


Learning Deterministic Visibly Pushdown Automata Under Accessible Stack

Jakub Michaliszyn ✉ 

University of Wrocław, Poland

Jan Otop ✉ 

University of Wrocław, Poland

Abstract

We study the problem of active learning deterministic visibly pushdown automata. We show that in the classical L^* -setting, efficient active learning algorithms are not possible. To overcome this difficulty, we propose the accessible stack setting, where the algorithm has the read and write access to the stack. In this setting, we show that active learning can be done in polynomial time in the size of the target automaton and the counterexamples provided by the teacher. As counterexamples of exponential size are inevitable, we consider an algorithm working with words in a compressed representation via (visibly) Straight-Line Programs. Employing compression allows us to obtain an algorithm where the teacher and the learner work in time polynomial in the size of the target automaton alone.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory

Keywords and phrases visibly pushdown automata, automata inference, minimization

Digital Object Identifier 10.4230/LIPIcs.MFCS.2022.74

Funding This work was supported by the National Science Centre (NCN), Poland under grant 2020/39/B/ST6/00521.

1 Introduction

Visibly pushdown automata (VPA), also known as input-driven pushdown automata, are a subclass of pushdown automata, in which the type of stack operation is determined by the input letter [33, 4]. This implies that the stack height, but not its content, is determined by the input word. This enables the construction of the *product* VPA from two given VPA. In consequence, the class of visibly pushdown languages recognized by VPA is closed under intersection, union and complement, which makes the universality problem decidable.

Even though VPA can be determinized, *deterministic* VPA (DVPA) enjoy better complexity properties: DVPA recognizing the union, the intersection or the complement of other DVPA can be computed in polynomial time. In consequence, inclusion, equivalence and universality for DVPA can be decided in polynomial time. In contrast, the universality problem for VPA is **ExpTime**-complete, which implies that the exponential blow-up in the complementation (resp., determinization) of VPA is unavoidable. These properties render DVPA attractive for verification [16, 2, 28, 15], XML processing [31], approximating recurrent neural networks [12] and others [22].

For successful applications of DVPA, elimination of redundancy is essential. Furthermore, beyond minimization of a given DVPA [17], the automaton can be given implicitly, or before the minimization it can be too big to be constructed. The problem of minimization without explicit construction of the input DVPA can be addressed with active learning.

Active learning of automata has been originally proposed for deterministic finite-state automata (DFA) [6] with a *minimally adequate teacher*, which is an oracle for a hidden regular language L . *The teacher* answers *membership queries* whether a given word belongs to L , and *equivalence queries* whether a proposed automaton recognizes L , and if not it



© Jakub Michaliszyn and Jan Otop;
licensed under Creative Commons License CC-BY 4.0

47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022).

Editors: Stefan Szeider, Robert Ganian, and Alexandra Silva; Article No. 74; pp. 74:1–74:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

returns a counterexample. There is a polynomial-time algorithm, called the L^* algorithm, which only asks membership and equivalence queries and returns the minimal DFA \mathcal{A}_L recognizing L . Therefore, to construct the minimal DFA \mathcal{A}_L , we only need to be able to answer the above queries; the explicit construction of the input automaton can be avoided.

While the L^* algorithm is versatile and has been generalized to various types of automata, extensions of the L^* algorithm to pushdown languages are elusive [7]. The main problems are the lack of the counterpart of the Myhill-Nerode theorem for pushdown languages (despite some attempts [13]) and the complexity of equivalence checking for DPDA, which is decidable but currently the only upper bound is primitive recursive [41, 24]. However, there is a counterpart of Myhill-Nerode theorem for visibly pushdown languages [3] and equivalence checking for DVPA is decidable in polynomial time. Thus, we focus our research on DVPA.

Still, active learning visibly pushdown languages is elusive. First, the canonical DVPA defined based on the right congruence relation from [3] is not a minimal-size DVPA recognizing a given language; it can be even exponentially larger than a minimal equivalent DVPA [3]. Second, a minimal-size DVPA recognizing a given visibly pushdown language is not unique, which indicates difficulty in improving the definition of the canonical DVPA. Finally, the minimization problem for DVPA is **NP**-complete, which rules out possible generalizations of the L^* algorithm, which would return any minimal-size DVPA (unless **P=NP**). For these reasons, we adapt the active learning framework to learn DVPA rather than their languages, i.e., we consider queries, which depend on the automaton structure.

1.1 Our framework

In our framework, we consider a hidden DVPA, which we call the *target automaton*, and the teacher answering queries about it. The learning algorithm is expected to return a DVPA equivalent to the target DVPA. We assume that we can observe the stack content. To model this, we introduce a third type of queries, called *stack content queries*, which return the stack content of the hidden automaton upon reading a given input.

This alone is insufficient to circumvent the hardness inherited from minimization of DVPA (see Proposition 1). Thus, we additionally assume that we can change the stack content during the computation. To model writing, we employ *control words* consisting of standard letters and special *control letters*. A control letter is of the form \underline{u} , where u is the stack content. Upon reading \underline{u} , the automaton swaps its stack content to u , while retaining the current state. Control words can be sent by the algorithm as parameters for appropriate queries. The ability to swap the stack content to the one provided in the word makes it possible to reveal behavior of the target automaton under certain stack contents.

We first discuss the framework, in which there may be an arbitrary number of control letters altering the stack content. Such an approach directly reduces active learning of DVPA to active learning of regular languages. This technique can be employed for minimization of the product of DVPA while avoiding the need of the construction of the whole product in the first place (Section 5.1). The drawback is that the returned automata may contain states unreachable with any path that obeys stack operations (i.e., the usual runs in DVPA).

To avoid unreachable states, we propose another framework, in which all words contain at most one control letter (Section 6). In this setting, active learning can be done in polynomial time in the size of the target automaton and the counterexamples provided by the teacher. Unfortunately, the length of counterexamples may be exponential, making the whole learning process intractable. We consider two solutions: bounding the length of considered counterexamples (Section 7) and employing grammar-based compression (Section 8). This leads us to a scenario where the teacher and the learning algorithm work in time polynomial in the size of the target automaton alone, learning a DVPA equivalent to the target one.

1.2 Related work

Learning context-free languages has been intensively studied. Typically, the context-free grammars (CFGs) have been considered as the representation of a language rather than pushdown automata. On the negative side, it has been shown that learning a CFG from membership and equivalence queries is intractable under reasonable cryptographic assumptions [7]. To overcome this difficulty, as well as to learn grammars more humanly understandable, learning structurally equivalent grammars has been considered [40, 12]. Two grammars are strongly equivalent [37] if their sets of derivation trees with erased names of non-terminals are the same. The problem with that approach is the complexity of checking strong equivalence, which is in **ExpTime**, but **PSpace**-hard [39]. Furthermore, some grammar-based learning algorithms [40] are based on the L^* algorithm for regular tree languages. Basically, for a CFG G , one can learn a tree language of derivations in G of all words from $L(G)$. However, the minimal deterministic bottom-up tree automaton recognizing all derivation trees in G can be exponentially larger than G . This extends to visibly pushdown languages and such a tree automaton can be exponentially larger than a DVPA recognizing the visibly pushdown language.

Besides [40], there is a large body of work on learning CFGs [26, 43, 5, 42, 19]. Among them, [5] shares some similarities to our approach. However, the framework from [5] can be considered as learning the transitions of a pushdown automaton with known states, while in our approach we learn the set of states and the transitions.

While CFGs fit well into the active learning setting, pushdown automata are better suited and more popular in verification [1]. The above-mentioned algorithms cannot be used to learn automata. Active learning of automata is an active research topic. Initially, active learning has been developed for DFA in the seminal paper by Angluin [6]. Recently, there has been a renewed interest in learning various types of automata [11, 32, 9, 8, 10, 36, 34]. Learning based on syntactic properties of the target automaton has been recently proposed in [35], where a syntactic property called the *loop index* has been employed to overcome the difficulty of learning automata on infinite words.

Learning DVPA is studied in [23]. The algorithm provided there learns the canonical DVPA defined by the congruence relation introduced in [3]. It uses only membership and equivalence queries and works in polynomial time in the size of the canonical automaton. However, the size of the canonical DVPA can be exponential in the size of a minimal DVPA [3]. Therefore, the time and space complexity of the algorithm presented in [23] is exponential.

Another approach to learning VPA from [27] considers VPA with significant stack limitations so that every VPA has a unique minimal equivalent DVPA, computable in polynomial time. For this restricted class, polynomial-time learning using only membership and equivalence queries is possible.

A comprehensive survey of related work has been presented in [23, Chapter 7].

2 Preliminaries

Given a finite alphabet Σ of letters, a *word* w is a finite sequence of letters. For a word w , we define $w[i]$ as the i -th letter of w and $|w|$ as its length. We denote the set of all words over Σ by Σ^* . We use ϵ to denote the empty word.

The *index* of an equivalence relation is the number of its equivalence classes. An equivalence relation \equiv on words Σ^* is a *right congruence* relation if and only if for all words $w_1, w_2 \in \Sigma^*$ and all $a \in \Sigma$, if $w_1 \equiv w_2$, then $w_1a \equiv w_2a$.

Real-time Pushdown automata. A (*non-deterministic*) *real-time pushdown automaton* (rt-PDA) is a tuple consisting of: (1) Σ , an input alphabet, (2) Γ , a finite stack alphabet, (3) Q , a finite set of states, (4) $Q_0 \subseteq Q$, a set of initial states, (5) $\delta \subseteq Q \times \Sigma \times (\Gamma \cup \{\perp\}) \times Q \times \Gamma^*$, a finite transition relation, and (6) $F \subseteq Q$, a set of accepting states. We assume that \perp is never popped from the stack, i.e., for every $(q, a, \perp, q', u) \in \delta$, the stack update u starts with \perp . The size of an rt-PDA \mathcal{A} , denoted by $|\mathcal{A}|$, is defined as the number of states plus the number of transitions, i.e., $|Q| + |\delta|$. An rt-PDA is *deterministic* if $|Q_0| = 1$ and δ is a function from $Q \times \Sigma \times (\Gamma \cup \{\perp\})$ to $Q \times \Gamma^*$. Deterministic rt-PDA are denoted as rt-DPDA.

Semantics of rt-PDA. Consider an rt-PDA $\mathcal{A} = (\Sigma, \Gamma, Q, Q_0, \delta, F)$. A *configuration* of \mathcal{A} is a pair consisting of the current state $q \in Q$ of \mathcal{A} and its current stack content $u \in (\perp \cdot \Gamma^*)$. An rt-PDA defines an infinite-state transition system over its configurations $Q \times (\perp \cdot \Gamma^*)$, where configurations $Q_0 \times \{\perp\}$ are initial. For $a \in \Sigma$, let $\rightarrow_{\mathcal{A}}^a$ be an auxiliary relation defined such that $(q, uB) \rightarrow_{\mathcal{A}}^a (q', uv)$ if $(q, a, B, q', v) \in \delta$, where q, q' are states, uB, uv are stack contents with B being a single (top) symbol. For a word $w = a_1 \dots a_n$, we define $\rightarrow_{\mathcal{A}}^w$ as the composition of relation $\rightarrow_{\mathcal{A}}^{a_1} \dots \rightarrow_{\mathcal{A}}^{a_n}$, i.e., $(q, u_0) \rightarrow_{\mathcal{A}}^w (q', u_n)$ if there exist intermediate configurations $(q_1, u_1), \dots, (q_{n-1}, u_{n-1})$ such that for each $1 \leq i \leq n$ we have $(q_{i-1}, u_{i-1}) \rightarrow_{\mathcal{A}}^{a_i} (q_i, u_i)$.

A word w is *accepted* by \mathcal{A} if there are $q_0 \in Q_0$, $q \in F$ and $u \in \perp \cdot \Gamma^*$ such that $(q_0, \perp) \rightarrow_{\mathcal{A}}^w (q, u)$. The *language recognized by \mathcal{A}* , denoted $\mathcal{L}(\mathcal{A})$, is the set of words accepted by \mathcal{A} . For an rt-DPDA \mathcal{A} , we define the stack content upon reading w , denoted by $SC_{\mathcal{A}}(w)$, as the unique u such that $(q_0, \perp) \rightarrow_{\mathcal{A}}^w (q', u)$ for some q' .

Visibly pushdown automata. Consider a partition of the alphabet Σ into three sets $(\Sigma_c, \Sigma_l, \Sigma_r)$ called, respectively, the sets of *call*, *local* and *return* letters. We say that a rt-PDA $\mathcal{A} = (\Sigma, \Gamma, Q, Q_0, \delta, F)$ is a *visibly pushdown automaton* (VPA) (with respect to the partition $(\Sigma_c, \Sigma_l, \Sigma_r)$) if the following conditions hold:

- transitions over letters from Σ_c (*call transitions*) push a single symbol on the stack and do not depend on the stack content, i.e., if $(q, a, B, q', v) \in \delta$, then $v = BG$, where $G \in \Gamma$ and for all $B' \in \Gamma \cup \{\perp\}$ we have $(q, a, B', q', B'G) \in \delta$,
- transitions over letters from Σ_l (*local transitions*) neither change nor depend on the stack content, i.e., if $(q, a, B, q', v) \in \delta$, then $v = B$ and for all $B' \in \Gamma \cup \{\perp\}$ we have $(q, a, B', q', B') \in \delta$,
- transitions over letters from Σ_r (*return transitions*) pop a single symbol from a nonempty stack; if the stack is empty they leave \perp on the stack.

Since VPA are a subclass of rt-PDA, the definitions of size, determinacy, and semantics carry over from rt-PDA to VPA. Since DVPA are a subclass of rt-DPDA, the stack content upon reading w , $SC_{\mathcal{A}}(w)$, is well defined.

3 The L^* algorithm

We briefly describe a variant of the L^* algorithm for active learning of a hidden regular language L [6], which is similar to the algorithms we propose later on. To learn the language L , the algorithm asks the following queries to an oracle called *the teacher*:

- *membership queries*: given $w \in \Sigma^*$, is $w \in L?$, and
- *equivalence queries*: given a DFA \mathcal{A} , is $\mathcal{L}(\mathcal{A}) = L?$ If not, return a counterexample, which is a word distinguishing \mathcal{A} and L (i.e., from the symmetric difference of $\mathcal{L}(\mathcal{A})$ and L).

The L^* algorithm constructs the minimal DFA \mathcal{A}_L recognizing L by refining approximations of the right congruence relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$ defined such that $u \sim_L v$ if and only if $\forall w (uw \in L \iff vw \in L)$. Due to the Myhill–Nerode theorem, the right congruence relation \sim_L defines \mathcal{A}_L .

The algorithm maintains two sets of words: the set of *selectors* S containing ϵ , and the set of *test words* C . The set C defines an approximation \sim_L^C of the right congruence relation defined such that $u \sim_L^C v$ if and only if $\forall w \in C (uw \in L \iff vw \in L)$. The set S contains exactly one selector of each equivalence class of \sim_L^C . More precisely, the algorithm maintains two properties: *separability*: different words from S are not \sim_L^C -equivalent, and *closedness*: for every $s \in S$ and $a \in \Sigma$, there is some word $s' \in S$ such that $sa \sim_L^C s'$. Note that if the pair S, C satisfies the above properties, we can construct a DFA corresponding to S, C , i.e., a DFA whose states are S and the transition relation is defined based on \sim_L^C , i.e., the successor of $s \in S$ over a is $s' \in S$ such that $sa \sim_L^C s'$.

The algorithm starts with $S = \{\epsilon\}$ and $C = \emptyset$ and it iterates the following steps in the loop until it terminates in Step 2.

- Step 1.** Starting with separable S, C , close S, C , i.e., construct $S' \supseteq S$ and $C' \supseteq C$ that are closed and remain separable.
- Step 2.** Construct the DFA A corresponding to the separable and closed pair S', C' . Check whether A recognizes L ; if yes, return A (and terminate).
- Step 3.** Otherwise, take the counterexample w and find $s \in S$, $a \in \Sigma$ and a suffix w' of w such that $S \cup \{sa\}, C \cup \{w'\}$ is separable, i.e., sa is a new selector for $\sim_L^{C \cup \{w'\}}$.

The L^* algorithm returns the minimal DFA recognizing \mathcal{A}_L and it works in polynomial time in $|\mathcal{A}_L|$ and the total length of returned counterexamples.

3.1 Difficulty of adapting L^* to visibly pushdown languages

We discuss the difficulty of learning visibly pushdown languages, which motivates our learning framework presented in the following sections.

The key property used by the L^* -algorithm is that for every regular language there is the canonical DFA recognizing it, which is also the minimal-size DFA for that language. Visibly pushdown languages do not have this property [3]. First, minimal-size DVPA are not unique; for a given visibly pushdown language \mathcal{L} there can be multiple non-isomorphic minimal-size DVPA recognizing \mathcal{L} [3, Proposition 1]. Second, a notion of canonical automaton for visibly pushdown languages was proposed in [3], but the canonical automaton can be exponentially larger than a minimal-size DVPA.

Furthermore, assuming that $\mathbf{P} \neq \mathbf{NP}$, there is no polynomial-time L^* -type learning algorithm for visibly pushdown languages. L^* -type algorithms learn a minimal-size automaton and run in polynomial time, and hence such an algorithm for DVPA can be used to minimize DVPA in polynomial time by running the learning algorithm as a black-box and computing answers to all its queries in polynomial time. The automaton returned by the algorithm would be a minimal-size DVPA language-equivalent to the input one \mathcal{T} . However, the minimization problem for DVPA (its decision version) is \mathbf{NP} -complete [20]. For these reasons, we settle for learning automata rather than languages.

One may suspect that there is an L^* -type learning algorithm, which only queries the stack content along the run on the input word. We show, however, that the \mathbf{NP} -hardness proof from [20] can be adjusted to DVPA with singleton stack alphabets (i.e., where $|\Gamma| = 1$):

► **Proposition 1.** *The following problem is NP-complete: given a DVPA \mathcal{T} with a singleton stack alphabet (a deterministic visibly counter automaton) and $N > 0$, decide whether there is a DVPA with a singleton stack alphabet language-equivalent to \mathcal{T} that has at most N states.*

For DVPA over a singleton stack alphabet, the stack height, which is the only information the stack provides, can be deduced from the input word. Therefore, allowing for stack observation alone does not lead to a polynomial-time learning algorithm.

4 Theoretical underpinnings

We introduce *control words*, which are words able to alter automata stack content. Then, we state and prove a Myhill-Nerode type theorem for DVPA using control words.

Control words. For a stack alphabet Γ , let $\underline{\Gamma}$ be defined as the infinite set $\{\underline{\perp} \cdot u \mid u \in \Gamma^*\}$, consisting of *control letters*. The intuitive meaning of the letter $\underline{\perp} \cdot u$ is “set the stack to $\perp \cdot u$ ”. Given an alphabet Σ , a *control word* is a word over $\Sigma \cup \underline{\Gamma}$. The size of a control word w , denoted by $\text{size}(w)$, is the sum of the number of letters from Σ and the sum of lengths of control letters from $\underline{\Gamma}$. We will sometimes use the term *standard word* to emphasize that the considered word does not contain control letters, i.e., it belongs to Σ^* .

Restricted control words. Let Σ be a finite alphabet and Γ be a finite stack alphabet. We define the set of *initializing words* $\text{Init}^{\Sigma, \Gamma} = \underline{\Gamma} \cdot (\Sigma \cup \underline{\Gamma})^*$, the set of control words, in which the first letter is a control letter. Furthermore, we define *single-reset words* $\text{Reset}_1^{\Sigma, \Gamma} = \Sigma^* \cdot \underline{\Gamma} \cdot \Sigma^*$ to be the set of control words with exactly one control letter, and the set of *single-reset initializing words* $\text{Init}_1^{\Sigma, \Gamma} = \underline{\Gamma} \cdot \Sigma^*$, in which the first letter is a control letter and the remaining letters are standard. We will omit the superscript Σ, Γ for readability.

Automata processing control words. Let \mathcal{T} be a DVPA over the alphabet Σ and the stack alphabet Γ . We extend the relation $\rightarrow_{\mathcal{T}}^w$ defined over words to control words as follows: let $v \in \perp \cdot \Gamma^*$, for every configuration (q, u) we have $(q, u) \xrightarrow{[\underline{v}]}_{\mathcal{T}} (q', u')$ if and only if $q = q'$ and $u' = v$, i.e., a control letter \underline{v} resets the stack content of the automaton to v , while it retains the state. We straightforwardly generalize acceptance to control words. A control word w is *accepted* by \mathcal{T} if there are $q_0 \in Q_0$, $q \in F$ and $u \in \perp \cdot \Gamma^*$ such that $(q_0, \perp) \xrightarrow{w}_{\mathcal{T}} (q, u)$. The language of control words accepted by \mathcal{T} is denoted by $\underline{L}(\mathcal{T})$.

We define a counterpart of the right congruence of a regular language.

► **Definition 2.** *Let $C \subseteq \text{Init}$ be a set of initializing words. We define the relation $\equiv_{\mathcal{T}}^C$ on control words as follows: we have $w_1 \equiv_{\mathcal{T}}^C w_2$ if and only if (1) for each $v \in C$ we have $w_1 v \in \underline{L}(\mathcal{T}) \iff w_2 v \in \underline{L}(\mathcal{T})$ and (2) for each $v \in C$ we have $SC_{\mathcal{T}}(w_1 v) = SC_{\mathcal{T}}(w_2 v)$.*

For every C , the relation $\equiv_{\mathcal{T}}^C$ is an equivalence relation. The definition of $\equiv_{\mathcal{T}}^C$ is monotonic w.r.t. C , i.e., for all sets $C_1 \subseteq C_2$, the relation $\equiv_{\mathcal{T}}^{C_2}$ is a refinement of $\equiv_{\mathcal{T}}^{C_1}$. In particular, the index of $\equiv_{\mathcal{T}}^{C_1}$ does not exceed the index of $\equiv_{\mathcal{T}}^{C_2}$.

We require $C \subseteq \text{Init}$, because otherwise the relation $\equiv_{\mathcal{T}}^C$ may have infinitely many equivalence classes, which correspond to configurations rather than states of \mathcal{T} .

For C being a strict subset of control words, $\equiv_{\mathcal{T}}^C$ need not be a right congruence; $w_1 \equiv_{\mathcal{T}}^C w_2$ does not imply $w_1 a \equiv_{\mathcal{T}}^C w_2 a$. Intuitively, words w_1, w_2 may lead to the same state in the automaton, but different configurations and after a single transition over letter a we obtain two configurations with different states. Thus, the right-congruence notion is incompatible with DVPA. We solve these problems by introducing another notion of congruence and restricting considered sets of control words.

Control right congruence. A relation \simeq is a *control right congruence* if and only if for all words $w_1, w_2 \in (\Sigma \cup \Gamma)^*$ and $\alpha \in \Gamma \cdot \Sigma$ we have $w_1 \simeq w_2$ implies $w_1\alpha \simeq w_2\alpha$.

► **Example 3.** Consider the *equal-state* relation $\simeq_{\mathcal{T}}$: for all $w_1, w_2 \in \Sigma^*$, we have $w_1 \simeq_{\mathcal{T}} w_2$ if and only if configurations reached by the DVPA \mathcal{T} over w_1 and w_2 respectively, have the same state, i.e., there are a state q and stack contents u_1, u_2 such that $(q_0, \perp) \xrightarrow{w_1}_{\mathcal{T}} (q, u_1)$ and $(q_0, \perp) \xrightarrow{w_2}_{\mathcal{T}} (q, u_2)$. The relation $\simeq_{\mathcal{T}}$ need not be a right congruence relation as w_1 and w_2 can reach the same state with different symbols on the tops of stacks, and hence the states reached over w_1a and w_2a may differ, i.e., $w_1a \not\simeq_{\mathcal{T}} w_2a$. However, the relation $\simeq_{\mathcal{T}}$ is a control right congruence as the first control letter of $\alpha = ca$ resets the stack to the same value in w_1c and w_2c , and hence these words lead to the same configuration and the next transition over a leads to the same configuration. In consequence, $w_1ca \simeq_{\mathcal{T}} w_2ca$.

We show that $\equiv_{\mathcal{T}}^{\text{Init}}$ is a control right congruence with the index bounded by the number of states of \mathcal{T} . Therefore, there is a finite C such that $\equiv_{\mathcal{T}}^C$ and $\equiv_{\mathcal{T}}^{\text{Init}}$ coincide. We also show that $\equiv_{\mathcal{T}}^{\text{Init}}$ allows us to construct the transition relation of a DVPA corresponding to $\equiv_{\mathcal{T}}^{\text{Init}}$.

► **Lemma 4.** Let \mathcal{T} be a DVPA. (1) The relation $\equiv_{\mathcal{T}}^{\text{Init}}$ is a control right congruence, (2) the index of $\equiv_{\mathcal{T}}^{\text{Init}}$ is bounded by $|\mathcal{T}|$, and (3) the relation $\equiv_{\mathcal{T}}^{\text{Init}}$ defines a unique DVPA \mathcal{A} , which is language-equivalent to \mathcal{T} .

► **Example 5.** We discuss why the second condition of the definition of $\equiv_{\mathcal{T}}^C$ is needed. Consider $\Sigma = \Sigma_c \cup \Sigma_l \cup \Sigma_r$ and a DVPA $\mathcal{A} = (\Sigma, \Gamma, Q, q_0, \delta, F)$. We assume that the sets Q , Σ , Γ and $\{\#, \$\}$ are pairwise disjoint. Let $\Sigma'_c = \Sigma_c \cup Q$, $\Sigma'_r = \Sigma_r \cup \{\$\}$, $\Sigma'_l = \Sigma_l \cup \{\#\}$ and $\Sigma' = \Sigma'_c \cup \Sigma'_r \cup \Sigma'_l$.

We consider a language \mathcal{L} over the alphabet Σ' that consists of the words of the form $w_q\#w\#w_{\$}$, where $w_q \in Q^*$ and $w_{\$} \in \{\$\}^*$. The idea is as follows: we want to accept words that make the stack empty (hence we use $\$$ at the end, to remove remaining stack elements) such that \mathcal{A} after reading w is in one of the states listed in w_q .

The language \mathcal{L} can be recognized by the automaton \mathcal{A}' that works in three stages. First, it pushes all the symbols from w_q on the stack. Then, it emulates \mathcal{A} on w . Finally, it pops stack symbols and checks whether the state of \mathcal{A} after reading w is on the stack.

We argue that the automaton \mathcal{A}' has the following property: its behaviour on the w_q part has no impact on its behaviour on the w part, but is crucial for the acceptance. Thus, the symbols that are put on the stack in the w_q part are significant only in processing the $w_{\$}$ part. This shows *non-locality*: the behaviour of \mathcal{A}' on $w_{\$}$ is defined by w_q , which are separated by an arbitrary long word w .

Such non-locality is what often makes the learning problems intractable. To avoid this, we allow the learning algorithm to read the stack content, and we use the definition of the relation $\equiv_{\mathcal{T}}^C$. There, we require that for each $a \in \Sigma$ and for each $c \in \{\perp\gamma \mid \gamma \in \Gamma \cup \{\epsilon\}\}$ we have $SC_{\mathcal{T}}(w_1ca) = SC_{\mathcal{T}}(w_2ca)$. In this way, the parts of the automaton reading w_q and $w_{\$}$ can be learned independently, as the results of reading letters on the stack are immediate.

If control letters are applied after each transition, DVPA behaves essentially as a DFA (see Section 5) and hence we attempt to minimize the use of control letters. In particular, we argue that considering $C = \text{Init}_1$ instead of Init is sufficient, i.e., single-reset initializing words are sufficient. We formalize it below.

We say that a DVPA \mathcal{A} is *constructed* from the equivalence relation $\equiv_{\mathcal{T}}^{\text{Init}_1}$ if its states are selectors of the equivalence classes of $\equiv_{\mathcal{T}}^{\text{Init}_1}$ and if \mathcal{A} has a transition $\delta(s, a, \alpha, s', \gamma)$, then either $s\perp\alpha a \equiv_{\mathcal{T}}^{\text{Init}_1} s'$, or $\alpha = \perp$ and a is a return letter, and $s\perp\alpha a$ and s' are equivalent over words $\perp w'$ only. The stack content after reading $s\perp\alpha a$ is updated according to γ . Such an automaton is not necessarily unique as $\equiv_{\mathcal{T}}^{\text{Init}_1}$ is not a control right congruence in general. However, we show that any such DVPA is admissible:

■ **Table 1** Queries for learning with control words.

Query	membership query	stack-content query	equivalence query
Input	a control word w	a control word w	a DVPA \mathcal{A}
Output	whether \mathcal{T} accepts w	$SC_{\mathcal{T}}(w)$	YES if \mathcal{A} and \mathcal{T} are language-equivalent, otherwise a control word w distinguishing \mathcal{A} and \mathcal{T}

► **Lemma 6.** *Any DVPA constructed from $\equiv_{\mathcal{T}}^{\text{Init}}$ restricted to standard words is language-equivalent with \mathcal{T} .*

All equivalence classes of $\equiv_{\mathcal{T}}^{\text{Init}}$, which do not contain standard words are irrelevant for language-equivalence as they correspond to unreachable states. Thus, Lemma 6 implies that in order to learn a DVPA language-equivalent to \mathcal{T} , we can construct $\equiv_{\mathcal{T}}^{\text{Init}_1}$ restricted to standard words. This observation is the backbone of the learning algorithm presented in Section 6.

5 Learning with resets

We present the first active learning algorithm for DVPA based on control words and discuss its applications. We consider a learning framework with three types of queries: (modified) membership and equivalence queries and a new type called *stack-content queries*, which return the content of the stack upon reading a given word. The queries involve control letters, as we showed that using only standard words is insufficient (Section 3.1). Here, control words can have any number of control letters. In such a case, learning DVPA boils down to the standard learning of DFA. Indeed, with arbitrarily many control letters every path in a DVPA can be realized. Furthermore, we can address each transition and learn how the stack changes with stack-content queries. The solution for learning with unlimited control letters is a simple adaptation of the original L^* algorithm.

We consider an oracle, called the *teacher*, which has access to the target DVPA \mathcal{T} and answers the following three types of queries detailed in Table 1. Note that the counterexample can contain control letters as well.

We reduce active learning of DVPA to active learning of regular languages. Consider $\Sigma^{\text{lab}} = (\Sigma_c \times \Gamma) \cup \Sigma_l \cup (\Sigma_r \times (\Gamma \cup \{\perp\}))$, which characterizes labels of transitions of \mathcal{T} : (1) call transitions are characterized by a call letter $a \in \Sigma_c$ and a stack symbol $B \in \Gamma$ pushed to the stack, (2) local transitions are characterized by a local letter alone $a \in \Sigma_l$, and (3) return transitions are characterized by a call letter $a \in \Sigma_r$ and the top of the stack symbol $\gamma \in \Gamma \cup \{\perp\}$.

Let $\mathcal{L}^{\text{lab}}(\mathcal{T})$ be the language of words w over Σ^{lab} , which correspond to paths in \mathcal{T} terminating in an accepting state. Note that paths need not obey the stack; for instance pushing a symbol A can be followed by pulling a symbol B , and hence not all paths correspond to runs of \mathcal{T} . However, the language $\mathcal{L}^{\text{lab}}(\mathcal{T})$ is regular and hence can be learned with the L^* -algorithm for DFA. The detailed constructions have been relegated to the appendix.

► **Theorem 7.** *Assume that the teacher returns minimal-size counterexamples. Then, active learning of DVPA with a teacher answering queries detailed in Table 1 can be done in time polynomial in the size of the target automaton.*

■ **Table 2** Queries for learning with single resets.

Query	membership query	stack-content query	equivalence query
Input	a single-reset word w	a single-reset word w	a DVPA \mathcal{A}
Output	whether \mathcal{T} accepts w	$SC_{\mathcal{T}}(w)$	YES if \mathcal{A} and \mathcal{T} are language-equivalent, otherwise, a (standard) word w distinguishing \mathcal{A} and \mathcal{T}

5.1 Applications

The active learning of DVPA can be used for reducing the number of states of DVPA given implicitly. For instance, consider a model M consisting of k components $\mathcal{A}_1, \dots, \mathcal{A}_k$ being DVPA. Models that are products of smaller components occur frequently in verification tasks (e.g., to express union or intersection of languages) and there is a large body of work on efficient model checking on such models [38, 21]. Even though the size of M is the product of sizes of $\mathcal{A}_1, \dots, \mathcal{A}_k$, the number of reachable states may be considerably smaller, so it can be explicitly represented.

We can use the above active learning algorithm for DVPA to construct a reduced model of $\mathcal{A}_1 \times \dots \times \mathcal{A}_k$ without the need of constructing the whole product. Now having DVPA $\mathcal{A}_1, \dots, \mathcal{A}_k$, we can compute the values of membership and stack-content queries. For the equivalence queries with a given \mathcal{A} , we can check whether for all i we have $\mathcal{L}(\mathcal{A}_i) \supseteq \mathcal{L}(\mathcal{A})$, which implies that the language of the product of automata contains the language of \mathcal{A} . The opposite inclusion can be approximated with queries over random words, guided by various heuristics as is often the case with equivalence queries [12, 18]. It is difficult to answer equivalence queries for the product of automata as it subsumes the emptiness problem for intersections of regular languages, which is **PSpace**-complete.

Note that the constructed automaton can have states that are not reachable by any run (see Example 8). This is an inherent problem of considered words with multiple control letters; it diminishes the influence of the stack on reachability. To solve this problem, we focus on the single-reset approach presented in the following Section 6.

► **Example 8** (Automaton with redundant states). Consider a DVPA \mathcal{T} and its state q , which is reachable only with call transitions pushing A on the stack. Suppose that q has an outgoing return transition over $A' \neq A$, which leads to a component \mathcal{B} of \mathcal{T} not reachable with any other transition. Then, the algorithm from Theorem 7 explores the component \mathcal{B} of \mathcal{T} as it is reachable over words with multiple control letters and hence the returned DVPA contains redundant states.

6 Learning with a single reset

Here we study a learning framework with three types of queries as in Section 5, but restricted to words with one control letter. Those queries are detailed in Table 2. For a finite $C \subseteq \text{Init}_1$, these queries are sufficient to decide whether $w_1 \equiv_C^{\mathcal{T}} w_2$ holds over $w_1, w_2 \in \Sigma^*$; the first condition from the definition of $\equiv_C^{\mathcal{T}}$ (Definition 2) can be implemented with membership queries, and the second one with stack-content queries.

We present an active learning algorithm for DVPA, which works in polynomial time in size of the target automaton and the size of counterexamples provided by the teacher. The algorithm is a modification of the L^* algorithm, so we discuss only the differences with the vanilla L^* algorithm. The main difference is that while L^* constructs the right congruence relation of the target language \mathcal{L} , our algorithm constructs $\equiv_{\mathcal{T}}^{\text{Init}_1}$ over standard words.

74:10 Learning Deterministic Visibly Pushdown Automata Under Accessible Stack

Our algorithm, similarly to L^* , maintains two sets of words: a set of *selectors* S and a set of *test words* C such that S are selectors of equivalence classes of $\equiv_{\mathcal{T}}^C$. Selectors in $S \subset \Sigma^*$ are standard words over Σ and test words in $C \subset \text{Init}_1$ are single-reset initializing words. We start with $S = \{\epsilon\}$ and $C = \{\perp\epsilon\}$.

We adjust the terminology to the current setting as follows. *Separability* means that different words from S are not $\equiv_{\mathcal{T}}^C$ -equivalent. *Closedness* means that for every $s \in S$, $a \in \Sigma$ and $B \in \Gamma \cup \{\epsilon\}$, there is some word $s' \in S$ such that $s \cdot \perp \cdot B \cdot a \equiv_{\mathcal{T}}^C s'$. The algorithm maintains separability and some restricted form of closedness. The latter is due to the fact that some transitions may be unreachable in a DVPA. For example, if the only way to reach some state is to push B on the stack, then all the transitions assuming that the top of the stack is not B are unreachable and thus irrelevant. For that reason, two steps in the L^* algorithm: computing closure of S, C and generating the corresponding automaton, are interleaved and encapsulated in a single procedure *generate automaton*.

Generating automata. Given S, C and access to the teacher for \mathcal{T} , we construct a (partial) DVPA whose set of states is $S \subset \Sigma^*$, possibly extended with some additional states as discussed below. The initial state is $\epsilon \in S$. The accepting states are those states $w \in S$ such that the membership query on w returns true, i.e., w is accepted by \mathcal{T} . It remains to compute the transition function δ .

Consider a state $s \in S$, a letter $a \in \Sigma$ and $\gamma \in \Gamma \cup \{\perp\}$. To find $\delta(s, a, \gamma)$, we feed s to the target automaton \mathcal{T} , implant γ on the top of its stack and take the transition over a . Next, we identify the state from S equivalent to the current state of \mathcal{T} or create a new one, if there is no fitting one in S . The details are in the appendix.

Equivalence and processing a counterexample. Given a partial DVPA \mathcal{A} , we complete it to a DVPA \mathcal{A}^c by setting all undefined transitions so that their destination is ϵ and putting some designated $B_0 \in \Gamma$ on the stack over call transitions. Next, we test for equivalence and if it happens to be language-equivalent to \mathcal{T} we return \mathcal{A}^c . Otherwise, we get a counterexample that distinguishes \mathcal{A}^c and \mathcal{T} . We show how to use it to ensure progress of the algorithm. There are two possible cases of counterexamples: either the stack content diverges at some point, or the values of some suffixes mismatch. In both cases, we can modify S and C to ensure the progress. The details are in the appendix.

Irredundancy. Observe that all states in the returned automaton are reachable by some run (in contrast to the automaton from Example 8). To see this, note that we start with the initial state ϵ . Next, new successors are added either in the automata construction or in processing of a counterexample. In the former case, the new state is a successor of some reachable state over a consistent transition. That is, it is a call or local transition, which can be always executed, or it is a return transition and the stack content over w_s reaching $s \in S$ in the constructed automaton is consistent with the expected stack content in the transition. In the latter case, the successor va is added because the values of suffixes mismatch. There va is not $\equiv_{\mathcal{T}}^{C'}$ -equivalent to any state from S and hence upon reading va the automaton reaches the new state va . Therefore, the algorithm with single-reset words avoids computation of redundant states in contrast to the algorithm from Section 5.

Complexity. The algorithm always terminates and works in polynomial time in the size of the target automaton and the size of teachers' responses. The details are in the appendix.

► **Theorem 9.** *Active learning of DVPA with teacher answering queries detailed in Table 2 can be done in time polynomial in the sizes of the target automaton and teachers' responses.*

For L^* , if we assume that the teacher always returns a minimal-length counterexample, the algorithm works in polynomial time in the size of the target automaton alone. For DVPA, the shortest counterexample for language-equivalence can be of exponential-size in $|\mathcal{T}|$. In the following sections we propose two approaches to solve this problem: the first one is based on relaxed equivalence and the second one on employing grammar-based compression.

7 Active learning with bounded counterexamples

To avoid processing exponentially-long counterexamples, we can impose a bound on the length of admissible counterexamples. In some scenarios we may be interested in trading accuracy for performance and focus on words of length bounded by some fixed value t . A similar approach led to the development of *bounded model checking* methods, which are highly efficient [14, 25]. Furthermore, in applications of active learning of automata, equivalence queries, which are often costly to implement, are only approximated by equivalence over bounded-length words [12, 18].

Two DVPA $\mathcal{A}_1, \mathcal{A}_2$ are t -equivalent, where $t \in \mathbb{N}$, if and only if their languages agree on all words of the length bounded by t . We adjust the framework in a natural way: we replace equivalence queries with t -equivalence queries, which say whether a given DVPA \mathcal{A} is t -equivalent to \mathcal{T} . If it is not, then we require a counterexample of length bounded by t .

Under the modified framework, we can run the algorithm described in Section 6. Note that each counterexample there is bounded by t and the algorithm terminates when \mathcal{A} is t -equivalent to \mathcal{T} . Furthermore, each counterexample can be used to increase the size of S , which is bounded by the number of states of \mathcal{T} . In consequence, we have:

► **Theorem 10.** *Active learning of DVPA with teacher answering membership and stack-content queries defined in Table 2 and t -equivalence queries can be done in time polynomial in the size of the target automaton and t .*

Finally, observe that the teacher can answer t -equivalence queries in polynomial time in $|\mathcal{A}| + |\mathcal{T}| + t$. It suffices to construct a DFA \mathcal{A}_t accepting all words of the length at most t and compute the emptiness of the intersection of the languages \mathcal{A} , \mathcal{A}_t and \mathcal{T}^C , where \mathcal{T}^C is the complemented \mathcal{T} .

8 Active learning with compressed representation

Here we propose to solve the problem of exponential counterexamples using compression by means of visibly context-free grammars to represent counterexamples as well as selectors and test words. We show that in this setting the teacher can always provide a counterexample of polynomial size in the size of the target automaton and a candidate automaton sent in the equivalence query. If the teacher does so, the algorithm we propose works in polynomial time in size of the target automaton.

Context-free grammars (which are equivalent to pushdown automata) can generate finite languages with exponential-length words. This observation led to the development of grammar-based compression, where a word w is represented through a context-free grammar (called Straight-Line Programs or SLPs) generating exactly one word w . Equivalence of SLPs with other compression methods and algorithmic problems are discussed in [30].

■ **Table 3** Queries for learning with vSLP.

Query	membership query	stack-content query	equivalence query
Input	a vSLP representing a single-reset word w	a vSLP representing a single-reset word w	a DVPA \mathcal{A}
Output	whether \mathcal{T} accepts w	$SC_{\mathcal{T}}(w)$	YES if \mathcal{A} and \mathcal{T} are language-equivalent, otherwise, a vSLP representing a standard word w distinguishing \mathcal{A} and \mathcal{T}

We use polynomial-size SLPs to represent counterexamples as well as selectors. Note that processing SLPs can be expensive in general. There is a visibly pushdown language \mathcal{L}_h such that the membership problem for this language over words given by SLPs is **PSpace**-complete [29]. However, if we restrict SLPs to those that are represented by *visibly pushdown grammars* G (over the same alphabet partition as the visibly pushdown language \mathcal{L}_h) defined below, then the membership problem is equivalent to the emptiness problem for the intersection of \mathcal{L}_h and $\mathcal{L}(G)$, and hence it can be decided in polynomial time.

Visibly pushdown grammars. A *visibly pushdown grammar* (VPG) is a syntactically restricted pushdown grammar (see [4] for details). Consider a partition of Σ into $(\Sigma_c, \Sigma_l, \Sigma_r)$. A VPG is a triple $G = (V, S, P)$, where $V = V_0 \cup V_1$ is a set of *non-terminal symbols* partitioned into V_0 and V_1 , $I \in V$ is the *start symbol*, and P is the set of *productions* of the following form:

- $X \rightarrow \epsilon$, where $X \in V$
- $X \rightarrow aY$, where $X, Y \in V$, and if $X \in V_0$, then $a \in \Sigma_l$ and $Y \in V_0$
- $X \rightarrow aYbZ$, where $X \in V$, $Y \in V_0$, $a \in \Sigma_c$, $b \in \Sigma_r$, and if $X \in V_0$, then $Z \in V_0$.
- $X \rightarrow YZ$, where $X, Y \in V$, and if $X \in V_0$, then $Y, Z \in V_0$.

The last rule is not present in the original definition, but it can be safely added as VPL are closed under concatenation.

VPA and VPG are language-wise polynomially equivalent [4] (i.e., there is a polynomial-time procedure that, given a PDA, outputs a CFG of the same language and vice versa).

For a VPG G , we define derivation \rightarrow_G as a relation on $(\Sigma \cup V)^* \times (\Sigma \cup V)^*$ as follows: $w \rightarrow_G w'$ if and only if $w = w_1Xw_2$, with $X \in V$, and $w' = w_1uw_2$ for some $u \in (\Sigma \cup V)^*$ such that $X \rightarrow u$ is a production from G . We define \rightarrow_G^* as the transitive closure of \rightarrow_G . The *language generated by G* , denoted by $\mathcal{L}(G) = \{w \in \Sigma^* \mid s \rightarrow_G^* w\}$ is the set of words that can be derived from I . VPA and VPG are language-wise polynomially equivalent [4] (i.e., there is a polynomial-time procedure that, given a PDA, outputs a CFG of the same language and vice versa). The size of a VPG $G = (V, I, P)$, denoted by $|G|$, is $|V| + |P|$.

A *visibly Straight Line Program* (vSLP) is a visibly pushdown grammar G , which generates a single word denoted by w_G .

Now, selectors in S and test words in C will be represented by vSLPs. We introduce the framework and assume that words are represented as vSLPs, as detailed in Table 3. Importantly, the teacher can always return a polynomial size vSLP representing a counterexample.

► **Lemma 11.** *Given two DVPA $\mathcal{A}_1, \mathcal{A}_2$ such that $\mathcal{L}(\mathcal{A}_1) \neq \mathcal{L}(\mathcal{A}_2)$, there exists a polynomial size vSLP G such that w_G belongs to the symmetric difference of $\mathcal{L}(\mathcal{A}_1)$ and $\mathcal{L}(\mathcal{A}_2)$. Furthermore, this vSLP can be computed in polynomial time in $|\mathcal{A}_1| + |\mathcal{A}_2|$.*

The words represented by vSLPs can be efficiently processed. The procedure generating a DVPA from S, C requires efficient computation of concatenation of vSLPs and compressed membership and stack-content queries. Observe that for a polynomial-size vSLP G , the length of $SC_{\mathcal{T}}(w_G)$ is polynomial in $|\mathcal{T}|$ even though $|w_G|$ can be exponential and hence the answers to these queries have polynomial size. Processing counterexamples is more difficult as it involves finding the prefixes of the counterexample w_G with certain properties. Note that these prefixes can be identified with binary search and hence only polynomially many candidates need to be considered. It suffices to compute, for a given vSLP G , the prefix of the first i letters of w_G , which is possible in polynomial time in $|G|$. In consequence, we have the following:

► **Theorem 12.**

- (i) *The teacher can answer membership and equivalence queries with vSLP input in polynomial time in the input size and $|\mathcal{T}|$.*
- (ii) *The teacher can answer equivalence queries with polynomial size vSLP encoding a counterexample in polynomial time (in the input size and $|\mathcal{T}|$).*
- (iii) *For the teacher as in (i) and (ii), active learning of DVPA with teacher answering queries detailed in Table 3 can be done in time polynomial in the size of the target automaton alone.*

9 Future work

We have presented the first work on polynomial-time learning DVPA. The queries we use depend not only on the language of the automaton, but its structure as well. It allows us to bypass problems arising from the difficulty of minimizing DVPA.

There is an open field of possible future directions, which we discuss below.

Dropping the stack content queries. Control words are crucial in our solution, i.e, the ability to alter the stack. We also assume that the stack can be observed, using the stack content queries. While this is a reasonable assumption, we have not shown whether it is necessary. In particular, we may explore the scenario from Section 5.1, in which we attempt to minimize both the state space and the stack alphabet and hence we consider the stack alphabet as unknown. We leave this as an open question.

Queries with registers. Another interesting scenario, but closely related to the previous one, is where we cannot read and write the stack directly, but we can save its value to a register and then use it. This corresponds to the case when we have access to the device memory, but the stack is represented in a way that we cannot manipulate it (e.g. it is encrypted). Note that in Section 6, we either use the stack contents that have been produced by some runs with at most one symbol pushed on the top. Is active learning in polynomial time possible if the stack content can be only copied between runs?

Learning all rt-DPDA. In the algorithm presented in Section 6, we rely on the fact that call and local letters do not depend on the stack content. We can possibly improve the algorithm to omit that assumption at the cost of more complex algorithm. We leave this problem for further investigation.

Learning all DPDA. Another challenging area is learning DPDA. One of the key challenges is to deal with ϵ -transitions. The problem there is that for some DPDA there can be states that are reachable with only ϵ -transitions, i.e., states without selectors. A solution to this problem would require a new insight into the representation of states.

References

- 1 Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. Model checking procedural programs. In *Handbook of Model Checking*, pages 541–572. Springer, 2018.
- 2 Rajeev Alur, Kousha Etessami, and Parthasarathy Madhusudan. A temporal logic of nested calls and returns. In *TACAS*, pages 467–481. Springer, 2004.
- 3 Rajeev Alur, Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Congruences for visibly pushdown languages. In *ICALP 2005*, pages 1102–1114. Springer, 2005.
- 4 Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3):16:1–16:43, 2009. doi:10.1145/1516512.1516518.
- 5 Dana Angluin. Learning k-bounded context-free grammars. *Research report. Yale University. Dept. of Computer Science*, 557, 1987.
- 6 Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.
- 7 Dana Angluin and Michael Kharitonov. When won’t membership queries help? *J. Comput. Syst. Sci.*, 50(2):336–355, 1995.
- 8 Borja Balle and Mehryar Mohri. Learning weighted automata. In *CAI 2015*, pages 1–21, 2015.
- 9 Borja Balle and Mehryar Mohri. On the Rademacher complexity of weighted automata. In *ALT 2015*, pages 179–193, 2015.
- 10 Borja Balle and Mehryar Mohri. Generalization bounds for learning weighted automata. *Theor. Comput. Sci.*, 716:89–106, 2018.
- 11 Borja Balle, Prakash Panangaden, and Doina Precup. A canonical form for weighted automata and applications to approximate minimization. In *LICS 2015*, pages 701–712, 2015.
- 12 Benoît Barbot, Benedikt Bollig, Alain Finkel, Serge Haddad, Igor Khmel’nitsky, Martin Leucker, Daniel Neider, Rajarshi Roy, and Lina Ye. Extracting context-free grammars from recurrent neural networks using tree-automata learning and a* search. In Jane Chandlee, Rémi Eyraud, Jeff Heinz, Adam Jardine, and Menno Zaanen, editors, *Proceedings of the 15th International Conference on Grammatical Inference, 23-27 August 2021, Virtual Event*, volume 153 of *Proceedings of Machine Learning Research*, pages 113–129. PMLR, 2021. URL: <https://proceedings.mlr.press/v153/barbot21a.html>.
- 13 Jean Berstel and Luc Boasson. Towards an algebraic theory of context-free languages. *Fundamenta Informaticae*, 25(3, 4):217–239, 1996.
- 14 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Adv. Comput.*, 58:117–148, 2003. doi:10.1016/S0065-2458(03)58003-2.
- 15 Laura Bozzelli, Aniello Murano, and Adriano Peron. Context-free timed formalisms: Robust automata and linear temporal logics. *Information and Computation*, page 104673, 2020.
- 16 Swarat Chaudhuri and Rajeev Alur. Instrumenting c programs with nested word monitors. In *International SPIN Workshop on Model Checking of Software*, pages 279–283. Springer, 2007.
- 17 Patrick Chervet and Igor Walukiewicz. Minimizing variants of visibly pushdown automata. In *MFCSS 2007*, volume 4708 of *Lecture Notes in Computer Science*, pages 135–146. Springer, 2007. doi:10.1007/978-3-540-74456-6_14.
- 18 Hana Chockler, Pascal Kesseli, Daniel Kroening, and Ofer Strichman. Learning the language of software errors. *J. Artif. Intell. Res.*, 67:881–903, 2020. doi:10.1613/jair.1.11798.
- 19 Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

- 20 Olivier Gauwin, Anca Muscholl, and Michael Raskin. Minimization of visibly pushdown automata is np-complete. *Log. Methods Comput. Sci.*, 16(1), 2020.
- 21 Dimitra Giannakopoulou, Kedar S. Namjoshi, and Corina S. Pasareanu. Compositional reasoning. In *Handbook of Model Checking*, pages 345–383. Springer, 2018. doi:10.1007/978-3-319-10575-8_12.
- 22 David Hopkins, Andrzej S Murawski, and C-H Luke Ong. A fragment of ML decidable by visibly pushdown automata. In *ICALP*, pages 149–161. Springer, 2011.
- 23 Malte Isberner. *Foundations of Active Automata Learning: An Algorithmic Perspective*. PhD thesis, Technischen Universität Dortmund, 2015. URL: <https://eldorado.tu-dortmund.de/bitstream/2003/34282/1/Dissertation.pdf>.
- 24 Petr Jancar. Equivalence of pushdown automata via first-order grammars. *J. Comput. Syst. Sci.*, 115:86–112, 2021. doi:10.1016/j.jcss.2020.07.004.
- 25 Kareem Khazem and Michael Tautschnig. CBMC path: A symbolic execution retrofit of the C bounded model checker. In *TACAS*, volume 11429 of *LNCS*, pages 199–203, 2019. doi:10.1007/978-3-030-17502-3_13.
- 26 Bruce Knobe and Kathleen Knobe. A method for inferring context-free grammars. *Information and Control*, 31(2):129–146, 1976.
- 27 Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Minimization, learning, and conformance testing of boolean programs. In *CONCUR 2006*, pages 203–217. Springer Berlin Heidelberg, 2006.
- 28 Christof Löding, Parthasarathy Madhusudan, and Olivier Serre. Visibly pushdown games. In *FSTTCS*, pages 408–420. Springer, 2004.
- 29 Markus Lohrey. Leaf languages and string compression. *Inf. Comput.*, 209(6):951–965, 2011. doi:10.1016/j.ic.2011.01.009.
- 30 Markus Lohrey. Algorithmics on SLP-compressed strings: A survey. *Groups-Complexity-Cryptography*, 4(2):241–299, 2012.
- 31 Wim Martens, Frank Neven, Thomas Schwentick, and Geert Jan Bex. Expressiveness and complexity of XML schema. *ACM Transactions on Database Systems (TODS)*, 31(3):770–813, 2006.
- 32 Ines Marusic and James Worrell. Complexity of equivalence and learning for multiplicity tree automata. *Journal of Machine Learning Research*, 16:2465–2500, 2015.
- 33 Kurt Mehlhorn. Pebbling mountain ranges and its application of DCFL-recognition. In *ICALP 1980*, volume 85 of *LNCS*, pages 422–435. Springer, 1980. doi:10.1007/3-540-10003-2_89.
- 34 Jakub Michaliszyn and Jan Otop. Approximate learning of limit-average automata. In *CONCUR 2019*, pages 17:1–17:16, 2019. doi:10.4230/LIPIcs.CONCUR.2019.17.
- 35 Jakub Michaliszyn and Jan Otop. Learning deterministic automata on infinite words. In *ECAI 2020 – 24th European Conference on Artificial Intelligence*, volume 325, pages 2370–2377. IOS Press, 2020. doi:10.3233/FAIA200367.
- 36 Jakub Michaliszyn and Jan Otop. Non-deterministic weighted automata evaluated over Markov chains. *J. Comput. Syst. Sci.*, 108:118–136, 2020.
- 37 Marvin C Paull and Stephen H Unger. Structural equivalence of context-free grammars. *J. Comput. Syst. Sci.*, 2(4):427–463, 1968.
- 38 Doron Peled. Partial-order reduction. In *Handbook of Model Checking*, pages 173–190. Springer, 2018. doi:10.1007/978-3-319-10575-8_6.
- 39 Daniel J. Rosenkrantz and Harry B. Hunt III. The complexity of structural containment and equivalence. In Jeffrey D. Ullman, editor, *Theoretical Studies in Computer Science, to Seymour Ginsburg on the occasion of his 26th birthday*, pages 101–132. Academic Press, 1992. doi:10.1016/b978-0-12-708240-0.50009-5.
- 40 Yasubumi Sakakibara. Efficient learning of context-free grammars from positive structural examples. *Information and Computation*, 97(1):23–60, 1992.
- 41 Géraud Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *ICALP 1997*, pages 671–681, 1997. doi:10.1007/3-540-63165-8_221.

74:16 Learning Deterministic Visibly Pushdown Automata Under Accessible Stack

- 42 Luzi Sennhauser and Robert C. Berwick. Evaluating the ability of LSTMs to learn context-free grammars. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2018*, pages 115–124, 2018. doi:10.18653/v1/w18-5414.
- 43 Ehud Y Shapiro. Algorithmic program diagnosis. In *POPL*, pages 299–308, 1982.