Journal of Automata, Languages and Combinatorics **u** (v) w, x–y
© Otto-von-Guericke-Universität Magdeburg

On the complexity of membership and counting in height-deterministic pushdown automata

Nutan Limaye
The Institute of Mathematical Sciences,
Chennai 600 113, India.
e-mail: nutan@imsc.res.in

Meena Mahajan
The Institute of Mathematical Sciences,
Chennai 600 113, India.
e-mail: meena@imsc.res.in

and

Antoine Meyer
LIAFA, Université Paris Diderot – Paris 7,
Case 7014, 75205 Paris Cedex 13, France.
e-mail: ameyer@univ-paris-diderot.fr

ABSTRACT

Visibly pushdown languages properly generalise regular languages and are properly contained in deterministic context-free languages. The complexity of their membership problem is equivalent to that of regular languages. However, the corresponding counting problem – computing the number of accepting paths in a visibly pushdown automaton – could be harder than counting paths in a non-deterministic finite automaton: it is only known to be in LogDCFL.

We investigate the membership and counting problems for generalisations of visibly pushdown automata, defined using the notion of height-determinism. We show that, when the stack-height of a given pushdown automaton can be computed using a finite transducer, both problems have the same complexity as for visibly pushdown languages. We also show that when allowing pushdown transducers instead of finite-state ones, both problems become LogDCFL-complete; this uses the fact that pushdown transducers are sufficient to compute the stack heights of all real-time height-deterministic pushdown automata, and yields a candidate arithmetization of LogDCFL that is no harder than LogDCFL (our main result).

Keywords: pushdown automata, complexity, membership, counting

1. Introduction

There is a close connection between complexity classes and formal language theory. Over the years, various language classes have been studied from the perspective of complexity theory. Characterising the complexity of the membership problem has been the main goal in this approach. The study of language classes and their complexity under meaningful closures was first started by Sudborough [21, 20]. In [21], he showed that the *nondeterministic linear context-free languages* or LIN are complete for the complexity class NL (nondeterministic log-space). In [20], he defined two interesting complexity classes, namely LogCFL and $\mathsf{LogDCFL}$, as the log-space closures of context-free languages (CFL) and their deterministic counterparts (DCFL) respectively. Ibarra, Jiang, and Ravikumar [14] further studied subclasses of CFL such as $\mathsf{DLIN}_{LL(1)}$ (the deterministic counterpart of LIN defined by $LL(1)$ linear grammars), *Dyck2* (well-matched strings over two types of parentheses), and *bracketed expressions* and showed that they are contained in the circuit complexity class NC^1 comprising of polynomial size, logarithmic depth, bounded fan-in AND-OR circuits. Holzer and Lange [13] showed that deterministic linear context-free languages (DLIN), as defined via $LR(1)$ linear grammars, are equivalent to those accepted by deterministic 1-turn pushdown automata ($\mathsf{DPDA}_{\text{1-turn}}$), which are deterministic pushdown automata that never push after a pop move. They showed that deciding membership in a DLIN language is complete for L, in contrast to the result of [14]. Barrington made an important contribution to the above study [3], showing that the class of regular languages, REG, is complete for NC^1. (As the classes get smaller, completeness is not understood via L-reductions but via appropriate weaker notions such as AC^0-many-one-reductions.) See [15] for an overview of these results.

Visibly pushdown automata (VPA) are real-time pushdown automata whose stack behaviour is dictated solely by the input letter being read. They are essentially equivalent to input-driven PDA, IDA. Concretely, the input alphabet Σ of a VPA or IDA P can be partitioned into three distinct sets Σ_c, Σ_r, and Σ_i of *call*, *return*, and *internal symbols*. When reading a symbol in one of these sets, P respectively has to push a symbol on its stack, pop its topmost stack symbol, and perform a transition without accessing or modifying the stack. Hence for any such partition of Σ and any VPA or IDA, the stack height at any point along a run is determined by the sequence of symbols read so far. The complexity of the membership problem for IDA was considered in [18, 5, 12]. In [12] it is shown that languages accepted by such PDA are in NC^1. In [16, 17] it is observed that membership in VPA reduces to membership in IDA. A rigorous language-theoretic study of VPA was done in [1], where it is shown that they can be determinised. Thus they lie properly between REG and DCFL, and their membership problem is complete for NC^1.

A related line of study is understanding the power of counting. Counting complexity classes essentially describe sets of functions from words to integers which are computable by a given formalism. For instance, it is easy to see from the proof of [21] that the problem of computing the number of parse trees associated with a given word in a linear grammar, $\#\mathsf{LIN}$, is equivalent to that of counting accepting paths in an NL

machine. At the lower end, however, though Barrington's result showed that deciding membership in REG (and hence in the language of a nondeterministic finite-state automaton or NFA) is equivalent to NC^1, counting accepting paths in an NFA ($\#\mathsf{NFA}$) is not yet known to be equivalent to arithmetic NC^1, $\#\mathsf{NC}^1$ (which consists of functions computable by $O(\log n)$ depth, polynomial sized (NC^1-like) circuits where gates are interpreted as integer PLUS and MULTIPLY functions (instead of Boolean AND and OR)). In [11], a one-way containment is shown: $\#\mathsf{NFA} \subseteq \#\mathsf{NC}^1$, but to this day the converse reduction remains open[1]. A natural question to ask is what generalisation of NFA can capture $\#\mathsf{NC}^1$ in this setting. In [16], it was claimed that the generalisation to VPA adds no power, $\#\mathsf{VPA}$ is equivalent to $\#\mathsf{NFA}$. However, this claim was later retracted in [17], where it is shown, however, that $\#\mathsf{VPA}$ functions can be computed in $\mathsf{LogDCFL}$ (and are hence presumably weaker than $\#\mathsf{PDA}$ functions).

Our starting point in this note is a careful examination of what makes membership and path-counting easier in VPA than in general PDA. The intention is to identify a largest possible class of PDA for which the technique used for VPA can still be applied. This technique exploits the fact that, despite nondeterminism, all paths on a given input word have the same stack-profile, and furthermore, this profile is very easily computable. One can view the partitioning of the input alphabet as *height-advice* being provided to an algorithm for deciding membership. This naturally leads to the conjecture that PDA possessing easy-to-compute height-advice functions should be easier than general PDA. The *real-time height-deterministic* PDA defined by Nowotka and Srba ([19]), rhPDA, are a natural candidate: they are defined as precisely those PDA that possess height-advice functions. They also very naturally generalise a subclass of the *synchronised* PDA defined by Caucal ([7]), namely the subclass where the synchronisation function is the stack-height, and, as in general synchronised PDA, is computable by a finite-state transducer. (A recent result in [9] implies that this subclass in fact coincides with the entire class of synchronised PDA.) We provide a parameterised definition of rhPDA that captures this generalisation: the complexity of the transducer computing the height-advice function is the parameter. We then examine the complexity of membership and path-counting at either end of the parameterisation.

A related model equivalent to VPA is that of nested word automata (NWA), defined in [2] with a view to applications in software verification and XML document processing. Motley word automata (MWA) are defined in [4] as a generalisation of NWA. Our techniques can be used to show that the equivalence is indeed very strong: deciding membership and counting accepting paths in NWA and MWA are NC^1-equivalent to the same problems over VPA.

In the next section, we give some basic notations and definitions. In Section 3, we give a brief overview of Dymond's membership algorithm [12] for VPL, which yields the NC^1 upper bound. In Section 4, the membership problem and the counting problem for rhPDA are studied. In Section 5 we discuss the membership and the counting problem for NWA and MWA.

[1]In [11] a weaker converse is shown: every $\#\mathsf{NC}^1$ function can be expressed as the difference of two $\#\mathsf{NFA}$ functions.

2. Preliminary definitions

2.1. Pushdown automata and variants

A *pushdown automaton* (PDA) over Σ is a tuple $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$ where Q is a finite set of control states, $q_0 \in Q$ the initial state, $F \subseteq Q$ a set of accepting states, Γ a finite alphabet of stack symbols, Σ a finite alphabet of labels and δ a finite set of transition rules $pU \xrightarrow{a} qV$ with $p, q \in Q$, $U, V \in \Gamma^*$, and $a \in \Sigma \cup \{\epsilon\}$. There is in general no restriction on U and V in a transition rule. However, when $|UV| \leq 1$ in every rule, the automaton is usually called *weak*. Weak PDA have the same expressive power as general PDA. From here on, and for simplicity, we will only be considering weak PDA, unless otherwise stated. If no transition rule is labelled by ϵ, then the PDA is said to be *real-time*. A PDA is called *deterministic* (DPDA) if for every pair of rules $pU \xrightarrow{a} qV$, $pU \xrightarrow{b} q'V'$, whenever $a = b$ then $qV = q'V'$, and whenever $a = \epsilon$, then also $b = \epsilon$.

A *configuration* of P is a word of the form pW with $p \in Q$ and $W \in \Gamma^*$, where p is the current control state and W is the current stack content read from top to bottom. The stack height at configuration pW is $|W|$.

The semantics of P is defined with respect to its transition graph $G_P = \{pUW \xrightarrow{a} qVW \mid a \in \Sigma \cup \{\epsilon\}, pU \xrightarrow{a} qV \in \delta,\ W \in \Gamma^*\}$. Note that G_P is an infinite graph, but it is "finitely represented" by the finite set of transition rules. A *run* of P on input word $w \in \Sigma^*$ from configuration pW is a path in G_P between vertex pW and some vertex qW', written $pW \xrightarrow{w} qW'$. Such a run is successful (or accepting) if $pW = q_0$ and q belongs to the set F of accepting states of P. By $L(G_P, S, T)$ where S, T are sets of vertices of G_P, we mean the set of all words $w \in \Sigma^*$ such that $c \xrightarrow{w} c'$ for some configurations $c \in S$, $c' \in T$. The *language* of P is the set of all words w over which there exists an accepting run; i.e. it is the language $L(G_P, \{q_0\}, F\Gamma^*)$. (Deterministic) context-free languages, (D)CFL, are the languages accepted by PDA (DPDA).

An *input-driven* pushdown automaton (IDA) over Σ is a PDA P where Σ is partitioned as $\Sigma_c \cup \Sigma_r \cup \Sigma_i$ (for *call*, *return* and *internal*), and δ satisfies the following: for every rule $pU \xrightarrow{a} qV$ in δ, $|U| = 1$ and

- if $a \in \Sigma_c$, then $|V| = 2$.
- if $a \in \Sigma_i$, then $|V| = 1$.
- if $a \in \Sigma_r$, then $|V| = 0$.

Note that the PDA so defined are not weak; however, it is straightforward to define equivalent weak PDA satisfying

- If $a \in \Sigma_c$, then any rule $pU \xrightarrow{a} qV$ has $U = \epsilon$, $V \in \Gamma$.
- If $a \in \Sigma_i$, then any rule $pU \xrightarrow{a} qV$ has $U = V = \epsilon$.
- If $a \in \Sigma_r$, then any rule $pU \xrightarrow{a} qV$ has $U \in \Gamma$, $V = \epsilon$.

If additional transitions of the form $p\bot \xrightarrow{a} q\bot$ with $a \in \Sigma_r$ are allowed, where $\bot$ is a special marker indicating that the stack is empty, one obtains the class of *visibly* pushdown automata (VPA). Both formalisms are essentially equivalent except that

VPA are allowed to read return symbols even when the stack is empty. Both IDA and VPA can be nondeterministic, i.e. δ need not be a function.

VPA are strictly more expressive than IDA: for example, no IDA can accept the language L^* where $L = \{a^n b^{n+1} \mid n \geq 1\}$, but the VPA with p as its single initial and accepting state and the following set of rules can:

$$p \xrightarrow{a} qX \qquad q \xrightarrow{a} qX \qquad qX \xrightarrow{b} r \qquad rX \xrightarrow{b} r \qquad r\bot \xrightarrow{b} p$$

However, they are equivalent to IDA when run on *well-matched strings*, i.e. strings $w \in (\Sigma_c \cup \Sigma_i \cup \Sigma_r)^*$ such that every prefix w has at most as many letters in Σ_r as it has in Σ_c. Indeed, on such strings the automaton never encounters an empty stack while reading a pop letter. Membership testing for visibly pushdown languages can be easily reduced to that for input-driven languages, as observed in [16, 17]: for every VPA M, there is another VPA M' on a slightly enlarged alphabet, and for every input w to M there is an input $g(w)$ to M', such that M' never makes an empty-stack pop move on $g(w)$. (Thus $g(w)$ is a well-matched input.) In fact, M' is an IDA. Furthermore, M' has as many accepting paths on $g(w)$ as M has on w, and the function g is computable in NC^1 (defined below).

Since we are concerned only with classes at least as large as NC^1, the difference between VPA and IDA is irrelevant in this work. For this reason, in the rest of this paper we make no distinction between both notions, and implicitly consider only VPA acting on well-matched strings.

2.2. Circuits and complexity classes

We describe some circuit classes that are relevant to our study. A circuit is a directed acyclic graph with internal nodes labeled by operators, typically AND and OR, leaves labelled by literals x_i or $\bar{x_i}$, and a designated output node. It accepts an input $x = x_1 \dots x_n$ if, when the values of x are fed in at the leaves, the output node evaluates to 1. A family of circuits $\{C_n \mid n \geq 1\}$ accepts a language $L \subset \{0,1\}^*$ if for each $x \in \Sigma^*$, $x \in L$ if and only if $C_{|x|}$ accepts x. To extend this notion to languages over larger alphabets, we allow leaves to be labelled by (0-1 valued) predicates of the form $[i, a]$ for $i \in [n]$, $a \in \Sigma$. Such a predicate evaluates to 1 exactly when the ith letter of the input word, x_i, equals a. The family has polynomial size if for some fixed polynomial p, each C_n has at most $p(n)$ nodes. It has logarithmic depth if the longest path from a leaf to the output node in C_n has length $O(\log n)$. It is ($\mathcal{C}$-)uniform if there is an algorithm (running in $\mathcal{C}$) that on input 1^n, produces a description of the circuit C_n. We only consider uniform circuits in this paper.

By NC^1, we denote the class of languages L accepted by families of circuits of polynomial size, logarithmic depth, in which every node has bounded fan-in and internal nodes are labelled AND and OR. AC^0 is a subclass of NC^1, consisting of languages accepted by polynomial size constant depth circuits over AND and OR, where each node is allowed unbounded fan-in. TC^0 is another subclass of NC^1 consisting of polynomial size constant depth circuits where each gate is allowed unbounded fan-in, and gates compute MAJORITY and NOT. It is known that AC^0 is properly contained in TC^0, and that NC^1 is contained in L.

By $\#\mathsf{NC}^1$, we denote the class of functions from Σ^* to $\mathbb{N}$ computable by polynomial-size logarithmic depth circuits where the internal nodes are labelled PLUS and MULTIPLY. Also, for any class $\mathcal{A}$ of automata, we define the associated counting complexity class $\#\mathcal{A}$ as follows:

$$\#\mathcal{A} \;= \{f : \Sigma^* \to \mathbb{N} \mid \text{ for some } M \in \mathcal{A},\, f(x) = \#\mathsf{acc}_M(x) \text{ for all } x \in \Sigma^*\}$$

Here, $\#\mathsf{acc}_M(x)$ denotes the number of accepting paths in M on input $x \in \Sigma^*$.

Proposition 1 ([11, 17]) *The following inclusions hold:*

$$\begin{tikzcd}
 & \#\mathsf{NC}^1 \arrow[r] & \mathsf{L} \arrow[r] & \mathsf{LogDCFL} \\
\#\mathsf{NFA} \arrow[ur] \arrow[rr] & & \#\mathsf{VPA} \arrow[ur] &
\end{tikzcd}$$

Here, the containment involving both a function class $\mathcal{F}$ and a language class $\mathcal{C}$, $\mathcal{F} \subseteq \mathcal{C}$, is interpreted as: for every $f \in \mathcal{F}$, the language $L_f = \{\langle x, i, b\rangle \mid \text{the } i\text{th bit of } f(x) \text{ is } b\}$ is in $\mathcal{C}$. Also, $\mathsf{LogDCFL}$ is the class of languages logspace many-one reducible to some deterministic context-free language, or equivalently the class of languages accepted by DPDA possessing an auxiliary logspace read-write work tape (called $\mathsf{DAuxPDA}$) and running in polynomial time.

3. Revisiting the proof of $\mathsf{VPL} \subseteq \mathsf{NC}^1$

In [12], Dymond proved that the membership problem for VPL is in NC^1. Dymond's proof transforms the problem of recognition/membership to efficiently evaluating an expression whose values are binary relations over a finite set and whose operations are functional compositions and certain unary operations depending on the inputs. This transformation is done in NC^1. Containment in NC^1 follows from the result, due to Buss [6], that the evaluation of formulae involving k-ary operators over a finite domain is in NC^1.

For a VPA P, on any input w, the stack height after processing i letters, $h(i, w)$ (or simply $h(w)$ if $i = |w|$), is the same across any run. Let $w_{[i,j]}$ denote a well-matched substring $w(i) \dots w(j)$ of w for $1 \le i < j \le |w|$. Let $w_{[i]}$ denote $w_{[i,i]}$. We define a set of binary relations $\Rightarrow_P^{i,j}$ (or simply $\Rightarrow^{i,j}$ if P is clear from the context) on surface configurations $(q, \gamma) \in Q \times \Gamma$ (state and stack-top pair), as

$$(q, \gamma) \Rightarrow^{i,j} (q', \gamma') \iff q\gamma \xrightarrow[G_P]{w_{[i,j]}} q\gamma'.$$

Note that even though i, j are integer indices of this relation, the domain for any relation indexed by i, j is finite (i.e. $(Q \times \Gamma)^2$). These relations are expected to capture all the cases where surface configurations are reachable from one another without accessing the previous stack profiles. A pair (i, j) of indices is called *height-matched* if the string $w_{i+1} \dots w_j$ is well-matched. The relations $\Rightarrow^{i,j}$ are defined only for height-matched (i, j).

The relations $\Rightarrow^{i,j}$ where $j \in \{i, i+1\}$ can be obtained directly from w and δ. Given two relations $\Rightarrow^{i,j}$ and $\Rightarrow^{j,k}$ for height matched (i,j) and (j,k), the relation $\Rightarrow^{i,k}$ can be computed from these using composition as follows:

$$\Rightarrow^{i,j} \circ \Rightarrow^{j,k} = \left\{ (q,\gamma,q',\gamma') \;\middle|\; \begin{array}{c} \exists q'' \in Q, \gamma'' \in \Gamma : (q,\gamma) \Rightarrow^{i,j} (q'',\gamma''), \\ (q'',\gamma'') \Rightarrow^{j,k} (q',\gamma') \end{array} \right\}$$

Given the relation $\Rightarrow^{i,j}$, if $w_{[i-1]}$ is a call letter a and $w_{[j+1]}$ is the matching return letter b, then the relation $\Rightarrow^{i-1,j+1}$ can be computed as $Ext_{\{a,b\}}(\Rightarrow^{i,j})$ using the unary extension operation Ext defined as follows:

$$Ext_{\{a,b\}}(\Rightarrow^{i,j}) = \left\{ (q,\gamma,q',\gamma') \;\middle|\; \begin{array}{c} \exists q_1, q_2 \in Q, \gamma_1, \gamma_2 \in \Gamma : (q_1,\gamma_1) \Rightarrow^{i,j} (q_2,\gamma_2), \\ q \xrightarrow{a} q_1\gamma_1 \in \delta, \; q_2\gamma_2 \xrightarrow{b} q' \in \delta \end{array} \right\}$$

Given $\Rightarrow^{i,j}$, if $w_{[j+1]}$ is an internal letter c, then $\Rightarrow^{i,j+1} = Ext_{\{c\}}(\Rightarrow^{i,j})$ where:

$$Ext_{\{c\}}(\Rightarrow^{i,j}) = \left\{ (q,\gamma,q',\gamma') \,\middle|\, \exists q_1 \in Q : \; (q,\gamma) \Rightarrow^{i,j} (q_1,\gamma') \text{ and } q_1 \xrightarrow{c} q' \in \delta \right\}$$

Let $\Rightarrow^{Id}$ denote the identity relation, *i.e.* $(q,\gamma) \Rightarrow^{Id} (q',\gamma')$ if and only if $q = q'$ and $\gamma = \gamma'$.

Given a string w, the goal is to compute $\Rightarrow^{1,|w|}$. The main work is to figure out the correct indices for the relations and then the appropriate operations. But that can be accomplished essentially by computing stack heights for various configurations, which is easy for VPA.

Example 2 *Consider a* VPA *with $\Sigma_c = \{a\}, \Sigma_r = \{b\}, \Sigma_i = \{c\}$ on word $w = aabaabcbb$. The relation $\Rightarrow^{1,|w|}$ can be computed using $Ext_{\{a,b\}}, Ext_{\{c\}}$ and $\circ$ as follows:*

$$\Rightarrow^{1,|w|} = Ext_{\{a,b\}}(Ext_{\{a,b\}}(\Rightarrow^{Id}) \circ Ext_{\{a,b\}}(Ext_{\{a,b\}}(\Rightarrow^{Id}) \circ Ext_c(\Rightarrow^{Id}))).$$

As pointed out in [12], the above transformation works for potentially larger classes.

Remark 3 ([12]) *Dymond's* NC^1 *membership algorithm works for any pushdown automaton P satisfying the following three conditions.*

- *P should be real-time (i.e. have no ϵ-rules).*
- *Accepting runs should end with an empty stack (and in a final state).*
- *There should exist an* NC^1*-computable function h such that for $w \in \Sigma^*$ and $0 \le i \le |w|$, $h(i,w)$ is the height of the stack after processing the first i symbols of w. If P is non-deterministic, then $h(i,w)$ should be consistent with some run ρ of P on w; further, if P accepts w, then ρ should be an accepting run.*

Clearly, VPA satisfy these conditions. By definition, they have no ϵ-rules. Though they may not end with an empty stack, this can be achieved by appropriate padding that is computable in TC^0, see for instance [16, 17]. Though VPA may be nondeterministic, all runs have the same height profile, and the function $h(i, w)$ can in fact be computed in TC^0.

Since any computation up to NC^1 can be allowed for Dymond's proof to apply, VPL do not fully exploit Dymond's argument. We explore a generalisation of VPA allowing us to define more general classes for which Dymond's scheme (or its precursor from [5]) may work for deciding membership, and then examine the power of counting in these models.

4. More general height functions: height-determinism

As already mentioned, adding a pushdown stack to an NFA significantly increases the complexity of both membership and path-counting. However, if stack operations are restricted to an input-driven discipline, as in VPA, then membership is no harder than for NFA, and path-counting seems easier (in FLogDCFL, the class of functions computable by polynomial time DAuxPDA) than over general PDA. What is being exploited is that, despite nondeterminism, all paths on a given input word have the same stack-profile, and this profile is computable in NC^1 (and even in TC^0). One can view the partitioning of the input alphabet as *height-advice* being provided to an algorithm for deciding membership. This naturally leads to the question: what can be deduced from the existence of such height-advice, independent of how this function is computed?

The term *height-determinism*, coined by [19], captures precisely this idea. A PDA is height-deterministic if the stack height reached after any partial run depends only on the input word w which has been read so far, and not on non-deterministic choices performed by the automaton. Consequently, in any real-time height-deterministic pushdown automaton (rhPDA), all runs on a given input word have the same stack profile. Another way to put it is that for any rhPDA P, there exists a *height-advice function* h from Σ^* to $\mathbb{Z}$, such that $h(w)$ is the stack-height reached by P on any run over w.

Any rhPDA that accepts on an empty stack and whose height-advice function h is computable in NC^1 directly satisfies the conditions in Remark 3, and hence its membership problem lies in NC^1. In this section, we explore some subclasses of rhPDA and discuss the complexity of their membership and counting problems.

4.1. Definition and properties

Let us first give a formal definition of rhPDA.

Definition 4 (rhPDA, [19]) *A real-time (weak) pushdown automaton*[2] $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$ *is called* height-deterministic *if it is complete (for every input word, there is a run that reads the entire input), and* $\forall w \in \Sigma^*$, $q_0 \xrightarrow{w} q\alpha$ *and* $q_0 \xrightarrow{w} p\beta$ *imply* $|\alpha| = |\beta|$.

[2] In [19], the definition involves rules of the form $pX \xrightarrow{a} q\alpha$ where $\alpha \in \{\epsilon, X\} \cup \{YX | Y \in \Gamma\}$. This is not an essential requirement for the results presented here.

Note that the requirement that an rhPDA be complete can be interpreted in more than one way. As a syntactic requirement, the PDA is complete if for every node in G_P, and every letter $a \in \Sigma$, there is an outgoing edge labelled a. A (weaker) semantic requirement would be that this condition is met only on nodes reachable from the initial node. A more subtle (and also weaker) semantic requirement would be that for every word $w \in \Sigma^*$, there is a path $q_0 \xrightarrow{w} qW'$ in G_P for some $q \in Q$, $W' \in \Gamma^*$. For a DPDA, there is at most one computation path on any input. Thus, by introducing a dead state one can complete any weakly complete DPDA to fullfill the strong completeness condition. Hence, unless stated otherwise, we will always be using the stronger notion of syntactic completeness for DPDA.

The robustness of the notion of height-determinism is illustrated by the fact that rhPDA retain most good properties of VPA, even when the actual nature of the height-advice function is left unspecified. This had already been obtained in [7] for a slightly different class (which the authors of [19] admittedly used as a starting point in the elaboration of their paper).

Proposition 5 ([19, 7]) *Any* rhPDA *can be determinised. Consequently, for a fixed h, the class of languages accepted by* rhPDA *and whose height-advice function is h forms a boolean algebra (and properly includes regular languages). Moreover, language equivalence between two* rhPDA *with the same height-advice function is decidable.*

All these results are effective as soon as h is computable. Note that h being computable is a sufficient condition. However, it is not necessary: in fact one can do the product constructions for the union and intersections and other constructions even without knowing the the function h at all, as long as the two automata synchronize.

Since any deterministic real-time PDA is also height-deterministic, another consequence of the fact that rhPDA can be determinised is that the whole class rhPDA accepts precisely the class of real-time DCFL.

Something slightly stronger than determinisation is shown in [19] and will turn out to be useful for us.

Proposition 6 ([19]) *For every* rhPDA *A with initial state p_0, there is a language-equivalent real-time, complete* DPDA *B with initial state q_0 such that if w labels a path $p_0 \xrightarrow{w} pW$ in G_A and a path $q_0 \xrightarrow{w} qY$ in G_B, then $|W| = |Y|$.*

4.2. Instances of height-deterministic PDA

The definition of a rhPDA leaves the exact nature of the height-advice function h unspecified. This is troublesome, since h could be arbitrarily complex. We consider some classes of specific height-advice functions, the simplest non-trivial one being VPA.

Following the framework developed in [7], we consider classes $\mathcal{T}$ of transducers mapping words to integers. A *transducer* T over Σ and $\mathbb{Z}$ is a transition system $(C, c_0, F, (\Sigma \times \mathbb{Z}), \delta)$, where c_0 denotes the initial configuration and F a set of final configurations, and whose transitions described by δ are of the form $c \xrightarrow{a/k} d$ with

$c, d \in C$, $a \in \Sigma$ and $k \in \mathbb{Z}$. In such a rule, a is considered as an input and k as an output. A run $c_0 \xrightarrow{a_1/k_1} c_1 \ldots c_{n-1} \xrightarrow{a_n/k_n} c_n$ is associated with the pair $(w, k) = (a_1 \ldots a_n, k_1 + \ldots + k_n)$. Such a transducer defines a relation $g_T \subseteq \Sigma^* \times \mathbb{Z}$ defined as the set of all pairs (w, k) labelling an accepting run in T.

In our setting, we only consider both input-complete and input-deterministic transducers (i.e. transducers whose underlying Σ-labelled transition system is deterministic and complete), in which all configurations are final (in which case we omit F in the definition). Consequently, for any such transducer T the relation g_T is actually a function, and is defined over the whole set Σ^*. The transition graph G_P of a PDA P is said to be *compatible* with a transducer T if for every vertex s of G_P, if $u, v \in L(G_P, \{q_0\}, \{s\})$ then $g_T(u) = g_T(v)$. If T is compatible with P, then potentially T can be used to "synchronise" P: we can define a function g from the reachable nodes of G_P to integers, where $g(pW)$ is exactly the function $g_T(u)$ for some string $u \in L(G_P, \{q_0\}, \{pW\})$, and compatibility ensures that such a g is well-defined. In particular, if the function g so defined satisfies $g(pW) = |W|$, for all reachable nodes pW, we can say that T synchronises P via stack-height. In what follows, we are concerned with only stack-height synchronisation.

One may consider several kinds of transducers. The simplest class is *finite-state transducers* (FST), where the configuration space C is simply a finite set of control states (often written Q). One may also consider *pushdown transducers* (PDT) whose underlying Σ-labelled transition system is a PDA transition graph, or even more complex transducers (for instance defined using Turing machines).

Definition 7 *For any class $\mathcal{T}$ of complete deterministic transducers,* rhPDA($\mathcal{T}$) *is the class of* rhPDA *whose height function h can be computed by a transducer T in $\mathcal{T}$, in the sense that $h(w) = |g_T(w)|$ (absolute value of $g_T(w)$) for all w.*

Note that the height-advice function of any VPA running on well-matched strings can be computed by a single-state transducer, that reads letters and outputs $+1$ or -1 or 0 depending on whether the letter is in Σ_c or Σ_r or Σ_i. However, note that such single-state transducers can also compute stack-heights for languages that are provably not in VPL, as the following example shows.

Example 8 *The language $EQ(a, b) = \{w \mid |w|_a = |w|_b\}$ is not accepted by any* VPA *for any partition of $\{a, b\}$. But it is in* rhPDA(T) *for a single-state transducer T. Consider a finite transducer $T = (Q, q, \Sigma \times \mathbb{Z}, \delta_T)$ where $Q = \{q\}$, $\Sigma = \{a, b\}$, $\delta_T = \{q \xrightarrow{a/+1} q, q \xrightarrow{b/-1} q\}$, and a* PDA *$P = (Q_P, q_0, F, \Gamma, \Sigma, \delta)$ where $Q_P = \{q_a, q_b, q_0\}$, $F = \{q_0\}$, $\Gamma = \{X, A, B\}$, and δ is as follows:*

$$\begin{array}{llll} q_0 \xrightarrow{a} q_a X & q_a \xrightarrow{a} q_a A & q_a A \xrightarrow{b} q_a & q_a X \xrightarrow{b} q_0 \\ q_0 \xrightarrow{b} q_b X & q_b \xrightarrow{b} q_b B & q_b B \xrightarrow{a} q_b & q_b X \xrightarrow{a} q_0 \end{array}$$

Then T correctly computes the stack-heights of P.

Also, allowing more than one state in a FST provably enlarges the class of languages.

Example 9 *$REV = \{wcw^R \mid w \in \{a,b\}^*\}$ is not a* VPL*. The obvious* DPDA *accepting this has a height function computable by a two-state transducer, with loops labelled $a/+1$ and $b/+1$ on the first state, loops labelled $a/-1$ and $b/-1$ in the second, and a transition labelled $c/0$ from the first state to the second one. It is easy to see that for any* PDA *accepting REV, two states in the transducer are necessary for computing the stack height.*

Note that the PDA in Example 8 is complete in the weak sense. It, as well as the PDA in Example 9, can be completed by adding a dead state. The stack moves in the dead state should be consistent with the transducer. This consistency may require adding more than one dead state.

Further, in [19] there is a separating example in rhPDA but not in rhPDA (FST).

Example 10 *Consider the set of strings*

$$\left\{ a^m b^n w \;\middle|\; \begin{array}{l} m > n > 0,\ |w|_a = |w|_b \text{ and if } w \neq \epsilon, \\ \text{then the first letter of } w \text{ is } a \end{array} \right\}$$

This language can be accepted by an rhPDA*, but no* rhPDA(FST) *accepts it. The intuitive reason is that, for the* FST *T in Example 8 to synchronize with a* PDA *accepting words w such that $|w|_a = |w|_b$, it specifically relies on the fact that stack-height is computed* by absolute value*. However, this is only possible if the pushdown stack is initially empty when beginning to read w, which is not the case in the present example, where reading the prefix $a^m b^n$ may bring the stack to an arbitrary height for any* PDA *accepting this language. For a formal proof of this result, we refer the reader to the (unpublished) appendix of [19].*

From Proposition 11 below, it follows that this language is in fact in rhPDA (PDT).

In the remainder of this section, we will focus on the classes rhPDA(FST) and rhPDA(PDT), and also to some extent on the class rhPDA($\mathsf{rDPDA}_{\text{1-turn}}$), where the transducer is a 1-turn PDT. A 1-turn PDA or PDT is one that satisfies the following property: on any run on any word, once a rule of the form $qA \xrightarrow{a} p$ is used (that is, once the PDA pops an element from the stack), then all subsequent rules are of the form $q\gamma \xrightarrow{a} p$ (the PDA no longer pushes elements onto the stack).

The class we define as rhPDA(FST) is a subclass of the synchronised pushdown automata considered in [7] and later generalised in [10, 8, 9] using the formalism of graph grammars. In fact, this class coincides with the class of languages synchronized by *linear deterministic graph grammars*, or equivalently *linear regular graphs* (see [8] for more details on this equivalence).

Finally, we note that since, by definition, rhPDA are complete, it is in fact unnecessary to consider more complex transducers than deterministic and complete PDT. Formally:

Proposition 11 *For any* rhPDA *P whose height-advice function is h, there exists a deterministic and complete pushdown transducer T such that $h(w) = g_T(w)$ for all $w \in \Sigma^*$. That is, every* rhPDA *is in* rhPDA(PDT)*.*

Proof. Let $P = (Q, q_0, F, \Sigma, \Gamma, \delta)$. If P is syntactically complete, we can proceed as follows: define $P' = (Q, q_0, F, \Sigma, \Gamma, \delta')$ in which δ' is a subset of δ containing only the lexicographically first transitions for every nondeterministic transition defined in δ. This automaton is deterministic, and since rhPDA are complete, it is also complete. It has its own height-advice function h. But since the automaton P is height-deterministic, all runs of P, and in particular the lex-first run, have the same stack height. This implies that P and P' admit the same height-advice function h.

If P satisfies the weaker completeness requirement, we appeal to Proposition 6 and use the DPDA obtained there as P'.

It is now straightforward to define a deterministic and complete pushdown transducer T whose underlying pushdown automaton is P', and such that $g_T(w) = h(w)$ for any input word w (for this, each transition of T simply has to output the integer matching the stack movement performed by this transition. Note that it is dependent on the transitions locally and not on the entire run). By definition of P' and T and since T is complete, the height-advice function of P is well-defined and correctly computed by T. □

We now turn to studying the complexities of the membership and counting problems over rhPDA(FST) and rhPDA(PDT).

4.3. Complexity of the membership problem

As we already mentioned, rhPDA have exactly the same power as real-time DPDA in terms of accepted languages. Thus the membership question for the whole class rhPDA (and thus also for rhPDA(PDT)) is in LogDCFL.

It turns out that this is in fact a completeness result. It was shown by Sudborough [20] that the following language is a hardest DCFL and is complete for the class LogDCFL.

Definition 12 ([20]) *Let u be a string over the alphabet $\{(_1, (_2,)_1,)_2, [,], \#\}$ of the form $x_0[w_1\#z_1][w_2\#z_2]\ldots[w_k\#z_k]$ for some $k \in \mathbb{N}$ where $x_0 \in \{(_1, (_2\}^*$ and for all i such that $1 \le i \le k$, $w_i \in \{)_1\} \cdot \{(_1, (_2\}^*$ and $z_i \in \{)_2\} \cdot \{(_1, (_2\}^*$.*

A string u of this form is said to be in the language $DetCh(Dyck_2)$ if and only if for each $1 \le i \le k$, $\exists x_i \in \{w_i, z_i\}$ such that $x_0x_1\ldots x_k \in Dyck_2$. That is, there is a (deterministic) way to choose one of the two substrings w_i, z_i for each i such that all the chosen substrings put together in the correct order along with x_0 form a balanced string of parentheses over two types of parentheses.

The language $DetCh(Dyck_2)$ is deterministic context-free and is complete for the class LogDCFL*.*

A real-time DPDA P accepting $DetCh(Dyck_2)$ starts reading the string u and on x_0 simply pushes the string on the stack. The invariant it maintains is: the stack contains unmatched opening parentheses. After having processed $i-1$ blocks, suppose P has a type 1 parenthesis on the top of the stack. Then it chooses x_i to be w_i, pops the stack-top, and pushes all but the first letter of w_i on the stack. Otherwise, it chooses x_i to be z_i and w_i is read symbol by symbol and ignored by P. On reading z_i

the stack-top is popped and all but the first letter of z_i are pushed on the stack. The letters $[,], \#$ are treated as markers and appropriate state changes are performed over them. If finally the stack becomes empty, the string is accepted, else it is rejected. Thus this language can be accepted by a real-time DPDA; and hence membership testing for rhPDA is hard for $\mathsf{LogDCFL}$.

This settles the complexity of the membership question for the whole class rhPDA (and thus also for $\mathsf{rhPDA(PDT)}$); we have

Proposition 13 *The membership question for the class* rhPDA *(and thus also for* $\mathsf{rhPDA(PDT)}$*) is complete for* $\mathsf{LogDCFL}$ *under logspace many-one reductions.*

We observe easy bounds on the complexity of the height-advice function.

Lemma 14 *For a complete deterministic transducer T computing function g_T,*

1. *If T is a* FST*, then g_T is computable in* NC^1*.*
2. *If T is a real-time* $\mathsf{DPDA}_{1\text{-}turn}$*, or* $\mathsf{rDPDA}_{1\text{-}turn}$*, then g_T is computable in* L*.*
3. *If T is a* PDT*, then g_T is computable in* $\mathsf{LogDCFL}$*.*

Proof.
1. Let $q_0(a_1, k_1)q_1 \ldots q_{n-1}(a_n, k_n)q_n$ be the run of transducer T on input $w = (a_1 \ldots a_n)$. In NC^1, we can construct the run and hence the sequence $k_1, k_2, \ldots, k_n$. Now a TC^0 circuit can compute, for each i, the sum $s_i = \sum_{j=1}^{i} k_j$. Since the transducer value is s_i if $s_i \geq 0$ and $-s_i$ otherwise, overall, the function $T(w)$ is computable in NC^1.
2. It is known that $\mathsf{DPDA}_{\text{1-turn}}$ can be simulated in logspace ([13]). Thus if a function is computed by a $\mathsf{rDPDA}_{\text{1-turn}}$ transducer, a logspace machine can keep track of its output, and hence g_T is in L.
3. Given input x and an index $1 \leq i \leq |x|$, a $\mathsf{DAuxPDA}$ uses its stack for simulating the stack of T and the auxiliary work-tape to maintain a counter which sums all successive integers output by T. The $\mathsf{DAuxPDA}$ needs no more than linear time, and a logarithmic size counter suffices.

□

This allows us to apply Dymond's algorithm for $\mathsf{rhPDA(FST)}$.

Lemma 15 *For any fixed* $\mathsf{rhPDA(FST)}$*, the membership problem is in* NC^1*.*

Proof. This can be seen as easy corollary of Remark 3. The following simple reduction brings the given $\mathsf{rhPDA(FST)}$ to an appropriate form for Remark 3 to be applicable. Let P be a PDA in the class $\mathsf{rhPDA(FST)}$, and T a finite-state transducer computing the height-advice function of P. Given a string $w = a_1 \ldots a_n$, the membership problem asks whether $w \in L(P)$. Recall that P cannot have any ϵ-moves as it is real-time. And by Lemma 14, we can determine in NC^1 the height of the stack on any prefix of the input. To meet the second condition, we determine in NC^1 if the final stack height is some $k \neq 0$. In this case, we convert the string $w \in \Sigma^*$ to a string $wX^k \in (\Sigma \cup \{X\})^*$, where X is a new letter. We extend the PDA to a new PDA P' that has all the moves of P, and further, it reads X and pops the stack, without changing the acceptance status of the control state. Then $w \in L(P) \Leftrightarrow wX^k \in L(P')$, and P' satisfies all the conditions of Remark 3. □

This membership algorithm exploits Dymond's construction better than VPA, as the height function requires a possibly NC^1-complete computation (predicting states of the transducer). Recall that for VPA, the height function is computable in TC^0, a subclass of NC^1.

In [5], the membership problem for VPL is shown to be in L. We observe that their algorithm can be more explicitly implemented as a log-space machine with access to an oracle that supplies the stack height of the VPA after seeing a given word. That is, the log-space machine makes height queries to the height-function g of the VPL; we denote the class of such machines as L^g. In this form, it can be generalised to any rhPDA having height function g, as stated in Theorem 16 below. The proof follows from Lemmas 18 and 19, and the result, along with Lemma 14, yields the next corollary since $\mathsf{L}^{\mathsf{L}} = \mathsf{L}$.

Theorem 16 *For any fixed* rhPDA P *with height function* g*, the membership problem is in* L^g.

Corollary 17 *The membership problem for* $\mathsf{rhPDA}(\mathsf{rDPDA}_{\mathsf{1\text{-}turn}})$ *is in* L.

The class $\mathsf{rhPDA}(\mathsf{rDPDA}_{\mathsf{1\text{-}turn}})$ referred to here contains all languages accepted by real-time $\mathsf{DPDA}_{\mathsf{1\text{-}turn}}$ as well as languages accepted by $\mathsf{rhPDA}(\mathsf{FST})$. It is contained in DCFL.

Let P be an rhPDA. A string w is said to be *well-matched* with respect to P, if the stack height before and after processing w is the same, say h, and while processing w, it never becomes less than h. (The above definition of well-matched strings cannot be extended to a PDA that is not height-deterministic, because on the same string, the stack heights on different runs may be different.)

Lemma 19 uses the algorithm from [5] to establish the L^{g_T} bound for well-matched inputs, and Lemma 18 brings the input in that form.

Lemma 18 *For every* $\mathsf{rhPDA}(T)$ P *over an alphabet* Σ*, there is a corresponding* $\mathsf{rhPDA}(T')$ P' *over an alphabet* Σ' *and a* $\mathsf{L}^{g_{T'}}$ *many-one reduction* f *such that for every* $x \in \Sigma^*$, $\#\mathsf{acc}_P(x) = \#\mathsf{acc}_{P'}(f(x))$*, and* $f(x)$ *is well-matched.*

Proof. The $\mathsf{rhPDA}(T')$ P' is essentially the same as P. It has two new input symbols A, B, and a new stack symbol X. On seeing an A, X is pushed, and on B X is expected and popped. P' has a new state q' that is the only initial state. P' expects an input from $A^*\Sigma^*B^*$. On the prefix of A's it pushes X's. When it sees the first letter from Σ, it starts behaving like P. The only exception is when P performs a pop move on $\perp$, P' performs the same move on X. On the trailing suffix of B's it pops X's. It is straightforward to design δ' from δ. The transducer T' exactly mimics the transducer T, except when it sees an A it outputs $+1$ and on B it outputs -1 on any configuration.

Let $|x| = n$. The logspace machine with oracle access to the height computing function $g_{T'}$ does the following: It queries the oracle for the height of the string $A^n x$, say d. It then outputs $y = A^n x B^d$. By the way P' is constructed, it should be clear that $\#\mathsf{acc}_P(x) = \#\mathsf{acc}_{P'}(y)$ and that P', on y, never pops on an empty stack. In fact y is well-matched. □

Lemma 19 (Variant of algorithm 2 of [5]) *Let T be a transducer, and let P be a* $\mathsf{rhPDA}(T)$ *accepting well-matched strings. Given an input string x, checking membership of x in $L(P)$ can be done in* L^{g_T}.

Proof. In [5], a logspace algorithm for membership testing when P is a VPA is given. In [17], this algorithm is modified and a recursive procedure is described to obtain a $\mathsf{LogDCFL}$ bound for counting accepting paths in a VPA. We essentially use the recursive procedure from [17]. However, if it is used as is, the bound obtained will be $\mathsf{LogDCFL}^{g_T}$ for height function g_T, so some further modifications are needed.

We describe some details and all the changes made to the proof of [17] for the sake of completeness. Let the PDA be $P = (Q, \Sigma, Q_{in}, \Gamma, \delta, Q_F)$. Let $x_{ij} = x_{i+1}..x_j$ be a well-matched substring of the string x. (Define $x_{ii} = \epsilon$, the empty string.) Define a $\big((|Q| \times |\Gamma|) \times (|Q| \times |\Gamma|)\big)$ matrix over $\{0, 1\}$, where each row and column is indexed by a state-stack-top pair (surface configuration). The entry indexed by $[(q, X), (q', X')]$ is 1 if and only if $X = X'$ and P has a run from surface configuration (q, X) to (q', X') on the input string x_{ij}. We will call such a matrix the table T_{ij} corresponding to the string x_{ij}. P has an accepting run on x if and only if the entry $[(q_0, \perp), (q, \perp)]$ is 1 for some $q \in Q_F$ in the table corresponding to x_{0n}. Thus, it is sufficient to compute this table. However, in order to do so, we may have to compute many/all such tables.

We say that an interval $r = [i, j]$ is *valid* if $i \leq j$ and x_r, the string represented by the interval, is well-matched; otherwise it is said to be *invalid.* A *fragment* is a pair (r, Λ) where Λ is a pair (r', T'), r and r' are valid intervals, T' is a table. The fragments that arise in the algorithm satisfy the properties: (1) the interval r' is nested inside the interval r, and (2) T' is the table corresponding to the string $x_{r'}$, that is, $T' = T_{r'}$. For $r = (i, j)$, $\Lambda = (r', T')$ is *trivial* if $r' = [l, l]$ where $l = \lceil (i+2j)/3 \rceil$ (and hence $x_{r'} = \epsilon$), and T' is the identity table Id. The recursive procedure $\mathcal{T}$ takes a fragment (r, Λ) as an input and computes the table T_r, assuming that $T' = T'_{r'}$ where r' is a valid interval nested inside r. The main call made to the procedure is $([0, n], \Lambda)$ with trivial Λ.

The procedure $\mathcal{T}$ does the following: If the size of $r - r'$ is at most 2, then it computes the table T_r immediately from δ and T'. If the size of $r - r'$ is more than 2, then it breaks r into three valid intervals $r_1, r_2, r - (r_1 \cup r_2)$, where (1) the size of each of $r_1, r_2, r - (r_1 \cup r_2)$ is small (in two stages, each subinterval generated will be at most three-fourth the size of $r - r'$), (2) one of r_1, r_2 completely contains r', (3) r_1, r_2 are contiguous with r_1 preceding r_2. It then creates fragments (r_1, Λ_1) and (r_2, Λ_2) where $\Lambda_1 = \Lambda$ and Λ_2 is trivial if r_1 contains r', and $\Lambda_2 = \Lambda$ and Λ_1 is trivial if r_2 contains r'. Now it evaluates these fragments recursively to obtain the tables $\mathcal{T}(r_1, \Lambda_1) = T_{r_1}$, $\mathcal{T}(r_2, \Lambda_2) = T_{r_2}$, and obtains the table $T_{r_3} = T_{r_1} \times T_{r_2}$, where $r_3 = r_1 \cup r_2$ and the $\times$ represents Boolean matrix product. Setting $\Lambda_3 = (r_3, T_{r_3})$, it finally makes the recursive call $\mathcal{T}(r, \Lambda_3)$ to compute T_r. In [5], it is shown that such fragments can always be defined and can be found deterministically and uniquely. (We will discuss the complexity of finding such fragments shortly.) It is also shown that the tables computed by the above recursion procedure have the following property: for the table T corresponding to the interval $r = [i, j]$, the $[(q, X), (q', X')]$-th entry is 1 exactly when the machine has at least 1 path from (q, X) to (q', X') on string x_{ij}. This proof is by induction on the length of the intervals.

Note that the above procedure yields a $O(\log n)$ depth recursion tree (see [17], [5] for detailed proofs), with each internal node having three children corresponding to the three recursive calls made. The leaves of this recursion tree are disjoint effective intervals (for fragment $(r, (r', T'))$, the effective interval is $r - r'$). As the main call is made to the fragment $([0, n], \Lambda)$ with trivial Λ, the size of such a tree will be $O(n)$. The traversal of such a tree can be performed by a Turing machine that uses $O(\log n)$ space and makes queries to the oracle machine providing height-advice for machine P as follows: At any point the current fragment being evaluated is remembered in $O(\log n)$ space. (At the beginning the interval $[0, n]$ is remembered). Let the three recursive calls made for a given fragment (r, Λ) be called LEFT, RIGHT, and OTHER ((r_1, Λ_1), (r_2, Λ_2), (r, Λ_3)) respectively. Let the label of the node be the string $w \in \{\text{LEFT}, \text{RIGHT}, \text{OTHER}\}^*$ that indicates its position in the recursion tree.

Claim 20 *Given the label (the path from the root) of any node in the recursion tree, the intervals corresponding to the fragment associated with the node can be computed in* L^{g_T}.

The input to the logspace oracle machine querying the height function of the transducer, g_T, is the string x and a label $w = w_1 w_2 \dots w_k$ where k is $O(\log |x|)$ and w is a string over the alphabet $\{\text{LEFT}, \text{RIGHT}, \text{OTHER}\}$. The machine is expected to compute the indices of the interval corresponding to the label and the gap indices for that interval. The following algorithm will compute these indices. Let $trivial([i, j])$ for $0 \leq i \leq j \leq n$ be defined as the trivial interval corresponding to $[i, j]$, that is $[\lceil (i+2j)/3 \rceil, \lceil (i+2j)/3 \rceil]$. The work-tape of the machine is initialised with the triple $(0, r, r')$ where $r = [0, n]$ and $r' = trivial(r)$.

A prefix of the label corresponds to a node in the recursion tree. After having read $w_1 w_2 \dots w_m$, suppose the the node in the tree corresponding to this prefix is an interval $[i_m, j_m]$ with a gap $[i'_m, j'_m]$. Then the invariant maintained after having read $w_1 w_2 \dots w_m$ is that the worktape contains (m, r, r') where $r = [i_m, j_m]$ and $r' = [i'_m, j'_m]$.

Supposing the work-tape has a correct triple $(m-1, r, r')$ after having read $m-1$ bits of the label, we now describe how to compute the next triple (m, s, s') upon reading w_m. For the current pair (r, r'), steps to compute the intervals r_1, r_2, r_3 will be discussed shortly. Supposing one can do this in L^{g_T}, now the letter w_m is read and the pair (s, s') is obtained as follows:

w_m	LEFT		RIGHT		OTHER
s	r_1		r_2		r
s'	r'	if $r' \subseteq r_1$	r'	if $r' \subseteq r_2$	$r_3 = r_1 \cup r_2$
	$trivial(r_1)$	otherwise	$trivial(r_2)$	otherwise	

The modifications continue as long as $|r - r'| > 2$ after which r and r' can be thought of as left unchanged. (In the algorithm, this case will not arise.)

We now describe how to compute r_1, r_2, r_3 given r and r'. Let $r = [i, j]$ and $r' = [i', j']$ be such that $i \le i' \le j' \le j$ and $(r - r') > 2$. Consider the larger of the two subintervals $[i, i']$ and $[j', j]$. Break it into two equal size parts. Consider the part closer to r'. In this, find an index t such that the height of the stack of P just after reading x_t (denoted as $h(t)$) is the lowest in that part. Now find two more points b, a such that $b \le t \le a$, $h(b) = h(t) = h(a)$, and the interval $[b, a]$ is the maximal valid subinterval containing t and within $[i, j]$. (Note: b, t, a need not be all distinct.) Let $r_1 = [b, t]$, $r_2 = [t, a]$. Once r_1, r_2 are fixed, the three fragments can be found as described above. Thus, finding the three fragments essentially boils down to finding b, t, a. These values a, b, t can be found by the base logspace machine by querying the height computing function.

Example 21 *Consider the word $w = aabaabcbb$ from the* VPL *of Example 2. The recursion tree has depth 2. For simplicity, we write (i, j, i', j') to refer to a node with intervals $r = [i, j]$ and $r' = [i', j']$. The root of the tree is $(0, 9, 6, 6)$ and has $(b, t, a) = (1, 3, 8)$, thus its three children are $(1, 3, 3, 3)$, $(3, 8, 6, 6)$, $(0, 9, 1, 8)$. The middle child here needs further expansion: it has $(b, t, a) = (4, 4, 7)$, and thus its three children are $(4, 6, 6, 6)$, $(6, 7, 7, 7)$, $(3, 8, 4, 7)$.*

Claim 22 *The tree can be traversed in L^{g_T}.*

From Claim 20, we know that once the label is available the interval itself can be computed in L^g. We now see how the remaining computations can be performed by a logspace base machine.

The depth of the tree is $O(\log n)$. Thus, the size of any label is $O(\log n)$. Depending on which child is going to be evaluated, the label is updated by suffixing it with the appropriate letter LEFT, RIGHT, or OTHER. Also if the step results in computing a table, that table is stored along with the previously added suffix. At any stage, the number of tables to be remembered is also at most the maximum depth of the tree. Each table is of size $O(1)$. Thus the overall space required is $O(\log n)$. To move along the recursion tree, the intervals need to be computed; this can be done in L^{g_T} by the previous claim. Thus a full traversal can be done in L^{g_T}. □

4.4. Complexity of the counting problem

The aspect of rhPDA which interests us in this study is that it is a nondeterministic model capturing the deterministic class LogDCFL. It thus provides a way of arithmetising LogDCFL, simply by counting the number of accepting paths on each word in a rhPDA. We call the class of such functions #rhPDA. In particular, we consider the classes #rhPDA(FST) and #rhPDA(PDT).

We have seen that although rhPDA(FST) properly generalises VPA (Example 8), the membership problem has the same complexity as that over VPA (Lemma 15). It turns out that even the path-counting problem has the same complexity.

Theorem 23 #rhPDA(FST) $\equiv$ #VPA *(via* NC^1 *many-one reductions).*

Proof. VPA are contained in the class rhPDA(FST), so we only need to show that computing #rhPDA(FST) functions reduces to computing #VPA functions. This is easy to observe, since the functions computed by an FST can be computed in NC^1.

Let P be an rhPDA with height-advice computed by FST T. A naive approach would be to construct a single PDA P' that simulates (P, T) by running PDA P along with transducer T. However, such a PDA P' will not necessarily be a VPA. Now consider the string rewritten using an enriched alphabet which consists of the input letter along with a tag indicating whether P should push or pop. On this enriched alphabet, if the tags are correct, then a PDA that simulates the original PDA P (i.e. ignores the tags) behaves like a VPA. But by Lemma 14, the correct tags for any word can be computed in NC^1.

Formally, let $P = (Q^P, q_0^P, F, \Gamma, \Sigma, \delta^P)$ and $T = (Q^T, q_0^T, \Sigma, \delta^T)$. We construct a VPA M as follows: $M = (Q^P, q_0^P, F, \Gamma, \Sigma', \delta)$, where $\Sigma' = \Sigma \times \{c, r, i\}$, the partition of Σ' is defined as: $\Sigma'_x = \Sigma \times \{x\}$ for $x \in \{c, r, i\}$, and δ is the same as δ^P (i.e. it ignores the second component of the expanded alphabet).

Given input $w = a_1 \dots a_n$, consider the string $w' = \langle a_1, t_1 \rangle \dots \langle a_n, t_n \rangle$, where $t_j \in \{c, r, i\}$, and $t_j = c, r, i$ depending on whether $|g_T(a_1 \dots a_j)| - |g_T(a_1 \dots\ a_{j-1})| = 1$ or -1 or 0. By Lemma 14, we can produce the string w' in NC^1.

It is straightforward to see that accepting runs of M on w' are in one-to-one correspondence with accepting runs of P on w. □

Theorem 24 shows that membership and counting for rhPDA have the same complexity, a situation rather unusual for nondeterministic complexity classes.

Theorem 24 #rhPDA *is in* LogDCFL.

The proof proceeds in several stages. To compute a #rhPDA function f on input x, we first compute $f(x)$ modulo several small (logarithmic) primes, and then reconstruct $f(x)$ from these residues. This is the standard Chinese remainder technique (see for instance [22]), and we use it as stated formally below.

Lemma 25 (folklore) *Let P be a fixed* rhPDA. *There is a constant $c \geq 0$, depending only on P, such that given input x, the number of accepting paths of P on input x can be computed in logarithmic space with oracle access to the language L_{res} defined below. (Here p_i denotes the ith prime number.)*

$L_{res} = \{ \langle x, i, j, b \rangle | 1 \leq i \leq |x|^c,$ *the jth bit of* $\#\mathsf{acc}_P(x) \bmod p_i$ *is* b $\}$

We now show that L_{res} can be computed by a polynomial time $\mathsf{DAuxPDA}$ machine making oracle queries to the height-advice function g_T. This follows from the technique of [5] as used in [17] to show that $\#\mathsf{VPA}$ functions are in $\mathsf{LogDCFL}$.

Lemma 26 *If P is any* rhPDA *and T a* PDT *computing its height-advice function, then L_{res} is in* $\mathsf{LogDCFL}^{g_T}$*.*

Proof. First, we note that by the Prime Number Theorem, all the primes required in L_{res} can be represented in $O(\log |x|)$ bits, and hence can be obtained in logarithmic space. So we can assume that the primes are explicitly available.

In [17], it is shown that if the fixed rhPDA P is in fact a VPA, then the language L_{res} is in $\mathsf{LogDCFL}$ without any oracle. In the proof of Lemma 19, we modified this proof from [17] for giving neat parameterisation for membership problems of subclasses of rhPDA. Here, we will modify it similarly to obtain the required bound on the language L_{res} corresponding to a rhPDA.

Essentially the same recursion procedure from [5] which was described in the proof of Lemma 19 is used here. Here we only describe the required changes.

The semantics of the matrices T_{ij} is slightly different here: a number k is stored at the entry $((q, X), (q', X))$ if there are $k \pmod{p_i}$ paths in the VPA M starting in configuration (q, X) and ending in configuration (q', X) having read the well-matched string x_{ij}.

A similar recursion tree is obtained. Claim 20 remains unchanged, however the matrices stored along any root to leaf path on tree are now of size $O(\log n)$. Claim 22 thus changes to:

Claim 27 *The tree can be traversed in* $\mathsf{LogDCFL}^{g_T}$*.*

Since along any root to leaf path there can be $O(\log n)$ such tables to be remembered during traversal, a logspace transducer does not suffice as the base machine. The $\mathsf{DAuxPDA}$ base machine uses its stack to store the tables while performing the traversal of the tree. Everything else goes through as in proof of Claim 22. □

Lemmas 14 and 26 together imply that L_{res} is in $\mathsf{LogDCFL}(\mathsf{LogDCFL})$. This is not adequate for us, since it is not known whether $\mathsf{LogDCFL}(\mathsf{LogDCFL}) \subseteq \mathsf{LogDCFL}$. (Relativising a space-bounded class is always tricky. Here, we have a pushdown class with auxiliary space, making the relativisation even more sensitive.) However, we further note that the $\mathsf{LogDCFL}^{g_T}$ machine accepting L_{res} makes oracle queries which all have short representations: each query can be written in logarithmic space. (Strictly speaking, the input x is also part of the query. But for eliminating the oracle, this plays no role.) In such a case, we can establish a better bound, which may be of independent interest:

Lemma 28 *Let $L(M^A)$ be the language accepted by a poly-time* $\mathsf{DAuxPDA}$ *M which makes $O(\log n)$-bits oracle queries to a language $A \in$* $\mathsf{LogDCFL}$*. Then $L(M^A) \in$* $\mathsf{LogDCFL}$*.*

Proof. Consider a $\mathsf{DAuxPDA}$ M' that has three auxiliary tapes s_1, s_2, s_3 of size $O(\log n)$ bits each. The machine starts simulating M using s_1. When M is computing query bits, it notes down the query bits on tape s_2. When the query is computed, it is on the tape s_2 of M'. At this stage, M' marks the stack with a special stack marker to indicate that the simulation of the oracle machine is going to begin. It then starts simulating the machine for A, say M'', using the tape s_3 as a work tape and tape s_2 as the input tape. Once the simulation of M'' is completed, the answer to the query is available on s_3. Machine M' now pops the stack till the special marker is popped. At this stage it has all the information needed to resume the computation of M. $\square$

Combining these lemmas proves Theorem 24, since $\mathsf{L(LogDCFL)}$ equals $\mathsf{LogDCFL}$.

5. Related models: nested and motley words

In [2], Alur and Madhusudan defined nested word automata (NWA) as an equivalent model for VPA, motivated by applications in software verification and XML document processing. In [4], Blass and Gurevich defined motley word automata (MWA) as a generalisation of NWA. The definitions of models of NWA and MWA are orthogonal to the notion of height-determinism. However, we observe that their complexity bounds are the same as that of VPL for both membership and counting problems.

We begin with definitions of NWA and MWA.

A *nested relation* ν of width n, for $n \geq 0$, is a binary relation over $[1, n]$ such that (1) if $\nu(i, j)$ then $i < j$; (2) if $\nu(i, j)$ and $\nu(i', j')$ then either $\{i, j\} = \{i', j'\}$ or $\{i, j\} \cap \{i', j'\} = \emptyset$, and (3) if $\nu(i, j)$ and $\nu(i', j')$ and $i < i'$ then either $j < i'$ or $j' < j$.

If ν is a nested relation with $\nu(i, j)$, then i is the *call-predecessor* of j and j is the *return-successor* of i. The definition requires that each position has at most one call-predecessor or at most one return-successor but not both.

A nested word over an alphabet Σ is a pair (w, ν) such that $w \in \Sigma^*$, and ν is a nested relation of width $|w|$. A position $k \in [1, |w|]$ of w is a *call position* if $(k, j) \in \nu$ for some j, a *return position* if $(i, k) \in \nu$ for some i, and an *internal position* otherwise.

Definition 29 (NWA) *A nested word automaton (*NWA*) A over an alphabet Σ is a tuple $(Q, q_0, F, \Sigma, \delta)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states , $\delta = \langle \delta_c, \delta_i, \delta_r \rangle$ is a set of transitions such that $\delta_c \subseteq Q \times \Sigma \times Q$, $\delta_i \subseteq Q \times \Sigma \times Q$ and $\delta_r \subseteq Q \times Q \times \Sigma \times Q$ are the transitions for call, internal, and return positions respectively.*

A starts in state q_0 and reads the word left to right. At a call or internal position, the next state is determined by the current state and input symbol, while at a return position, the next state can also depend on the state just before the matching call-predecessor. A run ρ of the automaton A over a nested word $nw = (a_1 \dots a_n, \nu)$ is a sequence $q_0, \dots, q_n$ over Q such that for each $1 \leq j \leq n$,

- if j is a call position, then $(q_{j-1}, a_j, q_j) \in \delta_c$
- if j is an internal position, then $(q_{j-1}, a_j, q_j) \in \delta_i$
- if j is a return position with call-predecessor k, then $(q_{j-1}, q_{k-1}, a_j, q_j) \in \delta_r$.

A accepts the nested word nw if $q_n \in F$. The language $L(A)$ of a nested-word automaton A is the set of nested words it accepts.

A *motley word* mw of dimension d over Σ is a tuple $(w, \nu_1, \ldots, \nu_d)$, where $w \in \Sigma^*$ and $\nu_1, \ldots, \nu_d$ are nested relations of width $|w|$.

Definition 30 (MWA) *A motley word automaton (*MWA*) A of dimension d is a direct product $A_1 \times \ldots \times A_d$ of d* NWA *$A_1, \ldots, A_d$.*[3]

A *run* of A on dimension d motley word $mw = (w, \nu_1, \ldots, \nu_d)$ with $|w| = n$ is a sequence $(q_0^1, \ldots, q_0^d), \ldots, (q_n^1, \ldots, q_n^d)$ of states of A such that every $(q_0^k, \ldots, q_n^k)$ is a run of A_k on the nested word (w, ν_k). A run of A on mw is *accepting* (or mw is accepted by A) if each of the d constituent runs is. $L(A)$ is defined as usual.

The languages of nested/motley words accepted by NWA or MWA are called *regular* nested/motley languages. Regular motley languages strictly generalise regular nested languages [4], since for some $i \neq j$, the same position can be a call-position for ν_i and a return position for ν_j.

It is shown in [2] (Theorem 6) that for a fixed NWA, the membership question is in NC^1. The analogous question for a fixed MWA is easily seen to have the same complexity, since it involves answering membership questions for d different, but fixed, NWA, where d is the dimension of the MWA. Thus

Proposition 31 *The membership problem for any regular motley languages is in* NC^1.

In both models, NWA and MWA, non-determinism is allowed in the definition. We show that path-counting in NWA and MWA is equivalent to that in VPA. This does not follow from the equivalence of membership testing; rather, it requires that the equivalence be demonstrated by a parsimonious reduction.

Lemma 32 *For each* NWA *A, there is a* VPA *M, and for each nested word nw over Σ there is a word w over Σ^M, such that $\#\mathsf{acc}_A(nw) = \#\mathsf{acc}_M(w)$. Further, w can be constructed from nw by an* AC^0 *circuit, for a suitable encoding of nw.*

Proof. Let $A = (Q, q_0, F, \Sigma, \delta^A)$ be the NWA. We first describe the construction of the VPA. We let $M = (Q, q_0, F, \Gamma, \Sigma^M, \delta^M)$, where $\Gamma = Q$, $\Sigma^M = \{c, r, i\} \times \Sigma$, and δ^M is defined as follows: for $(q, a, q') \in \delta_c^A$, δ^M has the rule $q \xrightarrow{(c,a)} q'q$; for $(q, a, q') \in \delta_i^A$, δ^M has the rule $q \xrightarrow{(i,a)} q'$; and for $(q, q', a, q'') \in \delta_r^A$, δ^M has the rule $qq' \xrightarrow{(r,a)} q''$.

Now we describe the encoding of the nested word $nw = (a, \nu)$. This is a string over $\Sigma \cup \{0, 1, \#, (,)\}$ where $\{0, 1, \#, (,)\}$ is assumed to be disjoint from Σ. Each pair (i, j) in ν is encoded as an opening bracket, followed by a bit string describing i in binary, followed by a $\#$, then a bit string describing j in binary, and then a closing bracket. The nested word nw is presented by giving a, followed by a list of the encodings of all pairs in ν.

[3]As NWA are in general non-deterministic, so are motley automata. A MWA $A_1 \times \ldots \times A_d$ is deterministic if every nested word automata A_k is so.

The AC^0 reduction works as follows:

Given a $nw = (a_1 \dots a_n, \nu)$ over Σ, encode it as a word w over Σ^M as follows: $w = (x_1, a_1) \dots (x_n, a_n)$ where $x_j = c$ if j is a call position, $x_j = r$, if j is a return position and $x_j = i$ if j is an internal position. An AC^0 circuit will determine if j is a call position, by comparing j with the strings appearing between (and # in the input (first co-ordinate of the ν relation). From the definition of the nesting relation, a valid encoding of a nested word will have exactly one such occurrence of j if j is indeed a call position. Similarly, an AC^0 circuit can determine whether a position j is a return position (comparing strings between # and)) or an internal position (checking if there is no occurrence of j anywhere in the encoding of ν).

It is straightforward to see that this reduction is parsimonious (path preserving). That is, the number of accepting paths of A on $nw = (a, \nu)$ is exactly the same as the number of accepting paths of M on w. □

Lemma 33 $\#\mathsf{MWA} \equiv \#\mathsf{NWA} \equiv \#\mathsf{VPA}$*, (via* TC^0 *reductions).*

Proof. $\#\mathsf{NWA} \leq \#\mathsf{VPA}$: This follows directly from Lemma 32.

$\#\mathsf{VPA} \leq \#\mathsf{NWA}$: We first describe how to come up with a NWA A from a VPA M. We set $Q' = Q \times \Gamma$, $q'_0 = (q_0, \perp)$, $F' = \{(q, \perp) \mid q \in F\}$ and δ' can be described as follows: If for $a \in \Sigma_c$, $q \xrightarrow{a} (p, A) \in \delta$ then for all $B \in \Gamma$, $((q, B), a, (p, A)) \in \delta'_c$. If for $a \in \Sigma_i$, $q \xrightarrow{a} p \in \delta$ then for all $B \in \Gamma$, $((q, B), a, (p, B)) \in \delta'_i$. If for $a \in \Sigma_r$, $(q, A) \xrightarrow{a} p \in \delta$ then for all $B \in \Gamma$, for all $p' \in Q$ $((q, A), (p', B), a, (p, B)) \in \delta'_r$.

We now describe how to convert a word $w \in \Sigma^*$ where $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$ to a nested word $nw = (a_1 ... a_n, \nu)$. The string $a_1 ... a_n$ is exactly w. The nesting relation ν can be computed as follows: For every $1 \leq j \leq n$ such that $a_j \in \Sigma_r$, find the matching call position $i < j$ and put the pair (i, j) in ν. The positions of the internal letters do not feature in the relation ν as per the definition of ν.

It is easy to see that accepting paths of M on w and of A on nw are in one-to-one correspondence, and that nw can be obtained from w in TC^0.

$\#\mathsf{MWA} \equiv \#\mathsf{NWA}$: By definition, $\#\mathsf{MWA}$ contains $\#\mathsf{NWA}$. To see the converse reduction, we define for any MWA M an NWA N that acts on d copies of the input string. Formally, the NWA N is described as: $Q = \cup_{j=1}^{d} Q^j$; $q_0 = q_0^1$; $F = F^d$; if $d > 1$ then $\Sigma' = \Sigma \cup \{\$\}$ else $\Sigma' = \Sigma$; $\delta_c = \cup_{j=1}^{d} \delta_c^j$; $\delta_r = \cup_{j=1}^{d} \delta_r^j$; if $d > 1$ $\delta_i = \cup_{j=1}^{d} \delta_i^j \cup \{(p, \$, q) \mid \forall p \in F^{l-1}, q = q_0^l, 2 \leq l \leq d\}$, else $\delta_i = \delta_i^1$.

The nested word $nw = (b_1 ... b_{(n+1)d}, \nu)$ can be obtained from the motley word mw as follows: $b_1 ... b_{(n+1)d}$ is a string obtained by taking d copies of $a_1 ... a_n$ separated by \$. The nesting relation puts in each ν_i with an appropriate shift:
$\nu = \{((n+1)(l-1) + i, (n+1)(l-1) + j) \mid (i, j) \in \nu_l, 2 \leq l \leq d, 1 \leq i < j \leq n\}$.

Again, it is easy to see that accepting paths of M on mw and of A on nw are in one-to-one correspondence, and that nw can be obtained from mw in TC^0. □

From [2], Proposition 31 and Lemma 33 we get the following:

Theorem 34 *Deciding membership and counting accepting paths in* NWA *and* MWA *are equivalent, via* TC^0*-many-one reductions, to the corresponding problems over* VPA*.*

6. Conclusion

We have studied a range of real-time height-deterministic pushdown automata lying between visibly and real-time deterministic pushdown automata. Figures 1, 2 and 3 depict the relations between language classes, their closures under appropriate reductions, and the corresponding counting classes. (Dashed arrows denote incomparability, solid arrows denote containment, and arrows with a cut denote proper containment.)

$$\begin{tikzcd}
 & \mathsf{VPL} \arrow[r, "/" marking] \arrow[dd, dashed, leftrightarrow] & \mathsf{rhPDA(FST)} \arrow[r] \arrow[rr, bend left, "/" marking] & \mathsf{rhPDA(rDLIN)} \arrow[r] \arrow[dd, dashed, leftrightarrow] & \mathsf{rhPDA} \arrow[dr, "/" marking] & \\
\mathsf{REG} \arrow[ur, "/" marking] \arrow[dr, "/" marking] & & & & & \mathsf{DCFL} \\
 & \mathsf{rDLIN} \arrow[rr, "/" marking] & & \mathsf{DLIN} \arrow[urr, "/" marking] & &
\end{tikzcd}$$

Figure 1: Summary of language classes

$$\begin{tikzcd}
\boxed{\begin{array}{l} \mathsf{NC}^1, \mathsf{REG}, \mathsf{VPL}, \\ \mathsf{rhPDA(FST)}, \\ \mathsf{DLIN}\ _{LL(1)} \end{array}} \arrow[r] & \boxed{\mathsf{rhPDA(rDLIN)}} \arrow[r] & \boxed{\begin{array}{l} \mathsf{L}, \mathsf{DLIN} \\ _{LR(1)} \end{array}} \arrow[r] & \boxed{\begin{array}{l} \mathsf{rhPDA}, \\ \mathsf{DCFL}, \\ \mathsf{LogDCFL} \end{array}}
\end{tikzcd}$$

Figure 2: Summary of language classes closures

$$\begin{tikzcd}
\mathsf{NC}^1 \arrow[r] & \#\mathsf{NFA} \arrow[r] & \boxed{\begin{array}{l} \#\mathsf{VPL}, \\ \#\mathsf{rhPDA(FST)} \end{array}} \arrow[r] & \#\mathsf{rhPDA(rDLIN)} \arrow[r] & \#\mathsf{rhPDA} \arrow[r] & \mathsf{LogDCFL}
\end{tikzcd}$$

Figure 3: Summary of counting classes

Some open questions remain. First, it would be interesting to investigate additional classes lying between rhPDA(FST) and rhPDA(PDT). Such classes need independent study, both for their properties as language classes and to investigate their closures under appropriate reductions.

Secondly, the only known upper bound for #VPL, #rhPDA(FST) and #rhPDA(rDLIN), is LogDCFL. It would be interesting to refine this bound. We believe that #VPL should indeed be reducible to a much smaller class, either L, or maybe even #NFA. Actually proving such a bound may well yield a different membership algorithm for VPLs.

References

[1] R. Alur, P. Madhusudan, Visibly pushdown languages. In: *36th STOC*. ACM, 2004, 202–211.

[2] R. Alur, P. Madhusudan, Adding nesting structure to words. In: *10th DLT*. LNCS 4036, 2006, 1–13.

[3] D. Barrington, Bounded-Width Polynomial-Size Branching Programs Recognize Exactly Those Languages in NC^1. *Journal of Computer and System Sciences* **38** (1989) 1, 150–164.

[4] A. Blass, Y. Gurevich, *A note on nested words*. Technical Report MSR-TR-2006-139, Microsoft Research, October 2006. `http://research.microsoft.com/~gurevich/Opera/180.pdf`

[5] B. V. Braunmuhl, R. Verbeek, Input-driven languages are recognized in $\log n$ space. In: *4th FCT*. LNCS 158, 1983, 40–51.

[6] S. Buss, The Boolean Formula Value Problem Is in ALOGTIME. In: *19th STOC*. ACM, 1987, 123–131.

[7] D. Caucal, Synchronization of Pushdown Automata. In: *10th DLT*. LNCS 4036, Springer, 2006, 120–132.

[8] D. Caucal, Boolean algebras of unambiguous context-free languages. In: *28th FSTTCS*. LIPIcs 2, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008, 83–94.

[9] D. Caucal, Synchronization of regular automata. In: *34th MFCS*. LNCS 5734, Springer, 2009, 2–23.

[10] D. Caucal, S. Hassen, Synchronization of Grammars. In: *3rd CSR*. LNCS 5010, Springer, 2008, 110–121.

[11] H. Caussinus, P. McKenzie, D. Thérien, H. Vollmer, Nondeterministic NC^1 computation. *Journal of Computer and System Sciences* **57(2)** (1998), 200–212.

[12] P. Dymond, Input-driven languages are in $\log n$ depth. *Information Processing Letters* **26** (1988), 247–250.

[13] M. Holzer, K.-J. Lange, On the Complexities of Linear LL(1) and LR(1) Grammars. In: *9th FCT*. LNCS 710, 1993, 299–308.

[14] O. Ibarra, T. Jiang, B. Ravikumar, Some subclasses of context-free languages in NC^1. *Information Processing Letters* **29** (1988), 111–117.

[15] K.-J. Lange, Complexity and Structure in Formal Language Theory. In: *8th Conference on Computational Complexity*. IEEE Computer Society, 1993, 224–238.

[16] N. Limaye, M. Mahajan, B. V. R. Rao, Arithmetizing Classes arround NC^1 and L. In: *24th STACS*. LNCS 4393, 2007, 477–488.

[17] N. Limaye, M. Mahajan, B. V. R. Rao, Arithmetizing Classes arround NC^1 and L. *Theory of Computing Systems* (2009), to appear. (spl. issue for STACS 2007). See also Electronic Colloquium on Computational Complexity, ECCC TR07-87.

[18] K. Mehlhorn, Pebbling mountain ranges and its application to DCFL recognition. In: *7th ICALP*. LNCS 85, 1980, 422–432.

[19] D. Nowotka, J. Srba, Height-Deterministic Pushdown Automata. In: *MFCS*. LNCS 4708, 2007, 125–134.

[20] I. Sudborough, On the Tape Complexity of Deterministic Context-Free Languages. *Journal of the ACM* **25(3)** (1978), 405–414.

[21] I. H. Sudborough, A Note on Tape-Bounded Complexity Classes and Linear Context-Free languages. *Journal of the ACM* **22** (1975) 4, 499–500.

[22] H. Vollmer, *Introduction to Circuit Complexity: A Uniform Approach*. Springer, 1999.