Low-Latency Sliding Window Algorithms for Formal Languages

Moses Ganardi ✉ 🄳
Max Planck Institute for Software Systems, Kaiserslautern, Germany

Louis Jachiet ✉ 🄳
LTCI, Télécom Paris, Institut Polytechnique de Paris, France

Markus Lohrey ✉ 🄳
Universität Siegen, Germany

Thomas Schwentick ✉ 🄳
TU Dortmund University, Germany

───── **Abstract** ─────

Low-latency sliding window algorithms for regular and context-free languages are studied, where latency refers to the worst-case time spent for a single window update or query. For every regular language L it is shown that there exists a constant-latency solution that supports adding and removing symbols independently on both ends of the window (the so-called two-way variable-size model). We prove that this result extends to all visibly pushdown languages. For deterministic 1-counter languages we present a $\mathcal{O}(\log n)$ latency sliding window algorithm for the two-way variable-size model where n refers to the window size. We complement these results with a conditional lower bound: there exists a fixed real-time deterministic context-free language L such that, assuming the OMV (online matrix vector multiplication) conjecture, there is no sliding window algorithm for L with latency $n^{1/2-\epsilon}$ for any $\epsilon > 0$, even in the most restricted sliding window model (one-way fixed-size model). The above mentioned results all refer to the unit-cost RAM model with logarithmic word size. For regular languages we also present a refined picture using word sizes $\mathcal{O}(1)$, $\mathcal{O}(\log \log n)$, and $\mathcal{O}(\log n)$.

1 Introduction

In this paper, we investigate sliding window algorithms for formal languages. In the basic sliding window model, an infinite stream $s = a_1 a_2 a_3 \ldots$ of symbols from a finite alphabet Σ is read symbol by symbol from left to right. It works *one-way* and with a *fixed window size n*. The *window content* is the suffix of length n of the prefix of the stream s seen so far. Thus, in each step, a new right-most symbol is read into the window and the left-most symbol is moved out of the window. A sliding window algorithm for a language $L \subseteq \Sigma^*$ has to indicate at every time instant whether the current window content belongs to L (initially the window is filled with some dummy symbol). The two resources that one typically tries to minimize are memory and the worst-case time spent per incoming symbol. It is important to note that the model only has access to the letter currently read. In particular, if the algorithm wants to know the precise content of the current window or, in particular, which letter moves out of the window, it has to dedicate memory for it. For general background on sliding window algorithms see [1, 9].

We refer to the above sliding window model as the *one-way, fixed-size model*. A more general variant is the one-way, *variable-size* sliding window model. In this model the arrival of new symbols and the expiration of old symbols are handled independently, i.e., there are update operations that add a new right-most symbol and an operation that removes the left-most symbol. Therefore the size of the window can grow and shrink. This allows to model for instance a time-based window that contains all data values that have arrived in the last t seconds for some fixed t. If the arrival times are arbitrary then the window size may vary. The *two-way model* is a further generalization whose update operations allow to add and remove symbols on both sides of the window. It can be combined with both the fixed-size and the variable-size model. The variable-size two-way model is the most general model; it is also known as a deque (double-ended queue); see [25, Section 2.2.1]. An algorithm also needs to handle query operations, asking whether the current window content is in the language L.

There are two important complexity measures for a sliding window algorithm: its *space complexity* and its *latency* (or *time complexity*), i.e. the time required for a single update or query operation. Both are usually expressed depending on the window size n (for a fixed-size sliding window algorithm) or the maximal window size n that occurs during a run of the sliding window algorithm (for a variable-size sliding window algorithm). In this paper we are mainly interested in the *latency* of sliding window algorithms. Since it turns out that space complexity is an important tool for proving lower bounds on the latency of sliding window algorithms, we first discuss known results on space complexity.

Space complexity of sliding window algorithms.   The space complexity of sliding window algorithms for formal languages in the one-way (fixed-size and variable-size) model has been studied in [15, 16, 17, 19, 21] and in [18, Section 9] for the two-way variable-size model. For regular languages, the main result of [17] is a space trichotomy for the one-way case: the space complexity of a regular language is either constant, logarithmic or linear. This result holds for the fixed-size model as well as the variable-size model, although the respective language classes differ slightly. For the two-way variable-size model a space trichotomy has been shown in [18]. Table 1 summarizes some of the main results of [16, 17, 18] in more detail (ignore the word size bound for the moment). The results on the two-way fixed-size model in Table 1 are shown in this paper. In that table,

- Reg denotes the class of all regular languages;
- Len denotes the class of *regular length languages*, i.e., regular languages $L \subseteq \Sigma^*$, for which either $\Sigma^n \subseteq L$ or $\Sigma^n \cap L = \emptyset$, for every n;
- LI denotes the class of all *regular left ideals*, i.e., regular languages of the form $\Sigma^* L$ for a regular language L;
- SL denotes the class of *suffix languages*[1], i.e., languages of the form $\Sigma^* w$;
- Triv denotes class of trivial languages, i.e., the class consisting of $\emptyset$ and Σ^* only;
- Fin denotes the class of finite languages;
- $\langle A_1, \ldots, A_n \rangle$ denotes the Boolean closure of $\bigcup_{1 \leq i \leq n} A_i$.

Note that these classes are defined with respect to an alphabet, e.g. a^* is trivial if the alphabet is $\{a\}$ but non-trivial if the alphabet is $\{a, b\}$. In Table 1, we write $f \in \bar{\Theta}(g)$ for $f, g : \mathbb{N} \to \mathbb{R}$ iff $f \in \mathcal{O}(g)$ and there is a constant $c > 0$ with $f(n) \geq c \cdot g(n)$ for infinitely many n.

---

[1]   In [16, 17], we used instead of SL the boolean closure of SL (the so-called suffix testable language); but this makes no difference, since we are only interested in the boolean closures of language classes.

■ **Table 1** Summary of results for regular languages and constant latency. The columns correspond to the 4 different sliding window models (1F = one-way fixed-size, 1V = one-way variable-size, 2F = two-way fixed-size, 2V = two-way variable-size). The rows correspond to different combinations of word size and space in bits. For all three combinations the latency is $\mathcal{O}(1)$. Note that the language classes in each column yield a partition of Reg.

	1F	1V	2F	2V
word size: $\mathcal{O}(1)$ space in bits: $\mathcal{O}(1)$	$\langle\mathsf{Len},\mathsf{SL}\rangle$	Triv	$\langle\mathsf{Len},\mathsf{Fin}\rangle$	Triv
word size: $\bar{\Theta}(\log\log n)$ space in bits: $\bar{\Theta}(\log n)$	$\langle\mathsf{Len},\mathsf{LI}\rangle \setminus \langle\mathsf{Len},\mathsf{SL}\rangle$	$\langle\mathsf{Len},\mathsf{LI}\rangle \setminus \mathsf{Triv}$	$\emptyset$	$\mathsf{Len} \setminus \mathsf{Triv}$
word size: $\bar{\Theta}(\log n)$ space in bits: $\bar{\Theta}(n)$	$\mathsf{Reg} \setminus \langle\mathsf{Len},\mathsf{LI}\rangle$	$\mathsf{Reg} \setminus \langle\mathsf{Len},\mathsf{LI}\rangle$	$\mathsf{Reg} \setminus \langle\mathsf{Len},\mathsf{Fin}\rangle$	$\mathsf{Reg} \setminus \mathsf{Len}$

Some of the results from [16, 17] were extended to (subclasses of) context-free languages in [15, 21]. A space trichotomy was shown for visibly pushdown languages in [15], whereas for the class of all deterministic context-free languages the space trichotomy fails [21].

Content of the paper. In this paper we consider the latency of sliding window algorithms for regular and deterministic context-free languages in all four of the above models: one-way and two-way, fixed-size and variable-size. These models are formally defined in Section 2. As the algorithmic model, we use the standard RAM model. The word size (register length) is a parameter in this model and we allow it to depend on the fixed window size n (in the fixed-size model) or the maximal window size n that has occurred in the past (for the variable-size model). More precisely, depending on the language class, the word size can be $\mathcal{O}(1)$ (resulting in the *bit-cost model*), $\mathcal{O}(\log\log n)$, or $\mathcal{O}(\log n)$. We assume the unit-cost measure, charging a cost of 1 for each basic register operation.

The bit-cost model serves as a link to transfer lower bounds: it is a simple observation, formalized in Lemma 1, that the sliding window time complexity for a language L in the bit-cost model is at least the logarithm of the minimum possible space complexity. In fact, this lower bound holds even with respect to the non-uniform bit-probe model, where we have a separate algorithm for each window size n. And, again by Lemma 1, lower bounds on the latency in the bit-cost model translate to lower bounds on the word size for unit-cost algorithms with constant latency.

In Section 3, we study the latency of sliding window algorithms for regular languages and offer a complete picture. Our contribution here is mainly of algorithmic nature, since most of the lower bounds are simple consequences of space lower bounds, that were shown in [16, 17, 18]. The main result of the first part is that these lower bounds can be achieved by concrete algorithms. More precisely, there are algorithms that (1) achieve the optimal latency with respect to the bit-cost model and (2) constant latency with respect to unit-cost model, and (3) also have optimal space complexity. The precise results are summarized in Table 1 for the unit-cost model. In all cases, the sliding window algorithms have constant latency. For example, languages from $\langle\mathsf{Len},\mathsf{LI}\rangle$ have one-way sliding window algorithms with constant latency on unit-cost RAMs with word size $\mathcal{O}(\log\log n)$ and thus $\mathcal{O}(\log\log n)$ latency in the bit-cost model. These algorithms have space complexity $\mathcal{O}(\log n)$. Moreover, unless the language belongs to $\langle\mathsf{Len},\mathsf{LI}\rangle$ (for the fixed-size model) or Triv (for the variable-size model)

these resource bounds cannot be improved. Note that, while for three of the four models there is a trichotomy, the two-way fixed-size model is an outlier: it has a dichotomy, since there are no languages of intermediate complexity.

In Section 4 we consider the latency for deterministic context-free languages (DCFL) and here the study is more of an explorative nature. Since every DCFL has a linear time parsing algorithm [24], one might hope to get also a low-latency sliding window algorithm. Our first result tempers this hope: assuming the OMV (online matrix vector multiplication) conjecture [23], we show that there exists a fixed real-time DCFL L such that no algorithm can solve the sliding window problem for L on a RAM with logarithmic word size with latency $n^{1/2-\epsilon}$ for any $\epsilon > 0$, even in the one-way fixed-size model (the most restricted model). This motivates to look for subclasses that allow more efficient sliding window algorithms. We present two results in this direction. We show that for every visibly pushdown language [2] there is a two-way variable-size sliding window algorithm on a unit-cost RAM with word size $\mathcal{O}(\log n)$ and constant latency. Visibly pushdown languages are widely used, e.g. for describing tree-structured documents and traces of recursive programs. They share many of the nice algorithmic and closure properties of regular languages. Finally, we show that for every deterministic one-counter language there is a two-way variable-size sliding window algorithm on a unit-cost RAM with word size $\mathcal{O}(\log n)$ and latency $\mathcal{O}(\log n)$.

Related work. The latency of regular languages in the sliding window model has been first studied in [28], where it was shown that in the one-way, fixed-size model, every regular language has a constant latency algorithm on a RAM with word size $\log n$ (the result is not explicitly stated in [28] but directly follows by using the main result of [28] for the transformation monoid of an automaton). Our upper bound results for general regular languages rely on this work and we extend its techniques to visibly pushdown languages.

A sliding window algorithm can be viewed as a dynamic data structure that maintains a dynamic string w (the window content) under very restricted update operations. Dynamic membership problems for more general updates that allow to change the symbol at an arbitrary position have been studied in [3, 13, 14].

Standard streaming algorithms (where the whole history and not only the last n symbols is relevant) for visibly pushdown languages (and subclasses) were studied in [4, 5, 6, 10, 12, 26, 27]. These papers investigate the space complexity of streaming. Update times of streaming algorithms for timed automata have been studied in [22].

2 Sliding window model

Throughout this paper we use $\log n$ as an abbreviation for $\lceil \log_2 n \rceil$.

Consider a function $f : \Sigma^* \to C$ for some finite alphabet Σ and some countable set C. We will view the sliding window problem for the function f as a dynamic data structure problem, where we want to maintain a word $w \in \Sigma^*$, called the *window*, which undergoes changes and admits membership queries to L. Altogether, we consider the following operations on Σ^*, where for a word $w = a_1 \cdots a_n \in \Sigma^*$ we write $|w| = n$ for its length and $w[i : j] = a_i \cdots a_j$ for the factor from position i to position j (which is ε if $i > j$).

- rightpush(a): Replace w by wa.
- leftpush(a): Replace w by aw.
- leftpop():   Replace w by $w[2 : |w|]$ (which is ε if $w = \varepsilon$).
- rightpop(): Replace w by $w[1 : |w| - 1]$ (which again is ε if $w = \varepsilon$).
- query():   Return the value $f(w)$.

In most cases, the function f will be the characteristic function of a language $L \subseteq \Sigma^*$; in this case we speak of the sliding window problem for the language L.

In the *two-way model* all five operations are allowed, whereas in the *one-way model*, we only allow to add symbols on the right and to remove symbols on the left, that is, it allows only the operations rightpush(a), leftpop(), and query(). In the *variable-size window model* the operations can be applied in arbitrary order, but in the *fixed-size window model*, push operations always need to be followed directly by a pop operation on the other side. More formally, each rightpush(a) needs to be immediately followed by a leftpop() and (in the two-way model), each leftpush(a) needs to be immediately followed by a rightpop(). In particular, no query can occur between a leftpush(a) and the subsequent rightpop(). Therefore, as the name suggests, in the fixed-size model the string always has the same length n, for some n, if we consider a push and its successive pop operation as one operation.

In the variable-size model the window w is initially ε, whereas in the fixed-size model it is initialized as $w = \square^n$ for some default symbol $\square \in \Sigma$. We allow algorithms a preprocessing phase and disregard the time they spend during this initialization. In the fixed-size model the algorithm receives the window size n for its initialization.

We denote the four combinations of models by 1F, 1V, 2F, and 2V, where 1 and 2 refer to one-way and two-way, respectively, and F and V to fixed-size and variable-size respectively.

We use two different computational models to present our results, the uniform *word RAM model* for upper bounds and the non-uniform *cell probe model* for lower bounds.

Word RAM model.   We present algorithmic results (i.e., upper bounds) in the *word RAM model* with maximal word size (or register length) of $b(n)$ bits, for some *word size function* $b(n)$. Algorithms may also use registers of length smaller than $b(n)$, for a more fine-grained analysis. In the fixed-size model n is the fixed window size, whereas in the variable-size model n is the maximum window size that has appeared in the past. In particular, if the window size increases then also the allowed word size $b(n)$ increases, whereas a subsequent reduction of the window size does not decrease the allowed word size. As usual, all RAM-operations on registers of word size at most $b(n)$ take constant time (unit-cost assumption). For a model $M \in \{1F, 1V, 2F, 2V\}$, an M-algorithm (for a function f) is a sliding window algorithm that supports the operations of model M.

An M-algorithm $\mathcal{A}$ has *latency* (or *time complexity*) $T(n)$ if for all n the following hold:
- If $M \in \{1F, 2F\}$, then in every computation of window size n, all operations of model M are handled within $T(n)$ steps by $\mathcal{A}$.
- If $M \in \{1V, 2V\}$, then in every computation of maximal window size n, all operations of model M are handled within $T(n)$ steps by $\mathcal{A}$.

Space complexity is defined accordingly and refers to the number of bits used by the algorithm.

Cell probe model.   For lower bounds we use the *cell probe model*. We formalize the model only for sliding window algorithms. For a model $M \in \{1F, 1V, 2F, 2V\}$, an M-algorithm in the cell probe model is a collection $\mathcal{A} = (\mathcal{A}_n)_{n \geq 0}$, where $\mathcal{A}_n$ is an M-algorithm for window size n (if $M = 1F$ or $M = 2F$), respectively, maximal window size n (if $M = 1V$ or $M = 2V$). Furthermore, we only count the number of read/write accesses to memory cells and disregard computation completely. We also say that $\mathcal{A} = (\mathcal{A}_n)_{n \geq 0}$ is a *non-uniform M-algorithm*.

More formally, fix a word size function $b = b(n)$. In the cell probe model an M-algorithm $\mathcal{A}_n$ for (maximal) window length n is a collection of decision trees $t_{n,\mathsf{op}}$ for every operation op of model M. Each node of $t_{n,\mathsf{op}}$ is labelled with a register operation read(R_i) or write(R_i, u) where i is a register address and $u \in \{0,1\}^{b_i}$. Here, $b_i \leq b(n)$ denotes the size (in bits) of register i. A node labelled with read(R_i) has 2^{b_i} children, one for each possible value of register i. A node labelled with write(R_i, w) has exactly one child. Moreover, the leaves of

the decision tree for query() are labelled with output values (0 or 1). The latency $T^{\mathcal{A}}(n)$ of $\mathcal{A}$ is the maximal height of a decision tree $t_{n,\mathsf{op}}$. The space complexity $S^{\mathcal{A}}(n)$ of $\mathcal{A}$ is the sum over the bit lengths of the different registers referenced in all trees $t_{n,\mathsf{op}}$.

By minimizing for every n the number of bits used in the trees $t_{n,\mathsf{op}}$, it follows that for every language $L \subseteq \Sigma^*$ there is a (non-uniform) M-algorithm $\mathcal{B}$ with optimal space complexity $S^{\mathcal{B}}(n)$ for every n. We denote this optimal space complexity by $S_L^M(n)$; see also [16]. Since for every language $L \subseteq \Sigma^*$ there is a non-uniform M-algorithm that stores the window explicitly with $n \cdot \log |\Sigma|$ bits, it holds $S_L^M(n) \leq n \cdot \log |\Sigma|$.

Space complexity in the sliding window model was analyzed in [15, 16, 17, 19, 21] for the one-sided models (1F and 1V) and [18, Section 9] for the model 2V (the model 2F has not been studied so far). In these papers, the space complexity was defined slightly different but equivalent to our definition. Note that lower bounds (for space and time) that are proved for the cell probe model also hold for the (uniform) RAM model.

As mentioned before, we will use the non-uniform cell probe model only for lower bounds. Lower bounds on the space complexity of sliding window algorithms yield lower bounds on the latency, as well. In fact, all our (unconditional) lower bounds stem from space lower bounds with the help of the following lemma; shown in the long version [20]. We mainly apply space lower bounds from [16, 17, 18].

▶ **Lemma 1.** *For each model $M \in \{1\mathsf{F}, 1\mathsf{V}, 2\mathsf{F}, 2\mathsf{V}\}$ and each non-uniform M-algorithm $\mathcal{A}$ with word size $b(n)$ for some language L, it holds $b(n) \cdot T^{\mathcal{A}}(n) \geq \log S_L^M(n) - \mathcal{O}(1)$.*

3 Regular languages

As mentioned in the introduction, the class of regular languages satisfies a space trichotomy in the models 1F and 1V [16, 17]: a regular language either has space complexity $\bar{\Theta}(n)$ or $\bar{\Theta}(\log n)$ or $\mathcal{O}(1)$. In the light of Lemma 1, the best we can therefore hope for are constant latency sliding-window algorithms with word size $\mathcal{O}(\log n)$, $\mathcal{O}(\log \log n)$ and $\mathcal{O}(1)$, respectively. It turns out that such algorithms actually exist. In the following, we consider each of these three levels separately. We present algorithms and confirm their optimality by corresponding lower bounds.

Before we start, let us fix our (standard) notation for finite automata. A *deterministic finite automaton* (DFA) is a tuple $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is an alphabet, $q_0 \in Q$ is the initial state, $\delta \colon Q \times \Sigma \to Q$ is the transition function and $F \subseteq Q$ is the set of final states. The transition function δ is extended to a function $\delta \colon Q \times \Sigma^* \to Q$ in the usual way. The language accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

3.1 Logarithmic word size

For the upper bound, we show that regular languages have constant latency sliding-window algorithms with logarithmic word size and optimal space complexity in the two-way variable-size model and thus in all four models.

To this end, we start from a known 1V-algorithm for evaluating products over finite monoids and adapt it so that it also works for the two-way model and meets our optimality requirements. Recall that a *monoid* is a set $\mathcal{M}$ equipped with an associative binary operation on $\mathcal{M}$. Let $\mathsf{prod}_{\mathcal{M}} \colon \mathcal{M}^* \to \mathcal{M}$ be the function which maps a word over $\mathcal{M}$ to its product. There is a folklore simulation of a queue (aka. 1V-sliding window) by two stacks, which takes constant time per operation on *average*, see [28]. This idea can be turned into a

Figure 1 Left: A guardian at position p, storing all suffixes of $m_1 \ldots m_{p-1}$ and all prefixes of $m_p \ldots m_n$. Right: As soon as p escapes the range $[\frac{1}{4}n, \frac{3}{4}n]$ a new guardian (in green) is started at $\frac{n}{2}$.

1V-algorithm for $\mathsf{prod}_{\mathcal{M}}$, if $\mathcal{M}$ is a finite monoid, taking constant time on average and $\mathcal{O}(n)$ space. Tangwongsan, Hirzel, and Schneider presented a *worst-case* constant latency algorithm [28].

▶ **Theorem 2** (c.f. [28])**.** *Let $\mathcal{M}$ be a fixed finite monoid (it is not part of the input). Then there is a 1V-algorithm for $\mathsf{prod}_{\mathcal{M}}$ with word size $\mathcal{O}(\log n)$ and latency $\mathcal{O}(1)$.*

An immediate corollary of Theorem 2 is that every regular language L has a 1V-algorithm with word size $\mathcal{O}(\log n)$ and latency $\mathcal{O}(1)$. For this, one takes for the monoid $\mathcal{M}$ in Theorem 2 the transformation monoid of a DFA for L. The transformation monoid of a DFA with state set Q and transition function $\delta : Q \times \Sigma \to Q$ is the submonoid of Q^Q (the set of all mappings on Q) generated by the functions $q \mapsto \delta(q, a)$, where $a \in \Sigma$. The monoid operation is the composition of functions: for $f, g \in Q^Q$ we define $fg \in Q^Q$ by $(fg)(q) = g(f(q))$ for all $q \in Q$.

In order to obtain for every regular language a 2V-algorithm with word size $\mathcal{O}(\log n)$, latency $\mathcal{O}(1)$, and space complexity $\mathcal{O}(n)$, we strengthen Theorem 2:

▶ **Theorem 3.** *Let $\mathcal{M}$ be a fixed finite monoid. Then there is a 2V-algorithm for $\mathsf{prod}_{\mathcal{M}}$ with word size $\mathcal{O}(\log n)$, latency $\mathcal{O}(1)$, and space complexity $\mathcal{O}(n)$.*

Proof sketch. We only consider the case, that the maximal window size n is known at advance (i.e., during initialization). The general case, where the maximal window size n is not known, can be dealt with Lemma 18 from Appendix A. We outline an algorithm with word size $\mathcal{O}(\log n)$, latency $\mathcal{O}(1)$, and space complexity $\mathcal{O}(n)$.

The limit n allows us to pre-allocate a circular bit array of size $\mathcal{O}(n)$, in which the window content $w = m_1 m_2 \cdots m_\ell \in \mathcal{M}^*$ ($\ell \le n$) is stored in a circular fashion and updated in time $\mathcal{O}(1)$. The current window length ℓ is stored in a register of bit length $\log n$. A constant number of pointers into w, including pointers to the first and to the last entry are also stored. Furthermore, to keep track of the product $m_1 m_2 \cdots m_\ell$ under the allowed operations, some sub-products $m_i m_{i+1} \cdots m_j$ for $i < j$ have to be stored as well. Storing all such products would require quadratic space. Instead, the algorithm stores (again in a circular fashion) for some index $p \in [1, \ell]$, called the *guardian*, all products $m_i m_{i+1} \cdots m_p$ ($i \in [1, p-1]$) and $m_p m_{p+1} \cdots m_j$ ($j \in [p+1, \ell]$); see Figure 1 for an illustration. If these products are only stored for all $i \in [k, p-1]$, $j \in [p+1, \ell]$ for some k, ℓ, then we speak of a *partial guardian*. As long as the guardian p satisfies $1 < p < n$, a push operations just adds one product and a pop operation removes one. A $\mathsf{leftpop}()$ decreases p and a $\mathsf{leftpush}(a)$ increases it. However, the algorithm only works correctly as long as the guardian is strictly between 1 and n.

Therefore, we enforce the invariant $p \in [\frac{1}{8}\ell, \frac{7}{8}\ell]$. To guarantee this invariant, we start a new guardian p', initially set to $\frac{\ell}{2}$, whenever the old guardian p escapes the interval $[\frac{1}{4}\ell, \frac{3}{4}\ell]$. We need to make sure that the computation of the products for the new guardian is fast enough such that (i) p stays in $[\frac{1}{8}\ell', \frac{7}{8}\ell']$ while the guardian p' is partial, where ℓ' is the window size at that point, and (ii) p' stays in $[\frac{1}{4}\ell', \frac{3}{4}\ell']$. A simple calculation shows that it suffices if the guardian p' is complete after $\frac{1}{7}\ell$ steps where ℓ is the window size when p'

was initialized. Indeed, in worst case, the length of the string after $\frac{1}{7}\ell$ steps is $\frac{6}{7}\ell$. If the guardian is initially at position $\frac{1}{4}\ell$, it might afterwards be, again in worst case, at position $(\frac{1}{4} - \frac{1}{7})\ell = \frac{6}{56}\ell = \frac{1}{8} \times \frac{6}{7}\ell$. The dual case is analogous.

In each step (application of an operation), 8 new products are computed in a balanced[2] fashion, so that, after at most $\frac{\ell}{7}$ steps all products are available for the new guardian p'. In fact, if ℓ denotes the window size when the computation of the new guardian starts, after $\frac{\ell}{7}$ steps $\frac{8}{7}\ell$ products are computed, covering the potential window size after these steps.

The two guardians can be stored in registers of $\log n$ size and for the two collections of partial products $\mathcal{O}(n)$ bits suffice. Moreover, all update operations can be carried out within a constant number of steps. ◀

Using again the transformation monoid of a regular language we obtain from Theorem 3:

▶ **Corollary 4.** *Every regular language L has a* 2V*-algorithm with word size $\mathcal{O}(\log n)$, latency $\mathcal{O}(1)$ and space complexity $\mathcal{O}(n)$.*

The lower bounds for the $\mathcal{O}(\log n)$-time level can be summarized as follows; see [20] for the proof. Recall that non-uniform M-algorithms refer to the cell probe model from Section 2.

▶ **Theorem 5.** *For a regular language L and sliding-window model M, every non-uniform M-algorithm with word size 1 for L has latency at least $\log n - \mathcal{O}(1)$ for infinitely many n in each of the following three cases:*
(a) $M \in \{\mathsf{1F}, \mathsf{1V}\}$ *and* $L \notin \langle \mathsf{Len}, \mathsf{LI} \rangle$,
(b) $M = \mathsf{2V}$ *and* $L \notin \mathsf{Len} = \langle \mathsf{Len} \rangle$,
(c) $M = \mathsf{2F}$ *and* $L \notin \langle \mathsf{Len}, \mathsf{Fin} \rangle$.
In (b) and (c) the lower bound $\log n - \mathcal{O}(1)$ holds for all n.

▶ **Corollary 6.** *In each of the cases of Theorem 5, any M-algorithm with latency $\mathcal{O}(1)$ requires word size $\Omega(\log n)$.*

3.2 Sublogarithmic word size

There are two combinations of subclasses of regular languages and sliding window models, for which we obtain constant latency algorithms with word size $\mathcal{O}(\log \log n)$:

▶ **Theorem 7.** *Let L be a regular language and M a sliding-window model. If (i) $L \in \langle \mathsf{Len}, \mathsf{LI} \rangle$ and $M = \mathsf{1V}$ or (ii) $L \in \mathsf{Len}$ and $M = \mathsf{2V}$, then there exists an M-algorithm for L with the following properties, where n is the maximum window size:*
- *The algorithm uses a RAM with a bit array of length $\mathcal{O}(\log n)$ and a constant number of pointers into the bit array. Each pointer can be stored in a register of length $\mathcal{O}(\log \log n)$.*
- *The latency of the algorithm is $\mathcal{O}(1)$.*

A detailed proof of Theorem 7 can be found in Appendix B. For both cases it is known that a sliding window algorithm (1V in case (i), 2V in case (ii)) with logarithmic space complexity exists [16]. These algorithms mainly use counters of maximal size n (the window size) and every window update makes a constant number of the following counter operations: increments, decrements, comparison of two counter values (where one of the two counters is always the same), and reset to zero. One could store the counters in the standard binary representation, but then the operations on counters would take time $\mathcal{O}(\log n)$. To overcome

---

[2] In principle, four products are computed for each side, but if the word grows towards one side, this can be reflected in the choice of the next products.

this problem, we use the counting techniques from [14], that allows to do the following counter operations in time $\mathcal{O}(\log \log n)$ for a fixed maximal counter size n: increment, decrement and comparison with a fixed number. We have to adapt this technique, since we also have to reset counters and compare counters. Moreover, since we work in the variable-size model, we have no limit on the size of the counters.

The lower bounds for the $\log \log n$-time level follow again from known space lower bounds in the one-way model [16], see [20].

▶ **Theorem 8.** *For a regular language L and sliding-window model M, every non-uniform M-algorithm with word size 1 for L has latency at least $\log \log n - \mathcal{O}(1)$ for infinitely many n in each of the following two cases:*
(a) *$M = 1\mathsf{V}$ and L is not trivial,*
(b) *$M = 1\mathsf{F}$ and $L \notin \langle \mathsf{Len}, \mathsf{SL} \rangle$,*
In (a) the lower bound $\log n - \mathcal{O}(1)$ holds for all n.

▶ **Corollary 9.** *In each of the cases of Theorem 8, any M-algorithm with latency $\mathcal{O}(1)$ requires word size $\Omega(\log \log n)$.*

We finally turn to regular languages that have constant latency sliding window algorithms on a RAM with word size $\mathcal{O}(1)$. The proof of the following theorem can be found in [20].

▶ **Theorem 10.** *Let L be a regular language and M a model. In the following cases, L has a constant latency M-algorithm with word size $\mathcal{O}(1)$:*
(a) *$M = 2\mathsf{V}$ and L is trivial.*
(b) *$M = 2\mathsf{F}$ and $L \in \langle \mathsf{Len}, \mathsf{Fin} \rangle$*
(c) *$M = 1\mathsf{F}$ and $L \in \langle \mathsf{Len}, \mathsf{SL} \rangle$*

4 Context-free languages

One natural question is which extensions of the regular languages admit constant latency sliding window algorithms. There is certainly no hope to go up to the whole class of context-free languages as this would yield a linear time algorithm for parsing context-free languages. We will show that, even for real-time deterministic context-free languages, there is also little hope to find a constant latency uniform fixed-size sliding window algorithm even though all deterministic context-free languages can be parsed in linear time. More precisely, we will construct a real-time deterministic context-free language for which we prove a lower bound of $\Omega(n^{1/2 - o(1)})$ per update, under the OMV conjecture [23].

On the positive side, in this section we will present constant latency sliding window algorithms for the class of visibly pushdown languages and algorithms with logarithmic latency for deterministic 1-counter languages.

4.1 Lower bound for real-time deterministic context-free languages

First let us recall the online matrix-vector multiplication problem that we will use in our reduction. The *Online Matrix-Vector multiplication* (OMV) problem is the following: the input consists of a Boolean matrix $M \in \{0, 1\}^{n \times n}$ and n Boolean vectors $V_1, \ldots, V_n \in \{0, 1\}^{n \times 1}$ and the vector $M \cdot V_i$ must be computed before the vector V_{i+1} is read. The OMV conjecture [23] states that a RAM with register length $\log n$ cannot solve the OMV problem in time $\mathcal{O}(n^{3 - \epsilon})$ for any $\epsilon > 0$. The OMV conjecture implies tight lower bounds for a number of important problems, like subgraph connectivity, Pagh's problem, d-failure connectivity, decremental single-source shortest paths, and decremental transitive closure; see [23].

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \longrightarrow \$\, m_{13}\, m_{12}\, m_{11}\, \$\, m_{23}\, m_{22}\, m_{21}\, \$\, m_{33}\, m_{32}\, m_{31}\, \#\, v_1\, v_2\, v_3$$

Figure 2 Encoding of a matrix-vector product.

Recall that a real-time deterministic context-free language L is a language that is accepted by a deterministic pushdown automaton without ε-transitions. Hence, the automaton reads an input symbol in each computation step.

▶ **Lemma 11.** *There exists a real-time deterministic context-free language L such that any (uniform) $\mathsf{1F}$-algorithm for L with logarithmic word size and latency $t(n)$ yields an algorithm for the OMV problem with latency $\mathcal{O}(n^2) \cdot t(\mathcal{O}(n^2))$.*

Proof. Let n be the dimension of the OMV problem that we want to solve. We define the window size $m = 3n + n^2 + 1 = \Theta(n^2)$. The reduction will be based on a language $L \subseteq \{a, 0, 1, \$, \#\}^*$ that contains the word $a^j \mathrm{enc}(M)\#\mathrm{enc}(V)a^{n-j}$ (for $M \in \{0,1\}^{n \times n}$, $V \in \{0,1\}^{n \times 1}$, and $1 \le j \le n$) if and only if $M \cdot V$ contains a 1 on its j-th coordinate, for an encoding function enc that we now present. Since we do not care about strings that are not of this form, it does not matter which of them are in L. In fact, a string of the wrong form shall be in L, if and only if the automaton below accepts it.

The encoding of matrices and vectors is illustrated in Figure 2. A Boolean vector $V = [v_1, \ldots, v_n]^\mathsf{T} \in \{0,1\}^{n \times 1}$ is encoded as the binary string $v_1 \cdots v_n \in \{0,1\}^n$. A matrix $M \in \{0,1\}^{n \times n}$ is encoded row by row, with the first row first, and the last row last. Each row starts with the dedicated symbol \$, followed by the encoding of the row (a word over the alphabet $\{0,1\}$) in reverse order. Thus, the encoding of the j-th row starts with $M_{j,n}$ and ends with $M_{j,1}$).

We can construct a real-time deterministic pushdown automaton $\mathcal{P}$ which accepts the word $a^j \mathrm{enc}(M)\#\mathrm{enc}(V)a^{n-j}$ for M, V, j as above, if and only if the j-th entry of the product $M \cdot V$ is 1: The pushdown automaton reads the a^j-prefix on the stack and skips to the j-th row encoding of the matrix by popping a from the stack on every read row delimiter \$. Then it reads the j-th row of M in reverse on the stack, skips to the encoding of V, and can verify whether the j-th row of M and V both have 1-bits at a common position.

Now let us suppose that we have a uniform fixed-size sliding window algorithm $\mathcal{A}$ for L with logarithmic word size and latency $t(n)$. Given the matrix M, we initialize an instance of $\mathcal{A}$ with window size $m = 3n + n^2 + 1 = \Theta(n^2)$ and fill the sliding window with $a^{2n}\mathrm{enc}(M)\#$. This word has length $\mathcal{O}(n^2)$, so this initial preprocessing takes time $\mathcal{O}(n^2) \cdot t(\mathcal{O}(n^2))$.

From the data structure prepared with the window content $a^{2n}\mathrm{enc}(M)\#$ we can get the last bit of the vector $M \cdot V_1$ by loading V_1 into the window with n updates each taking time $t(m) = t(\mathcal{O}(n^2))$. This yields the window content $a^n\mathrm{enc}(M)\#\mathrm{enc}(V_1)$, which is in L if and only if $(M \cdot V_1)_n = 1$. Then loading an a into the window we obtain the window content $a^{n-1}\mathrm{enc}(M)\#\mathrm{enc}(V_1)a$. It belongs to L if and only if $(M \cdot V_1)_{n-1} = 1$. We repeat the process to obtain all bits of $M \cdot V_1$. Computing $M \cdot V_1$ thus requires $2n - 1$ updates and thus time $\mathcal{O}(n) \cdot t(\mathcal{O}(n^2))$. After computing $M \cdot V_1$ we need to compute $M \cdot V_2$, and the other products $M \cdot V_i$ next, and for that we would like to reset the algorithm $\mathcal{A}$ so that the data structure is prepared with the word $a^{2n}\mathrm{enc}(M)\#$, again. Here the final "trick" is applied: instead of doing a new initialization for each vector, requiring $\mathcal{O}(n^2) \cdot t(\mathcal{O}(n^2))$ steps each time, we rather do a rollback: to this end, the sliding window algorithm $\mathcal{A}$ is modified so that each time it changes the value of some register ℓ in memory, it writes ℓ and the old value of register

ℓ into a log. By rolling back this log it is able to undo all changes during the processing of V_1. The extra running time for keeping the log and rolling back the computation is proportional to the number of changes in memory and thus needs time only $\mathcal{O}(n) \cdot t(\mathcal{O}(n^2))$.

Overall the time to deal with one vector is $\mathcal{O}(n) \cdot t(\mathcal{O}(n^2))$ which yields a total running time of $\mathcal{O}(n^2) \cdot t(\mathcal{O}(n^2))$ for OMV. ◀

Lemma 11 states that a sliding window for L with logarithmic word size and latency $t(n) = \mathcal{O}(n^{1/2-\epsilon})$ would yield an algorithm for OMV with a running time of $\mathcal{O}(n^{3-2\epsilon})$, which would contradict the OMV conjecture.

▶ **Corollary 12.** *There exists a fixed deterministic context-free language L such that, conditionally to the OMV conjecture, there is no (uniform) $\mathsf{1F}$-algorithm for L with logarithmic word size and latency $n^{1/2-\epsilon}$ for any $\epsilon > 0$.*

4.2   Visibly pushdown languages

In this section, we provide a constant latency $\mathsf{2V}$-algorithm for visibly pushdown languages. We first define visibly pushdown automata and their languages (for more details see [2]) and then show the upper bound result.

Visibly pushdown automata are like general pushdown automata, but the input alphabet is partitioned into call letters (that necessarily trigger a push operation on the stack), return letters (that necessarily trigger a pop operation on the stack), and internal letters (that do not change the stack). Formally, a *pushdown alphabet* is a triple $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_{int})$ consisting of three pairwise disjoint alphabets: a set of *call letters* Σ_c, a set of *return letters* Σ_r and a set of *internal letters* Σ_{int}. We identify Σ with the union $\Sigma = \Sigma_c \cup \Sigma_r \cup \Sigma_{int}$. The set W of *well-nested* words over Σ is defined as the smallest set such that (i) $\{\varepsilon\} \cup \Sigma_{int} \subseteq W$, (ii) W is closed under concatenation and (iii) if $w \in W$, $a \in \Sigma_c$, $b \in \Sigma_r$ then also $awb \in W$. Every well-nested word over Σ can be uniquely written as a product of *Dyck primes* $D = \Sigma_{int} \cup \{awb : w \in W, a \in \Sigma_c, b \in \Sigma_r\}$ (W is a free submonoid of Σ^* that is freely generated by D). Note that every word $w \in \Sigma^*$ has a unique factorization $w = stu$ with $s \in (W\Sigma_r)^*$, $t \in W$ and $u \in (\Sigma_c W)^*$. To see this, note that the maximal well-matched factors in a word $w \in \Sigma^*$ do not overlap. If these maximal well-matched factors are removed from w then a word from $\Sigma_r^* \Sigma_c^*$ must remain (other one of the removed well-matched factors would be not maximal); see also [15, Section 5].

A *visibly pushdown automaton (VPA)* is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, \bot, q_0, \delta, F)$ where Q is a finite state set, Σ is a pushdown alphabet, Γ is the finite stack alphabet containing a special symbol $\bot$ (representing the bottom of the stack), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and $\delta = \delta_c \cup \delta_r \cup \delta_{int}$ is the transition function where $\delta_c \colon Q \times \Sigma_c \to (\Gamma \setminus \{\bot\}) \times Q$, $\delta_r \colon Q \times \Sigma_r \times \Gamma \to Q$ and $\delta_{int} \colon Q \times \Sigma_{int} \to Q$. The set of *configurations* Conf is the set of all words αq where $q \in Q$ is a state and $\alpha \in \bot(\Gamma \setminus \{\bot\})^*$ is the *stack content*. We define $\hat{\delta} \colon \text{Conf} \times \Sigma \to \text{Conf}$ as follows, where $p \in Q$ and $\alpha \in \bot(\Gamma \setminus \{\bot\})^*$:

- If $a \in \Sigma_c$ and $\delta(p, a) = (\gamma, q)$ then $\hat{\delta}(\alpha p, a) = \alpha \gamma q$.
- If $a \in \Sigma_{int}$ and $\delta(p, a) = q$ then $\hat{\delta}(\alpha p, a) = \alpha q$.
- If $a \in \Sigma_r$, $\gamma \in \Gamma \setminus \{\bot\}$, and $\delta(p, a, \gamma) = q$ then $\hat{\delta}(\alpha \gamma p, a) = \alpha q$.
- If $a \in \Sigma_r$ and $\delta(p, a, \bot) = q$ then $\hat{\delta}(\bot p, a) = \bot q$ (so $\bot$ is not popped from the stack).

As usual we inductively extend $\hat{\delta}$ to a function $\hat{\delta}^* \colon \text{Conf} \times \Sigma^* \to \text{Conf}$ where $\hat{\delta}^*(c, \varepsilon) = c$ and $\hat{\delta}^*(c, wa) = \hat{\delta}(\hat{\delta}^*(c, w), a)$ for all $c \in \text{Conf}$, $w \in \Sigma^*$ and $a \in \Sigma$. In the following, we write $\hat{\delta}$ for $\hat{\delta}^*$. The *initial* configuration is $\bot q_0$ and a configuration c is *final* if $c \in \Gamma^* F$. A word $w \in \Sigma^*$ is *accepted* from a configuration c if $\hat{\delta}(c, w)$ is final. The VPA $\mathcal{A}$ *accepts* w if w is

accepted from the initial configuration. The set of all words accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$; the set of all words accepted from c is denoted by $L(c)$. A language L is a *visibly pushdown language (VPL)* if $L = L(\mathcal{A})$ for some VPA $\mathcal{A}$.

Now we are ready to state the main result of this section.

▶ **Theorem 13.** *Every visibly pushdown language L has a* 2V*-algorithm with latency $\mathcal{O}(1)$, space complexity $\mathcal{O}(n \log n)$ and word size $\mathcal{O}(\log n)$.*

For the proof of Theorem 13 we will use the 2V-algorithm for $\mathsf{prod}_{\mathcal{M}}$ from the proof of Theorem 3 ($\mathcal{M}$ is again a finite monoid). In the following, we call the data structure behind this algorithm a $\mathsf{DABA}(\mathcal{M})$ data structure, where DABA stands for deamortized banker's algorithm (the name for the data structure in [28] used for the proof of Theorem 2). In the proof of Theorem 3, $\mathsf{DABA}(\mathcal{M})$ stores a sequence of monoid elements $m_1, m_2, \ldots, m_k \in \mathcal{M}$ and a $\mathsf{prod}_{\mathcal{M}}$-query returns the monoid product $m_1 m_2 \cdots m_k$. For our application of the $\mathsf{DABA}(\mathcal{M})$ data structure to visibly pushdown languages in the next section, we have to store sequences of pointers $p_1, p_2, \ldots, p_k$. Following pointer p_k we can determine a monoid element m_i (and some additional data values that will be specified below). When applying a $\mathsf{prod}_{\mathcal{M}}$-query to such a sequence of pointers, the monoid product $m_1 m_2 \cdots m_k \in \mathcal{M}$ is returned. In addition, we have the update-operations from 2V-model that allow to remove the first or last pointer or to add a new pointer at the beginning or end. All operations work in constant time.

Proof sketch of Theorem 13. Let us fix a (deterministic) VPA $\mathcal{A} = (Q, \Sigma, \Gamma, \bot, q_0, \delta, F)$ with $\Sigma = (\Sigma_c, \Sigma_r, \Sigma_{int})$. As usual, we denote with Q^Q the monoid of all mappings from Q to Q with composition of functions as the monoid operation (see also Section 3.1). Notice that $\mathcal{A}$ can only see the top of the stack when reading return symbols. Therefore, the behavior of $\mathcal{A}$ on a well-nested word is determined only by the current state and independent of the current stack content. We can therefore define a mapping $\phi \colon W \to Q^Q$ by $\hat{\delta}(\bot p, w) = \bot \phi(w)(p)$ for all $w \in W$ and $p \in Q$. Note that this implies $\hat{\delta}(\alpha p, w) = \alpha \, \phi(w)(p)$ for all $w \in W$ and $\alpha p \in \mathrm{Conf}$. The mapping ϕ is a monoid morphism (recall that W is a monoid with respect to concatenation).

For the further consideration, it is useful to extend ϕ to a mapping $\phi \colon \Sigma^* \to Q^Q$, which will be no longer a monoid morphism. To do this we first define $\phi(a) : Q \to Q$ for letters $a \in \Sigma_r \cup \Sigma_c$ by $\delta(p, a, \bot) = \phi(a)(p)$ for $a \in \Sigma_r$ and $\delta(p, a) = (\gamma, \phi(a)(p))$ for $a \in \Sigma_c$ (and some $\gamma \in \Gamma \setminus \{\bot\}$). Note that for a return letter a, $\phi(a)$ is the state transformation induced by the letter a on the stack only containing $\bot$. Consider now an arbitrary word $w \in \Sigma^*$. As mentioned above, there is a unique factorization

$$w = w_0 a_1 w_1 a_2 w_2 \cdots a_k w_k \in (W \Sigma_r)^* W (\Sigma_c W)^* \tag{1}$$

such that $k \geq 0$, $w_i \in W$ for all $0 \leq i \leq k$ and for some $s \in [0, k]$ (the *separation position*) we have $a_1, \ldots, a_s \in \Sigma_r$ and $a_{s+1}, \ldots, a_k \in \Sigma_c$. We then define the mapping $\phi(w) : Q \to Q$ as $\phi(w) = \phi(w_0)\phi(a_1)\phi(w_1) \cdots \phi(a_k)\phi(w_k)$. Again, notice that the mapping $\phi : \Sigma^* \to Q^Q$ is not a monoid homomorphism. However, ϕ captures the behaviour of $\mathcal{A}$ on a word w:

▷ **Claim 14.** For every word $w \in \Sigma^*$ and all states $q \in Q$ we have $\hat{\delta}(\bot q, w) = \alpha \phi(w)(q)$ for some stack content $\alpha \in \bot(\Gamma \setminus \{\bot\})^*$.

By this claim, shown in the long version [20], a variable-size sliding window algorithm for $L(\mathcal{A})$ only needs to maintain the state transformation $\phi(w)$ for the current window content w.

With a well-nested word $w \in W$ we associate a node-labelled ordered tree $T(w)$ as follows, where $\mathrm{id}_Q \in Q^Q$ is the identity mapping on Q:
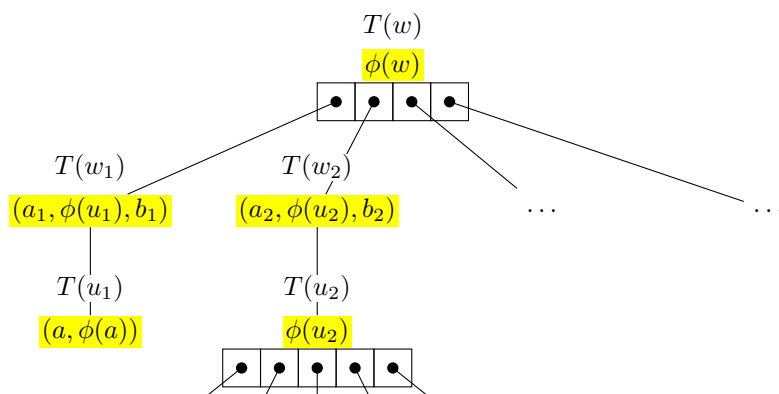
Figure 3 The data structure of the 2V-algorithm for VPLs. A well-nested word w decomposes into Dyck primes $w = w_1 w_2 \cdots w_k$ where each w_i is either an internal letter or consists of a call letter a_i, a well-nested word u_i and a return letter b_i. The node of w stores the state transformation $\phi(w)$ together with a $\mathrm{DABA}(Q^Q)$ instance, which maintains a list of the children w_i and their state transformations $\phi(w_i)$.

1. If $w = \varepsilon$ then $T(w)$ consists of a single node labelled with the pair $(\varepsilon, \mathrm{id}_Q)$.
2. If $w = a \in \Sigma_{int}$ then $T(w)$ consists of a single node labelled with $(a, \phi(a))$.
3. If w is a Dyck prime aub with $a \in \Sigma_c$, $u \in W$ and $b \in \Sigma_r$, then $T(w)$ consists of a root node, labelled with $(a, \phi(w), b)$, with a single child which is the root of the tree $T(u)$.
4. If $w = w_1 \cdots w_k$ for Dyck primes $w_1, \dots, w_k$ and $k \geq 2$, the root of $T(w)$ is labelled with $\phi(w)$ and it has the roots of the trees $T(w_1), \dots, T(w_k)$ k as children (from left to right).

See Figure 3 for an illustration of the tree $T(w)$. We also speak of a tree $T(w)$ of type (i) (for $1 \leq i \leq 4$). Moreover, we say that a node v is of type (i) if the subtree rooted in v is of type (i). Note that the type of node can be obtained from its label.

We have to implement several operations on such trees. In order to spend only constant time for each of the operations, we maintain the children $v_1, \dots, v_k$ of a node v of type (4) as a $\mathrm{DABA}(Q^Q)$ data structure which we denote by $\mathsf{DABA}(v)$ (note that v_i is the root of the tree $T(w_i)$). This data structure is needed in order to maintain the value $\phi(w) = \phi(w_1) \cdots \phi(w_k)$ (the label of v). For the implementation of $\mathsf{DABA}(v)$ we have to slightly extend the data structure from the proof of Theorem 3: there, every entry of the data structure stores an element of the monoid $\mathcal{M}$ (here, Q^Q). Here, the entries of $\mathsf{DABA}(v)$ are pointers to the children $v_1, \dots, v_k$. Note that from v_i we can obtain in constant time the monoid value $m_i := \phi(w_i) \in Q^Q$ as its label. The partial products $m_i \cdots m_j$ for the guardians as well as the additional $\mathcal{O}(1)$ many pointers to entries of the DABA data structure are treated as in the proof of Theorem 3. For the purpose of maintaining $\phi(w)$ only the monoid elements of $m_1, \dots, m_k$ are relevant, but we use $\mathsf{DABA}(v)$ also in order to store the list of children of v and hence the tree structure of $T(w)$. Intuitively, a tree $T(w)$ can be seen as a nested DABA data structure for the well-nested word w.

Assume now that the current window content is $w \in \Sigma^*$ and consider the unique factorization for w in (1) with the separation position $s \in [0, k]$. For a symbol $a \in \Sigma_c \cup \Sigma_r$ we write $\langle a \rangle$ for the pair $(\phi(a), a)$ below. Our 2V-algorithm stores on the top level two lists and a tree (here, again, every tree $T(w_i)$ is represented by a pointer to its root):

- the descending list $L_\downarrow = [T(w_0), \langle a_1 \rangle, T(w_1), \langle a_2 \rangle, \dots, T(w_{s-1}), \langle a_s \rangle]$
- the separating tree $T_s = T(w_s)$
- the ascending list $L_\uparrow = [\langle a_{s+1} \rangle, T(w_{s+1}), \dots, \langle a_k \rangle, T(w_k)]$.

These two lists are maintained by the DABA data structures $\mathsf{DABA}_\downarrow$ and $\mathsf{DABA}_\uparrow$, respectively. These DABA data structures maintain the aggregated state transformations $\phi(w_0 a_1 w_1 a_2 \cdots w_{s-1} a_s)$ and $\phi(a_{s+1} w_{s+1} \cdots a_k w_k)$ from which, together with $\phi(w_s)$ (which can be obtained from the root of T_s) we can obtain the value $\phi(w)$, which, in turn, allows to check whether $w \in L(\mathcal{A})$ by Claim 14. Note that $L_\downarrow = []$ (the empty list) in case $s = 0$ and $L_\uparrow = []$ in case $k = s$. Since each of lists $L_\downarrow$ and $L_\uparrow$ can be empty, we will need all four types of operations (leftpop, rightpop, leftpush, and rightpush) for $L_\downarrow$ and $L_\uparrow$. Therefore, the symmetric DABA data structure from Theorem 3 is really needed here (even if we only would aim for a 1V-algorithm).

Using this data structure, it is not difficult (but a bit tedious) to implement all window update operations in constant time, see [20]. Note that the above data structure uses $\mathcal{O}(n \log n)$ bits: we have to store $\mathcal{O}(n)$ many pointers of bit length $\mathcal{O}(\log n)$. This concludes our proof sketch of Theorem 13. ◀

4.3   Deterministic 1-counter automata

In this section we show that every deterministic 1-counter language has a 2V-algorithm with latency $\mathcal{O}(\log n)$ on a RAM with word size $\mathcal{O}(\log n)$. A deterministic 1-counter automaton (DOCA) $\mathcal{A}$ is a deterministic finite automaton equipped with a single $\mathbb{N}$-counter. Its transition function δ specifies for every combination of current state q and emptiness condition of the counter ($= 0$ or > 0) either an ε-move (where no input symbol is read) or a unique a-move for every input symbol a. A move is specified by the next state q' and a counter update $d \in \{-1, 0, 1\}$ (where $d \geq 0$ if the counter is zero). To simplify the presentation in the following, we only work with DOCAs without ε-moves; see Appendix C for the general and formal definition. A *deterministic 1-counter language* is the language accepted by a DOCA.

The set of *configurations* of a DOCA $\mathcal{A}$ with state set Q is $Q \times \mathbb{N}$. Every word $w \in \Sigma^*$ induces an *effect* $\hat{\delta}_w : Q \times \mathbb{N} \to Q \times \mathbb{N}$ where $\hat{\delta}_w(q, m) = (q', m')$ if and only if reading w starting from configuration (q, m) reaches configuration (q', m'). It is easy to see that $\hat{\delta}_w$ can be completely reconstructed from the restriction of $\hat{\delta}_w$ to the set $Q \times [0, |w|]$, since along every run of length $|w|$ starting with a counter of at least $|w|$, the counter can only become zero in the last step. Therefore, if $\hat{\delta}_w(q, |w|) = (q', m)$ then $\hat{\delta}_w(q, |w| + d) = (q', m + d)$. In the following the effect $\hat{\delta}_w$ of a word w is stored by its values on all configurations from $Q \times [0, |w|]$. For this, $\mathcal{O}(|w|)$ many registers of bit length $\mathcal{O}(\log |w|)$ suffice.

Observe that (i) given a configuration $c \in Q \times [0, n]$ and the effect $\hat{\delta}_w$ of a word of length at most n, one can compute $\hat{\delta}_w(c)$ in constant time, and (ii) given the effects $\hat{\delta}_u, \hat{\delta}_v$ of two words u, v of length at most n, one can compute the effect $\hat{\delta}_{uv}$ in time $\mathcal{O}(n)$. For DOCAs with ε-transitions we can also represent effects (slightly more involved) so that properties analogous to (i) and (ii) hold; see Appendix C for details.

▶ **Theorem 15.** *Every deterministic 1-counter language L has a 2V-algorithm with latency $\mathcal{O}(\log n)$, space complexity $\mathcal{O}(n \log^2 n)$ and word size $\mathcal{O}(\log n)$.*

Proof sketch. Let w be the current window and n its length and let $\mathcal{A}$ be a deterministic 1-counter automaton for L. Effects of words are always taken with respect to $\mathcal{A}$. Our 2V-algorithm will preserve the following invariants:

1. The current window w is factorized into blocks $B_0, \ldots, B_m$ where B_i has length 2^{a_i} and for some $k \in [-1, m]$ we have $a_0 < a_1 < \cdots < a_k$ and $a_{k+1} > a_{k+2} > \cdots > a_m$. A block of length 2^a is also called a level-a block in the following.

2. Every level-a block B is recursively factorized into two level-$(a-1)$ blocks that we call B's left and right half. So the blocks $B_0, \ldots, B_m$ are the maximal blocks, i.e., every other block is contained in some B_i. The collection of all blocks is stored as a forest of full binary trees $T_0, \ldots, T_m$, where T_i is the tree for block B_i.

3. For a certain time instant, let the age of a block B be the number of window updates that have occurred between the first point of time, where B is completely contained in the window and the current point of time. Note that B can either enter the window on the left or on the right end. If B is a level-a block and has age at least $2^a - 1$, then the effect of B must be completely computed and stored in the tree node corresponding to block B. We call such a block *completed*. In particular, the effect of a block of length 1 has to be available after one further update.

The following claim is an immediate consequence of the 3rd invariant.

▷ **Claim 16.** If a level-a block B has at least $2^a - 1$ many symbols to its left as well as at least $2^a - 1$ many symbols to its right in the window, then its age is at least $2^a - 1$, so it must be completed.

Thus, at every time instant, on every level i, there can be at most 2 non-completed blocks, namely the left most and right most block.

In Appendix D we show the following claim, from which it is easy to conclude that the query time is bounded by $\mathcal{O}(\log n)$.

▷ **Claim 17.** At each time instant the window w factors into $\mathcal{O}(\log n)$ completed blocks.

It remains to describe how to deal with window updates and how to preserve the invariants. We first consider the operation $\mathsf{leftpush}(a)$. Let t be the current point of time. We measure time in terms of window updates; every window update increments time by 1. Let us assume that $a_k \geq a_{k+1}$ (if $a_k < a_{k+1}$ then one has to replace k by $k+1$ below).

Basically we make an increment on the binary representation of the number $2^{a_0} + \cdots + 2^{a_k}$. In other words, we consider the largest number $j \leq k$ such that $a_i = i$ for all $0 \leq i \leq j$ and replace the blocks $B_0, \ldots, B_j$ together with the new a in the window by a single level-$(j+1)$ block B'_{j+1}. Thereby also one new level-i subblock B'_i of B'_j for every $0 \leq i \leq j$ arises: we have $B'_0 = a$ and $B'_i = B'_{i-1}B_{i-1}$ for $1 \leq i \leq j+1$. By invariant 3, all blocks $B_0, \ldots, B_{j-1}$ are completed. If $j < k$ then B_j is also completed, but if $j = k$ (and $a_k > a_{k+1}$) then we can only guarantee that the left half of B_j is completed; its right half might be still non-completed. Let us assume that $j = k$, which is the more difficult case.

The effects of the new blocks $B'_0, B'_1, \ldots, B'_k$ can be computed bottom up as follows: The effect of $B'_0 = a$ is immediately computed when a arrives in the window. If the effect of B'_{i-1} is already computed, then using the equality $B'_i = B'_{i-1}B_{i-1}$ and using the fact that B_{i-1} is completed, we can compute the effect of B'_i in time $c \cdot 2^i$ for some constant c. In total, the computation of the effects of all blocks $B'_0, B'_1, \ldots, B'_k$ needs time $\sum_{0 \leq i \leq k} c \cdot 2^i \leq c \cdot 2^{k+1}$. We amortize this work over the next $2^k - 1$ window updates by doing a constant amount of work in each step. Thereby we ensure that at time instant $t + 2^i - 1$, the new blocks $B'_0, \ldots, B'_i$ are completed for every $0 \leq i \leq k$. In particular, at time instant $t + 2^k - 1$, the block B'_k is completed. But at that time instant, also B_k must be completed (if it is still in the window – due to pops B_k might haven been disappeared in the meantime) and we can reach the goal of completing B'_{k+1} at time $t + 2^{k+1} - 1$ by still doing only a constant amount of work in each step. This ensures that invariant 3 is preserved for every new block B'_i. Moreover, before another level-i block arises at the left end of the window (which can only happen every 2^i steps), the new block B'_i is completed. Since the same arguments apply

to rightpush(a), it follows that at every time instant, on each level i only two blocks are in the process of completion (one that was created on the left end and one that was created on the right end). Since there are at most $\log(n)$ levels and for the completion of each block a constant amount of work is done in each step, we get the time bound $\mathcal{O}(\log n)$.

For leftpop(), one has to remove all blocks along the leftmost path in the tree T_0 for block B_0, which results in a sequence of smaller blocks of length $2^0, \ldots, 2^{a-1}$ if B_0 is a level a-block. These blocks are already present in the tree T_0. Some of the blocks on the leftmost path of T_0 might be not completed so far. Of course we stop the computation of their effects. Invariant 1 is preserved, also in case $k = -1$ (where $a_0 > a_1 > \ldots > a_m$).

Since the data structure is symmetric, the operations rightpop() and rightpush(a) can be implemented in the same way. The algorithm uses space $\mathcal{O}(n \log^2 n)$: the dominating part are the values of the effect functions. On each level these are at most n numbers of bit length $\mathcal{O}(\log n)$. Moreover there at at most $\log n$ levels. This concludes the proof. ◀

5   Open problems

We conclude with some open problems: In Theorem 3 we assume that the size of the finite automaton for L is a constant. One should also investigate how the optimal word size and latency depend on the number of states of the automaton. For space complexity, this dependency is investigated in [16].

We showed that there is a real-time deterministic context-free language L such that, conditionally to the OMV conjecture, there is no 1F-algorithm for L with logarithmic word size and latency $n^{1/2-\epsilon}$ for any $\epsilon > 0$. The best known upper bound in this setting for deterministic context-free languages we are aware of is $\mathcal{O}(n/\log n)$. It is open, whether every deterministic context-free language has a one-way fixed-size sliding window algorithm with logarithmic word size and latency $n^{1-\epsilon}$ for some $\epsilon > 0$.

For every deterministic one-counter language L, we showed that there is a 2V-algorithm with latency $\mathcal{O}(\log n)$ and word size $\mathcal{O}(\log n)$. Here, it remains open, whether the latency can be further reduced, maybe even to a constant. Also, our space bound $\mathcal{O}(n \log^2 n)$ is not optimal. It would be nice to reduce it to $\mathcal{O}(n)$ without increasing the latency. The same problem appears for visibly pushdown languages, where our current space bound is $\mathcal{O}(n \log n)$ (with constant latency). Finally, it would be interesting to see, whether the 2V-algorithm with latency $\mathcal{O}(1)$ for visibly pushdown languages can be extended to the larger class of operator precedence languages [11, 8].

References

1 Charu C. Aggarwal. *Data Streams – Models and Algorithms*. Springer, 2007.

2 Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pages 202–211. ACM, 2004. `doi:10.1145/1007352.1007390`.

3 Antoine Amarilli, Louis Jachiet, and Charles Paperman. Dynamic membership for regular languages. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 116:1–116:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.116`.

4 Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma. Streaming algorithms for language recognition problems. *Theoretical Computer Science*, 494:13–23, 2013. `doi:10.1016/j.tcs.2012.12.028`.

5 Ajesh Babu, Nutan Limaye, and Girish Varma. Streaming algorithms for some problems in log-space. In *Proceedings of the 7th Annual Conference on Theory and Applications of Models of Computation, TAMC 2010*, volume 6108 of *Lecture Notes in Computer Science*, pages 94–104. Springer, 2010. `doi:10.1007/978-3-642-13562-0_10`.

6   Gabriel Bathie and Tatiana Starikovskaya. Property testing of regular languages with applications to streaming property testing of visibly pushdown languages. In *Proceedings of the 48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, volume 198 of *LIPIcs*, pages 119:1–119:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.119`.

7   Stanislav Böhm, Stefan Göller, and Petr Jancar. Equivalence of deterministic one-counter automata is NL-complete. In *Proceedings of the 45th ACM Symposium on Theory of Computing, STOC 2013*, pages 131–140. ACM, 2013. `doi:10.1145/2488608.2488626`.

8   Stefano Crespi-Reghizzi and Dino Mandrioli. Operator precedence and the visibly pushdown property. *Journal of Computer and System Sciences*, 78(6):1837–1867, 2012. `doi:10.1016/j.jcss.2011.12.006`.

9   Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, 31(6):1794–1813, 2002. `doi:10.1137/S0097539701398363`.

10   Eldar Fischer, Frédéric Magniez, and Tatiana Starikovskaya. Improved bounds for testing Dyck languages. In *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1529–1544. SIAM, 2018. `doi:10.1137/1.9781611975031.100`.

11   Robert W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, 1963. `doi:10.1145/321172.321179`.

12   Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre. Streaming property testing of visibly pushdown languages. In *Proceedings of the 24th Annual European Symposium on Algorithms, ESA 2016*, volume 57 of *LIPIcs*, pages 43:1–43:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.ESA.2016.43`.

13   Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum. Dynamic algorithms for the Dyck languages. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures, WADS 1995*, volume 955 of *Lecture Notes in Computer Science*, pages 98–108. Springer, 1995. `doi:10.1007/3-540-60220-8_54`.

14   Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *Journal of the ACM*, 44(2):257–271, 1997. `doi:10.1145/256303.256309`.

15   Moses Ganardi. Visibly pushdown languages over sliding windows. In Rolf Niedermeier and Christophe Paul, editors, *Proceedings of the 36th International Symposium on Theoretical Aspects of Computer Science, STACS 2019*, volume 126 of *LIPIcs*, pages 29:1–29:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.STACS.2019.29`.

16   Moses Ganardi, Danny Hucke, Daniel König, Markus Lohrey, and Konstantinos Mamouras. Automata theory on sliding windows. In *Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, volume 96 of *LIPIcs*, pages 31:1–31:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.STACS.2018.31`.

17   Moses Ganardi, Danny Hucke, and Markus Lohrey. Querying regular languages over sliding windows. In *Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016*, volume 65 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.FSTTCS.2016.18`.

18   Moses Ganardi, Danny Hucke, and Markus Lohrey. Querying languages over sliding windows. *CoRR*, abs/1702.04376v1, 2017. `arXiv:1702.04376`.

19   Moses Ganardi, Danny Hucke, and Markus Lohrey. Randomized sliding window algorithms for regular languages. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, volume 107 of *LIPIcs*, pages 127:1–127:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ICALP.2018.127`.

20   Moses Ganardi, Louis Jachiet, Markus Lohrey, and Thomas Schwentick. Low-latency sliding window algorithms for formal languages. *CoRR*, abs/2209.14835, 2022. `arXiv:2209.14835`.

21   Moses Ganardi, Artur Jez, and Markus Lohrey. Sliding windows over context-free languages. In *Proceedings of the 43rd International Symposium on Mathematical Foundations of Computer Science, MFCS 2018*, volume 117 of *LIPIcs*, pages 15:1–15:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018.

22   Alejandro Grez, Filip Mazowiecki, Michał Pilipczuk, Gabriele Puppis, and Cristian Riveros. Dynamic Data Structures for Timed Automata Acceptance. In *Proceedings of the 16th International Symposium on Parameterized and Exact Computation, IPEC 2021*, volume 214 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.IPEC.2021.20`.

23   Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. `doi:10.1145/2746539.2746609`.

24   Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. `doi:10.1016/S0019-9958(65)90426-2`.

25   Donald E. Knuth. *The art of computer programming, Volume I: Fundamental Algorithms, 3rd Edition*. Addison-Wesley, 1997. URL: `https://www.worldcat.org/oclc/312910844`.

26   Andreas Krebs, Nutan Limaye, and Srikanth Srinivasan. Streaming algorithms for recognizing nearly well-parenthesized expressions. In *Proceedings of the 36th International Symposium on Mathematical Foundations of Computer Science, MFCS 2011*, volume 6907 of *Lecture Notes in Computer Science*, pages 412–423. Springer, 2011. `doi:10.1007/978-3-642-22993-0_38`.

27   Frédéric Magniez, Claire Mathieu, and Ashwin Nayak. Recognizing well-parenthesized expressions in the streaming model. *SIAM Journal on Computing*, 43(6):1880–1905, 2014. `doi:10.1137/130926122`.

28   Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017*, pages 66–77. ACM, 2017. `doi:10.1145/3093742.3095107`.

A   Additional material for the proof of Theorem 3

▶ **Lemma 18.** *Let T be a non-decreasing function and let A be a 2V-algorithm for a function f such that, when initialized with the empty window and a maximal window size of n,*
- *A works on a unit-cost RAM with word size $\mathcal{O}(\log n)$,*
- *the initialization takes time $T(n)$,*
- *all later window operations run in time $T(n)$ and*
- *A stores in total at most $\mathcal{O}(n)$ bits.*

With such an A, we can build a 2V-algorithm A^* for f without the maximal window size limitation and with latency $\mathcal{O}(T(4n))$, space complexity $\mathcal{O}(n)$ and word size $\mathcal{O}(\log n)$.

Proof. To get the statement of the lemma but with an amortized complexity bound, we could design the algorithm A^* to work just like any "dynamic array": let us denote with A_n the version of the algorithm that works for window size up to n. Assume that we currently work with A_n. If the window size grows to n we switch to A_{2n} and if the window size shrinks to $n/4$ we switch to $A_{n/2}$. To do the switching, we transfer the current window content using rightpush-operations (we could also use leftpush) of A_{2n} or $A_{n/2}$ from the current version A_n to the new version. We call this a reset. If we reset to A_n we know that the window size was $n/2$. Therefore, the reset takes time $(n/2 + 1) \cdot T(n)$ (the +1 comes from the initialization which takes time $T(n)$). On the other hand, the next reset only happens if the window size grows to $2n$ or shrinks to $n/4$. Therefore, we make at least $n/4$ non-reset operations before the next reset happens. This leads to an amortized latency of $\mathcal{O}(T(4\ell))$, where ℓ is the current window size. The term $T(4\ell)$ comes from the fact that at each time instant we work with a version A_n, where n is at most a factor 4 larger than the actual window size.

To get non-amortized time bounds we need to maintain two instances of A at all time. One instance A_{cur} solving the problem (just like in the amortized case) and one instance A_{nxt} that we prepare in the background so that whenever we need to double or halve the maximal window size, the instance A_{cur} can just be replaced with A_{nxt}. Our algorithm thus stores two instances of A (A_{cur} and A_{nxt}), a copy of the current word w in the window (as an amortized circular buffer), some bookkeeping to know what is the size p of the prefix of w that has been loaded into A_{nxt}, the current size of w and the parameter n of the maximal window size of A_{cur}. Overall the algorithm stores $\mathcal{O}(\ell)$ bits where ℓ is the current window size ℓ. To prepare A_{nxt}, we decide that whenever the current window size is above $n/2$, we prepare A_{nxt} to work with $2n$ and whenever the window size is below $n/2$, we prepare A_{nxt} to work with $n/2$. Note that each time the window size crosses the $n/2$ threshold, our algorithm will reset A_{nxt} but each reset only takes time $T(2n)$ or $T(n/2)$, hence time $T(2n)$ at most (if T is monotone).

Propagating the effect of each window operation on A^* to A_{cur} and w is easy, we just apply the effect for w and call the right operation for A_{cur}. For A_{nxt}, if the window operation is a leftpop() or leftpush(a), we carry out the same operation in A_{nxt}. This ensures that at any point during the algorithm, A_{nxt} stores a prefix of the current window whose length $p-1$ (case leftpop()) or $p+1$ (case leftpush(a)). If the operation is a rightpop() or rightpush(a), we simply ignore it in A_{nxt}. Now, in both cases, i.e., for every update operation, we call rightpush(a) in A_{nxt} for up to 5 elements a ($w[p+1]$, $w[p+2]$, $w[p+3]$, $w[p+4]$, and $w[p+5]$) so that overall p (the length of the prefix of the window copied to A_{nxt}) increases by at least 4. Note that there might be less than 5 elements to push if A_{nxt} is almost ready.

Finally, note that because we switch between A_{cur} and A_{nxt} when the window size reaches $n/4$ or n and we restart the above copying process from A_{cur} to A_{nxt} whenever the window size becomes larger or smaller than $n/2$, we know that at least $n/4$ window updates must occur between the last restart and the actual switch from A_{cur} to A_{nxt}. Therefore by increasing the size of the prefix covered by A_{nxt} by 4 for each window update, we know that A_{nxt} covers the full window whenever we have to switch. Overall we do have a scheme with the right latency as we only do a constant number of calls to algorithm A for each window operation. ◀

B   Proof of Theorem 7

We first consider the case that $L \in \mathsf{Len}$ and $M = 2\mathsf{V}$. In principle, it suffices to keep track of the current window size n and to check whether it is in some fixed semilinear set S. In fact, from L one compute a number N and two finite sets A and B such that a string is in L if and only if its length n satisfies (1) $n \geq N$ and $n \bmod N \in A$ or (2) $n < N$ and $n \in B$.

Obviously, the number n can be stored with $\mathcal{O}(\log n)$ bits. However, n needs to be incremented and decremented and compared, and the naive way of doing that may require the manipulation of $\log n$ bits for one operation, in worst case. Therefore we need to make use of a more sophisticated data structure that allows updates with a constant number of operations that manipulate only $\mathcal{O}(\log \log n)$ bits each.

Whenever $n \geq N$, the data structure uses a counter m that stores $n - N$, and a variable r of constant size that keeps track of $n \bmod N$ to check whether $n \bmod N \in A$. Whenever $n < N$, it stores n in a variable of constant size and maintains whether $n \in B$ holds.

During initialization, m is set to zero, and whenever $n < N$ it stays at zero. The challenging part is to maintain m if $n > N$ and to recognize whenever a phase with $n > N$ ends by reaching $m = 0$. This can be done using the counting techniques from [14]. In a

nutshell, the method basically works just as incrementing a binary number by, say, a Turing machine. However, to avoid long delays, it encodes the position of the head of the Turing machine in the string (by an underscore of the digit at that position) and each "move" of the Turing machine corresponds to an incremented number. E.g., the numbers 0,1,2,3,4 could be represented by $00\underline{0}, 00\underline{1}, 01\underline{0}, 01\underline{1}, 0\underline{1}0$. It is not obvious how to determine the number represented by such a string. However, for our purposes it suffices to increment numbers, to compare them with a fixed constant number, and to initialize them with a fixed number.

The algorithm maintains the number m by a bit string s of length ℓ for some ℓ with $2^{\ell+1} - \ell - 2 \geq m$. The number ℓ is stored with $\mathcal{O}(\log \ell)$ bits. One position x in s is considered as marked and stored with $\mathcal{O}(\log \ell)$ bits as well. The representation of the number m by s and x is defined as follows (where the marked bit in s is underlined and u, v are bit strings):

- The number 0 is represented as $0^{\ell-1}\underline{0}$.
- If number t is represented by $u\underline{0}$ then $t+1$ is represented by $u\underline{1}$.
- If number t is represented by $u0\underline{1}v$ then $t+1$ is represented by $u1\underline{0}v$.
- If number t is represented by $u\underline{0}0v$ then $t+1$ is represented by $u0\underline{0}v$.
- If number t is represented by $u1\underline{1}v$ then $t+1$ is represented by $u\underline{1}0v$.
- If number t is represented by $\underline{1}0^{\ell-1}$ then $t+1$ is represented by $1\underline{0}0^{\ell-1}$, and ℓ subsequently has to be incremented by 1.

The last of these cases does not appear in [14]; it is needed since in [14] m and ℓ are fixed, which is not the case in our application. It is a crucial observation that all bits to the right of the marked position are always zero. In particular this allows to identify when a situation of the form $\underline{1}0^{\ell-1}$ is obtained. We note that additional leading zeros do not spoil the representation of a number. Therefore ℓ needs never be decreased when the window size n and hence m is decreased.

One can show that in this way every number is represented in a unique way, ignoring unmarked leading zeros; see [14]. The above rules also specify how to increment and decrement m (for decrement one has to reverse the rules). Note that the rules modify s only in a local way; therefore they can be implemented in time $\mathcal{O}(1)$ on a RAM of word size $\mathcal{O}(\log \log n)$.

It remains to explain how to check whether $m = 0$ holds. To this end, an additional number is stored, which is the minimal position y in the bit string such that all positions to the left of it are 0 (the positions are numbered from right to left, i.e., the right-most position has number 0). For y, $\mathcal{O}(\log \ell)$ bits suffice, as well. Its manipulation is straightforward with the above rules.

If n is the maximal window size seen in the past, the algorithm stores a bit array of length $\mathcal{O}(\log n)$ for the bit string s and three registers of bit length $\mathcal{O}(\log \log n)$ for the numbers ℓ, x, y. This completes the description of the case that $L \in \mathsf{Len}$ and $M = 2\mathsf{V}$.

We now consider the case that $L \in \langle \mathsf{Len}, \mathsf{LI} \rangle$ and $M = 1\mathsf{V}$. It suffices to consider the cases that $L \in \mathsf{Len}$ and $L \in \mathsf{LI}$: if we have a boolean combination of such languages, we can run the sliding window algorithms for these languages in parallel and combine their results according to the boolean formula. The case $L \in \mathsf{Len}$ is covered by the first part of the proof (for $M = 2\mathsf{V}$), so it suffices to consider the case $L \in \mathsf{LI}$ and $M = 1\mathsf{V}$. Hence, L is a left ideal. Let $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ be a DFA for the *reversal language* $L^\mathsf{R} = \{w^\mathsf{R} : w \in L\}$ of the left ideal L where the reversal of a word $w = a_1 a_2 \cdots a_n \in \Sigma^*$ is $w^\mathsf{R} = a_n \cdots a_2 a_1$. Since L is a left ideal we can assume that F contains a unique final state q_F, which is also a sink, i.e. $\delta(q_F, a) = q_F$ for all $a \in \Sigma$.

We use a simplified version of the $\mathcal{O}(\log n)$-space *path summary algorithm* from [16]. A *path summary* is an unordered list $(Q_1, n_1)(Q_2, n_2) \ldots (Q_k, n_k)$ where the $Q_i \subseteq Q$ are pairwise disjoint and nonempty and the $n_i \in [0, n]$ are pairwise different. Note that $k \leq |Q|$

which is a constant in our setting. The meaning of a pair (Q_i, n_i) is the following, where w is the current window content: Q_i is the set of all states q for which n_i is the length of the shortest suffix s of w such that $\delta(q, s^R) = q_F$. Furthermore, if there exists no such suffix for a state q then the state q does not appear in any set Q_i. Clearly $w \in L$ if and only if $q_0 \in \bigcup_{i=1}^{k} Q_i$. Hence, it suffices to maintain a path summary for the current window content w. We first describe, how the operations can be handled, in principle.

For leftpop(), the algorithm removes the unique pair (Q_i, n_i) with $n = n_i$ if it exists. For rightpush(a), it replaces each pair (Q_i, n_i) by $(\{p \in Q \setminus \{q_F\} : \delta(p, a) \in Q_i\}, n_i + 1)$ if $\{p \in Q \setminus \{q_F\} : \delta(p, a) \in Q_i\}$ is non-empty, otherwise the pair is removed from the path summary. Finally, the pair $(\{q_F\}, 0)$ is added to the path summary.

It is easy to verify that the path summary is correctly maintained, in this way. However, some care is needed to guarantee the bounds claimed in the statement of the theorem.

We first observe that the length k of the path summary is bounded by the constant $|Q|$. Thus the algorithm stores at most $|Q|$ many pairs (Q_i, n_i). They can be stored in a bit array of length $\mathcal{O}(\log n)$ that is divided into $|Q|$ many chunks. Every chunk stores one pair (Q_i, n_i). The state set $Q_i \subseteq Q$ is stored with $|Q|$ many bits. The number n_i can be stored with $\log n$ many bits. Every chunk has an additional *activity bit* that signals whether the chunk is active or not. This is needed since in the path summary the algorithm has to be able to remove and add pairs (Q_i, n_i). If the activity bit is set to 0 then the chunk is released and can used for a new pair, later on. In addition the algorithm stores the window size n.

We have seen in the case $L \in$ Len, how the kind of counters that are needed for a path summary and the window size can, in principle, be implemented such that a single update works with a constant number of operations that manipulate only $\mathcal{O}(\log \log n)$ bits. However, there are still two challenges that need to be mastered: (1) the counters n_i need to be compared with the number n (the window size), which itself can change, and (2) when a chunk is deactivated, its counter n_i may represent any number, but when its re-activated it should be 0, again. Towards (1), the algorithm maintains a second counter, m_i for each chunk, which is supposed to represent $n - n_i$. Thus, to test $n = n_i$ it suffices to check whether $m_i = 0$ and we know from the Len-case how this can be done. Note that for an leftpop(), m_i must be decremented and for a rightpush(a), m_i does not change.

For (2), the algorithm uses a "lazy copying" technique (from 0 and from n, respectively). The algorithm stores two additional numbers b_i and d_i of size $\mathcal{O}(\log \log n)$ for each counter m_i and one additional number a_i of size $\mathcal{O}(\log \log n)$ for each counter n_i.

We first describe, how to deal with n_i. If chunk i becomes activated, n_i is supposed to be initialized to 0, but the actual memory that it occupies might consist of arbitrary bits (inherited from the previous counter for which the chunk was used). Overwriting these bits would require $\Theta(\log n)$ bit operations, but the algorithm only can manipulate $\Theta(\log \log n)$ bits per operation. Therefore, n_i is represented by some bits of chunk i and all other bits are considered as being zero (independently of what they actually are). The additional number a_i tells how many of the last bits of the chunk for n_i are valid for the representation of n_i. When chunk i is activated, n_i should be 0 and therefore a_i can be set to 0, signifying that all bits of n_i are zero. In each subsequent step, a_i is incremented by 2 and two additional positions in the chunk are set to zero until a_i equals the length of the bit string.

For m_i, we use a similar technique but the situation is slightly more complicated since m_i should be initially set to n. If the marked position of n is position k then both numbers b_i and d_i are set to k. This indicates that all positions of m_i up to b_i are as in n and all positions from d_i on are as in n. Which clearly means that m_i is n, as required. Subsequently, b_i is decremented by 2 in each step, d_i is incremented by 2 in each step and the respective bits are copied from n to the chunk. Note that these copied bits of n have not changed there values since chunk i has been activated (this holds since we copy 2 bits in each step).

C   Additional details on deterministic 1-counter automata

To extend the algorithm from Section 4.3 to DOCAs with ε-transitions we work with a slightly different, yet equivalent, definition of DOCAs from [7]: A *deterministic 1-counter automaton* is a tuple $\mathcal{A} = (Q_s, Q_r, \Sigma, \delta, \pi, \rho, q_0, F)$, where Q_s is a finite set of stable states, Q_r is a finite set of reset states ($Q_s \cap Q_r = \emptyset$), Σ is a finite input alphabet, $\delta : Q_s \times \Sigma \times \{0,1\} \to (Q_s \cup Q_r) \times \{-1, 0, 1\}$ is the transition function, $\pi : Q_r \to \mathbb{N}$ maps every reset state q to a period $\pi(q) > 0$, $\rho : \{(q, k) \mid q \in Q_r, 0 \le k < \pi(q)\} \to Q_s$ is the reset mapping, q_0 is the initial state, and $F \subseteq Q$ is the set of final states. It is required that if $\delta(q, a, i) = (q', j)$ then $i + j \ge 0$ to prevent the counter from becoming negative. Let $Q = Q_s \cup Q_r$. The set of configurations of $\mathcal{A}$ is $Q \times \mathbb{N}$. For a configuration (q, m) and $k \in \mathbb{N}$ we define $(q, m) + k = (q, m + k)$. Intuitively, $\mathcal{A}$ reads an input letter whenever it is in a stable state and changes its configuration according to δ. If $\mathcal{A}$ is in a reset state q it resets the counter to zero and goes into a stable state that is determined (via the reset mapping ρ) by $m \bmod \pi(q)$ when m is the current counter value. Formally, we define the mappings $\hat{\delta} : Q \times \mathbb{N} \times \Sigma \to Q \times \mathbb{N}$ and $\hat{\rho} : Q \times \mathbb{N} \to Q_s \times \mathbb{N}$ as follows, where $\mathsf{sign} : \mathbb{N} \to \{0, 1\}$ is the signum function restricted to the natural numbers:

- If $q \in Q_r$ and $m \in \mathbb{N}$ then $\hat{\rho}(q, m) = (\rho(q, m \bmod \pi(q)), 0)$.
- If $q \in Q_s$ and $m \in \mathbb{N}$ then $\hat{\rho}(q, m) = (q, m)$.
- If $q \in Q_s$ and $m \in \mathbb{N}$ then $\hat{\delta}(q, m) = \hat{\rho}(\delta(q, a, \mathsf{sign}(m)) + m)$.
- If $q \in Q_r$ and $m \in \mathbb{N}$ then $\hat{\delta}(q, m) = \hat{\delta}(\hat{\rho}(q, m))$ (note that $\hat{\rho}(q, m) \in Q_s \times \{0\}$, for which $\hat{\delta}$ has been defined in the previous point).

We extend $\hat{\delta}$ to a function $\hat{\delta} : Q \times \mathbb{N} \times \Sigma^* \to Q \times \mathbb{N}$ in the usual way: $\hat{\delta}(q, x, \varepsilon) = (q, x)$ and $\hat{\delta}(q, x, aw) = \hat{\delta}(\hat{\delta}(q, x, a), w)$. Then, $L(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, 0, w) \in F \times \mathbb{N}\}$ is the language accepted by $\mathcal{A}$. A language L is a deterministic 1-counter language if $L = L(\mathcal{A})$ for some deterministic 1-counter automaton $\mathcal{A}$.

Using the function $\hat{\delta}$ we can define also runs of $\mathcal{A}$ (we speak of $\mathcal{A}$-runs) on a word w in the usual way: For a configuration (q, m) and a word $w = a_1 a_2 \cdots a_k$ the unique $\mathcal{A}$-run on the word w starting in (q, m) is the sequence of configurations $(q_0, m_0), (q_1, m_1), \ldots, (q_k, m_k)$ where $(q_i, m_i) = \hat{\delta}(q_0, 0, a_1 \cdots a_i)$. We denote this run with $\mathsf{run}(q, m, w)$.

For a word $w \in \Sigma^*$ we define the *effect* $\hat{\delta}_w : Q \times \mathbb{N} \to Q \times \mathbb{N}$ by $\hat{\delta}_w(q, m) = \hat{\delta}(q, m, w)$. It specifies how w transforms configurations. It turns out that $\hat{\delta}_w$ can be completely reconstructed from restriction $\hat{\delta}_w {\upharpoonright}_{Q \times [0, |w| + p]}$ of $\hat{\delta}_w$ to the set $Q \times [0, |w| + p]$, where p is the least common multiple of all $\pi(q)$ for $q \in Q_r$. To see this, assume that (q, m) is a configuration with $m > |w| + p$. If in $\mathsf{run}(q, |w| + p, w)$ no state from Q_r is visited and $\hat{\delta}_w(q, |w| + p) = (q', m')$, then we have $\hat{\delta}_w(q, m) = (q', m' + m - |w| - p)$: basically, we obtain $\mathsf{run}(q, m, w)$ by shifting $\mathsf{run}(q, |w| + p, w)$ upwards by $m - |w| - p$. On the other hand, if a reset state that appears in $\mathsf{run}(q, |w| + p, w)$ then $\hat{\delta}_w(q, m) = \hat{\delta}_w(q, |w| + i)$, where i is any number in $[0, p]$ such that $|w| + i \equiv m \bmod p$.

In the following, we always assume that the effect $\hat{\delta}_w$ of a word w is stored by $\hat{\delta}_w {\upharpoonright}_{Q \times [0, |w| + p]}$, for which $\mathcal{O}(|w|)$ many registers of bit length $\mathcal{O}(\log |w|)$ suffice. Given the effects $\hat{\delta}_u$ and $\hat{\delta}_v$ of two words u, v of length at most n (stored in $\mathcal{O}(n)$ many registers of bit length $\mathcal{O}(\log n)$), we can compute the effect $\hat{\delta}_{uv}$ in time $\mathcal{O}(n)$ on a RAM with word size $\mathcal{O}(\log n)$. The computation of $\hat{\delta}_{uv}$ on an argument from $Q \times [0, |uv| + p]$ only involves simple arithmetic operations and can be done in constant time.

D   Proof of Claim 17

We show the claim for the case $a_k > a_{k+1}$ (the cases $a_k < a_{k+1}$ and $a_k = a_{k+1}$ can be treated analogously). The middle block B_k can be factorized into $2a_k$ completed blocks, since if we consider the binary tree T_k for B_k, then only the blocks on the left most and right most root-

leaf path of T_k can be non-completed. This follows from Claim 16 since each level-b subblock of B_k that does not belong the left-most or right-most path in T_k has another level-b subblock to its left as well as to its right. For the blocks $B_{k+1}, \ldots, B_m$ we show that $B_{k+1} \ldots B_m$ can be factorized into at most $a_{k+1} + m - k - 1 \in \mathcal{O}(\log n)$ completed blocks, whereas $B_0 \cdots B_{k-1}$ can be factorized into at most $a_{k-1} + k - 1 \in \mathcal{O}(\log n)$ completed blocks. Consider the level-a_i block B_i for some $1 \leq i \leq k - 1$. Every level-b subblock of B_i that does not belong to the left most path in T_i has at least $2^b - 1$ many symbols to its right (namely the block B_k) as well as to its left (namely another level-b subblock of B_i), and is therefore completed. But for blocks on the left most path in T_i the same is true when they belong to some level $b \leq a_{i-1}$, because then the block B_{i-1} is to its left. Hence, there are at most $a_i - a_{i-1}$ non-completed blocks in T_i and they form an initial part of the left most path. Removing those blocks from T_i leads to a factorization of B_i into at most $a_i - a_{i-1} + 1$ completed blocks. Finally, for the first block B_0 we obtain with the same argument a factorization into at most a_0 completed blocks. By summing over all block B_i ($0 \leq i \leq k - 1$) we obtain a factorization of $B_0 \cdots B_{k-1}$ into at most $a_0 + \sum_{1 \leq i \leq k-1}(a_i - a_{i-1} + 1) = a_{k-1} + k - 1$ completed blocks. For $B_{k+1} \cdots B_m$ we can argue analogously. The above arguments also show how to compute a factorization of the window into completed blocks. This factorization can be represented by a sequence of pointers to the tree nodes that correspond to the completed blocks in the factorization.