Membership Testing: Removing Extra Stacks from Multi-stack Pushdown Automata

Nutan Limaye and Meena Mahajan

The Institute of Mathematical Sciences, Chennai 600 113, India.
{nutan,meena}@imsc.res.in

Abstract. We show that fixed membership testing for many interesting subclasses of multi-pushdown machines is no harder than for pushdowns with single stack. The models we consider are MVPA, OVPA and MPDA, which have all been defined and studied in the past.
Multi-stack pushdown automata, MPDA, have ordered stacks with pop access restricted to the stack-top of the first non-empty stack. The membership for MPDAs is known to be in NSPACE(n) and in P. We show that the P-time algorithm can be implemented in the complexity class LogCFL; thus membership for MPDAs is LogCFL-complete.
It follows that membership testing for ordered visibly pushdown automata OVPA is also in LogCFL.
The membership problem for multi-stack visibly pushdown automata, MVPA, is known to be NP-complete. However, many applications focus on MVPA with $O(1)$ phases. We show that for MVPA with $O(1)$ phases, membership reduces to that in MPDAs, and so is in LogCFL.

1 Introduction

Pushdown machines are the machines having a finite control and access to a stack. The languages accepted by such machines are called context-free languages, CFLs. For a fixed machine M, given a string w, the membership problem asks whether $w \in L(M)$. It is of interest due to its implications for parsing and model checking problems. The first polynomial time algorithm for the membership problem for CFLs was given by Cocke, Kasami, and Younger (see for instance [1]).

Let us denote the membership problem for the class of languages $\mathcal{L}$ by MEM($\mathcal{L}$). If $\mathcal{A}$ is a class of automata accepting the language class $\mathcal{L}$, then we use MEM($\mathcal{L}$) and MEM($\mathcal{A}$) interchangeably.

The problems log-space many-one reducible to MEM(CFL) define a complexity class called LogCFL [2]. LogCFL is a subclass of P and is also known to be contained in NC, *i.e.* efficiently parallelizable.

The membership problem for many subclasses of CFLs has been studied rigorously. The set of languages log-space many-one reducible to MEM(DCFL), where DCFL denotes deterministic context-free languages, define the complexity class LogDCFL which is a subclass of LogCFL [2]. It is also known that the membership problems for linear and deterministic linear context-free languages, MEM(LIN) and MEM(DLIN), are complete for the complexity classes nondeterministic and deterministic log-space, NL and Log respectively [3, 4]. Another interesting subclass of CFLs is visibly pushdown languages, VPLs ([5, 6]). These are the languages accepted by visibly pushdown automata (VPA) which are ϵ-moves-free pushdown automata whose stack behaviour is dictated solely by the input letter under consideration. They are also referred to as input-driven PDA. MEM(VPL) is known to be complete for the class NC^1 [7] of languages accepted by families of polynomial-size log-depth bounded fan-in circuits.

A natural generalisation of pushdown machines is pushdown machines with more than one stack. This generalisation, unfortunately, is not smooth in terms of the power of these machines: A pushdown automaton with two or more stacks is known to recognise all recursively enumerable languages. The model in its full generality is thus intractable. However, for certain model checking applications, pushdown automata with two or more stacks are useful. Hence, some restrictions of multi-stack machines have been considered in the literature.

One possible restriction is a 2-stack VPA. One can consider various models depending on whether to allow simultaneous stack changes or depending on the order of accessing the stacks. Such models indeed have been considered recently (see *e.g.* [8, 9]). A language-theoretic study, as well as the membership problem complexity for these models, are important.

Here we focus on the membership problem for three different models which have been defined and studied in the literature.

The first model is one recently considered by La Torre *et al.* [9]: a pushdown machine equipped with two stacks where the access to both the stacks is completely dictated by the input alphabet. This is a natural generalisation of VPLs and a proper restriction of general pushdown automaton having more than one stack. They call such machines multi-stack visibly pushdown machines, MVPA. In their definition, these machines cannot simultaneously access both stacks. On reading any input letter, the MVPA either pushes on one of the stacks or pops from one of the stacks. A *phase* of the input string is a substring such that while reading it, all the pop moves of the machine are on the same stack. In [9], it is shown that MEM(MVPL), where MVPL denotes the class of languages accepted by MVPA, is NP-complete. The proof of NP hardness is a reduction from an instance of SAT. For a fixed MVPA M, a string w is constructed from an n-variable formula such that it has n phases. That the number of phases depends on the input formula is important for the proof of hardness.

In this paper, we consider a restriction of the above problem, where the number of phases is a constant. We define another version of the membership problem, $\mathsf{MEM}(\mathsf{MVPL}_k)$. For a fixed MVPA M and fixed positive integer k, the problem $\mathsf{MEM}(\mathsf{MVPL}_k)$ is to decide whether a given $w \in \Sigma^*$ is in $L_k(M)$, where $L_k(M)$ denotes the language $\{w \in \Sigma^* \mid w \text{ is accepted by } M \text{ with } \leq k \text{ phases}\}$.

This restriction of MVPA, where the number of phases is bounded, is also useful for many applications and has been defined and considered in [9]. The class is known to generalise VPLs and is properly contained in context-sensitive languages. In this paper, we show that the problem $\mathsf{MEM}(\mathsf{MVPL}_k)$ is in LogCFL.

In order to show this, we need another model of multi-pushdown machines defined by Cherubini *et al.* [10]. They define a restriction of multi-pushdown machines wherein there is an order given to the stacks of the machine. The machine is allowed to push on any stack. However, pop moves are allowed only on the first non-empty stack. We denote such machines by PD_n, where n is the number of stacks in the machine. We denote the class of languages accepted by these as L_{PD_n}. A restriction of PD_n, namely PD_2, was studied in [11], where it was shown that $\mathsf{MEM}(\mathsf{PD}_2)$ is in P. Later, in [12], a P-time upper bound for $\mathsf{MEM}(\mathsf{PD}_n)$ was established. We give a reduction from $\mathsf{MEM}(\mathsf{MVPL}_k)$ to $\mathsf{MEM}(\mathsf{PD}_k)$.

We then prove a LogCFL upper bound for $\mathsf{MEM}(\mathsf{PD}_k)$. This improves the P-time upper bound of [12]. Also, combined with our previous reduction, this gives a LogCFL upper bound for the problem $\mathsf{MEM}(\mathsf{MVPL}_k)$. The languages accepted by MVPA within two phases are a proper subclass of context sensitive languages, a proper generalisation of VPLs, and are incomparable with CFLs. Hence, this implies the same upper bound as for CFLs for an incomparable class. However, we do not know if $\mathsf{MEM}(\mathsf{MVPA}_k)$ for any fixed k is hard for LogCFL.

Recently, Carotenuto *et al.* [8] defined another class of two-stack pushdown machines, 2-OVPA. Like MVPAs, these machines have a visible access to their stacks, *i.e.* the stack movement is completely dictated by the input alphabet. There is also an order among the stacks and the second stack is popped only if the first is empty. This model is interesting because emptiness and inclusion are decidable, and languages accepted by such machines form a Boolean algebra [8]. The generalisation where the number of stacks is k, k-OVPA, is also considered in [8]. The language class accepted is contained in L_{PD_k}. Thus, the LogCFL upper bound we prove also applies to this language class.

The main results of our paper can be summarised as follows:

Theorem 1. *For every fixed* $k \geq 1$, $\mathsf{MEM}(\mathsf{MVPL}_k) \leq \mathsf{MEM}(\mathsf{PD}_k)$.

Theorem 2. *For every fixed* $k \geq 1$, $\mathsf{MEM}(\mathsf{PD}_k)$ *is in* LogCFL.

Corollary 1. $\forall k \geq 1$, $\mathsf{MEM}(\mathsf{MVPL}_k)$ *and* $\mathsf{MEM}(k\text{-}\mathsf{OVPA})$ *are in* LogCFL.

2 Preliminaries

Circuits and Complexity A *Boolean circuit* C_n on n inputs is a directed acyclic graph, with a designated sink (out-degree zero vertex) called the output gate. All the vertices except sources (in-degree zero vertices) are labelled by $\vee$ and $\wedge$. Sources are labelled by $\{0,1\}$ or by predicates of the form $[x_i, a, 1, 0]$ where $i \in [n]$. Such a predicate takes the value 1 if $x_i = a$ and 0 otherwise[1].

A Boolean circuit C can be unwound into a tree T_C (by duplicating nodes). A *proof tree* T' of C on input w is a subtree of T_C with the following properties:

[1] This convention of labelling leaves with predicates is used, for *e.g.* , in [13], to deal with languages over non-binary alphabets.

(1) The output gate is in T'. (2) For every $\vee$-gate in T', one of its children is in T'. (3) For every $\wedge$-gate in T', all its children are in T'. (4) All the nodes in T' evaluate to 1 on input w.

A proof tree exists if and only if C_n accepts w. In general the proof tree could be of size exponential in n. C is said to have poly-proof-tree-size if whenever a string w is accepted by C, there is a proof tree on w of size poly($|w|$).

The complexity class LogCFL is the class of languages log-space many-one reducible to some CFL, and is known to be equivalent to the class of languages accepted by circuits having polynomial sized proof trees. See *e.g.* [2, 14, 15].

Visible two stack machines ([9]). An MVPA M is a pushdown machine having two stacks, where the access to the stacks is restricted in the following way: The input alphabet Σ is partitioned into 5 sets. A letter from Σ_c^j causes a push move on stack j, that from Σ_r^j forces a pop move on stack j, and both the stacks are left unchanged on letters from Σ_i. (The subscripts c, r, i denote call, return and internal respectively.) Formally, an MVPA $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ is a two-stack nondeterministic pushdown machine where Q is a set of finite states, Σ is the finite alphabet which is a union of 5 disjoint sets $\Sigma_c^0, \Sigma_r^0, \Sigma_c^1, \Sigma_r^1, \Sigma_i$, q_0 is the initial state, $F \subseteq Q$ is a set of final states, Γ is the finite stack alphabet containing a special bottom-of-stack symbol $\perp$ that is never pushed or popped, and δ has the following structure: $\delta_i \subseteq Q \times \Sigma_i \times Q$, and for $j \in \{0, 1\}$, $\delta_c^j \subseteq Q \times \Sigma_c^j \times Q \times \Gamma \setminus \{\perp\}$ and $\delta_r^j \subseteq Q \times \Sigma_r^j \times \Gamma \times Q$.

The machine is allowed to pop on an empty stack; that is, on reading a letter from Σ_r^j and seeing $\perp$ on the jth stack top, the machine can proceed with a state change leaving the $\perp$ untouched.

A phase is a substring of the input string $w \in \Sigma^*$ during which pop moves happen only on one of the stacks. Define the set

$$\mathsf{PHASE}_k = \{w \;\mid w \in \Sigma^*,\ \text{number of phases in } w \leq k\}$$

Clearly, for any fixed partition of Σ, PHASE_k is a regular set.

Let M be a fixed MVPA M and k a fixed positive integer. Its k-phase language $L_k(M)$ is defined as $L_k(M) = L(M) \cap \mathsf{PHASE}_k$. By taking a direct product of a finite state automaton accepting PHASE_k with MVPA M, we can obtain an MVPA $M' = \langle M, k \rangle$ such that $L(M') = L_k(M') = L_k(M)$. In Section 3, we assume that the given MVPA M satisfies $L(M) = L_k(M)$.

Ordered multi-stack machines and grammars ([10–12]).

A PD_k $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a k-stack pushdown machine where Q, Σ, q_0, Z_0, F are as usual, and the transition function δ is of the form $\delta \subseteq Q \times (\Sigma \cup \epsilon) \times \Gamma \times Q \times (\Gamma^*)^k$.

A *configuration* is a $(k+2)$-tuple, $\langle q, w, \gamma_1, \ldots, \gamma_k \rangle$ where $q \in Q$, $w \in \Sigma^*$, and $\gamma_i \in \Gamma^*$ for each i represents the contents of the ith stack. The *initial* configuration on word x is $\langle q_0, x, Z_0, \epsilon, \ldots, \epsilon \rangle$. A configuration is called a *final* configuration if $q \in F$.

If there is a transition $(q', \alpha_1, \ldots, \alpha_k) \in \delta(q, a, A)$, the machine in state q can read a letter a from the input tape, pop A from the first non-empty stack, push α_i on stack i for each $i \in [k]$, and move to state q'. Formally, $\langle q, aw, \epsilon, \ldots, \epsilon, A\gamma_i, \ldots, \gamma_k \rangle \vdash \langle q', w, \alpha_1, \ldots, \alpha_{i-1}, \alpha_i\gamma_i, \ldots, \alpha_k\gamma_k \rangle$.

If $(q', \alpha_1, \ldots, \alpha_k) \in \delta(q, \epsilon, A)$ then $\langle q, w, \epsilon, \ldots, \epsilon, A\gamma_i, \ldots, \gamma_k\rangle \vdash$ $\langle q', w, \alpha_1, \ldots, \alpha_{i-1}, \alpha_i\gamma_i, \ldots, \alpha_k\gamma_k\rangle$.

The PD_k M accepts a string w if it can move from $\langle q_0, w, Z_0, \epsilon, \ldots, \epsilon\rangle$ to some $\langle q, \epsilon, \gamma_1, \ldots, \gamma_k\rangle$ where $q \in F$. The set of all the strings accepted by M is the language accepted by M, denoted $L(M)$.

Theorem 3. *([12]) For a fixed PD_k, given an input string $w \in \Sigma^*$, checking if $w \in L(M)$ is in P. i.e. $\mathsf{MEM}(\mathsf{PD}_k) \in \mathsf{P}$.*

In [11], PD_k are characterized by grammars. We describe the D^2-grammars that correspond to languages accepted by PD_2. A D^2-grammar G is a 4-tuple $G = (N, \Sigma, P, S)$ where N, Σ, S are as usual, and P has productions of the form: $A \to w(\alpha)(\beta)$ where $A \in N$, $w \in \Sigma^*$ and $\alpha, \beta \in N^*$.

Sentential forms in a derivation are of the form $x(\alpha)(\beta)$ where $x \in \Sigma^*$, $\alpha, \beta \in N^*$. The initial sentential form is $(S)(\epsilon)$. If $A \to w(\alpha)(\beta)$ is a production rule, then $w'(A\alpha')(\beta') \Rightarrow w'w(\alpha\alpha')(\beta\beta')$ and $w'(\epsilon)(A\beta') \Rightarrow w'w(\alpha)(\beta\beta')$ are the only valid derivations using this rule. Note that only *leftmost* derivations are allowed. We say that $A \Rightarrow^* w(\alpha)(\beta)$ if $(A)(\epsilon) \Rightarrow^* w(\alpha)(\beta)$ and that $A \Rightarrow^* w$ if $(A)(\epsilon) \Rightarrow^* w(\epsilon)(\epsilon)$. The language generated is the set $L(G) = \{w \mid S \Rightarrow^* w\}$.

Theorem 4. *([11]) Every D^2-grammar G has an equivalent normal form D^2-grammar G' where each production is of one of the following types:*

- *$A \to (BC)(\epsilon); A, B, C \in N$ (branching production)*
- *$A \to (\epsilon)(B); A, B \in N$ (chain production)*
- *$A \to a; A \in N, a \in \Sigma$. (terminal production).*

A derivation in such a grammar is said to be a *normal form derivation* if whenever a non-terminal A is rewritten by a chain production, say $A \to (\epsilon)(B)$, then that occurrence of B is eventually rewritten by either a branch production or a terminal production. That is, no variable participates in two chain rules. For every derivation, there is an equivalent normal form derivation [11].

A typical derivation in this grammar arising from the use of a branching production produces non-contiguous substrings. Say $A \to (BC)(\epsilon) \in P$. Also say $B \Rightarrow^* \beta_1(\epsilon)(\beta) \Rightarrow^* \beta_1\beta_2(\epsilon)(\epsilon)$ and $C \Rightarrow^* \gamma_1(\epsilon)(\gamma) \Rightarrow^* \gamma_1\gamma_2(\epsilon)(\epsilon)$. Then $A \Rightarrow (BC)(\epsilon) \Rightarrow^* \beta_1(C)(\beta) \Rightarrow^* \beta_1\gamma_1(\epsilon)(\gamma\beta) \Rightarrow^* \beta_1\gamma_1\gamma_2(\epsilon)(\beta) \Rightarrow^* \beta_1\gamma_1\gamma_2\beta_2(\epsilon)(\epsilon)$. Thus, we say that in the string $\beta_1\gamma_1\gamma_2\beta_2$, the substring $\beta_1\beta_2$ is produced by B with a *gap*, and the gap is filled by C with the substring $\gamma_1\gamma_2$.

A chain production does not explicitly give rise to a gap in the string. However, the application of a chain production swaps the order of substrings being produced by the non-terminals in the first list. Say $A \to (\epsilon)(B)$ and $B \Rightarrow^* \beta$; *i.e.* A produces a string β via a chain rule. Also say $C \Rightarrow^* \gamma$. Consider a sentential form $w(AC)(\delta)$. The string β produced by A appears in the final string *after* the string γ that is produced by C. That is, we get $w(AC)(\delta) \Rightarrow w(C)(B\delta) \Rightarrow^* w\gamma(\epsilon)(B\delta) \Rightarrow^* w\gamma\beta(\epsilon)(\delta)$. Hence when A produces a string via a chain production, we assume that β has a gap (of length 0) at the beginning (*before* β). Thus, a chain rule always results in a gap at the beginning.

Consider a terminal rule $A \rightarrow a$. Say A appears in some list in a sentential form. The terminal a produced by A appears before all the strings produced by all the non-terminals that follow A in its list. Consider sentential form $w(AC)(\delta)$. Then we get $w(AC)(\delta) \Rightarrow wa(C)(\delta) \Rightarrow^* wa\gamma$ where $C \Rightarrow^* \gamma$. Thus, a terminal production produces a gap (of length 0) at the end (*i.e. after* the terminal).

Ordered, visible two stacks machines ([8]). 2-OVPA are pushdown machines with two stacks, access to both of which is dictated by the input alphabet. The input alphabet Σ is a union of 8 disjoint finite sets: except for simultaneous pops on both stacks, all other combinations are allowed. Also the stacks are accessed in an ordered manner *i.e.* a pop is allowed on the second stack only if the first stack is empty. k-stack versions, k-OVPA, are defined similarly [8]. k-OVPA are essentially restrictions of PD_k, with the exception that they can also make moves when their stacks are empty. See [8] for formal definitions.

3 Reduction from $\mathsf{MEM}(\mathsf{MVPL}_k)$ to $\mathsf{MEM}(\mathsf{PD}_k)$

In this section, we consider the problem $\mathsf{MEM}(\mathsf{MVPL}_k)$ and establish Theorem 1. The simplest case is when $k = 1$; for all fixed MVPA M, $L_1(M) \in \mathsf{VPL}$. Since VPLs are known to be in NC^1 [7], for which membership in a fixed regular language is complete [16], $\mathsf{MEM}(\mathsf{MVPL}_1)$ reduces to $\mathsf{MEM}(\mathsf{NFA})$, where NFA are nondeterministic finite-state automata. But a PD_0 is precisely an NFA. Hence $\mathsf{MEM}(\mathsf{MVPL}_1)$ reduces to $\mathsf{MEM}(\mathsf{PD}_0)$.

For $k > 1$, we reduce this problem to $\mathsf{MEM}(\mathsf{PD}_k)$. As described in Section 2, we assume that $L_k(M) = L(M)$. We convert M into a multi-pushdown machine N having k stacks, called Main_i for $1 \leq i \leq k$, and show that $L(M)$ reduces to $L(N)$ (via logspace many-one reductions).

Consider a phase i in which stack-j ($j \in \{0, 1\}$) of machine M is being popped. The PD works in two stages – *mimic stage* and *buffer stage*. (Exception: phase k has only a mimic stage.)

In the Mimic stage, Main_i and Main_{i+1} contain the contents of stack j and $1 - j$ respectively and mimic the moves of machine M on these two stacks. The rest of the stacks are empty. (In particular for all $l < i$, Main_l are empty.) In the Buffer stage, Main_{i+1} is marked with a special symbol. The contents of Main_i are popped and are pushed onto top of the special symbol (in reversed order), and then popped and pushed into Main_{i+2}. Thus, the contents of Main_i are transferred into Main_{i+2} in the same order. Note that, the contents of Main_k need not be popped at all since there is no subsequent phase, and hence k stacks suffice in N.

To carry out these phases, the input string is padded with some new extra letters by a function f. On reading these letters, N does the necessary transfers. As the next phase expects to pop stack Main_{i+1}, after such a transfer all the stacks are ready for next processing step. More formally,

Lemma 1. *Fix a* MVPA M *and an integer* k. *There exist a* PD_{2k+2} N *and a function* $f \in \mathsf{Log}$*, such that* $\forall w \in \Sigma^*$, $w \in L_k(M) \Leftrightarrow f(w) \in L(N)$.

A small technical difficulty is that MVPAs are allowed pop operations on empty stacks, but PDs cannot make any move if all stacks are empty. If a prefix of an input string has unmatched pop letters (pops on empty stack), then during the mimic phase the simulating machine N may get stuck. To prevent this, we pad the input string with a suffiently long prefix that causes push moves on both the stacks. This boosts the heights of the stacks and ensures that the resulting string has no unmatched pop move. Formally, we show the following:

Lemma 2. *Fix a* MVPA M*. There exists another* MVPA M' *and a function* $g \in$ Log *such that for every string* $w \in \Sigma^*$*,* $w \in L(M) \Leftrightarrow g(w) \in L(M')$*,* M *on* w *and* M' *on* $g(w)$ *have the same number of phases, and* M' *never pops or pushes on empty stack.*

We will call strings obtained by reduction g as *extended* strings and machine M' thus obtained a *good* MVPA. By Lemma 2, we assume that we have a good MVPA M that never uncovers the bottom-of-stack marker (except at the beginning) on either stack on the inputs that it receives.

For an extended string w, let $ht^j(w)$ denote the height of stack-j of a good MVPA M after having processed the string w. Here, $j \in \{0, 1\}$. To compute the function f in Lemma 1, we need the values $ht^j(x)$ for each prefix x of w. These values are easy to compute:

Proposition 1. *For any extended input string* w*, computation of* $ht^j(w)$ *and demarcation of the string into its first* k *phases can be done in* Log*.*

Suppose we have the extended string $w = w_1 w_2 ... w_k$ (on the extended alphabet Σ) already marked with the phases. That is, w_i is the string processed in the ith phase, and the individual strings $w_1, w_2, \ldots, w_k$ are known. Let k_i denote the height of the stack that was popped in phase i, after having processed the ith phase. We have ensured that $k_i \geq 1$ for all i. Let $U, V, W, Z, \#$ be new letters not in Σ. Then f is defined as below. (No padding is needed after w_k.) $f(w) = Z w_1 \# U^{k_1+1} V^{k_1+1} \# W w_2 \# U^{k_2+1} V^{k_2+1} \# W \ldots w_i \# U^{k_i+1} V^{k_i+1} \# W \ldots w_k$

For the PD_k $N = (Q', \Sigma', \Gamma', \delta', q'_0, F')$, Q' consists of $3k$ copies of the states of M, 3 copies for each phase. The first copy is used during the mimic stage and the second and third copies are used for the first and the second steps in the buffer stage respectively. The padding symbol # is used in order to mark the stack Main_{i+1} with a special marker before the buffer-stage begins and then to pop the marker after the contents on top of it are moved into Main_{i+2}. Also Γ' consists of k copies of Γ, with the i-th copy used as the stack alphabet for Main_i.

Formally, the invariant maintained with respect to M can be stated as follows:

Lemma 3. *Machine* M *on input* w *has a non-deterministic path* ρ *in which for each* $i \in [k]$*, after phase* i *(where phase* i *pops stack* j*)* β_i *is on stack* j*,* α_i *is on stack* $1-j$ *and* M *is in state* q *if and only if machine* N *has a non-deterministic path* ρ' *along which for each* $i \in [k]$*, after reading the prefix up to and including* w_i *in* $f(w)$*,* (1) $\beta_i Z_0$ *is on* Main_i*,* (2) $\alpha_i Z_0$ *is on* Main_{i+1}*,* (3) *all the other stacks are empty, and* (4) *the state reached is* $[q^{(1)}, i]$*.*

That is, the runs of machines M *and* N *are in one-to-one correspondence.*

It follows that, M accepts w if and only if N accepts $f(w)$; hence Lemma 1.

4 The LogCFL upper bound for $\mathsf{MEM}(\mathsf{PD}_{\boldsymbol{k}})$

In this section, we show that membership testing for a fixed PD_k is in LogCFL.

The main structure of our LogCFL algorithm closely follows that of the P-time algorithm for membership testing for PD_2 as given in [11]. So we first describe it in some detail (Section 4.1), following the presentation from [17]. We then give (Section 4.2) a different implementation of the same algorithm and improve the upper bound to LogCFL, thus establishing Theorem 2 for $k = 2$. A P-time algorithm for $\mathsf{MEM}(\mathsf{PD}_k)$ is given in [12]. It is very similar to the algorithm from [11]. In Section 4.3, we discuss the changes needed to be made in our implementation for the LogCFL bound to hold for all fixed k, thereby establishing Theorem 2.

4.1 Outline of the P-time algorithm of [11, 17]

The P-time algorithm uses the characterization of PD_2 via D^2 grammars in normal form, and normal-form derivations, as described in Section 2. Given an input $w \in \Sigma^*$, the algorithm needs to keep track of substrings of w being produced with gaps. This is done as follows: A table T in constructed such that any entry in the table is indexed by four indices, $T(i, j, r, s)$. The algorithm fills entries in the table with subsets of N. A non-terminal A is in $T(i, j, r, s)$ if and only if A generates the string $w_{i+1} \dots w_j$ with a gap of length s at position $i + 1 + r$. Here r is the offset from $i + 1$ where the gap begins. The table entry $T(i, j, r, s)$ deals with the interval $inv = [i + 1, j]$ modulo the gap interval $gap = [i + r + 1, i + r + s]$. Let $l = j - i$ denote the total length of the interval and $l' = j - i - s$ denote the actual length of the interval under consideration *i.e.* length of the interval without the gap. The table is filled starting from smaller values of l. Further, the table entries with intervals of the same length l are filled starting from $l' = 1$ going up to $l' = l$. All entries are first initialized to contain the empty set.

For fixed values of l and l', we call a tuple $\langle i, j, r, s\rangle$ *valid for l and l'* if and only if $j = i + l$, $s = l - l'$ and $i + r + s \leq j$ (*i.e.* $r \leq l'$).

For $l = 1$ all the entries are filled by the following two rules, using information from the input and the fixed grammar.

1. $T(i, i + 1, 1, 0) = \{A \mid A \to w_{i+1}\}$
2. $T(i, i + 1, 0, 0) = \{A \mid A \to (\epsilon)(B), B \to w_{i+1}\}$

In the first (second) rule, the table entries correspond to intervals of size 1, where the zero-length gap is at the end (beginning, respectively). It contains the non-terminals that produce the terminal w_{i+1} using a terminal (chain, respectively) production.

As the value of l increases, depending on the position and size of the gap, various rules are used to fill up the table. For $l > 1$, the following rules are applied to fill the table entries corresponding to valid tuples:

Rule 1: This rule is applied provided the interval size is at least 2, and values of r', s' satisfy $r' < r, s < s' < j - i = l$.

$$T(i,j,r,s) = T(i,j,r,s) \cup \left\{ A \;\middle|\; \begin{array}{l} A \to (BC)(\epsilon), \\ B \in T(i,j,r',s'), \\ C \in T(i+r', i+r'+s', r-r', s) \end{array} \right\}$$

For this update, the algorithm uses values from $T(i,j,r',s')$ and $T(i+r', i+r'+s', r-r', s)$. These values are already available. To see this, note that for $T(i,j,r',s')$, the actual interval length is $j-i-s'$ which is strictly less than l' as $s' > s$, and for $T(i+r', i+r'+s', r-r', s)$, the interval length is s' where $s' < l$.

Rule 2: $T(i,j,0,s) = T(i,j,0,s) \cup \{A \mid A \in T(i+s,j,0,0)\}$. This rule is applied when the offset r is zero, *i.e.* when the gap is on the left. Note that this rule makes no update when the length s of the gap is zero.

For this update, the algorithm uses values from $T(i+s,j,0,0)$ (for which length of the interval $j-i-s < l$). This value is already available.

Rule 3: $T(i,j,r,s) = T(i,j,r,s) \cup \{A \mid A \in T(i,j-s,r,0)\}$. This rule is applied when the gap of length s is on the right. This happens when the gap stretches all the way till j, *i.e.* $i+r = j-s$. Note that this rule makes no update when the length s of the gap is zero.

For this update, the algorithm uses values from $T(i,j-s,r,0)$ (for which length of the interval is $j-s-i < l$). These values are already available.

Rule 4: $T(i,j,0,0) = T(i,j,0,0) \cup \{A \mid A \to (\epsilon)(B), B \in T(i,j,r',0)\}$. This rule is applied when s and r are both zero. And $0 \le r' \le j-i$.

For this update, the algorithm uses values from $T(i,j,r',0)$ checking if $A \to (\epsilon)(B)$ and $B \in T(i,j,r',0)$ for some $0 \le r' \le j-i$. Now for $T(i,j,r',0)$, the l and l' values are the same as that for $T(i,j,0,0)$. So we cannot immediately conclude that the required values are already available. However, for fixed l, l', the P-time algorithm performs steps $1,2,3$ before the step 4. Steps $2,3$ leave entries unchanged if $s = 0$. It is sufficient to argue that step 1 in fact puts B in $T(i,j,r',0)$, which is then used in step 4. Suppose not. *i.e.* suppose B is written in $T(i,j,r',0)$ by rule 4. Let $r' = 0$, as rule 4 cannot have been applied if $r' \ne 0$. Also as B is written in $T(i,j,r',0)$ by rule 4, there exists a $C \in N$ and a rule $B \to (\epsilon)(C)$ such that $B \Rightarrow (\epsilon)(C) \Rightarrow^* w_{i+1} \dots w_j$. But then the complete derivation is $A \Rightarrow (\epsilon)(B) \Rightarrow (\epsilon)(C) \Rightarrow^* w_{i+1} \dots w_j$. This contradicts the assumption that we have a normal form derivation. Hence, the required values are already available even for this step.

After a systematic looping through these indices, finally the entry of interest $T(0,n,0,0)$ is filled. If $S \in T(0,n,0,0)$, then the algorithm returns 'yes', else it returns 'no'. The time complexity of the algorithm is $O(n^6)$.

4.2 The LogCFL algorithm ($k = 2$)

We now give a top-down algorithm to fill up the table T. We will see that it can be implemented by a poly sized circuit having $\wedge$ and $\vee$ gates and having poly sized proof trees. From [14, 2] it follows that this algorithm is in LogCFL.

The polynomial time algorithm that fills up the table can be viewed as a polynomial sized circuit. However, this circuit need not have polynomial size proof trees. In particular, the index computations may blow up the proof-tree size. We note that these index computations are independent of the input, and give a way to build a circuit with small proof-trees.

For each $l' \leq l \leq n$, for all valid tuples corresponding to these values of l, l', and for each $A \in N$, we introduce 5 gates: an OR gate $\langle A, i, j, r, s\rangle$ called a *main* gate, and 4 intermediate gates $X^1_{A,i,j,r,s}$, $X^2_{A,i,j,r,s}$, $X^3_{A,i,j,r,s}$, $X^4_{A,i,j,r,s}$ called *auxiliary* gates. We design the circuit in such a way that $\langle A, i, j, r, s\rangle = 1$ if and only if $A \in T(i, j, r, s)$. The root of the circuit is labelled $\langle S, 0, n, 0, 0\rangle$. The circuit connections are as follows:

$$\begin{aligned}
\langle A, i, j, r, s\rangle &= \bigvee_{k\in[4]} X^k_{A,i,j,r,s} \\
X^1_{A,i,j,r,s} &= \bigvee_{\substack{r' < r \\ s < s' < j-i-r+1 \\ \{B, C \mid A \to (BC)(\epsilon)\}}} \begin{pmatrix} \langle B, i, j, r', s'\rangle \bigwedge \\ \langle C, i+r', i+r'+s', r-r', s\rangle \end{pmatrix} \\
X^2_{A,i,j,r,s} &= \begin{cases} \langle A, i+s, j, 0, 0\rangle & \text{if } r = 0 \\ 0 & \text{otherwise} \end{cases} \\
X^3_{A,i,j,r,s} &= \begin{cases} \langle A, i, j-s, r, 0\rangle & \text{if } i+r = j-s \\ 0 & \text{otherwise} \end{cases} \\
X^4_{A,i,j,r,s} &= \begin{cases} \bigvee_{0\leq r'\leq j-i, \{B \mid A\to(\epsilon)(B)\}} X^1_{B,i,j,r',0} & \text{if } r, s = 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

This finishes the description of all the non-leaf gates. The input gates are predicates and their values are propagated via the following depth-1 circuit.

$$\begin{aligned}
\langle A, i, i+1, 1, 0\rangle &= \bigvee_{\{a \mid (A\to a\in P)\}} [i, a, 1, 0] \\
\langle A, i, i+1, 0, 0\rangle &= \bigvee_{\{a \mid \exists B(B\to a\in P)\wedge(A\to(\epsilon)(B)\in P)\}} [i, a, 1, 0]
\end{aligned}$$

Note that the above connections give an acyclic digraph of depth $O(n^2)$.

It is now easy to see the following claim, and hence the correctness of the above circuit follows from the correctness of P-time algorithm.

Lemma 4. $\langle A, i, j, r, s\rangle = 1$ *if and only if* $A \in T(i, j, r, s)$.

The LogCFL bound for MEM(PD$_2$) now follows from the following claim:

Claim. The circuit constructed above has polynomial-size proof-trees.

4.3 The LogCFL algorithm for MEM(PD$_k$)

The grammars [10] corresponding to PD$_k$ have rules with a single non-terminal belonging to one of the k lists on the left hand side and at most k lists of non-terminals on the right hand side. The normal form of the grammar is as follows:

- $(A)_h \rightarrow (BC)_1;\ k \geq h \geq 1$ (branch production; always expands into list 1)
- $(A)_h \rightarrow (B)_g;\ k \geq g > h \geq 1$ (chain production; from list h to a later list g)
- $(A)_h \rightarrow a;\ a \in T;\ k \geq h \geq 1$ (terminal production)

Now, any typical string derived by a non-terminal can have as many as 2^{k-1} gaps; see [12]. If $(A)_h \rightarrow (BC)_1$ is a branch rule, and B, C derive strings γ and δ respectively, then the string derived from A is a systematic merge of γ and δ. In the case when $k = 2$, only one gap was possible, whereas here we need to keep track of 2^{k-1} gaps to interleave γ and δ properly. Arrays $\tilde{r}$ and $\tilde{s}$, of length 2^{k-1} each, keep track of the off-sets and the lengths of the gaps.

Each table entry is indexed by $i, j, \tilde{r}, \tilde{s}$, as in the case $k = 2$. But now the tables are $2^k + 2$ dimensional (as each $\tilde{r}$ and $\tilde{s}$ are 2^{k-1} length arrays). The table entries contain non-terminals and they are filled in such a way that a non-terminal A belongs to a certain entry $T_{i,j,\tilde{r},\tilde{s}}$ if and only if the string $w_i \dots w_j$ with gap off-sets as in $\tilde{r}$ and gap sizes as in $\tilde{s}$ can be obtained from A. The rules for filling up the table are slightly more complicated. However, they simply involve some index manipulations. These can be implemented as we did for $k = 2$. Once these rules are established, the order of filling up the entries and hence the rest of the algorithm is exactly the same. Thus, we obtain Theorem 2.

4.4 Bounds for MVPA and OVPA

From Theorems 1 and 2, it follows that MEM(MVPA$_k$) is in LogCFL.

To see this bound for MEM(k-OVPA), Theorem 2 should suffice, since as claimed in [8], k-OVPA are a special case of PD$_k$. However, there is a slight subtlety here. k-OVPA are allowed to "pop" on an empty stack: if all stacks are empty (they contain only the special letter $\perp$), then the k-OVPA can still proceed with its computation even on a return letter. However, a PD$_k$ in a similar configuration is stuck and cannot make any move. So it is technically not completely correct to say that k-OVPA are PD$_k$. However, we can handle this exactly as we did for MVPA in Lemma 2, reducing MEM(k-OVPA) to MEM(PD$_k$).

5 Discussion

Our results show that adding more stacks to a PDA does not make the fixed membership problem harder than that for ordinary pushdown automata if stack access is restricted to visible behaviour with $O(1)$ phases, or if the the stacks are ordered and the stack pop access is restricted to the first non-empty stack,

Some interesting questions remain unanswered: What complexity classes are characterized by MEM(MVPA$_k$) and MEM(OVPA)? These problems lie somewhere between NC^1 and LogCFL. And what is the complexity of the general membership problem for these models, where the machine and the word are both part of the input?

Acknowledgments: The authors are grateful to the referees for presentation-related comments, and for noting that Theorem 1 needs only k stacks, not $2k-2$.

References

1. Hopcroft, A., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (2001)
2. Sudborough, I.H.: On the tape complexity of deterministic context-free language. Journal of Association of Computing Machinery **25(3)** (1978) 405–414
3. Sudborough, I.H.: A note on tape-bounded complexity classes and linear context-free languages. Journal of Association of Computing Machinery **22** (1975) 499–500
4. Holzer, M., Lange, K.J.: On the complexities of linear LL(1) and LR(1) grammars. In: 9th International Symposium on Fundamentals of Computation Theory FCT, London, UK, Springer (1993) 299–308
5. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: 36th ACM Symposium on Theory of Computing (STOC 2004). (2004) 202–211
6. Mehlhorn, K.: Pebbling mountain ranges and its application to DCFL recognition. In: 7th International Colloquium on Automata, Languages and Programming. (1980) 422–432
7. Dymond, P.W.: Input-driven languages are in $\log n$ depth. Information Processing Letters **26** (1988) 247–250
8. Carotenuto, D., Murano, A., Peron, A.: 2-visibly pushdown automata. In: 11th Developments in Language Theory Conference. (2007) 132–144
9. La Torre, S., Madhusudan, P., Parlato, G.: A robust class of context-sensitive languages. In: 22nd Symposium on Logic in Computer Science. (2007) 161–170
10. Cherubini, A., Breveglieri, L., Citrini, C., Crespi Reghizzi, S.: Multipushdown languages and grammars. International Journal of Foundations of Computer Science **7(3)** (1996) 253–292
11. Cherubini, A., Pietro, P.S.: A polynomial-time parsing algorithm for a class of non-deterministic two-stack automata. In: 4th Italian Conference on Theoretical Computer Science. (1992) 150–164
12. Cherubini, A., Pietro, P.S.: A polynomial-time parsing algorithm for k-depth languages. Journal of Computer and System Sciences **52**(1) (1996) 61–79
13. Allender, E., Jiao, J., Mahajan, M., Vinay, V.: Non-commutative arithmetic circuits: depth reduction and size lower bounds. Theoretical Computer Science **209** (1998) 47–86
14. Ruzzo, W.: Tree-size bounded alternation. Journal of Computer and System Sciences **21** (1980) 218–235
15. Vollmer, H.: Introduction to Circuit Complexity: A Uniform Approach. Springer-Verlag New York Inc. (1999)
16. Barrington, D.: Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. Journal of Computer and System Sciences **38** (1989) 150–164
17. Pietro, P.S.: Two-stack automata. Rapporto Interno n. 92-073, Dipartimento Di Elettronica e Informazione, Politecnico di Milano, Milano. (October 1992) http://home.dei.polimi.it/sanpietr/pubs/twostack92.ZIP.