel (eds.): Proceedings of the 16th Conference on Implementation and Application of Automata, number 6807 in LNCS, pages 178–189, Blois, France, July 2011. Springer.

union, complementation, and concatenation finitely often. They obey nice characterizations in terms of aperiodic monoids and permutation-free DFAs [16].

Here we investigate their operation problem for NFAs with respect to the basic operations union, intersection, complementation, concatenation, Kleene star, and reversal. It turns out that for arbitrary star-free languages in most cases exactly the same tight bounds as in the general case are reached. This nicely complements the results recently obtained for the operation problem of star-free languages accepted by DFAs [6]. These results are summarized in Table 1 where we also list the results for DFAs accepting star-free languages [6], for comparison reasons.

	Star-free language accepted by ...	
Operation	DFA	NFA
$\cup$	mn	$m+n+1$
$\cap$	mn	mn
$\sim$		2^n
$\cdot$	$(m-1)2^n+2^{n-1}$	$m+n$
$*$	$2^{n-1}+2^{n-2}$	$n+1$
R	$2^n-1 \leq \cdot \leq 2^n$	$n+1$

Table 1: Deterministic and nondeterministic state complexities for the operation problem on star-free languages summarized. The results for DFAs are from [6].

For unary star-free languages the bound of the general case is met for the Kleene star and for reversal. The lower bound for concatenation is in the same order of magnitude as the upper bound, i.e., $\Theta(m+n)$. In case of the complementation of such a language the upper bound of $O(n^2)$ that can also be met in the order of magnitude. The lower bound for the union of two unary star-free languages misses the upper bound of the general case by a single state. The only still open case is for the intersection of two unary star-free languages. We can provide an upper bound of $\min\{\max\{xm-x-m, yn-y-n\}+1, mn\}$ where x denotes the length of the shortest word unequal to λ accepted by the m-state automaton and y the length of the one of the n-state automaton. Whether the upper bound can be met is still unknown. At the moment the best lower bound is given by $\min\{m,n\}$. We summarize our results in Table 2 and also list the results for DFAs accepting unary star-free languages [6], for comparison reasons.

	Star-free *unary* language accepted by ...	
Operation	DFA	NFA
$\cup$	$\max\{m,n\}$	$m+n \leq \cdot \leq m+n+1$
$\cap$	$\max\{m,n\}$	$\min\{m,n\} \leq \cdot \leq \min\{\max\{xm-x-m, yn-y-n\}+1, mn\}$
$\sim$		$\Theta(n^2)$
$\cdot$	$m+n-1$	$\Theta(m+n)$
$*$	$n^2-7n+13$	$n+1$
R	n	n

Table 2: Deterministic and nondeterministic state complexities for the operation problem on unary star-free languages summarized. The results for DFAs are from [6]. The numbers x and y denote the shortest words unequal to λ of the languages accepted by the m-state resp. n-state automaton.

References

[1] H. Bordihn, M. Holzer, M. Kutrib, Determinization of finite automata accepting subregular languages. *Theoret. Comput. Sci.* **410** (2009), 3209–3222.

[2] J. Brzozowski, Roots of star events. *J. ACM* **14** (1967), 466–477.

[3] J. Brzozowski, R. Cohen, On decompositions of regular events. *J. ACM* **16** (1969), 132–144.

[4] J. Brzozowski, G. Jirásková, B. Li, J. Smith, Quotient Complexity of bifix-, factor-, and subword-free languages. In: *International Conference on Automata and Formal Languages (AFL 2011)*. College of Nyíregyháza, 2011, 123–137.

[5] J. Brzozowski, G. Jirásková, C. Zou, Quotient complexity of closed languages. In: *Computer Science Symposium in Russia (CSR 2010)*. LNCS 6072, Springer, 2010, 84–95.

[6] J. Brzozowski, B. Liu, Quotient Complexity of star-free languages. In: *International Conference on Automata and Formal Languages (AFL 2011)*. College of Nyíregyháza, 2011, 138–152.

[7] M. Chrobak, Finite automata and unary languages. *Theoret. Comput. Sci.* **47** (1986), 149–158.

[8] M. Chrobak, Errata to "finite automata and unary languages". *Theoret. Comput. Sci.* **302** (2003), 497–498.

[9] Y. Han, K. Salomaa, Nondeterministic state complexity for suffix-free regular languages. In: *Descriptional Complexity of Formal Systems (DCFS 2010)*. EPTCS 31, 2010, 189–204.

[10] Y. Han, K. Salomaa, D. Wood, Nondeterministic state complexity of basic operations for prefix-free regular languages. *Fund. Inform.* **90** (2009), 93–106.

[11] M. Holzer, M. Kutrib, Nondeterministic descriptional complexity of regular languages. *Int. J. Found. Comput. Sci.* **14** (2003), 1087–1102.

[12] G. Jirásková, M. Krausová, Complexity in prefix-free regular languages. In: *Descriptional Complexity of Formal Systems (DCFS 2010)*. EPTCS 31, 2010, 197–204.

[13] G. Jirásková, T. Masopust, Complexity in union-free regular languages. In: *Developments in Language Theory (DLT 2010)*. LNCS 6224, Springer, 2010, 255–266.

[14] G. Jirásková, P. Olejár, State Complexity of intersection and union of suffix-free languages and descriptional complexity. In: *Non-Classical Models of Automata and Applications (NCMA 2009)*. books@ocg.at 256, Austrian Computer Society, 2009, 151–166.

[15] O. Lupanov, A comparison of two types of finite sources. *Problemy Kybernetiki* **9** (1963), 321–326. (in Russian), German translation: Über den Vergleich zweier Typen endlicher Quellen. Probleme der Kybernetik 6 (1966), 328-335.

[16] R. McNaughton, S. Papert, *Counter-Free Automata*. Number 65 in Research Monographs, MIT Press, 1971.

[17] A. R. Meyer, M. J. Fischer, Economy of description by automata, grammars, and formal systems. In: *Symposium on Switching and Automata Theory (SWAT 1971)*. IEEE, 1971, 188–191.

[18] F. Moore, On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Trans. Comput.* **20** (1971), 1211–1214.

[19] M. RABIN, D. SCOTT, Finite automata and their decision problems. *IMB J. Res. Dev.* **3** (1959), 114–125.

[20] K. SALOMAA, S. YU, NFA to DFA transformation for finite languages over arbitrary alphabets. *J. Autom., Lang. Comb* **2** (1997), 177–186.

[21] S. YU, Regular languages. In: *Handbook of Formal Languages*. 1, Springer, 1997, 41–110.

[22] S. YU, State complexity of regular languages. *J. Autom., Lang. Comb.* **6** (2001), 221–234.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 63 – 65

Restarting Automata and Rational Relations

Norbert Hundeshagen Friedrich Otto

Fachbereich Elektrotechnik/Informatik, Universität Kassel
34109 Kassel
{hundeshagen,otto}@theory.informatik.uni-kassel.de

Abstract

We study restarting automata with window size one that accept only regular languages. Moreover we describe two methods to associate *transductions* (that is, binary relations) with restarting automata of these types. We prove that the class of input-output relations of some restarting automata with window size one coincides with the class of *rational relations*. Then we define the *restarting automaton with output* and we focus on its computational power in comparison to the classical model.

1. Introduction

Automata with a restart operation were introduced originally to describe a method of grammar-checking for the Czech language (see, e.g., [5]). These automata started the investigation of restarting automata as a suitable tool for modeling the so-called *analysis by reduction*, which is a technique that is often used (implicitly) for developing formal descriptions of natural languages based on the notion of *dependency* [6, 9]. In particular, the Functional Generative Description (FGD) for the Czech language (see, e.g., [7]) is based on this method.

FGD is a dependency based system, which translates given sentences into their underlying tectogrammatical representations, which are (at least in principle) disambiguated. Thus, the real goal of performing analysis by reduction on (the enriched form of) an input sentence is not simply to accept or to reject this sentence, but to extract information from that sentence and to translate it into another form (be it in another natural language or a formal representation). Therefore, we are interested in *transductions* (that is, binary relations) and in ways to compute them by certain types of restarting automata.

Here we only study restricted types of restarting automata with window size one. First we identify those restarting automata that characterize the regular languages. Then following [4] we associate a binary relation with the *characteristic language* of a restarting automaton, motivated by the way in which the so-called *proper language* of a restarting automaton is defined. In this way we obtain a characterization for the class of *rational relations* (see, e.g., [1, 2]) in terms of monotone deterministic nonforgetting restarting automata with window size one.

The second part of our work concerns the relation between the automaton described above and restarting automata with output.

2. Restarting Automata with Window Size One

A large variety of types of restarting automata has been developed over the years. Here we are only interested in the RR-automaton with window size one. Such an automaton consists of a finite-state control, a single flexible tape with end markers, and a read/write window of size one. Formally, it is described by a 7-tuple $M = (Q, \Sigma, \text{¢}, \$, q_0, 1, \delta)$, where Q is a finite set of states, Σ is a finite input alphabet, the symbols $\text{¢}, \$ \notin \Sigma$ are used as markers for the left and right border of the work space, respectively, $q_0 \in Q$ is the initial state, 1 denotes the size of the *read/write window*, and δ is the *transition relation*.

Without looking at details (an overview can be found in [8]) the computation of these kinds of automata can be controlled by different modes of operation. Here we distinguish between monotone, deterministic and nonforgetting restarting automata. Further, restarting automata with window size one can be seen as an extension of finite-state acceptors, thus the class of regular languages is a subclass of the class of languages accepted by restarting automata. On the other hand we show that there are several types of deterministic, monotone, nonforgetting restarting automata that characterize the regular languages [3].

Obviously these types of automata are of special interest when dealing with rational relations. Therefore we want to study transductions that are computed by RR(1)-automata. Let $M = (Q, \Gamma, \text{¢}, \$, q_0, 1, \delta)$ be an RR(1)-automaton working on Γ, which contains the proper subalphabet Σ (input alphabet) and the *output alphabet* $\Delta = \Gamma \setminus \Sigma$. By Pr^{Σ} and Pr^{Δ} we denote the projections from Γ^* onto Σ^* and Δ^*. Then with M we associate the following *input/output relation*:

$$Rel_{\text{io}}(M) = \{\, (u,v) \in \Sigma^* \times \Delta^* \mid \exists w \in L(M) : u = \mathsf{Pr}^{\Sigma}(w) \text{ and } v = \mathsf{Pr}^{\Delta}(w) \,\}.$$

This definition leads to the characterization of the rational relations by several types of restarting automata.

3. Restarting Transducer

Instead of associating a relation R to a classical restarting automaton, we propose to compute a relation by a restarting automaton with output. To formalize this idea, we introduce the so-called *restarting transducer* (or RR(1)-Td, for short). A RR(1)-Td is an 8-tuple $T = (Q, \Sigma, \Delta, \text{¢}, \$, q_0, 1, \delta)$, where everything is defined as for the corresponding restarting automaton, but additionally Δ is a finite output alphabet, and δ is extended such that the transducer outputs some symbols over Δ in each restart or accept step.

With T we now associate a relation $Rel(T)$ that consists of all pairs of words (u,v), for which started on input $u \in \Sigma^*$, T accepts and produces the output $v \in \Delta^*$. We show that the relations computed by several types of restarting transducers with window size one are proper subclasses of the classes of input/output relations that are computed by the corresponding restarting automata. In this way we obtain characterizations for some subclasses of the rational relations.

References

[1] J. BERSTEL, *Transductions and Context-Free Languages*. Teubner Studienbücher, Teubner, Stuttgart, 1979.

[2] C. CHOFFRUT, K. C. II, Properties of finite and pushdown transducers. *SIAM Journal on Computing* **12** (1983) 2, 300–315.

[3] N. HUNDESHAGEN, F. OTTO, Characterizing the regular languages by nonforgetting restarting automata. In: G. MAURI, A. LEPORATI (eds.), *DLT 2011, Proc.*. Lecture Notes in Computer Science 6795, Springer, Berlin/Heidelberg, 2011, 288–299.

[4] N. HUNDESHAGEN, F. OTTO, M. VOLLWEILER, Transductions computed by PC-systems of monotone deterministic restarting automata. In: M. DOMARATZKI, K. SALOMAA (eds.), *CIAA 2010, Proc.*. Lecture Notes in Computer Science 6482, Springer, Berlin/Heidelberg, 2011, 163–172.

[5] V. KUBOŇ, M. PLÁTEK, A grammar based approach to a grammar checking of free word order languages. In: *COLING 94, Kyoto, Proc.*. II, 1994, 906–910.

[6] M. LOPATKOVÁ, M. PLÁTEK, V. KUBOŇ, Modeling syntax of free word-order languages: Dependency analysis by reduction. In: V. MATOUŠEK, P. MAUTNER, , T. PAVELKA (eds.), *Text, Speech and Dialogue*. Lecture Notes in Computer Science 3658, Springer, Berlin/Heidelberg, 2005, 748–748.

[7] M. LOPATKOVÁ, M. PLÁTEK, P. SGALL, Towards a formal model for Functional Generative Description: Analysis by reduction and restarting automata. *The Prague Bulletin of Mathematical Linguistics* **87** (2007), 7–26.

[8] F. OTTO, Restarting automata. In: Z. ESIK, C. MARTÍN-VIDE, V. MITRANA (eds.), *Recent Advances in Formal Languages and Applications*. Studies in Computational Intelligence 25, Springer, Berlin/Heidelberg, 2006, 269–303.

[9] P. SGALL, E. HAJIČOVÁ, J. PANEVOVÁ, *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects*. Reidel, Dordrecht, 1986.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 67 – 70

On the Complexity of Pushdown Transducers and Two-Way Transducers

Marian Kogler

Institute of Computer Science, Martin Luther University Halle-Wittenberg
Von-Seckendorff-Platz 1, 06120 Halle (Saale)
kogler@informatik.uni-halle.de

1. Introduction and Definitions

Calude, Salomaa and Roblot [1] introduced finite-state complexity, a variant of Kolmogorov complexity using a restricted definition of one-way finite-state transducers. In this paper, we consider two modifications of finite-state complexity, based on restricted definitions of pushdown transducers and two-way finite-state transducers, respectively.

In algorithmic information theory, the Kolmogorov complexity of a string is defined as the length of the shortest description of the string. Usually, this is accomplished by giving a string representation of a Turing machine and an input such that the Turing machine halts and outputs the expected string.

Finite-state complexity restricts the descriptions to finite-state machines. The finite-state complexity of a string is the length of the shortest description of the string using only a finite-state transducer and its input. Without loss of generality, we restrict ourselves to finite-state transducers where every state is a final state, i.e. $Q = Q_F$.

More formally, the finite-state complexity $C_S(x)$ of a string x is defined as the shortest string representing a transducer and an input string such that the transducer gives the output string x, i.e.

$$\min_{\sigma \in S, y \in X^*} \{|\sigma| + |y| : \sigma(y) = x\},$$

where S is a set of strings representing transducers and X is the alphabet. In the following, we will only use the two-letter alphabet $\{0,1\}$, unless otherwise specified.

This definition can easily be repurposed to allow not only (one-way) finite-state transducers, but also two-way finite-state transducers and pushdown transducers. In the following, we will use C_S for one-way finite-state complexity, C_S^2 for two-way finite-state complexity, and C_S^P for pushdown complexity, with regard to some set S containing string representation of transducers.

2. Encoding

A string representation of a transducer, or an encoding of a transducer into a string $\sigma \in \{0,1\}^*$ can be constructed in several ways. Throughout the paper, we will use a simple encoding, with states numbered and encoded by the binary representation of the number (the initial state is always assigned the number 0), symbols are retained verbatim, non-empty strings are prefixed with a 0, the empty word is mapped to 1, and, if applicable, a two-symbol code for the direction of movement. After every character, a 0 will be placed if the string continues, and a 1 if it ends, therefore forming a prefix-free code and ensuring no ambiguities.

The input state and symbol and pushdown input symbol (if applicable) are not represented in the encoding; rather, only the outputs are encoded, and ordered such that the first two outputs correspond to the outputs for state 0 and input 0 and 1, respectively; the next two outputs represent the transitions of the input state 1; and so on.

As an example, consider the (simplified; we assume that all states are also final states, and we restrict ourselves to a binary alphabet) two-way transducer $T = (Q, q_0, \delta)$ (with $\delta \subseteq Q \times \{0,1\}^* \times \{0,1\}^* \times Q \times \{L,R,S\}$) where

$$\begin{aligned} Q &= \{q_0, q_1\} \\ \delta &= \{(q_0, 0, e, q_0, R), (q_0, 1, 01, q_1, R), (q_1, 0, 10, q_1, L)\} \\ &\cup \{(q_1, 1, e, q_2, L), (q_2, 0, e, q_2, S), (q_2, 1, e, q_2, S)\} \end{aligned}$$

We construct the encoding as follows:

- q_0 is assigned the number 0, q_1 is assigned 1, q_2 is assigned 10.
- L is assigned 0, R is assigned 1, S is assigned 10.
- $(q_0, 0, e, q_0, R)$ is encoded as 11 | 01 | 11.
- $(q_0, 1, 01, q_1, R)$ is encoded as 000011 | 11 | 11.
- $(q_1, 0, 10, q_1, L)$ is encoded as 001001 | 11 | 01.
- $(q_1, 1, e, q_2, L)$ is encoded as 11 | 1011 | 01.
- $(q_2, 0, e, q_2, S)$ is encoded as 11 | 1011 | 1001.
- $(q_2, 1, e, q_2, S)$ is encoded as 11 | 1011 | 1001.

The transducer can therefore be unambiguously represented as the string 11 | 01 | 11 || 000011 | 11 | 11 || 001001 | 11 | 01 || 11 | 1011 | 01 || 11 | 1011 | 1001 || 11 | 1011 | 1001.

3. Complexity Results

Some results follow immediately; for instance, it is easy to see that given a reasonable encoding S, for all $x \in \{0,1\}^*$, $C_S^2 \le c \cdot C_S + d$ and $C_S^P \le c \cdot C_S + d$, i.e. the two-way finite state complexity and the pushdown complexity of a given string don't exceed the one-way finite-state complexity by more than a constant multiplicative factor.

On the other hand, it is also easy to see that there are strings x such that $C_S^2 = c \cdot C_S + d$ and $C_S^P = c \cdot C_S + d$, i.e. strings whose descriptional complexity can not be improved by using two-way finite-state transducers or pushdown transducers instead of one-way finite-state transducers.

We give a series of mappings to show there is no constant upper bound for the possible improvements; there are no constants c and d such that $C_S^2 \geq c \cdot C_S + d$ and $C_S^P \geq c \cdot C_S + d$ holds for all $x \in \{0,1\}^*$, but show the improvements are within polynomial bounds for both two-way finite transducers and pushdown transducers.

4. Simulation Results

We show that every two-way finite transducer can be simulated by a one-way finite transducer, with a polynomial input blow-up, allowing us to give an upper bound.

Since the number of reversals is bounded by the number of states and the number of letters in the alphabet, this can be achieved by repeating (and reversing) the input. Markers need to be inserted to allow the one-way finite transducer to find the correct location in the string; they can either be represented as additional symbols or be encoded. Additional states have to be added to encode the target and the current location.

If additional symbols are used, the number of symbols in the alphabet of the one-way finite-state transducer is bounded by the number of symbols in the alphabet of the two-way finite-state transducer, plus the length of the input and the length of the input increases by a constant multiplicative factor.

In the case of additional symbols not being used and the one-way finite-state transducer being confined to a two-letter alphabet, the length of the input of the two-way finite-state transducer is in $O(n \log n)$ of the input of the one-way finite-state transducer, since every symbol of the input needs to be preceded with a distinct binary number.

In both cases, the number of states of the one-way finite-state transducer is bounded linearly in the number of states in the two-way finite-state transducer and the length of the input (of the one-way finite-state transducer), since both the current symbol of the two-way finite-state transducer and the already processed parts of the number of the current symbol need to be stored in the state. This is necessary even in the case of a pushdown transducer simulating a two-way finite-state transducer.

5. Final Remarks

We conclude our paper with general remarks on the power of one-way finite transducers, two-way finite transducers and pushdown transducers by showing that while every mapping that can be performed by a one-way finite transducer can also be performed by a two-way finite transducer or a pushdown transducer, there are mappings which can only be performed by two-way finite transducers, but not by pushdown transducers, and vice versa.

Finally, we consider the state complexity of two-way finite transducers in comparison to pushdown transducers (the case of one-way finite transducers is trivial for state complexity of two-way finite-state and pushdown transducers, since two-way finite-state transducers require at least the same number of states, and pushdown transducers can simulate every one-way finite transducer with a constant number of states) and give upper and lower bounds with witnesses.

References

[1] C. Calude, K. Salomaa, T. Roblot, Finite-State Complexity and the Size of Transducers. In: I. McQuillan, G. Pighizzini (eds.), *DCFS*. EPTCS 31, 2010, 38–47.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 71 – 75

String Assembling Systems

Martin Kutrib Matthias Wendlandt

Institut für Informatik, Universität Gießen,
Arndtstr. 2, 35392 Gießen
{kutrib,matthias.wendlandt}@informatik.uni-giessen.de

Abstract

We introduce and investigate string assembling systems which form a computational model that generates strings from copies out of a finite set of assembly units. The underlying mechanism is based on piecewise assembly of a double-stranded sequence of symbols, where the upper and lower strand have to match. The generation is additionally controlled by the requirement that the first symbol of a unit has to be the same as the last symbol of the strand generated so far, as well as by the distinction of assembly units that may appear at the beginning, during, and at the end of the assembling process. We start to explore the generative capacity of string assembling systems. In particular, we prove that any such system can be simulated by some nondeterministic one-way two-head finite automaton, while the stateless version of the two-head finite automaton marks to some extent a lower bound for the generative capacity. Moreover, we obtain several incomparability and undecidability results and present questions for further investigations.

1. Introduction

The vast majority of computational models in connection with language theory processes or generates words, that is, strings of symbols out of a finite set. The possibilities to control the computation naturally depend on the devices in question. Over the years lots of interesting systems have been investigated. With the advent of investigations of devices and operations that are inspired by the study of biological processes, and the growing interest in nature-based problems modeled in formal systems, a very old control mechanism has been rekindled. If the raw material that is processed or generated by computational models is double stranded in such a way that corresponding symbols are uniquely related (have to be identical, for example), then the correctness of the complementation of a strand is naturally given. The famous Post's Correspondence Problem can be seen as a first study showing the power of double-stranded string generation. That is, a list of pairs of substrings $(u_1,v_1),(u_2,v_2),\dots,(u_k,v_k)$ is used to generate synchronously a double-stranded string, where the upper and lower string have to match. More precisely, a string is said to be generated if and only if there is a nonempty finite sequence of indices $i_1,i_2,\dots,i_k$ such that $u_{i_1}u_{i_2}\cdots u_{i_k} = v_{i_1}v_{i_2}\cdots v_{i_k}$. It is well-known that it is undecidable whether a PCP generates a string at all [6]. A more recent approach are sticker systems [1, 3, 5], where basically the pairs of substrings may be connected to form pieces that have to fit to the already generated part of the double strand. In addition, for variants the pieces may be added from left as well as from right. So, the generation process is subject to control mechanisms and restrictions given, for example, by the shape of the pieces.

Here we consider string assembling systems that are also double-stranded string generation systems. As for Post's Correspondence Problem the basic assembly units are pairs of substrings that have to be connected to the upper and lower string generated so far synchronously. The substrings are not connected as may be for sticker systems. However, we have two further control mechanisms. First, we require that the first symbol of a substring has to be the same as the last symbol of the strand to which it is connected. One can imagine that both symbols are glued together one at the top of the other and, thus, just one appears in the final string. Second, as for the notion of strictly locally testable languages [4][9] we distinguish between assembly units that may appear at the beginning, during, and at the end of the assembling process.

2. Definitions

As mentioned before, a string assembling system generates a double-stranded string by assembling units. Each unit consists of two substrings, the first one is connected to the upper and the second one to the lower strand. The corresponding symbols of the upper and lower strand have to be equal. Moreover, a unit can only be assembled when the first symbols of its substrings match the last symbols of their strands. In this case the matching symbols are glued together on at the top of the other. The generation has to begin with a unit from the set of initial units. Then it may continue with units from a different set. When a unit from a third set of ending units is applied the process necessarily stops. The generation is said to be valid if and only if both strands are identical when the process stops. More precisely:

Definition 2.1 *A* string assembling system (SAS) *is a quadruple* $\langle \Sigma, A, T, E\rangle$, *where* Σ *is the finite, nonempty set of* symbols *or* letters, $A \subset \Sigma^+ \times \Sigma^+$ *is the finite set of* axioms *of the forms* (uv, u) *or* (u, uv), *where* $u \in \Sigma^+$ *and* $v \in \Sigma^*$, $T \subset \Sigma^+ \times \Sigma^+$ *is the finite set of* assembly units, *and* $E \subset \Sigma^+ \times \Sigma^+$ *is the finite set of* ending assembly units *of the forms* (vu, u) *or* (u, vu), *where* $u \in \Sigma^+$ *and* $v \in \Sigma^*$.

The next definition formally says how the units are assembled.

Definition 2.2 *Let* $S = \langle \Sigma, A, T, E\rangle$ *be an SAS. The* derivation relation $\Rightarrow$ *is defined on specific subsets of* $\Sigma^+ \times \Sigma^+$ *by* $(uv, u) \Rightarrow (uvx, uy)$ *if*

i) $uv = ta$, $u = sb$, *and* $(ax, by) \in T \cup E$, *for* $a, b \in \Sigma$, $x, y, s, t \in \Sigma^*$, *and*

ii) $vx = yz$ *or* $vxz = y$, *for* $z \in \Sigma^*$.

$(u, uv) \Rightarrow (uy, uvx)$ *if*

i) $uv = ta$, $u = sb$, *and* $(by, ax) \in T \cup E$, *for* $a, b \in \Sigma$, $x, y, s, t \in \Sigma^*$, *and*

ii) $vx = yz$ *or* $vxz = y$, *for* $z \in \Sigma^*$.

A derivation is said to be *successful* if it initially starts with an axiom from A, continues with assembling units from T, and ends with assembling an ending unit from E. The process necessarily stops when an ending assembly unit is added. The sets A, T, and E are not necessarily disjoint.

The *language* $L(S)$ *generated* by S is defined to be the set

$$L(S) = \{\, w \in \Sigma^+ \mid (p,q) \Rightarrow^* (w,w) \text{ is a successful derivation} \},$$

where $\Rightarrow^*$ refers to the reflexive, transitive closure of the derivation relation $\Rightarrow$.

In order to clarify our notation we give a meaningful example.

Example 2.3 *The following SAS* $S = \langle \{a,b,c\}, A, T, E \rangle$ *generates the non-context-free language* $\{\, a^n b^n c^n \mid n \geq 1 \}$.

$$A = \{(a,a)\}, \quad T = T_a \cup T_b \cup T_c, \quad E = \{(c,c)\}, \text{ where}$$
$$T_a = \{(aa,a),(ab,a)\}, \quad T_b = \{(bb,aa),(bc,ab)\}, \quad T_c = \{(cc,bb),(c,bc),(c,cc)\}.$$

.

3. Generative Capacity

Theorem 3.1 *Let* $S = \langle \Sigma, A, T, E \rangle$ *be an SAS. There exists a nondeterministic one-way two-head finite automaton* M *that accepts* $L(S)$.

The previous theorem and its preceding discussion together with the proper inclusion of NL in NSPACE(n), which in turn is equal to the family of context-sensitive languages, reveals the following corollary.

Corollary 3.2 *The family of languages generated by SAS is properly included in* NL *and, thus, in the family of context-sensitive languages.*

Combining Theorem 3.1 and Example 2.3 we obtain the following relations to context-free languages.

Lemma 3.3 *The family of languages generated by SAS is incomparable with the family of (deterministic) (linear) context-free languages.*

Although the basic mechanisms of sticker systems and string assembling systems seem to be closely related, their generative capacities differ essentially. While the copy language $\{\, \$_1 w \$_2 w \$_3 \mid w \in \Sigma^+ \}$ is generated by an SAS, it is not generated by any sticker system. So, to some extent, SAS can copy while sticker systems cannot.

Conversely, some variant of the mirror language $\{\, w \mid w \in \{a,b\}^* \text{ and } w = w^R \}$ is generated by many variants of sticker systems (that can generate all linear context-free languages), but cannot be generated by any SAS, since it cannot be accepted by any nondeterministic one-way two-head finite automaton. So, to some extent, sticker systems can handle mirrored inputs while SAS cannot.

Following the discussion preceding Theorem 3.1, the simulation of SAS by nondeterministic one-way two-head finite automata gives only a rough upper bound for the generative capacity of SAS. Interestingly, the stateless version of the two-head finite automaton marks to some extent a lower bound for the generative capacity. More precisely, up to at most four additional symbols in the words generated, any stateless nondeterministic one-way two-head finite automaton can be simulated by some SAS. Actually, such a stateless automaton is a one-state device so that the transition function maps the two input symbols currently scanned to the head movements. Therefore, the automaton cannot accept by final state. It rejects if any of the heads falls off the tape or if the transition function is not defined for the current situation. If the transition function instructs both heads to stay, the automaton halts and accepts [2, 8].

Theorem 3.4 *Let $M = \langle \{s\}, \Sigma, \delta, \rhd, \lhd, s, \emptyset \rangle$ be a stateless nondeterministic one-way two-head finite automaton and \$, #, ?, ! $\notin \Sigma$. There exists a string assembling system S such that any word generated by S contains each of the symbols \$, #, ?, ! at most once, and $h(L(S)) = L(M)$, for the homomorphism $h(\$) = h(\#) = h(?) = h(!) = \lambda$ and $h(a) = a$, for $a \in \Sigma$.*

Stateless multi-head finite automata are studied in detail in [2, 8]. Though it is an open problem whether the additional symbols used in the simulation of the previous proof are necessary, there is a language generated by SAS which is not accepted by any stateless nondeterministic one-way two-head finite automaton.

Lemma 3.5 *The language $\{ a^{2n} \mid n \geq 1 \}$ is generated by an SAS but not accepted by any stateless nondeterministic one-way two-head finite automaton.*

4. Decidability Problems

It seems to be an obvious choice to proof the undecidability of several problems for SAS by reduction of Post's Correspondence Problem (PCP) (see, for example, [7]).

Let Σ be an alphabet and an instance of the PCP be given by two lists $\alpha = u_1, u_2, \ldots, u_n$ and $\beta = v_1, v_2, \ldots, v_n$ of words from Σ^+. It is well-known that it is undecidable whether a PCP has a solution [6], that is, whether there is a nonempty finite sequence of indices $i_1, i_2, \ldots, i_k$ such that $u_{i_1} u_{i_2} \cdots u_{i_k} = v_{i_1} v_{i_2} \cdots v_{i_k}$. In the sequel we call $i_1, i_2, \ldots, i_k$ as well as $u_{i_1} u_{i_2} \cdots u_{i_k}$ a solution of the PCP. We start to show that emptiness is undecidable from which further undecidability results follow.

Theorem 4.1 *Emptiness is undecidable for SAS.*

From the construction in the proof of Theorem 4.1 and the undecidability of emptiness we can derive several further undecidability results immediately.

Theorem 4.2 *Finiteness, infiniteness, equivalence, and inclusion are undecidable for SAS.*

Since SAS have been seen to generate even non-context-free languages, the questions whether regularity or context-freeness are decidable arises immediately.

Theorem 4.3 *Regularity and context-freeness are undecidable for SAS.*

References

[1] R. FREUND, G. PĂUN, G. ROZENBERG, A. SALOMAA, Bidirectional sticker systems. *Pacific Symposium on Biocomputing (PSB 1998)* (1998), 535–546.

[2] O. IBARRA, J. KARHUMÄKI, A. OKHOTIN, On stateless multihead automata: Hierarchies and the emptiness problem. *Theoret. Comput. Sci.* **411** (2009), 581–593.

[3] L. KARI, G. PĂUN, G. ROZENBERG, A. SALOMAA, S. YU, DNA computing, sticker systems, and universality. *Acta Inform* **35** (1998), 401–420.

[4] R. MCNAUGHTON, Algebraic decision procedures for local testability. *Math. Systems Theory* **8** (1974), 60–76.

[5] G. PĂUN, G. ROZENBERG, Sticker systems. *Theoret. Comput. Sci.* **204** (1998), 183–203.

[6] E. L. POST, A variant of a recursively unsolvable problem. *Bull. AMS* **52** (1946), 264–268.

[7] A. SALOMAA, *Formal Languages*. Academic Press, 1973.

[8] L. YANG, Z. D. ANDO. H. IBARRA, On stateless automata and P systems. *Workshop on Automata for Cellular and Molecular Computing* (2007), 144–157.

[9] Y. ZALCSTEIN, Locally testable languages. *J. Comput. System Sci.* **6** (1972), 151–167.

Descriptional Complexity of Two-Way Pushdown Automata With Restricted Head Reversals

Andreas Malcher$^{(A)}$ Carlo Mereghetti$^{(B)}$ Beatrice Palano$^{(B)}$

$^{(A)}$Institut für Informatik, Universität Gießen
Arndtstraße 2, 35392 Gießen, Germany
malcher@informatik.uni-giessen.de

$^{(B)}$Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano
via Comelico 39/41, 20135 Milano, Italy
{mereghetti,palano}@dsi.unimi.it

Abstract

Two-way nondeterministic pushdown automata (2PDA) are classical nondeterministic pushdown automata (PDA) enhanced with two-way motion of the input head. In this paper, the subclass of 2PDA accepting bounded languages and making at most a constant number of input head turns is studied with respect to descriptional complexity aspects. In particular, the effect of reducing the number of pushdown reversals to a constant number is of interest. It turns out that this reduction leads to an exponential blow-up in case of nondeterministic devices, and to a doubly-exponential blow-up in case of deterministic devices. If the restriction on boundedness of the languages considered and on the finiteness of the number of head turns is dropped, the resulting trade-offs are no longer bounded by recursive functions, and so-called non-recursive trade-offs are shown.

1. Introduction

Descriptional complexity is an area of theoretical computer science in which one of the main questions is how succinctly a formal language can be described by a formalism in comparison with other formalisms. A fundamental result is the exponential trade-off between nondeterministic and deterministic finite automata [11]. A further exponential trade-off is known to exist between unambiguous and deterministic finite automata, whereas the trade-offs between alternating and deterministic finite automata [8] as well as between deterministic pushdown automata (PDA) and deterministic finite automata [12] are bounded by doubly-exponential functions. Other doubly-exponential trade-offs exist between the complement of a regular expression and conventional regular expressions [4], and between constant height PDA and deterministic finite automata [1].

Summary of a paper presented at DCFS 2011, Limburg, Germany.

Apart from such trade-offs bounded by recursive functions, Meyer and Fischer [11] first showed the existence of trade-offs which cannot be bounded by any recursive function – so-called non-recursive trade-offs – between context-free grammars generating regular languages and finite automata. Nowadays, many non-recursive trade-offs are known, and surveys on recursive and non-recursive trade-offs may be found in [3, 5].

In this paper, we study the descriptional complexity of two-way pushdown automata (2PDA) which are conventional PDA with the possibility of moving the input head in both directions. 2PDA are a strong computational model: it is currently unknown whether their computational power equals that of linear bounded automata. Moreover, it is not known whether or not nondeterministic and deterministic variants describe the same language class. Thus, we consider here the subclass of those 2PDA where the number of reversals of the input head is bounded by some fixed constant. These head-turn bounded 2PDA have nice decidable properties when the languages accepted are letter-bounded (bounded, for short), i.e., they are subsets of $a_1^* a_2^* \cdots a_m^*$, where $a_1, a_2, \ldots, a_m$ are pairwise distinct symbols. In this case, the languages accepted are semilinear [6] and, due to decidability results on semilinear languages shown in [2], one obtains that the questions of emptiness, universality, inclusion, and equivalence are decidable. It is shown in [7] that these questions are decidable also for deterministic 2PDA where the number of turns of the pushdown store is bounded by some constant.

The results of the paper are as follows. Concerning recursive trade-offs for 2PDA accepting bounded languages, we first consider the nondeterministic case and compare head-turn bounded 2PDA (htb-2PDA) versus head-turn bounded and pushdown-turn bounded 2PDA ((htb,ptb)-2PDA). It turns out that the reduction of pushdown reversals leads to an exponential trade-off. This generalizes a similar result for one-way PDA: the trade-off between PDA and PDA with a fixed finite number of pushdown reversals is non-recursive when arbitrary languages are considered [9] whereas the trade-off becomes exponential in case of bounded languages [10]. As a second result, we convert head-turn bounded nondeterministic and deterministic 2PDA into equivalent deterministic devices which are additionally pushdown-turn bounded, obtaining a doubly-exponential trade-off. The main idea in the conversion is to reduce membership of bounded languages generated by context-free grammars to solving linear systems of Diophantine equations, which is shown to be manageable by deterministic 2PDA. In a second step, we generalize this result for the conversion of 2PDA with certain properties.

Finally, we consider arbitrary languages instead of bounded languages. We get non-recursive trade-offs between two-way and one-way devices. Furthermore, non-recursive trade-offs exist between 2PDA with an arbitrary and a constant number of head reversals both in the deterministic and nondeterministic case.

2. Results

Theorem 2.1 *Let $L \subseteq a_1^* a_2^* \cdots a_m^*$ be accepted by some htb-2PDA of size n. Then, L can also be accepted by some (htb,ptb)-2PDA of size $2^{O(n^2)}$.*

Theorem 2.2 *Let $L \subseteq a_1^* a_2^* \cdots a_m^*$ be accepted by some htb-2PDA of size n. Then, L can also be accepted by some (htb,ptb)-2DPDA whose size is bounded by a doubly-exponential function of $O(n^2)$.*

Considering unbounded languages instead of bounded languages, we obtain the following non-recursive trade-offs.

$$\text{(htb,ptb)-2DPDA} \overset{\textit{non-rec}}{\longrightarrow} \text{DPDA and (htb,ptb)-2PDA} \overset{\textit{non-rec}}{\longrightarrow} \text{PDA}.$$

It is currently unknown whether or not 2PDA and 2DPDA have the same computational power. Thus, it is not clear whether there is a recursive or non-recursive trade-off between 2PDA and 2DPDA.

If the restriction on the boundedness of the head turns is removed additionally, we obtain machines which may accept non-semilinear bounded languages and obtain the following non-recursive trade-offs.

$$\text{2DPDA} \overset{\textit{non-rec}}{\longrightarrow} \text{htb-2DPDA and 2PDA} \overset{\textit{non-rec}}{\longrightarrow} \text{htb-2PDA}.$$

References

[1] V. GEFFERT, C. MEREGHETTI, B. PALANO, More concise representation of regular languages by automata and regular expressions. *Information and Computation* **208** (2010), 385–394.

[2] S. GINSBURG, *The Mathematical Theory of Context-Free Languages*. McGraw-Hill, New York, 1966.

[3] J. GOLDSTINE, M. KAPPES, C. KINTALA, H. LEUNG, A. MALCHER, D. WOTSCHKE, Descriptional complexity of machines with limited resources. *Journal of Universal Computer Science* **8** (2002), 193–234.

[4] M. HOLZER, M. KUTRIB, The complexity of regular(-like) expressions. In: Y. GAO, H. LU, S. SEKI, S. YU (eds.), *Developments in Language Theory (DLT 2010, London, Canada*. LNCS 6224, Springer-Verlag, Berlin, 2010, 16–30.

[5] M. HOLZER, M. KUTRIB, Descriptional complexity—an introductory survey. In: C. MARTÍN-VIDE (ed.), *Scientific Applications of Language Methods*. Imperial College Press, 2010, 1–58.

[6] O. IBARRA, A note on semilinear sets and bounded-reversal multihead pushdown automata. *Information Processing Letters* **3** (1974), 25–28.

[7] O. IBARRA, T. JIANG, N. TRAN, H. WANG, New decidability results concerning two-way counter machines. *SIAM Journal on Computing* **24** (1995), 123–137.

[8] E. LEISS, Succinct representation of regular languages by Boolean automata. *Theoretical Computer Science* **13** (1981), 323–330.

[9] A. MALCHER, On recursive and non-recursive trade-offs between finite-turn pushdown automata. *Journal of Automata, Languages and Combinatorics* **12** (2007), 265–277.

[10] A. MALCHER, G. PIGHIZZINI, Descriptional complexity of bounded context-free languages. In: T. HARJU, J. KARHUMÄKI, A. LEPISTÖ (eds.), *Developments in Language Theory (DLT 2007)*. LNCS 4588, Springer-Verlag, Berlin, 2007, 312–323.
(See also `http://arxiv.org/abs/0905.1045` for an extended version).

[11] A. MEYER, M. FISCHER, Economy of descriptions by automata, grammars, and formal systems. In: *IEEE Symposium on Foundations of Computer Science*. 1971, 188–191.

[12] L. VALIANT, Regularity and related problems for deterministic pushdown automata. *Journal of the ACM* **22** (1975), 1–10.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 81 – 83

Algorithms and Pseudo-Periodicity in Words

Florin Manea$^{(B)}$ Robert Mercaş$^{(A)}$ Cătălin Tiseanu$^{(B)}$

$^{(B)}$Faculty of Mathematics and Computer Science, University of Bucharest
Str. Academiei 14, RO-010014 Bucharest, Romania,
flmanea@fmi.unibuc.ro, ctiseanu@gmail.com

$^{(A)}$Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
PSF 4120, D-39016 Magdeburg, Germany
robertmercas@gmail.com

Abstract

Periodicity is a long investigated property on words. One of the generalizations regarding this concept was introduced by L. Kari et al., and consideres a word to be θ-periodic, if it is the prefix of a word consisting of concatenations of itself and the application of a morphic (antimorphic) involution to it. Starting from this, the authors investigate the minimum length a word should have, such that, if it has two different θ-periods m and n, then it also has another θ-period t given by the root of the two words that give the initial periods. In this paper, we extend the notion of θ-periodicity to arbitrary functions, and investigate the problem both from the combinatorial and algorithmical point of view.

1. Introduction

Let V be a nonempty finite set of symbols called an *alphabet*, and every element $a \in V$ a *letter*. A *word* w over V is a (finite) sequence of letters from V, while the *length* of it is denoted by $|w|$ and represents the number of letters in the sequence. The *empty word* is the sequence of length zero and is denoted by ε.

A word u is a *factor* of a word v if $v = xuy$ for some x, y. We say that the word u is a *prefix* of v if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$. We denote by $w[i]$ the symbol at position i in w, and by $w[i..j]$ the factor of w starting at position i and ending at position j, consisting of the concatenation of the symbols $w[i], \ldots, w[j]$, where $1 \leq i \leq j \leq n$.

The *period* of a partial word w over V is a positive integer p such that $w[i] = w[j]$ whenever we have $i \equiv j \ mod\ p$. In such a case, we say w is p-$periodic.$

For a morphism $f : V^* \to V^*$, we say that w is f-periodic if it is a prefix of $t\{t, f(t)\}^+$, for some word t of length smaller than w. If $w \in t\{t, f(t), \ldots, f^{k+1}(t)\}^+$, for some word t and some minimal integer k such that f^{k+1} is the identity, we call w f^*-periodic.

The powers of a word w are defined recursively by $w^0 = \varepsilon$ and $w^n = ww^{n-1}$, for $n \geq 1$. We say that w is a f-power, if $w \in t\{t, f(t)\}^+$, for some word t.

For a complete view on basic combinatorial definitions we refer to [1, 2, 6], while definitions needed to follow the algorithms are found in [5]. We stress out that all time bounds provided in this paper hold on the *unit-cost RAM model*.

$^{(A)}$The work of Robert Mercaş is supported by the *Alexander von Humboldt Foundation.*

2. Results

These following two results generalize the case of the original Fine and Wilf theorem [3], as well as the case of involutions, presented by Kari et. al. [2, 4].

Theorem 2.1 *Let u and v be two words over an alphabet V and $f : V^* \to V^*$ a bijective morphism such that $f(a) \in V$, for all $a \in V$. Let $k \geq 0$ be the minimum integer such that f^{k+1} is the identity (clearly, $k \leq |V|$). If the words $u\{u, f(u), \ldots, f^k(u), v, f(v), \ldots, f^k(v)\}^*$ and, respectively, $v\{u, f(u), \ldots, f^k(u, v, f(v), \ldots, f^k(v)\}^*$ have a common prefix of length greater or equal to $|u| + |v| - \text{gcd}(|u|, |v|)$, then there exists $t \in V^*$ such that $u, v \in t\{t, f(t), \ldots, f^k(t)\}^*$. Moreover, the bound $|u| + |v| - \text{gcd}(|u|, |v|)$ is optimal.*

A similar results does not hold for antimorphisms, even if we allow the size of the common prefix to be arbitrarily large.

Example 2.2 *Let us consider the words $u = abc$ and $v = ab$, and the antimorphism f described by $f(a) = e$, $f(b) = d$, $f(c) = c$, $f(d) = b$, $f(e) = a$. It is easy to see that f is even more an involution. The infinit word $w = ab\,c\,(de)^\omega$ can be writen as $w = uf(v)^\omega = vf(u)f(v)^\omega$, where by x^ω we denote an infinite repetition of the word x. It is easy to see that all three words u, v and w are f-aperiodic.*

Theorem 2.3 *Let u and v be two words over an alphabet V and $f : V^* \to V^*$ a bijective antimorphism such that $f(a) \in V$, for all $a \in V$. Denote by $k \geq 0$ the minimum integer such that f^{k+1} is the identity. If $u\{u, f(u)\}^*$ and $v\{v, f(v)\}^*$ have a common prefix of length greater or equal to $2|u| + |v| - \text{gcd}(|u|, |v|)$, then there exists $t \in V^+$, such that $u, v \in t\{t, f(t), \ldots, f^k(t)\}^*$.*

Next, we give some algorithmical results concerning f-periodicity on words.

Problem 1 *Given an anti-/morphism $f : V^* \to V^*$ and a word w, determine if w is f- or f^*-periodic.*

Theorem 2.4 *Let f be an uniform anti-/morphism. We can decide whether w, a length n word, is f-periodic in time $\mathcal{O}(n \log\log n)$.*

Theorem 2.5 *Let f be an uniform anti-/morphism. We can decide whether w, a length n word, is f^*-periodic in time $\mathcal{O}(n \log\log n)$, where the constant depends only on the degree of f.*

Theorem 2.6 *Let f be an arbitrary increasing anti-/morphism. We can decide whether w, a length n word, is f-periodic in time $\mathcal{O}(n \log n)$.*

We continue by considering the following problem:

Problem 2 *Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists f an anti-/morphism on V such that w is f-periodic.*

First notice that one can always give a positive answer to this problem. Indeed, if $w = aw'$, we just take f as a anti-/morphism with $f(a) = w'$, and we obtain that $w = af(a)$. Next we consider several more difficult cases of this problem and assume first that f is a morphism.

If we consider the problem “Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists a morphism f on V, such that $w \in t\{t, f(t)\}^+$, where the length of t is upper bounded by a constant k”, we easily obtain that this problem is decidable in polynomial time, but the degree of the polynomial depends on the constant k.

Also, the problem “Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists an uniform morphism f on V, such that w is f-periodic” can be solved in polynomial time. If we restrict ourselves to finding a morphsim with $|f(a)| = k$ for all the letters $a \in V$, we obtain that the previously mentioned problem can be solved in $\mathcal{O}(n \log\log n)$ time, where the constant hidden by the $\mathcal{O}$ denotation depends on k.

Finally, the problem “Given $w \in V^*$ and $\ell \leq |w|$, with $|w| \geq 2$, decide whether there exists an arbitrary increasing morphism f on V, such that w is in $t\{t, f(t)\}^+$, with $2 \leq |t| \geq \ell$”, is **NP**-complete.

A quite similar discussion can be made for antimorphisms.

First, if we consider the problem “Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists an antimorphism f on V such that $w = tf(t)$”, we easily obtain that this problem is decidable in polynomial time.

Also, if we consider the problem “Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists an antimorphism f on V such that $w \in t\{t, f(t)\}^+$, where the length of t is upper bounded by a constant k”, we obtain again that this problem is decidable in polynomial time, but the degree of the polynomial depends on the constant k.

It is not surprising to see that the problem “Given $w \in V^*$, with $|w| \geq 2$, decide whether there exists an uniform antimorphism f on V such that w is f-periodic” can be solved in polynomial time. Moreover, when we restrict ourselves to finding an antimorphsim with $|f(a)| = k$ for all the letters $a \in V$, we are able to solve this problem in $\mathcal{O}(n \log\log n)$ time, where the constant hidden by the $\mathcal{O}$ denotation depends on k.

Finally, the problem “Given $w \in V^*$ and $\ell \leq |w|$, with $|w| \geq 2$, decide whether there exists an arbitrary increasing antimorphism f on V such that w is in $tf(t)\{t, f(t)\}^+ \cup t^2\{t, f(t)\}^+$, with $2 \leq |t| \geq \ell$”, is **NP**-complete.

References

[1] C. CHOFFRUT, J. KARHUMÄKI, Combinatorics of Words. In: G. ROZENBERG, A. SALOMAA (eds.), *Handbook of Formal Languages*. 1, Springer-Verlag, 1997, 329–438.

[2] E. CZEIZLER, L. KARI, S. SEKI, On a special class of primitive words. *Theoretical Computer Science* **411** (2010), 617–630.

[3] N. J. FINE, H. S. WILF, Uniqueness Theorem for Periodic Functions. *Proceedings of the American Mathematical Society* **16** (1965), 109–114.

[4] L. KARI, S. SEKI, An improved bound for an extension of Fine and Wilf’s theorem, and its optimality. *Fundamenta Informaticae* **191** (2010), 215–236.

[5] D. E. KNUTH, *The Art of Computer Programming*. 1: Fundamental Algorithms, Addison-Wesley, 1968.

[6] M. LOTHAIRE, *Combinatorics on Words*. Cambridge University Press, 1997.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 85 – 88

On Periodic Partial Words

Florin Manea[(A)] Robert Mercaş[(B)] Cătălin Tiseanu[(A)]

[(A)]Faculty of Mathematics and Computer Science, University of Bucharest,
Str. Academiei 14, RO-010014 Bucharest, Romania,
flmanea@fmi.unibuc.ro, ctiseanu@gmail.com

[(B)]Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik
PSF 4120, D-39016 Magdeburg, Germany
robertmercas@gmail.com

Abstract

In this paper we investigate several periodicity-related algorithms for partial words. First, we show that all periods of a partial word of length n are determined in $\mathcal{O}(n \log n)$, and provide algorithms and data structures that help us answer in constant time queries regarding the periodicity of their factors. For this we need a $\mathcal{O}(n^2)$ preprocessing time and a $\mathcal{O}(n)$ updating time, whenever the words are extended by adding a letter. In the second part we show that substituting letters of a word w with holes, with the property that no two holes are too close to each other, to make it periodic can be done in optimal time $\mathcal{O}(|w|)$. Moreover, we show that inserting the minimum number of holes such that the word keeps the property can be done as fast.

Keywords: Combinatorics on Words, Periodicity, Partial Words.

Let A be a nonempty finite set of symbols called an *alphabet*. Each element $a \in A$ is called a *letter*. A *full word* over A is a finite sequence of letters from A, while a *partial word* over A is a finite sequence of letters from $A_\diamond = A \cup \{\diamond\}$, the alphabet A with the hole symbol $\diamond$.

The *length* of a partial word u is denoted by $|u|$ and represents the total number of symbols in u. The *empty word* is the sequence of length zero and is denoted by ε. A partial word u is a *factor* of a partial word v if $v = xuy$ for some x, y. We say that u is a *prefix* of v if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$. We denote by $u[i]$ the symbol at position i in u, and by $u[i..j]$ the factor of u starting at position i and ending at position j, consisting of the concatenation of the symbols $u[i], \dots, u[j]$, where $1 \leq i \leq j \leq n$.

The powers of a partial word u are defined recursively by $u^0 = \varepsilon$ and for $n \geq 1$, $u^n = uu^{n-1}$. The *period* of a partial word u over A is a positive integer p such that $u[i] = u[j]$ whenever $u[i], u[j] \in A$ and $i \equiv j \ mod\ p$. In such a case, we say u is p-$periodic$. If u is p-periodic and $|u| > p$ we say that u is non-trivially periodic.

If u and v are two partial words of equal length, then u is said to be *contained* in v, denoted by $u \subset v$, if $u[i] = v[i]$, for all $u[i] \in A$. Partial words u and v are *compatible*, denoted by $u \uparrow v$, if there exists w such that $u \subset w$ and $v \subset w$.

A partial word u is said to be d-valid, for some positive integer d, if $u[i..i+d-1]$ contains at most one $\diamond$-symbol, for all $1 \leq i \leq |u| - d + 1$.

For a complete view on the basic definitions regarding combinatorics on words and partial words we refer the reader to [5, 9, 1]. The basic definitions needed to follow the algorithms presented here are found in [8]; we just stress out that all time bounds provided in this paper hold on the *unit-cost RAM model*.

Let us first consider the problem of computing all the periods of a given partial word, as well as the following problem.

Problem 1
1. *Given a partial word w, of length n, over an alphabet V, preprocess it in order to answer the following types of queries:*
Is $w[i..j]$ p-periodic?*, denoted* **per**(i,j,p)*, where* $i \leq j, i,j,p \in \{1,\ldots,n\}$.
Which is the minimum period of $w[i..j]$?*, denoted* **minper**(i,j)*, where* $i \leq j$*,* $i,j \in \{1,\ldots,n\}$.
2. *Consider the following update operation for a partial word w: add a symbol $a \in V \cup \{\diamond\}$ to the right end of w, to obtain wa. Preprocess w and define a method to update the data structures constructed during the preprocessing process, in order to answer in constant time* **per** *queries, for a word obtained after several update operations were applied to w.*

The results we obtain are the following:

Theorem 1 *Let w be a partial word of length n.*
1. All the periods of w can be computed in time $\mathcal{O}(n \log n)$.
2. The partial word w can be processed in time $\mathcal{O}(n^2)$ in order to answer in constant time **per** *and* **minper** *queries. After an update operation, the previously constructed data structures are updated in $\mathcal{O}(n)$ time, and both* **per** *and* **minper** *queries are answered in time $\mathcal{O}(1)$.*

This result is particularly useful in two applications: finding the minimal period of a partial word and deciding whether a word is primitive. We are aware of the claims and proofs that these problems can be solvable in linear time (see [3] and, respectively, [2]). However, the algorithms proposed in these papers are relying on the fact that one can find all factors of a partial word ww that are compatible with w in linear time, by extending to the case of partial words some string matching algorithms for full words, that work in linear time. The proof of this fact was not given formally, and we are not convinced that such results actually hold. We refer the reader, for instance, to the discussions in [7].

The result represents also an improvement of the results presented in [6]. In that paper the preprocessing time needed to answer in constant time queries asking whether a factor of a word is a k-repetition, k-free or overlap-free, where k is a positive integer, is $\mathcal{O}(n^2)$, and the time needed for updating the data structures is $\mathcal{O}(n \log n)$ in a worst case analysis and $\mathcal{O}(n)$ in an amortized analysis. Moreover, several complicated data structures are used there. It is immediate how **per** queries can be used to answer, in $\mathcal{O}(1)$ time, queries asking whether a factor is a repetition, in fact a particular case of **per** queries, and, as shown in [6], all the other types of mentioned queries. Thus, following the reasoning in the afore mentioned paper, one can easily obtain that the preprocessing time needed to answer in constant time queries asking whether a factor of a word is a k-repetition, k-free or overlap-free, for some positive integer k, is $\mathcal{O}(n^2)$, and the time needed for updating the data structures is $\mathcal{O}(n)$ in the worst case. Furthermore, the data structures that we need here are quite simple.

In the following we change our focus to constructing, in linear time, a p-periodic partial word, starting from a full word, by replacing some of its symbols with holes such that no two consecutive holes are too close one to the other:

Problem 2 *[4] Given a word $w \in V^*$, and positive integers d and p, $d,p \leq |w|$, decide whether there exists a p-periodic d-valid partial word contained in w.*

The input of this problem consists of the word w and the numbers p and d. The alphabet V can be arbitrarily large, but, clearly, we can assume that $|V| \leq n$ (that is, we do not care about symbols that do not appear in w).

We were able to improve the algorithmic result from [4] as follows.

Theorem 2 *Problem 2 can be decided in linear time $\mathcal{O}(|w|)$. A solution for this Problem can be obtained in the same time complexity.*

The proof of this result is based on a linear time algorithm that transforms an instance of Problem 2 to an instance of the **2-SAT** problem.

More over, we can consider the following optimization problem:

Problem 3 *Given a word $w \in V^*$ and two positive integers d and p, both less than $|w|$, construct a p-periodic d-valid partial word contained in w, and having a minimum number of holes.*

In this case, we get the following result, by combining the solution of Problem 2 with a dynamic programming strategy:

Theorem 3 *Problem 3 can be decided in linear time. A solution for this Problem can also be obtained in the same time complexity.*

Once again this result is optimal, with respect to the time complexity.

For an extensive presentation of the results of this paper, including their proofs, see [10] and the technical report [11].

References

[1] F. BLANCHET-SADRI, *Algorithmic Combinatorics on Partial Words*. Chapman & Hall / CRC Press, 2008.

[2] F. BLANCHET-SADRI, A. R. ANAVEKAR, Testing primitivity on partial words. *Discrete Applied Mathematics* **155** (2007) 3, 279–287.

[3] F. BLANCHET-SADRI, A. CHRISCOE, Local periods and binary partial words: an algorithm. *Theoretical Computer Science* **314** (2004) 1–2, 189–216.

[4] F. BLANCHET-SADRI, R. MERCAŞ, A. RASHIN, E. WILLETT, An Answer to a Conjecture on Overlaps in Partial Words Using Periodicity Algorithms. In: A. H. DEDIU, A. M. IONESCU, C. MARTÍN-VIDE (eds.), *LATA 2009*. Lecture Notes in Computer Science 5457, Springer, Berlin, Germany, 2009, 188–199.

[5] C. CHOFFRUT, J. KARHUMÄKI, Combinatorics of Words. In: G. ROZENBERG, A. SALOMAA (eds.), *Handbook of Formal Languages*. 1, Springer, 1997, 329–438.

[6] A. DIACONU, F. MANEA, C. TISEANU, Combinatorial queries and updates on partial words. In: *FCT 2009*. Springer, Berlin, Heidelberg, 2009, 96–108.

[7] M. J. FISCHER, M. S. PATERSON, String matching and other products. In: *Complexity of Computation*. SIAM-AMS Proceedings 7, 1974, 113–125.

[8] D. E. KNUTH, *The Art of Computer Programming*. 1: Fundamental Algorithms, Addison-Wesley, 1968.

[9] M. LOTHAIRE, *Combinatorics on Words*. Cambridge University Press, 1997.

[10] F. MANEA, R. MERCA¸S, C. TISEANU, Periodicity Algorithms for Partial Words. In: *MFCS 2011*. Springer, Berlin, Heidelberg, 2011, 472–484.

[11] F. MANEA, R. MERCAŞ, C. TISEANU, *Periodicity Algorithms for Partial Words*. Technical report, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, 2010.
`http://theo.cs.uni-magdeburg.de/pubs/preprints/pp-afl-2011-03.pdf`

THEORIE-TAG 2011
J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 89 – 92

Pushdown Automata with Translucent Pushdown Symbols

Benedek Nagy[(A)] Friedrich Otto[(B)] Marcel Vollweiler[(B)]

[(A)]Department of Computer Science, Faculty of Informatics
University of Debrecen, 4032 Debrecen, Egyetem tér 1., Hungary
nbenedek@inf.unideb.hu

[(B)]Fachbereich Elektrotechnik/Informatik
Universität Kassel, 34109 Kassel, Germany
{otto,vollweiler}@theory.informatik.uni-kassel.de

Abstract

Pushdown automata with translucent pushdown symbols are presented. Such a device is a pushdown automaton M that is equipped with a 'transparency relation' τ. In a transition step, M does not read (and replace) the topmost symbol on its pushdown store, but the topmost one that is only covered by translucent symbols. We prove that these automata accept all recursively enumerable languages. Then we concentrate on a restricted version of pushdown automata with translucent pushdown symbols that work in real-time or in quasi-real-time. We compare the corresponding language classes to other classical complexity classes, and we study closure properties for these language classes.

1. Introduction

The *pushdown automaton* (pda, for short) is one of the most basic and most important machine models considered in computability and formal language theory (see, e.g., [5]). It characterizes the context-free languages, and this characterization remains valid even in the absence of states. On the other hand, a (deterministic) pda with two pushdown stores can simulate a single-tape Turing machine, and hence, it is a model of a universal computing device. Further, nondeterministic pdas with three pushdown stores running in real-time characterize the class Q of *quasi-real-time languages*, which coincides with the complexity class $\mathsf{NTIME}(n)$ of languages that are accepted by nondeterministic Turing machines in linear time [2].

In the literature also many other variants of pdas have been studied. For example, the *stack automaton* is a pda that has a two-way read-only input, and that can scan the complete contents of its pushdown store (see, e.g., [5]). A *flip-pushdown automaton* is a pda that can reverse the contents of its pushdown stack [4, 11]. And in [6] the so-called *shadow-pushdown automaton* is introduced, which has a 'half-translucent' copy for each of its pushdown symbols. Each

[(A)]The first author was supported by the TÁMOP 4.2.1/B-09/1/KONV-2010-0007 project, which is implemented through the New Hungary Development Plan, co-financed by the European Social Fund and the European Regional Development Fund.

pushdown operation accesses the topmost ‘normal’ pushdown symbol on the pushdown store together with the half-translucent symbol directly above it. By putting certain restrictions on this type of pushdown automaton, all the classes of the Chomsky hierarchy can be characterized.

Here, motivated by recent work on finite-state acceptors and pushdown automata with *translucent input symbols* [7, 8, 9], we introduce and study pushdown automata with *translucent pushdown symbols*.

2. Pushdown Automata with Translucent Pushdown Symbols

A pushdown automaton with translucent pushdown symbols (tl-PDA for short) is a usual pushdown automaton that is equipped with an additional ‘transparency relation’ τ:

$$M = (Q,\Sigma,\Gamma,\$,\perp,q_0,F,\delta,\tau),$$

whereby Q is a finite set of internal states, Σ is the input alphabet, Γ is the pushdown alphabet, \$ is a right end marker for the input tape, $\perp$ is the bottom symbol of the pushdown, q_0 is the initial state, F is a set of final states, and δ is the transition relation. To each pair (q,a) consisting of an internal state q of M and an input symbol a, τ assigns a subset of pushdown symbols that are ‘translucent’ (not visible) for state q and input symbol a. Accordingly, in a corresponding transition step, M does not read (and replace) the topmost symbol on its pushdown store, but the topmost visible one that is only covered by translucent symbols, that is, by symbols from the set $\tau(q,a)$.

In general a tl-PDA is nondeterministic: $\delta \subseteq (Q \times (\Sigma \cup \{\$,\varepsilon\}) \times (\Gamma \cup \{\perp\})) \times (Q \times \Gamma^*)$. If, however, the transition relation is a partial function $\delta : (Q \times (\Sigma \cup \{\$,\varepsilon\}) \times (\Gamma \cup \{\perp\})) \to (Q \times \Gamma^*)$, then M is a *deterministic pushdown automaton with translucent pushdown symbols*, abbreviated as tl-DPDA. Here it is required that $\delta(q,a,B) = \emptyset$ for *all* $a \in \Sigma \cup \{\$\}$ and *all* $B \in \Gamma \cup \{\perp\}$, if $\delta(q,\varepsilon,A) \neq \emptyset$ for some $A \in \Gamma \cup \{\perp\}$. Thus, in state q an ε-step is applicable only if in that particular state, no other transition can be used at all. This restriction is stronger than the corresponding restriction for traditional deterministic pushdown automata because of the transparency relation τ.

At the beginning of each computation a tl-(D)PDA M is in the initial state, the pushdown store is empty (except for the $\perp$-symbol), the input word followed by the right end marker \$ is placed on the input tape, and the input tape window is positioned at the first symbol of the input. The automaton M accepts the input if, beginning at the initial situation, it reaches a final state and the input has completely been read (the input tape window is positioned on the \$-symbol).

The language classes characterized by deterministic and nondeterministic tl-PDAs are denoted with $\mathcal{L}(\mathsf{tl\text{-}DPDA})$ and $\mathcal{L}(\mathsf{tl\text{-}PDA})$.

We say that a tl-(D)PDA works in real-time if it processes exactly one input symbol in each computation step and moves the window of the input tape one position to the right. It works in quasi-real-time if there exists a constant c so that the automaton is allowed to apply at most c computation steps between processing two input symbols or reading the \$-symbol. The according language classes are denoted with $\mathcal{L}_{\mathrm{rt}}(\mathsf{tl\text{-}(D)PDA})$ and $\mathcal{L}_{\mathrm{qrt}}(\mathsf{tl\text{-}(D)PDA})$.

3. Results

Our main result is the computational universality of tl-PDAs even in the deterministic case ($\mathcal{L}(\textsf{tl-(D)PDA}) = \textsf{RE}$). We prove this with a step-by-step simulation of a deterministic on-line Turing machine with a single working tape (on-DTM$_1$). This device consists of a finite control, a one-way read-only input tape with a right end marker \$, and a working tape (see, e.g., [1] or [10] for further details), and it is well-known that it accepts all recursively enumerable languages.

The simulation can also be applied to nondeterministic on-TM$_1$'s as well as to (deterministic and nondeterministic) machines that work in real-time, whereby real-time is defined for on-line Turing machines similar as for the tl-PDAs. For the deterministic case we show that this inclusion is even proper using a result of [3] ($\mathcal{L}_{\text{rt}}(\textsf{on-DTM}_1) \subset \mathcal{L}_{\text{rt}}(\textsf{tl-DPDA})$, $\mathcal{L}_{\text{rt}}(\textsf{on-TM}_1) \subseteq \mathcal{L}_{\text{rt}}(\textsf{tl-PDA})$). Moreover all on-DTMs with multiple working tapes can be simulated by quasi-real-time tl-DPDAs: $\bigcup_{k\geq 1} \mathcal{L}(\textsf{on-DTM}_k) \subseteq \mathcal{L}_{\text{qrt}}(\textsf{tl-DPDA})$.

Further, it is shown that $\mathcal{L}_{\text{rt}}(\textsf{tl-DPDA})$ is not a subset of $\mathcal{L}(\textsf{on-DTM}_k)$ for any $k \geq 1$.

In addition, the set of quasi-real-time languages $\textsf{Q}$ can be accepted by tl-PDAs working in quasi-real-time ($\textsf{Q} \subseteq \mathcal{L}_{\text{qrt}}(\textsf{tl-PDA})$). This class is known as the set of all languages accepted by any on-line multi-tape Turing machine working in real-time [2] and coincides with the class $\textsf{NTIME}(n)$.

The table below shows some closure properties for the language classes considered. The meaning of the operations are from left to right as follows: union, intersection, complementation, intersection with a regular language, concatenation, application of an ε-free morphism, and inverse morphism. The +-symbol denotes that a language class is closed under a particular operation and the question mark denotes that this is still open.

Type of tl-PDA	**Operations**						
	$\cup$	$\cap$	^{c}	$\cap_{REG}$	$\cdot$	h_ε	h^{-1}
$\mathcal{L}_{\text{rt}}(\textsf{tl-DPDA})$	?	?	+	+	?	?	?
$\mathcal{L}_{\text{qrt}}(\textsf{tl-DPDA})$	+	+	+	+	?	?	+
$\mathcal{L}_{\text{rt}}(\textsf{tl-PDA})$	+	?	?	+	+	+	?
$\mathcal{L}_{\text{qrt}}(\textsf{tl-PDA})$	+	+	?	+	+	+	+

4. Conclusion

The tl-PDA is a quite powerful machine model, as all recursively enumerable languages are accepted by it. Even when restricted to real-time computations, the deterministic tl-PDA accepts languages that are not even accepted by deterministic real-time on-line Turing machines with any fixed number of working tapes. In the nondeterministic case, all quasi-real-time languages are accepted by tl-PDAs that run in quasi-real-time. However, it remains open whether this inclusion is proper. Also it is still open whether each real-time tl-DPDA can be simulated by a deterministic on-line Turing machine running in real-time and using multiple working tapes or if deterministic on-line Turing machines with more than one working tape can be simulated by tl-DPDAs both working in real-time. Finally, we have established a number of closure properties for the language classes that are defined by deterministic and nondeterministic tl-PDAs running in real-time or quasi-real-time. However, we have not been able to establish any non-closure properties yet. This is certainly an area where more work needs to be done.

References

[1] S. AANDERAA, On k-tape versus $(k-1)$-tape real time computation. In: R. KARP (ed.), *Complexity of Computation, SIAM-AMS Symp. in Appl. Math., Proc.*. American Mathematical Society, Providence, RI, 1974, 75–96.

[2] R. BOOK, S. GREIBACH, Quasi-realtime languages. *Math. System Theory* **4** (1970), 97–111.

[3] D. HOLT, C. RÖVER, On real-time word problems. *J. London Math. Soc.* **67** (2003), 289–301.

[4] M. HOLZER, M. KUTRIB, Flip-pushdown automata: $k+1$ pushdown reversals are better than k. In: J. BAETEN, J. LENSTRA, J. PARROW, G. WOEGINGER (eds.), *ICALP 2003, Proc.*. LNCS 2719, Springer-Verlag Berlin, 2003, 490–501.

[5] J. HOPCROFT, J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, M.A., 1979.

[6] B. NAGY, An automata-theoretic characterization of the Chomsky hierarchy. In: J. KRATOCHVIL, A. LI, J. FIALA, P. KOLMAN (eds.), *TAMC 2010, Proc.*. LNCS 6108, Springer-Verlag Berlin, 2010, 361–372.

[7] B. NAGY, F. OTTO, CD-systems of stateless deterministic R(1)-automata accept all rational trace languages. In: A. DEDIU, H. FERNAU, C. MARTÍN-VIDE (eds.), *LATA 2010, Proc.*. LNCS 6031, Springer-Verlag Berlin, 2010, 463–474.

[8] B. NAGY, F. OTTO, An automata-theoretical characterization of context-free trace languages. In: I. ČERNÁ, T. GYIMÓTHY, J. HROMKOVIČ, K. JEFFEREY, R. KRÁLOVIČ, M. VUKOLIČ, S. WOLF (eds.), *SOFSEM 2011: Theory and Practice of Computer Science, Proc.*. LNCS 6543, Springer-Verlag Berlin, 2011, 406–417.

[9] B. NAGY, F. OTTO, Finite-state acceptors with translucent letters. In: G. BEL-ENGUIX, V. DAHL, A. D. L. PUENTE (eds.), *BILC 2011: AI Methods for Interdisciplinary Research in Language and Biology, Proc.*. SciTePress, Portugal, 2011, 3–13.

[10] K. REISCHUK, *Komplexitätstheorie, Band I: Grundlagen*. B.G. Teubner, Stuttgart, Leipzig, 1999.

[11] P. SARKAR, Pushdown automaton with the ability to flip its stack. In: *Electronic Colloquium on Computational Complexity (ECCC)*. 2001.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 93 – 95

On the Interval-Bound Problem for Weighted Timed Automata

Karin Quaas

Institut für Informatik, Universität Leipzig
04009 Leipzig

During the last years, *weighted timed automata* [2, 3] have received much attention in the real-time community. A weighted timed automaton is a timed automaton [1] extended with weight variables, whose values may grow or fall linearly with time while being in a state, or discontinuously grow or fall during a state change. As an example (taken from [5]), consider the weighted timed automaton in Fig. 1. It shows a finite automaton with three states, one real-valued clock variable x and one real-valued weight variable w. The initial value of x is zero. The value of x grows continuously while time elapses in a state, whereas the change from one state to another does not cost any time. At edges one may put some restrictions on the value of x; for instance, the edge from s_3 to s_1 can only be taken if the value of x equals 1. Moreover, one may reset the value of x to zero, indicated by the label $x := 0$. The initial value of w is set to 2. While being in a state, the value of w grows linearly with time depending on the rate of the state. For instance, in state s_1 the value of w decreases with rate 3, whereas in state s_2 it increases with rate 6. One may also discontinuously add or subtract some value from the value of w during a state change (not shown in this example). In this way, weighted timed automata can be used to model both continuous and discrete production and consumption of resources, like energy, bandwidth or money. This allows for interesting applications e.g. in operations research, in particular, *optimal* scheduling.

Recently [5], three interesting resource scheduling problems for weighted timed automata were introduced: the existence of an infinite run during which the values of the weight variables never fall below zero (*lower-bound*), never fall below zero and never exceed a given upper bound (*interval-bound*), and never fall below zero and never exceed a given *weak* upper bound, meaning that when the weak upper bound is reached, the value of the weight variable is not increased but maintained at this level (*lower-weak-upper-bound*). For instance, the weighted timed automaton in Fig. 1 allows for an infinite run such that the value of w is always within the interval $[0,2]$: One may always spend exactly two third time units in state s_1 (decreasing the value of w to 0) and one third time units in s_2 (increasing the value of w back to 2). In contrast to this, there is no infinite run such that the value of w is always within the interval $[0,2]$ if

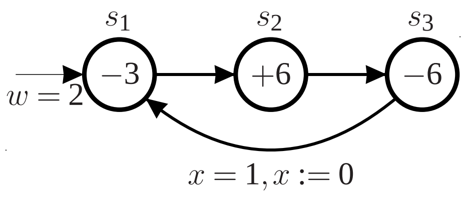


Figure 1: A weighted timed automaton with one clock variable and one weight variable.

the initial value of w is $1\frac{1}{2}$. However, there is an infinite run if we consider 2 as a *weak* upper bound [5].

For weighted timed automata with one clock and one weight variable, the lower-bound-problem and the lower-weak-upper-bound-problem are decidable in polynomial time [5], albeit with the restriction that the weighted timed automaton does not allow for discontinuous updates of the weight variables at the edges. In [4] it is proved that the lower-bound-problem is also decidable if this restriction is lifted and if the values of the weight variables may not only change linearly but also exponentially in time. The interval-bound-problem is undecidable for weighted timed automata with one clock and one weight variable *in a game setting*; in the corresponding existential setting, however, the problem was open.

In a recent paper [10], we show that the interval-bound-problem is undecidable for weighted timed automata with two clocks and two weight variables (or more). The proof is a reduction from the infinite computation problem for two-counter machines [9]. The undecidability proof does not require discrete updates of the weight variables at the edges of the automaton. A similar proof idea can be used to show that the interval-bound-problem is also undecidable for weighted timed automata with one clock, two weight variables and discrete updates of the weight variables.

As a second main result, we show that the interval-bound-problem is PSPACE-complete if we restrict the time domain to the natural numbers. This result is irrespective of the number of clocks and weight variables. The proof for PSPACE-membership is based on a polynomial-time reduction to the recurrent reachability problem for timed automata [7, 1]. We show that one can encode the values of natural-number valued weight variables into the values of clock variables.

For real-valued clocks, Bouyer and Markey recently proved that the interval-bound-problem is undecidable for weighted timed automata with two clocks and one weight variable [6]. The undecidability of the same problem for weighted timed automata with one clock, two weight variables and discrete updates of the weight variables has independently shown by Fahrenberg et al. [8]. Table 1 summarizes the results known so far for real-valued clocks and weight variables. The grey cells indicate instances of the problem for which there is no result known so far. For instance, the decidability of the interval-bound problem for weighted timed automata over the reals with exactly one clock or one weight variable is still an interesting open problem.

	One clock, one weight variable		Two clocks, two weight variables	
$\exists$ infinite run?	without edge weight	with edge weight	without edge weight	with edge weight
$w_i \in [0,\infty]$	P [5]	EXPTIME [5]		
$w_i \in [0,u]$			undecidable [10]	undecidable [10]

	Two clocks, one weight variable		One clock, two weight variables	
$\exists$ infinite run?	without edge weight	with edge weight	without edge weight	with edge weight
$w_i \in [0,\infty]$				
$w_i \in [0,u]$		undecidable [6]		undecidable [10, 8]

Table 1: Results for the Interval-Bound-Problem

References

[1] R. ALUR, D. L. DILL, A theory of timed automata. *Theoretical Computer Science* **126** (1994) 2, 183–235.

[2] R. ALUR, S. LA TORRE, G. J. PAPPAS, Optimal Paths in Weighted Timed Automata. *Theoretical Computer Science* **318** (2004), 297–322.

[3] G. BEHRMANN, A. FEHNKER, T. HUNE, K. LARSEN, P. PETTERSSON, J. ROMIJN, F. VAANDRAGER, Minimum-Cost Reachability for Priced Timed Automata. In: M. D. D. BENEDETTO, A. SANGIOVANNI-VINCENTELLI (eds.), *HSCC*. LNCS 2034, Springer, 2001, 147–161.

[4] P. BOUYER, U. FAHRENBERG, K. G. LARSEN, N. MARKEY, Timed Automata with Observers under Energy Constraints. In: K. H. JOHANSSON, W. YI (eds.), *Proceedings of the 13th International Conference on Hybrid Systems: Computation and Control (HSCC'10)*. ACM Press, Stockholm, Sweden, 2010, 61–70.
`http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/BFLM-hscc10.pdf`

[5] P. BOUYER, U. FAHRENBERG, K. G. LARSEN, N. MARKEY, J. SRBA, Infinite Runs in Weighted Timed Automata with Energy Constraints. In: F. CASSEZ, C. JARD (eds.), *FORMATS*. LNCS 5215, Springer, 2008, 33–47.

[6] P. BOUYER-DECITRE, N. MARKEY, Personal communication.

[7] C. COURCOUBETIS, M. YANNAKAKIS, Minimum and Maximum Delay Problems in Real-Time Systems. *Formal Methods in System Design* **1** (1992), 385–415.

[8] U. FAHRENBERG, L. JUHL, K. G. LARSEN, J. SRBA, Energy Games in Multiweighted Automata. In: *ICTAC 2011*. 2011. To appear.

[9] M. MINSKY, *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ, 1967.

[10] K. QUAAS, On the Interval-Bound Problem for Weighted Timed Automata. In: A. H. DEDIU, S. INENAGA, C. MARTÍN-VIDE (eds.), *LATA 2011*. LNCS 6638, Springer, 2011, 452–464.

HR* Graph Conditions Versus Counting Monadic Second-Order Graph Formulas

Hendrik Radke

Carl von Ossietzky-Universität Oldenburg
hendrik.radke@informatik.uni-oldenburg.de

Abstract

Graph conditions are a visual formalism to express properties of graphs as well as access conditions for graph transformation rules. Since nested graph conditions are equivalent to first-order logic on graphs, non-local conditions cannot be expressed by them. We propose HR* graph conditions as a generalization of nested graph conditions which can express a multitude of interesting non-local properties (like the existence of a path of arbitrary length, circle-freeness or an even number of nodes). This is achieved by using hyperedges as placeholders and substituting these according to a hyperedge replacement system. We compare the expressiveness of HR* graph conditions to counting monadic second-order graph formulas.

1. Introduction

Increasing complexity in software systems calls for design concepts that allow an intuitive overview of a system. We use graph transformation systems and model states by graphs and state changes by graph programs. Structural system properties can be described by nested graph conditions [4]. However, nested graph conditions cannot describe non-local properties like connectedness or circle-freeness [1], which are of interest in constraints or application conditions. We therefore propose an extension of nested graph conditions that can express such properties. This extension is called HR* *conditions* [7], and adds variables in the form of hyperedges to the conditions. These hyperedges serve as placeholders for graphs and can be substituted according to a given hyperedge replacement system.

2. HR* Graph Conditions

HR* graph conditions combine graphs (or rather: graph morphisms) with first-order logic operators and hyperedge replacement.

Definition 2.1 (HR* **(graph) conditions)** *A* HR* *(graph) condition over P is of the form* true, $\exists(P \to C, c)$*, or* $\exists(P \sqsupseteq C, c)$*, where c is a* HR* *condition over C. Furthermore, Boolean expressions over conditions over C (i.e. $c \wedge d$ or $\neg c$) are conditions over C. The following abbreviations are used: $\forall(P \to C, c)$ abbreviates $\neg\exists(P \to C, \neg c)$, $\exists(P \to C)$ abbreviates $\exists(P \to C, \mathrm{true})$ and $\exists(C, c)$ abbreviates $\exists(P \to C, c)$ if P is clear from the context (i.e. if P is the empty graph or the condition is a subcondition in some bigger condition).*

All HR* graph conditions in this paper are conditions over the empty graph $\emptyset$, so the last abbreviation is used extensively to make the conditions shorter and easier to read. To each HR* graph condition belongs a hyperedge replacement system which regulates the substitution of variables. The semantics of HR* graph conditions is straightforward: To check whether a graph satisfies a HR* condition, the variables in the condition are substituted, then the graphs in the condition are matched to the graph in question.

Definition 2.2 (semantics of HR* **graph conditions)** *A graph G satisfies condition c, short $G \models c$, iff there is a substitution σ according to the hyperedge replacement system, and a morphism $p\colon \emptyset \to G$ satisfies c by σ. A morphism $p\colon P \to G$ satisifies condition $\exists(P \to C, c)$ by σ iff there is an injective morphism $q\colon \sigma(C) \to G$ such that $p = q \circ \sigma(a)$ and q satisfies c by σ. p satisfies condition $\exists(P \sqsupseteq C, c)$ by σ iff $p(\sigma(C)) \subseteq p(\sigma(P))$ and $p(\sigma(C))$ satisfies c. Every morphism satisfies* true *by σ. The satisfaction of the Boolean operators is as usual.*

Example 2.3 *The following example* HR* *condition expresses the property "There is a path where every node has (at least) three more outgoing edges to different nodes".*

$$\exists\Big(\underset{1}{\bullet}\xrightarrow{\;+\;}\underset{2}{\bullet}, \forall\Big(\underset{1}{\bullet}\xrightarrow{\;+\;}\underset{2}{\bullet} \sqsupseteq \underset{3}{\bullet}, \exists\Big(\begin{smallmatrix}\bullet & \bullet & \bullet\\ \nwarrow & \uparrow & \nearrow\\ & \underset{3}{\bullet} & \end{smallmatrix}\Big)\Big)\Big) \text{ *with* } \underset{1}{\bullet}\xrightarrow{\;+\;}\underset{2}{\bullet} ::= \underset{1}{\bullet}\longrightarrow\underset{2}{\bullet} \mid \underset{1}{\bullet}\longrightarrow\bullet\xrightarrow{\;+\;}\underset{2}{\bullet}$$

The existence of the path is defined by the first part and the hyperedge replacement system given. The universal quantifier quantifies over all nodes of the path, and finally, the innermost quantor ensures the existence of three outgoing edges to different nodes.

Theorem 2.4 *For finite graphs, the problem whether a graph G satisfies an* HR* *condition c is decidable [5]*

Proof sketch: In a naïve approach, one could, for every hyperedge in c, derive every graph permitted by the corresponding replacement system. Afterwards, it is easy to eliminate subconditions of the form $\exists(P \sqsupseteq C, c)$ by checking inclusion of C in P, yielding either $\neg$true or c. The resulting conditions are nested graph conditions, for which the satisfaction problem is decidable [6]. Since every hyperedge replacement system can be transformed into a montonous one [3] and no condition with a generated graph larger than G may be satisfied, the number of nested graph conditions to check is finite.

3. Expressiveness

Since HR* graph conditions are a generalization of nested graph conditions, they are at least as expressive as first-order logic on graphs. That means one can express properties like "every a-labeled node has an incoming edge from a c-labeled node": $\forall\Big(\mathrm{a}, \exists(\mathrm{a} \leftarrow \mathrm{c})\Big)$.

We have also shown in [5] that they are more expressive than monadic second-order logic (MSO): Every MSO formula can be transformed into an equivalent HR* condition. An MSO property like "the graph contains no circle" can easily be expressed by the HR* condition

$$\neg\exists(\bullet \circlearrowleft +) \text{ with } \underset{1}{\bullet}\xrightarrow{\;+\;}\underset{2}{\bullet} ::= \underset{1}{\bullet}\longrightarrow\underset{2}{\bullet} \mid \underset{1}{\bullet}\longrightarrow\bullet\xrightarrow{\;+\;}\underset{2}{\bullet}.$$

The condition

$$\exists\left(\boxed{\text{even}}, \nexists\left(\boxed{\text{even}}\,\bullet\right)\right) \text{ with } \boxed{\text{even}} ::= \emptyset \;\Big|\; \begin{matrix}\bullet\\ \bullet\end{matrix}\boxed{\text{even}}$$

corresponds to the property “the graph has an even number of nodes”, which cannot be expressed in MSO, but in *counting* MSO, which leads to the next theorem.

Theorem 3.1 *For every counting MSO formula over nodes (CMSO$_1$) f, there is an equivalent* HR* *graph condition* Cond(f).

CMSO$_1$ (as defined in [2]) is an extension of MSO by an additional “counting” quantor $\exists^{(m)}x.\phi(x)$. A graph satisfies such a formula iff $\phi(x)$ holds for exactly n nodes in the graph with $n \equiv 0 \pmod{m}$.

This can be translated into a HR* condition by using hyperedge replacement to count the nodes: It is easy to construct a grammar which generates all discrete graphs (i.e. with no edges) with $n * m$ nodes, where m is a fixed number and $n \in \mathbb{N}$ is variable. For all nodes inside the generated subgraph, the property to be counted is checked. Also, the property must not hold for any node outside of the generated subgraph. For a CMSO$_1$ formula $f = \exists^{(m)}x.\phi(x)$, let $\mathrm{Cond}(f) = \exists(\boxed{\text{Y}}, \forall(\underset{1}{\bullet} \sqsubseteq \boxed{\text{Y}}, \mathrm{Cond}(\phi(x))[x/\underset{1}{\bullet}]) \wedge \nexists(\boxed{\text{Y}}\ \underset{2}{\bullet}, \mathrm{Cond}(\phi(x)[x/\underset{2}{\bullet}])))$, with the hyperedge replacement system $Y ::= \emptyset \mid \boxed{\text{Y}}\ D_m$, where D_m is the discrete graph consisting of exactly m nodes.

Proof sketch: Assume that the theorem holds for ϕ, i.e. $G \models \mathrm{Cond}(\phi) \iff G \models \phi$. Using the semantics of HR* conditions, a graph G satisfies Cond$(\exists^{(m)}x.\phi(x))$ iff there is a substituion for Y such that all nodes x in Y fulfill Cond$(\phi(x))$, and no node z outside Y fulfills Cond$(\phi(z))$. This implies that the number of nodes that fulfill Cond(ϕ) is equal to the number of nodes in Y. It is easy to show that Y can be substituted by exactly $n * m$ nodes, where m is a fixed number and $n \in \mathbb{N}$ arbitrary. By our assumption, it follows that the number of nodes in G satisfy ϕ is exactly $n * m$, vulgo $0 \mod m$.

Summarizing the above, we can say that HR* graph conditions are a graphical formalism to express graph properties that is at least as strong as monadic second order formulas on graphs with node counting. Future work will give more insights into the expressiveness of the conditions. For example, it is still unknown whether HR* graph conditions are as strong as MSO formulas with *edge* counting, though the author suspects that this is not the case. It might also be worthwile to compare the power of the conditions to second-order formulas on graphs.

References

[1] H. GAIFMAN, On Local and Non-Local Properties. In: J. STERN (ed.), *Proceedings of the Herbrand Symposium: Logic Colloquium’81*. North Holland Pub. Co., 1982, 105–135.

[2] T. GANZOW, S. RUBIN, Order-Invariant MSO is Stronger than Counting MSO in the Finite. In: *25th Int. Symposium on Theoretical Aspects of Computer Science (STACS)*. 2008, 313–324.

[3] A. HABEL, *Hyperedge replacement: grammars and languages*. LNCS 643, Springer, 1992.

[4] A. HABEL, K.-H. PENNEMANN, Correctness of High-Level Transformation Systems Relative to Nested Conditions. *MSCS* **19** (2009), 245–296.

[5] A. HABEL, H. RADKE, Expressiveness of Graph Conditions with Variables. *ECEASST* **30** (2010).

[6] K.-H. PENNEMANN, *Development of Correct Graph Transformation Systems*. Ph.D. thesis, Universität Oldenburg, 2009.

[7] H. RADKE, Weakest Liberal Preconditions relative to HR* Graph Conditions. In: *3rd Int. Workshop on Graph Computation Models (GCM 2010), Preproceedings*. 2010, 165–178. ISSN 0929-0672.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 101 – 104

Automata for Languages Defined by Backreferencing

Daniel Reidenbach Markus L. Schmid

Department of Computer Science, Loughborough University,
Loughborough, Leicestershire, LE11 3TU, United Kingdom
{D.Reidenbach,M.Schmid}@lboro.ac.uk

Abstract

We introduce and investigate Nondeterministically Bounded Modulo Counter Automata (NBMCA), which are automata that comprise a constant number of modulo counters, where the counter bounds are nondeterministically guessed, and this is the only element of nondeterminism. This model is tailored for languages defined by backreferencing.

1. Languages Defined by Backreferencing

It is a common and convenient way to define a formal language by specifying a pattern that all words in the languages need to follow. Such a pattern is often considered to be a factorisation with conditions on the factors. An example is the regular set of all words $w \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ with $w = u \cdot v$, $u \in \{\mathtt{a}, \mathtt{b}\}^*$, $|u| \geq 10$, or the set of all words $w' \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}^*$ with $w' = u_1 \cdot u_2 \cdot u_3 \cdot u_4$, $|u_1|_{\mathtt{a}} = |u_4|_{\mathtt{a}}$, $|u_2|_{\mathtt{b}} = |u_3|_{\mathtt{b}}$ (where $|u|_x$ denotes the number of occurrences of symbol x in u), which is a context-free language. The special condition that several factors of the factorisation need to be identical is called a *backreference*. Using backreferences we can easily define languages that are not even context-free anymore, e. g., the set of all words w that can be factorised into $w = u \cdot v \cdot v \cdot u$. This is exactly the concept that is used in the models of so-called *pattern languages* as introduced by Angluin [1], their extensions, and *extended regular expressions with backreferences* (*REGEX* for short) (see, e. g., [2]). REGEX languages can be considered a generalisation of pattern languages. While backreferences clearly increase the expressive power, they are also responsible for the membership problem of these language classes to become NP-complete. This is particularly worth mentioning as today's text editors and programming languages (such as Perl, Python, Java, etc.) use individual notions of extended regular expressions, and they all provide so-called *REGEX engines* to conduct a *match test*, i. e., to compute the solution to the membership problem for any language given by a REGEX and an arbitrary string. Hence, despite its theoretical intractability, algorithms that perform the matchtest for REGEX are a practical reality. This suggests that there is a need for researching this particular NP-complete problem in order to optimise these REGEX engines. However, while in fact recent work focuses on implementations of REGEX engines, the aspect of backreferences is mostly neglected. For example, many implementations of REGEX engines impose restrictions on the backreferences – e. g., by limiting their number to 9 – in order to manage their computational complexity and recent developments of REGEX engines that are particularly tailored to efficiency even completely abandon the support of backreferences (see, e. g., Google's RE2 [3] and Le Maout [6]), so that they can make use of the well-developed theory of finite automata as acceptors of regular languages. These optimised approaches are mainly based on deterministic finite automata (DFA), which contrasts with the situation that most existing implementations of extended regular expressions with backreferences are more or less sophisticated backtracking-based brute-force algorithms that make no use of finite automata at all. This shows that, while finite automata generally seem to be the desired algorithmic framework, it is problematic to incorporate the concept of backreferences into this approach. Thus, the provided formal models do not fit with the practical needs that led to the introduction of backreferences.

We now informally introduce an automaton model that is tailored to handle backreferences, namely the Nondeterministically Bounded Modulo Counter Automata (NBMCA). Furthermore, we exhibit its main features that distinguish it from existing models and, hence, shall be the focus of our analysis. The model comprises a two-way input head and a number of counters. For every counter of an NBMCA an individual counter bound is provided, and every counter can only be incremented and counts modulo its counter bound. The current counter values and counter bounds are hidden from the transition function, which can only check whether a counter has reached its bound. By performing a reset on a counter, the automaton nondeterministically guesses a new counter bound between 0 and $|w|$, where w is the input word. This guessing of counter bounds is the only possible nondeterministic step of NBMCA, and the transition function is defined completely deterministically.

As described above, the membership problem of languages like pattern languages or REGEX languages can be solved by checking whether or not there exists a certain factorisation for the input word. It is easy to see that the complexity of this task results from the potentially large number of different factorisations rather than from checking whether or not a factorisation satisfies the required conditions (as long as these conditions are not too complex). In this regard, NBMCA can be seen as classical *deterministic* finite automata equipped with the ability to nondeterministically guess a factorisation of the input word and to maintain this factorisation for the time of the computation. This is done by interpreting the nondeterministically chosen counter bounds as lengths of factors, and the finite state control can then be used in order to compare factors or to check them for certain regular conditions. This approach provides an interface for tackling the intrinsic complexity of the problem, i. e., reducing the number of factorisations that need to be checked, while at the same time sophisticated DFA based algorithms can be incorparated where factors are deterministically checked for regular conditions.

Compared to classical multi-head automata, NBMCA have several particularities. All additional resources the automaton is equipped with, namely the counters, are tailored to storing positions in the input word. This aspect is not really new and can be compared with the situation of multi-head automata with only one reading head and several blind heads (see, e. g., Ibarra and Ravikumar [5]). However, there is still one difference: the counters of NBMCA are quite limited in their ability to change the positions they represent, since their values can merely be incremented and count modulo a counter bound, which is nondeterministically guessed. Moreover, the nondeterminism of NBMCA, which merely allows positions in the input word to be guessed, differs quite substantially from the common nondeterminism of automata, which is provided by a nondeterministic transition function and is, thus, designed to allow a choice of several next transitions. Nevertheless, these automata often use their nondeterminism to actually guess a certain position of the input. For example, a pushdown automaton that recognises $\{ww^R \mid w \in \Sigma^*\}$ needs to perform an unbounded number of guesses even though only one specific position, namely the middle one, of the input needs to be found. Despite this observation, the nondeterminism of NBMCA might be weaker, as it seems to *solely* refer to positions in the input. Hence, it might be worthwhile to investigate that special kind of nondeterminism.

2. Formal Details of the Automaton Model

A *Nondeterministically Bounded Modulo Counter Automaton*, NBMCA(k) for short, is a tuple $M := (k, Q, \Sigma, \delta, q_0, F)$, where $k \in \mathbb{N}$ is the number of *counters*, Q is a finite nonempty set of *states*, Σ is a finite nonempty alphabet of *input symbols*, $q_0 \in Q$ is the *initial state* and $F \subseteq Q$ is the set of *accepting states*. The mapping $\delta : Q \times \Sigma \times \{\mathtt{t}_0, \mathtt{t}_1\}^k \to Q \times \{-1, 0, 1\} \times \{0, 1, \mathtt{r}\}^k$ is called the *transition function*. An input to M is any word of the form ¢w\$, where $w \in \Sigma^*$ and the symbols ¢, \$ (referred to as *left* and *right endmarker*, respectively) are not in Σ. Let $(p, b, s_1, \dots, s_k) \to_\delta (q, r, d_1, \dots, d_k)$. We call the element b the *scanned input symbol* and r the *input head movement*. For each $j \in \{1, 2, \dots, k\}$, the element $s_j \in \{\mathtt{t}_0, \mathtt{t}_1\}$ is the *counter message of counter* j, and d_j is called the *counter instruction for counter* j. The transition function δ of an NBMCA(k) determines whether the input heads are moved to the left ($r_i = -1$), to the right ($r_i = 1$) or left unchanged ($r_i = 0$), and whether the counters are incremented ($d_j = 1$), left unchanged ($d_j = 0$) or reset ($d_j = \mathtt{r}$). In case of a reset, the counter value is set to 0 and a new counter bound is nondeterministically guessed. Hence, every counter is bounded, but these bounds are chosen in a nondeterministic way.

Let M be an NBMCA and $w := a_1 \cdot a_2 \cdot \dots \cdot a_n$, $a_i \in \Sigma$, $1 \le i \le n$. A *configuration of* M *(on input* w*)* is an element of the set $\widehat{C}_M := \{[q, h, (c_1, C_1), \dots, (c_k, C_k)] \mid q \in Q, 0 \le h \le n+1, 0 \le c_i \le C_i \le n, 1 \le i \le k\}$. The pair (c_i, C_i), $1 \le i \le k$, describes the current configuration of the i^{th} counter, where c_i is the *counter value* and C_i the *counter bound*. The element h is called the *input head position*. An *atomic move* of M is denoted by the relation $\vdash_{M,w}$ over the set of configurations. Let $(p, b, s_1, \dots, s_k) \to_\delta (q, r, d_1, \dots, d_k)$. Then, for all c_i, C_i, $1 \le i \le k$, where $c_i < C_i$ if $s_i = \mathtt{t}_0$ and $c_i = C_i$ if $s_i = \mathtt{t}_1$, and for every h, $0 \le h \le n+1$, with $a_h = b$, we define $[p, h, (c_1, C_1), \dots, (c_k, C_k)] \vdash_{M,w} [q, h', (c'_1, C'_1), \dots, (c'_k, C'_k)]$. Here, the elements h' and c'_j, C'_j, $1 \le j \le k$, are defined in the following way. $h' := h + r$ if $0 \le h + r \le n+1$ and $h' := h$ otherwise. For each $j \in \{1, \dots, k\}$, if $d_j = \mathtt{r}$, then $c'_j := 0$ and, for some $m \in \{0, 1, \dots, n\}$, $C'_j := m$. If, on the other hand, $d_j \neq \mathtt{r}$, then $C'_j := C_j$ and $c'_j := c_j + d_j \mod (C_j + 1)$.

To describe a *sequence of (atomic) moves of* M *(on input* w*)* we use the reflexive and transitive closure of the relation $\vdash_{M,w}$, denoted by $\vdash^*_{M,w}$. M accepts the word w if and only if $\widehat{c}_0 \vdash^*_{M,w} \widehat{c}_f$, where $\widehat{c}_0 := [q_0, 0, (0, C_1), \dots, (0, C_k)]$ for some $C_i \in \{0, 1, \dots, |w|\}$, $1 \le i \le k$, is an *initial configuration*, and $\widehat{c}_f := [q_f, h, (c_1, C_1), \dots (c_k, C_k)]$ for some $q_f \in F$, $0 \le h \le n+1$ and $0 \le c_i \le C_i \le n$, $1 \le j \le k$, is a *final configuration*. In every computation of an NBMCA, the counter bounds are nondeterministically initialised.

3. Properties of NBMCA

For every $k \in \mathbb{N}$, 2NFA(k) are *nondeterministic two-way automata* with k input heads.

Theorem 3.1 *For every* $k \in \mathbb{N}$,

– $\mathcal{L}(\text{NBMCA}(k)) \subseteq \mathcal{L}(\text{2NFA}(2k+1))$,

– $\mathcal{L}(\text{2NFA}(k)) \subseteq \mathcal{L}(\text{NBMCA}(\lceil \frac{k}{2} \rceil + 1))$,
– $\mathcal{L}(\text{NBMCA}(k)) \subset \mathcal{L}(\text{NBMCA}(k+2))$.

This means that neither the restrictions on the counters of NBMCA nor the special nondeterminism constitute a restriction on the expressive power. Thus, NBMCA can be used whenever classical multi-head automata can be applied, but due to their specific counters and nondeterminism they are particularly suitable algorithmic tools for recognising those languages introduced in Section 1.

For all $m_1, m_2, l, k \in \mathbb{N}$, let (m_1, m_2, l)-REV-NBMCA(k) denote the class of NBMCA(k) that perform at most m_1 input head reversals, at most m_2 counter reversals (i. e., a counter is incremented that has already reached its counter bound) and resets every counter at most l times in every accepting computation.

Theorem 3.2 *The emptiness, infiniteness and disjointness problems are decidable for the class* (m_1, m_2, l)-REV-NBMCA.

In order to answer the question of whether it is possible to ease the strong restriction of (m_1, m_2, l)-REV-NBMCA a little without losing the decidability results, we investigate decidability questions with respect to the class (m, ∞, l)-REV-NBMCA. Ibarra [4] shows that with respect to classical counter machines the typical decision problems remain undecidable if only the reversals of the input head are bounded and counter reversals are unrestricted. However, with respect to the class (m, ∞, l)-REV-NBMCA we observe that the modulo counters can still be considered restricted, since the number of resets is bounded. Intuitively, this suggests that the restrictions of the class (m, ∞, l)-REV-NBMCA might still be strong enough to provide decidability. Nevertheless, we also obtain for (m, ∞, l)-REV-NBMCA(k) that all problems are undecidable, even for small m and k, and no counter resets:

Theorem 3.3 *The emptiness, infiniteness, universe, equivalence, inclusion and disjointness problems are undecidable for the class* $(3, \infty, 0)$-REV-NBMCA(3).

References

[1] D. ANGLUIN, Finding patterns common to a set of strings. In: *Proc. 11th Annual ACM Symposium on Theory of Computing*. 1979, 130–141.

[2] C. CÂMPEANU, K. SALOMAA, S. YU, A formal study of practical regular expressions. *International Journal of Foundations of Computer Science* **14** (2003), 1007–1018.

[3] R. COX, *RE2*. Google, 2010. `http://code.google.com/p/re2/`.

[4] O. H. IBARRA, Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM* **25** (1978), 116–133.

[5] O. H. IBARRA, B. RAVIKUMAR, On partially blind multihead finite automata. *Theoretical Computer Science* **356** (2006), 190–199.

[6] V. L. MAOUT, Regular Expressions at Their Best: A Case for Rational Design. In: *Proc. 15th International Conference on Implementation and Application of Automata, CIAA 2010*. Lecture Notes in Computer Science 6482, 2011, 310–320.

THEORIE-TAG 2011 J. Dassow und B. Truthe (Hrsg.): Theorietag 2011, Allrode (Harz), 27. – 29.9.2011
Otto-von-Guericke-Universität Magdeburg S. 105 – 107

Monoid control for grammars, automata, and transducers

Georg Zetzsche

TU Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
zetzsche@cs.uni-kl.de

Abstract

During recent decades, classical models in language theory have been extended by control mechanisms defined by monoids. We study which monoids cause the extensions of context-free grammars, finite automata, or finite state transducers to exceed the capacity of the original model. Furthermore, we investigate when, in the extended automata model, the nondeterministic variant differs from the deterministic one in capacity. We show that all these conditions are in fact equivalent and present an algebraic characterization. In particular, the open question of whether every language generated by a valence grammar over a finite monoid is context-free is provided with a positive answer.

The idea of equipping classical models of theoretical computer science with a monoid (or a group) as a control mechanism has been pursued in recent decades by several authors [3, 5, 6, 7, 8, 10, 11]. This interest is justified by the fact that these extensions allow for a uniform treatment of a wide range of automata and grammar models: Suppose a storage mechanism can be regarded as a set of states on which a set of partial transformations operates and a computation is considered valid if the composition of the executed transformations is the identity. Then, this storage constitutes a certain monoid control.

For example, in a pushdown storage, the operations *push* and *pop* (for each participating stack symbol) and compositions thereof are partial transformations on the set of words over some alphabet. In this case, a computation is considered valid if, in the end, the stack is brought back to the initial state, i.e., the identity transformation has been applied. As further examples, blind and partially blind multicounter automata (see [4]) can be regarded as finite automata controlled by a power of the integers and of the bicyclic monoid (see [10]), respectively.

Another reason for studying monoid controlled automata, especially in the case of groups, is that the word problems of a group G are contained in a full trio (such as the context-free or the indexed languages) if and only if the languages accepted by valence automata over G are contained in this full trio (see, for example, [6, Proposition 2]). Thus, valence automata offer an automata theoretic interpretation of word problems for groups.

A similar situation holds for context-free grammars where each production is assigned a monoid element such that a derivation is valid as soon as the product of the monoid elements (in the order of the application of the rules) is the identity. Here, the integers, the multiplicative group of $\mathbb{Q}$, and powers of the bicyclic monoid lead to additive and multiplicative valence grammars and Petri net controlled grammars, respectively. The latter are in turn equivalent to matrix grammars (with erasing and without appearance checking, see [1] for details). Therefore, the investigation of monoid control mechanisms promises very general insights into a variety of models.

One of the most basic problems regarding these models is the characterization of those monoids whose use as control mechanism actually increases the power of the respective model. For monoid controlled automata, such a characterization has been achieved by Mitrana and Stiebe [7] for the case of groups. The author of this work was informed by an anonymous referee that a characterization for arbitrary monoids had been found by Render [9]. For valence grammars, that is, context-free grammars with monoid control, very little was known in this respect up to date. It was an open problem whether valence grammars over finite monoids are capable of generating languages that are not context-free[1] (see [3, p. 387]).

Another important question considers for which monoids the extended automata can be determinized, that is, for which monoids the deterministic variant is as powerful as the nondeterministic one. Mitrana and Stiebe [7] have shown that automata controlled by a group cannot be determinized if the group contains at least one element of infinite order. However, the exact class of monoids for which automata can be determinized was not known to date.

The contribution of this work is twofold. On the one hand, the open question of whether all languages generated by valence grammars over finite monoids are context-free is settled affirmatively. On the other hand, we use an algebraic dichotomy of monoids to provide a characterization for all the conditions above. Specifically, we show that the following assertions are equivalent:

- Valence grammars over M generate only context-free languages.
- Valence automata over M accept only regular languages.
- Valence automata over M can be determinized.
- Valence transducers over M perform only rational transductions.
- In each finitely generated submonoid of M, only finitely many elements possess a right inverse.

Note that the equivalence of the second and the last assertion has been established independently by Render [9].

Remark This is an extended abstract of the conference contribution [12].

References

[1] J. Dassow, S. Turaev, Petri Net Controlled Grammars: the Power of Labeling and Final Markings. *Romanian Journal of Information Science and Technology* **12** (2009) 2, 191–207.

[1]Note, however, that *valence grammars with target sets* over finite monoids are known to generate all matrix languages[2].

[2] H. FERNAU, R. STIEBE, Valence grammars with target sets. In: *Words, Semigroups, and Transductions*. World Scientific, Singapore, 2001.

[3] H. FERNAU, R. STIEBE, Sequential grammars and automata with valences. *Theoretical Computer Science* **276** (2002), 377–405.

[4] S. A. GREIBACH, Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science* **7** (1978) 3, 311 – 324.

[5] M. ITO, C. MARTÍN-VIDE, V. MITRANA, Group weighted finite transducers. *Acta Informatica* **38** (2001), 117–129.

[6] M. KAMBITES, Formal Languages and Groups as Memory. *Communications in Algebra* **37** (2009), 193–208.

[7] V. MITRANA, R. STIEBE, Extended finite automata over groups. *Discrete Applied Mathematics* **108** (2001) 3, 287–300.

[8] G. PĂUN, A new generative device: Valence grammars. *Revue Roumaine de Mathématiques Pures et Appliquées* **25** (1980), 911–924.

[9] E. RENDER, *Rational Monoid and Semigroup Automata*. Ph.D. thesis, University of Manchester, 2010.

[10] E. RENDER, M. KAMBITES, Rational subsets of polycyclic monoids and valence automata. *Information and Computation* **207** (2009) 11, 1329–1339.

[11] E. RENDER, M. KAMBITES, Semigroup automata with rational initial and terminal sets. *Theoretical Computer Science* **411** (2010) 7-9, 1004–1012.

[12] G. ZETZSCHE, On the Capabilities of Grammars, Automata, and Transducers Controlled by Monoids. In: *Automata, Languages and Programming 38th International Colloquium, ICALP 2011, Zürich, Switzerland, July 4-8, 2011, Proceedings, Part II*. LNCS 6756, Springer, 2011, 222–233.

Autorenverzeichnis