Synchronizing automata over nested words

Dmitry Chistikov[1], Pavel Martyugin[2], and Mahsa Shirmohammadi[3]

[1] Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern and Saarbrücken, Germany, dch@mpi-sws.org
[2] Institute of Mathematics and Computer Science, Ural Federal University, Ekaterinburg, Russia, martuginp@gmail.com
[3] University of Oxford, UK, mahsa.shirmohammadi@cs.ox.ac.uk

Abstract. We extend the concept of a synchronizing word from finite-state automata (DFA) to nested word automata (NWA): A well-matched nested word is called synchronizing if it resets the control state of any configuration, i.e., takes the NWA from all control states to a single control state.
We show that although the shortest synchronizing word for an NWA, if it exists, can be (at most) exponential in the size of the NWA, the existence of such a word can still be decided in polynomial time. As our main contribution, we show that deciding the existence of a short synchronizing word (of at most given length) becomes PSPACE-complete (as opposed to NP-complete for DFA). The upper bound makes a connection to pebble games and Strahler numbers, and the lower bound goes via cost-optimal synchronizing words for DFA, an intermediate problem that we also show PSPACE-complete.

1 Introduction

The concept of a synchronizing word for finite-state machines has been studied in automata theory for more than half a century [22, 19]. Given a deterministic finite automaton (DFA) $\mathcal{D}$ over an input alphabet Σ, a word w is called *synchronizing* for $\mathcal{D}$ if, no matter which state $q \in Q$ the automaton $\mathcal{D}$ starts from, the word w brings it to some specific state $\bar{q}$ that only depends on w but not on q. Put differently, a synchronizing word *resets* the state of an automaton. If the state of $\mathcal{D}$ is initially unknown to an observer, then feeding $\mathcal{D}$ with input w effectively restarts $\mathcal{D}$, making it possible for the observer to rely on the knowledge of the current state henceforth.

In this paper we extend the concept of synchronizing words to so-called *nested words*. This is a model that extends usual words by imparting a parenthetical structure to them: some letters in a word are declared *calls* and *returns*, which are then matched to each other in a uniquely determined "nesting" (non-crossing) way. On the language acceptor level, this hybrid structure (linear sequence of letters with matched pairs) corresponds to a pushdown automaton where every letter in the input word is coupled with the information on whether the automaton should push, pop, or not touch the pushdown (the stack). Such machines were first studied by Mehlhorn [15] under the name of *input-driven pushdown automata* in 1980 and have recently received a lot of attention under the name of *visibly pushdown automata*. The latter term, as well as the model of nested words and *nested word automata* (in NWA the matching relation remains a separate entity, while in input-driven pushdown automata it is encoded in the input alphabet), is due to Alur and Madhusudan [1].

The tree-like structure created by matched pairs of letters occurs naturally in many domains; for instance, nested words mimic traces of programs with procedures (which have pairs of calls and returns), as well as documents in eXtensible Markup Language (XML documents, ubiquitous today, have pairs of opening and closing tags). This makes the nested words model very appealing; at the same time, nested words and NWA enjoy many nice properties of usual words and finite-state machines: for example, constructions of automata for operations over languages, and many decidability properties naturally carry over to nested words—a fact widely used in software verification (see, e.g., [6] and references therein). This suggests that the classic concept of a synchronizing word may have an interesting and meaningful extension in the realm of nested words.

Our contribution and discussion. Nested word automata are essentially an expressive subclass of pushdown automata and, as such, define infinite-state transition systems (although the number of *control states* is only finite, the number of *configurations*—incorporating the state of the pushdown store—is already infinite). Finding the right definition for *synchronizing nested words* becomes for this reason a question of relevance: in the presence of infinitely many configurations not all of them may even have equal-length paths to a single designated one (this phenomenon is known and arises, for instance, in weighted automata [5]). In fact, any nested word w, given as input to an NWA, changes the stack height in a way that does not depend on the initial control state (and can only depend on the initial configuration if w has unmatched returns). We thus choose to define synchronizing words as those that reset the control state of the automaton and leave the pushdown store (the stack) unchanged (Definition 1; cf. location-synchronization in [5]). Consider, for instance, an XML processor that does not keep a heap storage and invokes and terminates its internal procedures in lock-step with opening and closing tags in the input; our definition of a synchronizing word corresponds to an XML document that "resets" the local variables.

Building on this definition, we show that shortest synchronizing words for NWA can be exponential in the size of the automaton (Example 2), in contrast to the case of DFA: every DFA with n states, if it has a synchronizing word, also has one of length polynomial in n. The best known worst-case upper bound on the length of a synchronizing word is $(n^3 - n)/6$, due to Pin [17]; Černý conjectured in the 1960s [21] that $(n-1)^2$ is a valid upper bound, but as of now there is a gap between his quadratic lower bound and the cubic upper bound of Pin (see [22] for a survey). In the case of nested words, the exponential comes from the repeated doubling phenomenon, typical for pushdown automata.

Although the length of the shortest synchronizing word can be exponential, it turns out that the existence of such a word—which, in fact, cannot be longer than exponential—can be decided in polynomial time (Theorem 3), akin to the DFA case. However, generalizing the definition in standard ways (synchronizing from a subset instead of all states, or to a subset of states instead of singletons) raises the complexity to exponential time (Theorem 4); for DFA, the complexity is polynomial space [18]. The lower bounds are by reduction from the intersection nonemptiness problem, which is known to be complete for polynomial space in the case of DFA [12] and which we observe to be complete for exponential time over nested words (Lemma 5).

Our main technical contribution is characterizing the complexity of deciding existence of *short* synchronizing words, where the bound on the length is given as part of the input. In the DFA case, this problem is **NP**-complete as shown by Eppstein [7], and for NWA it becomes **PSPACE**-complete (Theorem 6). We believe that both upper and lower bound techniques that we use to prove this result are of interest.

Specifically, for the upper bound (Section 4) we first encode the unranked trees (which represent nested words) with ranked trees. This reduces the search for a short synchronizing nested word to the search for a tree that satisfies a number of local properties. These properties, in turn, can be captured as acceptance by a certain tree automaton of exponential size. We then show that guessing an accepting computation for such a machine—which amounts to guessing an exponentially large tree—can be done in polynomial space. To do this, we rely on the concept of *(black) pebbling games*, developed in the theory of computational complexity for the study of deterministic space-bounded computation (see, e.g., [20, Chapter 10]). We simulate optimal strategies for trees in such games [13], whose efficiency is determined by so-called *Strahler numbers* [10]. Previous use of this technique in formal language theory and verification is primarily associated with derivations of context-free grammars, see, e.g., [8, 9] and [10] for a survey. In this body of work, closest to ours are apparently arguments due to Chytil and Monien [3]. We believe that our key procedure—which can decide *bounded nonemptiness* of *succinct tree automata*—may be of use in other domains as well.

Finally, for the matching polynomial-space lower bound (Section 5) we construct a two-step reduction from the problem of existence of *carefully* synchronizing words for partial DFA, whose hardness is known [14]. We define an intermediate problem of *cost-optimal synchronization* for DFA, where every letter in the alphabet comes with a price and the task is to decide existence of a synchronizing word whose total cost does not exceed the budget. We show that this natural problem is complete for polynomial space (this strengthens previous results from [11, 5], where prices can be state-dependent). After this, we basically simulate price-equipped DFA using NWA and relying on the above-mentioned repeated doubling phenomenon. We find it interesting that this "counting" feature of nested words alone is a ground for hardness.

We mention without proof that some of our techniques naturally extend to (going via) tree automata over ranked trees.

2 Nested words and nested word automata

A *nested word* of length k over a finite alphabet Σ is a pair $u = (x, \nu)$, where $x \in \Sigma^k$ and ν is a *matching relation* of length k, that is, a subset $\nu \subseteq \{-\infty, 1, \ldots, k\} \times \{1, \ldots, k, +\infty\}$ such that, first, if $\nu(i,j)$ holds, then $i < j$; second, for $1 \le i \le k$ each of the sets $\{j \mid \nu(i,j)\}$ and $\{j \mid \nu(j,i)\}$ contains at most one element; third, whenever $\nu(i,j)$ and $\nu(i',j')$, it cannot be the case that $i < i' \le j < j'$. We assume that $\nu(-\infty, +\infty)$ never holds.

If $\nu(i,j)$, then the position i in the word u is said to be a *call*, and the position j a *return*. All positions from $\{1, \ldots, k\}$ that are neither calls nor returns are *internal*. A call (a return) is *matched* if ν matches it to an element of $\{1, \ldots, k\}$ and *unmatched* otherwise. We shall call a nested word *well-matched* if it has no unmatched calls and no unmatched returns.

Define a *nested word automaton* (an NWA) over the input alphabet Σ as a structure $\mathcal{A} = (Q, \Gamma, \delta, q_0, \gamma^{\mathsf{i}})$, where:

- Q is a finite non-empty set of control states,
- Γ is a finite set of stack symbols,
- $\delta = (\delta^{\mathsf{call}}, \delta^{\mathsf{int}}, \delta^{\mathsf{ret}})$, where
  - $\delta^{\mathsf{int}} \colon Q \times \Sigma \to Q$ is an internal transition function,
  - $\delta^{\mathsf{call}} \colon Q \times \Sigma \to Q \times \Gamma$ is a call transition function,
  - $\delta^{\mathsf{ret}} \colon \Gamma \times Q \times \Sigma \to Q$ is a return transition function,
- $q_0 \in Q$ is the initial control state, and
- $\gamma^{\mathsf{i}} \in \Gamma$ is the initial stack symbol.

A *configuration* of $\mathcal{A}$ is a tuple $(q, s) \in Q \times \Gamma^*$. We say $(q, s) \xrightarrow{w} (q', s')$ for nested words w by imposing the following conditions. Suppose $w = (x, \nu)$ is of length 1, then:

- if w is an internal position, then $\delta^{\mathsf{int}}(q, x) = q'$ and $s' = s$;
- if w is a call, then $\delta^{\mathsf{call}}(q, x) = (q', \gamma)$ and $s' = s\gamma$ for some $\gamma \in \Gamma$;
- if w is a return, then:
  - either $\delta^{\mathsf{ret}}(\gamma, q, x) = q'$ and $s = s'\gamma$,
  - or $\delta^{\mathsf{ret}}(\gamma^{\mathsf{i}}, q, x) = q'$ and $s = s' = \varepsilon$.

Now $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$, and the input words on top of the arrow notation are concatenated appropriately.

Alternatively, nested words can be seen as words over a *nested alphabet* $\langle\Sigma \cup \Sigma \cup \Sigma\rangle$, where $\langle\Sigma$ and $\Sigma\rangle$ are disjoint copies of Σ that contain letters of the form $\langle a$ and $a\rangle$, respectively, for each $a \in \Sigma$. Every word w over this nested alphabet is unambiguously associated with a matching relation ν_w of length $|w|$ where positions with elements of $\langle\Sigma$, Σ, and $\Sigma\rangle$ are calls, internal positions, and returns, respectively; the word w can thus identified with a nested word (w, ν_w). The automaton $\mathcal{A}$ can then be viewed as an ϵ-free pushdown automaton over the nested alphabet $\langle\Sigma \cup \Sigma \cup \Sigma\rangle$ in which the direction of stack operations (i.e., whether the automaton pushes, pops, or does not touch the stack) are determined by whether the current position belongs to $\langle\Sigma$, Σ, or $\Sigma\rangle$. Such automata are known under the names *input-driven pushdown automata* and *visibly pushdown automata*. A *path* (*run*, *computation*) through an automaton $\mathcal{A}$ driven by an input word $u = a_1 \dots a_k$, where each $a_i \in \langle\Sigma \cup \Sigma\rangle \cup \Sigma$, is a sequence of configurations (q_i, s_i), $i = 0, \dots, k$, with $(q_0, s_0) = (q_0, \varepsilon)$ and $(q_{i-1}, s_{i-1}) \xrightarrow{a_i} (q_i, s_i)$ for all i.

3 Synchronizing words for NWA

Informally, a well-matched nested word u is synchronizing for an NWA $\mathcal{A}$ if it takes $\mathcal{A}$ from all control states to some single control state. Note that the result of feeding any well-matched word to an NWA does not depend on the stack contents; if $(q', s') \xrightarrow{u} (q'', s'')$ and u is well-matched, then $s' = s''$. This lets us extend the definition of $\longrightarrow$ to sets of states: we write $(Q', s) \xrightarrow{u} (Q'', s)$ if, first, the word u is well-matched, second, for all $q' \in Q'$ there exists a $q'' \in Q''$ such that $(q', s) \xrightarrow{u} (q'', s)$, and, third, for every state $q'' \in Q''$ there exists a $q' \in Q'$ such that $(q', s) \xrightarrow{u} (q'', s)$. If $Q'' = \{q''\}$, we write (q'', s) instead of $(\{q''\}, s)$.

Definition 1. *A well-matched nested word u is* synchronizing *for an NWA $\mathcal{A} = (Q, \Gamma, \delta, q_0, Q^{\mathsf{f}}, \gamma^{\mathsf{i}})$ if there exists a control state $\bar{q} \in Q$ such that the relation $(Q, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$ holds. (By the remark above, this happens if and only if for all $q \in Q$ and for all $s \in \Gamma^*$ the relation $(q, s) \xrightarrow{u} (\bar{q}, s)$ holds.)*

Remark. Definition 1 crucially relies on the nested structure of the input word, in that this structure determines the stack behaviour of the NWA. Extending this definition to the general case of pushdown automata (PDA) would face the difficulties outlined in the introduction; to the best of our knowledge, no such extension has been proposed to date. The term “synchronization” in the context of PDA is known to be used when referring to the agreement between the transitions taken by the automaton and an external structure [2]: in NWA, for example, the input symbols and the stack actions are synchronized (in this sense).

Example 2. Given $n \geq 1$, we construct an NWA $\mathcal{A}_n$ with $O(\log n)$ control states and $O(1)$ stack symbols such that the shortest synchronizing word for $\mathcal{A}_n$ has length exactly n.

Our construction is inductive. We first construct a family of *incomplete* NWA $\mathcal{B}_n$ with stack symbols $\{\mathsf{x}, \mathsf{y}\}$ and two designated states q_{x} and q_{y}. In $\mathcal{B}_n$, the shortest run from q_{x} to q_{y} is driven by some well-matched nested word w of length n, and along this run the state q_{y} is not visited. These NWA will be incomplete, in the sense that their transition functions will only be partial; redirecting all missing transitions to the initial state in would make these NWA complete. From $\mathcal{B}_n$, we construct another NWA $\mathcal{B}_{2n+4}$ and $\mathcal{B}_{2n+5}$ where the length of the shortest run between two new states in and out is exactly $2n + 4$ and $2n + 5$. The construction of $\mathcal{B}_{2n+4}$ is depicted in Figure 1. Here the shortest run from in to out is over $\mathsf{call}(\mathsf{x}) \cdot w \cdot \mathsf{ret}(\mathsf{x}) \cdot \mathsf{call}(\mathsf{y}) \cdot w \cdot \mathsf{ret}(\mathsf{y})$ and has length $2n+4$; splitting state q_{z} in two states where all transitions from one direct to the other,

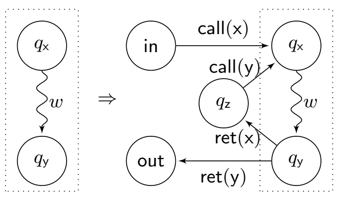

Figure 1. Doubling transformation

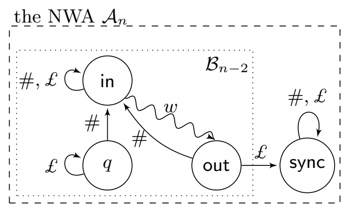

Figure 2. The NWA $\mathcal{A}_n$ from $\mathcal{B}_{n-2}$.

gives us $\mathcal{B}_{2n+5}$. We call this transformation *doubling*. For all $n \geq 4$ the NWA $\mathcal{B}_n$ can be constructed by several doubling transformations starting from one of the automata $\mathcal{B}_0, \mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ (which are simply NWAs with $1, 2, 3, 4$ states). The size of $\mathcal{B}_n$ is $O(\log n)$.

For all $n \geq 2$, from the NWA $\mathcal{B}_{n-2}$ we construct an NWA $\mathcal{A}_n$ where the shortest synchronizing word has length exactly n. Figure 2 shows the sketch of the construction: there are two new letters $\#$ and $£$ and a new absorbing state sync. From all states q of $\mathcal{A}_n$, the letter $\#$ resets the NWA to in whereas $£$-transitions are all self-loops except in the state out where out $\xrightarrow{\#}$ sync. All missing transitions are directed to the state in (note that even in case of DFA, existence synchronizing words in the presence of *partial* transition functions is **PSPACE**-complete [14]; it is thus of utmost importance that our NWA are complete). Observe that the shortest synchronizing word has length exactly n; it is $\# \cdot w \cdot £$ where w is the shortest word that brings $\mathcal{B}_{n-2}$ from in to out.

Remark. Our Example 2 seems to use a "non-uniform" set of call, return, and internal symbols, but this is easily remedied by making some of the symbols indistinguishable. All call positions in the word are simply call, and all return position are ret; in figures, the letter in parentheses is the pushed resp. popped stack symbol.

In decision problems that we study in this paper, the *size* of an automaton is proportional to $|\Gamma| \cdot |\Sigma| \cdot |Q|$.

Theorem 3. *If an NWA $\mathcal{A}$ has a synchronizing word, then it has one of length at most exponential in the size of $\mathcal{A}$. Moreover, the existence of a synchronizing word can be decided in time polynomial in the size of $\mathcal{A}$.*

Theorem 4. *The following decision problems, with an NWA $\mathcal{A}$ part of the input, are* **EXP***-complete:*
(1) *Given a subset $I \subseteq Q$, decide if there exists a well-matched nested word u such that $(I, \varepsilon) \xrightarrow{u} (\bar{q}, \varepsilon)$ for some state $\bar{q} \in Q$.*
(2) *Given a subset $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(Q, \varepsilon) \xrightarrow{u} (F', \varepsilon)$ for some subset $F' \subseteq F$.*
(3) *Given subsets $I \subseteq Q$ and $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(I, \varepsilon) \xrightarrow{u} (F', \varepsilon)$ for some subset $F' \subseteq F$.*

The corresponding decision problems for DFA are **PSPACE**-complete [18], where hardness is by reduction from the DFA intersection non-emptiness problem (see [23] for a more refined complexity analysis). In the NWA case, the proofs are an easy adaptation of these arguments and are based on the following observation, which can be proved by translation from tree automata or by a direct extension of Kozen's proof [12]:

Lemma 5. *The following problem is* **EXP***-complete: Given NWA $\mathcal{A}_1, \ldots, \mathcal{A}_m$, decide if there exists a well-matched word accepted by all $\mathcal{A}_i$.*

The following theorem is our main result.

Theorem 6. *The following problem* Short Synchronizing Nested Word *is* **PSPACE***-complete: Given an NWA $\mathcal{A}$ and an integer $\ell \geq 1$ written in binary, decide if $\mathcal{A}$ has a synchronizing word u of length at most ℓ.*

The corresponding decision problem for DFA is **NP**-complete [7]. (Note that deciding if the shortest synchronizing word has length exactly k, a related but different problem, is **DP**-complete [16].) Since any DFA with a synchronizing word has one of length cubic in its size, it does not matter for DFA if ℓ is written in binary or in unary. In contrast, as our Example 2 shows, NWA may need an exponentially long word for synchronization; this explains the choice of the setting above. (In the alternative version, i.e., if ℓ is written in unary, the problem is **NP**-complete: the upper bound is a guess-and-check argument, and hardness already holds for DFA.)

4 Upper bound of Theorem 6

In this section, we show that the following problem is in **PSPACE**: Given a nested word automaton $\mathcal{A}$ and an integer $\ell \geq 1$ written in binary, decide if there exists a synchronizing word for $\mathcal{A}$ of length at most ℓ. In fact, we can also adjust our arguments (see subsection 4.2) so that they give a **PSPACE** upper bound for another problem: Given a nested word automaton $\mathcal{A}$, two subsets of its control states $I, F \subseteq Q$, and an integer $\ell \geq 1$ written in binary, decide if there exists a well-matched word of length at most ℓ that takes all states in I to F.

The plan of the proof is as follows. We encode nested words using binary trees (subsection 4.1), so that runs of NWA correspond to computations of tree automata and synchronizing words to tuples of such computations (subsection 4.2). Thus the task of guessing a short synchronizing word is reduced to the task of guessing an accepting computation of a tree automaton on an unknown binary tree of potentially exponential size (Lemma 8); this is the same as guessing an exponentially large binary tree subject to local conditions. We prove that it's possible to solve this *bounded nonemptiness* problem in polynomial space, even if the tree automaton in question has exponentially many states and is only given in symbolic form (subsection 4.4); our solution relies on the concepts of pebble games and Strahler numbers (subsection 4.3).

4.1 Binary tree representation of nested words

In this subsection we describe a representation of nested words with binary trees that we use in the sequel. Because of space constraints, the full description can be found in Appendix and here we only give a short descriptive summary.

Nested words as binary trees. We denote the binary *tree representation* of a nested word u by $\mathsf{bin}(u)$. The explicit construction of $\mathsf{bin}(u)$ is not sophisticated (see Appendix for details), but nodes of $\mathsf{bin}(u)$ come in several different *types*. We did not attempt to minimize the number of these types; different representations are, of course, also possible.

Degree	Type	Notes
2	call-return binary	Associated with matched pair $\langle x_i, x_j \rangle$
2	auxiliary binary	*Corresponds* to positions $i < j$
1	call-return unary	Associated with matched pair $\langle x_i, x_j \rangle$
0	call-return leaf	Associated with matched pair $\langle x_i, x_j \rangle$, $j = i + 1$
0	internal leaf	Associated with internal letter x_i

We denote the set of types by Types; the degree of a node is, of course, the number of its children. Note that auxiliary binary nodes are not associated with any letters in the nested word, although they do correspond to pairs of positions in it.

In general, run the left-to-right DFS traversal on the tree $\mathsf{bin}(u)$ and *spell* the letters associated with the nodes in the natural way. Specifically, at any call-return node v *associated* with $i < j$, spell "$\langle x_i$" when entering and "$x_j\rangle$" when leaving the subtree rooted at v; at any internal leaf associated with i, spell "x_i". It is easy to see that the traversal of the entire tree $\mathsf{bin}(u)$ spells the word u, and every subtree spells some well-matched factor.

Claim 1. *For any nested word u of length ℓ its binary tree representation $\mathsf{bin}(u)$ has at most $2\ell - 1$ nodes. Moreover, if $\mathsf{bin}(u) = \mathsf{bin}(u')$, then $u = u'$.*

Trees as terms over a ranked alphabet. We now switch the perspective a little and look at binary tree representations as terms. Indeed, pick the ranked alphabet

$$\mathcal{F} \subseteq \mathsf{Types} \times (\langle \Sigma \times \Sigma \rangle \cup \Sigma \cup \{\varepsilon\}) \qquad (1)$$

as follows. All elements of $\mathcal{F}$ have *rank* 0, 1, or 2, according to their first (that is, Types-) component; the rank is simply the admissible number of children (degree). The second component stores the associated letter or pair of letters, if any; the value ε corresponds to the undefined association mapping. Since the Types-component already determines whether the second component should carry a pair of call and return letters, a single letter, or ε, we only take valid combinations into $\mathcal{F}$.

As this term representation is essentially the same as the binary representation defined above, we shall denote it by the same symbol $\mathsf{bin}(u)$; that is, $\mathsf{bin}(u)$ is a term over $\mathcal{F}$ for any non-empty well-matched word u. In what follows, we will mostly refer to $\mathsf{bin}(u)$ as a tree but treat it as a term.

4.2 From nested word automata to tree automata

From runs of NWA to runs of tree automata. Recall the definition of a *nondeterministic tree automaton* over a ranked alphabet $\mathcal{F}$ (see, e.g., [4]): such an automaton is a tuple $\mathcal{T} = (\mathcal{Q}, \mathcal{Q}^{\mathsf{f}}, \Delta)$ where $\mathcal{Q}$ is a finite set of states, $\mathcal{Q}^{\mathsf{f}} \subseteq \mathcal{Q}$ is a set of final states, and Δ is a set of transition rules. These rules have the form $f(q_1, \ldots, q_r) \mapsto q$ where $q, q_1, \ldots, q_r \in \mathcal{Q}$ and $r \geq 0$ is the rank of the symbol $f \in \mathcal{F}$; nondeterminism of $\mathcal{T}$ means that Δ can contain several rules with identical left-hand sides.

The semantics of tree automata is defined in the following manner. For any tree t over the ranked alphabet $\mathcal{F}$, we assign to any node v of t a state $q \in \mathcal{Q}$ inductively, phrasing it as "the subtree t_v rooted at v *evaluates* to state q" (as the automaton is nondeterministic, the same subtree may evaluate to several different states). The inductive assertion is that if f is the label of v, the subtree t_v evaluates to q, and its principal subtrees evaluate to $q_1, \ldots, q_r$, then the transition $f(q_1, \ldots, q_r) \mapsto q$ appears in Δ. The entire tree t is *accepted* if the root of t evaluates to some final state $\bar{q} \in \mathcal{Q}^{\mathsf{f}}$.

Lemma 7. *For any NWA $\mathcal{A}$ with states Q and for all pairs $\bar{p}, \bar{q} \in Q$, there exists a tree automaton $\mathcal{T}(\bar{p}, \bar{q})$ over the ranked alphabet $\mathcal{F}$ as in* (3) *that has the following property: $\mathcal{T}(\bar{p}, \bar{q})$ accepts a tree $\mathsf{bin}(u)$ if and only if the NWA $\mathcal{A}$ has a run on u that starts in state $\bar{p}$ and ends in state $\bar{q}$. Moreover, $\mathcal{T}(\bar{p}, \bar{q})$ can be constructed from $\mathcal{A}$ in time polynomial in the size of $\mathcal{A}$.*

The proof can be found in Appendix.

Synchronizing words and implicitly presented tree automata. We can now return to the synchronizing word problem. Suppose $\mathcal{A}$ is an NWA with states Q; now a well-matched nested word u is a synchronizing word for $\mathcal{A}$ if and only if there is a state $\bar{q} \in Q$ such that for all i the tree $\mathsf{bin}(u)$ is accepted by the automaton $\mathcal{T}(q_i, \bar{q})$; here we assume $Q = \{q_1, \ldots, q_n\}$. The following statement rephrases this condition in terms of *products* of tree automata (the definition is standard; see, e.g., [4, Section 1.3]).

Lemma 8. *An NWA $\mathcal{A}$ with states $Q = \{q_1, \ldots, q_n\}$ has a synchronizing word of length at most ℓ iff there exists a state $\bar{q} \in Q$ such that the product automaton $\mathfrak{A}_{\bar{q}} = \mathcal{T}(q_1, \bar{q}) \times \ldots \times \mathcal{T}(q_n, \bar{q}) \times \mathcal{N}_\ell$ accepts some tree over $\mathcal{F}$. Here $\mathcal{N}_\ell$ is a tree automaton that only depends on ℓ and Σ and accepts the set of trees of the form $\mathsf{bin}(u)$ where the nested word u has length length at most ℓ.*

Note that the set of states of $\mathfrak{A}_{\bar{q}}$, which we denote by $\mathfrak{Q}$, is, in general, exponential in the size of $\mathcal{A}$. Note, however, that (i) each state has a representation—as a tuple of n states of $\mathcal{T}(q_i, \bar{q})$ and a state of $\mathcal{N}_\ell$—polynomial in the size of $\mathcal{A}$ and ℓ and, moreover, that (ii) the following problems can be decided in **PSPACE** (and, in fact, in **P**, although we do not need to rely on this):

(a) given a state $\mathfrak{q} \in \mathfrak{Q}$, decide if $\mathfrak{q}$ is a final state of $\mathfrak{A}_{\bar{q}}$;
(b) given a symbol $f \in \mathcal{F}$ of rank r and states $\mathfrak{q}, \mathfrak{q}_1, \ldots, \mathfrak{q}_r \in \mathfrak{Q}$, decide if $f(\mathfrak{q}_1, \ldots, \mathfrak{q}_r) \mapsto \mathfrak{q}$ is a transition in $\mathfrak{A}_{\bar{q}}$.

We emphasize that the complexity bounds in these properties are given with respect to the size of $\mathcal{A}$ and ℓ, i.e., assuming that $\mathcal{A}$ and ℓ (and not $\mathfrak{A}_{\bar{q}}$!) are given as input. We will use these properties (i) and (ii) in the subsection 4.4; for brevity, we shall simply say that $\mathfrak{A}_{\bar{q}}$ is *implicitly presented in polynomial space.*

Claim 2. *The automaton $\mathfrak{A}_{\bar{q}}$ from Lemma 8 is implicitly presented in polynomial space and does not accept any tree with more than $2\ell - 1$ nodes.*

The second part of the claim follows from Claim 3 in subsection 4.1.

4.3 Pebble games and Strahler numbers

In this subsection we recall a classic idea that we use in the proof of Lemma 9 in the following subsection 4.4. We believe that the involved concepts, albeit classic, deserve more attention from our community than they have hitherto received.

An instance of the (*black*) *pebble game* (see, e.g., [20, Chapter 10]) is defined on a directed acyclic graph, G. The game is one-player; the player sees the graph G and has access to a supply of *pebbles.* A *strategy* in the game is a sequence of moves of the following kinds:

(a) if all immediate predecessors of a vertex v have pebbles on them, put a pebble on v;
(b) remove a pebble from a vertex of v.

The game starts with no pebbles on vertices of the graph; note that for any source v of G, the pre-condition for the move of the first kind is always satisfied. The strategy is *successful* if during its execution every sink of G carries a pebble at least once; the strategy is said to *use* k pebbles if the largest number of pebbles on G during its execution is k. The (*black*) *pebbling number* of G, denoted $\mathsf{peb}(G)$, is the smallest k for which there exists a successful strategy for G using k pebbles.

The black pebbling number captures space complexity of deterministic computations. Intuitively, think of G as a circuit, where sources are circuit inputs and sinks are circuit outputs; nodes with nonzero fan-in are gates that compute functions of their immediate predecessors. A strategy corresponds to computing the value of the circuit using auxiliary memory: *pebbling* a vertex (i.e., putting a pebble on it) corresponds to computing the value of the gate and storing it in memory; removing a pebble from the vertex corresponds to removing it from the memory. The pebbling number is thus (an abstraction of) the minimal amount of memory required to compute the value of the circuit.

Consider the case where the graph is a tree, $G = t$, with all edges directed towards the root; this corresponds to formulas, say arithmetic expressions. For trees, the pebbling number can be computed inductively: if t is a single-vertex tree, then $\mathsf{peb}(G) = 1$; suppose t has principal subtrees $t_1, \ldots, t_d$ and $\mathsf{peb}(t_1) \geq \mathsf{peb}(t_2) \geq \ldots \geq \mathsf{peb}(t_d)$, then $\mathsf{peb}(t) = \max(\mathsf{peb}(t_i) + i - 1)$ over $1 \leq i \leq d$. For binary trees (where all vertices have fan-in at most two, $d \leq 2$) the pebbling number (under different names) has been studied independently and rediscovered multiple times (although, to the best of our knowledge, no connection with the literature on pebbling games has ever been pointed out), see [13, 10]. The value $\mathsf{peb}(t) - 1$ is usually called the *Strahler number* of the tree t and is also known, e.g., as the Horton–Strahler number and as tree dimension; this is the maximal h such that t has the complete binary tree of height h as a minor.

In the sequel, we choose to talk about Strahler numbers but use the connection to pebble games. The key observation, following from the last characterization or from the recurrence above, is that the Strahler number of an m-node tree does not exceed $\lfloor \log_2(m+1) \rfloor - 1$, and this bound is tight. It corresponds to the pebbling strategy that, before pebbling any vertex v of indegree 2, first (*i*) recurses into the subtree with the larger Strahler number; (*ii*) places (inductively) a pebble on its root and removes all other pebbles from this subtree; and then (*iii*) recurses into the other subtree. We will use this strategy in the following subsection.

4.4 Bounded nonemptiness for implicitly presented tree automata

Here we combine the ideas from subsections 4.2 and 4.3 to prove Theorem 6.

Lemma 9. *For a tree automaton implicitly presented in polynomial space and a number m written in binary, one can decide in* **PSPACE** *if the automaton accepts some tree with at most m nodes.*

It is crucial that m constitute part of the input, because for *explicitly* presented tree automata the (non-)emptiness problem is **P**-complete, and an implicitly presented automaton can be exponentially big (this would give us an **EXP** upper bound). The upper bound on the size of the tree significantly shrinks the search space, so we refer to this problem as *bounded nonemptiness.* Assuming this lemma, the proof of Theorem 6 goes as follows.

Proof (of Theorem 6). Combine Lemma 8 and 9 with the fact that the automaton $\mathfrak{A}_{\bar{q}}$ from the former is implicitly presented in polynomial space. Indeed, suppose an NWA $\mathcal{A}$ with states Q and an integer ℓ are given. By Lemma 8, a synchronizing word for $\mathcal{A}$ of length at most ℓ exists if and only if there exists a state $\bar{q} \in Q$ such that the tree automaton $\mathfrak{A}_{\bar{q}}$ accepts some tree over the ranked alphabet $\mathcal{F}$; recall that this is the alphabet defined by (3) in subsection 4.1. First note that the state $\bar{q}$ can be guessed in polynomial space. Then recall from Claim 2 in subsection 4.2 that $\mathfrak{A}_{\bar{q}}$ only accepts trees with at most $2\ell - 1$ nodes; thus deciding its emptiness reduces to deciding its *bounded emptiness.* Again by Claim 2, $\mathfrak{A}_{\bar{q}}$ is implicitly presented in polynomial space, and thus we can apply Lemma 9 with $m = 2\ell - 1$. This concludes the proof. □

To prove Lemma 9, we design a decision procedure using the pebbling strategy for trees that we discussed in subsection 4.3.

Proof (of Lemma 9). Denote the tree automaton implicitly presented in polynomial space by $\mathfrak{A}_{\bar{q}}$, as above. We describe a procedure that guesses (with checks done on the fly) an accepting computation of $\mathfrak{A}_{\bar{q}}$. Since the number m is given in binary, we cannot afford to write down the entire accepted tree, as it could take up exponential space.

However, suppose that such a tree t exists and has $m' \leq m$ nodes; we assume without loss of generality that $m = m'$. Consider some pebbling strategy for t, as defined in subsection 4.3. Our procedure will guess moves of this strategy on the fly and simulate them; it will also guess the tree t in lockstep. More precisely, we maintain the following invariant. Take any time step and any vertex v and denote by t_v the subtree of t rooted at v. If the pebbling strategy prescribes that v should have a pebble, then our procedure keeps in memory a pair (q, k) where $\mathsf{q} \in \mathfrak{Q}$ is a state of $\mathfrak{A}_{\bar{q}}$ that t_v evaluates to, and k is the total number of nodes in t_v. Note that any such pair (q, k) takes up space polynomial in the size of the input: states of $\mathfrak{A}_{\bar{q}}$ have such representations by the assumptions of the lemma, and k never needs to grow higher than m.

We now describe how the moves of the strategy are simulated by our procedure. Suppose the strategy prescribes placing a pebble on a vertex v; by the rules of the pebble game, this means that all immediate predecessors $v_1, \ldots, v_d$ (if any) currently have pebbles on them. By our invariant, we already keep in memory corresponding pairs $(\mathsf{q}_1, k_1), \ldots, (\mathsf{q}_d, k_d)$. Our procedure now guesses the node v, i.e., its label $f \in \mathcal{F}$ in t. Then the procedure guesses a new state, $\mathsf{q} \in \mathfrak{Q}$, verifies in polynomial space that $f(\mathsf{q}_1, \ldots, \mathsf{q}_d) \mapsto \mathsf{q}$ is a transition in $\mathfrak{A}_{\bar{q}}$, and that $k = k_1 + \ldots + k_d + 1$ does not exceed m. If any check is failed, the procedure declares the current nondeterministic branch rejecting; if all the checks are passed, the procedure stores the pair (q, k). Naturally, whenever a strategy prescribes removing a pebble from a vertex, the procedure simply erases the corresponding pebble from the memory (in fact, since t is a tree, we can assume that every pair (q, k) is removed immediately after its use). At some point, the procedure guesses that the strategy can terminate; this means that the root of the tree t carries a pebble. The procedure picks some pair (q, k) from the memory and verifies in polynomial space that the state q is indeed final in $\mathfrak{A}_{\bar{q}}$. This signifies acceptance of t_v.

It remains to argue that the procedure only uses polynomial space. Since the tree t has m nodes, the upper bound on Strahler numbers tells us that the optimal strategy needs $\mathsf{peb}(t) \leq \lfloor \log_2(m + 1) \rfloor$ pebbles, which is polynomial in the size of the input. If some guessed step requires more, the strategy cannot be optimal, and the procedure declares the branch rejecting. This completes the proof. $\square$

The idea of the proof of Lemma 9 can be distilled in a different form: We can show that the *bounded emptiness* problem (are all trees up to a certain size rejected?) is in **PSPACE** for *succinct tree automata.* These are tree automata where the set of states, $\mathfrak{Q}$, can be exponentially large, but does not need to be written out explicitly, and the set of transitions and the set of final states are represented with Boolean circuits (or, alternatively, with logical formulas over an appropriate theory). The proofs follows that of Lemma 9.

5 Lower bound of Theorem 6

The matching lower bound for the Short Synchronizing Nested Word problem is established by a reduction from the *cost-optimal synchronizing problem*, that we introduce and provide the **PSPACE**-completeness.

5.1 Cost-optimal synchronizing problem in DFAs.

For the standard definition of DFAs, see the appendix. For a DFA $\mathcal{D} = \langle Q, \Sigma, \Delta \rangle$, a price function $\mathsf{price} : \Sigma \to \mathbb{N}$ assigns a price to each letter $a \in \Sigma$. The function is naturally extended to finite words: $\mathsf{price}(w \cdot a) = \mathsf{price}(w) + \mathsf{price}(a)$ where $w \in \Sigma^*$ and $a \in \Sigma$. The *cost-optimal synchronizing problem* asks, given a DFA equipped with a price function and given a $\mathsf{cost} \in \mathbb{N}$ in binary, whether the DFA has a synchronizing word w with price at most the estimated cost, i.e., $\mathsf{price}(w) \leq \mathsf{cost}$.

The cost-optimal synchronizing problem is solved in **(N)PSPACE** by guessing a synchronizing word w with length $|w| \leq 2^{|Q|}$ such that $\mathsf{price}(w) \leq \mathsf{cost}$. The cost-optimal synchronizing problem is **PSPACE**-complete by a reduction from the *carefully synchronizing* problem. The carefully synchronizing words generalizes the synchronizing words to finite-state automata with a partially defined transition function.

Theorem 10. *The cost-optimal synchronizing problem is* **PSPACE***-complete.*

Theorem 10 strengthens the **PSPACE**-hardness result provided in [14, 5], where the studied model is weighted automata with possibly negative weights on transitions (the proof in there rely on negative weights).

5.2 Reduction from cost-optimal synchronization to Short Synchronizing Nested Word

We prove the **PSPACE**-hardness of Short Synchronizing Nested Word by the following reduction: given a DFA $\mathcal{D} = \langle Q, \Sigma, \Delta \rangle$ equipped with function price and cost, we construct a NWA $\mathcal{A}$ and a length ℓ such that $\mathcal{D}$ has a synchronizing word w with $\mathsf{price}(w) \leq \mathsf{cost}$ if and only if $\mathcal{A}$ has a synchronizing nested word with length at most ℓ.

Below, we present the construction of $\mathcal{A}$ and ℓ and we prove the correctness of the reduction in Lemma 11. The intuition behind the reduction is to encode the price of letters a in $\mathcal{D}$ by the length of some particular well-matched nested words $a \cdot w_a$ in $\mathcal{A}$. To ensure that such a-transition is well-simulated, meaning that $\mathcal{A}$ is not *cheating* by reading a different word $w \neq a.w_a$, a *punishment* is considered. When $\mathcal{A}$ is punished, it is forced to read a long nested word w_{punish} which results in exceeding the length ℓ.

We use two types of gadgets $\mathsf{pay}_{q,a}$ (to simulate a with $a \cdot w_a$) and punish_q (to punish when $\mathcal{A}$ cheats) where $q \in Q$ and $a \in \Sigma$. By a slight abuse of notation, let $\mathsf{pay}_{q,a}$ and punish_q denote the states set of those gadgets as well. The set of states in $\mathcal{A}$ is $Q \cup \{\mathsf{frc}\} \cup \bigcup_{q \in Q, a \in \Sigma} (\mathsf{pay}_{q,a} \cup \mathsf{punish}_q \cup \{p_q, t_{q,a}\})$.

states	Σ	$\#$	$\mathsf{call}(\gamma)$	$\mathsf{ret}(\Gamma)$
$q \in Q$	$t_{q,a}$	p_q	$\gamma = \mathsf{x}$ self-loop	self-loop
frc	self-loop	p_q for some q	$\gamma = \mathsf{x}$ self-loop	self-loop

For all $q \in Q$ and $a \in \Sigma$

$t_{q,a}$	self-loop	p_q	$\gamma = \pounds$ in of $\mathsf{pay}_{q,a}$	self-loop
p_q	self-loop	p_q	$\gamma =$ ☹ in of punish_q	self-loop
$s \in \mathsf{punish}_q$	self-loop	p_q	see Figure 9 where • cheating transitions go to state in of itself; • from out, the transition $\mathsf{ret}($☹$)$ goes to q.	
$s \in \mathsf{pay}_{q,a}$	self-loop	p_q	see Figure 8 where • cheating transitions go to state err of itself; • from out, the transition $\mathsf{ret}(\pounds)$ goes to $\Delta(q,a)$; • from err, the transition $\mathsf{ret}(\pounds)$ goes to p_q.	

Table 1. Summary of the transition function δ of the NWA $\mathcal{A}$ constructed from the DFA $\mathcal{D} = \langle Q, \Sigma, \Delta \rangle$, where $\Gamma = \{\mathsf{x}, \mathsf{y}, \pounds,$ ☹$\}$. Here, the successor of states and letters for all transitions is stated, for example, $q \xrightarrow{\mathsf{call}(\mathsf{x})} q$ is the entry under $q \in Q$ and $\mathsf{call}(\gamma)$.

The set of stack symbols is $\Gamma = \{\mathsf{x}, \mathsf{y}, \pounds,$ ☹$\}$; the letters are $\Sigma \cup \{\#\}$ where $\#$ is a new letter. For all letters, the call and return transitions on state q and stack symbols γ are the same, thus we use $q \xrightarrow{\mathsf{call}(\gamma)} p$ if $\delta^{\mathsf{call}}(q,a) = (p,\gamma)$; and $q \xrightarrow{\mathsf{ret}(\gamma)} p$ if $\delta^{\mathsf{ret}}(\gamma,q,a) = p$ where $a \in \Sigma$. We provide Table 1 for a complete description of the transitions in $\mathcal{A}$, which we explain intuitively in following.

The gadgets $\mathsf{pay}_{q,a}$ and punish_q have similarity to Example 2:

Gadget $\mathsf{pay}_{q,a}$: it has three distinguished local states in, out and err; see Figure 8 in the appendix. All runs enter the gadget via an ingoing $\mathsf{call}(\pounds)$ transition to state in and would leave the gadget via states out or err by an outgoing $\mathsf{ret}(\pounds)$ transition. All such runs that finally leave the gadget are thus over well-matched words $w \in \mathsf{call}(\pounds) \cdot v \cdot \mathsf{ret}(\pounds)$. A run is *successful* if it leaves the gadget from state out; see that there is only a single word w_a with a successful run. We design the gadget such that $|a \cdot w_a| = \mathsf{price}(a)$. The only ingoing transition that enters $\mathsf{pay}_{q,a}$ is $t_{q,a} \xrightarrow{\mathsf{call}(\pounds)} \mathsf{in}$ where in is local for the gadget. The state $t_{q,a}$ stands for inputting a in the state q of the DFA; in fact, the construction is such that if a is input while synchronizing $\mathcal{D}$, then $\mathcal{A}$ cannot avoid $a \cdot w_a$, and thus the length of synchronizing word in $\mathcal{A}$ is added by $|a \cdot w_a|$. After reading $a \cdot w_a$ successfully, the gadget is left to the successor state $\Delta(q,a)$ of the DFA, thus the a-transition in DFA is simulated. The unsuccessful runs leave $\mathsf{pay}_{q,a}$ from err; in this cases, to show that the simulator has cheated, the state p_q is reached where the simulator would be punished by going through the gadget punish_q.

Gadget punish_q : it has two distinguished local states in and out; see Figure 9 in the appendix. As the main role, the gadget punish_q is used to punish $\mathcal{A}$ if it cheats while simulating $\mathcal{D}$. Similar to $\mathsf{pay}_{q,a}$, all runs which at some point leave the gadget are over well-matched words $w \in \mathsf{call}($☹$) \cdot v \cdot \mathsf{ret}($☹$)$. However, the

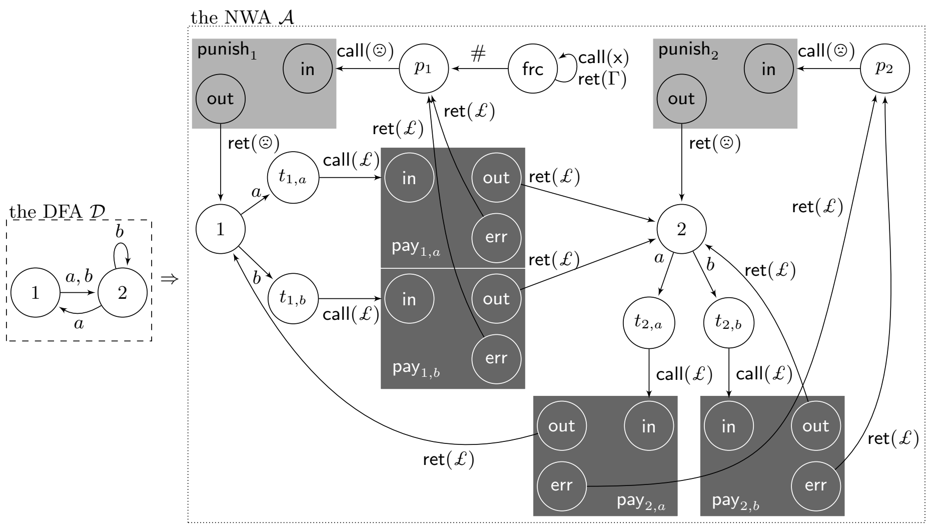

Figure 3. An example to illustrate the reduction from the cost-optimal synchronizing problem to the Short Synchronizing Nested Word. The DFA $\mathcal{D}$ has a cost-optimal synchronizing word, such as word b, if and only if $\mathsf{cost} \geq \mathsf{price}(b)$. For states $q = \{1, 2\}$ of the NWA $\mathcal{A}$, the #-transitions in q, all states of gadget punish_q and all states of gadgets $\mathsf{pay}_{q,a}, \mathsf{pay}_{q,b}$ lead to the state p_q. Moreover, all a-transitions and b-transitions in all states are self-loops, except in states $1, 2$. See Table 1 for the complete description of the transition function. Figures 8 and 9 in the appendix depicts the gadget $\mathsf{pay}_{q,a}$ and punish_q. The NWA $\mathcal{A}$ has a synchronizing nested word with length at most $\mathsf{cost} + |w_{\mathsf{punish}}| + 1$ if and only if $\mathcal{D}$ has a synchronizing word with price at most cost.

length of such words w_{punish} is very long (relative to the bound ℓ) in a way that the gadget can be visited, at most once. Hence, if $\mathcal{A}$ cheats and is punished, it cannot avoid inputting the long word w_{punish} while synchronizing. As the secondary role, we benefit from the fact that the gadget punish_q can be visited once, to shrink the whole states set of $\mathcal{A}$ to the subset Q, nominating copies of sates in $\mathcal{D}$. Entering simultaneously to all gadgets punish_q (where $q \in Q$) is then possible by a special letter # followed by an ingoing call(☹) transition; all simultaneous runs leave the gadgets punish_q by an outgoing ret(☹) transitions to states q where $q \in Q$. Thus the second role can be achieved by # that is forced to be read at least once while synchronizing $\mathcal{A}$, otherwise the *force state* frc would never be synchronized. The state frc has self-loops transitions for all transitions except for the letter #, where it goes to p_q for some $q \in Q$.

The states p_q stands for *punish time* where the punished run ends in state q: $p_q \xrightarrow{w_{\mathsf{punish}}} q$. For all $q \in Q$ and letters $a \in \Sigma$, we have $q \xrightarrow{a} t_{q,a}$ and $t_{q,a} \xrightarrow{w_a} \Delta(q, a)$. Let $\ell = 1 + |w_{\mathsf{punish}}| + \mathsf{cost}$. See Figure 3 for an example of the reduction.

Lemma 11. *The* Short Synchronizing Nested Word *is* **PSPACE**-*hard.*

Acknowledgements. The authors are grateful to Michael Wehar for comments.

References

1. Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.
2. Didier Caucal. Synchronization of pushdown automata. In *Developments in Language Theory, 10th International Conference, DLT 2006, Santa Barbara, CA, USA, June 26-29, 2006, Proceedings*, pages 120–132, 2006.
3. Michal Chytil and Burkhard Monien. Caterpillars and context-free languages. In *STACS 90, 7th Annual Symposium on Theoretical Aspects of Computer Science, Rouen, France, February 22-24, 1990, Proceedings*, pages 70–81, 1990.
4. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.
5. L. Doyen, L. Juhl, K. G. Larsen, N. Markey, and M. Shirmohammadi. Synchronizing words for weighted and timed automata. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 121–132, 2014.
6. Evan Driscoll, Aditya V. Thakur, and Thomas W. Reps. Opennwa: A nested-word automaton library. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 665–671, 2012.
7. David Eppstein. Reset sequences for monotonic automata. *SIAM J. Comput.*, 19(3):500–510, 1990.
8. Javier Esparza, Pierre Ganty, Stefan Kiefer, and Michael Luttenberger. Parikh's theorem: A simple and direct automaton construction. *Inf. Process. Lett.*, 111(12):614–619, 2011.
9. Javier Esparza, Pierre Ganty, and Rupak Majumdar. Parameterized verification of asynchronous shared-memory systems. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 124–140, 2013.
10. Javier Esparza, Michael Luttenberger, and Maximilian Schlund. A brief history of Strahler numbers. In *Language and Automata Theory and Applications - 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, pages 1–13, 2014.
11. Fedor M. Fominykh, Pavel V. Martyugin, and Mikhail V. Volkov. P(l)aying for synchronization. *Int. J. Found. Comput. Sci.*, 24(6):765–780, 2013.
12. Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266, 1977.
13. Thomas Lengauer and Robert Endre Tarjan. The space complexity of pebble games on trees. *Inf. Process. Lett.*, 10(4/5):184–188, 1980.
14. Pavel Martyugin. Computational complexity of certain problems related to carefully synchronizing words for partial automata and directing words for nondeterministic automata. *Theory Comput. Syst.*, 54(2):293–304, 2014.
15. Kurt Mehlhorn. Pebbling moutain ranges and its application of DCFL-recognition. In *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, pages 422–435, 1980.
16. Jörg Olschewski and Michael Ummels. The complexity of finding reset words in finite automata. In *Mathematical Foundations of Computer Science 2010, 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*, pages 568–579, 2010.

17. Jean-Eric Pin. On two combinatorial problems arising from automata theory. *North-Holland Mathematics Studies*, 75:535–548, 1983.
18. I. K. Rystsov. Polynomial complete problems in automata theory. *Inf. Process. Lett.*, 16(3):147–151, 1983.
19. Sven Sandberg. Homing and synchronizing sequences. In *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]*, pages 5–33, 2004.
20. John E. Savage. *Models of computation - exploring the power of computing.* Addison-Wesley, 1998.
21. Ján Černý, Alica Pirická, and Blanka Rosenauerová. On directable automata. *Kybernetika*, 07(4):(289)–298, 1971.
22. M. V. Volkov. Synchronizing automata and the cerny conjecture. In *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.
23. Michael Wehar. Hardness results for intersection non-emptiness. In *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, pages 354–362, 2014.

A Proof of Theorem 3

Suppose an NWA $\mathcal{A}$ has a synchronizing word. Then for every pair $p, q \in Q$, where Q is the set of states of $\mathcal{A}$, there exists a well-matched word that is *accepted* by the product automaton $\mathcal{A}(p) \times \mathcal{A}(q) \times \mathcal{W}$, where by $\mathcal{A}(p)$ and $\mathcal{A}(q)$ we denote disjoint copies of $\mathcal{A}$ with initial states p and q respectively, and $\mathcal{W}$ is a fixed NWA that only accepts well-matched words. As usual, a word is said to be accepted by an automaton if it brings it to some accepting state (where the set of such states is defined in advance). In this product automaton, accepting are all states of the form $(r, r, \bar{q})$ where $r \in Q$ is arbitrary and $\bar{q}$ is accepting in $\mathcal{W}$. Every synchronizing word is obviously accepted by the product automaton; moreover, since this automaton has polynomial size, translating it to a context-free grammar shows that the automaton accepts at least one word, say $w_{p,q}$, of at most exponential size (in terms of the size of $\mathcal{A}$).

Now observe that the task of synchronzing runs of $\mathcal{A}$ that start in different states $q_1, \ldots, q_n \in Q$ can be performed in steps: place n tokens on states of $\mathcal{A}$, pick any pair of them, say on p and q, and feed the machine with $w_{p,q}$. Now this pair of tokens is glued together; all tokens move some new locations, but their number is now $n - 1$. Repeating the procedure another $n - 2$ times then gives a synchronizing word for $\mathcal{A}$.

It only remains to note that the argument above also gives a sufficient condition for the existence of synchronizing words: indeed, if all product automata $\mathcal{A}(p) \times \mathcal{A}(q) \times \mathcal{W}$ have nonempty languages, then a synchronizing word can be constructed as described above, otherwise no such word can exist. Since emptiness for NWA is decidable in polynomial time, the theorem follows.

B Synchronization from subsets into subsets

In this section, we prove Theorem 4: let $\mathcal{A} = (Q, \Gamma, \delta, q_0, Q^{\mathsf{f}}, \gamma^{\mathsf{i}})$ be an NWA, the following problems are **EXP**-complete.

(1) Given a subset $I \subseteq Q$, decide if there exists a well-matched nested word u such that $(I, \bot) \xrightarrow{u} (\bar{q}, \bot)$ for some state $\bar{q} \in Q$. We may refer to this problem by *synchronizing a subset* problem.
(2) Given a subset $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(Q, \bot) \xrightarrow{u} (F', \bot)$ for some subset $F' \subseteq F$. We may refer to this problem by *synchronizing to a subset* problem.
(3) Given subsets $I \subseteq Q$ and $F \subseteq Q$, decide if there exists a well-matched nested word u such that $(I, \bot) \xrightarrow{u} (F', \bot)$ for some subset $F' \subseteq F$. We may refer to this problem by *synchronizing a subset to another subset* problem.

Observe that the synchronizing to a subset problem is a special case of the synchronizing a subset to another subset problem when $I = Q$. Since the upper and lower complexity bounds are matching, it suffices to provide the lower bound for the synchronizing to a subset problem, and the upper bound for the synchronizing a subset to another subset problem.

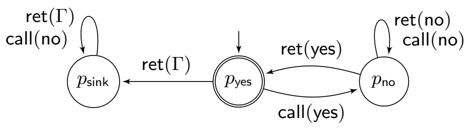

Figure 4. The NWA $\mathcal{A}^{\mathsf{wm}}$ used in the reductions to obtain **EXP**-hardness of synchronization from subsets into subsets. Since on all letters, call and return transitions behave the same, call transitions $\delta^{\mathsf{call}}(q,a) = (q',\gamma)$ is shown as $q \xrightarrow{\mathsf{call}(\gamma)} q'$, and return transitions $\delta^{\mathsf{ret}}(\gamma,q,a) = q'$ is shown as $q \xrightarrow{\mathsf{ret}(\gamma)} q'$. All not-drawn transitions are self-loops.

Establishing EXP-hardness.

We present reductions from the intersection nonemptiness problem for NWA, that is given NWA $\mathcal{A}_1,\ldots,\mathcal{A}_m$, decide if there exists a well-matched nested word accepted by all $\mathcal{A}_i$.

Let $\mathcal{A}^{\mathsf{wm}}$ be an NWA on the same alphabet of NWA $\mathcal{A}_1,\ldots,\mathcal{A}_m$. It has three states p_{yes}, p_{no} and p_{sink}. The state p_{yes} is initial and final state, and the state p_{sink} is absorbing. There are two stack symbols $\Gamma = \{\mathsf{yes},\mathsf{no}\}$; see Figure 4. The transitions in $\mathcal{A}^{\mathsf{wm}}$ are such that, for all $a \in \Sigma$,

– all internal transitions are self-loops: $\delta^{\mathsf{int}}(p,a) = p$ where $p \in \{p_{\mathsf{yes}},p_{\mathsf{no}}\}$,
– the call transition in p_{yes} and p_{no} always lead to p_{no}:

$$\delta^{\mathsf{call}}(p_{\mathsf{yes}},a) = (p_{\mathsf{no}},\mathsf{yes}) \text{ and } \delta^{\mathsf{call}}(p_{\mathsf{yes}},a) = (p_{\mathsf{no}},\mathsf{no}),$$

– the return transitions in p_{yes} go to p_{sink}: $\delta^{\mathsf{ret}}(\Gamma,p_0,a) = p_{\mathsf{sink}}$. However, the return transitions in p_{no} depends on the stack symbol:

$$\delta^{\mathsf{ret}}(\mathsf{no},p_{\mathsf{no}},a) = p_{\mathsf{no}} \text{ and } \delta^{\mathsf{ret}}(\mathsf{yes},p_{\mathsf{no}},a) = p_{\mathsf{yes}}.$$

Observe that $\mathcal{A}^{\mathsf{wm}}$ accepts all well-matched nested words; moreover, all runs, which start from the initial state p_{yes} and end in the final state p_0, are over well-matched nested words too.

Below when establishing reductions from the intersection nonemptiness problem for NWA, without loss of generality, we assume that $\mathcal{A}_m = \mathcal{A}^{\mathsf{wm}}$.

Synchronizing a subset problem. Let $\mathcal{A}_1,\ldots,\mathcal{A}_m$ be m NWA over the same alphabet Σ. We construct $\bar{\mathcal{A}}$ and set I such that there exists a well-matched nested word accepted by all $\mathcal{A}_i$ if and only if exists some well-matched nested word u and some state $\bar{q}$ in $\bar{\mathcal{A}}$ where $(I,\perp) \xrightarrow{u} (\bar{q},\perp)$.

The construction is as follows; see Figure 5. Let $q_1,\cdots,q_m$ be the initial states and $Q^{\mathsf{f}}_1,\cdots,Q^{\mathsf{f}}_m$ be the accepting sets for m NWAs. Let $\bar{\mathcal{A}}$ be the NWA that has one copy of each NWA $\mathcal{A}_i$ $(1 \le i \le m)$ and two new absorbing states sync and sink. For a new internal letter $\#$, let $\#$-transitions in all accepting

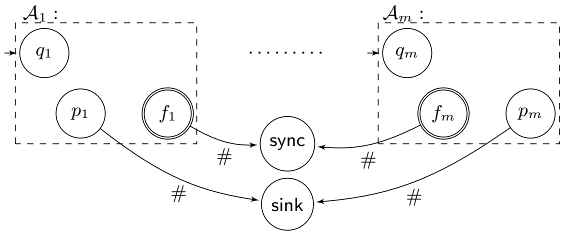

Figure 5. The sketch of the reduction from the intersection nonemptiness problem to the synchronizing a subset problem in NWAs.

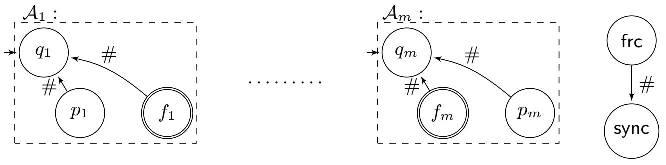

Figure 6. The sketch of the reduction from the intersection nonemptiness problem to the synchronizing to a subset problem in NWAs. All not-drawn transitions in sync and frc are self-loops.

states $q \in Q^{\mathsf{f}}_1 \cup \cdots \cup Q^{\mathsf{f}}_m$ of all $\mathcal{A}_i$ go to state sync whereas all #-transitions in all non-accepting states $q \notin Q^{\mathsf{f}}_1 \cup \cdots \cup Q^{\mathsf{f}}_m$ go to state sink. From the subset $I = \{q_1, \cdots, q_m, \mathsf{sync}\}$, a synchronizing word cannot escape sync because sync is an absorbing state. Let u be the shortest synchronizing word from I. We show that u always ends with #: (1) The only way that runs starting in the initial states $q_1, \cdots, q_n$ are synchronized in sync is inputting # at the right time when those runs are simultaneously in some accepting states in each copy. (2) Moreover, thanks to the fact that $\mathcal{A}_m = \mathcal{A}^{\mathsf{wm}}$, we know that all runs of $\mathcal{A}^{\mathsf{wm}}$ from q_m to Q^{f}_m are over well-matched nested words. Thus, there exits a well-matched nested word v such that $u = v \cdot \#$. One can verify that v is an accepting word for all $\mathcal{A}_i$. The **EXP**-hardness result follows.

Synchronizing to a subset problem. Let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be m NWA over the same alphabet Σ. We construct $\bar{\mathcal{A}}$ and set F such that there exists a well-matched nested word accepted by all $\mathcal{A}_i$ if and only if exists some well-matched nested word u and some state $\bar{q}$ in $\bar{\mathcal{A}}$ where $(Q, \bot) \xrightarrow{u} (F', \bot)$ for some subset $F' \subseteq F$.

The construction is as follows; see Figure 6. Let $q_1, \cdots, q_m$ be the initial states and $Q^{\mathsf{f}}_1, \cdots, Q^{\mathsf{f}}_m$ be the accepting sets for m NWAs. Let $\bar{\mathcal{A}}$ be the NWA that has one copy of each NWA $\mathcal{A}_i$ ($1 \leq i \leq m$) and two new states frc and sync. For a new internal letter #, let #-transitions in all states of $\mathcal{A}_i$ goes to q_i. The state sync is absorbing whereas the state frc has self-loops for all letters except $\#$. The $\#$-transitions in frc leads to sync. Let $F = Q_1^{\mathsf{f}} \cup \cdots \cup Q_m^{\mathsf{f}} \cup \{\mathsf{sync}\}$. A well-matched synchronizing word u for $\bar{\mathcal{A}}$ cannot escape sync because sync is an absorbing state. Let u be the shortest synchronizing word to the set F, thus u must have one occurrence of $\#$; otherwise state frc cannot be synchronized to F. Let $u = w \cdot \# \cdot v$ be such that v has no $\#$. As soon as the last $\#$ of u is read, each NWA $\mathcal{A}_i$ is reset to its initial state q_i. To be synchronized in F, each NWA $\mathcal{A}_i$ from q_i must reach the final state Q_i^{f} by inputting the word v. Recall that $\mathcal{A}_m = \mathcal{A}^{\mathsf{wm}}$. If w is not well-matched, then v would have to matched the pending calls of w that leads A^{wm} to reach p_{sink}, a contradiction with the fact that u synchronizes $\bar{A}$ to F. Thus, u and v are both well-matched. It remains to observe that each $\mathcal{A}_i$ have an accepting run over the well-matched nested word v to its final states. The **EXP**-hardness result follows.

Membership in EXP.

Synchronizing a subset problem. We reduce this problem to the synchronizing a subset to another subset problem, which we discuss below. Let $\mathcal{A} = (Q, \Gamma, \delta, q_0, Q^{\mathsf{f}}, \gamma^{\mathsf{i}})$ be an NWA with n states, and let $I \subseteq Q$. To decide whether there exists a well-matched nested word u and some state $\bar{q} \in Q$ such that $(I, \perp) \xrightarrow{u} (\bar{q}, \perp)$, we make $|Q|$ queries, for each $q \in Q$, to the the synchronizing the subset I to the singleton $\{q\}$.

Synchronizing a subset to another subset problem. Given $\mathcal{A}$ with set Q of states, $I \subseteq Q$ and $F \subseteq Q$, we reduce this problem to the emptiness problem of a product NWA, possibly exponential in the size of $\mathcal{A}$. The reduction is simple: for each state $q \in I$, we introduce an NWA $\mathcal{A}_q$ which is a copy of $\mathcal{A}$ where the initial state is q and the final states are F. Consider the product automata $\bar{A}$ of all NWA A_q with $q \in Q$, an accepting word synchronizes the set I to F in the original NWA $\mathcal{A}$. The size of this product is $|Q|^{|I|}$ which is exponential, and the emptiness problem for the product automata is in **P** $(|Q|^{|I|})$. Hence, **EXP** upper bound follows.

C Binary tree representation of nested words

In this subsection we describe a representation of nested words with binary trees that we use in the sequel.

Nested words as trees of unbounded degree. Given a non-empty well-matched nested word $u = (x, \nu)$ of length ℓ over an alphabet Σ, we define the (essentially standard) tree representation of u as follows. Recall that the matching relation satisfies $\nu \subseteq \{1, \ldots, k\}^2$, as u is well-matched, and that $\nu(i,j)$ implies $i < j$. Moreover, whenever $\nu(i,j)$ and $\nu(i',j')$, it cannot be the case that $i < i' \leq j < j'$; this means that the segments $[i,j]$, $[i',j'] \subseteq [1,\ell]$ are either disjoint or contained in one another. Therefore, this property also holds for the binary relation

$$\nu \cup \{(i,i) \mid \text{there is no } j \text{ such that } \nu(i,j) \text{ or } \nu(j,i)\} \cup \{(0, \ell+1)\}. \qquad (2)$$

In other words, the set defined by (2) forms the node set of an ordered rooted tree:

- a node (i', j') is a (non-strict) descendant of (i, j) if and only if $[i', j'] \subseteq [i, j]$;
- if nodes $v_1 = (i_1, j_1)$ and $v_2 = (i_2, j_2)$ are siblings, then either $i_1 \le j_1 < i_2 \le j_2$, in which case v_1 is to the left of (comes before) v_2, or $i_2 \le j_2 < i_1 \le j_1$, and then v_2 is to the left of (comes before) v_1.

The root of the tree is the pair $(0, \ell + 1)$.

Now take any non-root node $v = (i, j)$ of this tree. If $i < j$, then the ith position in u is a call and the jth position a return, and we associate v with the matched pair of letters $\langle x_i, x_j \rangle$ where $x = x_1 \dots x_\ell$; we write $\mu(v) = \langle x_i, x_j \rangle$. Otherwise $i = j$ and the ith position in u is internal; in this case we associate v with the letter a_i and write $\mu(v) = x_i$. We perform this for all non-root nodes v; the obtained ordered rooted tree is denoted by $\mathsf{tree}(u)$, the *simple tree representation* of the nested word $u = (x, \nu)$. Let V be the set of all nodes of $\mathsf{tree}(u)$; by convention, in the sequel we treat the values of the partial *association mapping* $\mu \colon V \rightharpoonup \langle \Sigma \times \Sigma \rangle \cup \Sigma$ as part of the tree itself.

We would like to remark that our simple tree representation is very similar to the mapping that transforms so-called "hedge words" into trees [1, subsection 7.1]. In our case, however, positions of x matched by ν do not have to carry identical letters from Σ; moreover, we add a special node as the root of the tree. Note that, in general, nodes of $\mathsf{tree}(u)$ can have unbounded degree (number of children).

Nested words as binary trees. The next step in our construction is "binarization" of the trees. Based on $\mathsf{tree}(u)$, we construct a new binary tree as follows. For every node v in $\mathsf{tree}(u)$ that has more than two children, say $v_1, \dots, v_k$ with $k \ge 3$, replace the star formed by v and $v_1, \dots, v_k$ by any ordered binary tree with root v and leaves $v_1, \dots, v_k$ (preserving the left-to-right DFS traversal order) where all non-leaf nodes have exactly 2 children (the number of new "auxiliary" nodes will be $k - 2$, not including v). We do not insist on picking any particular shape of the k-leaf tree, because we do not need to rely on uniqueness of representation. Similarly, if the root of $\mathsf{tree}(u)$ has more than one child, we perform a similar transformation to make the root a unary node. After this we remove the root (recall that it was added artificially in the first place).

The newly obtained tree is binary; we denote it by $\mathsf{bin}(u)$ and call it the *tree representation* of u. We will not use in the sequel the simple tree representation $\mathsf{tree}(u)$ defined above. While the construction of $\mathsf{bin}(u)$ is not sophisticated, nodes of $\mathsf{bin}(u)$ come in many different *types*; for the reader's convenience, we present a summary. We would like to emphasize that we did not attempt to minimize the number of these types; different representations are, of course, also possible.

Degree	Type	Notes
2	call-return binary	Associated with matched pair $\langle x_i, x_j \rangle$
2	auxiliary binary	*Corresponds* to positions $i < j$
1	call-return unary	Associated with matched pair $\langle x_i, x_j \rangle$
0	call-return leaf	Associated with matched pair $\langle x_i, x_j \rangle$, $j = i + 1$
0	internal leaf	Associated with internal letter x_i

We denote the set of types by Types; the degree of a node is, of course, the number of its children. Note that $\mathsf{auxiliary\ binary}$ nodes are not associated with any letters in the nested word, although they do correspond to pairs of positions in it.

In general, run the left-to-right DFS traversal on the tree $\mathsf{bin}(u)$ and *spell* the letters associated with the nodes in the natural way. Specifically, at any $\mathsf{call\text{-}return}$ node v associated with $i < j$, spell "$\langle x_i$" when entering and "$x_j\rangle$" when leaving the subtree rooted at v; at any $\mathsf{internal\ leaf}$ associated with i, spell "x_i". It is easy to see that the traversal of the entire tree $\mathsf{bin}(u)$ spells the word u, and every subtree spells some well-matched factor.

Claim 3. *For any nested word u of length ℓ its binary tree representation $\mathsf{bin}(u)$ has at most $2\ell - 1$ nodes. Moreover, if $\mathsf{bin}(u) = \mathsf{bin}(u')$, then $u = u'$.*

Trees as terms over a ranked alphabet. We now switch the perspective a little and look at binary tree representations as terms. Indeed, pick the ranked alphabet

$$\mathcal{F} \subseteq \mathsf{Types} \times (\langle \Sigma \times \Sigma \rangle \cup \Sigma \cup \{\varepsilon\}) \tag{3}$$

as follows. All elements of $\mathcal{F}$ have *rank* 0, 1, or 2, according to their first (that is, Types-) component; the rank is simply the admissible number of children (degree). The second component stores the associated letter or pair of letters, if any; the value ε corresponds to the undefined association mapping. Since the Types-component already determines whether the second component should carry a pair of call and return letters, a single letter, or ε, we only take valid combinations into $\mathcal{F}$.

As this term representation is essentially the same as the binary representation defined above, we shall denote it by the same symbol $\mathsf{bin}(u)$; that is, $\mathsf{bin}(u)$ is a term over $\mathcal{F}$ for any non-empty well-matched word u. In what follows, we will mostly refer to $\mathsf{bin}(u)$ as a tree but treat it as a term.

D Proof of Lemma 7

For the reader's convenience, we restate this lemma from subsection 4.2, in the proof of the upper bound of Theorem 6. We show that for any NWA $\mathcal{A}$ with states Q and for all pairs $\bar{p}, \bar{q} \in Q$, there exists a tree automaton $\mathcal{T}(\bar{p}, \bar{q})$ over the ranked alphabet $\mathcal{F}$ as in (3) that has the following property: $\mathcal{T}(\bar{p}, \bar{q})$ accepts a tree $\mathsf{bin}(u)$ if and only if the NWA $\mathcal{A}$ has a run on u that starts in state $\bar{p}$ and ends in state $\bar{q}$. Moreover, $\mathcal{T}(\bar{p}, \bar{q})$ can be constructed from $\mathcal{A}$ in time polynomial in the size of $\mathcal{A}$.

The key idea is as follows. States of $\mathcal{T}(\bar{p}, \bar{q})$ will be *summaries*, $\mathcal{Q} = Q^2$, and subtrees will be evaluated to summaries (p, q) so that the following condition holds. Take any subtree t_v of the input tree t; as discussed in subsection 4.1, the left-to-right DFS traversal of this subtree spells a word which is a well-matched factor u' of u. The automaton $\mathcal{T}(\bar{p}, \bar{q})$ will pick the states p, q in such a way that the NWA $\mathcal{A}$ will start and finish traversing u' in the states p and q respectively.

Naturally, nondeterministic guessing will be required for this construction to work.

We now make the details of the construction more precise. Define a *labeling* of t with respect to $\mathcal{A}$ as a function of the form $\lambda\colon V(t) \to Q^2$ where $V(t)$ denotes the set of nodes of t (or, equivalently, the set of ranked symbols in the term representation of t). The labeling λ is *consistent* with the NWA $\mathcal{A}$ if the following conditions are satisfied for all nodes v of t (we assume $\lambda(v) = (p, q)$ and $\lambda(v_s) = (p_s, q_s)$ for $s = 1, 2$).

1. If v is a call-return binary node associated with a matched pair $\langle x_i, x_j \rangle$ and v_1 and v_2 are its left and right children, then there exists a $\gamma \in \Gamma$ such that $\delta^{\mathsf{call}}(p, x_i) = (p_1, \gamma)$, $q_1 = p_2$, and $\delta^{\mathsf{ret}}(\gamma, q_2, x_j) = q$.
2. If v is an auxiliary binary node and v_1 and v_2 are its left and right children, then $p = p_1$, $q_1 = p_2$, and $q_2 = q$.
3. If v is a call-return unary node associated with a matched pair $\langle x_i, x_j \rangle$ and v_1 is its only child, then there exists a $\gamma \in \Gamma$ such that $\delta^{\mathsf{call}}(p, x_i) = (p_1, \gamma)$ and $\delta^{\mathsf{ret}}(\gamma, q_1, x_j) = q$.
4. If v is a call-return leaf associated with a matched pair $\langle x_i, x_j \rangle$, $j = i + 1$, then there exists a $\gamma \in \Gamma$ and an $r \in Q$ such that $\delta^{\mathsf{call}}(p, x_i) = (r, \gamma)$ and $\delta^{\mathsf{ret}}(\gamma, r, x_j) = q$.
5. If v is an internal leaf associated with internal letter x_i, then $\delta^{\mathsf{int}}(p, x_i) = q$.

Suppose $\lambda(\mathsf{root}(t)) = (\bar{p}, \bar{q})$ where $\mathsf{root}(t)$ is the root of t. Start the NWA $\mathcal{A}$ in the state p_0 and run the left-to-right DFS traversal of t; whenever the traversal spells a letter, give it to $\mathcal{A}$ as input. Now λ is consistent with $\mathcal{A}$ if and only if for every non-root node v with $\lambda(v) = (p, q)$ the NWA begins the computation on the corresponding well-matched factor in state p and leaves it in state q. Therefore $\mathcal{A}$ has a computation on u that starts in state $\bar{p}$ and terminates in state $\bar{q}$ if and only if there exists a consistent labeling λ of t such that $\lambda(\mathsf{root}(t)) = (\bar{p}, \bar{q})$. What remains is an easy exercise: define the transitions of $\mathcal{T}(\bar{p}, \bar{q})$ in such a way that $\mathcal{T}(\bar{p}, \bar{q})$ guesses a consistent labeling of $\mathsf{bin}(u)$; the existence of an appropriate set Δ follows from our definition of a consistent labeling. This completes the proof of Lemma 7.

E Standard definition of DFAs and synchronizing words.

In this section, we recall the standard definition of words and DFAs. A *word* over a set Σ of letters is a sequence $w = a_1 \cdots a_n$ of letters, where its length is $|w| = n$. We denote by Σ^* the set of all finite words over the finite alphabet Σ.

A deterministic finite-state automaton (DFA) is a tuple $\mathcal{D} = \langle Q, \Sigma, \Delta \rangle$ where Q is a finite set of states, Σ is a finite alphabet and the transition function $\Delta : Q \times \Sigma \to Q$ is totally defined. The function Δ extends to finite words, in a natural way: $\Delta(q, wa) = \Delta(\Delta(q, w), a)$ for all words $w \in \Sigma^*$ and letters $a \in \Sigma$; and it extends to set of states by $\Delta(S, w) = \bigcup_{q \in S} \Delta(q, w)$ where $S \subseteq Q$.

A word w is *synchronizing* for the DFA $\mathcal{D}$ if there exists some state $\bar{q} \in Q$ such that $\Delta(Q, w) = \{\bar{q}\}$. The *synchronizing problem* in DFAs asks, given a

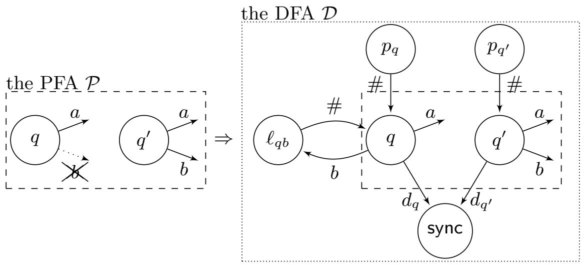

Figure 7. The sketch of the reduction from carefully synchronization to cost-optimal synchronization. All not-drawn transitions in $\mathcal{D}$ are self-loops.

DFA $\mathcal{D}$, whether there exists some synchronizing word for $\mathcal{D}$. It is known that $\mathcal{D}$ has a synchronizing word if and only if for all pairs of states $q, q' \in Q$, there exists a word v such that $\Delta(q, v) = \Delta(q', v)$ (see [22] for more details). The synchronizing problem in DFAs is thus in **NL** by a simple technique which we call *pairwise synchronization.* Given $\mathcal{D}$ with n states, set $S_n = Q$ and for all $i = n-1, \cdots, 1$ repeat the following: find a word v_i such that $\Delta(q, v_i) = \Delta(q', v_i)$ for any pair $q, q' \in S_{i+1}$ and let $S_i = \Delta(S_{i+1}, v)$. The word $w = v_{n-1} \cdots v_2 \cdot v_1$ is synchronizing for $\mathcal{D}$.

F Correctness of PSPACE-hardness reduction for the cost-optimal synchronizing problem

In this section we prove Theorem 10, showing that the cost-optimal synchronizing problem is **PSPACE**-complete. Membership in **PSPACE** is discussed in the main text and to establish the **PSPACE**-hardness, we present a reduction from carefully synchronizing problem.

The carefully synchronizing words generalizes the synchronizing words to finite-state automata with a partially defined transition function (PFAs).

A PFA is a tuple $\mathcal{P} = \langle Q, \Sigma, \delta \rangle$ where the transition function $\delta : Q \times \Sigma \to Q$ might not be defined for some states q and letters a. A synchronizing word $w = a_1 a_2 \cdots a_n$ for the PFA $\mathcal{P}$ must use only defined transitions, that means $\delta(q, a_1)$ is defined for all states $q \in Q$ and $\delta(q, a_{i+1})$ is defined for all $q \in \delta(Q, a_1 a_2 \cdots a_i)$ and all $1 \leq i < n$. The *carefully synchronizing problem* asks, given a PFA $\mathcal{P}$, whether there exists some synchronizing word for $\mathcal{P}$.

Reduction from carefully synchronizing problem. We prove that the cost-optimal synchronizing problem is **PSPACE**-hard, by the following reduction. Given a PFA $\mathcal{P} = \langle Q, \Sigma, \delta \rangle$, we construct a DFA $\mathcal{D}$ equipped with function price and cost such that $\mathcal{P}$ has a synchronizing word if and only if $\mathcal{D}$ has a synchronizing word w with $\mathsf{price}(w) \leq \mathsf{cost}$.

Below, we present the construction of $\mathcal{D}$ and we prove the correctness of the reduction in Lemma **??**. The sketch of reduction is depicted in Figure 7. The price associated to all $a \in \Sigma$ is $\mathsf{price}(a) = 0$. A new letter # is introduced with $\mathsf{price}(\#) = 2^{|Q|}$; and for all $q \in Q$, a new letter d_q is added with $\mathsf{price}(d_q) = 1$. Setting $\mathsf{cost} = 2^{|Q|}+1$ restricts $\mathcal{D}$ to input $\#$ at most once while synchronizing; moreover, if $\#$ is read then only one letter among $\{d_q \mid q \in Q\}$ can be chosen.

Let $\varDelta$ be the transition function of $\mathcal{D}$ where $\varDelta(q,a) = \delta(q,a)$ if $\delta(q,a)$ is defined for $q \in Q$ and $a \in \varSigma$. For all $q \in Q$, a new state p_q is added where all transitions are self-loops except $\#$-transition: $\varDelta(p_q, \#) = q$. For all $q \in Q$ and $a \in \varSigma$, if $\delta(q,a)$ is not defined, we then add a new state ℓ_{qa} where all transitions are self-loops except $\#$-transition: $\varDelta(\ell_{qa}, \#) = q$; we also define $\varDelta(q,a) = \ell_{qa}$. Hence, to synchronize the states p_q and ℓ_{qa} the letter $\#$ must be read at least once. It remains to define for all $q \neq q'$: $\varDelta(q,\#) = q$, $\varDelta(q,d_q) = \mathsf{sync}$ and $\varDelta(q,d_{q'}) = q$ where sync is a new state with no outgoing transitions. The automaton $\mathcal{D}$ can only be synchronized in sync, and since sync is reached only by some letter d_q, the automaton $\mathcal{D}$ must input d_q at least for one $q \in Q$.

To prove the correctness of the reduction, observe that if $\mathcal{P}$ has some synchronizing word v where $\delta(Q,v) = \{q\}$, then $\# \cdot v \cdot d_q$ is a synchronizing word for $\mathcal{D}$ with $\mathsf{price}(\# \cdot v \cdot d_q) = \mathsf{cost}$.

Now, assume that $\mathcal{D}$ has some synchronizing word with price at most cost; let w be the shortest such words. By construction, w must have exactly one occurrence of $\#$ and one occurrence of d_q for some state $q \in Q$. Since w is one of the shortest synchronizing word, thus $w = w_1 \cdot \# \cdot w_2 \cdot d_q$ where $w_1, w_2 \in \varSigma^+$. We prove that $w_2 = a_1 a_2 \cdots a_n$ is a valid synchronizing word for $\mathcal{P}$ by three observations:

- the set of reached states after inputting $\#$ is exactly Q.
- for all $1 \leq i \leq n$ and all states $q \in \varDelta(Q, a_1 a_2 \cdots a_i)$, the successor state is never ℓ_{qa_i}; otherwise since w_2 has no occurrence of $\#$, we get an immediate contradiction with the fact that w is a synchronizing word for $\mathcal{D}$. Thus, the automaton $\mathcal{D}$ only fires "defined" transitions of $\mathcal{P}$ while reading w_2.
- Since $\varDelta(q_1, d_q) \neq \varDelta(q_2, d_q)$ for all letters d_q and pairs of states $q_1 \neq q_2$, then $\varDelta(Q, w_2) = \delta(Q, w_2)$ is a singleton.

The **PSPACE**-hardness results follows.

A variant of the cost-optimal synchronization is studied, in [5], for weighted automata. In that model, the transitions of the automata are augmented with negative or positive weights. Their model can assigns different weights, even negative weights, to transitions on the same letter, thus their model generalizes ours where all transition on the same letter have the same price. However, the **PSPACE**-hardness result for the synchronizing problem with a bounded weight in their model heavily benefits from the freedom of choosing negative weights, and possibly different weights for the transitions on same letter. The obtained **PSPACE**-hardness result then cannot be used for cost-optimal synchronizing problem.

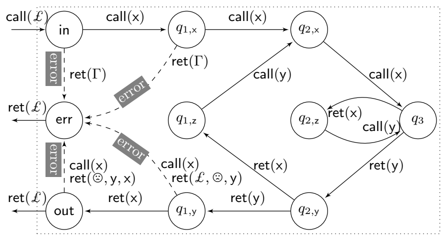

Figure 8. Gadget $\mathsf{pay}_{q,b}$ with a single well-matched word $w_a = \mathsf{call}(\pounds) \cdot v \cdot \mathsf{ret}(\pounds)$ where v has a run from in to out.

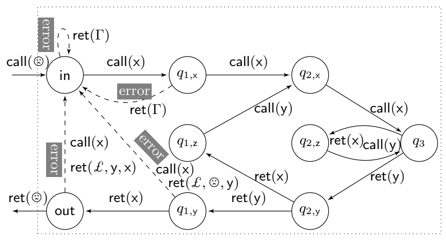

Figure 9. Gadget punish_q: the length of any word $\mathsf{call}(☹) \cdot v \cdot \mathsf{ret}(☹)$ that leave the gadget is at least 2^4.

G Correctness of PSPACE-hardness reduction for the Short Synchronizing Nested Word

In this section we prove Lemma 11 of the main text, showing that the Short Synchronizing Nested Word problem is **PSPACE**-hard. We present a reduction from the cost-optimal synchronization problem to the Short Synchronizing Nested Word in the main text, here, we provide the correctness.

To prove the correctness of the reduction, observe that if $\mathcal{D}$ has some synchronizing word $v = a_1 a_2 \cdots a_n$ with $\mathsf{price}(v) \leq \mathsf{cost}$, then

$$\# \cdot w_{\mathsf{punish}} \cdot a_1 \cdot w_{a_1} \cdot a_2 \cdot w_{a_2} \cdots a_n \cdot w_{a_n}$$

is a synchronizing word for $\mathcal{A}$ with length

$$|\# \cdot w_{\mathsf{punish}}| + \mathsf{price}(a_1) + \mathsf{price}(a_2) + \cdots + \mathsf{price}(a_n) \leq 1 + |w_{\mathsf{punish}}| + \mathsf{cost}.$$

For the converse direction, assume that $\mathcal{A}$ has some synchronizing word w with length at most ℓ. We prove that $\mathcal{D}$ has a synchronizing word with price at most cost.

We take a close look at the runs of $\mathcal{A}$ over $w = b_1 b_2 \cdots b_m$ starting only from the states $q \in Q$. Let $S_0 = Q$ and $S_i = \delta(S_{i-1}, b_i)$ for all $1 \leq i \leq m$. We first prove that $S_i \cap Q \neq \emptyset$ if and only if $S_i \subseteq Q$ for all $1 \leq i \leq m$. The proof is by an induction where the induction step ($S_0 \subseteq Q$) trivially holds.

For induction step (proving the statement for k), assume that for all $i < k$, we have $S_i \cap Q \neq \emptyset$ if and only if $S_i \subseteq Q$. For S_i and $j \in \mathbb{N}$, we call S_{i+j} the j-th successor of S_i; when $j = 1$, we simply call S_{i+1} the successor of S_i. Let $x < k$ be the biggest number such that $S_x \subseteq Q$. Since x is the biggest such numbers and since all transitions in $q \in Q$ are self-loops unless $\#$-transition and a-transitions where $a \in \Sigma$, thus there are two cases:

1. $S_{x+1} = \{p_q \mid q \in S_x\}$ implying that $b_{x+1} = \#$. Since all transitions in p_q are self-loops unless $\mathsf{call}(☹)$ transitions, all next successors of S_{x+1} are equal to itself until $\mathsf{call}(☹)$ is read, say at b_{y-1}, implying that $S_y =$

$\{\mathsf{in}$ of gadgets $\mathsf{punish}_q \mid q \in S_x\}$. If $x + 1 \le n < y$, then $S_n = S_{x+1}$ where $S_n \cap Q = \emptyset$ holds. For $y \le n$, consider the fact that all next successors of S_y consists only from the states of gadget punish_q unless either $\#$ is read again that cause the same case (the successor would be $\{p_q \mid q \in S_x\}$), or $\mathsf{ret}(☹)$ is read in the states out of all gadgets punish_q. Note that since all gadgets punish_q have the same constructions, thus all runs follow the same scenario and leave the gadget simultaneously. Let us say that $\mathsf{ret}(☹)$ is read such that $b_z = \mathsf{ret}(☹)$ and $S_{z-1} = \{\mathsf{out}$ of gadgets $\mathsf{punish}_q \mid q \in S_x\}$. If $y \le n < z$ then $S_n \cap Q = \emptyset$ holds. Otherwise, since by construction $S_{z-1} \xrightarrow{bz} S_x$, and since $x < n$ is the biggest number with $S_x \cap Q \neq \emptyset$, then $S_n = S_x \subseteq Q$.

Observe that in this case, $S_x = S_y$ for the two consecutive set S_x and S_y with non-empty intersection with Q. We refer to this by **no-$\mathsf{ret}(£)$-case**.

2. $S_{x+1} = \{t_{q,a} \mid q \in S_x, a = b_{x+1}\}$. Since all gadgets $\mathsf{pay}_{q,a}$ on the same letter a have the same construction; and since the only way to leave the gadgets $\mathsf{pay}_{q,a}$ are: (i) inputting $\mathsf{ret}(£)$ in the states err of the gadgets or inputting $\#$, where both lead to the successor $\{p_q \mid q \in S_x\}$, that is discuss in **no-$\mathsf{ret}(£)$-case**; (ii) inputting $\mathsf{ret}(£)$ in the states out of the gadgets. We discuss the latter in details. Let $b_y = \mathsf{ret}(£)$ and $S_{y-1} = \{\mathsf{out}$ of gadgets $\mathsf{pay}_{a,q} \mid q \in S_x, a = b_{x+1}\}$. By construction $S_y \subseteq Q$. If $x+1 \le n < y$ then $S_n \cap Q = \emptyset$ holds; otherwise since $x < n$ is the biggest number with $S_x \cap Q \neq \emptyset$, then $n = y$ and $S_n = \Delta(S_x, b_{x+1}) \subseteq Q$.

Observe that in this case, inputting $\mathsf{ret}(£)$ in states out of the gadget $\mathsf{pay}_{q,a}$ plays a key role to have $S_y = \Delta(S_x, b_{x+1})$ for the two consecutive set S_x and S_y with non-empty intersection with Q. We refer to this by **$\mathsf{ret}(£)$-case**.

The induction step is complete, and we then have that $S_i \cap Q \neq \emptyset$ if and only if $S_i \subseteq Q$ for all $1 \le i \le m$. We also have the following immediate result: either $S_i = S_j$ or $S_j = \Delta(S_i, b_{i+1})$ for all consecutive $S_i, S_j \subseteq Q$ (meaning that $1 \le i < j \le m$ and $S_k \cap Q = \emptyset$ for all $i < k < j$).

Let n be the size of $\{1 \le i \le m \mid S_i \subseteq Q\}$, the number of S_i with $S_i \subseteq Q$. We define a strictly increasing mapping $\mathsf{id} : \{1, \cdots, n\} \to \{1, \cdots, m\}$ such that $S_{\mathsf{id}(j)} \subseteq Q$. The sequence $S_{\mathsf{id}(1)} S_{\mathsf{id}(2)} \cdots S_{\mathsf{id}(n)}$ consists of all the successors of Q with non-empty intersection with Q. We have shown that either $S_{\mathsf{id}(i)} = S_{\mathsf{id}(i+1)}$ or $\Delta(S_{\mathsf{id}(i)}, b_{\mathsf{id}(i)+1}) = S_{\mathsf{id}(i+1)}$ where $1 \le i < n$. Let $T_1 T_2 \cdots T_k$ be the biggest subsequence of $S_{\mathsf{id}(1)} S_{\mathsf{id}(2)} \cdots S_{\mathsf{id}(n)}$, and $\mathsf{index} : \{1, \cdots, k\} \to \{1, \cdots, n\}$ be a strictly increasing mapping such that $T_j = S_{\mathsf{id}(\mathsf{index}(j))}$ and $\Delta(T_j, b_{\mathsf{id}(\mathsf{index}(j))+1}) = T_{j+1}$ (all **$\mathsf{ret}(£)$-cases**). Let $v = a_1 \cdots a_k$ be the word consisting of all $b_{\mathsf{id}(\mathsf{index}(j))+1}$ for $1 \le j \le k$. Since v is constructed from **$\mathsf{ret}(£)$-cases**, then $v \in \Sigma^*$.

We prove that v is a synchronizing word for $\mathcal{D}$ with $\mathsf{price}(v) \le \mathsf{cost}$. First, see that since $v = a_1 \cdots a_k$ is constructed from **$\mathsf{ret}(£)$-cases**, then $\Delta(T_j, a_j) = T_{j+1}$ for all $1 \le j < k$, as a result, the a_j-transitions of $\mathcal{D}$ is well-simulated. It remains to prove that T_k is a singleton. Towards contradiction, assume that there are

$q \neq q'$ such that $\{q, q'\} \subseteq T_k$. Let x be the biggest number $x \leq m$ where $S_x \subseteq Q$, we have shown that $T_k = S_x$, giving that $\{q, q'\} \subseteq S_x$. By a similar argument to **no-$\mathsf{ret}(\pounds)$-case**, the two states $\{q, q'\}$ would not be synchronized by $b_{x+1} b_{x+2} \cdots b_m$, a contradiction with the fact that w is a synchronizing word for $\mathcal{A}$. Then, T_k is a singleton and v is a synchronizing word for $\mathcal{D}$.

To complete the proof, we provide that $\mathsf{price}(v) \leq \mathsf{cost}$. We know by construction that w have the subword $\# \cdot w_{\mathsf{punish}}$ exactly once. Let w' be the subword of w after omitting $\# \cdot w_{\mathsf{punish}}$, thus $|w'| \leq \mathsf{cost}$. On the other hand, by construction of $v = a_1 \cdots a_k$ (from w'), we know that all letters a_i of v are correct simulations of an a_i-transition in $\mathcal{D}$, thus each a_i is followed by w_{a_i} where $|a_i \cdot w_{a_i}| = \mathsf{price}(a_i)$. We thus have $\mathsf{price}(v) < \mathsf{cost}$.