EXPLORING LOGCFL USING LANGUAGE THEORY

By

Nutan Limaye

The Institute of Mathematical Sciences, Chennai.

A thesis submitted to the
Board of Studies in Mathematical Sciences

In partial fulfillment of the requirements
For the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

December 2009

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Board, we recommend that the dissertation prepared by **Nutan Limaye** entitled "Exploring LogCFL using language theory" may be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

-- **Date :**
Chairman : V. Arvind (IMSc)

-- **Date :**
Convener : Meena Mahajan (IMSc)

-- **Date :**
Member : Somenath Biswas (IITK)

-- **Date :**
Member : Samir Datta (CMI)

-- **Date :**
Member : Kamal Lodaya (IMSc)

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to HBNI.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it may be accepted as fulfilling the dissertation requirement.

-- **Date :**
Guide : Meena Mahajan (IMSc)

DECLARATION

I hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and the work has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution or University.

Nutan Limaye

. . . . to Aai Baba

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Meena Mahajan, for making my years of graduation extremely fruitful, enjoyable, and memorable. When I joined the graduate program, all I had was interest in theoretical research. In some sense, Meena has moulded my personality as a researcher. I have been hugely benefited by numerous technical discussions I have had with her. I owe my interest in complexity theory and circuit complexity to her. Her patient guidance, support, and encouragement have led me towards the completion of this thesis. Also, her multifaceted personality, and her zeal and enthusiasm for a gamut of things (reading, cooking, biking, trekking) have had a great influence on me. For all the things I learned from her, technical and otherwise, I will forever be grateful.

I would like to thank Prof. Samir Datta for sharing his ideas, for getting me interested in problems related to planar graphs, and for many collaborations.

A special thanks to Prof. Ramanujam for being my mentor. On the most trying days in the last few years, his faith in me has helped me stay afloat.

I would like to thank Needhara and S P Suresh for helping me with the fonts and the diagrams and for the wonderful walks to Sri Krishna Sweets.

I wish to thank all my collaborators. Many of them – Antoine, Prajakta, Raghav – are close pals now. I would also like to thank the people with whom I have had useful discussion on technical topics related to this thesis or otherwise. These include Didier Caucal (for discussions on synchronized PDA), Rahul Muthu (for graph theory sessions), Srikanth Srinivasan (for discussions on pseudorandom generators, coding theory, S_2P, and many interesting puzzles), and Jacobo Töran (for inviting me to visit Ulm University and discussions on the hardness of graph isomorphism).

I also thank all the Computer Science faculty at the Institute of Mathematical Sciences, IMSc, who have been responsible for imparting to me the knowledge of various aspects of computer science during my course work and through numerous seminars.

The work leading to this thesis was carried out completely at IMSc. I would like to thank the institute for its financial support and the wonderful infrastructure and working environment that it provides to its members.

A warm thanks to the all my friends for making my social life extremely interesting and enjoyable. The books, the music, the movies, the long chai sessions, the long nightouts spent talking, the treks, the dance lessons, the cooking; what would I be without all this? A big thanks to Chikku for her support and for teaching me to become responsible.

Finally, but most importantly, I wish to thank my parents who have filled my life with joy and have always had confidence in me, even when I myself did not. If it wasn't for their encouragement and support, taking up research would have been nearly impossible. Words are not enough to express my gratitude for how much they have done for me.

Abstract

We explore the complexity classes contained in LogCFL by drawing connections to language theory. Along the way, we study many depth reduction techniques.

We consider two factor-2 approximation algorithms for Block sorting and show that both the algorithms can be implemented in LogCFL. We use depth reduction techniques and specialize them for these problems and give explicit NC^2 algorithms. We also study membership problem for multi-pushdown machines. The stacks in these machines are ordered and pop moves are allowed on the first non-empty stack. We prove that the membership problem for these machines is complete for LogCFL.

We consider visibly pushdown languages, VPLs, and many generalizations of VPLs and use these languages to draw connections to the complexity classes inside LogCFL. A VPA is a nondeterministic pushdown automaton with the restriction that the height of the stack for a given input is same along every nondeterministic run and is easy to compute. We abstract out this idea to define the notion of height-determinism and obtain generalizations of VPLs, namely rhPDA(FST), rhPDA($\mathsf{rDPDA}_{1\text{-turn}}$), rhPDA(PDT), where rhPDA denotes the realtime height deterministic PDA.

We prove that the height of the stack for these machines are functions (height functions) computable in NC^1, L, and LogDCFL, respectively, and prove that the membership problem for all three of them is in L^h where h is the complexity of the height function.

We also consider multi-stack pushdown machines with visible stacks. A phase is a set of consecutive computation steps during which the machine pops exactly one stack. We consider membership problem for multi-stack pushdown machines with bounded phases and show LogCFL upper bound.

Finally, we consider the counting problem for VPLs, and its generalizations, namely: rhPDA(FST), rhPDA($\mathsf{rDPDA}_{1\text{-turn}}$), and rhPDA(PDT), and prove LogDCFL upper bound for all. We also consider the counting functions corresponding to VPLs and analyze their closure properties.

Contents

List of Figures

List of Tables

1 Introduction

The subject of complexity theory deals with bounding the exact amount of resources needed to perform certain computational tasks. The resources may be time or space or any other relevant parameters. Complexity theory is a rich, flourishing, and a well-studied area in computer science. Over the years, many natural problems have been studied by complexity theorists. The progress in understanding the resources needed for solving the natural problems has been noteworthy. Also many relations between seemingly orthogonal resource bounded computations have been uncovered. For example, it is known that if a Turing machine uses nondeterminism and certain amount of space, the nondeterminism can be removed by squaring the space [47]. Many such containments and relations have been established among the known complexity classes. However, the progress made towards obtaining separations between the complexity classes has been unsatisfactory. Most of the containments are not known to be strict.

Language theory is one of the oldest areas of study in computer science. A languages can be thought of as a set of strings over an alphabet which is accepted by a machine. Here, a machine is a device consisting of a finite control and possibly a storage mechanism. The set of strings on which the computation of the machine ends in a favorable configuration is the language accepted by the machine. Finite state machines, pushdown machines, Turing machines *etc.* are examples of machines, and regular languages (REG), context-free languages (CFL), and recursively enumerable languages are the languages accepted by them, respectively. Many relations and containments regarding the language classes are known. And unlike in the case of complexity classes, many of the containments are known to be strict.

The work discussed in this thesis is inspired by the early works of Sudborough and Barrington who discovered close connections between language theory and complexity theory [48, 10]. The primary goal of this thesis is to understand the connection between the language classes and the complexity classes. The language classes we study properly generalize REG. The complexity classes to which they have connections are contained in the class P, a class of functions or languages computable by polynomial time Turing machines. The peculiarity of the complexity classes considered in this thesis is that they have polynomial time algorithms which are *depth reducible*. (We will give an informal description of the notion of depth reduction shortly.) Thus the other competing goal of this thesis is to better understand the known depth reduction techniques and to use them in newer contexts.

The two goals are very well motivated from complexity theoretic point of view. To state it in a nutshell, the motivation is to take another approach to attack the problem of separating the complexity classes. The main driving force is the belief that the knowledge about the language classes can become useful in order to improve our knowledge about the complexity classes. Also, this method benefits from witnessing depth reduction ideas at each juncture. In recent times, depth reduction is gaining importance in making concrete statements about fundamental problems in computer science [3, 1]. This motivates the need for understanding this technique.

In Section 1.2 we will summarize the primary contributions of this thesis. In order to describe the main results, we will need to develop some common vocabulary. The next section is devoted to building up the vocabulary.

1.1 Important notions

Each notion described in this section is discussed in detail in the main exposition. Here, we mainly give an informal description of the notions in order to build up a common set of words. If the notions are familiar, you may choose to jump to the next section.

1.1.1 Depth Reduction

Let A be an algorithm to solve a problem P that runs for time $t(n)$, where n is the length of the input and t is a polynomial in n. If there is a way to find polynomially many subproblems of the problem P such that each can be solved independent of all the others in time polylogarithmic in the length of the input (i.e. $O(\log^i t(n))$ for some $i \in \mathbb{N}$) and the answers of all the subproblems can be combined with an additive polylogarithmic overheard (of $O(\log^j t(n))$ for some $j \in \mathbb{N}$) then the implementation A is said to be *depth reducible*. It is also called *parallelizable*. This refers to the property of such implementations to solve many independent subparts simultaneously. (It should not be mistaken for processor efficient parallel implementations.)

1.1.2 Membership problem

Fix a machine. The membership problem deals with, given a word over the alphabet of the machine, checking whether it is accepted by the machine. In the literature, the problem is also sometimes referred to as fixed membership problem because the machine is fixed. The version of the problem where machine is also a part of the input has been considered in the literature. But our focus is on the question where the machine is fixed.

1.1.3 Counting problem

For a fixed machine, the membership problem deals with checking whether there is at least one accepting path in the machine for a given input. The counting problem asks for more. Fix a machine. Given an input over the alphabet of the machine, it asks for the exact number of accepting paths in the machine on that input.

Thus, the counting problem is at least as hard as the membership problem. If the underlying machine is deterministic then an algorithm for membership problem can solve the counting problem. The complexity of the counting problem is interesting when the fixed machine is nondeterministic.

1.1.4 Visibly Pushdown Language: The starting point

Visibly pushdown language, VPLs, are a proper subclass of context-free languages. They are defined as accepted by certain restricted pushdown machines as well as generated by certain restricted grammars. We will focus on the former. The pushdown machines that accept VPLs use their stack in a restricted manner. There are three

$$\begin{tikzcd}
\boxed{\mathsf{NC}^1} \arrow[r] & \boxed{\mathsf{L}} \arrow[r] \arrow[dr] & \boxed{\mathsf{LogDCFL}} \arrow[dr] & & & \\
& & \boxed{\mathsf{NL}} \arrow[r] & \boxed{\mathsf{LogCFL}} \arrow[r] & \boxed{\mathsf{NC}^2} \arrow[r] & \boxed{\mathsf{P}}
\end{tikzcd}$$

Figure 1.1: The complexity classes at the center stage of the thesis.

different ways in which the stack can be altered. Either one can push something on the stack, or pop from the stack, or leave it unchanged. If the input alphabet of a pushdown is partitioned into three parts and if each part contains letters of the alphabet which modify the stack in exactly one way then the languages accepted by such pushdown machines are called VPLs. The word visibly signifies that the input letter makes the stack movement transparent. The term was coined by Alur and Madhusudan in their paper [5]. However, such pushdown machines existed in the literature under the name of Input driven pushdowns, see for example [40, 15].

We use this PDA model as the starting point for drawing connection between language classes and complexity classes.

1.2 Contributions of this thesis

This thesis investigates the language classes from the perspective of a complexity theorists. We give a high level description of the contributions of this thesis here.

Figure 1.1 shows the complexity classes of concern for the purpose of this thesis. At this point it is not important to know the definitions of the classes. However, the main point to be emphasized is: the considered classes are all contained in P. We are inside that region of P for which parallelizable algorithms exist. The class NC^2 and NC^1 are the two ends of the spectrum of interest to us. The former class contains many interesting complexity classes and the latter is at the frontier of the lower bounds literature. It is this region of the complexity zoo that is known to benefit from connections to language theory. Our main contribution therefore is to further this study and examine this region under a magnifying glass.

The results obtained are of three types. The thesis is divided into three parts each dedicated to one of the types. Given below is a succinct summary of the flavor of the results established in each part of the thesis.

Part I	Depth reduction applied to natural problems to obtain improved upper bounds.
Part II	Membership problems for various languages giving close connections to the complexity classes.
Part III	Counting problems for various language classes.

Part I The first part of the thesis considers three problems that were known to be in polynomial time. The problems considered are natural problems, the first two of them arising in biology and the third one in parsing applications. These problems were not known to be hard for P. In this part, upper bound of LogCFL is obtained for all the three problems. The upper bound is an improvement over the known bounds for all the three problems and settles the complexity of the third one (i.e. LogCFL hardness was known for it and this upper bound makes the problem complete for LogCFL).

The ideas used for obtaining the upper bound arise from depth reduction literature. A detailed explanation of one such depth reduction is presented here. It is known that LogCFL is in NC^2 (see Figure 1.1). However, from a LogCFL algorithm an explicit NC^2 implementation may not be directly clear. Also, very often the NC^2 implementation is needed explicitly for application purposes. For two out of the three problems an explicit NC^2 algorithm is presented in this part of the thesis.

Part II The second part of the thesis deals with the membership problem for various generalizations of VPLs. The membership problem for VPLs is known to be as easy as that for regular languages [25]. This is in contrast to the membership problem for CFLs. Thus, these two known results grab the two ends of the spectrum of the complexity classes of our concern. The study done in this part involves obtaining a systematic generalization of VPLs to be able to traverse the spectrum. The main contributions here are: we obtain bounds on the membership problem for three generalizations of VPLs. We show that the weakest and the strongest generalizations characterize NC^1 and LogDCFL respectively. We give an upper bound of logspace for the intermediate generalization. Many interesting depth reduction techniques are implicitly used to prove these bounds. The methods developed here achieve the connections between complexity classes and language classes in a unifying manner.

Part III The third part deals with the counting problem for VPLs and for some of the generalizations of VPLs. Here, the counting problem for VPLs is shown to be in LogDCFL. The problem is at least as hard as its membership problem and hence at least as hard as the membership problem for regular languages. Thus, it sits between the classes NC^1 and LogDCFL. [1] We study the closure properties of the counting class corresponding to the VPLs in order to improve our understanding of this class.

The technique used for proving LogDCFL upper bound is a significant modification of a depth reduction method of [15]. The most important and interesting aspect of this modification is its increased scalability. We give LogDCFL upper bound for the counting problem of a powerful generalization of VPLs using the ideas coming from the same method. The most general language class we consider is equivalent to realtime DCFLs, and VPLs are a proper subclass of it. But the counting problem continues to have the same upper bound as for VPLs.

[1] In fact, it is also at least as hard as the counting problem for regular languages (the machine model considered for this is an NFA), and hence harder than a class which contains NC^1, namely #BWBP.

2 Preliminaries

Some results and notations that are used throughout the text are collected in this chapter for convenience. A reader may wish to review this chapter quickly at first and refer back to it as the concepts appear in the course of the text.

2.1 Language classes and automata models

A *pushdown automaton* (PDA) over a finite input alphabet Σ is a tuple $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$ where Q is a finite set of control states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final or accepting states, Γ is a finite alphabet of stack symbols, and δ is a finite set of transition rules: $pU \xrightarrow{a} qV$ with $p, q \in Q$, $U, V \in \Gamma^*$ and $a \in \Sigma \cup \{\epsilon\}$.

If all the rules $pU \xrightarrow{a} qV \in \delta$ are of the form $p \xrightarrow{a} q$ or $p \xrightarrow{a} qA$ or $pA \xrightarrow{a} q$ with $A \in \Gamma$, i.e. $|UV| \leqslant 1$ then P is called *weak*. If in every rule $pU \xrightarrow{a} qV \in \delta$, either U or V is ϵ, then P is called *one-way-strong* , *strong* otherwise. If a is ϵ, the move defined by δ is called an ϵ-move. A PDA is ϵ-*move-free* or *realtime* if δ has no ϵ-moves.

A *configuration* is a three tuple $\langle q, w, \gamma \rangle$, where $q \in Q, w \in \Sigma^*$, and $\gamma \in \Gamma^*$. The stack height at configuration $\langle q, w, \gamma \rangle$ is $|\gamma|$, where $|x|$ denotes the length of the string x. The *initial configuration* on input w is $\langle q_0, w, \epsilon \rangle$. A configuration is *final* if q is in F.

The transition relation δ gives a *step* relation between configurations. If $pU \xrightarrow{a} qV \in \delta$, then the PDA P in state p can read a letter a and if the stack-top is U, it can go to state q in one step replacing U with V. Formally, $\langle p, aw, U\gamma \rangle \vdash \langle q, w, V\gamma \rangle$. Here $\vdash$ denotes the step relation. A reflexive transitive closure of the step relation defines the runs of the PDA.

An input string w is said to be *accepted* by P, if P has a run that reaches a final configuration starting from $\langle q_0, w, \epsilon\rangle$. This acceptance condition is called *acceptance by final state*. A PDA is said to *accept with an empty stack* if starting from configuration $\langle q_0, w, \epsilon\rangle$, P reaches a configuration $\langle q, \epsilon, \gamma\rangle$ where $\gamma = \epsilon$. The set of strings accepted by PDA is called the language accepted by the PDA. The two acceptance conditions for PDA are equivalent and the class of languages accepted by PDA are called context-free languages (CFLs).

The semantics of P can also be defined with respect to its transition graph G_P whose vertices are of the form pW with $p \in Q$ and $W \in \Gamma^*$. And edges are given by $\{pUW \xrightarrow{a} qVW \mid pU \xrightarrow{a} qV \in \delta,\ W \in \Gamma^*\}$.

A *run* of P on input word $w \in \Sigma^*$ from a vertex pW is a path labeled w in G_P from vertex pW to some vertex 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 all words $w \in \Sigma^*$ such that for some vertices $c \in S$, $c' \in T$, there is a run from c to c' on w. The *language* accepted by a PDA P is the set of all words w over which there exists an accepting run, or equivalently it is the language $L(G_P, \{q_0\}, F\Gamma^*)$ which we also denote by $L(P)$.

A context free grammar G is a tuple (N, T, P, S) where N is a finite set of non-terminals, T is a finite set of terminals, S is a designated start non-terminal, and P is a finite set of productions of the form $A \rightarrow \gamma$ where $A \in N$, and $\gamma \in (T + N)^*$. A *sentential form* is a string over $(T + N)$. If $A \rightarrow w\gamma$ is a production in P then $\alpha A\beta \Rightarrow \alpha w\gamma\beta$ is a *derivation* in the grammar where $\alpha, \beta \in (T + N)^*$. We use $\Rightarrow^*$ to denote the reflexive transitive closure of $\Rightarrow$. The language $L(G)$ generated by a grammar G is set of all the strings in T^* that appear as a sentential form in some derivation starting from S, *i.e.,* $L(G) = \{w|S \Rightarrow^* w \text{ and } w \in T^*\}$. Context-free languages are also defined as languages generated by such grammars.

If for a PDA P, the transition rule δ is a (possibly partial) function then PDA P is said to be a *deterministic* PDA, DPDA. The languages accepted by such PDA are called *deterministic context-free languages*, DCFLs. If every run of the PDA is restricted such that once it pops something from the stack it never pushes on it, then it is called *one-turn* PDA, $\mathsf{PDA}_{1\text{-turn}}$. For a $\mathsf{PDA}_{1\text{-turn}}$, consider the time-vs-height of the stack graph, where time is on the x-axis and heights on the y-axis. It has non-decreasing heights till the first pop step. After which it has non-increasing heights. The turn refers to the turn that this plot makes at the first pop. The languages accepted by such

PDA are called *linear languages,* LIN. [1] One can define a deterministic counterpart of the same. It is called *one-turn* DPDA, $\mathsf{DPDA}_{1\text{-turn}}$. The languages accepted by such PDA are called *deterministic linear languages,* DLIN.

If we remove the stack from the description of a PDA, we are left with only the finite control, *i.e.,* with a finite set of states and a finite set of transition rules. The automata thus obtained are called *nondeterministic finite state automata,* NFA. Its deterministic counterpart is called *deterministic finite state automata,* DFA. NFA are known to be determinisable. The languages accepted by them are called *regular languages,* REG.

Consider the following example:

Example 2.1.1 *Let* $\Sigma = \{a, b\}$.

- *Consider the set of strings over* Σ *that end with the letter* b, $\Sigma^* b$. *These are accepted by a finite state automata.*
- *Consider the set of strings* $\{wcw^R \mid w \in \Sigma^*\}$. *We call this a set of marked palindromes (palindromes with a halfway marker). This is accepted by* $\mathsf{DPDA}_{1\text{-}turn}$. *But it is known that* NFA *do to not accept it.*
- *The set of palindromes,* $\{ww^R \mid w \in \Sigma^*\}$, *is accepted by a* $\mathsf{PDA}_{1\text{-}turn}$ *but not by a* $\mathsf{DPDA}_{1\text{-}turn}$ *or by a* DPDA.
- *The set of string with equal number of* a*'s and* b*'s,* $\mathsf{EQ}(a, b) = \{w \mid \#_a(w) = \#_b(w)\}$, *is accepted by a* DPDA *but not by* $\mathsf{PDA}_{1\text{-}turn}$.
- *The set of strings,* $\{w \mid$ *where* w *is not of the form* uu *for* $u \in \Sigma^*\}$, *is accepted by a* PDA *but by none of the above machine models.*

Figure 2.1 shows the various machine models we have defined. An arrow from machine model M to M' indicates that M' generalizes M. Figure 2.2 shows the corresponding language classes for the above machine models. The dotted arrows indicate proper containments. The dashed arrows between two language classes indicate that they are incomparable. The above relations and various properties of the language classes or the machine models discussed above are well-known. See for example, [31].

[1] The name *linear* comes from the grammar form for it. See for example [31].

$$\begin{tikzcd}
\boxed{\mathsf{DFA}} \arrow[r] \arrow[dr] & \boxed{\mathsf{DPDA}_{1\text{-turn}}} \arrow[r] \arrow[dr] & \boxed{\mathsf{DPDA}} \arrow[dr] & \\
& \boxed{\mathsf{NFA}} \arrow[r] & \boxed{\mathsf{PDA}_{1\text{-turn}}} \arrow[r] & \boxed{\mathsf{PDA}}
\end{tikzcd}$$

Figure 2.1: Machine models for the languages in Figure 2.2

$$\begin{tikzcd}
\boxed{\mathsf{REG}} \arrow[r, dotted] & \boxed{\mathsf{DLIN}} \arrow[r, dotted] \arrow[dr, dotted] & \boxed{\mathsf{DCFL}} \arrow[d, dashed, leftrightarrow] \arrow[dr, dotted] & \\
& & \boxed{\mathsf{LIN}} \arrow[r, dotted] & \boxed{\mathsf{CFL}}
\end{tikzcd}$$

Figure 2.2: The containment relations for the formal language classes

2.2 Complexity classes

2.2.1 Pushdown machines and complexity classes

In this section, we assume basic familiarity with the Turing machine model, the notion of polynomial time or logspace reductions, and the notion of complete problems for a complexity class.

The class of languages accepted by a nondeterministic Turing machine whose work-tape is upper bounded by $O(\log n)$-space defines a complexity class NL. The natural problem complete for this is directed graph reachability. L is the complexity class consisting of languages which are accepted by a deterministic Turing machine whose work-tape is bounded by logspace. The natural problem complete for this class is undirected graph reachability [44]. [2]

The class of languages logspace many-one reducible to a CFL is called LogCFL. This complexity class was defined by Sudborough [48]. A deterministic counterpart of the above, LogDCFL, is the class of languages logspace many-one reducible to a DCFL.

An *auxiliary pushdown automaton*, AuxPDA, is a pushdown automaton augmented with a $S(n)$-space work-tape. If a DPDA is augmented with a $S(n)$-space work-tape, the automata thus obtained is called *deterministic auxiliary pushdown*

[2] Here the problem is assumed to be hard under some reductions which are more restricted than logspace reductions. We will define this carefully below.

automaton, DAuxPDA.
Various bounds can be put on the work-tape size in the above defined machine model. The following are the complexity classes that can be derived by putting meaningful bounds on time and space used by the above machines.

AuxPDA-Time(f) is the class of languages accepted a AuxPDA having logspace bounded work-tape and time bounded by $O(f)$. Note that, due to auxiliary space, NL$\subseteq$ AuxPDA-Time($poly$). In the case of CFLs and PDAs we know that there is an exact correspondence between the two. Similarly, it is known that LogCFL $=$ AuxPDA-Time($poly$), and that LogDCFL $=$ DAuxPDA-Time($poly$) [48].

2.2.2 Alternating Turing machines and complexity classes

In this section we will formally define the computation model of alternating Turing machines and introduce basic notions related to the model. We will also define some complexity classes that are of interest to us.
An *alternating Turing machine*, ATM, is a quintuple $M = (Q, \Sigma, \delta, q_0, g)$ where, Q, Σ, δ are set of states, input alphabet and transition rules respectively. g is a state type function $g : Q \rightarrow \{\wedge, \vee, 0, 1\}$. Let $q \in Q$. If $g(q) = 0$ then q is a rejecting state. If $g(q) = 1$ then q is an accepting state. If $g(q) = \wedge$ then q is a universal state, and if $g(q) = \vee$ then q is an existential state.
ATM's are a generalization of nondeterministic Turing machines.

A computation of an ATM can be viewed as a tree of configurations. A tree is a *computation tree* of an ATM M on string w if its nodes are labeled with the configurations of M on w, such that the descendants of any non-leaf node labeled by a universal (an existential) configuration include all (respectively, one) of its successor configurations.

A computation tree is an *accepting tree* if its root is labeled with the initial configuration and all its leaves are accepting configurations.

The time or space utilized by the ATM is measured as follows: An ATM is said to use time $T(n)$ if for all the accepted inputs of length n there is an accepting computation tree of height $O(T(n))$. An ATM is said to use space $S(n)$ if each node of the tree can be labeled by a configuration using space $O(S(n))$. It is known that ATMs that use $O(\log n)$ space are computationally equivalent to deterministic polynomial time Turing machines *i.e.,* P.

Let M be an alternating Turing machine, and let $t : \mathbb{N} \to \mathbb{N}$. We say that M is *tree-size bounded by* t, if for every $x \in L(M)$, there is an accepting computation subtree of M on x which has $O(t(|x|))$ nodes.

Consider a nondeterministic Turing machine M and an ATM M' both using space $O(S)$. The height of their computation trees will be $2^{O(S)}$. The accepting subtree for M is a root to leaf path and hence of size $2^{O(S)}$. The accepting tree for M' can be potentially the whole computation tree and hence of size $2^{O(S)^{O(S)}}$.

Using tree-size as a parameter, a hierarchy of ATMs can be defined. It is not known whether any of the containments in the hierarchy are strict.

Given below are some complexity classes that can be obtained from the ATM model by bounding the tree-size parameter.

ASPACE-TREESIZE$(S(n), Z(n))$ is the class of all languages L for which there is an alternating Turing machine M which is space bounded by $S(n)$ and tree-size bounded by $Z(n)$, such that $L = L(M)$

The class ASPACE-TREESIZE$(S(n), Z(n))$ where $S(n) = O(\log n)$ and $Z(n) = O(poly(n))$ clearly contains NL and is clearly in P. It is known to be equivalent to the class AuxPDA-Time(poly) due to [45] and thus characterizes LogCFL.

2.2.3 Boolean Circuits and related complexity classes

Switching circuits were investigated in early papers by Lupanov and Shannon. The research of circuits was highly influenced by Savage's early papers and text book [46]. Savage related Boolean circuits to other well-established computation models like Turing machines. Also around 1980s some interesting lower bounds were obtained. Around the same time circuits were related to the parallel computation model, in particular, parallel random access machines (PRAMs).
This model of computation differs from the other models of computation in the following sense. In circuits there are only fixed number of input gates. Hence a single circuit only works on inputs of fixed length in contrast to the other models like Turing machines which work for inputs of arbitrary lengths. Thus to solve usual problems on circuits we need to look at a family of circuits as opposed to a single circuit.

A Boolean circuit C is a directed acyclic graph (DAG) whose source nodes are input gates, one of the sink node is a designated output gate, and intermediate nodes are $\wedge$ (AND), $\vee$ (OR), or $\neg$ (NOT) gates. The size of the circuit is the size of this DAG. Let C_n denote a circuits over n inputs. Let $\mathcal{C} = (C_n)_{n \in N}$ denote a family of circuits.

An input gate labeled 1 (0) is said to have value 1 (0, respectively). A gate labeled x_i ($\neg x_i$) is assigned the value of the variable x_i (negation of x_i, respectively). The value of a gate g labeled AND (OR) with predecessors $g_1, g_2, \ldots, g_k$ is $\wedge_{i=1}^{k} g_i$ ($\vee_{i=1}^{k} g_i$, respectively). A gate g labeled NOT has only one predecessor, say g'. The value of the gate g is $\neg g'$. The value of the circuit is 1 if and only if the value of its designated output gate is 1. A circuit on n inputs *accepts* $x = x_1 x_2 \ldots x_n \in \{0, 1\}^n$ if and only if it evaluates to 1 on x. A language accepted by a circuit family $\mathcal{C} = (C_n)_{n \in N}$ is a union over all $n \in \mathbb{N}$, of the set of strings accepted by each C_n.

Uniform circuit families: Informally, uniform circuit family is a circuit family with a finite description. Let the gates of the circuit be numbered in some order say $v_1, v_2, \ldots, v_s$. Let each gate of the circuit be encoded using its gate number, its type (the types are AND, OR, NOT and they too are numbered in some arbitrary order) and the numbers of its predecessor gates. Let $\hat{v}$ denote the encoding of gate v. Then the encoding of C, denoted as $\hat{C}$, is given by $(\hat{v}_1, \hat{v}_2, \ldots, \hat{v}_s)$. A circuit family $\mathcal{C} = (C_n)_{n \in N}$ of size s is

- logspace-uniform if the map $1^n \to \hat{C}_n$ is in $\mathsf{DSPACE}(\log n)$.
- P-uniform if the map $1^n \to \hat{C}_n$ is in $\mathsf{DTIME}(n^{O(1)})$.

The gates of the circuit may have bounded or unbounded fan-in (*i.e.,* in-degree of a node in the underlying DAG may be bounded or unbounded).

NC^k is the class of languages A for which there is a circuit family $\mathcal{C} = (C_n)_{n \in N}$ over bounded fan-in AND and OR gates of polynomial size and $O(\log^k n)$ depth that accepts A. $\mathsf{NC} = \bigcup_{k \geqslant 0} \mathsf{NC}^k$

AC^k (SAC^k) is the class of languages A for which there is a circuit family over unbounded fan-in AND and OR gates (over bounded fan-in AND gates and unbounded fan-in OR gates, respectively) of polynomial size and $O(\log^k n)$ depth that accepts A.

TC^k is the class of languages A for which there is a circuit family over unbounded fan-in AND, OR gates, and Majority gates of polynomial size and $O(\log^k n)$ depth that accepts A. The Majority gates evaluate to 1 if atleast half its inputs are 1, 0 otherwise.

The class AC is defined as $\mathsf{AC} = \bigcup_{k \geqslant 0} \mathsf{AC}^k$, SAC is defined as $\mathsf{SAC} = \bigcup_{k \geqslant 0} \mathsf{SAC}^k$, and TC is defined as $\mathsf{TC} = \bigcup_{k \geqslant 0} \mathsf{TC}^k$.

$$\begin{tikzcd}
\boxed{\mathsf{NC}^1} \arrow[r] & \boxed{\mathsf{L}} \arrow[r] \arrow[dr] & \boxed{\mathsf{LogDCFL}} \arrow[dr] & & \\
 & & \boxed{\mathsf{NL}} \arrow[r] & \boxed{\mathsf{LogCFL}} \arrow[r] & \boxed{\mathsf{NC}^2}
\end{tikzcd}$$

Figure 2.3: The containment relation for the complexity classes

The following are the known relations between these classes (see for example [53]).

Lemma 2.2.1 *For all* $k \geqslant 0$*, we have* $\mathsf{NC}^k \subseteq \mathsf{SAC}^k \subseteq \mathsf{AC}^k \subseteq \mathsf{TC}^k \subseteq \mathsf{NC}^{k+1}$*, hence* $\mathsf{AC} = \mathsf{NC} = \mathsf{SAC} = \mathsf{TC}$.

We will see in Chapter 3 why SAC^1 contains LogCFL. In fact, it is known that SAC^1 characterizes LogCFL [45, 51, 39].

There is a notion of reducibility that is coined from the class NC^1. We used the notion called logspace many-one reducibility to define LogCFL. We formally define notion of NC^1 many-one reducibility as follows:

For $A, B \subseteq \{0,1\}^*$ we say that $A \leqslant_m^{\mathsf{NC}^1} B$, if there is a function $f \in \mathsf{FNC}^1$ such that for all $x \in \{0,1\}^*$, we have: $x \in A \iff f(x) \in B$.

This notion of reducibility is helpful to prove hardness for smaller complexity classes. Here, FNC^1 refers to functional NC^1. If g is in FNC^1, the language

$$L_g = \{(x, i, b) \mid i\text{th bit of } g(x) \text{ is } b\} \text{ is in } \mathsf{NC}^1$$

Figure 2.3 depicts the containment relation between various complexity classes that we will use in the subsequent chapters. Note that it is not known whether any of these containments are strict.

3 A general paradigm for efficiently parallelizable problems

3.1 What is depth reduction?

It is believed that problems that are polynomial time complete have an inherently sequential solution. On the other hand, the problems that are solvable in NC have efficient parallel implementations. By efficient we mean they use polynomial sized circuits which have polylogarithmic depth.

In the literature of parallel computation these NC algorithms may be considered non-efficient in the following sense: If the problem for inputs of length n has circuits of size, say n^{10} then for all practical purposes for large n this gives impractical implementations (in spite of their polylogarithmic depth). However, from complexity theoretic point of view, this will be termed as efficient implementation. This is because, our measure of efficiency is the depth of the implementation.

It is not known whether any problem complete for P can be solved in NC, *i.e.,* whether $\mathsf{P} = \mathsf{NC}$? The technique of converting a long sequential computation into a short parallel computation, is called *depth reduction*. The $\mathsf{P} = \mathsf{NC}$? question can be restated as whether a depth reduced computation can be explicitly and efficiently designed for any polynomial time sequential computation.

Though this more general question is hard to answer, depth reduction results for various other sequential models of computation exist in the literature. Given a Boolean formula, which is essentially a tree with internal nodes labeled by Boolean operators $\{\vee, \wedge, \neg\}$ and leaves labeled by $\{x_1, x_2, \ldots, x_n, 0, 1\}$, and given an assignment for the variables $x_1, x_2, \ldots, x_n$, checking whether the formula evaluates to 1 or not, is referred to as Boolean formula value problem. The depth of this tree or the formula can be polynomial in n. The obvious sequential computation to solve this problem is a simple depth first traversal of the tree which will take time linear in the size of the tree. But it is known due to Buss [16] that this problem can be solved in NC^1. This means that there is a way to cleverly cut the underlying tree into subparts. Each subpart can be solved efficiently in parallel (in this case using polynomial size circuits of $O(\log n)$ depth or NC^1) and the solutions can in turn be combined efficiently in parallel (in NC^1) to solve the whole problem.

LogCFL is the largest Boolean complexity class inside P for which a general (not problem specific) depth reduction is known. A detailed explanation of a depth reduction result for LogCFL is presented in Section 3.2. The proof presented here is by Venkateswaran [51]. There are two more proofs for the same result extant in the literature due to Vinay, and Niedermeier and Rossmanith [52, 41].

In Section 3.3, Section 3.4, we consider three different problems. All of them were known to have polynomial time upper bound. We prove a LogCFL upper bound for all of them. The ideas used for obtaining the improved upper bound arise from depth reduction literature.

3.2 Depth Reduction for LogCFL

The first depth reduction for the class LogCFL (which is same as AuxPDA-Time(poly), see Section 2.2.1), was given by Venkateswaran [51]. The depth reduction proof used the ASPACE-TREESIZE characterization of LogCFL (see Section 2.2.2) and proved that ASPACE-TREE($O(\log n)$, poly(n)) is equal to SAC^1. This result combined with [45, 39] proves that LogCFL and SAC^1 are equal.

It is easy to see that SAC^1 is contained in ASPACE-TREE($O(\log n)$,poly(n)). The ATM simulating a uniform SAC^1 circuit simply makes universal guesses at AND gates and existential guesses at OR gates. It in effect guesses a proof tree on a run. The size of the circuit is polynomial, hence the space needed by the simulating ATM is $O(\log n)$ and the bounded fan-in of AND gates results in polynomially bounded proof tree.

In this section, we describe the proof of the other direction of this, namely

ASPACE-TREE($O(\log n)$,poly(n)) is contained in SAC^1.

Informally, this result says the following: given a restricted long computation, one can perform a depth reduction and convert this computation into an efficient parallel computation.

We now describe the depth reduction technique used by [51]. Let A be a language accepted by an ATM M that uses space $O(\log n)$ and has polynomial proof tree size. We assume without loss of generality that every configuration of M has at most two successor configurations. We first design M' simulating M such that configuration trees produced by M' will be shallow. To describe the simulation we need two definitions.

For a fixed input x let $T_M(x)$ be a computation tree of M on x. A *fragment* of M on x is a pair (r, Λ) where, r is a configuration of M and Λ is a set of configurations of M. Intuitively, Λ signifies a set of conditions under which we check whether r accepts. The *size of a fragment* (r, Λ) is the minimal number of nodes in the subtree $T' \subseteq T_M(x)$ witnessing that (r, Λ) is realizable.

A fragment (r, Λ) is said to be *realizable* if there is a subtree T' of $T_M(x)$ having one child of non-leaf existential node, all children of non-leaf universal node and the properties that:

- The root of T' is r
- Each leaf of T' is either accepting or an element of Λ
- No element in Λ is accepting and all elements of Λ appear as leaves in T'

This says that (r, Λ) is realizable if under the assumption that Λ holds, all the leaves that belong to the subtree rooted at r but not in Λ are accepting. Keeping the elements of Λ exclusive of the accepting leaves is to avoid the overkill. Elements of Λ are leaves because we assume values of them and providing any information about them (in terms of subtree rooted at them) is redundant.

A polynomial size proof tree can possibly have polynomial depth. Intuitively, to convert a polynomial depth proof tree into a logarithmic depth tree, we will need to come up with a good cutting strategy. The following lemma will give such a strategy.

Lemma 3.2.1 (r, Λ) *is realizable if and only if at least one of the following holds:*

1. *There is a tree* T' *with at most* 3 *nodes witnessing that* (r, Λ) *is realizable.*

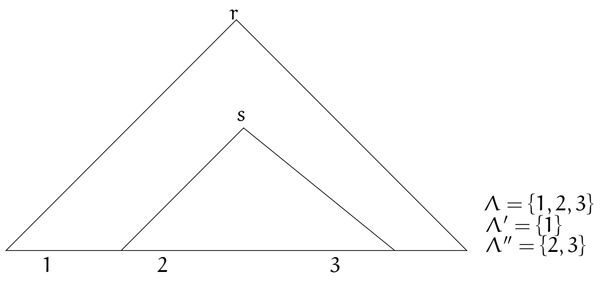


Figure 3.1: A possible configuration for $\Lambda, \Lambda', \Lambda''$ when $|\Lambda| = 3$.

2. (r, Λ) *can be divided into two fragments* $(r, \Lambda' \cup \{s\})$ *and* (s, Λ'') *that are both realizable and strictly of smaller size than* (r, Λ)*, where* $\Lambda = \Lambda' \cup \Lambda''$ *and* $\Lambda' \cap \Lambda'' = \emptyset$.

Proof: $(\Leftarrow)$: If 1 holds then trivially (r, Λ) is realizable. Else concatenate two trees T' and T'' witnessing realizability of $(r, \Lambda' \cup \{s\})$ and (s, Λ''), respectively, by replacing the leaf s in T' by the tree T''. This gives a tree witnessing realizability of (r, Λ).

$(\Rightarrow)$: Let T' be the tree witnessing realizability of (r, Λ). If it has at most three nodes then the first part of the lemma holds. Else there exists a non-leaf node $s \in T'$ which is not a root. Let Λ'' be those leaves in Λ which are descendants of s and $\Lambda' = \Lambda - \Lambda''$. Thus the second part of the lemma holds. □

Lemma 3.2.1 can be converted into an ATM M'. Let c_0 be the initial configuration of the ATM M. M' tries to prove that $(c_0, \emptyset)$ is realizable using the following algorithm:

M' *existentially* picks one of the following two possibilities

- M' tries to prove realizability of (r, Λ) by constructing a witnessing tree of size at most 3.
- M' *existentially* guesses s, Λ', Λ'',
  - verifies that $\Lambda = \Lambda' \cup \Lambda''$ and $\Lambda' \cap \Lambda'' = \emptyset$
  - *universally* starts two recursive calls **realize**$(r, \Lambda' \cup \{s\})$ and **realize**(s, Λ'')

The correctness follows from Lemma 3.2.1. We now analyze resources needed for M' and prove that $L(M')$ is in SAC^1. Note that the following two propositions hold:

Proposition 3.2.2 *The size of Λ need not be greater than three.*

Proof: When Λ has exactly three elements, there exists a node s which is a common ancestor of *exactly* two elements of Λ. (See for example Figure 3.1.) Let Λ'' consist of those two elements and let the third element of Λ be now called Λ'. Thus when Λ is of size three we have a way to reduce sizes of subsequent Λ' and Λ''. □

Proposition 3.2.3 *There is always a good cut that gives sufficiently small fragments.*

Proof: The classical tree-separator theorem states that: Let T be the tree with more than three nodes, say z nodes, where each node in T has at most two successors. There is a node v in T such that, if z_1 is the number of descendants of v including v and z_2 is the number of nodes in T which are not proper descendants of v then z_1, z_2 are at least one third and at most two third of z.

Such a v will be selected to get the desired *good* cut. We will have to make following two calls to the **realize** routine: **realize**$(r, \Lambda' \cup \{v\})$ and **realize**(v, Λ''). Note that Λ'' is a subset of Λ, and Λ' equals $\Lambda - \Lambda''$. □

Note that, Λ' may be equal to Λ before making the cut. Thus the size of Λ may increase by (at most) one in the next step after the cut.

As long as size of Λ is less than three, we can appeal to Proposition 3.2.3 and split the fragment according to the *good* cut. In the process the size of Λ might increase by one. When the size of Λ is three, we can appeal to Proposition 3.2.2 and split the fragment such that in the next step size of Λ is two. ($|\Lambda'| = 1, |\Lambda' \cup \{s\}| = 2, |\Lambda''| = 2$). Hence, at least in every other step we will appeal to Proposition 3.2.3 and reduce the fragment size by a constant fraction. Thus, logarithmic recursion depth will suffice. We have a circuit of depth $O(\log n)$. Now note that universal branches are made to **two** calls **realize**$(r, \Lambda' \cup \{s\})$ and **realize**(s, Λ''). Thus resulting circuit has unbounded fan-in OR gates, but only bounded fan-in AND gates and log-depth. This proves that $L(M')$ is in SAC^1.

Observe that, what we get is in fact a uniform SAC^1 circuit. The direct connection language for the circuit is in NC^1.

3.3 Block sorting, block merging, and block deletion

In this section, we will consider some problems arising from biology and will apply technique of depth reduction to them in order to obtain LogCFL upper bound.

3.3.1 Overview of the BLOCK SORTING problem

The BLOCK SORTING problem arises in optical character recognition, and is also a natural restriction of the well-studied genome rearrangement problem of SORTING BY TRANSPOSITIONS. Simply stated, the problem is as follows: given a string which is a permutation of n elements, bring it to the identity permutation $\mathtt{id}_n$ with as few block moves as possible. A block move consists of moving a block – a maximal substring that is also a substring of $\mathtt{id}_n$ – so that it merges with another block. For instance, in the string 6 2 3 4 1 5, the blocks are 6, 2 3 4, 1 and 5, and a block sorting sequence is 6 2 3 4 1 5 to 6 1 2 3 4 5 to 1 2 3 4 5 6.

Since attempts at designing polynomial time algorithms for BLOCK SORTING failed, various heuristics for the problem were considered [27, 33, 35]. Finally the problem was shown to be NP-hard by Bein *et al.* [11], and so attention shifted to efficient approximibility [1]. The structure of the problem guarantees a simple factor-3 approximation, and none of the earlier heuristics obtained a factor better than that. The first factor-2 approximation was devised by Mahajan *et al.* [36, 37]. It defined a related problem, BLOCK MERGING, and showed that (a) BLOCK MERGING can be solved optimally in P-time, and (b) BLOCK MERGING approximates BLOCK SORTING to within a factor of 2. The BLOCK MERGING algorithm runs in $O(n^3)$ time. Subsequently, another faster factor-2 approximation algorithm was devised by Bein *et al.* [12]. This considered the related problems of COMPLETE ABSOLUTE BLOCK DELETION and ABSOLUTE BLOCK DELETION, and showed that (a) ABSOLUTE BLOCK DELETION can be solved optimally in P-time, and (b) ABSOLUTE BLOCK DELETION approximates BLOCK SORTING to within a factor of 2. The ABSOLUTE BLOCK DELETION algorithm runs in $O(n^2)$ time. The two approximation algorithms are quite different in flavor, and are incomparable: there are infinite families of instances where one outperforms the other in terms of the actual approximation factor achieved [38]. A point in common to both is that the P-time implementation is via dynamic programming.

In Section 3.3.3, we prove the following theorem which improves the known upper bound for BLOCK MERGING and ABSOLUTE BLOCK DELETION:

Theorem 3.3.1 BLOCK MERGING *and* ABSOLUTE BLOCK DELETION *can be solved in*

[1]In contrast, the status of the SORTING BY TRANSPOSITIONS problem is still open – it is neither known to be in P-time nor known to be NP-hard. However, it has several factor 3/2 approximation algorithms [8, 24, 28] and most recently a factor-11/8 approximation algorithm [26].

LogCFL.

From the theorem it follows that there is an NC algorithm that approximates BLOCK SORTING within a factor of 2. But this does not immediately give an explicit NC algorithm. As we saw in Section 3.2, placing LogCFL in NC requires fairly intricate complexity-theoretic arguments. In Section 3.3.4, we unfold the depth reduction proofs of Venkateswaran and Vinay [51, 52] and specialize them to BLOCK MERGING and ABSOLUTE BLOCK DELETION to obtain explicit NC^2 algorithms for both. This provides concrete illustrative case studies for the depth reduction techniques of [51, 52]. Our algorithms are self-contained without any references to the complexity-theoretic results.

In the following section we will describe a few definitions and known lemmas regarding BLOCK MERGING and ABSOLUTE BLOCK DELETION.

3.3.2 Approximation algorithms for BLOCK SORTING

A *block* in permutation π is a maximal substring of π which is also a substring of the identity permutation $\mathtt{id}_n$. A *block move* in permutation π is the operation of picking a block of π and placing it elsewhere in the string so as to merge with another block. Let $\mathsf{bs}(\pi)$ be the minimum number of block moves required to obtain $\mathtt{id}_n$ from π.

Given a permutation π, it can be uniquely decomposed into maximal increasing substrings. $\mathcal{S}_\pi$ is the multiset $\mathcal{S}$ consisting of these substrings as sequences $S_1, S_2, ..., S_k$. Let $\mathsf{bm}(\mathcal{S})$ denote the minimum number of block moves needed to transform a given multiset $\mathcal{S} = \{S_1, S_2, ..., S_k\}$, of disjoint increasing sequences whose union is $[n]$, into a multiset $\mathcal{M}_n = \{\mathtt{id}_n, \epsilon, \epsilon, ..., \epsilon\}$. Here a block move places a block from one sequence into another sequence. For any given π, we can create an instance $\mathcal{S}_\pi$ of BLOCK MERGING.

Given a graph $G = (V, E)$, and an ordering on its vertices, edges (i, j) and (k, l) are said to be *crossing* if $i \leqslant k < j \leqslant l$ or $k \leqslant i < l \leqslant j$. A subset E' of E is said to be *non-crossing* set if no two edges of E' cross. Given a π we can obtain $\mathcal{S}_\pi = \{S_1, S_2, ..., S_k\}$. A graph $G_1 = (V_1, E_1)$ can be defined using $\mathcal{S}_\pi$ as follows: the vertex set is $[n]$. Two vertices u, v share an edge if there exists some sequence S_p in $\mathcal{S}_\pi$ such that both u and v are a part of S_p. The edge is directed from u to v if u appears before v in $\mathtt{id}_n$ and v to u otherwise. $V_1 = [n]$ and $E_1 = \{(u, v) \mid u < v, \exists p \in [k] : u, v \in S_p\}$. Let $\mathsf{ncs}(\pi)$ denote the size of the largest non-crossing set for this graph.

Given a permutation π, $\mathsf{abd}(\pi)$ equals the minimum number of block deletions needed to convert π into a single increasing sequence. $\mathsf{cabd}(\pi)$ equals the minimum number of block deletions needed to empty π.

The following are the key lemmas from [36, 37, 12] that will be used crucially in the later sections.

Lemma 3.3.2 *[36, 37]*

1. $\mathsf{bs}(\pi) \leqslant \mathsf{bm}(\mathcal{S}_\pi) \leqslant 2\mathsf{bs}(\pi)$
2. $\mathsf{bm}(\mathcal{S}_\pi) = n - \mathsf{ncs}(\pi)$
3. $\mathsf{ncs}(\pi) = c(1,n)$ *where:*

$$\begin{aligned}
&\textit{For } i \in [n], & c(i,i) &= 0\\
&\textit{For } i \in [n-1], & c(i,i+1) &= \begin{cases} 1 & \textit{if } (i,i+1) \in E \\ 0 & \textit{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\textit{For } 1 \leqslant i \leqslant j-2 \leqslant n-2 \\
&c(i,j) = \max\{t(i,j), q(i,j)\}\\
&t(i,j) = \begin{cases} 1 + c(i+1,j-1) & \textit{if } (i,j) \in E \\ 0 & \textit{otherwise} \end{cases}\\
&q(i,j) = \max_{i<k<j} c(i,k) + c(k,j)
\end{aligned}$$

Lemma 3.3.3 *[12]*

1. $\mathsf{bs}(\pi) \leqslant \mathsf{abd}(\pi) \leqslant 2\mathsf{bs}(\pi)$.
2. $\mathsf{abd}(\pi)$ *can be computed from* $\mathsf{cabd}(\pi)$ *in* $O(n^2)$ *time.*
3. $\mathsf{cabd}(\pi) = t(1,n)$ *where* $t(i,i) = 1$ *for all* i, $t(i,j) = 0$ *for all* $j < i$, *and for* $1 \leqslant i < j \leqslant n$,

$$t(i,j) = \begin{cases} \min\left\{ \begin{array}{c} 1 + t(i+1,j) \\ t(i+1,p-1) + t(p,j) \end{array} \right\} & \textit{if } i < p \leqslant j \textit{ where } \pi_p = \pi_i + 1 \\ 1 + t(i+1,j) & \text{otherwise} \end{cases}$$

3.3.3 BLOCK MERGING and ABSOLUTE BLOCK DELETION in LogCFL

LogCFL algorithm for BLOCK MERGING: Here we will give a AuxPDA-Time(poly) M for BLOCK MERGING. The input to the machine M is a diagraph G and $\langle 1, n, k\rangle$. M returns 1 if $c(1, n)$ is at least k and 0 otherwise. From Lemma 3.3.2 it is known that $\mathsf{ncs}(\pi)$ equals $c(1, n)$ where G is obtained appropriately from π. Hence this machine M will in fact answer the question $\mathsf{ncs}(\pi) \geqslant k$?

We will use the dynamic algorithm for computing $c(1, n)$ and give a AuxPDA simulation for it. We will argue that this AuxPDA will run for polynomial time. Hence, achieve a LogCFL bound.

```
1. Initialize work tape with $\langle 1, n, k\rangle$.
2. For any intermediate stage for tuple $\langle i, j, l\rangle$ on work tape,
   (We have to verify that $c(i, j) \geqslant l$.)
3.    **If** $((i = j)$ AND $l = 0)$ OR $([(j = i + 1)$ AND $(l = 1)$ AND $(i, j) \in E])$
4.    **Then**      **If** stack empty **Then** accept **Else** pop
5.    **Else**
         (a) guess $i \leqslant u < j$
         (b) **If** $[(u = i)$ AND $((i, j) \in E)]$ **Then** replace $\langle i, j, l\rangle$ by $\langle i + 1, j - 1, l - 1\rangle$
         (c) **Elseif** $[(u = i)$ AND $((i, j) \notin E)]$ **Then** abandon this path
         (d) **Elseif** $[(u \neq i)$ **Then** guess $0 \leqslant l' \leqslant l$, push $\langle u, j, l - l'\rangle$, write $\langle i, u, l'\rangle$.
```

The non-crossing set can also be explicitly constructed by making a minor modification to the above procedure. Change the AuxPDA algorithm at steps 3 (in the case where $i \neq j$) and 5(b) to output the edge (i, j). On any accepting path of this modified AuxPDA, the output edges put together will give a non-crossing set of size at least k.

The AuxPDA runs in polynomial time, because every unit interval (*i.e.,* the leaf of the proof tree) is visited at most once. Note that the polynomial time dynamic algorithm for computing $\mathsf{ncs}(\pi)$ had a structure such that max operator had unbounded fan-in while the $plus$ operator had bounded fan-in. Intuitively, for dynamic algorithms with such structure, there will be a hope to find a LogCFL algorithm. Of course, it will be essential that the number of smallest intervals that algorithm computes values of, will have to be polynomially many, without which the proof tree size will not be polynomially bounded.

LogCFL algorithm for ABSOLUTE BLOCK DELETION: We give a AuxPDA implementation for $\mathsf{cabd}(\pi) = t(1, n)$ that runs in polynomial time. This is merely a nondeterministic recursive implementation of the recurrence relation for $t(i, j)$, with the recursion records explicitly stacked onto the pushdown store. Observe that in the recurrence for $t(i, j)$, if $\pi_i + 1 = \pi_{i+1}$, then for each $j \geqslant i + 1$, $t(i, j)$ in fact equals $t(i + 1, j)$; we build this into the recursive program.

The runtime of the AuxPDA is $n^{O(1)}$ because on any computation path, the value of l in the triple (i, j, l) is partitioned non-trivially with each push-and-rewrite, and is initially at most n.

```
Input:    $\pi, k$
Output:   Yes if $\mathsf{cabd}(\pi) \leqslant k$, No otherwise.
Method:   Let $|\pi| = n$.
Initialization: Guess $1 \leqslant l \leqslant k$.
(Now verify that there is a complete block deletion sequence of length exactly $l$.)
    Write $(1, n, l)$ on work tape.
    Repeat forever: Let the tape hold $(i, j, l)$, where $i \leqslant j$ and $l \geqslant 1$.
            (Verify that $\pi_i, \dots, \pi_j$ can be emptied with $l$ deletions.)
        Find the largest $q \in \{i, ..., j\}$ such that $\pi_i ... \pi_q$ is a block in $\pi_i ... \pi_j$.
        If $q = j$ AND $l = 1$ Then
            pop stack top onto tape, overwriting $(i, j, l)$.
            (If stack is empty, accept and halt.)
        Elseif $(q < j$ AND $l = 1)$OR $(q = j$AND $l > 1)$ Then reject and halt.
        Else Let $p$ be such that $\pi_p = \pi_q + 1$. ($p$ is undefined if $\pi_q = n$.)
            If $p \in \{i, \dots, j\}$ Then
                nondeterministically choose $b \in \{0, 1\}$.
                If $b = 0$ Then replace tape with $(i, q, 1)$
                                  and push $(q + 1, j, l - 1)$.
                Else Guess a $1 \leqslant l' \leqslant l$
                    Replace tape with $(q + 1, p - 1, l')$
                                  and push $(p, j, l - l')$.
            Else replace tape with $(i, q, 1)$ and push $(q + 1, j, l - 1)$.
```

In the algorithm above, for $l = 1$, the tape is replaced with the tuple $(i, q, 1)$. This step is redundant in the above algorithm, however we use this crucially while designing an NC algorithm.

Now we come to absolute block deletion.

The method from [12] is as follows: construct a weighted DAG G with $V = \{0, 1, ..., n+1\}$, $E = \{(i,j) | \ i < j\}$, and for all $i < j$, $w(i,j)$=$t_{i+1,j-1}$. The weight of the minimum weight $0, n$ path in this graph equals $\mathsf{abd}(\pi)$. Note that all the edge weights in G are polynomially bounded (in fact, bounded by n, since $t(i,j)$ is at most $j - i + 1$).

In such a situation (poly bounded weighted DAG), the shortest path computation can be done in NL with oracle access to the edge weights. We have seen that computing all the values $t_{i,j}$ is in AuxPDA-Time(poly) = LogCFL. (Actually, we have seen that checking whether $t(i,j)$ at most k is in LogCFL. Checking equality follows from the fact that LogCFL is known to be closed under complementation and intersection, see [14, 53].) Combining the two in the obvious way, we get an AuxPDA-Time(poly) for $\mathsf{abd}(\pi)$; hence $\mathsf{abd}(\pi)$ can be computed in LogCFL.

3.3.4 Case studies: NC algorithms for BLOCK MERGING and ABSOLUTE BLOCK DELETION

NC algorithm for BLOCK MERGING**:** We now give a $O(\log n)$ depth circuit for $\mathsf{bm}(\pi)$. We use the construction of Venkateswaran [51] that converts a polynomial-sized proof tree into a SAC^1 circuit. The idea, as applied to block merging, can be stated as follows: Any accepting computation path of the AuxPDA actually constructs a non-crossing set of size, say, l. The construction of this set can be represented as a tree. Each internal node labeled (i,j) has two children; if the edge (i,j) is used in the set, then one child is trivially a leaf checking that this edge exists. An example is depicted in Figure 3.2. Clearly, this tree is of polynomial size. (In fact, it has exactly l leaves.) In such a tree rooted at r, the tree-separator theorem guarantees that there is always a unique node u which divides the tree in roughly equal sized parts. Once this node is located, the two sections of the tree can be evaluated in parallel: verify T_u, and verify T_r assuming T_u. (The assumption corresponds to a pruning of the tree.) If we repeatedly use this argument, then due to halving of the sizes at successive steps, total time to verify becomes logarithmic in the size of the tree. But we may need to keep track of too many pruned points. Therefore, we use this splitting

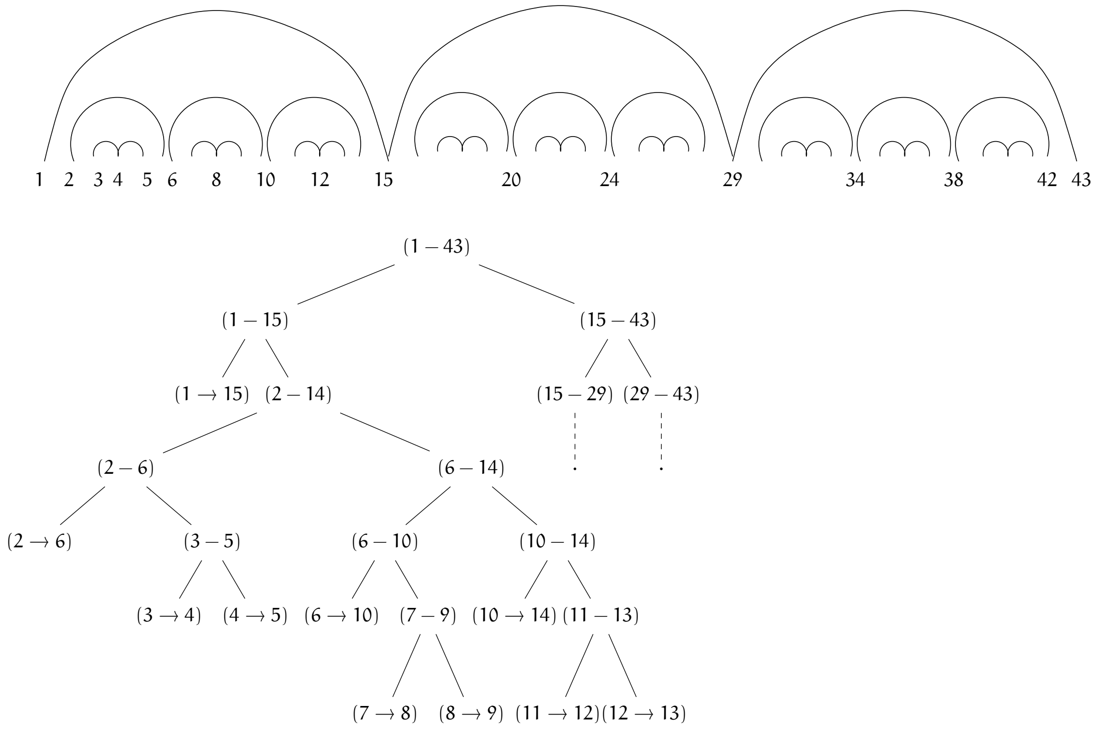


Figure 3.2: The construction tree for a non-crossing set

only if the number of cuts already made is not too large (in fact, at most 2). If it is 3, then we split not necessarily at the separator node, but at a node which will decrease the number of cuts in each part. Such a node always exists; simply take the node which is the least-common-ancestor of exactly two cut-points. This gives rise to a restructured tree for the same non-crossing set, but with depth $6 \log n$. (The non-balanced splitting is never used in two consecutive stages.) Figure 3.3 illustrates the restructured tree for the example of Figure 3.2.

Since we do not know the trees (or their splitting points) a priori, we check all possibilities; there are only polynomially many splitting points. Thus, along with each label (i, j, l), we also carry a parameter d, which acts as a time stamp when the gate gets activated. By the above argument, d ranges from 1 to $6 \log n$.

We design a circuit, for finding whether $\mathsf{ncs}(\pi) \geqslant k$ given π and k. The labels of the gates are of the form $((i, j, l) \mid \mathcal{H}, d)$, where $1 \leqslant i \leqslant j \leqslant n$, $0 \leqslant d \leqslant 6 \log n$, and $\mathcal{H}$ contains 0, 1, 2 or 3 triples of the form (i', j', l'). The tuples in $\mathcal{H}$ correspond to disjoint $(i'j')$ intervals, each contained in the interval (i, j), and are ordered left-to-right. The l' values add up to at most l.

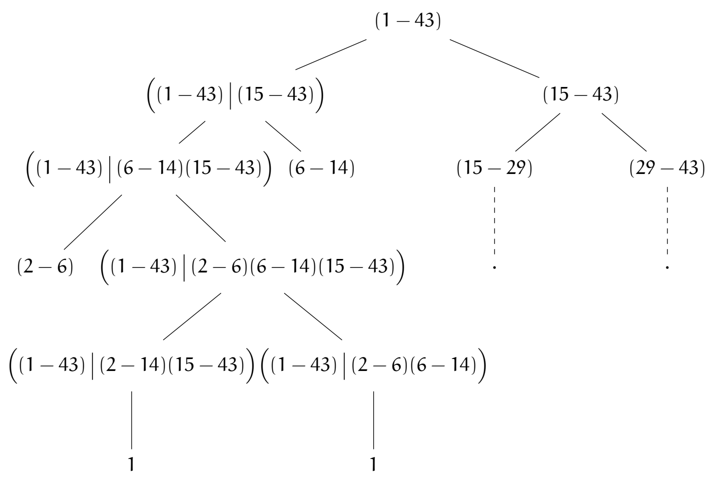


Figure 3.3: The restructured tree for constructing the non-crossing set of Figure 3.2

A gate labeled $((ijl), d)$ will try to find whether the interval (i, j) has a non-crossing set of size $\geqslant l$ (*i.e.,* $c(i, j) \geqslant l$). The gate labeled $((ijl) \mid \mathcal{H}, d)$ will try to find whether the interval (i, j) has a non-crossing set of size $\geqslant l$ under the hypothesis that the triples in $\mathcal{H}$ are true.

A gate labeled $((ijl) \mid \mathcal{H}, d)$ with $|\mathcal{H}| = t \leqslant 2$ can be evaluated using the values computed by gates with at most $t+1$ hypotheses at time $d-1$. Ranging over all new hypotheses $(ijl)'$, if $(ij)'$ subsumes some intervals of $\mathcal{H}$, then (ijl) can be verified with $(ijl)'$ replacing these intervals in $\mathcal{H}$, and $(ijl)'$ itself is verified assuming the subsumed intervals. If $(ij)'$ does not subsume any interval of $\mathcal{H}$, it can be added to $\mathcal{H}$, and verified independently.

A gate labeled $((ijl) \mid \mathcal{H}, d)$ with $|\mathcal{H}| = 3$ can be evaluated using the values computed by gates with 2 hypotheses at time $d - 1$. Ranging over all choices of $(ijl)'$, we want to verify (ijl) with the first two hypotheses replaced by the new choice, and the new hypothesis verified independently assuming the first two hypotheses of $\mathcal{H}$.

Formally, the gates' operations are as described below.

1. The gate labeled $((ijl), 1)$ is initialized to 1 if $l = 0$ or if $l = 1$ and (i, j) is an edge in G, and to 0 otherwise.

2. The gate labeled $((ijl) \mid \mathcal{H}, 1)$ is initialized to 1 if the l's in $\mathcal{H}$ add up to at least l, or if the l's in $\mathcal{H}$ add up to at least $l - 1$ and (i, j) is an edge in G, and to 0 otherwise.

3. The gate labeled $((ijl)|\ (ijl)_1(ijl)_2, d)$, checks all choices of a new hypothesis $(ijl)'$ which may subsume 0,1 or 2 of the given hypotheses. For each choice, it evaluates a bit as described below, and then computes the logical OR of all these bits.

   (a) $(ijl)'$ subsumes both hypotheses.

   $$\bigvee_{\substack{i\leqslant i'\leqslant i_1;\\ j_2\leqslant j'\leqslant j;\\ l_1+l_2\leqslant l'\leqslant l}} \left[\Big((ijl)\,\Big|\,(ijl)', d-1\Big) \wedge \Big((ijl)'\,\Big|\,(ijl)_1(ijl)_2, d-1\Big)\right]$$

   A gate performs this big (with polynomial inputs) $\vee$ in $O(1)$ depth (or in depth $O(\log n)$ using small (with $O(1)$ inputs) $\vee$ gates).

   (b) $(ijl)'$ subsumes first hypothesis.

   $$\bigvee_{\substack{i\leqslant i'\leqslant i_1;\\ j_2\leqslant j'\leqslant j;\\ l_1+l_2\leqslant l'\leqslant l}} \left[\Big((ijl)\,\Big|\,(ijl)'(ijl)_2, d-1\Big) \wedge \Big((ijl)'\,\Big|\,(ijl)_1, d-1\Big)\right]$$

   (c) $(ijl)'$ subsumes second hypothesis. Similar to above case.

   (d) $(ijl)'$ subsumes none of the hypotheses and $(ijl)'$ is on left of $(ijl)_1$

   $$\bigvee_{\substack{i\leqslant i'\leqslant i_1;\\ i'\leqslant j'\leqslant i_1;\\ 0\leqslant l'\leqslant l-(l_1+l_2)}} \left[\Big((ijl)\,\Big|\,(ijl)'(ijl)_1(ijl)_2, d-1\Big) \wedge ((ijl)', d-1)\right]$$

   (e) $(ijl)'$ subsumes none of the hypotheses and $(ijl)'$ is on right of $(ijl)_2$. Similar to above case.

   (f) $(ijl)'$ subsumes none of the hypotheses and $(ijl)'$ is between $(ijl)_1$ $(ijl)_2$. Similar to above case.

4. The gates with labels $((ijl)|\mathcal{H})$ where $|\mathcal{H}|$ is 0 or 1 can be described similarly.

5. For the gate with the label $((ijl)|\ (ijl)_1(ijl)_2(ijl)_3, d)$, guess a new hypothesis $(ijl)'$ necessarily subsuming exactly 2. For $(ijl)'$ subsuming $(ijl)_1$ and $(ijl)_2$

the expression is:

$$\bigvee_{\substack{i \leqslant i' \leqslant i_1; \\ j_2 \leqslant j' \leqslant i_3; \\ l_1 + l_2 \leqslant l' \leqslant l - l_3}} \left[\Big((ijl) \,\Big|\, (ijl)'(ijl)_3, d-1 \Big) \wedge \Big((ijl)' \,\Big|\, (ijl)_1 (ijl)_2, d-1 \Big) \right]$$

Similar expression can be obtained when $(ijl)'$ subsumes $(ijl)_2$ and $(ijl)_3$.

6. The output is the value $\bigvee_{d=1}^{6 \log n}((1, n, k), d)$.

We can also carry another parameter *ptr* in the labels of the gates. It will hold a pointer to the gate-pair which yields the non-crossing set corresponding to the current interval. With this book-keeping, we can also explicitly extract the non-crossing set by an $O(\log n)$ depth circuit.

NC **algorithm for** ABSOLUTE BLOCK DELETION**:** To obtain a parallel algorithm from cabd( )we use the depth-reduction proof idea originally due to [52], and generalized in [4], see also [53]. The idea, as specialized to the problem of computing cabd(π), can be described as follows: Any accepting computation path of the AuxPDA constructs a specific complete block deletion sequence. The construction can be depicted as a tree: the block deletion sequence of length l for $\pi_i \dots \pi_j$ is comprised of two sequences which can be independently constructed. (One replaces the tape, the other is stacked.) For instance, for $\pi = 5\ 6\ 2\ 7\ 1\ 3\ 8\ 4$, Figure 3.4 shows the trees corresponding to the two 5-move sequences $(56, 7, 1, 8, 234)$ and $(2, 567, 1, 8, 34)$. Here a node labeled (i, j, l) checks the predicates asserting that the sequence $i \dots j$ can be emptied with at most l block deletions. Clearly, any such tree is binary (each internal node has exactly two children), has l leaves if it witnesses an l-length sequence, and can have depth up to $O(l)$. (Each leaf corresponds to a block deletion.)

Since the two children of a node can be verified in any order, consider a reordering of the tree such that at each internal node, the heavier child (with more leaves in its subtree) appears as the right child of its parent. This implies that any root-to-leaf path has at most $\log n$ left moves. Along the rightmost path (called the spine), we can identify an edge $(w \rightarrow u)$ where the number of subtree-leaves first drops below $l/2$. Let u's other (lighter) child be v. Let $P(x)$ denote the predicate

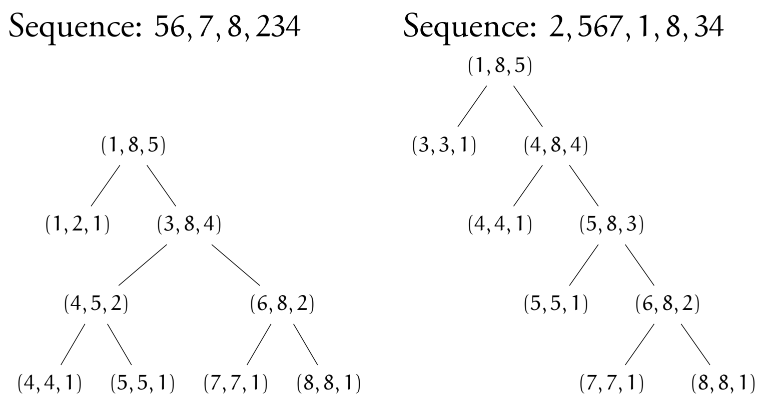


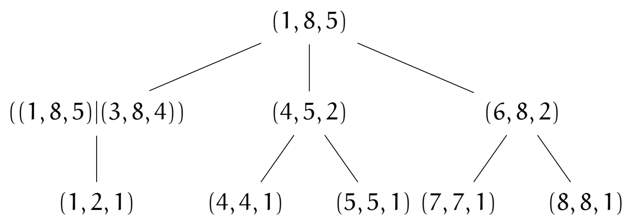
Figure 3.4: The construction trees for two complete block deletion sequences for 5 6 2 7 1 8 3 4

being checked at node x. Then, the predicate at the root r can be checked via $P(r) = (P(r)$ given $P(u))$AND $P(v)$AND $P(w)$, since $P(v)$ and $P(w)$ together imply $P(u)$. And node v is not on the spine (it is at a distance of 1 from the spine). Repeatedly divide the segments of the spine, with division points chosen to halve the number of leaves, until verifying $P(r)$ is reduced to verifying other predicates at distance 1 from the spine. This results in a restructuring of the tree, and yields a new tree of depth at most $O(\log n)$. Figure 3.5 shows the restructured trees corresponding to those in Figure 3.4.

Unfortunately, the trees themselves are not known to us a priori, let alone the optimal division points for restructuring. So we check all possible division points that may work for a potential unknown tree. There are only polynomially many such points (guess the three triples corresponding to nodes u, v, w), hence this is feasible. We abandon a particular choice if it leads to the construction of a path with more than $\log n$ left moves. So we also need to carry this value along with the node label.

The circuit can now be described as follows: Initially, compute and store π^{-1}. Then compute and store, for each $i \leqslant j$, the largest $q = q(i,j)$ such that $q \in \{i, ..., j\}$ and $\pi_i ... \pi_q$ is a block in $\pi_i ... \pi_j$. Also, compute and store $p = p(i,j)$ such that if $\pi_q = n$ then p is undefined; else let $\pi_r = \pi_q + 1$, and $p = r$ if $q + 1 \leqslant r \leqslant j$, undefined otherwise.

Assign a gate to each label $[i, j, l, r]$ where $i \leqslant j$, $0 \leqslant r \leqslant \log n$, $1 \leqslant l \leqslant k$. Also, assign a gate to each label $([i, j, l, r] \mid [i_1, j_1, l_1, r])$ where $i \leqslant i_1 \leqslant j_1 \leqslant j$, $\{i, j\} \neq \{i_1, j_1\}$, Sequence: 56, 7, 8, 234



Sequence: 2, 567, 1, 8, 34

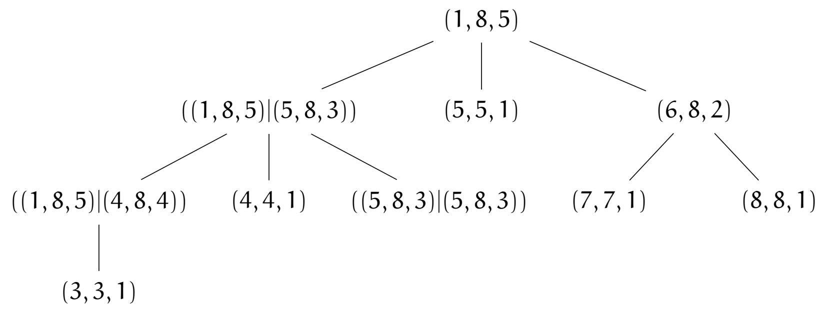


Figure 3.5: The restructured trees for constructing the complete block deletion sequences of Figure 3.4.

and $1 \leqslant l_1 \leqslant l$. These are the "valid labels" for gates that we refer to below. In the description below if label of a gate is not valid, it is to be read as a constant gate with value 0. These gates function as follows:
$[i,j,l,r]$ = True if $q(i,j) = j$ and $l = 1$,
$[i,j,l,r]$ = False if $[q(i,j) = j \wedge l > 1] \vee [q(i,j) < j \wedge l = 1]$, and otherwise

$$[i,j,l,r] = \bigvee \Big([i,j,l,r] \,\Big|\, [i_u,j_u,l_u,r]\Big) \wedge [i_v,j_v,l_v,r+1] \wedge [i_w,j_w,l_w,r]$$

where the $\vee$ is over all triples $[i,j,l]_u, [i,j,l]_v, [i,j,l]_w$ satisfying

- $[i,j,l,r] \mid [i_u,j_u,l_u,r]$ is a valid label,
- $l_v \leqslant l_w \leqslant l/2$ and $l_v + l_w = l_u \geqslant l/2$
- $[i_v,j_v,l_v]$ and $[i_w,j_w,l_w]$ can possibly be children of $[i_u,j_u,l_u]$ in some tree (that is, the intervals $[i_v,j_v]$ and $[i_w,j_w]$ lie inside $[i_u,j_u]$ and cover it, except possibly for a prefix $[i_u, q(i_u,j_u)]$).

Similarly, $\Big([i,j,l,r] \,\Big|\, [i',j',l',r]\Big)$ = True if $[i,j,l] = [i',j',l']$, and

$\bigvee \Big([i,j,l,r] \,\Big|\, [i_u,j_u,l_u,r]\Big) \wedge [i_v,j_v,l_v,r+1] \wedge \Big([i_w,j_w,l_w,r] \,\Big|\, [i',j',l',r]\Big)$; otherwise

where the $\vee$ is over all triples $[i,j,l]_u, [i,j,l]_v, [i,j,l]_w$ satisfying

- $[i,j,l,r] \mid [i_u,j_u,l_u,r]$ and $([i_w,j_w,l_w,r] \mid [i',j',l',r])$ are valid labels,
- $l_v \leqslant l_w$, $l_v + l_w = l_u$ and $l_w - l' \leqslant (l-l')/2 \leqslant l_u - l'$.
- $[i_v,j_v,l_v]$ and $[i_w,j_w,l_w]$ can possibly be children of $[i_u,j_u,l_u]$ in some tree.

Finally, a new gate evaluates and outputs the value $\vee_{l=1}^{k}(1,n,l,0)$.

This procedure will check whether $t(1,n) \leqslant k$. By invoking this procedure in parallel on the n distinct choices for k, we obtain the smallest k for which this holds, namely, $t(1,n)$ itself. (Also, by exiting at appropriate stages (or making more copies), $t(i,j)$ can be evaluated for each $i \leqslant j$.)

Clearly, this algorithm can be implemented by a depth $O(\log n)$ circuit. A little more, straightforward, book-keeping, allows us to also extract the witnessing complete block deletion sequence within the same depth.

Now we come to absolute block deletion. As described in Section 3.3.3, this is in NL, once the $t(i,j)$ values are computed. To uncover an NC algorithm for this stage, we need not use the complex depth reduction for LogCFL. NL is in NC by a classical, far simpler, argument due to Savitch, see for instance [47]. Applied here, it yields the following algorithm:
Assign a gate to each triple $[i,j,w]$ where $i,j,w \in \{1,\ldots,n\}$; it is expected to return true if and only if there is a path of weight at most w from i to j. Then

$[i,j,w]$ = True if $i = j$, and otherwise

$[i,j,w] = \bigvee_{i \leqslant k \leqslant j} \left([i,k,\lfloor \frac{w}{2} \rfloor] \wedge [k,j,\lceil \frac{w}{2} \rceil]\right)$

Clearly, this can be implemented by SAC^1 circuit (in $O(\log n)$ depth). (This is another way of saying that NL is in SAC^1.)

Hardness results: For BLOCK MERGING the only hardness known is TC^0 hardness. (The hardness is due to reduction to membership problem for Dyck sets.) It would be nice to know (in the best case) LogCFL hardness or hardness for some slightly larger class, say L or LogDCFL. Unfortunately, none of these are known.

In the following section, we will see another problem for which we will prove an upper bound of LogCFL. This problem is also known to be hard for LogCFL. Thus, this problem is more interesting for us.

3.4 Multi-pushdown machines

3.4.1 Membership problem

In this section and in the next chapter, we will consider a problem called the *membership problem*. Consider a fixed machine M (the machine can be for example a finite state automaton, or a pushdown automaton etc.) over an alphabet Σ. Typically, the machine consists of a finite control, a finite set of rules that describe the way the machine moves from one configuration to another reading an input from Σ^*. The machine also has marked initial and final configurations. If on an input string $w \in \Sigma^*$, the machine starts from one of its initial configurations and ends up in one of the final configurations then w is said to be *accepted by the machine* M. The set of strings from Σ^* that are accepted by the machine is called the *language of the machine*, denoted as $L(M)$.

The membership problem for a fixed M, deals with given a input string $w \in \Sigma^*$, deciding whether w is in $L(M)$. Let $\mathcal{A}$ be a set of machines/automata. And let $\mathcal{L}$ be a set of languages accepted by $\mathcal{A}$. We use either $\mathsf{MEM}(\mathcal{A})$ or $\mathsf{MEM}(\mathcal{L})$ to denote the membership problem for the machines in $\mathcal{A}$.

The membership problem is interesting for two different reasons. Firstly, it is well motivated from the application point of view. The membership problem is also known as the parsing problem. The parsing problem has been of interest since the time of automation back in the seventies [31]. For various different machine models, this problem has been studied. Efficient algorithms for various classes of machines have been developed. Secondly, the problem is of interest from complexity theory point of view. For many machine models, the problem becomes complete for a complexity class. (We will elaborate on the connection between membership problem and complexity classes in the next chapter.)

The parsing community only needs that useful class of machine models $\mathcal{A}$ have polynomial time algorithms for membership testing. However, it is fascinating for complexity theorists to obtain equivalent characterizations for known complexity classes. The complexity class of our interest, namely LogCFL, has been characterized by the membership problem for CFLs, MEM(CFL). It is known that MEM(CFL) is complete for LogCFL [48].

In this section, we consider membership problem for machines that consist of multiple pushdowns along with finite control. In general, with two stacks one can simulate any Turing machine. The head movement of the machine is captured at the top of the stacks. But the model that we consider here is somewhat restricted. And with this added restriction, the membership problem becomes much easier. In fact, we get a LogCFL upper bound. This is another illustration of the depth reduction technique.

3.4.2 Definitions and known results

A PD_k $M = (Q, q_0, F, \Gamma, \Sigma, \delta, Z_0)$ is a k-stack pushdown machine where Q is a finite set of states, q_0 is the start state, $F \subseteq Q$ is a set of final states, Γ is a finite set of stack alphabet, Σ is a finite input alphabet, Z_0 (this component is different from the standard definition of a PDA described in Section 2.1) is the bottom of the stack marker, 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, $\langle q, w, \gamma_1, \dots, \gamma_k\rangle$ 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 a word x is $\langle q_0, x, Z_0, \epsilon, \dots, \epsilon\rangle$. A configuration is called a *final* configuration if q is in F.

If there is a transition $(q', \alpha_1, \dots, \alpha_k) \in \delta(q, a, A)$, 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,
$\langle q, aw, \epsilon, \dots, \epsilon, A\gamma_i, \dots, \gamma_k\rangle \vdash \langle q', w, \alpha_1, \dots, \alpha_{i-1}, \alpha_i\gamma_i, \dots, \alpha_k\gamma_k\rangle$.

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

The PD_k M accepts a string w if it can move from $\langle q_0, w, Z_0, \epsilon, \dots, \epsilon\rangle$ to some $\langle q, \epsilon, \gamma_1, \dots, \gamma_k\rangle$ where $q \in F$. The set of all the strings accepted by M is the language accepted by M, denoted $L(M)$. This model was defined by Cherubini *et al.* in [20]. The membership problem for PD_k was considered by Cherubini *et al.* [22]. They proved the following:

Theorem 3.4.1 *([22]) For a fixed* PD_k*, given an input string* $w \in \Sigma^*$*, checking if* $w \in L(M)$ *is in* P*-time. i.e.,* $\mathsf{MEM}(\mathsf{PD}_k) \in \mathsf{P}$*-time.*

In [21], 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 $L(G) = \{w \mid S \Rightarrow^* w\}$.

Example 3.4.2 *The language* $\{a^n b^n c^n \mid n \geqslant 0\}$ *is accepted by the following* D^2 *grammar:* $G = (N, \Sigma, P, S)$*, where* $N = \{S, B, C\}$*,* $\Sigma = \{a, b, c\}$*, and* P *consists of the following productions:* $S \to a(SB)(C)|\epsilon$*,* $B \to b(\epsilon)(\epsilon)|\epsilon$*, and* $C \to c(\epsilon)(\epsilon)|\epsilon$*.*

The initial sentential form is $(S)(\epsilon)$*. And a typical derivation can be given as follows:* $(S)(\epsilon) \to a(SB)(C) \Rightarrow^{k-1} a^k(SB^k)(C^k) \to a^k(B^k)(C^k) \Rightarrow^k a^k b^k(\epsilon)(C^k)$ $\Rightarrow^k a^k b^k c^k(\epsilon)(\epsilon)$*, where* $\Rightarrow^k$ *indicates* k *derivation steps.*

Theorem 3.4.3 *([21]) For every* D^2*-grammar* G *there is 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 occurrence of any variable participates in two chain rules. For every derivation, there is an equivalent normal form derivation [21].

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 β 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 $A \to a$. 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).

3.4.3 Reviewing the P-time algorithm for MEM(PD_2)

The main result we intend to establish is the following theorem:

Theorem 3.4.4 *For every fixed* $k \geqslant 1$, MEM(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 [21]. So in this section, we first describe the P-time algorithm in some detail following the presentation from [43]. We then give (Section 3.4.4) a different implementation of the same algorithm and improve the upper bound to LogCFL, thus establishing Theorem 3.4.4 for $k = 2$. A P-time algorithm for MEM(PD_k) is given in [22]. It is very similar to the algorithm from [21]. In Section 3.4.5, we discuss the changes needed to be made in our implementation for the LogCFL bound to hold for all fixed k, thereby proving Theorem 3.4.4.

The P-time algorithm uses the characterization of PD_2 via D^2 grammars in normal form, and normal-form derivations.

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 is 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 $i + r + s \leqslant j$ (*i.e.,* $r \leqslant l'$).

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 $0 \leqslant r' \leqslant j - i$.

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 $0 \leqslant r' \leqslant j - i$. 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 $r' \neq 0$. 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 $O(n^6)$.

3.4.4 LogCFL **algorithm for** MEM(PD$_2$)

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 [45, 48] 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 $l' \leqslant l \leqslant n$, 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 labeled $\langle S, 0, n, 0, 0\rangle$. The circuit connections are as follows:

$$\langle A, i, j, r, s\rangle = \bigvee_{k\in[4]} X^k_{A,i,j,r,s}$$

$$X^1_{A,i,j,r,s} = \bigvee_{\substack{r' < r \\ s < s' < j-i-r+1 \\ \{B,C \mid A \to (BC)(\epsilon)\}}} \begin{pmatrix} \langle B, i, j, r', s'\rangle \wedge \\ \langle C, i+r', i+r'+s', r-r', s\rangle \end{pmatrix}$$

$$X^2_{A,i,j,r,s} = \begin{cases} \langle A, i+s, j, 0, 0\rangle & \text{if } r = 0 \\ 0 & \text{otherwise} \end{cases}$$

$$X^3_{A,i,j,r,s} = \begin{cases} \langle A, i, j-s, r, 0\rangle & \text{if } i+r = j-s \\ 0 & \text{otherwise} \end{cases}$$

$$X^4_{A,i,j,r,s} = \begin{cases} \bigvee_{0\leqslant r'\leqslant j-i, \{B \mid A\to(\epsilon)(B)\}} X^1_{B,i,j,r',0} & \text{if } r, s = 0 \\ 0 & \text{otherwise} \end{cases}$$

This finishes the description of all the non-leaf gates.

A predicate $[i, a, 1, 0]$ takes value 1 if ith symbol of the input is the letter a, 0 otherwise. We define such predicates in order to describe the input gates of the circuit as done in Allender *et al.* [4]. The values of the input gates are propagated via the following depth-1 circuit.

$$\langle A, i, i+1, 1, 0\rangle = \bigvee_{\{a \mid (A\to a \in P)\}} [i, a, 1, 0]$$

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

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 3.4.5 $\langle A, i, j, r, s\rangle = 1$ *if and only if* $A \in T(i, j, r, s)$.

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

Claim 3.4.6 *The circuit constructed above has polynomial size proof trees.*

Proof: The leaves of the proof tree are unit intervals representing a letter from the input string. The number of distinct unit intervals is equal to the length of the input and every unit interval appears at most once in the proof tree. This gives a polynomial size proof tree. □

It is easy to see that the circuit is uniform.

3.4.5 LogCFL algorithm for MEM(PD_k)

The grammars [20] 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:

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

Now, any typical string derived by a non-terminal can have as many as 2^{k-1} gaps; see [22]. If $(A)_h \to (BC)_1$ 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 3.4.4.

3.5 Concluding Remarks

In this chapter we reviewed a depth reduction technique by [51]. Using the ideas of depth reduction, we obtained new LogCFL upper bounds for

- BLOCK MERGING
- ABSOLUTE BLOCK DELETION
- $\mathsf{MEM}(\mathsf{PD}_k)$.

As case studies we unfolded the depth reduction from [51, 52] to explicitly describe NC^2 algorithms for BLOCK MERGING and ABSOLUTE BLOCK DELETION.

4 Membership problem for generalization of VPLs

4.1 Why study the membership problem?

In this chapter, we will consider the membership problem for various restrictions of CFLs and some restrictions of context-sensitive languages and show improved upper bounds within LogCFL.

The study conducted here is of interest from the complexity theoretic point of view. Here we characterize various complexity classes (between NC^1 and NC^2) by the membership problem for subclasses of CFLs. In the past, $MEM(\mathcal{L})$ for the class of languages that are proper restrictions of CFL have been studied. They are known to characterize known complexity classes; *i.e.,* they can be solved using resource bounds corresponding to a complexity class for which they are also hard.

From the years of study, a lot is known about the structure of the language classes. The fact that the membership problem for these language classes has a connection to known complexity classes is very useful. This is because, as compared to language classes, a lot less is known about complexity classes. The hope, therefore is, to make connections between language classes and complexity classes to understand the structure of the complexity classes.

Recall that we use $\mathcal{A}$ to denote a set of machines/automata, $\mathcal{L}$ to denote a set of languages accepted by $\mathcal{A}$. We use either $MEM(\mathcal{A})$ or $MEM(\mathcal{L})$ to denote the membership problem for the machines in $\mathcal{A}$. We will relate the complexity of $MEM(\mathcal{L})$ for these language classes to complexity classes. The complexity classes of inter-

$$\begin{tikzcd}
\boxed{\begin{array}{l}\mathsf{NC}^1,\\ \mathsf{MEM(REG)}\end{array}} \arrow[r] & \boxed{\begin{array}{l}\mathsf{L},\\ \mathsf{MEM(DLIN)}\end{array}} \arrow[r] \arrow[dr] & \boxed{\begin{array}{l}\mathsf{LogDCFL},\\ \mathsf{MEM(DCFL)}\end{array}} \arrow[dr] & \\
 & & \boxed{\begin{array}{l}\mathsf{NL},\\ \mathsf{MEM(LIN)}\end{array}} \arrow[r] & \boxed{\begin{array}{l}\mathsf{LogCFL},\\ \mathsf{MEM(CFL)}\end{array}}
\end{tikzcd}$$

Figure 4.1: Connection between MEM($\mathcal{L}$) and complexity classes

est to us are those between NC^1 and LogCFL. The containment relations between the complexity classes we consider are shown in Figure 2.3. Note that none of the containments here are known to be strict.

Recall that MEM(CFL) is complete for for the class LogCFL [48]. The set of languages logspace many-one reducible to MEM(DCFL) define the complexity class LogDCFL which is a subclass of LogCFL [48]. Sudborough [49] proved that membership problem for LIN languages, MEM(LIN), is complete for NL. Holzer *et al.* [30] proved that MEM(DLIN) (where the definition of DLIN is acceptance by DPDA$_{1\text{-turn}}$) is complete for logspace. Barrington [10] proved a similar correspondence between MEM(REG) and NC^1. (Refer to sections 2.1, 2.2.1, and 2.2.3 to recall the definitions of the above language classes and complexity classes.)

The known connections between membership problem for subclasses of CFLs and complexity classes are depicted in Figure 4.1.

4.2 Basic facts about VPLs and IDLs

Recently, many new subclasses of CFLs have been defined. Visibly pushdown languages, denoted as VPL, is one such subclass of CFLs. VPLs were first defined by Mehlhorn [40] and studied for the complexity of MEM(VPL). But later, they were rediscovered by Alur *et al.* [5]. They mainly studied language theoretic properties of VPLs. Informally, VPLs are the languages accepted by pushdown machines called visibly pushdown automata (VPA) which are ϵ-moves-free pushdown automata whose stack behavior (push/pop/no change) is dictated solely by the input letter under consideration.

More formally, VPA can be defined as follows:

Definition 4.2.1 (Visibly pushdown automaton) *A* visibly pushdown automaton *(*VPA*) over an input alphabet* Σ *is a* PDA $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$. *Here,* Q *is a finite set of states,* q_0 *is an initial state,* $F \subseteq Q$ *is a set of final states,* Γ *is a finite stack alphabet which contains a special bottom-of-stack marker* $\perp$*, the finite input alphabet* Σ *is partitioned as* $\Sigma_c \cup \Sigma_r \cup \Sigma_i$*, and transition relation is a finite set* δ *contained in the union of* $(Q \times \Sigma_c \times Q \times \Gamma \setminus \{\perp\})$, $(Q \times \Gamma \times \Sigma_r \times Q)$, *and* $(Q \times \Sigma_i \times Q)$. *On reading a letter from a particular part of the input alphabet, the* PDA *is allowed to make exactly one of the three moves: push, pop, and no change.*

VPA *has no* ϵ*-moves. The language accepted by* VPA M *is denoted as* $L(M)$. *The languages accepted by* VPA *are called visibly pushdown languages,* VPL*s.*

A VPA *is deterministic if the* δ *relation is a (possibly partial) function.*

The following two theorems state known interesting properties of VPA.

Theorem 4.2.2 (Determinisation [5]) *Given a* VPA M *over an input alphabet* Σ *with the three partitions* Σ_c, Σ_r, *and* Σ_i, *there exists a deterministic* VPA M' *over the same alphabet and the same partition of the alphabet such that* $L(M) = L(M')$.

Theorem 4.2.3 (Boolean Closure [5]) *Let* L_1 *and* L_2 *be two* VPL*s accepted by* VPA M_1 *and* M_2 *respectively over the same tri-partitioned input alphabet* Σ. *Then* $L_1 \cup L_2$, $L_1 \cap L_2$, $\overline{L_1}$, $L_1 \cdot L_2$, *and* L_1^* *are also* VPL*s accepted by* VPA *over the same tri-partitioning of* Σ.

The input driven PDA as defined by [40] are slightly different from VPA.

Definition 4.2.4 (Input driven PDA) *An input driven pushdown automaton* $P = (Q, q_0, F, \Gamma, \Sigma, \delta, Z_0)$ *is a realtime* PDA *with tri-partitioned input alphabet* $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i$. Z_0 *is a special bottom-of-stack marker not contained in* Γ*, and transition relation* δ *is a finite set contained in the union of* $(Q \times \Gamma \times \Sigma_c \times Q \times \Gamma^2)$, $(Q \times \Gamma \times \Sigma_r \times Q)$, *and* $(Q \times \Gamma \times \Sigma_i \times Q \times \Gamma)$.

There are two primary differences between the definitions of input driven PDA and VPA.

- In all the moves of an input driven PDA, the transitions may depend on the current stack-top unlike in the case of VPA where they are allowed to depend on the stack-top only during the pop moves. That is, VPA are weak PDA but input driven PDA are not. The latter has rules of the form $pU \to qV$, where $|U| = 1$ and $|V| \leqslant 2$.

But by remembering the stack-top in the state for push and local moves, one can easily obtain (by using standard techniques from [31]) a VPA from an input driven PDA.

- The transition function of a VPA is allowed to have transitions of the form $(p\ \bot) \xrightarrow{a} (q\ \bot)$, for a in Σ_r *i.e.,* a VPA is allowed to "pop" on an empty stack, however an input driven PDA is not. Due to this restriction, IDL are a proper subclass of VPLs. Example 4.2.5 gives a language that is a VPL but not an input driven language, IDL.

  Example 4.2.5 *A set of strings of the form* $\{(a^n b^{n+1})^* \mid n \geqslant 0\}$ *is accepted by a* VPA *but not by any input driven* PDA.

A string over a tri-partitioned alphabet is called *well-matched*, if every prefix of the string has at least as many push letters as pop letters, and the total number of push letters in the string equals the total number of pop letters.

A VPA that accepts only well-matched strings, does not pop on an empty stack. Let wm(VPL) denote the set of languages accepted by such VPA. Clearly, wm(VPL) can be accepted by input driven PDA. Note that there are input driven languages which contain strings that are not well-matched. For example $\{a^m b^n \mid m \geqslant n\}$ is an IDL. Thus, wm(VPL) $\subset$ IDL. And we know that IDL $\subset$ VPL.

The following lemma tells us that it is enough to consider wm(VPL) as long as TC^0 computations are allowed.

Lemma 4.2.6 *For every* VPA M *over alphabet* Δ*, there is a corresponding* VPA M' *over an alphabet* Δ' *and a* TC^0 *many-one reduction* g *such that for every* $x \in \Delta^*$*,*

1. $\#acc_M(x) = \#acc_{M'}(g(x))$, *and*

2. $g(x)$ *is well-matched.*

Proof: Let $M = (Q, \Delta, Q_{in}, \Gamma, \delta, Q_F)$. The VPA $M' = (Q', \Delta', Q'_{in}, \Gamma', \delta', Q'_F)$ is essentially the same as M. It has two new input symbols A, B, and a new stack symbol X. A is a push symbol on which X is pushed, and B is a pop symbol on which X is expected and popped. M' has a new state q' that is the only initial state. M' expects an input from $A^*\Delta^*B^*$. On the prefix of A's it pushes X's. When it sees the first letter from Δ, it starts behaving like M. The only exception is when M performs a pop move on $\perp$, M' can perform the same move on $\perp$ or on X. On the trailing suffix of B's it pops X's. It is straightforward to design δ' from δ.

Let $|x| = n$. The TC^0 circuit does the following. It counts the difference d between the number of push and pop symbols in $A^n x$. It then outputs $y = A^n x B^d$. By the way M' is constructed, it should be clear that $\#\mathsf{acc}_M(x) = \#\mathsf{acc}_{M'}(y)$ and that M', on y, never pops on an empty stack. In fact y is well-matched. □

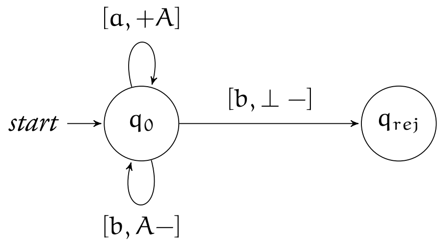
As a corollary we get the following:

Corollary 4.2.7 TC^0 *closures of* $wm(\mathsf{VPL})$*,* IDL*, and* VPL *are equivalent, i.e.,* $\mathsf{TC}^0(wm(\mathsf{VPL})) = \mathsf{TC}^0(\mathsf{IDL}) = \mathsf{TC}^0(\mathsf{VPL})$*.*

Henceforth, we do not differentiate between $wm(\mathsf{VPL})$, IDLs, and VPLs as long as we are allowed to use TC^0 reductions.

REG is contained in VPL for any partition of the input alphabet. The accepting VPA ignores the stack and simulates the NFA moves by its finite control. Given below are two examples of VPLs that are known to be non-regular.

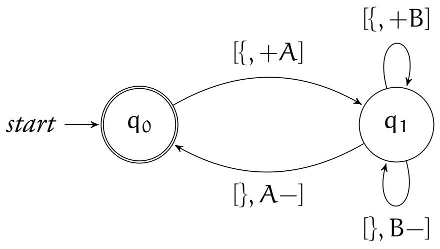
Example 4.2.8 *The language* $\{a^n b^n \mid n \geqslant 0\}$ *is accepted by a* VPA *for which the partition of the alphabet is:* $\Sigma_c = \{a\}$, $\Sigma_r = \{b\}$ *, and* $\Sigma_l = \phi$.



Here, we indicate the transitions between the states as in the case of an NFA. The changes on the stack are indicated by means of arrows. In particular, on reading a push letter, the letter to be pushed is indicated with a + sign on the arrow which points to the state reached after such a push. On reading a pop letter, the stack-top being read is indicated with a − sign on the arrow with again arrow pointing towards the state reached after such a pop move. The initial state is indicated by an unlabeled arrow pointing into it. The final states are marked with concentric circles. We will use the same notation for describing a VPA in all the subsequent examples.

In Example 4.2.8, the VPA has no accept state. The acceptance is by an empty stack. But one can easily change the VPA slightly and make it accept by a final state to abide by Definition 4.2.1.

Example 4.2.9 $\mathtt{Dyck}_k = \{$*balanced strings over* k *pairs of parenthesis*$\}$. *All the opening brackets form* Σ_c *and all the closing brackets form* Σ_r. Σ_l *is empty. The accepting* VPA *matches the stack-top with the pop letter and proceeds if they are a pair of parenthesis of the same type. Given below is a* VPA *accepting* $\mathtt{Dyck}_1$ *with a final state.*

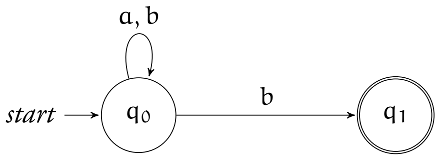


Thus, VPLs strictly contain REG.

In [5], it was shown that VPLs are determinisable; *i.e.,* they are contained in DCFLs. Also the language $\mathtt{EQ}(a, b)$ from Example 2.1.1 is known to be not acceptable by any VPA (*i.e.,* for any partition of the input alphabet). Thus, VPLs are known to be strictly contained in DCFLs.

It is known that NFA can be determinised [31]. But there are languages for which it is most natural and easy to come up with a nondeterministic finite state automata.

Example 4.2.10 *Consider the regular language from Example 2.1.1* ($\Sigma^* b$). *The* NFA *accepting this language can be given as follows:*

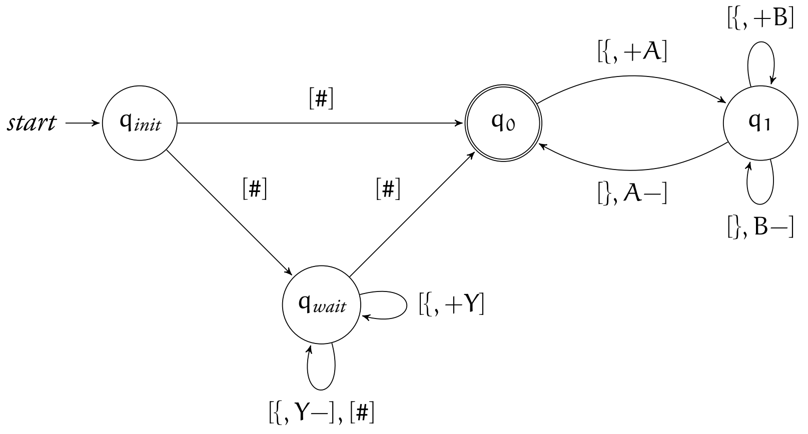


Given below is a VPL for which the most natural VPA accepting it is nondeterministic.

Example 4.2.11 *Let* $\mathsf{LastDyck}_k$ *consist of set of strings each consisting of marked substrings out of which the last substring is in* Dyck_k*; i.e.,*

$$\mathsf{LastDyck}_k = \{\#w_1\#w_2\#\ldots\#w_n \mid w_n \in \mathsf{Dyck}_k, n > 0\}.$$

Here, the input alphabet Σ *consists of* k *types of parentheses,* $\{[_1, [_2, \ldots, [_k,]_1,]_2, \ldots,]_k\}$ *and the marker* $\#$. Σ_c *consists of all the opening parentheses,* Σ_r *consists of all the closing parentheses, and* Σ_i *consist of the marker. Each* w_i *is a string over* $(\Sigma_c \cup \Sigma_r)$.



The VPA *accepting this language needs to guess at every* $\#$ *whether the string following it is the last substring or not. After this a check for Dyck is done by a deterministic* VPA *as described in Example 4.2.9.*

4.3 Known algorithms for MEM(VPL)

The membership problem for VPLs was first studied by Mehlhorn [40] where a space upper bound of $O((\log n/\log\log n)^2)$ was proved. This was improved by Braunmühl *et al.* in [15] where a logspace upper bound was proved for the problem. Later, Dymond [25] settled the complexity of the problem by proving an upper bound of NC^1. As MEM(REG) is already known to be hard for NC^1, it follows that MEM(VPL) is hard for NC^1 and hence NC^1-complete.

Thus, from the known result about MEM(VPL) we can modify Figure 4.1 and get Figure 4.2.

We review three known algorithms for MEM(VPL) in the following three sections. We first revisit the NC^1 algorithm for MEM(VPL) by Dymond [25]. This is the best

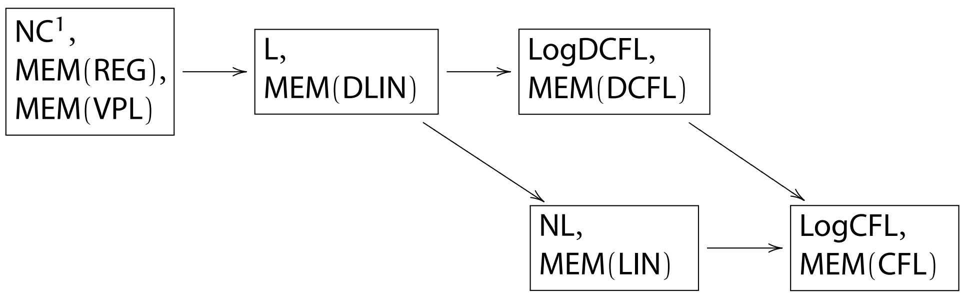


Figure 4.2: Connection between $\mathsf{MEM}(\mathcal{L})$ and complexity classes

known algorithm for MEM(VPL) while the other two presented here are useful for developing parsing algorithms for generalizations of VPLs.

4.3.1 NC^1 algorithm by Dymond [25]

In [25], 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 on a finite set and whose operations are functional compositions and certain unary operations depending on the inputs. This transformation is done in NC^1. The containment in NC^1 follows from the result, due to Buss [16], that the evaluation of formulae involving expressions as k-ary functions over a finite domain is in NC^1. We will discuss Buss's algorithm in more detail in Section 5.4.1. Here we describe Dymond's algorithm for MEM(VPL).

As TC^0 is contained in NC^1, applying Lemma 4.2.6, we can assume that the input is always a well-matched string and that the VPA never pops on an empty stack.

For a VPA, on an input w, let $h(i, w)$ denote the stack height after processing i letters. We use $h(w)$ to denote $h(|w|, w)$. We make following two observations about the height function:

Observation 4.3.1 *For any* VPA*, on an input* w*,* $h(i, w)$ *is the same across any run.*

Proof Sketch. As the stack movement depends on the partition of the input alphabet, height of the stack depends only on the input. The nondeterministic choices affect the state reached or the stack contents, but not the stack height. □

Observation 4.3.2 *For a* VPA *that never pops on an empty stack, on an input* w*,* $h(i, w)$ *can be computed in* TC^0*.*

Proof Sketch. To compute $h(i, w)$, it is sufficient to compute the difference between the number of push letters and pop letters in first i letters of w. It is known that this can be done in TC^0. □

Define a set of binary relations on Q (*i.e.,* on subset of $Q \times Q$), denoted $\Rightarrow^{i,j}$ for $1 \leqslant i \leqslant j \leqslant |w|$. In fact, they can be thought of as tables with rows and columns indexed by Q. The $[q, q']$th entry of the table corresponding to the interval $[i, j]$ is set to 1, $\Rightarrow^{i,j} [q, q'] = 1$, if and only if starting in state q with nothing on the stack, the VPA ends in state q' with again nothing on the stack on reading string $w_{i+1} \dots w_j$. And it is set to 0 otherwise.

A pair (i, j) of indices is called *height-matched* if the string $w_{i+1} \dots w_j$ is well-matched. The binary relations $\Rightarrow^{i,j}$ are defined only for height-matched (i, j). We wish to compute the value of $\Rightarrow^{0,n}$. (It is crucial that the input string w is well-matched.)

Note that i, j are integer indices of this relation, however, the domain for any relation indexed by i, j is finite (*i.e.,* $Q \times Q$). These relations are expected to capture all cases where the state, stack-top pairs (surface configurations) are reachable from one another without accessing the previous stack profiles. A unary operation, and a composition operation, are defined on these relations. Assuming that the tables for the smaller subintervals are available, these operations help in computing the tables for the larger subintervals. Suppose we have tables corresponding to consecutive intervals ($\Rightarrow^{i,i'}$ and $\Rightarrow^{i',j}$), say T_1 and T_2 respectively. Then the composition operator is defined in a standard way:

$$T_1 \circ T_2 = \Big\{ (q, q') \,\Big|\, \exists q'' \text{ such that } (q, q'') \in T_1, (q'', q') \in T_2 \Big\}$$

This gives a table for $T_{i,j}$ given the tables $T_{i,i'}$ and $T_{i',j}$. If we have a table T, we will define two unary operators. The first unary operator (which we will denote by $\mathsf{Ext}_{\{a,b\}}$) is defined when $a \in \Sigma_c$ and $b \in \Sigma_r$ and is given as follows:

$$\mathsf{Ext}_{\{a,b\}}(T) = \left\{ (q, q') \,\middle|\, \begin{array}{c} \exists q_1, q_2 \in Q, \exists A \in \Gamma : T[q_1, q_2] = 1, \\ (q_1, A) \in \delta(q, a), q' \in \delta(q_2, b, A) \end{array} \right\}$$

The second unary operator is defined for $c \in \Sigma_l$ and given as:

$$\mathsf{Ext}_{\{c\}}(T) = \Big\{(q, q') \,\Big|\, \exists q_1 \text{ such that } T[q, q_1] = 1 \text{ and } q' \in \delta(q_1, c)\Big\}$$

These compute tables for $T_{i-1,j+1}$ and $T_{i,j+1}$ given table $T_{i,j}$, respectively. $\mathsf{Ext}_{\{a,b\}}$ is applied at height-matched indices (i, j) if for any index between i and j the height of the stack is strictly more than the height at i or j and w_i and w_{j+1} are a and b respectively. $\mathsf{Ext}_{\{c\}}$ is applied if w_{j+1} is c and c is in Σ_i.

Let $\mathrm{Id}_{|Q|}$ denote the identity table, *i.e.,* $\mathrm{Id}_{|Q|}[q, q'] = 1$ if and only if $q = q'$. Then $\mathsf{Ext}_{\{a,b\}}(\mathrm{Id}_{|Q|})$ and $\mathsf{Ext}_{\{c\}}(\mathrm{Id}_{|Q|})$ give the table generated by the moves made by the VPA on reading the string ab and c, respectively.

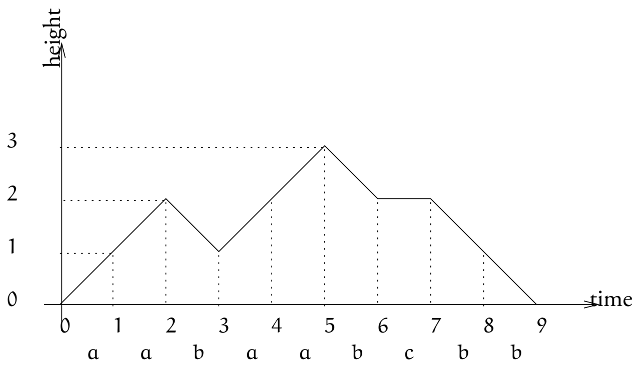


Figure 4.3: Height vs time profile of a VPA with $\Sigma_c = \{a\}, \Sigma_r = \{b\},$ and $\Sigma_l = \{c\}$ on word $w = aabaabcbb$

Example 4.3.3 *In Figure 4.3, we have shown a run of a* VPA *with* $\Sigma_c = \{a\}, \Sigma_r = \{b\}, \Sigma_l = \{c\}$ *on word* $w = aabaabcbb$. *The formula over* $\mathsf{Ext}_{\{a,b\}}, \mathsf{Ext}_{\{c\}}$ *and* $\circ$ *that we obtain for this is given as follows:*

$$\mathsf{Ext}_{\{a,b\}}(\mathsf{Ext}_{\{a,b\}}(\mathrm{Id}_{|Q|}) \circ \mathsf{Ext}_{\{a,b\}}(\mathsf{Ext}_{\{a,b\}}(\mathrm{Id}_{|Q|}) \circ \mathsf{Ext}_c(\mathrm{Id}_{|Q|})))$$

Given a string 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 the stack heights for various configurations, which is easy for VPL.

As pointed out in [25], the above transformation works not only for VPA but for a potentially larger class of PDA.

Observation 4.3.4 ([25]) *The* NC^1 *membership algorithm for* VPL*s also works for any pushdown automaton* M *satisfying the following three conditions.*

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

Clearly, VPA satisfy these conditions. By definition, they have no ϵ-moves. Though they may not end with an empty stack, this can be achieved by appropriate padding which is computable in TC^0 (Lemma 4.2.6). Though VPA may be nondeterministic, all runs have the same height profile, (Observation 4.3.1). Application of Lemma 4.2.6 also makes sure that the function $h(i, w)$ can be computed in TC^0, (Observation 4.3.2).

4.3.2 LogDCFL algorithm by Braunmühl *et al.* [15]

In [15], it is shown that deciding membership in a fixed VPL is in L by first describing a $O(\log^2 n)$ space, $O(n \log n)$ time procedure and then modifying it to lower the space bound to $O(\log n)$ (at the cost of time going up to $O(n^2 \log n)$). We note that the first algorithm in fact has a LogDCFL (DAuxPDA-Time(poly)) implementation.

These are weaker bounds as compared to the bound given by Dymond. (See Figure 2.3 to recall the relationship between L and NC^1.) However, we review these bounds mainly because this technique generalizes to give bounds on the complexity of membership problem of larger class of languages. We adapt the LogDCFL algorithm in Chapter 5 and the logspace algorithm later in this chapter.

We now give an overview of the first algorithm from [15], stating it explicitly as a LogDCFL procedure. We use the characterization of LogDCFL as languages accepted by polynomial-time DAuxPDA.

The algorithm of [15] assumed that VPA accept only well-matched strings. By Lemma 4.2.6, we know that this is not a restriction.

Lemma 4.3.5 (Algorithm 1 of [15]) *Let* $M = (Q, \Delta, Q_{in}, \Gamma, \delta, Q_F)$ *be a* VPA *accepting well-matched strings. Given an input string* x*, checking if* $x \in L(M)$ *can be done in* LogDCFL.

Proof: 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 $(|Q||\Gamma| \times |Q||\Gamma|)$ matrix over $0, 1$, where each row and column is indexed by a state-stacktop pair (surface configuration). The entry indexed by $[(q, X), (q', X')]$ is 1 if and only if $X = X'$ and M goes from surface configuration (q, X) to (q', X') while processing the string x_{ij}. We will call such a matrix the table T_{ij} corresponding to the string x_{ij}. M has an accepting run on the string x if and only if the $[(q_0, \perp), (q, \perp)]$-th entry is 1 for some $q \in Q_F$ in the table T_{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 $i \leqslant j$ 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 $l = \lceil (i + 2j)/3 \rceil$ (this is the value of l used in [15] to obtain balanced cuts), $x_{r'} = \epsilon$, 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 [15], it is shown that such fragments can always be defined and can be found deterministically and uniquely. We will describe this strategy, as well its complexity, shortly. It is also shown, and it is easy to see, that the tables computed by 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, 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 the 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 $O(n)$. Also note that the depth-first traversal of the recursion tree generated by the above procedure can be performed in LogDCFL. This is because the deterministic AuxPDA will stack one fragment (say (r_1, Λ_1)) and process the other fragment (r_2, Λ_2) on the logspace work-tape. Once it finishes processing both these fragments, it will then have Λ_3 on its work-tape and hence can start processing (r, Λ_3). This amounts to a depth-first traversal of the recursion tree. As the size of the tree is of $O(n)$, the DAuxPDA will run in time $p(n)$ for some polynomial p, provided the selection of the subintervals r_1, r_2 from (r, Λ) can be done on a logspace work-tape.

Now we describe the deterministic logspace procedure to compute intervals r_1 and r_2. Let $r = [i, j]$ and $r' = [i', j']$ be such that $i \leqslant i' \leqslant j' \leqslant j$ 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 M just after reading x_t (denoted as $h(t)$) is the lowest in that part. Now find two more points b, a such that $h(b) = h(t) = h(a)$, and the interval $[b, a]$ is the maximal valid subinterval containing t and within $[i, j]$. Let $r_1 = [b, t]$, $r_2 = [t, a]$. See for example Figure 4.4. Observe that r_1 and r_2 are contiguous with r_1 preceding r_2. Also, r' is fully contained inside either r_1 or r_2. (Why? Consider the case when $t \leqslant i' \leqslant j'$. t was the lowest in the part preceding r', thus $h(t) \leqslant h(i')$. If $a < j'$, then by maximimality of $[a, b]$, $h(a + 1) < h(a) = h(t) \leqslant h(i')$, and $a + 1 \in [t + 1, j']$. By choice of t, $a + 1 > i'$. Thus $a + 1 \in r'$, and $h(a + 1) < h(i') = h(j')$, contradicting the fact that r' is a valid interval. Hence it must be the case that $t \leqslant i' \leqslant j' \leqslant a$, and so r_2 contains r'. The case when $i' \leqslant j' \leqslant t$ is similar.)

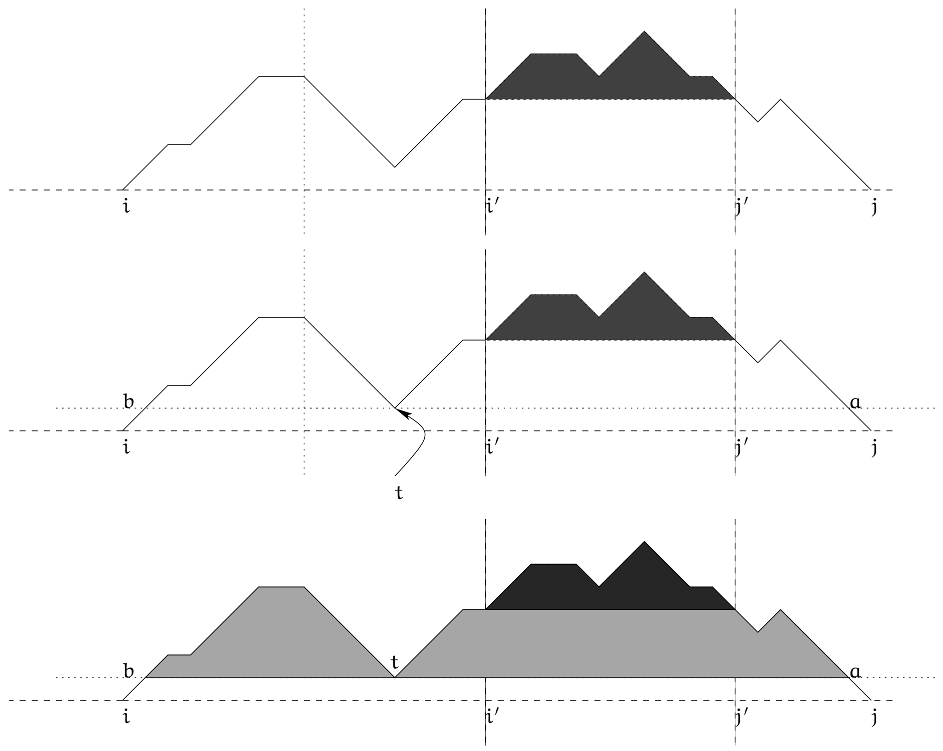


Figure 4.4: Finding b, t, a given i, j and i', j'.

It is easy to see that $r - r_3$ is of size at most three-fourth the size of $r - r'$, and so is the interval that contains r' (either r_1 or r_2). The same may not be true of the third part; the subinterval which does not contain r', say r_b, can be as large as $r - r'$. But it is easy to observe that at next step, this part will get tri-partitioned into intervals with sizes at most two third the size of r_b each.

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. This can be done in TC^0 for the input string x over pushdown alphabet Δ, and hence in L. $\square$

4.3.3 Logspace algorithm by Braunmühl *et al.* [15]

Now we describe the second algorithm of [15] due to which we get a logspace upper bound for membership testing of VPLs.

Lemma 4.3.6 (Algorithm 2 of [15]) *Let M be a VPA accepting well-matched strings over an alphabet Δ. Given an input string x, checking if $x \in L(M)$ can be done in L.*

Proof:

We describe the modifications to be made in the proof of Lemma 4.3.5 to get the logspace bound.

Note that all the fragments need not be carried along explicitly. Just remembering the path in the recursion tree, and the tables for all nodes on the path, suffices. Say the recursion tree is labelled as follows: The three children of a node are called l, r, o to mean 'left', 'right' and 'other', for the recursive calls $\mathcal{T}(r_1, \Lambda_1)$, $\mathcal{T}(r_2, \Lambda_2)$, and $\mathcal{T}(r, \Lambda_3)$ respectively. Label a node by a string $w \in \{l, r, o\}^*$ to denote the position of the node in the tree. (e.g label the leftmost leaf by l^d where d is depth of that leaf, label the root of the tree by ϵ.) It is easy to see (and this was used in [15] to prove correctness of their algorithm) that if one knows the label for a node, then reconstructing the intervals r, r' at this node from this label is possible in L (in fact a more general statement will be proved in Claim 4.4.3). They also observed that computing the next label from the current node label can be done in L (in fact, it is easy to note that this can be done in TC^0.) Thus at any stage our algorithm needs to remember the label of the current node being processed, and appropriate tables. We already saw that the table size is $O(1)$. Our procedure needs to know at most one table (the table in the Λ part of the fragment) per node along the current path.

As the depth of the recursion is bounded by $O(\log n)$, the depth of any node is also bounded $O(\log n)$. Thus, the label size, and the number of tables that need to be stored, are both at most $O(\log n)$ for any node using which the tree can be traversed (something more general holds, see Claim 4.4.4). Hence the procedure does not need to remember more than $O(\log n)$ bits at any stage of the recursion. □

4.4 Extending the logspace algorithm

We observe that the above logspace algorithm can be implemented as L^h where h is the height function for the VPA. The base logspace machine needs to figure out the break points a, b, t for which it needs the height of the stack at various time-steps. Once that is provided by an oracle, the rest of the recursion takes place in logspace.

Fix a PDA M on input alphabet Σ. We assume that the PDA satisfies the following conditions:

condition 1: In PDA M makes no ϵ moves and the height of the stack on all the runs is the same for a given input string.

condition 2: Let $h : \Sigma^* \to \mathbb{Z}^+$ be a function computing the height of the stack reached after having read a string from Σ^*, *height function*. Let h be computable in complexity class C.

For such PDA we can generalize Lemma 4.2.6 and Lemma 4.3.6 as follows:

Lemma 4.4.1 *For every* M *that satisfies conditions* 1 *and* 2*, there is another* PDA M' *on* Σ' *and a many-one reduction* g *in* C *such that* M' *satisfies conditions* 1 *and* 2 *and for every* $x \in \Sigma^*$*,*

- $\#\mathsf{acc}_M(x) = \#\mathsf{acc}_{M'}(g(x))$*, and*
- $g(x)$ *is well-matched.*

Lemma 4.4.2 *For every* M *on input alphabet* Σ *that satisfies conditions* 1 *and* 2 *and accepts well-matched input strings from* Σ^**, given an input* $x \in \Sigma^*$*, the tables corresponding to well-matched substrings of* x *can be computed in* L^h*. In particular, checking if* $x \in L(M)$ *can be done in* L^h*.*

We will also give details of the following two claims which we used in the proof of Lemma 4.3.6 implicitly. We describe them in their full generality, as they will be used later.

Claim 4.4.3 *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^h.

Proof: The input to the logspace oracle machine querying the height function of the transducer, h, is a label $w = w_1 w_2 \dots w_k$ where k is $O(\log n)$ length string over the alphabet $\{l, r, o\}$. 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 $\mathtt{trivial}([i, j])$ for $0 \leqslant i \leqslant j \leqslant n$ be defined as the trivial interval corresponding to $[i, j]$, that is $[\lceil 2j + i/3 \rceil, \lceil 2j + i/3 \rceil]$. The work-tape of the machine is initialized with $(r = [0, n], r' = \mathtt{trivial}(r))$.

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 $[i, j]$ with a gap $[i', j']$. Then the invariant maintained after having read $w_1 w_2 \dots w_m$ is $r = [i, j]$ and $r' = [i', j']$. Supposing the work-tape has a correct pair r, r' after having read $m - 1$-bits of the label, we now describe how to compute the next r, r' upon reading w_m. Supposing for the current pair (r, r'), the intervals r_1, r_2, r_3 can be computed in L^h, the bit w_m is read and the pair (r, r') is modified as follows: For each such pair if the bit is ‘l‘ (‘r‘) then r is set to r_1 (r_2, respectively). If $r' \subseteq r_1$ ($r' \subseteq r_2$) then r' is left unchanged else it is modified to $\mathtt{trivial}(r_1)$ ($\mathtt{trivial}(r_2)$, respectively). If the bit is ‘o‘ then r is left unchanged and r' is set to $r - r_3$. 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' in L^h. In the proof of Lemma 4.3.5, we saw that to compute these intervals we need to compute three points a, b, and t. Observe that these points depend only on the height of the stack. Hence they can be computed using a logspace machine that queries the height function.

□

Claim 4.4.4 *The tree can be traversed in* L^h*.*

Proof: From Claim 4.4.3, we know that once the label is available the interval itself can be computed L^h. We now see how the rest of computations 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 appropriate letter l, r or o. Also if the step results in computing a table, that table is stored along with the just added suffix. At any stage, the tables to be remembered are 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 are needed to be computed which can be done in L^h by the previous claim. Thus an overall traversal can be done in L^h.

□

We make the following two observations about the proofs of above claims:

Observation 4.4.5 (Regarding Claim 4.4.3) *Given the label (the path from the root) of any node in the recursion tree, the computation of the intervals corresponding to the fragment associated with the node is independent of the table entries and table sizes.*

Observation 4.4.6 (Regarding Claim 4.4.4) *As long as the table entries are of size* $O(1)$*, the recursion tree can be traversed in* L^h*.*

In the next section, we will consider PDA satisfying conditions 1 and 2 and study the complexity of membership problem for such PDA in a unified manner using the above lemmas and claims.

4.5 Stack-Synchronized PDA: a generalization of VPA

The height of the stack on any run of a VPA is easy to compute, since the input alphabet is partitioned into push/pop/internal letters. We generalize this notion following the framework developed by Caucal [19] to define synchronized PDA. We capture the notion of *e*asy to compute height functions using a transducer. Consider a complete deterministic transducer T that reads letters and outputs integers. It defines a map from strings to numbers as follows: trace the path of T on input $w = a_1 \dots a_n$. Let the numbers output along the way be $k_1, \dots, k_n$. Then $T(w) = \sum_{i=1}^{n} k_i$, and define $T'(w) = |T(w)|$.

For IDLs, a single state transducer that outputs a $+1$ on push letters, -1 on pop letters, and 0 on internal letters will correctly compute its stack height. For wm(VPL), the same transducer will work. Such a transducer, may not compute the height for a VPA due to pop moves on empty stack. But due to Corollary 4.2.7, TC^0(VPL) = TC^0(wm(VPL)). Hence for TC^0 closure of VPLs, similar transducer can be defined.

However, note that such single-state transducers can also compute stack-heights for languages that are provably not VPLs. For the language $EQ(a,b)$ from Example 2.1.1, we give a PDA whose stack-height is computed by a single state transducer.

Example 4.5.1 *Recall,* $EQ(a,b) = \{w \mid |w|_a = |w|_b\}$. *This language is not accepted by any* VPA *for any partition of the input alphabet. Consider transducer* $T = (Q, q, \Sigma, \delta_T)$ *where* $Q = \{q\}$, $\Sigma = \{a, b\}$, $\delta_T = \{(q,(a,+1)q),(q,(b,-1)q)\}$, *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.* A' *correctly computes the stack-heights of* P.

$q_0 \xrightarrow{a} (q_a, X)$	$q_a \xrightarrow{a} (q_a, A)$	$(q_a, A) \xrightarrow{b} q_a$	$(q_a, X) \xrightarrow{b} q_0$
$q_0 \xrightarrow{b} (q_b, X)$	$q_b \xrightarrow{b} (q_b, B)$	$(q_b, B) \xrightarrow{a} q_b$	$(q_b, X) \xrightarrow{a} q_0$

Note that P *is in fact deterministic.*

We can generalize this by allowing more states in T, or larger output alphabet for T, or both. If for each w, $T(w)$ correctly describes the stack-height of a PDA M on input w, then we say that M is *stack-synchronized* by T. This is a special case of Caucal's synchronized PDA [19] , and we define it more formally below.

4.5.1 Stack Synchronized PDA: formal definition

Definition 4.5.2 *We consider a class of finite transducers mapping words to integers. A transducer* T *over* Σ *and* $\mathbb{Z}$ *is a finite automaton* $(Q, q_0, F, (\Sigma \times \mathbb{Z}), \delta)$ *whose transitions are labelled with pairs* (a, k), *where* a *is a letter and* k *an integer. The first component of any such label is considered as an input, and the second component as an output. A run* $q_0(a_1, k_1)q_1 \ldots q_{n-1}(a_n, k_n)q_n$ *is associated to 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 (not getting stuck on any input) and input-deterministic transducers (*i.e.,* transducers whose underlying automaton is deterministic), in which all states are final. Consequently, for any such transducer T the relation g_T is actually a function, and is defined over the whole set Σ^*.

If the transitions of T are labelled over $\Sigma \times \{-1, 0, 1\}$, then T is said to be *incremental.* 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 $|g_T(u)| = |g_T(v)|$. Note that here we are using the graph based definition of a PDA from Section 2.1.

Definition 4.5.3 *A pushdown automaton* P *is stack-synchronized by a transducer* T *if the transition graph* G_P *of* P *is compatible with* T*, and further, the absolute value of the function* g_T*,* $||g_T||$*, computes the stack-height in the sense: if* $u \in L(G_P, \{q_0\}, \{pW\})$ *then* $||g_T(u)|| = |W|$.

Thus, a stack-synchronized pushdown automaton, SSPDA *is specified by a pair* (P, T) *where* P *is a* PDA*,* T *a transducer, and* P *is stack-synchronized by* T*. The* SSPDA *is said to be weak or one-way-strong if* P *is weak or one-way-strong.*

Example 4.5.4 $\mathsf{REV} = \{wcw^R \mid w \in \{a, b\}^*\}$ *is not a* VPL*. Consider the standard* DPDA *accepting this. It is stack-synchronized by a two-state transducer, one state (push state) having* $+1$ *on* a *and* b *and other (pop state) with* -1 *on* a *and* b*. The transducer moves from the push state to the pop state on seeing the letter* c*. It is known that for any* PDA *accepting REV, two states in the transducer are essential for stack-synchronization.*

Example 4.5.5 $\mathsf{EQ}_{k,l}(a, b) = \{w \mid k|w|_a = l|w|_b\}$ *is accepted by a strong* SSPDA*. Modify the transducer from Example 4.5.1 so that* $g_T(q, a) = +k$ *and* $g_T(q, b) = -l$*. We can define a* PDA*, similar to that in Example 4.5.1, accepting this language and synchronized by* T*. However, no weak* SSPDA *can accept this language unless* $k = l = 1$.

As observed in Example 4.5.1, even SSPDA synchronized by a single-state transducer properly generalize VPA. The generalization from single to multiple-state transducers is also proper, as is the generalization where the transducer is a generalization of *incremental.* (See Example 4.5.4 and Example 4.5.5 given above.)

The class we define as SSPDA is a restricted (and simpler) subclass of the synchronized pushdown automata considered by Caucal in [19]. Even though Caucal's results require, for a PDA to be synchronized by a transducer T, that the transition graph of P' satisfy some additional geometric properties with respect to T, these properties are always satisfied when only considering stack-synchronization.

Consequently, SSPDA enjoy all the good properties of the more general class of synchronized PDA. In particular for any fixed transducer T, the family of languages accepted by all PDA stack-synchronized by T contains all regular languages, is a sub-family of deterministic realtime context-free languages and forms an effective Boolean algebra. These results about synchronized PDA are established in [19].

4.5.2 Membership problem for SSPDA

We now discuss the complexity of the membership problem for SSPDA. We first prove a lemma that generalises Observation 4.3.1 and Observation 4.3.2 which we proved for VPLs. (Recall that VPA are stack-synchronised by a single state transducer):

Proposition 4.5.6 *For any fixed* SSPDA (P, T)*, and an input word* w*, the stack-height of* P *after processing* i *letters of* w*,* $h(w, i)$*, is the same across all nondeterministic runs. The function* $h(w, i)$ *is computable in* NC^1.

Proof: Even though P may be nondeterministic, the stack-synchronization ensures that all runs have the same stack-height profile. Let $q_0(a_1, k_1)q_1 \dots q_{n-1}(a_n, k_n)q_n$ be the run of transducer T on input $w = (a_1 \dots a_n)$. In NC^1, we can construct the run and hence the sequence $k_1, k_2, \dots, k_n$. Now a TC^0 circuit can compute, for each i, the sum $s_i = \sum_{j=1}^{i} k_j$. Since the stack-height is s_i if $s_i \geqslant 0$ and $-s_i$ otherwise, overall, the function $h(i, w)$ is computable in NC^1. □

This proposition along with Observation 4.3.4 allows us to extend Dymond's algorithm for SSPDA.

Lemma 4.5.7 *For any fixed weak* SSPDA M*, the membership problem is in* NC^1.

Proof: Proposition 4.5.6 implies that conditions 1 and 2 from Section Section 4.4 hold for SSPDA. Thus, we can apply Lemma 4.4.1 and obtain another well-matched SSPDA M'. For M', Observation 4.3.4 applies. Hence, we have NC^1 bound. □

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

An alternate approach for proving Lemma 4.5.7 is by reducing the membership testing for SSPDA to that for VPA. This turns out to be useful in more general settings of 'arithmetization' to be considered in the next chapter.

Let us first consider the alternate approach for weak SSPDA. We reduce membership testing in (P, T) to that in some VPA that combines the operation of P and T.

A naive approach is to convert a weak SSPDA (P, T) to a VPA 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 Proposition 4.5.6, the correct tags for any word can be computed in NC^1. More formally, the alternate proof for Lemma 4.5.7 is given below:

Proof: Formally, given a SSPDA (P, T) where $P = (Q^P, q_0^P, F, \Gamma, \Sigma, \delta^P)$ and $T = (Q^T, \Sigma, q_0^T, \delta^T, f_T)$, we construct a VPA M as follows: $M = (Q, q_0^P, F, \Gamma, \Sigma', \delta)$, 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 $w_T = \langle a_1, t_1 \rangle \dots \langle a_n, t_n \rangle$, where $t_j \in \{c, r, i\}$, and $t_j = c, r, i$ depending on whether $h(w, j) = h(w, j-1) + 1$ or $h(w, j-1) - 1$ or $h(w, j-1)$. By Proposition 4.5.6, we can produce the string w_T in NC^1. □

One can show that M accepts the string w_T if and only if P accepts w. The above proof works as long as P is weak and T is incremental (*i.e.,* f_T maps strings over Σ^* to the set $\{-1, 0, +1\}$).

If P is not weak but is one-way-strong, then a slight modification of the above construction works. On a letter a_i, if P has to push/pop k letters, then in w_T we introduce $k-1$ dummy call/return letters immediately after a_i so that M can push/pop one symbol per input letter. Since M must know what letters to push/pop, we put this information into the state set of M.

Formally, we first describe the mapping from w to w_T. Given an input string $w = a_1 \dots a_n$, we encode each letter a_j by a string v_j as follows:

If $h(w,j) - h(w,j-1) = +k$ for $k \in \mathbb{N}$, then $v_j = \langle a_j, c\rangle c^{k-1}$.
If $h(w,j) - h(w,j-1) = -k$ for $k \in \mathbb{N}$, then $v_j = \langle a_j, r\rangle r^{k-1}$.
If $h(w,j) - h(w,j-1) = 0$, then $v_j = \langle a_j, i\rangle$.

Now $w_T = v_1 v_2 \ldots v_n$. We can produce the string w_T in NC^1.

The VPA $M = (Q, Q_0, F, \Gamma, \Sigma', \delta)$ is constructed as follows. Define m to be the $\max\{|U|, |V| \mid pU \xrightarrow{a} qV \in \delta^P\}$. Then $Q = (Q^P \times \Gamma^{\leqslant m})$, $Q_0 = (Q_0^P \times \{\epsilon\})$, $F = (F^P \times \Gamma^{\leqslant m})$, $\Sigma' = (\Sigma \times \{c, r, i\}) \cup \{c, r\}$, and the partition of Σ' is defined as: $\Sigma'_x = (\Sigma \times \{x\}) \cup \{x\}$ for $x \in \{c, r\}$, $\Sigma'_i = \Sigma \times \{i\}$.

The transition function δ is defined as follows:
For $p \xrightarrow{a} q \in \delta^P$, δ includes $(p, \epsilon) \xrightarrow{(a,i)} (q, \epsilon)$.
For $p \xrightarrow{a} qV\alpha \in \delta^P$, δ includes $(p, \epsilon) \xrightarrow{(a,c)} (q, \alpha)V$.
For $pU\beta \xrightarrow{a} q \in \delta^P$, δ includes $(p, \epsilon)U \xrightarrow{(a,r)} (q, \beta)$.
For each $(q, W\gamma) \in Q$, δ includes $(q, W\gamma) \xrightarrow{c} (q, \gamma)W$ and $(q, W\gamma)W \xrightarrow{r} (q, \gamma)$.
This finishes the description of the VPA. Thus, we have the following lemma:

Lemma 4.5.8 *The membership problem for one-way-strong* SSPDA *is in* NC^1.

The proof of the above result does not directly extend to strong SSPDA. We may have $h(w, i) - h(w, i-1) = +k$, but the PDA doesn't merely push k symbols; it pops l symbols and then pushes $k + l$ symbols. Figuring out if this happens requires tracing out the computation of not just T but also P, which we do not know how to do in NC^1. However, it turns out that every strong SSPDA has an equivalent one-way-strong SSPDA.

Lemma 4.5.9 *For any strong* PDA *stack-synchronized by some transducer* T*, there exists an equivalent one-way-strong* PDA *stack-synchronized by* T.

Proof: Let $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$ be a PDA. Let k be the smallest integer such that for every transition $pU \xrightarrow{a} qV$ in δ, $\min\{|U|, |V|\} \leqslant k$ (note that $k = 0$ if and only if P is one-way-strong). We define a new PDA $P' = (Q', q'_0, F', \Gamma', \Sigma, \delta')$ with $\Gamma' = \Gamma \cup \{X\}$, $Q' = Q \times \Gamma^k$, $q'_0 = (q_0, X^k)$, $F' = F \times \Gamma'^k$ and

$$\delta' = \{(p, S)W \xrightarrow{a} (q, T)W' \mid \exists pU \xrightarrow{a} qV \in \delta,\ SW = UU',\ TW' = VU' \text{ and } \min\{|SW|, |TW'|\} = k\}.$$

Intuitively, P' is identical to P except that it simulates the top-most k stack cells of P using an enlarged control state set. It starts with a “virtual” stack containing k dummy symbols X (which will always remain on the stack since X is not part of P’s original stack alphabet) and proceeds similarly to P while suitably updating the top-most k stack cells inside its control state after each simulated transition.

It is not difficult to show that the transition graphs G_P and $G_{P'}$ of P and P' are isomorphic, where any configuration pUV of P with $|U| = k$ corresponds to the configuration $(p, U)VX^k$ of P', and any configuration pU with $|U| < k$ to the configuration $(p, UX^{k-|U|})X^{|U|}$. Hence by definition of q_0' and F', P and P' accept the same language.

Also note that the stack policy of P' is identical to that of P: every transition rule $(p, S)W \xrightarrow{a} (q, T)W'$ of P' corresponding to a transition rule $pU \xrightarrow{a} qV$ of P induces a stack height difference $d = |W'| - |W|$. Since $|S| = |T| = k$ and $SW = UU'$, $TW' = VU'$, we have $d = |TW'| - |SW| = |VU'| - |UU'| = |V| - |U|$. Another way to see this is to note that corresponding configurations of P and P' in the previously mentioned isomorphism have precisely the same stack height. Hence P' is synchronized by T.

Finally, by definition of δ', in every transition rule $(p, S)W \xrightarrow{a} (q, T)W'$ of P' we have either $|W| = 0$ or $|W'| = 0$, meaning that P' is one-way-strong. □

Putting together Lemma 4.5.7, Lemma 4.5.8, and Lemma 4.5.9 we get the following corollary:

Corollary 4.5.10 *For any* SSPDA, *the membership problem is in* NC^1.

4.5.3 Advantages of the model

SSPDA are a restriction of the more general model defined in [19]. The model starts from the class of languages that we understand well, namely VPLs. And perfectly abstract out the properties of VPLs that give them a nice structure and thereby provide a handle to generalize VPLs.

To us, this gives a natural way to understand the effect of existence of height function on the complexity of membership problem for languages between VPLs and DCFLs. This also gives us a way to quantify the hardness of the height function and give unified bounds for all the generalizations.

In the next section we will see the most general class of languages which arise from Caucal’s idea of generalizing language classes from VPLs.

4.6 Realtime height-deterministic PDA

We start with recalling a few things here. Adding a pushdown stack to an NFA significantly increases the complexity of the membership problem (NC^1 to LogCFL, refer Section 4.1). However, if stack operations are restricted to an input driven discipline, as in VPA, then membership is no harder than for NFA. What is being exploited (by VPA or by the SSPDA model we defined) is that, despite nondeterminism, all paths on a given input word have the same stack-profile, and this profile is computable in NC^1 for SSPDA and in TC^0 for VPA. One can view the partitioning of the input alphabet as providing an advice regarding the height of the stack to an algorithm that decides membership. This naturally leads to the question: what can be deduced from the existence of such a height function, independently of how this function is computed?

To precisely capture this idea,we use the term *height-determinism*, coined by [42]. 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 nondeterministic choices performed by the automaton. Consequently, in any (realtime) 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 should exist a *height function* h from Σ^* to integers, 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 h is computable in NC^1 directly satisfies the conditions in Observation 4.3.4, 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 problem.

4.6.1 Formal Definitions and properties of rhPDA

Definition 4.6.1 (rhPDA, [42]) *A realtime pushdown automaton*[1] $P = (Q, q_0, F, \Gamma, \Sigma, \delta)$ *is called height-deterministic if it is complete (does not get stuck* [2] *on any run), and* $\forall w \in \Sigma^*$, $q_0 \xrightarrow{w} q\alpha$ *and* $q_0 \xrightarrow{w} q\beta$ *imply* $|\alpha| = |\beta|$.

[1]In [42], 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.

[2]a PDA is stuck if for the current stack-top and state, no move is defined for the current input letter

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 $q \in Q$, $W \in \Gamma^*$.

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 function is left unspecified.

Proposition 4.6.2 ([42, 19]) *Any* rhPDA *can be determinised. Consequently, for a fixed* h*, the class of languages accepted by* rhPDA *and whose height function is* h *forms a boolean algebra (and properly includes regular languages). Moreover, language equivalence between two* rhPDA *with the same height function is decidable.*

All these results are effective as soon as h is computable. Since any deterministic realtime PDA can be completed and is height-deterministic, another consequence of the fact that rhPDA can be determinised is that the whole class rhPDA accepts precisely the class of realtime DCFL.

Something slightly stronger than determinisation is shown in [42] and will turn out to be useful for us.

Proposition 4.6.3 ([42]) *For every* rhPDA *A, there is an equivalent realtime, complete* DPDA B *(accepting the same language) such that if* $q_0 \xrightarrow{w} qW$ *labels a path in* G_A *and* $p_0 \xrightarrow{w} pY$ *labels a path in* G_B*, then* $|W| = |Y|$.

4.6.2 Instances of height-deterministic PDA

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

Definition 4.6.4 *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.

The definition of a transducer is same as in Definition 4.5.2.

In this setting, like in the case of SSPDA, 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^*.

One may consider several kinds of transducers.

The class referred to as rhPDA(FST) is same as SSPDA. We from now on refer to SSPDA as rhPDA(FST). As an aside, we note that [42] considers the class rhPDA(FST) as equivalent to synchronized PDA. This is not guaranteed to be true and has to be proved, since [19] also permits synchronization by norms other than stack-height.

The other classes we consider are rhPDA(PDT) where PDT stands for a pushdown transducer, and also to some extent on the class rhPDA(rDPDA$_{\text{1-turn}}$), where the transducer is a 1-turn PDT.

We first note that it is in fact unnecessary to consider more complex transducers than deterministic and complete PDTs. Formally:

Proposition 4.6.5 *For any* rhPDA P *whose height 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, the we can proceed as follows: We 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 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 function h.

If P satisfies the weaker completeness requirement, appeal to Proposition 4.6.3 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 (each transition of T simply has to output the integer matching the stack movement performed by this transition). By definition of P' and T and since T is complete, P's height function is correctly computed by T.

□

4.6.3 Complexity of the membership problem

As we already mentioned, rhPDA have exactly the same power as realtime DPDA in terms of accepted languages. Thus the membership problem 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 [48] that the following language is a hardest DCFL and is complete for the class LogDCFL.

Definition 4.6.6 ([48]) *Let* u *be a string over an alphabet* $\{(_1, (_2,)_1,)_2, [,], \#\}$ *in the form*

$$x_0[w_1\#z_1][w_2\#z_2]\ldots[w_k\#z_k] \textit{ for some } k \in \mathbb{N} \textit{ where,}$$

$$\begin{array}{lll} & x_0 \in & \{(_1, (_2\}^*, \\ \forall i : 1 \leqslant i \leqslant k & w_i \in &)_1 \cdot \{(_1, (_2\}^*, \\ \textit{and}\, \forall i : 1 \leqslant i \leqslant k & z_i \in &)_2 \cdot \{(_1, (_2\}^*. \end{array}$$

A string u *of this form is said to be in the language* $\mathrm{DetCh}(\mathrm{Dyck}_2)$ *if and only if for each* $1 \leqslant i \leqslant k$, $\exists x_i \in \{w_i, z_i\}$ *such that* $x_0x_1\ldots x_k \in \mathrm{Dyck}_2$ *that is if and only if 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 $\mathrm{DetCh}(\mathrm{Dyck}_2)$ *is deterministic context-free and is complete for the class* LogDCFL.

A realtime DPDA 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 it has type 1 parenthesis on the stack-top then it decided to choose x_i to be w_i, pops the stack-top, and pushes all but the first letter of w_i on the stack. Otherwise, x_i is z_i and w_i is read bit-by-bit and ignored by the DPDA. On z_i the stack-top is popped and all but the first letter of z_i is 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 language is accepted by the DPDA, else rejected. Thus this language can be accepted by a realtime DPDA; and hence membership testing for rhPDA is hard for LogDCFL.

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

Proposition 4.6.7 *The membership question for the class* rhPDA *(and thus also for* rhPDA(PDT)*) is complete for* LogDCFL *under logspace many-one reductions.*

We already saw that for complete deterministic finite state transducer, the height function can be computed in NC^1. We observe easy bounds on the complexity of the height function computed by other transducers:

Lemma 4.6.8 *For a complete deterministic transducer* T *computing function* g_T,

1. *If* T *is a* $rDPDA_{1\text{-}turn}$*, then* g_T *is computable in* L.
2. *If* T *is a* PDT*, then* g_T *is computable in* LogDCFL.

Proof:

1. It is known that $DPDA_{1\text{-}turn}$ can be simulated in logspace ([30]). Thus if a function is computed by a $rDPDA_{1\text{-}turn}$ transducer, a logspace machine can keep track of its output, and hence g_T is in L.
2. Given input x and an index $1 \leqslant i \leqslant |x|$, a 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 DAuxPDA needs no more than linear time, and a logarithmic size counter suffices.

□

Using Lemma 4.4.1 and Lemma 4.6.8 we get the following corollary:

Corollary 4.6.9 *The membership problem for* $rhPDA(rDPDA_{1\text{-}turn})$ *is in* L.

The class $rhPDA(rDPDA_{1\text{-}turn})$ referred to here contains languages accepted by realtime $DPDA_{1\text{-}turn}$ as well as languages accepted by rhPDA(FST). It is contained in DCFLs.

4.7 Restricted multi-stack machines

4.7.1 Multi-stack visibly pushdown automata

In this section, we discuss membership problem for a model recently considered by La Torre *et al.* [34]: 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 generalization 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 [34], 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 section, we consider a restriction of the above problem, where the number of phases is a constant. We define another version of the membership problem, MEM(MVPL_k). For a fixed MVPA M and fixed positive integer k, the problem MEM(MVPL_k) is to decide whether a given $w \in \Sigma^*$ is in $L_k(M)$, where $L_k(M)$ denotes the language $\{w \in \Sigma^* \mid w \text{ is accepted by } M \text{ with } \leqslant k \text{ phases}\}$.

This restriction of MVPA, where the number of phases is bounded, is also useful for many applications and has been defined and considered in [34]. The class is known to generalize VPLs and is properly contained in context-sensitive languages. In this paper, we show that the problem MEM(MVPL_k) is in LogCFL.

In order to show this, we use the other model of multi-pushdown machines, PD_k, defined by Cherubini *et al.* [20] which we considered in Section 3.4. Recall that it is 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. Also recall, Theorem 3.4.4, that MEM(PD_k) is in LogCFL.

Here, we give a reduction from MEM(MVPA_k) to MEM(PD_k). The LogCFL upper bound for MEM(PD_k) thus gives a LogCFL upper bound for MEM(MVPL_k). Note that MVPA does not have an ordering restriction on the stack usage. The reduction therefore is not immediately obvious. The languages accepted by MVPA within two phases are a proper subclass of context sensitive languages, a proper generalization of VPLs, and are incomparable with CFLs.

This result implies the same upper bound as for MEM(CFL) for an incomparable class of languages. (CFLs and MVPL_k are incomparable.) However, we do not know if MEM(MVPA_k) for any fixed k is hard for LogCFL.

4.7.2 Formal definition of bounded phase MVPA

Visible two stack machines ([34]). 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

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

Clearly, for any fixed partition of Σ, PHASE_k is a regular set. The finite state automaton accepting PHASE_k for fixed k is depicted in Figure 4.5. In the figure, $X = \Sigma_c^0 \cup \Sigma_c^1 \cup \Sigma_r^0$ and $Y = \Sigma_c^0 \cup \Sigma_c^1 \cup \Sigma_r^1$. (Moves on letters from Σ_i are not shown. For any such letter, the automaton stays in its current state.)

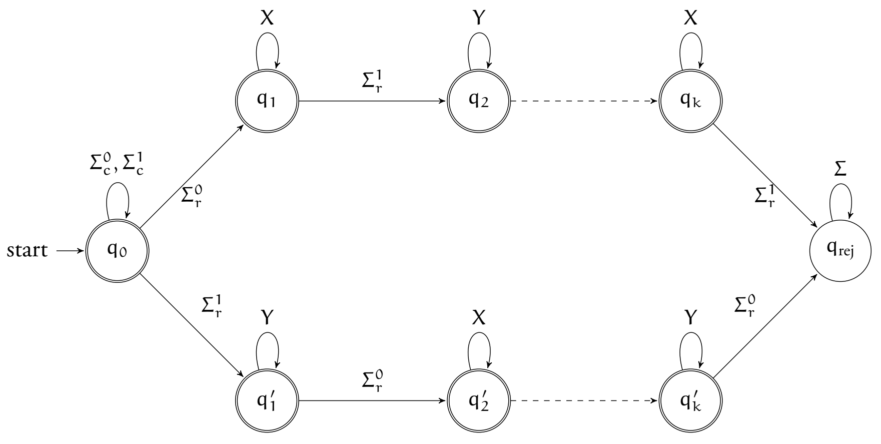


Figure 4.5: Finite state automaton accepting PHASE_k (for k odd)

A k-MVPA is an MVPA M such that language accepted by it is $L(M) \cap \mathsf{PHASE}_k$.

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 4.7.3, we assume that the given MVPA M satisfies $L(M) = L_k(M)$.

4.7.3 $\mathsf{MEM}(\mathsf{MVPL}_k)$ is in LogCFL

In this section, we consider the problem $\mathsf{MEM}(\mathsf{MVPL}_k)$ and establish the following theorem:

Theorem 4.7.1 *For every fixed* $k \geqslant 1$, $\mathsf{MEM}(\mathsf{MVPL}_k) \leqslant \mathsf{MEM}(\mathsf{PD}_k)$. *The reduction is a logspace many-one reduction.*

From Section 3.4.4 we know that $\mathsf{MEM}(\mathsf{PD}_k)$ is in LogCFL. Using this and Theorem 4.7.1 we get LogCFL upper bound for $\mathsf{MEM}(\mathsf{MVPL}_k)$. 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 [25], for which membership in a fixed regular language is complete [9], $\mathsf{MEM}(\mathsf{MVPL}_1)$ reduces to MEM(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 4.7.2, we assume that $L_k(M) = L(M)$. We convert M into a multi-pushdown machine N

having k stacks, called Main_i for $1 \leqslant i \leqslant k$, 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 is 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 4.7.2 *Fix a* MVPA M *and an integer* k. *There exist a* PD_k N *and a function* $f \in \mathsf{L}$, *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 sufficiently 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. (A similar idea was used in Lemma 4.2.6 for making VPAs height matched.) Formally, we show the following:

Lemma 4.7.3 *Fix a* MVPA M. *There exists another* MVPA M' *and a function* $g \in \mathsf{L}$ *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 on an empty stack.*

Proof Sketch. Let $g : \Sigma^* \to (\Sigma \cup \{X, Y\})^*$ where $X, Y \notin \Sigma$. For each $w \in \Sigma^*$, $g(w) = X^{|w|+1}Y^{|w|+1}w$. We convert the fixed MVPA M into another MVPA M':

$M' = (Q', \Sigma', \Gamma', \delta', q_0', F')$ where $\Sigma_c^{0'} = \Sigma_c^0 \cup \{X\}$, $\Sigma_c^{1'} = \Sigma_c^1 \cup \{Y\}$, and other parts Σ_r^j, Σ_i, of the alphabet are the same. The machine M' has two new letters in its stack alphabet, say A, B. A (B) is pushed on stack 0 (stack 1 respectively) while reading X (Y respectively). Also, if $(q, a, \perp, p) \in \delta$ for $a \in \Sigma_r^0$, then $(q, a, A, p) \in \delta'$. If $(q, a, \perp, p) \in \delta$ for $a \in \Sigma_r^1$, then $(q, a, B, p) \in \delta'$. On string w, machine M' behaves essentially the same as machine M; only the pop-on-empty-stack moves are replaced by pop-A or pop-B.

It is easy to check that for $w \in \Sigma^*$, $w \in L(M) \Leftrightarrow g(w) \in L(M')$ 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 4.7.3, 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 4.7.2, we need the values $ht^j(x)$ for each prefix x of w. These values are easy to compute:

Proposition 4.7.4 *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* L.

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.

We describe the reduction assuming that the first phase pops on stack-0. The other case can be handled similarly.

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 $k_i \geqslant 1$ for all i. Let $U, V, W, Z, \#$ be new letters not in Σ. During the mimic stage, the strings w_i are read. During the buffer stage, reading the marker #, the top of the second non-empty stack is marked with a special symbol. (Recall that there can be at most two non-empty stacks.) Reading the string of Us, the contents of the first non-empty stack are pushed above the marked second non-empty stack. These contents are in turn pushed into the next empty stack reading the string of Vs. The letter W is used for popping out the special symbol. The letter Z is used for a technical reason to maintain certain invariants. Then f is defined as below. (No padding is needed after w_k.) $f(w) = Zw_1\#U^{k_1+1}V^{k_1+1}\#Ww_2\#U^{k_2+1}V^{k_2+1}\#W\ldots w_i\#U^{k_i+1}V^{k_i+1}\#W\ldots w_k$. (The string $f(w)$ can be prefixed with 1 or 2 indicating that the first phase pops stack 1 or 2, respectively.)

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 4.7.5 *Machine M on input w has a nondeterministic 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 nondeterministic 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 4.7.2 and Theorem 4.7.1.

The problem $\mathsf{MEM}(\mathsf{MVPA}_k)$ is known to be hard for NC^1 because a special case of it $\mathsf{MEM}(\mathsf{VPA})$ is hard for NC^1. But no better hardness result is known. It will be interesting to have at least L or $\mathsf{LogDCFL}$ hardness for this.

Also, the only way to obtain upper bound is through a reduction to PD_k. Probably, lack of hardness should be thought of as a hint for existence of better upper bounds obtained possibly by bypassing this reduction.

4.8 Concluding Remarks

In this chapter we studied the membership problem for various subclasses of CFLs and generalisations of VPLs. The main goal of this was to characterize various complexity classes in the landscape between NC^1 and LogCFL.

The various language classes that we considered were chosen as likely candidates for refining the region of complexity classes in the following arm of containments: L $\subseteq$ LogDCFL $\subseteq$ LogCFL. There were some language classes which we discarded either because they did not add to the knowledge about this range of complexity classes because their membership problem trivially reduced to MEM(VPL) (*e.g.,* nested word languages [6], motley word languages [13]) or their membership problem did not seem to fit the LogCFL regime in any obvious way (*e.g.,* 2-visibly pushdown automata [18]).

This region is of interest for obtaining hardness results for many natural problems. For example, consider the graph isomorphism problem: given two graphs, check whether there is an edge preserving bijection between their vertices. This problem is known to be hard for GapL. No other hardness result is known. Of course, it is hard for L, but it would be interesting to see a LogCFL or a LogDCFL hardness. Given that both are not known, it may make sense to look at classes between L and LogDCFL and prove hardness with respect to these. We feel that studying MEM(L) is one of the promising approaches of making a progress in this direction.

5 Counting problem for VPLs and their generalization

5.1 Arithmetic functions and complexity classes

The Boolean complexity classes we have been discussing give bounds on resources needed to compute Boolean functions, $\{f \mid f : \Sigma^* \to \{0, 1\}\}$. In this chapter we discuss arithmetic functions, $\{f \mid f : \Sigma^* \to \mathbb{N}\}$ and complexity classes dealing with them. Given a Boolean complexity class defined via acceptance of a resource bounded nondeterministic Turing machine, defining the corresponding arithmetic class can be done in a natural way as follows: Let M be a nondeterministic Turing machine. M computes a Boolean function f if and only if for all the inputs $x \in \Sigma^*$, $f(x) = 1$ if and only if M has at least one accepting path over x in M. M is said to compute an arithmetic function f if and only if for all inputs $x \in \Sigma^*$, $f(x) = k$ if and only if M has k accepting paths over x in M. We will call the process of coming up with an arithmetic complexity class corresponding to a Boolean complexity class (in this case, by changing the acceptance criteria) *arithmetization*.

In fact, for any nondeterministic machine (such as PDA, $\mathsf{PDA}_{1\text{-turn}}$, VPA, NFA, rhPDA, SSPDA, MVPA), such an accepting criterion can be defined which will yield a set of arithmetic functions counting the accepting paths in these machines. Formally, for any class $\mathcal{C}$ of automata, its arithmetic version $\#\mathcal{C}$ is defined as follows:

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

If this is our method for defining a set of arithmetic functions, it places an immediate hurdle in arithmetizing deterministic Turing machines or other deterministic computation models. In particular if we start with a deterministic complexity class, we don't get any arithmetic complexity class that corresponds to it in this way. But complexity theorists would like to define some set of arithmetic functions for each Boolean complexity class. In particular for L and for P-time, there are no known corresponding arithmetic complexity classes. (We will discuss the importance of arithmetic complexity classes for complexity theorists, shortly.)

Counting the number of proof trees is another known method of arithmetization. (Recall the definition of proof tree from Section 2.2.2). This can be used for arithmetizing Boolean circuit classes. Inspired by the circuit characterization of #LogCFL [52], this method was used to arithmetize NC^1 by Caussinus *et al.* in [29]. (Jiao in [32] had suggested this method for arithmetizing NC^1.) Let the arithmetization thus obtained be denoted as $\#NC^1$.

In the past, another approach to arithmetize Boolean complexity classes which have deterministic underlying models was used in [29]. One of the ways to arithmetize NC^1 in [29] was by counting the number of accepting paths in a NFA. Let us denote the class thus defined as #NFA. They also proved that $\#NFA \subseteq \#NC^1$.

We now discuss the importance of the arithmetization of Boolean complexity classes and that of arithmetic functions. For nondeterministic complexity classes such as NP, NL, LogCFL arithmetizations have been considered (by counting the number of accepting paths) in the past by [50, 7, 52] respectively. Arithmetizations are denoted as #P, #L, #LogCFL, respectively. These complexity classes are properly characterized by certain natural problems. Computing permanent of a matrix over $\mathbb{N}$ characterizes #P. Its determinant can be computed in #L(and characterizes a complexity class called GapL). And evaluation of poly-degree polynomials characterizes #LogCFL. Similarly, in [29] it is shown that #NFA is characterized by multiplication of constant-sized matrices, where the matrices are over $\mathbb{N}$. Thus counting accepting paths in resource bounded machines or NFA corresponds to computing natural and useful arithmetic functions.

Motivated by the work of [29] and by even more fundamental reason that arithmetic functions are really important and natural, we define arithmetic functions corresponding to counts of accepting paths for all the non-deterministic computations we have seen in the preceding chapters. We analyze their complexity. We use the notion of height determinism defined in Chapter 4 in order to obtain the complexity bounds.

In Section 5.2, we define the arithmetization of VPA, denoted as #VPA, and study its complexity. We prove that #VPA$\subseteq$ FLogDCFL. (A containment $\mathcal{F} \subseteq \mathcal{C}$ involving both a function class $\mathcal{F}$ and a language class $\mathcal{C}$ means: $\forall f \in \mathcal{F}, L^f \in \mathcal{C}$, where $L^f = \{\langle x, i, b\rangle \mid$ the ith bit of $f(x)$ is $b\}$.) We also prove some closure properties for the class #VPA.

In Section 5.3, we study the arithmetization of rhPDA(FST) and rhPDA. We prove that #rhPDA(FST) is complexity theoretically equivalent to #VPA.

Recall that, MEM(rhPDA) is complete for LogDCFL. The class #rhPDA thus gives a handle for arithmetizing a deterministic complexity class LogDCFL. To our knowledge this is the first arithmetization of LogDCFL known in the literature. For #rhPDA, we get FLogDCFL upper bound. Thus we have an arithmetization for the deterministic complexity class LogDCFL using the nondeterministic model of rhPDA. And the arithmetization gives a set of functions which are of the same complexity as their Boolean equivalents. A few observations regarding this result will be discussed in Section 5.3.2.

In Boolean setting, MEM(VPL) and MEM(REG) both characterized NC^1. But their arithmetizations seem to have a large gap. We end this chapter with a list of hurdles we have faced while trying to close this gap.

5.2 Counting Paths in a VPA

The following is the main result established in this section:

Theorem 5.2.1 *Arithmetization of* VPA *is contained in* FLogDCFL, *i.e.,* #VPA $\subseteq$ FLogDCFL.

We also define another class of languages called Mod_pVPA as follows:

For any function class $\#\mathcal{C}$, let $\mathsf{Mod}_p\mathcal{C}$ denote the class of languages L such that there is an $f \in \#\mathcal{C}$ satisfying

$$\forall x \in \{0,1\}^* : \quad x \in L \Longleftrightarrow f(x) \equiv 0 \mod p$$

As an aside we also prove the following corollary:

Corollary 5.2.2 *For each fixed* k, $\mathsf{Mod}_k\mathsf{VPA} \subseteq \mathsf{L}$.

The theorem is established by essentially using Lemma 4.3.5 and making appropriate changes to it. The idea is the same, namely of building tables corresponding to well-matched strings and then combining them together. The modifications needed in order to combine them correctly are described in detail in Section 5.2.1. The main difference is that the table entries are now over $\mathbb{N}$. Combining such tables may blow up the entries to exponential values. The technique of Chinese remaindering is used for bounding the size of each table entry. The algorithm can then be implemented by an oracle DAuxPDA making queries to the height function. The height function for VPA is in TC^0. Section 5.2.2 describes these steps in more details.

We analyze the class #VPA for its closure properties in Section 5.2.3.

5.2.1 Modifications to the LogDCFL algorithm for MEM(VPL)

To adapt this algorithm to the arithmetic setting, we now consider the tables that we defined in the proof of Lemma 4.3.5, and lift them over to $\mathbb{N}$. The entry $[(q, X), (q', X')]$ of the table corresponding to the interval (i, j) is filled with k if and only if $X = X'$ and the VPA starting in state q reaches q' along k different paths, processing the string $x_{(i,j)}$.

Consider the fragment (r, Λ), where $\Lambda = (r', T')$. The interval r' is a sub-interval of the interval r. Let two strings over the input alphabet, say u, v, be such that $x_r = ux_{r'}v$ (note that one of u, v, $x_{r'}$ can possibly be ϵ). The modifications to procedure $\mathcal{T}$ are given below.

If $|x_r| \leqslant 2$ (in this case Λ will be trivial), then the table T_r can be filled as follows: the $[(q, X), (q', X')]$-th entry will be set to k if $X = X'$ and the machine M can start from surface configuration (q, X), read x_r, and reach configuration (q', X') in *exactly* k ways. This can be filled by simply looking up the transition function δ'. Thus at the base case, we can fill the tables.

If $|x_r| > 2$ but $|uv| \leqslant 2$ (so Λ is not trivial), then assume that inductively, we have the table $T'_{r'}$ computed correctly. Then set the $[(q, X), (q', X)]$-th entry of the table

T_r as follows:

$$T_r^{[(q,X),(q',X)]} = \sum_{q_1,q_2 \in Q} \#[(q,X) \rightsquigarrow_u (q_1,X)] \cdot T_{r'}^{[(q_1,X),(q_2,X)]} \cdot \#[(q_2,X) \rightsquigarrow_v (q',X)] \tag{5.1}$$

$\#[(q,X) \rightsquigarrow_u (q',X')]_M$ is used to indicate the number of different ways in which the PDA M can go from (q,X) to (q',X') reading the string u. M is omitted if it is clear from the context.

The only other case is that u is a push letter and v is a pop letter; since $ux_{r'}v$ and $x_{r'}$ are well-matched. In this case,

$$T_r^{[(q,X),(q',X)]} = \sum_{\substack{q_1,q_2 \in Q,\\ Y \in \Gamma}} \#[(q,X) \rightsquigarrow_u (q_1,Y)] \,.\, T_{r'}^{[(q_1,Y),(q_2,Y)]} \,.\, \#[(q_2,Y) \rightsquigarrow_v (q',X)] \tag{5.2}$$

Both these cases can be combined into the following single equation:

$$\begin{aligned} T_r^{[(q,X),(q',X)]} &= \sum_{s_1,s_2 \in Q \times \Gamma} \#[(q,X) \rightsquigarrow_u s_1] \cdot T_{r'}^{[s_1,s_2,]} \cdot \#[s_2 \rightsquigarrow_v (q',X)] \\ T_r^{[(q,X),(q',X')]} &= 0 \quad \text{if } X \neq X' \end{aligned} \tag{5.3}$$

Since $T'_{r'}$ is available through recursive computation, and since the other terms in these equations can be found from δ, $\mathcal{T}$ can compute T_r.

For handling the case when $|uv| > 2$, we just redefine the $\times$-operator as matrix multiplications over $\mathbb{N}$. Let T_b denote the table corresponding to interval r_b, for $b = 1, 2$, where r_1 and r_2 are contiguous with r_1 preceding r_2. Then the table T_3 for $r_3 = r_1 \cup r_2$ is given by $T_3 = T_1 \times T_2$; that is,

$$T_3^{[(q,X),(q',X')]} = \sum_{p,Y \in Q \times \Gamma} T_1^{[(q,X),(p,Y)]} \,.\, T_2^{[(p,Y),(q',X')]} \tag{5.4}$$

(Inductively, this sets an entry to be 0 if $X \neq X'$.)

Under this semantics for tables and $\times$ operator, we can establish the following:

Claim 5.2.3 *For every interval* $r = [i,j]$ *arising in the recursion tree on input* $([0,n], ([2n/3, 2n/3]\ , \mathtt{id}))$, *the* $[(q,X),(q',X')]$*-th entry of the table* T_r *computed by* $\mathcal{T}$ *equals the number of distinct paths of* M *from* (q,X) *to* (q',X) *on string* x_{ij}.

Proof:(of claim) The procedure $\mathcal{T}$ processes intervals as fragments. The correctness proof proceeds by induction on the effective size of the interval; that is, for a recursive call on input $(r, (r', T'))$, we show by induction on the size of $r - r'$ that if T' is correct for r', then $\mathcal{T}$ returns the correct table for r.

The base case is when $r - r'$ is an interval of size 2 or less. If $|r'| = 0$, then $\mathcal{T}$ computes T_r directly from δ and so is correct. If $r' \neq 0$, then correctness follows from Equation 5.3.

For the inductive case, consider a fragment where $|r - r'| > 2$. $\mathcal{T}$ computes fragments (r_1, Λ_1) and (r_2, λ_2) and makes recursive calls. Assume that $\{b, c\} = \{1, 2\}$ and that r_c contains r'. As argued in [15], the effective interval in fragment (r_c, Λ_c) is strictly smaller than $|r - r'|$, and so by induction, $\mathcal{T}$ correctly computes T_{r_c}. r_b has a trivial pair attached and may not be smaller. Assume for now that it is smaller, so by induction, $\mathcal{T}$ correctly computes T_{r_b} as well. Now T_{r_3} is computed by Equation 5.4 which correctly combines paths over x_{r_1} and x_{r_2}. Finally, $\mathcal{T}$ is invoked with $(r, (r_3, T_3))$, and by induction, this call terminates with the correct value of T_r.

Suppose now that r_b is not smaller than $r - r'$. (This can happen, for instance, if r_c contains just r' and r_b is all the rest of r.) But then $\mathcal{T}$, while processing (r_b, Λ_b), makes calls with inputs (r_{bl}, Λ_{bl}) where $l \in \{1, 2\}$, and each call has a smaller effective interval length. So $T_{r_{b1}}$ and $T_{r_{b2}}$ are computed correctly by induction, and $T'' = T_{r_{b1} \cup r_{b2}}$ is obtained via Equation 5.4 which correctly combines paths. Then $\mathcal{T}$ is invoked with $(r_b, (r_{b1} \cup r_{b2}, T''))$. By induction, this call terminates with the correct value of T_{r_2}. □

5.2.2 #VPA is contained in FLogDCFL

The modified recursive procedure for computing the newly defined tables over $\mathbb{N}$ cannot directly be implemented in LogDCFL, because each entry of a table may need polynomially many bits. (The number of paths of a VPA on any string cannot exceed $2^{O(n)}$, so polynomially many bits suffice.)

The Chinese Remaindering Technique: We will use the following result, and its algorithmic version stated below.

Lemma 5.2.4 (Chinese Remainder Theorem CRT; folklore) *For each k, let P_k be the product of the first k primes, $p_1 < \ldots < p_k$. Then each integer in the interval $[0, P_k)$ is uniquely determined by its residues modulo these k primes.*

Lemma 5.2.5 (Algorithmic version of CRT; [23], see also [2]) *For each* k, *let* P_k *be the product of the first* k *primes,* $p_1 < \ldots < p_k$. *Given a* k*-tuple* $\langle a_1, \ldots a_k \rangle$ *where for each* i, $a_i \in [0, p_i)$, *the unique integer* $a \in [0, P_k)$ *such that* $a_i = a \bmod p_i$ *for each* i *can be computed in* L.

The technique of Chinese remaindering can be used in order to obtain LogDCFL upper bound as follows: if we were to perform all the operations modulo small (logarithmically many bits) primes, then, analogous to Lemma 4.3.5, the modified procedure can be implemented in FLogDCFL. The fragments that get pushed on to the stack and processed on the work tape will have $O(\log n)$-bit representations owing to not only the indices of the intervals but also the tables. (In the previous case, the tables were of size $O(1)$.) This can be handled by a DAuxPDA. If we do the above implementation for sufficiently many primes, then by Chinese Remaindering (Lemma 5.2.5) we will be able to recover the exact number in FL. Overall, we have that counting number of accepting paths in machine M over input x can be performed in $\mathsf{FL}^{\mathsf{FLogDCFL}} = \mathsf{FLogDCFL}$. In conjunction with Lemma 4.2.6, this shows that #VPA$\subseteq$ FLogDCFL, establishing Theorem 5.2.1.

We now prove Corollary 5.2.2.

Proof:[of Corollary 5.2.2] Note that it suffices to compute the table operations, as defined in Equations 5.1,5.2, and 5.4, modulo k. Hence, the tables will be of size $O(|Q|^2|\Gamma|^2 k) = O(1)$.

As the tables are of size $O(1)$, we can apply Observation 4.4.5 and Observation 4.4.6. Now similar to Lemma 4.4.2, the tables corresponding to well-matched intervals can be computed in L^h. Here, the height function is in TC^0 and we know that $\mathsf{L}(\mathsf{TC}^0)$ is in L. Thus, the corollary.

□

Note that if the recursion tree is to be addressed by indices explicitly and not by node labels, then Claim 4.4.3 and Observation 4.4.5 will not be applicable. In this case, we will have to generalize Lemma 4.3.5 that refers to the recursion tree explicitly by indices of the intervals. And this will give a $\mathsf{LogDCFL}^h$ bound as opposed to L^h bound.

In the following table we summarize the various results obtained in the arithmetic setting, using the algorithms from [15]:

Recursion tree node description	Table entries over $\mathbb{Z}$/ $\mathbb{Z}_p$	Bounds
explicit indices	p is a constant	LogDCFL
only path labels	p is a constant	L
explicit indices	p is poly-valued	LogDCFL
only path labels	p is poly-valued	LogDCFL

Table 5.1: Results obtained using algorithms from [15]

5.2.3 Closure properties of #VPA

Recall that VPA enjoy many nice Boolean closure properties [5]. In this section we examine the closure properties of #VPA.

We first discuss what we mean by closure for the class #VPA. We know that when a function f is in the class #VPA, there is a nondeterministic VPA M, that is, there is a pushdown machine with tri-partitioned input alphabet Σ, which on reading any input $x \in \Sigma^*$ generates as many accepting paths as $f(x)$. Therefore, a function in #VPA can be qualified with two things: the VPA itself and the partition of the input alphabet.

We consider two different types of notions for closure of #VPA.

- Type 1: Same input alphabet, same partition: Here, the closures we show are for a set of functions computed by VPA that have the same input alphabet and the same partition of the input alphabet. In fact, the Boolean closures shown by [5] are also for VPA with the same input alphabet and the same partition of the input alphabet.

- Type 2: Same input alphabet, different partition: Here, the closures are for a set of functions computed by VPA that have the same input alphabet but possibly different partitions. Note that the first notion of closure is more strict than this.

  However, to prove closures here, we need to cheat. We know that the class #VPA is hard for NC^1. Therefore, in some sense, even if we take NC^1 or AC^0 closure of the class #VPA, the resulting class of functions, *i.e.,* AC^0(#VPA), may not be much more powerful as compared to #VPA. Consider say two functions, f_1 and f_2 which are in #VPA, possibly by VPA having different partitions of the same input alphabet Σ. Now, say, we wish to prove that $f_1 + f_2$ is also in #VPA. We instead prove that $f_1 + f_2$ is in AC^0(#VPA).

For this we design a VPA M' which may have possibly different input alphabet Σ' and/or different partition of the input alphabet. And a function $g \in AC^0$ such that for every $x \in \Sigma^*$, $g(x) \in \Sigma'^*$ and $f_1(x) + f_2(x)$ equals the number of accepting paths in M' on $g(x)$.

Type 1 closure properties:

The closure properties for this type can be stated as follows:

Theorem 5.2.6 *For a fixed* $k \in \mathbb{N}$, *let* $M_1, M_2, \ldots, M_k$ *be* VPA *over an alphabet* Σ *having the same partition of the input alphabet. Let* $f_1, f_2, \ldots, f_k$ *be functions from* Σ^* *to* $\mathbb{Z}$. *If* $\forall x : x \in \Sigma^*$ *and* $\forall i : 1 \leqslant i \leqslant k$, $f_i(x) = \#acc_{M_i}(x)$, *then, there exist* VPA $M, M'N, N'$ *over* Σ *with the same partition of* Σ *as* M_1, *such that for each* $x \in \Sigma^*$,

$$\begin{array}{lll}
(a) & \#acc_M(x) & = & f_1(x) + f_2(x) + \ldots + f_k(x). \\
(b) & \#acc_{M'}(x) & = & f_1(x) \times f_2(x) \times \ldots \times f_k(x). \\
(c) & \#acc_N(x) & = & (f_1(x))^k \\
(d) & \#acc_{N'}(x) & = & \left(\begin{array}{c} f_1(x) \\ k \end{array} \right).
\end{array}$$

We prove the above theorem for $k = 2$. It is easy to generalize the methods for any fixed k.

Proof:[Theorem 5.2.6] The main ideas for proving the closures (a) and (b) above, are similar to those used for proving Boolean closures. To compute addition, one constructs a VPA by taking union of the two VPA M_1 and M_2. For multiplication, a product VPA is constructed. By correctly defining the acceptance criteria, one can generate the desired number of accepting paths.

In the constructions, having the same input alphabet and the same partition of the input alphabet plays a vital role.

Let $M_1 = (Q^1, q_0^1, F^1, \Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i, \Gamma^1, \delta^1)$ and $M_2 = (Q^2, q_0^2, F^2, \Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i, \Gamma^2, \delta^2)$. Note that here Q^1 and Q^2 have to be disjoint. The VPA M that we design for computing $f_1 + f_2$ on reading any letter in its initial state, nondeterministically simulates the moves of machines M_1 and M_2. And from there on, depending on the current state and stack-top pair, it simulates a move if it is defined in either δ_1 or δ_2

for the pair. (Note that simply taking the union of the two start states would suffice. We do this stick to the definition of a VPA which insists that there is a unique initial state.) Formally, $M = (Q, q_0, F, \Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_i, \Gamma, \delta)$, where $Q = Q^1 \cup Q^2 \cup \{q_0\}$, $F = F^1 \cup F^2$, $\Gamma = \Gamma^1 \cup \Gamma^2$, and δ is a union of δ^1, δ^2, and δ' where, δ' is defined as follows:

$\forall a \in \Sigma_c$: $q_0 \xrightarrow{a} q'X' \in \delta' \iff q_0^1 \xrightarrow{a} q'X' \in \delta^1$ or $q_0^2 \xrightarrow{a} q'X' \in \delta^2$

$\forall a \in \Sigma_i$: $q_0 \xrightarrow{a} q' \in \delta' \iff q_0^1 \xrightarrow{a} q' \in \delta^1$ or $q_0^2 \xrightarrow{a} q' \in \delta^2$

$\forall a \in \Sigma_r$: $q_0 \perp\xrightarrow{a} q' \perp \in \delta' \iff q_0^1 \perp\xrightarrow{a} q' \perp \in \delta^1$ or $q_0^2 \perp\xrightarrow{a} q' \perp \in \delta^2$

For such a machine we prove the following, more general, claim.

Claim 5.2.7 *On input* $x \in \Sigma^*$, M_1 *reaches a set of configuration* $S_1 \subseteq Q^1 \times (\Gamma^1)^*$ *and* M_2 *reaches a set of configurations* $S_2 \subseteq Q^2 \times (\Gamma^2)^*$ *if and only if* M *reaches* $S_1 \cup S_2$.

We prove this by induction on the length of the input. But suppose this claim holds, then it is easy to see that the machine M will correctly compute the addition of f_1 and f_2 and hence the first part of the theorem.

Proof: [Claim 5.2.7] For $|x| = 1$, there are three cases, namely $x \in \Sigma_c, x \in \Sigma_r$, and $x \in \Sigma_i$. We elaborate on the first case, the others can be worked out similarly. For $j \in \{1, 2\}$, $q_0^j \xrightarrow{x} S_j \subseteq (Q^j \times \Gamma^j)$. By construction, $q_0 \xrightarrow{x} \{S_1, S_2\}$. Thus the base case hold.

We assume that the claim holds for $|x| < k$. Now say $x = x'a$ and $|x| = k$. Here too three cases arise: $a \in \Sigma_c, a \in \Sigma_r$, and $a \in \Sigma_i$ and we consider the first case. The other cases are similar to this. Say on x', M_j reaches $S_j \subseteq Q^j \times (\Gamma^j)^h$ where $h = ht(x')$. But induction hypothesis, M reaches $(S_1 \cup S_2)$. On reading a, whatever subsets are reached by M_1 and M_2, are due to a move of the machines starting from one of the configurations among S_1 and S_2. By the definition of δ all those moves are possible in M. And hence M reaches all the configurations reachable by either M_1 or M_2. $\square$

Now we construct M' over the same alphabet Σ with the same partition as for M_1 and M_2 which is such that $\forall x$, $\#acc_{M'}(x) = f_1(x) \times f_2(x)$. It is essentially the product of the two VPA M_1 and M_2. The stack alphabet is also the product of the stack alphabets of M_1 and M_2. The moves here can be simulated mainly because the partition of the input alphabet is the same for both M_1 and M_2. We describe the construction for the sake of completeness.

$M' = (Q = Q^1 \times Q^2, q_0 = (q_0^1, q_0^2), F = F^1 \times F^2, \Sigma, \Gamma = \Gamma^1 \times \Gamma^2, \delta)$. Here, the transition function δ is defined as follows:

$$\left\{ (p^1,p^2)(\alpha^1,\alpha^2) \xrightarrow{a} (q^1,q^2)(\beta^1,\beta^2) \;\middle|\; \begin{array}{l} p^1\alpha^1 \xrightarrow{a} q^1\beta^1 \in \delta^1 \\ \text{and} \\ p^2\alpha^2 \xrightarrow{a} q^2\beta^2 \in \delta^2 \end{array} \right\}$$

where,

$$|\alpha^1| = |\alpha^2| = \begin{cases} 0 & \text{if } a \in \Sigma_i \text{ or } \Sigma_c \\ 1 & \text{if } a \in \Sigma_r \end{cases}$$

$$|\beta^1| = |\beta^2| = \begin{cases} 0 & \text{if } a \in \Sigma_i \text{ or } \Sigma_r \\ 1 & \text{if } a \in \Sigma_c \end{cases}$$

The following claim can be proved using induction on the length of the input and this will prove the second part of the theorem.

Claim 5.2.8 *On input* $x \in \Sigma^*$, M_1 *reaches a set of configuration* $S_1 \subseteq Q^1 \times (\Gamma^1)^*$ *and* M_2 *reaches a set of configurations* $S_2 \subseteq Q^2 \times (\Gamma^2)^*$ *if and only if* M' *reaches* $S_1 \times S_2$.

This proves the part (b) of the theorem. Part (c) trivially follows from (b) (taking all the k functions to be f_1). Part (d) needs extra work. We now prove the part (d).

The idea is similar to the one used for proving closure under multiplication. Here too, we construct product pushdown automata using product of M_1 with itself. To generate $\binom{f_1}{2}$ (= m, say) many accepting paths, the product automata must disallow same runs to be simulated on the two components of the product automata. That is, the number m corresponds to all those pairs of accepting runs (ρ, ρ') of M_1 for which $\rho \neq \rho'$.

This is achieved by maintaining a flag in the state space. This flag is off initially, to indicate that the runs have been similar till now. The flag remains off as long as the moves being made on the two components are the same. The flag is turned on as soon as two different moves are made on the two components of the product automata. At this point, the first component is forced to move to a (lexicographically) smaller state, stack-top pair as compared to the one that the second component would go to. Once the flag is on, it remains so for the rest of the run, and all the moves are allowed on both the components.

We now give a formal description of the automata N. $M' = (Q = (Q^1 \times Q^1 \times \{0,1\}), q_0 = (q_0^1, q_0^1, 0), F = (F^1 \times F^1 \times \{0,1\}), \Sigma, \Gamma = \Gamma^1 \times \Gamma^1, \delta)$. Here we assume lexicographic ordering on the state and stack alphabet. We assume that $\perp$ is defined to be strictly smaller than all the letter in Γ^1. And we assume (without loss of generality) that states are all smaller than the stack letters. This gives a total order, say $<_{M^1}$, on the state, stack-top pairs.

The transition function δ is defined as follows:

For all $a \in \Sigma_c$:

$(q,q,0) \xrightarrow{a} (p,p,0)(X,X) \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} pX \in \delta^1$

$(q,q,0) \xrightarrow{a} (p,p',1)(X,X') \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} pX \in \delta^1$, $q \xrightarrow{a} p'X' \in \delta^1$ and $pX <_{M^1} p'X'$

$(q,q',1) \xrightarrow{a} (p,p',1)(X,X') \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} pX \in \delta^1$ and $q' \xrightarrow{a} p'X' \in \delta^1$

For all $a \in \Sigma_i$:

$(q,q,0) \xrightarrow{a} (p,p,0) \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} p \in \delta^1$

$(q,q,0) \xrightarrow{a} (p,p',1) \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} p \in \delta^1$, $q \xrightarrow{a} p' \in \delta^1$ and $p <_{M^1} p'$

$(q,q',1) \xrightarrow{a} (p,p',1) \in \delta \quad \Leftrightarrow \quad q \xrightarrow{a} p \in \delta^1$ and $q' \xrightarrow{a} p' \in \delta^1$

For all $a \in \Sigma_r$:

$(q,q,0)(X,X) \xrightarrow{a} (p,p,0) \in \delta \quad \Leftrightarrow \quad qX \xrightarrow{a} p \in \delta^1 \; X \in \Gamma^1 \cup \{\perp\}$

$(q,q,0)(X,X') \xrightarrow{a} (p,p',1) \in \delta \quad \Leftrightarrow \quad qX \xrightarrow{a} p \in \delta^1$, $qX' \xrightarrow{a} p' \in \delta^1$ and $pX <_{M^1} p'X'$

$(q,q',1)(X,X') \xrightarrow{a} (p,p',1) \in \delta \quad \Leftrightarrow \quad qX \xrightarrow{a} p \in \delta^1$ and $q'X' \xrightarrow{a} p' \in \delta^1$

The following claim can be proved by induction for correctness of the construction:

Claim 5.2.9 *On input* $x \in \Sigma^*$, M_1 *reaches a set of configurations* $S_1 \subseteq Q^1 \times (\Gamma^1)^*$ *if* *and only if* N *reaches the following set of configurations*

$$S'_1 = \left\{ ((q,q',1),(\gamma,\gamma')) \;\middle|\; \begin{array}{c} (q,\gamma) <_{M^1} (q',\gamma') \\ (q,\gamma) \in S_1 \text{ and } (q',\gamma') \in S_1 \end{array} \right\} \\ \bigcup \\ \{((q,q,0),(\gamma,\gamma)) \mid (q,\gamma) \in S_1\}$$

□

Note that the input need not be well-matched for these closure properties to hold.

Type 2 closure properties

The main theorem we prove in this section can be stated as follows:

Theorem 5.2.10 *For a fixed* $k \in \mathbb{N}$*, let* $M_1, M_2, \ldots, M_k$ *be* VPA *over an alphabet* Σ *having possibly different partitions of the input alphabet. Let* $f_1, f_2, \ldots, f_k$ *be functions from* Σ^* *to* $\mathbb{Z}$*. If* $\forall x : x \in \Sigma^*$ *and* $\forall i : 1 \leqslant i \leqslant k$, $f_i(x) = \#acc_{M_i}(x)$*, then,*

- *there is a* VPA M *over another alphabet* Σ' *and a function* $g : \Sigma^* \to \Sigma'^*$ *in* AC^0 *such that for all* $x \in \Sigma^*$, $f_1(x) + f_2(x) + \ldots + f_k(x) = \#acc_M(g(x))$.
- *there is a* VPA M' *over another alphabet* Σ' *and a function* $g : \Sigma^* \to \Sigma'^*$ *in* AC^0 *such that for all* $x \in \Sigma^*$, $f_1(x) \times f_2(x) \times \ldots \times f_k(x) = \#acc_{M'}(g(x))$.
- *If* $f'(x, 1^n)$ *is defined as* $(f_1(x))^n$*, then (for* $f_1 \in$ #VPA *as assumed above) there is a* VPA N *and a function* $g : (\Sigma^* \times 1^*) \to \Sigma'^*$ *in* AC^0 *such that for all* $x \in \Sigma^*$ *and* $n \in \mathbb{N}$, $f'(x, 1^n) = \#acc_N(g(x, 1^n))$.

Proof: We prove the first two parts of the theorem for $k = 2$. It is easy to generalize the following techniques for any fixed k. The VPA we design for addition here, has its input alphabet as two copies of Σ. This can be thought of as the letters from Σ being colored red and black $\Sigma_{red}, \Sigma_{black}$. The AC^0 reduction thus makes two copies of input x separated by a new delimiter, say \$, with the first copy being red and the second black. The VPA we design works on the union of $\Sigma_{red}, \Sigma_{black}, \{\$\}$, Σ' say, as

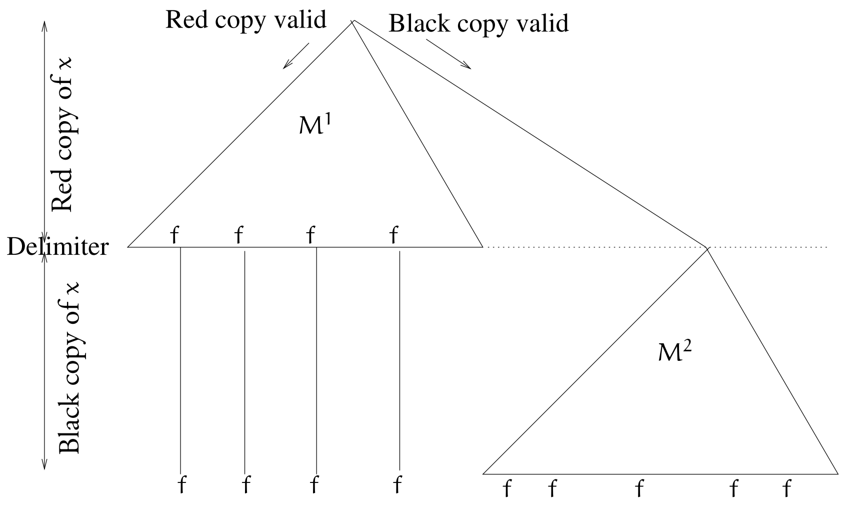


Figure 5.1: The number of accepting paths in M is the sum of the number of accepting paths in M^1 and M^2.

the input alphabet. Within the red and the black copy, the partition respects that of Σ^1 and Σ^2, respectively. The function f_1+f_2 is simulated as $[f_1 \cdot 1 + 1 \cdot f_2]$. The machine, in its initial state, nondeterministically guesses the copy of x which is valid for the run. Suppose for a run, the red copy is guessed to be valid, then it simulates M^1 on the first copy. Upon seeing a delimiter, the machine deterministically pushes or pops depending on the partition of the letters read from the second copy. But the state does not change. Thus if it reaches an accept state of M^1 during the run, it remains in it without increasing the number of accepting paths any further. Symmetrically, if the machine guesses the black copy to be valid, it deterministically pushes or pops a fake symbol on the stack during the red copy and upon a reading a delimiter, it starts simulating M^2 on the black copy treating the fake stack symbol similar to the bottom of the stack. (This is to avoid the issues regarding the well-matchedness. If we wish to make the string well-matched we may have to use TC^0 reductions. But as seen here, that is not necessary)

Figure 5.1 gives an intuitive idea as to why the construction should work.

For multiplication, the VPA M' again has Σ' as its input alphabet. For this machine the stack alphabet is a union of Γ^1 and Γ^2, and it is important that they are disjoint. If not, the copies of stack alphabet are also colored. The machine simulates the moves of M^1 on the first copy of x. Upon reading the delimiter the accepting paths of this simulation continue with simulating M^2. The letters from the alphabet Γ^1 are treated as bottom of the stack marker during the simulation of M^2. This achieves the multiplication.

This same idea is extended further in order to obtain the third part of the theorem.

To prove the third part, *i.e.,* powering of the #VPA functions can be computed in AC^0 closure of #VPA, we give a reduction. The reduction itself is computable in AC^0. Given a number n (in unary), the task is to compute f^n. First we convert every string x into a string x' containing n copies of x separated by delimiters, say \$, and an extra string of padding letter A which was not originally in Σ. This addition letter is treated as a pop letter. The reduction produces a string x' which is $(xA^{|x|}\$)^n$. This reduction can be performed in AC^0. We design a machine N over an alphabet $\Sigma' = \Sigma \cup \{\$, A\}$.

The machine N simulates the moves of M^1. The additional internal moves on \$ take the state control from a final state to the initial state and get stuck on all the other states. Suppose on x, on a run, machine M^1 reaches a final state with an empty stack. The machine N restarts the simulation of the machine M, starting from the next copy of x in x' after having read the delimiter \$ which follows the current copy of x. This process repeats itself (every time the machine reaches an accept state at the end of reading a copy of x, *i.e.,* at most n times) thereby resulting in powering the number of accepting paths. Assuming that M^1 ends up with an empty stack after processing x, the additional padding with a string of A is not needed. However in general, on an input x, M^1 need not end up with an empty stack. The padding is added to take care of the remnant stack after every simulation of M^1. The stack can be at most of height $|x|$ and the added pop letter A makes sure that the stack is empty before the second simulation of M^1 begins. □

The results we obtained are summarized in Table 5.2:

5.3 Realtime height-deterministic PDA.

The aspect of rhPDA which interests us in this study of counting problems is that it is a nondeterministic model capturing the deterministic class LogDCFL. It thus provides a way of arithmetizing LogDCFL, simply by counting the number of accepting paths on each word in an rhPDA. We call the class of such functions #rhPDA. In

Types	Operations	Complexity
Same partition of the input alphabet for all the machines	$\sum_{i=1}^{k} f_i$, $\prod_{i=1}^{k} f_i$, f^k, $\left(\begin{array}{c} f \\ k \end{array}\right)$. for constant k	in #VPA
Different partition of the input alphabet	$\sum_{i=1}^{k} f_i$, $\prod_{i=1}^{k} f_i$, f^n for constant k, and given n.	AC^0 reducible to #VPA

Table 5.2: Analysis of #VPA functions

particular, we consider the classes #rhPDA(FST) and #rhPDA(PDT).

5.3.1 Bounds on #rhPDA(FST)

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

Theorem 5.3.1 #rhPDA(FST) $\equiv$ #VPA *(via* NC^1 *reductions).*

Proof: By Lemma 4.5.9, we know that any strong SSPDA can be converted into an equivalent one-way-strong SSPDA. And from Lemma 4.5.8 we know that the membership problem for a one-way-strong SSPDA reduces to that of a VPA. Thus, combining these lemmas and Lemma 4.5.7, membership problem for any SSPDA reduces to that of a VPA. In addition, if these reductions are parsimonious, we would establish the required result.

The transformation from strong SSPDA to one-way-strong SSPDA is a simple rewriting of the configurations of the strong SSPDA and hence is parsimonious. The reductions in Lemma 4.5.8 and Lemma 4.5.7 are padding functions. The additional moves on the padding letters in the new machines are deterministic. Hence, again the number of paths are preserved. Thus the theorem.

□

5.3.2 Bounds on #rhPDA(PDT)

The main theorem of this section, Theorem 5.3.2, shows that the membership problem and the counting problem for rhPDA have the same complexity, a situation rather unusual for nondeterministic complexity classes. This calls for some discussion.

We have defined an arithmetization for LogDCFL and as we show below, we have upper bounded its complexity by the same complexity class LogDCFL. Thus, this arithmetization does not yield functions which are more powerful. Usually the arithmetizations defined for nondeterministic complexity classes seem to have higher complexity than their Boolean equivalents. It may be interesting to define and study other arithmetizations of LogDCFL.

If we compare this result to arithmetizations of deterministic complexity classes, in particular $\#\mathsf{NC}^1$ we might find this arithmetization interesting. For years, the problem whether $\#\mathsf{NC}^1 = \mathsf{NC}^1$ has been open. This problem is still open, but the gap between $\#\mathsf{NC}^1$ and NC^1 is very small unlike the gap between NP and #P. Many complexity theorists believe this to be true. That is, they believe that when one arithmetizes inherently deterministic class, the class of functions obtained cannot be too much more powerful. Which is what we have for the class LogDCFL.

Theorem 5.3.2 #rhPDA *is in* LogDCFL.

The proof proceeds in several stages, similar to the proof for LogDCFL upper bound of #VPA. 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 using Lemma 5.2.5 specialized to rhPDA as given below:

Lemma 5.3.3 (folklore) *Let* P *be a fixed* rhPDA*. There is a constant* $c \geqslant 0$*, 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* i*th prime number.)*

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

We now need to bound the complexity of the oracle language L_{res} itself. L_{res} itself can be computed by a DAuxPDA with oracle access to the height function g_T. In fact, the procedure from Section 5.2.2 can be described as a LogDCFL algorithm making oracle calls to the height function.

Lemma 5.3.4 *If* P *is any* rhPDA *and* T *a* PDT *computing its height function, then* L_{res} *is in* $\mathsf{LogDCFL}^{g_T}$.

In the case of VPA, the height function is in TC^0. However, here the height function is in LogDCFL(see Lemma 4.6.8). Lemma 4.6.8 and Lemma 5.3.4 together imply that L_{res} is in LogDCFL(LogDCFL). This is not adequate for us, since it is not known whether LogDCFL(LogDCFL) $\subseteq$ 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 5.3.5 *Let* $L(M^A)$ *be the language accepted by a poly-time* DAuxPDA M *which makes* $O(\log n)$*-bits oracle queries to a language* $A \in$ LogDCFL. *Then* $L(M^A) \in$ LogDCFL.

Proof: Consider a 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. $\square$

As L(LogDCFL) equals LogDCFL, combining these lemmas we get Theorem 5.3.2

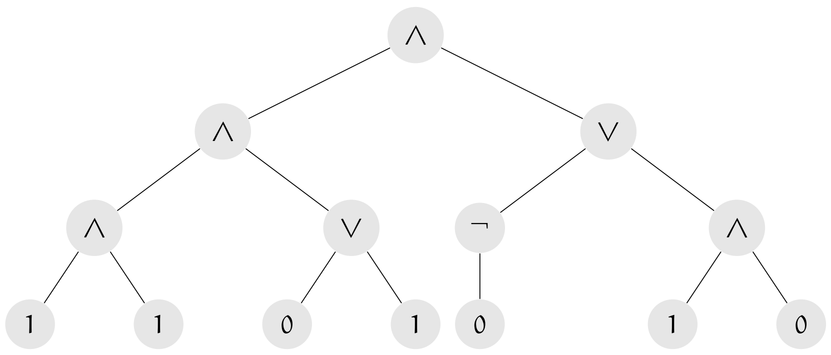
5.4 Barriers in improving parallelizability for counting problem of VPLs

The best known bound for the membership problem for VPLs, NC^1, is by [25]. Currently, the best known bound we have for #VPL is LogDCFL which is obtained by tinkering the algorithm of [15]. We tried to improve this bound by trying to modify the Buss's algorithm [16], which is at the heart of the NC^1 algorithm of [25]. We have not managed to obtain any improvement in the bound for #VPL. In the following section, we present Buss's algorithm. In Section 5.4.2 we present Buss's algorithm adapted for proving NC^1 upper bound for $\mathsf{MEM}(\mathsf{VPL})$. In Section 5.4.3 we discuss the hurdles we faced while trying to generalize Buss's algorithm.

5.4.1 Boolean sentence value problem in NC^1 [16]

A *Boolean sentence* is a tree with leaves labeled by $\{0, 1\}$ and internal nodes labeled by the Boolean operators $\vee, \wedge, \neg$. The sentence may be specified either by pointers or as a string over the alphabet $\{\vee, \wedge, \neg, (,), 0, 1\}$. The complexity of the problem is sensitive to the specification of the input. Buss assumes that the input is specified as a string. We will assume without loss of generality that arity of each Boolean operator is at most 2.

Example 5.4.1 *Consider a Boolean formula given below:*



The above sentence specified as a string in infix notation can be given as:
$(((1 \wedge 1) \wedge (0 \vee 1)) \wedge ((\neg 0) \vee (1 \wedge 0)))$

We present the approach of Buss's algorithm from [17]. The main steps involved in getting the NC^1 bound are as follows:

1. Convert the given Boolean sentence into a normal form called Postfix Longest Operand First form (PLOF form).
2. Play Pebbler-Challenger game on this sentence such that Pebbler has a winning strategy if and only if the sentence evaluates to 1.
3. Prove the following two things about the game:
    - The game can be played in NC^1.

- Given the moves of the game, the validity of the moves and the winner can be decided in NC^1.

The conversion of the given sentence into PLOF is straightforward and we will not reproduce the details of this conversion here. Our focus will be the second step. We will first play a very easy game on the sentence and note that analysis of this game does not give NC^1 bound. After this, we will describe the extra work done in [16] in order to get this bound.

PLOF is the normal form for Boolean sentences such that the operands come after the corresponding operands. Also if the two operands are not of the same size then the longer operand comes first. (Note that the operators are commutative.) The sentence from Example 5.4.1 can be written in PLOF as follows (we assume without loss of generality that 1 is longer than 0): $11 \wedge 10 \vee \wedge 10 \wedge 0 \neg \vee \wedge$.

It is essentially a type of re-writing which additionally involves counting the sizes of the operands attached to each operator. Buss defines such a function, denoted as *count* and proves that it can be computed in NC^1. After this, the conversion to PLOF follows easily.

PEBBLER-CHALLENGER **game** We now specify the rules of a very easy game being played on the Boolean sentence. The game consists of multiple rounds. A round consists of a PEBBLER's move followed by a CHALLENGER's move. The PEBBLER is allowed to pebble the node with a colored pebble. The colors of the pebbles are either 0 or 1. The CHALLENGER is allowed to challenge one of the pebbled nodes. The game is described below:

Round 1	
PEBBLER:	Pebbles the root with a pebble of color 1.
CHALLENGER:	Challenges the root.
[Round i]	
PEBBLER:	Let v be the node challenged in round $i-1$. PEBBLER guesses a divider u for the subtree rooted at v. Also guesses its value as either 0 or 1. Note that the PEBBLER can not go below any earlier pebbled node.
CHALLENGER:	Either decides to continue challenging v or challenges u.

The game ends when all the children of the challenged node are pebbled. If the sentence value is 1, then for every round PEBBLER has a way to pebble the divider node correctly with its value. Hence, finally, the challenged node will have children consistently labeled by PEBBLER's pebbles. If sentence value is 0, the root is pebbled wrongly and there will be a witnessing subtree that will force the root value to be zero. By staying in the original position or by shifting (depending on whether the newer pebbled value is right or wrong respectively) CHALLENGER will be able to win the game. Thus, the PEBBLER wins the game if and only if the value of the sentence is 1.

The PEBBLER's moves are simply guesses. PEBBLER needs that at least one guess is right about the values. The CHALLENGER's moves, on the other hand, are verifications of the pebbled values. In this game, the PEBBLER's guesses can be simulated by an $\vee$ gate and CHALLENGER's verifications can be simulated by an $\wedge$ gate. This gives us the circuit over $\wedge, \vee$. If the PEBBLER guesses the dividers correctly, the circuit corresponding to correct guesses will be of depth $O(\log n)$. However, the in-degree of $\vee$ gates will be high due to guesses involving the labels of the divider nodes. Thus, the circuit we will obtain will be a SAC^1 circuit as opposed to NC^1.

This simple game now needs a modification in order to make the in-degree of the $\vee$ gates small. This is in fact the main contribution of Buss's work. It is clear that to achieve depth of $O(\log n)$ it is important that the size of the subproblem reduces by a constant fraction at every round. In order to achieve this, Buss uses the structure of the sentence as a string, as opposed to thinking of it as a tree.

The whole string [1] of length 2^k for some k is cut into three parts of size 2^{k-1} each. These cuts are predecided and depend only on the length of the sentence (and not on the specific sentence itself). There are four nodes marked which are pebbled in each round. Once the length of the sentence and round number are fixed, these four places also get fixed. PEBBLER now only specifies pebble values for these nodes. The CHALLENGER's moves are exactly same as in the previous game. Thus, PEBBLER's guesses are now constant bit long, giving us the desired NC^1 bound.

For details regarding the exact nodes to be pebbled, we refer the reader to [17].

[1] it is possible to assume that the length of the sentence is a power of 2. If not, one can pad the string with $\neg$ and make it a power of 2. The final value will be negated if the number of $\neg$ gates added in this process is odd.

5.4.2 Modification to Buss's algorithm for MEM(VPL) in NC^1

We recall some properties of the sentence that Dymond obtains from given input w for the problem MEM(VPL) from Section 4.3.1. The sentence is over unary operators $\mathrm{Ext}_{\{a,b\}}$, $\mathrm{Ext}_{\{c\}}$, and binary operator compose, $\circ$. Note that, the compose operator is not commutative. Therefore, to convert it into PLOF Dymond [25] introduces another operator. We will call it ord-compose and denote it by $\circ'$. In [25] $a \circ' b$ is defined as $b \circ a$. To make the length of the sentence a power of 2, we define another operator called id which is an identity map. $\mathrm{id}(a,b) = (a,b)$.

Now the sentence in the PLOF form has five operators, each a finite relation.

The Buss's Pebbler-Challenger game on this now does not use only 2 colored pebbles. The range of each operator is $Q \times Q$. The Pebbler puts a pebble on a node to indicate the value taken by the node upon evaluation. Hence, the Pebbler needs $2^{|Q|^2}$-many pebbles.

In Round 0, the Pebbler pebbles the root with $S_0 = \{(q_0, q) \mid q \in F\}$. The Challenger challenges the root. For round i, depending upon the Challenger's move in the previous round, Pebbler decides to shift to the appropriate portion of the sentence and for that subsection of the sentence pebbles 4 nodes with pebbles colored with subsets of $Q \times Q$. The rest of the rules of the game remain the same.

Lemma 5.4.2 Pebbler *wins if and only if* $w \in L(M)$ *if and only if* $(q_0, \perp) \to^w (q, \perp)$ *for some* $q \in F$.

If $(q_0, \perp) \to^w (q, \perp)$, then starting with root pebbled as S_0, Pebbler has a way to pebble each node u correctly with the pebble colored S, where S is the value of the node u. Else, Challenger can keep shifting the focus of the game to that subsentence which eventually leads to contradiction (the root of the subsentence, not necessarily maximal, due to which the root of the whole sentence cannot evaluate to S_0).

5.4.3 Arithmetizing Buss's technique

We now discuss how the above sentence over $\mathrm{Ext}_{\{a,b\}}$, $\mathrm{Ext}_{\{c\}}$, and $\circ$ is already good enough to compute #VPA functions (here, $a \in \Sigma_c, b \in \Sigma_r, c \in \Sigma_l$ as in Section 4.3.1). Let $T_{i,j}$ be $Q \times Q$ matrices over $\mathbb{N}$ as defined in Section 5.2.1 (referred to as T_r).

Redefine the $\circ$ operator as $\times$ (matrix multiplication) and change the semantics of $\mathrm{Ext}_{\{a,b\}}$ ($\mathrm{Ext}_{\{c\}}$) to the following: It takes as an input a $Q \times Q$ integer matrix for an interval (i, j) and gives out a matrix corresponding to the interval $(i-1, j+1)$ ($(i, j+1)$ respectively). It is defined in such a way that the output matrix has k in its (q, q')th entry if and only if there are k different ways in which VPA goes from state q to state q' on the interval $(i-1, j+1)$ ($(i, j+1)$, respectively) assuming the input table T is correctly filled (*i.e.,* entries in T count paths as defined in Section 5.2.1).

The following method of filling up the entries will ensure such a definition: fill the entry $\mathrm{Ext}_{\{a,b\}}[q, q']$ with

$$\sum_{q_1, q_2 \in Q} f_{qq'}(q_1, q_2),$$

where $f_{qq'}(q_1, q_2)$ equals $l \times m$ for $l =$ the number of $A \in \Gamma$ such that $\delta(q, a) = (q_1, A)$ (*i.e.,* the number of different stack-tops which allow for a successful extension of the existing interval) and $\delta(q_2, A, b) = q'$ and $m = T[q_1, q_2]$ (*i.e.,* the number of different ways in which the VPA goes from state q_1 to state q_2 as per the table T). (Similarly, $\mathrm{Ext}_{\{c\}}[q, q']$ can be filled.)

It is easy to see that, with this new definition of the operators, the same sentence as constructed by [25] will evaluate #VPA. But for these operators, there is no known nice algorithm that will evaluate the sentence in $\#\mathsf{NC}^1$. We do not even know a logspace algorithm for this. In other words, the depth reduction for such sentences is not known.

The following two things seem to be at the heart of the problems encountered while trying to arithmetize Buss's approach:

- the $\circ$ operator is not commutative. However, this is not a big hurdle. There are known methods for evaluating sentences over non-commutative multiplications [17].
- there seems to be no obvious way to express the Ext operators as small circuit over $+$ and $\times$.

5.5 Concluding Remarks

- We have the same, LogDCFL, upper bound for both #rhPDA and #VPA. Now we discuss two different issues arising from this result.

    1. We know that the membership problem and hence the counting problem for rhPDA is hard for LogDCFL. Therefore, this result gives a class of arithmetic functions, *i.e.,* functions whose range is $\mathbb{Z}$, which are equivalent to a Boolean deterministic complexity class LogDCFL. This is interesting because no such set of arithmetic functions was known before this work. It gives more avenues to understand the class LogDCFL. The result also proves that these counting functions are only as powerful as the functions computable by LogDCFL. The arithmetic functions corresponding to nondeterministic Boolean complexity classes are usually much more powerful than their Boolean counterparts. But arithmetic functions corresponding to a deterministic complexity class are believed to be as powerful as the deterministic complexity class itself. This is indeed the case, for our result.
    2. This also indicates that there is a scope for improvement in the upper bound for the counting problem for VPLs. Our attempts for the same have failed on various fronts. We give an overview of various attempts we made to improve the upper bound for the counting problem for VPLs and the intuition behind why they could not go through. It is an interesting open problem to improve this upper bound or prove impossibility of any such improvement.

- Another interesting open question is to bound the complexity of the counting problem for k phase visible multi-stack machines, MVPA_k. In order to prove LogCFL upper bound for their membership problem, the following steps were involved:

    1. Reduction from $\mathsf{MEM}(\mathsf{MVPL}_k)$ to $\mathsf{MEM}(\mathsf{PD}_k)$, Theorem 4.7.1.
    2. Conversion of the PD_k into a D^k grammar, [21].
    3. Conversion of the D^k grammar into a normal form grammar, [21].

4. Finally giving a LogCFL upper bound for parsing the normal form grammars, Section 3.4.4 and Section 3.4.5.

It is easy to see that the reduction from $\mathsf{MEM}(\mathsf{MVPL}_k)$ to $\mathsf{MEM}(\mathsf{PD}_k)$ is parsimonious. If all the above steps are parsimonious, then we get a #LogCFL upper bound for $\#\mathsf{MVPL}_k$.

Bibliography

[1] M. Agrawal and V. Vinay. Arithmetic circuits: A chasm at depth four. In *FOCS 2008*, pages 67–75, 2008.

[2] E. Allender. The division breakthroughs. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 74, 2001.

[3] E. Allender and M. Koucký. Amplifying lower bounds by means of self-reducibility. In *Proceedings of the 2008 IEEE 23rd Annual Conference on Computational Complexity*, pages 31–40, 2008.

[4] Eric Allender, Jia Jiao, Meena Mahajan, and V Vinay. Non-commutative arithmetic circuits: depth reduction and size lower bounds. *Theoretical Computer Science*, 209:47–86, 1998.

[5] R. Alur and P. Madhusudan. Visibly pushdown languages. In *STOC*, pages 202–211, 2004.

[6] R. Alur and P. Madhusudan. Adding nesting structure to words. In *Tenth International Conference on Developments in Language Theory*, pages 1–13, 2006.

[7] Carme Álvarez and Birgit Jenner. A very hard log-space counting class. *Theor. Comput. Sci.*, 107(1):3–30, 1993.

[8] Vineet Bafna and Pavel Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, may 1998. preliminary version in SODA 95, pp. 614–623.

[9] D.A. Barrington. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[10] David.A.Mix Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1. *JCSS*, 38(1):150–164, 1989.

[11] W. W. Bein, L. L. Larmore, S. Latifi, and I. H. Sudborough. Block sorting is hard. *International Journal of Foundations of Computer Science*, 14(3):425–437,

june 2003. Special issue for ISPAAN 2002 Symp on Parallel Architectures, Algorithms and Networks.

[12] W.W. Bein, L.L. Larmore, L. Morales, and I.H. Sudborough. A faster and simpler 2-approximation algorithm for block sorting. In *Proc. of 15th International Symposium on Fundamentals of Computation Theory, LNCS 3623*, pages 115–124, 2005.

[13] Andreas Blass and Yuri Gurevich. A note on nested words. Technical Report MSR-TR-2006-139, Microsoft Research, October 2006.

[14] A. Borodin, S. Cook, P. Dymond, W. Ruzzo, and M. Tompa. Two applications of inductive counting for complementation problems. *SIAM Journal of Computation*, 18(3):559–578, 1989.

[15] B. Von Braunmühl and R. Verbeek. Input-driven languages are recognized in $\log n$ space. In *Proc. FCT Conference, LNCS*, pages 40–51, 1983.

[16] S. Buss. The Boolean formula value problem is in ALOGTIME. In *STOC*, pages 123–131, 1987.

[17] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput*, 21:755–780, 1992.

[18] Dario Carotenuto, Aniello Murano, and Adriano Peron. 2-visibly pushdown automata. In *11th international conference on Developments in Language Theory 2007*, pages 132–144, 2007.

[19] Didier Caucal. Synchronization of pushdown automata. In *Developments in Language Theory*, pages 120–132, 2006.

[20] A. Cherubini, L. Breveglieri, C. Citrini, and S. Crespi Reghizzi. Multipushdown languages and grammars. *International Journal of Foundations of Computer Science*, 7(3):253–292, 1996.

[21] Alessandra Cherubini and Pierluigi San Pietro. A polynomial-time parsing algorithm for a class of non-deterministic two-stack automata. In *4th Italian Conference on Theoretical Computer Science*, pages 150–164, 1992.

[22] Alessandra Cherubini and Pierluigi San Pietro. A polynomial-time parsing algorithm for k-depth languages. *Journal of Computer and System Sciences*, 52(1):61–79, 1996.

[23] A Chiu, G Davida, and B Litow. Division in logspace-uniform NC^1. *RAIRO Theoretical Informatics and Applications*, 35:259–276, 2001.

[24] D. A. Christie. *Genome Rearrangement Problems*. PhD thesis, Univ. of Glasgow, 1999.

[25] Patrick W. Dymond. Input-driven languages are in $\log n$ depth. *Information Processing Letters*, 26:247–250, 1988.

[26] I. Elias and T. Hartman. A 1.375 approximation algorithm for sorting by transpositions. In *Proc. 5th International Workshop on Algorithms in Bioinformatics WABI, LNCS 3692*, 2005.

[27] R. Gobi, S. Latifi, and W.W. Bein. Adaptive sorting algorithms for evaluation of automatic zoning employed in OCR devices. In *Proceedings of the 2000 International Conference on Imaging Science, Systems, and Technology*, pages 253–259, 2000.

[28] Tzvika Hartman. A simpler 1.5 approximation algorithm for sorting by transpositions. In *Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching, LNCS 2676*, pages 156–169. Springer-Verlag, 2003.

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

[30] Markus Holzer and Klaus-Jörn Lange. On the complexities of linear LL(1) and LR(1) grammars. In *FCT '93: Proceedings of the 9th International Symposium on Fundamentals of Computation Theory*, pages 299–308, London, UK, 1993. Springer.

[31] A. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 2001.

[32] J. Jiao. Some questions concerning circuit counting classes and other low level complexity classes. In *Manuscript*, 1992.

[33] J. Kanai, S.V. Rice, and T.A. Nartker. Automatic evaluation of OCR zoning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17:86–90, 1995.

[34] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *22nd Annual IEEE Symposium on Logic in Computer Science LICS*, pages 161–170, 2007.

[35] S. Latifi. How can permutations be used in the evaluation of automatic zoning evaluation? In *ICEE 1993, Amir Kabir University*, 1993.

[36] M. Mahajan, R. Rama, V. Raman, and S. Vijayakumar. Merging and sorting by strip moves. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), LNCS 2914*, pages 314–325. Springer-Verlag, 2003.

[37] M. Mahajan, R. Rama, V. Raman, and S. Vijayakumar. Approximate block sorting. *International Journal of Foundations of Computer Science*, page to appear, 2005.

[38] Meena Mahajan, Raghavan Rama, and S. Vijayakumar. Block sorting: a characterization and some heuristics. *Nordic J. of Computing*, 14(1):126–150, 2007.

[39] P. McKenzie, K. Reinhardt, and V. Vinay. Circuits and context-free languages. In *5th Annual International Computing and Combinatorics Conference (COCOON)*, July 1999.

[40] K. Mehlhorn. Pebbling mountain ranges and its application to DCFL recognition. In *Proc. 7th ICALP*, pages 422–432, 1980.

[41] Rolf Niedermeier and Peter Rossmanith. Unambiguous auxiliary pushdown automata and semi-unbounded fan-in circuits. *Information and Computation*, 118(2):227–245, 1995.

[42] Dirk Nowotka and Jirí Srba. Height-deterministic pushdown automata. In *MFCS*, pages 125–134, 2007.

[43] P. San Pietro. 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.

[44] Omer Reingold. Undirected ST-connectivity in log-space. In *STOC '05: Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 376–385, 2005.

[45] W.L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21:218–235, 1980.

[46] J. Savage. *The Complexity of Computing*. John Wiley and Sons, 1976.

[47] Walter J Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[48] I. Sudborough. On the tape complexity of deterministic context-free language. *Journal of Association of Computing Machinery*, 25(3):405–414, 1978.

[49] I. H. Sudborough. A note on tape-bounded complexity classes and linear context-free languages. *J. ACM*, 22(4):499–500, 1975.

[50] L. G. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[51] H. Venkateswaran. Properties that characterize LogCFL. *Journal of Computer and System Sciences*, 43:380–404, 1991.

[52] V Vinay. Counting auxiliary pushdown automata and semi-unbounded arithmetic circuits. In *Proceedings of 6th Structure in Complexity Theory Conference*, pages 270–284, 1991.

[53] H. Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York Inc., 1999.

A list of publications included in the thesis

- **Membership testing: Removing extra stacks from multi-stack pushdown automata.**
  Nutan Limaye and Meena Mahajan.
  in Proceedings of 3rd International Conference on Language and Automata Theory and Applications LATA, April 2009, Tarragona, Spain. Springer-Verlag Lecture Notes in Computer Science series Volume 5457 pp.493–504.

- **On the complexity of membership and counting in height-deterministic pushdown automata.**
  Nutan Limaye, Meena Mahajan, and Antoine Meyer.
  in Proceedings of 3rd International Computer Science Symposium in Russia CSR, June 7–12, 2008, Moscow. Springer-Verlag Lecture Notes in Computer Science series Volume 5010 pp.240–251.

- **Arithmetizing Classes around NC^1 and L.**
  Nutan Limaye, Meena Mahajan, and B. V. Raghavendra Rao.
  to appear in Theory of Computing systems , special issue for STACS 2007.

A list of other publications

- **A Log-space Algorithm for Canonization of Planar Graphs**
  Samir Datta, Nutan Limaye, Prajakta Nimbhorkar, Thomas Thierauf, and Fabian Wagner
  in Proceedings of the 2009 24th Annual IEEE Conference on Computational Complexity CCC, July 15–18, 2009, Paris, France. pp. 203–214.

- **Longest paths in planar DAGs in unambiguous logspace.**
  Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar.
  to appear in the special issue of Chicago Journal of Theoretical Computer Science, CJTCS

- 3**-connected planar graph isomorphism in Logspace.**
  Samir Datta, Nutan Limaye, and Prajakta Nimbhorkar.
  appeared in Proceedings of Foundations of Software Technology and Theoretical Computer Science, FSTTCS, December 9-11, 2008, Bangalore, India.

- **Planarity, Determinants, Permanents, and (Unique) Perfect Matchings.**
  Samir Datta, Raghav Kulkarni, Nutan Limaye, and Meena Mahajan.
  to appear in the journal Transactions on Computation Theory.
- **Upper Bounds for Monotone Planar Circuit Value and Variants.**
  Nutan Limaye, Meena Mahajan, and Jayalal Sarma M. N.
  appeared in the journal Computational Complexity, Volume 18, 2009, pp. 377–412.