Adaptive Synchronisation of Pushdown Automata

A. R. Balasubramanian
Technische Universität München, Germany

K. S. Thejaswini
Department of Computer Science, University of Warwick, Coventry, UK

Abstract

We introduce the notion of adaptive synchronisation for pushdown automata, in which there is an external observer who has no knowledge about the current state of the pushdown automaton, but can observe the contents of the stack. The observer would then like to decide if it is possible to bring the automaton from any state into some predetermined state by giving inputs to it in an *adaptive* manner, i.e., the next input letter to be given can depend on how the contents of the stack changed after the current input letter. We show that for non-deterministic pushdown automata, this problem is 2-EXPTIME-complete and for deterministic pushdown automata, we show EXPTIME-completeness.

To prove the lower bounds, we first introduce (different variants of) subset-synchronisation and show that these problems are polynomial-time equivalent with the adaptive synchronisation problem. We then prove hardness results for the subset-synchronisation problems. For proving the upper bounds, we consider the problem of deciding if a given alternating pushdown system has an accepting run with at most k leaves and we provide an $n^{O(k^2)}$ time algorithm for this problem.

2012 ACM Subject Classification Theory of computation → Grammars and context-free languages; Theory of computation → Problems, reductions and completeness

Keywords and phrases Adaptive synchronisation, Pushdown automata, Alternating pushdown systems

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.17

Related Version *Full Version*: https://arxiv.org/abs/2102.06897

Funding *A. R. Balasubramanian*: Supported by funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 787367 (PaVeS).
K. S. Thejaswini: Supported by the EPSRC grant EP/P020992/1 (Solving Parity Games in Theory and Practice).

Acknowledgements We would like to thank Dmitry Chistikov for referring us to previous works on this topic.

1 Introduction

The notion of a synchronizing word for finite-state machines is a classical concept in computer science which consists of deciding, given a finite-state machine, whether there is a word which brings all of its states to a single state. Intuitively, assuming that we initially do not know which state the machine is in, such a word *synchronises* it to a single state and assists in regaining control over the machine.

This idea has been studied for many types of finite-state machines [24, 22, 2, 9] with applications in biocomputing [3], planning and robotics [10, 19] and testing of reactive systems [18, 14]. In recent years, the notion of a synchronizing word has been extended to various *infinite-state systems* such as timed automata [8], register automata [20], nested word automata [7], pushdown and visibly pushdown automata [11, 12]. In particular, for the pushdown case, Fernau, Wolf and Yamakami [12] have shown that this problem is undecidable even for deterministic pushdown automata.

When the finite-state machine can produce outputs, the notion of synchronisation has been further refined to give rise to synchronisation under *partial observation* or *adaptive synchronisation* (See Chapter 1 of [5] and [17]). In this setting, there is an external observer who does not know the current state of the machine, however she can give inputs to the machine and observe the outputs given by the machine. Depending on the outputs of the machine, she can *adaptively* decide which input letter to give next. In this manner, the observer would like to bring the machine into some predetermined state. Larsen, Laursen and Srba [17] describe an example of adaptive synchronisation pertaining to the orientation of a simplified model of satellites, in which they observe that adaptively choosing the input letter is sometimes necessary in order to achieve synchronisation. In this paper, we extend this notion of adaptive synchronisation to pushdown automata (PDA). In our model, the observer does not know which state the PDA is currently in, but can observe the contents of the stack. She would then like to decide if it is possible to synchronise the PDA into some state by giving inputs to the PDA adaptively, i.e., depending on how the stack changes after each input. To the best of our knowledge, the notion of adaptive synchronisation has not been considered before for any class of infinite-state systems.

This question is a natural extension of the notion of adaptive synchronisation from finite-state machines to pushdown automata. Further, it is mentioned in the works of Lakhotia, Uday Kumar and Venable as well as Song and Touili [21, 16] that several antivirus systems determine whether a program is malicious by observing the calls that the program makes to the operating system. With this in mind, Song and Touili use pushdown automata [21] as abstractions of programs where a stack stores the calls made by the program and use this abstraction to detect viruses. Hence, we believe that our setting of being able to observe the changes happening to the stack can be practically motivated.

Our main results regarding adaptive synchronisation are as follows: We show that for non-deterministic pushdown automata, the problem is 2-EXPTIME-complete. However, by restricting our input to deterministic pushdown automata, we show that we can get EXPTIME-completeness, thereby obtaining an exponential reduction in complexity.

We also consider a natural variant of this problem, called *subset adaptive synchronisation*, which is similar to adaptive synchronisation, except the observer has more knowledge about which state the automaton is initially in. We obtain a surprising result that shows that this variant is polynomial-time equivalent to adaptive synchronisation, unlike in the case of finite-state machines. Furthermore, for the deterministic case of this variant, we obtain an algorithm that runs in time $O\left(n^{ck^3}\right)$ where n is the size of the input and k is the size of the subset of states that the observer believes the automaton is initially in. This gives a polynomial time algorithm if k is fixed and a quasi-polynomial time algorithm if $k = O(\log n)$.

Used as a subroutine in the above decision procedure, is an $O\left(n^{ck^2}\right)$ time algorithm to the following question, which we call the *sparse-emptiness problem*: Given an alternating pushdown system and a number k, decide whether there is an accepting run of the system with at most k leaves. Intuitively, such a run means that the system has an accepting run in which it uses only "limited universal branching". We note that such a notion of alternation with "limited universal branching" has recently been studied by Keeler and Salomaa for alternating finite-state automata [15]. Our problem can be considered as a generalisation of one of their problems (Corollary 2 of [15]) to pushdown systems. We think that this problem and its associated algorithm might be of independent interest.

Roadmap. In Section 2, we introduce notations. In Section 3, we discuss different variations of the problem. In Sections 4 and 5 we prove lower and upper bounds respectively. Proofs of some of our technical results can be found in the long version of this extended abstract available on arXiv [1].

2 Preliminaries

Given a finite set X, we let X^* denote the set of all words over the alphabet X. As usual, the concatenation of two words $x, y \in X^*$ is denoted by xy.

2.1 Pushdown Automata

We recall the well-known notion of a pushdown automaton. A pushdown automaton (PDA) is a 4-tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ where Q is a finite set of *states*, Σ is the *input alphabet*, Γ is the *stack alphabet* and $\delta \subseteq (Q \times \Sigma \times \Gamma) \times (Q \times \Gamma^*)$ is the *transition relation*. Alternatively, sometimes we will describe the transition relation δ as a function $Q \times \Sigma \times \Gamma \mapsto 2^{Q \times \Gamma^*}$. We will always use small letters $a, b, c, \ldots$ to denote elements of Σ, capital letters $A, B, C, \ldots$ to denote elements of Γ and Greek letters $\gamma, \eta, \omega, \ldots$ to denote elements of Γ^*.

If $(p, a, A, q, \gamma) \in \delta$ then we sometimes denote it by $(p, A) \overset{a}{\hookrightarrow} (q, \gamma)$. We say A is the *top* of the stack that is *popped* and γ is the string that is *pushed* onto the stack. A *configuration* of the automaton is a tuple (q, γ) where $q \in Q$ and $\gamma \in \Gamma^*$. Given two configurations $(q, A\gamma)$ and $(q', \gamma'\gamma)$ of $\mathcal{P}$ with $A \in \Gamma$, we say that $(q, A\gamma) \xrightarrow{a} (q', \gamma'\gamma)$ iff $(q, A) \overset{a}{\hookrightarrow} (q', \gamma')$.

As is usual, we assume that there exists a special *bottom-of-the-stack* symbol $\perp \in \Gamma$, such that whenever some transition pops $\perp$, it pushes it back in the bottom-most position. A PDA is said to be deterministic if for every $q \in Q$, $a \in \Sigma$ and $A \in \Gamma$, $\delta(q, a, A)$ has exactly one element. If a PDA is deterministic, we further abuse notation and denote $\delta(q, a, A)$ as a single element and not as a set.

2.2 Adaptive Synchronisation

We first expand upon the intuition given in the introduction for adaptive synchronisation with the help of a running example. Consider the pushdown automaton as given in Figure 1 where we do not know which state the automaton is in currently, but we do know that the stack content is $\perp$. To synchronise the automaton to the state 4 when the stack is visible, the observer has a strategy as depicted in Figure 2. The labelling of the nodes of the tree intuitively denotes the "knowledge of the observer" at the current point in the strategy and the labelling of the edges denotes the letter that she inputs to the PDA. Initially, according to the observer, the automaton could be in any one of the 4 states. The observer first inputs the letter $\square$. If the top of the stack becomes $\bullet$, then she knows that the automaton is currently either in state 1 or 2. On the other hand, if the top of the stack becomes $\bullet$, then the observer can deduce that the automaton is currently in state 3 or 4. From these two scenarios, by following the appropriate strategy depicted in the figure, we can see that she can synchronise the automaton to state 4. However, if the stack was hidden to the observer, reading either $\diamond$ or $\square$ does not change the knowledge of the observer and therefore, there is no word that can be read that would synchronise the automaton to any state.

We now formalize the notion of an adaptive synchronizing word that we have so far described. Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ be a PDA. Given $S \subseteq Q$, $a \in \Sigma$ and $A \in \Gamma$, let $T^a_{S,A} := \{t \in \delta \mid t = (p, a, A, q, \gamma)$ where $p \in S\}$. Intuitively, if the observer knows that $\mathcal{P}$ is currently in some state in S and the top of the stack is A and she chooses to input a, then $T^a_{S,A}$ is

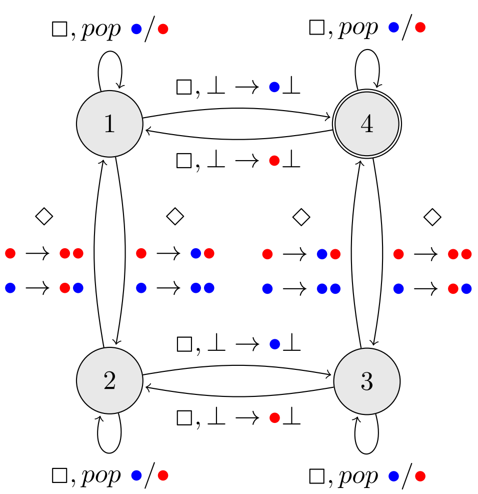

Figure 1 A label of the form $a, A \rightarrow \gamma$ means that if the input is a and if the top of the stack is A, then pop A and push γ.

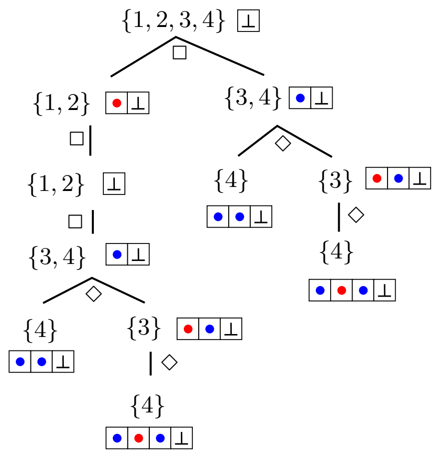

Figure 2 A synchroniser between $(\{1,2,3,4\}, \bot)$ and state 4 for the PDA in Figure 1.

the set of transitions that might take place. We define an equivalence relation $\sim^a_{S,A}$ on the elements of $T^a_{S,A}$ as follows: $t_1 \sim^a_{S,A} t_2 \iff \exists \gamma \in \Gamma^*$ such that $t_1 = (p_1, a, A, q_1, \gamma)$ and $t_2 = (p_2, a, A, q_2, \gamma)$. Notice that if $t_1 \sim^a_{S,A} t_2$ then the observer cannot distinguish occurrences of t_1 from occurrences of t_2. In our running example, if we take $S = \{3, 4\}$, $a = \diamond$ and $A = \bullet$, it is easy to see that $T^a_{S,A}$ is $\{(3, \diamond, \bullet, 4, \bullet\bullet), (4, \diamond, \bullet, 3, \bullet\bullet)\}$ and these two transitions are not in the same equivalence class under $\sim^a_{S,A}$.

The relation $\sim^a_{S,A}$ partitions the elements of $T^a_{S,A}$ into equivalence classes. If E is an equivalence class of $\sim^a_{S,A}$, then notice that there is a word $\gamma \in \Gamma^*$ such that all the transitions in E pop A and push γ onto the stack. This word γ will be denoted by $word(E)$. If we define $next(E) := \{q \mid (p, a, A, q, word(E)) \in E\}$, then $next(E)$ contains all the states that the automaton can move to if *any* of the transitions from E occur. Now, suppose the observer knows that $\mathcal{P}$ is currently in some state in S with A being at the top of the stack. Assuming she inputs the letter a and observes that A has been popped and $word(E)$ has been pushed, she can deduce that $\mathcal{P}$ is currently in some state in $next(E)$. In our running example of $S = \{3, 4\}$, $a = \diamond$ and $A = \bullet$, there are two equivalence classes $E_1 = \{(3, \diamond, \bullet, 4, \bullet\bullet)\}$ and $E_2 = \{(4, \diamond, \bullet, 3, \bullet\bullet)\}$ with $next(E_1) = \{3\}$, $next(E_2) = \{4\}$, $word(E_1) = \{\bullet\bullet\}$ and $word(E_2) = \{\bullet\bullet\}$.

A *pseudo-configuration* of the automaton $\mathcal{P}$ is a pair (S, γ) such that $S \subseteq Q$ and $\gamma \in \Gamma^*$. The pseudo-configuration (S, γ) captures the knowledge of the observer at any given point. Given a pseudo-configuration $(S, A\gamma)$ and an input letter a, let $Succ(S, A\gamma, a) := \{(next(E_1), word(E_1)\gamma), \ldots, (next(E_k), word(E_k)\gamma)\}$ where $E_1, \ldots, E_k$ are the equivalence classes of $\sim^a_{S,A}$. Each element of $Succ(S, A\gamma, a)$ will be called a possible successor of $(S, A\gamma)$ under the input letter a. The function $Succ$ captures all the possible pseudo-configurations that could happen when the observer inputs a at the pseudo-configuration $(S, A\gamma)$.

We now define the notion of a *synchroniser* which will correspond to a strategy for the observer to synchronise the automaton into some state. Let $I \subseteq Q$, $s \in Q$ and $\gamma \in \Gamma^*$. (The I stands for ***Initial set of states***, and the s stands for ***synchronising state***). A *synchroniser* between the pseudo-configuration (I, γ) and the state s, is a labelled tree T such that

- All the edges are labelled by some input letter $a \in \Sigma$ such that, for every vertex v, all its outgoing edges have the same label.

- The root is labelled by the pseudo-configuration (I, γ).
- Suppose v is a vertex which is labelled by the pseudo-configuration $(S, A\eta)$. Let a be the unique label of its outgoing edges and let $Succ(S, A\eta, a)$ be of size k. Then v has k children, with the i^{th} child labelled by the i^{th} pseudo-configuration in $Succ(S, A\eta, a)$.
- For every leaf, there exists $\eta \in \Gamma^*$ such that its label is $(\{s\}, \eta)$.

In addition, if all the leaves are labelled by $(\{s\}, \perp)$, then T is called a *super-synchroniser* between (I, γ) and s. We use the notation $(I, \gamma) \underset{\mathcal{P}}{\Rightarrow} s$ (resp. $(I, \gamma) \xRightarrow[\mathcal{P}]{\text{sup}} s$) to denote that there is a synchroniser (resp. super-synchroniser) between (I, γ) and s in the PDA $\mathcal{P}$. (When $\mathcal{P}$ is clear from context, we drop it from the arrow notation).

2.3 Different Formulations

We now formally introduce the problem which we will refer to as the *adaptive synchronising problem* (Ada-Sync) and it is defined as the following:

Given: A PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ and a word $\gamma \in \Gamma^*$
Decide: Whether there is a state s such that $(Q, \gamma) \Rightarrow s$

The Det-Ada-Sync problem is the same as Ada-Sync, except that the given pushdown automaton is deterministic. Notice that we can generalise the adaptive synchronising problem by the following *subset adaptive synchronising problem* (Subset-Ada-Sync): Given a PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$, a subset $I \subseteq Q$ and a word $\gamma \in \Gamma^*$, decide if there is a state s such that $(I, \gamma) \Rightarrow s$. Similarly, we can define Det-Subset-Ada-Sync.

▶ Remark 1. One can also frame both of these problems in various other ways such as "Given $\mathcal{P}, \gamma$ and q does $(Q, \gamma) \Rightarrow q$?" or "Given $\mathcal{P}, \gamma, I$, is there a q such that $(I, \gamma) \xRightarrow{\text{sup}} q$" etc. We chose this version, because this is similar to the way it is defined for the finite-state version (Problem 1 of [17]). In order to make the lower bounds easier to understand, we introduce a few different variants of Ada-Sync and Subset-Ada-Sync in Section 3 and conclude that they are all polynomial-time equivalent with Ada-Sync. We defer a detailed analysis of the different variants of this problem to future work.

▶ Remark 2. One can relax the notion of a synchroniser and ask instead for an adaptive "homing" word, which is the same as a synchroniser, except that we now only require that if (S, γ) is the label of a leaf then S is *any* singleton. Intuitively, in an adaptive homing word, we are content with knowing the state the automaton is in after applying the strategy, rather than enforcing the automaton to synchronise into some state. To keep the discussion focused on the synchronising problem, in the main paper, we present only the results regarding Ada-Sync and Subset-Ada-Sync. In the full version of the paper, we state the homing word problem formally and prove that it is polynomial-time equivalent to Ada-Sync.

The main results of this paper are now as follows:

▶ **Theorem 3.** *Ada-Sync and Subset-Ada-Sync are both* 2-EXPTIME*-complete. Det-Ada-Sync and Det-Subset-Ada-Sync are both* EXPTIME*-complete.*

3 Equivalence of Various Formulations

In this section, we show that the problems Ada-Sync and Subset-Ada-Sync are polynomial-time equivalent to each other. A similar result is also shown for their corresponding deterministic versions. We note that such a result is not true for finite-state (Moore) machines (Table 1 of [17]) and so we provide a proof of this here, because it illustrates the significance of the stack in the pushdown version.

▶ **Lemma 4.** *Ada-Sync (resp. Det-Ada-Sync) is polynomial time equivalent to Subset-Ada-Sync (resp. Det-Subset-Ada-Sync).*

Proof. It suffices to show that Subset-Ada-Sync (resp. Det-Subset-Ada-Sync) can be reduced to Ada-Sync (resp. Det-Ada-Sync) in polynomial time.

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ be a PDA with $I \subseteq Q$ and $\gamma \in \Gamma^*$. Let q_I be some fixed state in the subset I. Construct $\mathcal{P}'$ from $\mathcal{P}$ by adding a new stack letter # and the following new transitions: Upon reading any $a \in \Sigma$, if the top of the stack is #, then any state $q \in I$ pops # and stays at q whereas any state $q \notin I$ pops # and moves to q_I. Notice that $\mathcal{P}'$ is deterministic if $\mathcal{P}$ is.

It is clear that if $(I, \gamma) \underset{\mathcal{P}}{\Longrightarrow} s$ for some state s, then $(Q, \#\gamma) \underset{\mathcal{P}'}{\Longrightarrow} s$. We now claim that the other direction is true as well. To see this, suppose there is a synchroniser in $\mathcal{P}'$ (say T) between $(Q, \#\gamma)$ and some state s. It is easy to see that, irrespective of the label of the outgoing edge from the root of T, there is only one child of the root which is labelled by (I, γ). Now, no transition pushes # onto the stack and so nowhere else in the synchroniser does # appear in the label of some vertex. It is then easy to see that if we remove the root of T, we get a synchroniser between (I, γ) and s in $\mathcal{P}$. ◀

Lemma 4 allows us to introduce a series of problems which we can prove are poly-time equivalent to Ada-Sync. The reason to consider these problems is that lower bounds for these are substantially easier to prove than for Ada-Sync. The three problems are as follows:

1. Given-Sync: Given a PDA $\mathcal{P}$, a subset I, a word γ and also *a state* s, check if $(I, \gamma) \Rightarrow s$.
2. Super-Sync has the same input as Given-Sync, except we ask if $(I, \gamma) \overset{\text{sup}}{\Longrightarrow} s$.
3. Special-Sync is the same as Super-Sync but restricted to inputs where γ is $\bot$.

▶ **Lemma 5.** *Subset-Ada-Sync, Given-Sync, Super-Sync and Special-Sync are all poly. time equivalent. Further the same applies for their corresponding deterministic versions.*

Because of this lemma, for the rest of this paper, we will only be concerned with the Special-Sync problem, where given a PDA $\mathcal{P}$, a subset I and a state s, we have to decide if $(I, \bot) \overset{\text{sup}}{\Longrightarrow} s$.

4 Lower Bounds

To prove the lower bounds, we introduce the notion of an alternating extended pushdown system (AEPS), which is an extension of pushdown systems with Boolean variables and alternation.

4.1 Alternating Extended Pushdown Systems

An *alternating extended pushdown system* (AEPS) $\mathcal{A}$ is a tuple $(Q, V, \Gamma, \Delta, \mathit{init}, \mathit{fin})$ where Q and V are finite sets of states and Boolean variables respectively, Γ is the stack alphabet, $\mathit{init}, \mathit{fin} \in Q$ are the initial and final states respectively. $\mathcal{A}$ has no input letters but it has a stack to which it can pop and push letters from Γ. Each variable in V is of Boolean type and a transition of $\mathcal{A}$ can apply simple tests on these variables and depending on the outcome, can update their values. A configuration of $\mathcal{A}$ is a tuple (q, γ, F) where $q \in Q, \gamma \in \Gamma^*$ and $F \colon V \to \{0, 1\}$ is a function assigning a Boolean value to each variable.

Let `test` denote the set of *tests* given by $\{v \overset{?}{=} b : v \in V, b \in \{0, 1\}\}$ and let `cmd` denote the set of *commands* given by $\{v \to b : v \in V, b \in \{0, 1\}\}$. A *consistent command* is a conjunction of elements from `cmd` such that for every $v \in V$, both $v \to 0$ and

$v \to 1$ are not present in `cmd`. The transition relation Δ consists of transitions of the form $(q, A, G) \hookrightarrow \{(q_1, \gamma_1, C_1), \ldots, (q_k, \gamma_k, C_k)\}$ where $q, q_1, \ldots, q_k \in Q$, $A \in \Gamma, \gamma_1, \ldots, \gamma_k \in \Gamma^*$, G is a conjunction of elements from `test` and each C_i is a consistent command. Intuitively, at a configuration $(q, A\gamma, F)$ the machine *non-deterministically* selects a transition of the form $(q, A, G) \hookrightarrow \{(q_1, \gamma_1, C_1), \ldots, (q_k, \gamma_k, C_k)\}$ such that the assignment F satisfies the conjunction G and then *forks* into k copies in the configurations $(q_1, \gamma_1\gamma, F[C_1]), \ldots, (q_k, \gamma_k\gamma, F[C_k])$ where $F[C_i]$ is the function obtained by updating F according to the command C_i. With this intuition in mind, we say that a transition $(q, A, G) \hookrightarrow \{(q_1, \gamma_1, C_1), \ldots, (q_k, \gamma_k, C_k)\}$ is enabled at a configuration $(p, B\gamma, F)$ iff $p = q, B = A$ and F satisfies all the tests in G.

A *run* from a configuration (q, η, H) to a configuration (q', η', H') is a tree satisfying the following properties: The root is labelled by (q, η, H). If an internal node n is labelled by the configuration $(p, A\gamma, F)$ then there exists the following transition: $(p, A, G) \hookrightarrow \{(p_1, \gamma_1, C_1), (p_2, \gamma_2, C_2), \ldots, (p_k, \gamma_k, C_k)\}$ which is enabled at $(p, A\gamma, F)$ such that the children of n are labelled by $(p_1, \gamma_1\gamma, F[C_1])$,..., $(p_k, \gamma_k\gamma, F[C_k])$, where $F[C_i](v) = b$ if C_i contains a command of the form $v \to b$ and $F[C_i](v) = F(v)$ otherwise. Finally all the leaves are labelled by (q', η', H'). If a run exists between (q, η, H) and (q', η', H') then we denote it by $(q, \eta, H) \xrightarrow[\mathcal{A}]{*} (q', \eta', H')$. An *accepting run* from a configuration (q, η, H) is a run from (q, η, H) to $(\mathit{fin}, \bot, \mathbf{0})$ where $\mathbf{0}$ is the zero function. An accepting run of an AEPS is simply an accepting run from the initial configuration $(\mathit{init}, \bot, \mathbf{0})$. The emptiness problem is then to decide whether a given AEPS has an accepting run.

By a simple adaptation of the EXPTIME-hardness proof for emptiness of alternating pushdown systems which have no Boolean variables (Theorem 5.4 of [6], Prop. 31 of [23]) we prove that

▶ **Lemma 6.** *The emptiness problem for AEPS is* 2-EXPTIME-*hard.*

An AEPS $\mathcal{A}$ is called a *non-deterministic extended pushdown system* (NEPS) if every transition of $\mathcal{A}$ is of the form $(p, A, F) \hookrightarrow \{(q, \gamma, C)\}$. By Theorem 2 of [13] we have that

▶ **Lemma 7.** *The emptiness problem for NEPS is* EXPTIME-*hard.*

▶ Remark 8. The hardness result for AEPS could also be inferred from Theorem 10 of [13]. Because we use a different notation, for the sake of completeness, we provide the proofs of both of these lemmas in the full version of the paper.

4.2 Reduction from Alternating Extended Pushdown Systems

▶ **Theorem 9.** *Special-Sync, Subset-Ada-Sync and Ada-Sync are all* 2-EXPTIME-*hard. Det-Special-Sync, Det-Subset-Ada-Sync and Det-Ada-Sync are all* EXPTIME-*hard.*

In this subsection, we provide the proof sketches of Theorem 9 by a reduction from the emptiness problem for AEPS to Special-Sync. Let $\mathcal{A} = (Q, V, \Gamma, \Delta, \mathit{init}, \mathit{fin})$ be an AEPS. Without loss of generality, we can assume that if $(q, A, G) \hookrightarrow \{(q_1, \gamma_1, C_1), (q_2, \gamma_2, C_2), \ldots, (q_k, \gamma_k, C_k)\} \in \Delta$, then $\gamma_i \neq \gamma_j$ for $i \neq j$. (This can be accomplished, by prefixing new characters to each γ_i, moving to some intermediate states and then popping the new characters and moving to the respective q_i's). Having made this assumption, the reduction is described below.

From the given AEPS $\mathcal{A}$, we now construct a pushdown automaton $\mathcal{P}$ as follows. The stack alphabet of $\mathcal{P}$ will be Γ. For each transition $t \in \Delta$, $\mathcal{P}$ will have an input letter `in`(t). $\mathcal{P}$ will also have another input letter `end`. The state space of $\mathcal{P}$ will be the set $Q \cup (V \times \{0, 1\}) \cup \{q_{acc}, q_{rej}\}$, where q_{acc} and q_{rej} are two states, which on reading any input letter, will leave the stack untouched and simply stay at q_{acc} and q_{rej} respectively.

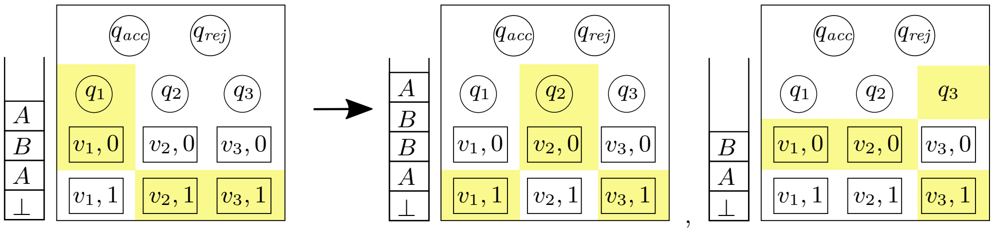

Figure 3 Let t be the transition $(q_1, A, [v_1? = 0, v_3? = 1]) \hookrightarrow \{(q_2, AB, [v_1 \leftarrow 1, v_2 \leftarrow 0]), (q_3, \epsilon, [v_2 \leftarrow 0])\}$ in $\mathcal{A}$. In $\mathcal{A}$, using t, the configuration $C_1 := (q_1, ABA\bot, [v_1 = 0, v_2 = 1, v_3 = 1])$ can fork into $C_2 := (q_2, ABBA\bot, [v_1 = 1, v_2 = 0, v_3 = 1])$ and $C_3 := (q_3, BA\bot, [v_1 = 0, v_2 = 0, v_3 = 1])$.

We now give an intuition behind the transitions of $\mathcal{P}$. Given an assignment $F : V \to \{0, 1\}$ of the Boolean variables V, and a state q of $\mathcal{A}$, we use the notation $[q, F]$ to denote the subset $\{q\} \cup \{(v, F(v)) : v \in V\}$ of states of $\mathcal{P}$. Intuitively, a configuration (q, γ, F) of $\mathcal{A}$ is simulated by its corresponding *pseudo-configuration* $([q, F], \gamma)$ in $\mathcal{P}$.

▶ **Example 10.** The caption of Figure 3 describes an example, where there is a transition t in $\mathcal{A}$, and a configuration C_1 forks into two configurations C_2 and C_3 in $\mathcal{A}$ by using t. The diagram in Figure 3 illustrates the simulation of the forking on the corresponding pseudo-configurations of C_1, C_2, C_3 that the automaton $\mathcal{P}$ will achieve when reading the letter $\texttt{in}(t)$. The shaded part along with the stack content on the left before the arrow denotes the pseudo-configuration of C_1 and upon reading $\texttt{in}(t)$ from this pseudo-configuration, we get two possible successors, each of which correspond to the pseudo-configurations of C_2 and C_3 respectively.

Now we give a formal description of the transitions of $\mathcal{P}$. Let $t = (q, A, G) \hookrightarrow \{(q_1, \gamma_1, C_1), \ldots, (q_k, \gamma_k, C_k)\}$ be a transition of $\mathcal{A}$. Let $p \in Q$. Upon reading $\texttt{in}(t)$, if $p \neq q$ then p immediately moves to the q_{rej} state. Further, even state q moves to the q_{rej} state if the top of the stack is not A. However, if the top of the stack is A, then q pops A and *non-deterministically* pushes any one of $\gamma_1, \ldots, \gamma_k$ onto the stack and if it pushed γ_i, then q moves to the state q_i.

Let $(v, b) \in V \times \{0, 1\}$. Upon reading $\texttt{in}(t)$, if the test $v \stackrel{?}{=} (1 - b)$ appears in the guard G, then (v, b) immediately moves to the q_{rej} state. (Notice that this is a purely syntactical condition on $\mathcal{A}$). Further, if the top of the stack is not A, then once again (v, b) moves to q_{rej}. If these two cases do not hold, then (v, b) pops A and *non-deterministically* picks an $i \in \{1, \ldots k\}$ and pushes γ_i onto the stack. Having pushed γ_i, if C_i does not update the variable v, it stays in state (v, b); otherwise if C_i has a command $v \to b'$, it moves to (v, b').

Finally, upon reading $\texttt{end}$, the states in $[\mathit{fin}, \mathbf{0}]$ move to the q_{acc} state and all the other states in $Q \cup (V \times \{0, 1\})$ move to the q_{rej} state. Notice that there are no outgoing transitions from q_{rej} and so there is no way to move from q_{rej} to q_{acc}.

The following two facts can be easily inferred from the construction of $\mathcal{P}$:

Fact A: Suppose t is a transition of $\mathcal{A}$ which is not enabled at the configuration $(q, A\gamma, F)$. Then, upon reading $\texttt{in}(t)$, there is at least one possible successor (S, η) of the pseudo-configuration $([q, F], A\gamma)$ such that $q_{rej} \in S$.

Fact B: Suppose the configuration $(q, A\gamma, F)$ forks into the following configurations $(q_1, \gamma_1\gamma, F_1), \ldots, (q_k, \gamma_k\gamma, F_k)$ using the transition t in the AEPS $\mathcal{A}$. Then, the possible successors from the pseudo-configuration $([q, F], A\gamma)$ upon reading $\texttt{in}(t)$ in the PDA $\mathcal{P}$ are $([q_1, F_1], \gamma_1\gamma), \ldots, ([q_k, F_k], \gamma_k\gamma)$.

Using these 2 facts, we can then prove that $\mathcal{A}$ has an accepting run iff there is a super-synchroniser in $\mathcal{P}$ between $([init, \mathbf{0}], \bot)$ and q_{acc}. Intuitively, if we have an accepting run of $\mathcal{A}$, then the observer, using Fact B, has a strategy to force $\mathcal{P}$ into one of the states in $([fin, \mathbf{0}])$ with the stack content being $\bot$. Once she does that, she can input the letter `end` and synchronise to the state q_{acc}.

For the reverse direction, with a little case-analysis, we can show that in any super-synchroniser between $([init, \mathbf{0}], \bot)$ and q_{acc}, all non-leaf nodes must be a pseudo-configuration of some configuration in $\mathcal{A}$, and all the parents of a leaf must be labelled by $([fin, \mathbf{0}], \bot)$ Intuitively, then by Facts A and B, such a super-synchroniser must be a simulation of a run in $\mathcal{A}$ (similar to Figure 3) and hence, we can translate it back to an accepting run in $\mathcal{A}$.

Notice that $\mathcal{P}$ is deterministic if $\mathcal{A}$ is non-deterministic. Hence, by Lemmas 6 and 7, we obtain Theorem 9.

5 Upper Bounds

In this section, we will give algorithms that solve Special-Sync and Det-Special-Sync. We first give a reduction from Special-Sync to the problem of checking emptiness in an alternating pushdown system, which we define below. Then, we show that for Det-Special-Sync, the same reduction produces alternating pushdown systems with a "modular" structure, which we exploit to reduce the running time.

5.1 Adaptive Synchronisation for Non-deterministic PDA

An alternating pushdown system (APS) is an alternating extended pushdown system which has no Boolean variables. Since there are no variables, we can suppress any notation corresponding to the variables, e.g., configurations can be just denoted by (q, γ). It is known that the emptiness problem for APS is in EXPTIME (Theorem 4.1 of [4]). We now give an exponential time reduction from Special-Sync to the emptiness problem for APS.

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ be a PDA with $I \subseteq Q$, $s \in Q$. Construct the following APS $\mathcal{A}_\mathcal{P} = (2^Q, \Gamma, \Delta, I, \{s\})$ where Δ is defined as follows: Given $S \subseteq Q, a \in \Sigma$ and $A \in \Gamma$, let $E_1, \ldots, E_k$ be the equivalence classes of the relation $\sim^a_{S,A}$ as defined in subsection 2.2. Then, we have the following transition in $\mathcal{A}_\mathcal{P}$:

$$(S, A) \hookrightarrow \{(next(E_1), word(E_1)), (next(E_2), word(E_2)), \ldots, (next(E_k), word(E_k))\} \quad (1)$$

The following fact is immediate from the definition of a super-synchroniser and from the construction of $\mathcal{A}_\mathcal{P}$.

▶ **Proposition 11.** *Let $S \subseteq 2^Q, \gamma \in \Gamma^*$. Then a labelled tree T is a super-synchroniser between (S, γ) and s in $\mathcal{P}$ if and only if T is an accepting run from (S, γ) in $\mathcal{A}_\mathcal{P}$.*

By Theorem 4.1 of [4], emptiness for APS can be solved in exponential time and so

▶ **Theorem 12.** *Special-Sync is in* 2-EXPTIME

5.2 Adaptive Synchronisation for Deterministic PDA

Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta)$ be a deterministic PDA with $I \subseteq Q, s \in Q$. We have the following proposition, whose proof follows from the fact that $\mathcal{P}$ is deterministic.

▶ **Proposition 13.** *Suppose $S \subseteq Q, a \in \Sigma, A \in \Gamma$ and suppose $E_1, \ldots, E_k$ are the equivalence classes of $\sim^a_{S,A}$. Then, $|S| \geq \sum_{i=1}^{k} |next(E_i)|$.*

Now, given $\mathcal{P}$, consider the APS $\mathcal{A}_{\mathcal{P}} = (2^Q, \Gamma, \Delta, I, \{s\})$ that we have constructed in subsection 5.1. By Proposition 13, we now have the following lemma.

► **Lemma 14.** *For any $S \in 2^Q, \gamma \in \Gamma^*$, any accepting run of $\mathcal{A}_{\mathcal{P}}$ from the configuration (S, γ) has at most $|S|$ leaves.*

The following corollary follows from the lemma above.

► **Corollary 15.** *Any accepting run of $\mathcal{A}_{\mathcal{P}}$ has at most $|I|$ leaves.*

► **Example 16.** Let $\mathcal{P}$ be the deterministic PDA from Figure 1. Figure 4 shows an example of an accepting run in the corresponding APS $\mathcal{A}_{\mathcal{P}}$ from $I := \{1, 2, 3, 4\}$. Notice that there are $|I| = 4$ leaves in this run.

Corollary 15 motivates the study of the following problem, which we call the *sparse emptiness* problem for APSs (Sparse-Empty):

Given: An APS $\mathcal{A}$ and a number k in unary.
Decide: Whether there exists an accepting run for $\mathcal{A}$ with at most k leaves

We prove the following theorem about Sparse-Empty in the next section.

► **Theorem 17.** *Given $\mathcal{A}$ and k, the Sparse-Empty problem can be solved in time $O(|\mathcal{A}|^{ck^2})$ for a fixed constant c.*

Now, because of Proposition 13 and because of the structure of the transitions of $\mathcal{A}_{\mathcal{P}}$ (as given by equation (1)), it is sufficient to restrict the construction of $\mathcal{A}_{\mathcal{P}}$ to only those states which have cardinality at most $|I|$ and hence, it can be assumed that $|\mathcal{A}_{\mathcal{P}}| \leq |\mathcal{P}|^{4|I|}$. This fact, along with Proposition 11, Corollary 15 and Theorem 17 implies the following theorem.

► **Theorem 18.** *Given an instance $(\mathcal{P}, I, s)$ of Det-Special-Sync, checking if $(I, \perp) \xRightarrow[\mathcal{P}]{sup} s$ in time $O(n^{4ck^3})$ where $n = |\mathcal{P}|$ and $k = |I|$ and c is some fixed constant.*

► Remark 19. Note that the algorithm to solve Det-Special-Sync on an instance $(\mathcal{P}, I, s)$, although in EXPTIME, is polynomial if $|I|$ is fixed and quasi-polynomial if $|I|$ is $O(\log |\mathcal{P}|)$.

5.3 "Sparse Emptiness" Checking of Alternating Systems

This subsection is dedicated to proving Theorem 17. We fix an alternating pushdown system $\mathcal{A} = (Q, \Gamma, \Delta, \mathit{init}, \mathit{fin})$ and a number k for the rest of this subsection. A k-accepting run of $\mathcal{A}$ is defined to be an accepting run of $\mathcal{A}$ with at most k leaves. We now split the desired algorithm for Sparse-Empty into three parts. Finally, we give its runtime analysis.

Compressing k-accepting runs of $\mathcal{A}$

We define a non-deterministic pushdown system (NPS) to be a non-deterministic extended pushdown system which has no Boolean variables. From $\mathcal{A}$, we can derive a NPS obtained by deleting all transitions which produces a universal branching, i.e, of the form $(q, A) \hookrightarrow \{(q_1, \gamma_1), \dots, (q_k, \gamma_k)\}$ with $k > 1$. We will denote this NPS by $\mathcal{N}$. Emptiness of NPS is known to be solvable in polynomial time (Theorem 2.1 of [4]). To exploit this fact for our problem, we propose the following notion of a *compressed* accepting run of $\mathcal{A}$. Intuitively, a compressed accepting run is obtained from an accepting run of $\mathcal{A}$ by "compressing" a series of transitions belonging to the non-deterministic part $\mathcal{N}$, into a single transition. An intuition of a compressed accepting run is captured by Figure 5, which is obtained by compressing the run depicted in Figure 4.

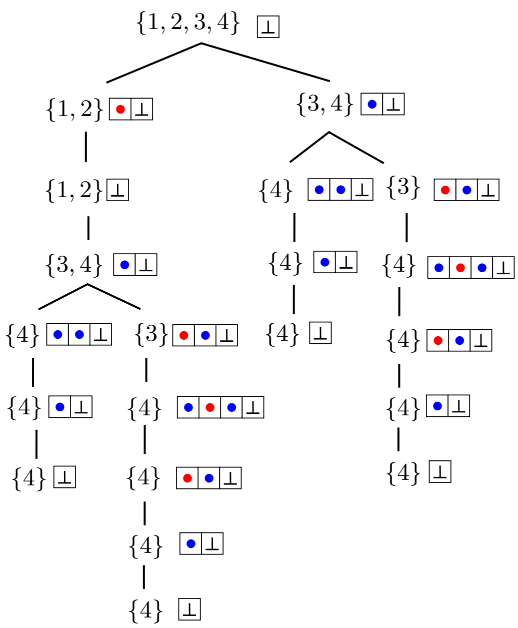

Figure 4 An accepting run of $\mathcal{A}_\mathcal{P}$ for the deterministic PDA $\mathcal{P}$ given in Figure 1.

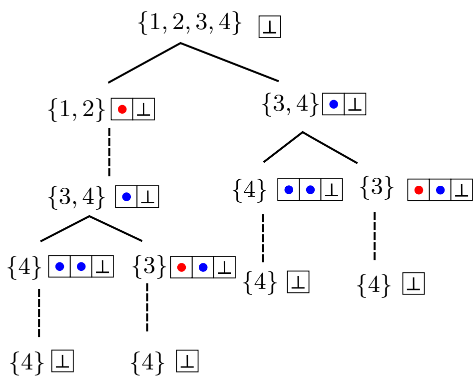

Figure 5 A compressed accepting run of $\mathcal{A}_\mathcal{P}$ for the deterministic PDA $\mathcal{P}$ given in Figure 1, obtained by compressing the run from Figure 4.

Given a tree, we say that a vertex v in the tree is *simple* if it has exactly one child and otherwise we say that it is *complex* (Note that all leaves are complex). A *compressed accepting run* of $\mathcal{A}$ from the configuration (p, η) is a labelled tree such that: The root is labelled by (p, η). If v is a simple vertex labelled by (q, γ) and u is its only child labelled by (q', γ') then u is a complex vertex and $(q, \gamma) \xrightarrow[\mathcal{N}]{*} (q', \gamma')$. If v is a complex vertex labelled by $(q, A\gamma)$ and $v_1, \dots, v_k$ are its children with $k > 1$, then there is a transition $(q, A) \hookrightarrow \{(q_1, A_1), \dots, (q_k, A_k)\}$ in $\mathcal{A}$ such that the label of v_i is $(q_i, A_i\gamma)$. Finally, all the leaves are labelled by $(\mathit{fin}, \bot)$. A compressed accepting run of $\mathcal{A}$ is a compressed accepting run from $(\mathit{init}, \bot)$ and a k-compressed accepting run is a compressed accepting run with at most k leaves. We now have the following lemma.

▶ **Lemma 20.** *There is a k-accepting run of $\mathcal{A}$ from a configuration (p, η) iff there is a k-compressed accepting run of $\mathcal{A}$ from (p, η).*

Searching for k-compressed accepting runs

To fully use the result of Lemma 20, we need some results about non-deterministic pushdown systems, which we state here. Recall that $\mathcal{N}$ is an NPS over the states Q and stack alphabet Γ obtained from the APS $\mathcal{A}$. We say that $M = (Q^M, \Gamma, \delta^M, F^M)$ is an $\mathcal{N}$-*automaton* if M is a non-det. finite-state automaton over the alphabet Γ with accepting states F^M such that for each state $q \in Q$, there is a unique state $q^M \in Q^M$. The set of configurations of $\mathcal{A}$ that are stored by M (denoted by $\mathcal{C}(M)$) is defined to be the set $\{(q, \gamma) : \gamma \text{ is accepted in } M \text{ from the state } q^M\}$. In the above definition, note that Q^M can potentially have more states other than the set $\{q^M \mid q \in Q\}$.

▶ **Example 21.** Let us consider the pushdown automaton in Figure 1, and let $\mathcal{N}$ be the NPS obtained by ignoring the input alphabets $\square$ and $\diamond$. Then observe that from all the states 1,2,3 and 4, with any content on the stack, one can reach state 4 with an empty stack, by popping out all the elements. So, the set of configurations from wich there is an accepting run is $\{(i, \gamma) \mid i \in \{1, 2, 3, 4\}, \gamma \in \perp \cdot \{\bullet, \bullet\}^*\}$. One can define the $\mathcal{N}$-automaton M for it, as an automaton with five states $\{q_1, q_2, q_3, q_4, q_f\}$ where q_f is a final state and each of q_1, q_2, q_3 and q_4 on reading $\perp$ goes to q_f and stays in q_f on reading $\bullet$ or $\bullet$. It is easy to see that this automaton accepts all words of the form $\perp \cdot \{\bullet, \bullet\}^*$.

▶ **Theorem 22** (Section 2.3 and Theorem 2.1 of [4])**.** *Given an $\mathcal{N}$-automaton M, in time polynomial in $\mathcal{N}$ and M, we can construct an $\mathcal{N}$-automaton M' which has the* same states *as M such that M' stores the set of predecessors of M, i.e., $\mathcal{C}(M') = \{(q', \gamma') : \exists (q, \gamma) \in \mathcal{C}(M)$ such that $(q', \gamma') \xrightarrow[\mathcal{N}]{*} (q, \gamma)\}$.*

We say that an unlabelled tree is *structured*, if the child of every simple vertex is a complex vertex. An ℓ-structured tree is simply a structured tree which has at most ℓ leaves. Notice that the height of an ℓ-structured tree is $O(\ell)$ and since it has at most ℓ leaves, it follows that a ℓ-structured tree can be described using a polynomial number of bits in ℓ. Hence, the number of ℓ-structured trees is $O(2^{\ell^c})$ for some fixed c.

Now let us come back to the problem of searching for k-accepting runs of $\mathcal{A}$. By Lemma 20 it suffices to search for a k-compressed accepting run of $\mathcal{A}$. Notice that if we take a k-compressed accepting run and remove its labels, we get a k-structured tree. Now, suppose we have an algorithm `Check` that takes a k-structured tree T and checks if T can be labelled to make it a k-compressed accepting run of $\mathcal{A}$. Then, by calling `Check` on every k-structured tree, we have an algorithm to check for the existence of a k-compressed accepting run of $\mathcal{A}$. Hence, it suffices to describe this procedure `Check` which is what we will do now.

The algorithm `Check`

Let T be a k-structured tree. For each vertex v in the tree T, `Check` will assign a $\mathcal{N}$-automaton M_v such that M_v will have the following property:

> Invariant (*) : A configuration $(q, \gamma) \in \mathcal{C}(M_v)$ iff all the vertices of the subtree rooted at v can be labelled such that the resulting labelled subtree is a compressed accepting run of $\mathcal{A}$ from (q, γ).

The construction of each M_v is as follows: Let Q be the states and Δ be the transitions of the alternating pushdown system $\mathcal{A}$.

- Suppose vertex v is a leaf. We let M_v be an automaton such that $\mathcal{C}(M_v) = \{(\mathit{fin}, \perp)\}$. Notice that such a M_v can be easily constructed in polynomial time.
- Suppose vertex v is simple and u is its child. We take M_u and use Theorem 22 to construct the $\mathcal{N}$-automaton M_v. Note that M_v has the same set of states as M_u.
- Suppose v is complex and suppose $v_1, \ldots, v_\ell$ are its children. For each $1 \leq i \leq \ell$ and for every configuration (q, γ) of $\mathcal{A}$, let $\delta_i(q^{M_{v_i}}, \gamma)$ denote the set of states that the automaton M_{v_i} will be in after reading γ from the state $q^{M_{v_i}}$. To construct M_v first do a product construction $M_{v_1} \times M_{v_2} \times \cdots \times M_{v_\ell}$, so that the resulting product automaton stores precisely the set of configurations which are stored by each of the individual automata $M_{v_1}, \ldots, M_{v_\ell}$. Then, for each $q \in Q$, add a state q^{M_v}. Then for each transition $(p, A) \hookrightarrow \{(p_1, \gamma_1), \ldots, (p_\ell, \gamma_\ell)\}$ in Δ, add a transition in M_v, which upon reading A, takes p^{M_v} to any of the states in $\delta_1({p_1}^{M_{v_1}}, \gamma_1) \times \delta_2({p_2}^{M_{v_2}}, \gamma_2) \times \cdots \times \delta_l({p_\ell}^{M_{v_\ell}}, \gamma_\ell)$. Intuitively, we accept a word $A\gamma$ from the state p^{M_v} if for each i, the word $\gamma_i\gamma$ can be accepted from the state ${p_i}^{M_{v_i}}$.

▶ **Proposition 23.** *For each vertex v of the tree T, M_v satisfies invariant (*)*

Finally, we accept iff $(init, \perp) \in \mathcal{C}(M_r)$ where r is the root of the tree. The correctness of `Check` follows from the proposition above.

Running time analysis

Let us analyse the running time of `Check`. Let T be a k-structured tree and therefore T has $O(k^2)$ vertices. `Check` assigns to each vertex v of T an automaton M_v. We claim that the running time of `Check` is $O(k^2 \cdot |\mathcal{A}|^{ck^2})$ (for some fixed constant c) because of the following facts:

1) By induction on the structure of the tree T, it can be proved that, there exists a constant d, such that if h_v is the height of a vertex v and l_v is the number of leaves in the sub-tree of v, then the number of states of M_v is $O(|\mathcal{A}|^{dh_v l_v})$ (Recall that $h_v l_v$ is at most $O(k^2)$).
2) If an $\mathcal{N}$-automaton has n states, then the number of transitions it can have is $O(|\mathcal{A}| \cdot n^2)$.
3) For a vertex v with children $v_1, \ldots, v_\ell$, M_v can be constructed in polynomial time in the size of $|M_{v_1}| \times |M_{v_2}| \times \ldots |M_{v_\ell}|$ and $|\mathcal{A}|$.

Notice that everything else apart from Fact 1) is easy to see. To prove Fact 1), we proceed by bottom-up induction on the structure of the tree T. For the base case when the vertex v is a leaf, notice that we can easily construct the required automaton M_v with at most $O(|\mathcal{A}|)$ states. Suppose, v is a simple vertex and u its only child. By Theorem 22, M_v has the same set of states as M_u. By induction hypothesis, the number of states of M_u is $O\left(|\mathcal{A}|^{dh_u l_u}\right)$ and so the number of states of M_v is $O\left(|\mathcal{A}|^{dh_v l_v}\right)$. Suppose v is a complex vertex and $v_1, \ldots, v_\ell$ are its children. Let h be the maximum height amongst the vertices $v_1, \ldots, v_\ell$. By induction hypothesis, the number of states of each M_{v_i} is $O\left(|\mathcal{A}|^{dhl_{v_i}}\right)$. It is then clear that the number of states of M_v is $O\left(\prod_{i=1}^{\ell} |\mathcal{A}|^{dhl_{v_i}} + |\mathcal{A}|\right) = O\left(|\mathcal{A}|^{dhl_v} + |\mathcal{A}|\right) = O\left(|\mathcal{A}|^{d(h+1)l_v}\right) = O\left(|\mathcal{A}|^{dh_v l_v}\right)$.

Now the final algorithm for Sparse-Empty simply iterates over all k-structured trees and calls `Check` on all of them. Since the number of k-structured trees is at most $f(k)$ where f is an exponential function, it follows that the total running time is $O\left(f(k) \cdot k^2 \cdot |\mathcal{A}|^{ck^2}\right) = O(|\mathcal{A}|^{ek^2})$ for some constant e.

6 Conclusion

Our results can be considered as a step in the research direction recently proposed by Fernau, Wolf and Yamakami in [12], in which the authors prove that the synchronisation problem for PDAs is undecidable when the stack is *not* visible. They also suggest looking into different variants of synchronisation for PDAs with a view towards the decidability and complexity frontier. Within this context, we believe we have proposed a natural variant of synchronisation in which the observer can see the stack and given decidability and complexity-theoretic optimal results for both the non-deterministic and the deterministic cases.

As future work, it might be interesting to consider the adaptive synchronising problem for subclasses of pushdown automata such as one-counter automata and visibly pushdown automata. It might also be interesting to consider the problem of looking for *short* adaptive synchronisers, i.e., adaptive synchronisers whose size is not bigger than a given bound.

References

1. A. R. Balasubramanian and K. S. Thejaswini. Adaptive synchronisation of pushdown automata, 2021. arXiv:2102.06897.
2. Marie-Pierre Béal, Eugen Czeizler, Jarkko Kari, and Dominique Perrin. Unambiguous automata. *Math. Comput. Sci.*, 1(4):625–638, 2008. doi:10.1007/s11786-007-0027-1.
3. Yaakov Benenson, Rivka Adar, Tamar Paz-Elizur, Zvi Livneh, and Ehud Shapiro. Dna molecule provides a computing machine with both data and fuel. *Proceedings of the National Academy of Sciences*, 100(5):2191–2196, 2003.
4. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference, Warsaw, Poland, July 1-4, 1997, Proceedings*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997. doi:10.1007/3-540-63141-0_10.
5. Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, volume 3472 of *LNCS*. Springer, 2005.
6. Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. doi:10.1145/322234.322243.
7. Dmitry Chistikov, Pavel Martyugin, and Mahsa Shirmohammadi. Synchronizing automata over nested words. *J. Autom. Lang. Comb.*, 24(2-4):219–251, 2019. doi:10.25596/jalc-2019-219.
8. Laurent Doyen, Line Juhl, Kim Guldstrand Larsen, Nicolas Markey, and Mahsa Shirmohammadi. Synchronizing words for weighted and timed automata. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, volume 29 of *LIPIcs*, pages 121–132. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.FSTTCS.2014.121.
9. Laurent Doyen, Thierry Massart, and Mahsa Shirmohammadi. The complexity of synchronizing markov decision processes. *J. Comput. Syst. Sci.*, 100:96–129, 2019. doi:10.1016/j.jcss.2018.09.004.
10. David Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990.
11. Henning Fernau and Petra Wolf. Synchronization of deterministic visibly push-down automata. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020, December 14-18, 2020, BITS Pilani, K K Birla Goa Campus, Goa, India (Virtual Conference)*, volume 182 of *LIPIcs*, pages 45:1–45:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSTTCS.2020.45.
12. Henning Fernau, Petra Wolf, and Tomoyuki Yamakami. Synchronizing deterministic push-down automata can be really hard. In Javier Esparza and Daniel Král', editors, *45th International Symposium on Mathematical Foundations of Computer Science, MFCS 2020, August 24-28, 2020, Prague, Czech Republic*, volume 170 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.MFCS.2020.33.
13. Patrice Godefroid and Mihalis Yannakakis. Analysis of boolean programs. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2013. doi:10.1007/978-3-642-36742-7_16.
14. F. C. Hennine. Fault detecting experiments for sequential circuits. In *1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, pages 95–110, 1964.

15 Chris Keeler and Kai Salomaa. Alternating finite automata with limited universal branching. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications - 14th International Conference, LATA 2020, Milan, Italy, March 4-6, 2020, Proceedings*, volume 12038 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2020. `doi:10.1007/978-3-030-40608-0_13`.

16 Arun Lakhotia, Eric Uday Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31(11):955–968, 2005. `doi:10.1109/TSE.2005.120`.

17 Kim Guldstrand Larsen, Simon Laursen, and Jirí Srba. Synchronizing strategies under partial observability. In Paolo Baldan and Daniele Gorla, editors, *CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings*, volume 8704 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2014. `doi:10.1007/978-3-662-44584-6_14`.

18 D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

19 Balas K Natarajan. An algorithmic approach to the automated design of parts orienters. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 132–142. IEEE, 1986.

20 Karin Quaas and Mahsa Shirmohammadi. Synchronizing data words for register automata. *ACM Trans. Comput. Log.*, 20(2):11:1–11:27, 2019. `doi:10.1145/3309760`.

21 Fu Song and Tayssir Touili. Pushdown model checking for malware detection. *Int. J. Softw. Tools Technol. Transf.*, 16(2):147–173, 2014. `doi:10.1007/s10009-013-0290-1`.

22 Mikhail V. Volkov. Synchronizing automata and the cerny conjecture. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, volume 5196 of *Lecture Notes in Computer Science*, pages 11–27. Springer, 2008. `doi:10.1007/978-3-540-88282-4_4`.

23 Igor Walukiewicz. Pushdown processes: Games and model-checking. *Inf. Comput.*, 164(2):234–263, 2001. `doi:10.1006/inco.2000.2894`.

24 Ján Černý. Poznámka k homogénnym experimentom s konečnými automatmi. *Matematicko-fyzikálny časopis*, 14(3):208–216, 1964.