Sliding Window Algorithms for Regular Languages

Moses Ganardi, Danny Hucke, and Markus Lohrey

Universität Siegen
Department für Elektrotechnik und Informatik
Hölderlinstrasse 3
D-57076 Siegen, Germany
{ganardi,hucke,lohrey}@eti.uni-siegen.de

Abstract. This paper gives a survey on recent results for sliding window streaming algorithms for regular languages. Details can be found in the recent papers [18, 19].

Keywords: automata theory, streaming algorithms, sliding window algorithms, regular languages

1 Introduction

Streaming algorithms. Streaming algorithms [1] process an input sequence $a_1 a_2 \cdots a_m$ of data values from left to right. Random access to the input is not allowed, and at time instant t the algorithm has only direct access to the current data value a_t and its goal is to compute an output value $f(a_1 a_2 \cdots a_t)$ for a certain function f. During this process it is quite often infeasible and in many settings also not necessary to store the whole history $a_1 a_2 \cdots a_t$. Such a scenario arises for instance when searching in large databases (e.g., genome databases or web databases), analyzing internet traffic (e.g. click stream analysis), and monitoring networks. Ideally, a streaming algorithm works in constant space, in which case the algorithms is a deterministic finite automaton (DFA), but polylogarithmic space with respect to the input length might be acceptable, too.

The first papers on streaming algorithms as we know them today are usually attributed to Munro and Paterson [24] and Flajolet and Martin [16], although the principle idea goes back to the work on online machines by Hartmanis, Lewis and Stearns from the 1960's [23, 27]. Extremely influential for the area of streaming algorithms was the paper of Alon, Matias, and Szegedy [2] on computing frequency moments in the streaming model.

The sliding window model. The streaming model sketched above is also known as the *standard streaming model.* One missing aspect of the standard model is the fact that data items are usually no longer important after a certain time. For instance, in the analysis of a time series as it may arise in medical monitoring, web tracking, or financial monitoring, data items are usually outdated after a certain time. The *sliding window model* is an alternative streaming model that can capture this aspect. Two variants of the sliding window model can be found in the literature; see e.g. [3]:

- *Fixed-size model:* In this model the algorithm works on a sliding window of a certain fixed length n. While reading the input word $w = a_1 a_2 \cdots a_m$ symbol by symbol from left to right it has to output at every time instant $n \leq t \leq m$ a value $f(a_{t-n+1} \cdots a_t)$ that depends on the n last symbols. The number n is also called the *window size.*
- *Variable-size model:* Here, the sliding window $a_{t-n+1} a_{t-n+2} \cdots a_t$ is determined by an adversary. At every time instant the adversary can either remove the first data value from the sliding window (expiration of a value), or add a new data value at the right end (arrival of a new value).

In the seminal paper of Datar et al. [15], where the (fixed-size) sliding window model was introduced, the authors show how to maintain the number of 1's in a sliding window of size n over the alphabet $\{0, 1\}$ in space $\frac{1}{\varepsilon} \cdot \log^2 n$ if one allows a multiplicative error of $1 \pm \varepsilon$. A matching lower bound is proved as well in [15]. Following the work of Datar et al., a large number of papers that deal with the approximation of statistical data over sliding windows followed. Let us mention the work on computation of the variance and k-median [4], quantiles [3], and entropy [8] over sliding windows. Other computational problems that have been considered for the sliding window model include optimal sampling [9], various pattern matching problems [10–13], database querying (e.g. processing of join queries [20]) and graph problems (e.g. checking for connectivity and computation of matchings, spanners, and spanning trees [14]). Further references on the sliding window model can be found in the surveys [1, Chapter 8] and [7].

Language recognition in the streaming model. A natural problem that has been surprisingly neglected for the streaming model (in particular the sliding window model) is language recognition. The goal is to check whether an input string belongs to a given language L. Let us quote Magniez, Mathieu, and Nayak [22]: "Few applications [of streaming] have been made in the context of formal languages, which may have impact on massive data such as DNA sequences and large XML files. For instance, in the context of databases, properties decidable by streaming algorithm have been studied [26, 25], but only in the restricted case of deterministic and constant memory space algorithms." For Magniez et al. this was the starting point to study language recognition in the streaming model. Thereby they restricted their attention to the above mentioned standard streaming model. Note that in the standard model the membership problem for a regular language is trivial to solve: One simply has to simulate a DFA on the stream and thereby only store the current state of the DFA. In [22] the authors presented a randomized streaming algorithm for the (non-regular) Dyck language D_s with s pairs of parenthesis that works in space $\mathcal{O}(\sqrt{n} \log n)$ and time polylog(n) per symbol. The algorithm has a one-sided error: it accepts with small probability also words that do not belong to D_s. An almost matching lower bound of $\Omega(\sqrt{n \log n})$ for two-sided errors is proved in [22] as well. Further investigations on streaming language recognition for various subclasses of context-free languages can be found in [5, 6, 17, 21].

Let us emphasize that all the papers cited in the last paragraph exclusively deal with the standard streaming model. Language recognition problems for the sliding window model have been completely neglected so far. This is surprising, since even for regular languages the membership problem becomes non-trivial in the sliding window model. This was the starting point for our work on streaming [18, 19] that mainly deals with the membership problem for regular languages in the sliding window model. Before we explain the results from [18, 19] in Section 4, we formally define the various streaming models in the next section.

2 Streaming Algorithms as Automata

We use standard definitions from automata theory. A *nondeterministic finite automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$ where Q is a finite set of states, Σ is an alphabet, $I \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation and $F \subseteq Q$ is the set of final states. A *deterministic finite automaton* (DFA) $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ has a single initial state $q_0 \in Q$ instead of I and a transition function $\delta\colon Q \times \Sigma \to Q$ instead of the transition relation Δ. A *deterministic automaton* has the same format as a DFA, except that the state set Q is not required to be finite. If $\mathcal{A}$ is deterministic, the transition function δ is extended to a function $\delta\colon Q \times \Sigma^* \to Q$ in the usual way and we define $\mathcal{A}(x) = \delta(q_0, x)$ for $x \in \Sigma^*$. The language accepted by $\mathcal{A}$ is $L(\mathcal{A}) = \{w \in \Sigma^* \colon \delta(q_0, w) \in F\}$.

Recall from the introduction that a streaming algorithm reads an input word $w = a_1 a_2 \cdots a_m$ from left to right and computes at every time instant $0 \le t \le m$ a value $f(a_1 a_2 \cdots a_t)$ for some target function f. In this paper we make two restrictions:

- The data values a_i are from some finite alphabet Σ. This rules out streaming algorithms that read for instance a natural number in each time unit.
- The target function is boolean-valued, i.e., $f : \Sigma^* \to \{0, 1\}$.

These two restrictions imply that a streaming algorithm can be seen as a deterministic automaton, possibly with an infinite state set. Moreover, in order to make statements about the space complexity of a streaming algorithm, we also have to fix an encoding of the automaton states by bit strings. Formally, a *streaming algorithm* over Σ is a deterministic (possibly infinite) automaton $\mathcal{A} = (S, \Sigma, s_0, \delta, F)$, where the states are encoded by bit strings. We describe this encoding by an injective function $\mathrm{enc}\colon S \to \{0,1\}^*$. The *space function* $\mathrm{space}(\mathcal{A}, \cdot)\colon \Sigma^* \to \mathbb{N}$ specifies the space used by $\mathcal{A}$ on a certain input: For $w \in \Sigma^*$ let $\mathrm{space}(\mathcal{A}, w) = \max\{|\mathrm{enc}(\mathcal{A}(u))| : u \in \mathrm{Pref}(w)\}$, where $\mathrm{Pref}(w)$ denotes the set of prefixes of w. We also say that $\mathcal{A}$ is a *streaming algorithm for* the accepted language $L(\mathcal{A})$.

3 Sliding Window Streaming Models

In the above streaming model, the output value of the streaming algorithm at time t depends on the whole past $a_1 a_2 \cdots a_t$ of the data stream. However, in many practical applications one is only interested in the relevant part of the past. Two formalizations of "relevant past" can be found in the literature:

- Only the suffix of $a_1 a_2 \cdots a_t$ of length n is relevant. Here, n is a fixed constant. This streaming model is called the *fixed-size sliding window model.*
- The relevant suffix of $a_1 a_2 \cdots a_t$ is determined by an adversary. In this model, at every time instant the adversary can either remove the first symbol from the active window (expiration of a data value), or add a new symbol at the right end (arrival of a new data value). This streaming model is also called the *variable-size sliding window model.*

In the following two subsections, we formally define these two models.

3.1 Fixed-Size Sliding Windows

Given a word $w = a_1 a_2 \cdots a_m \in \Sigma^*$ and a window length $n \geq 0$, we define $\mathrm{last}_n(w) \in \Sigma^n$ by

$$\mathrm{last}_n(w) = \begin{cases} a_{m-n+1} a_{m-n+2} \cdots a_m, & \text{if } n \leq m, \\ \Box^{n-m} a_1 \cdots a_m, & \text{if } n > m, \end{cases}$$

which is called the *active window.* Here $\Box \in \Sigma$ is an arbitrary symbol, which fills the initial window. A sequence $\mathcal{A} = (\mathcal{A}_n)_{n \geq 0}$ is a *fixed-size sliding window algorithm* for a language $L \subseteq \Sigma^*$ if each $\mathcal{A}_n$ is a streaming algorithm for the language

$$L_n := \{w \in \Sigma^* : \mathrm{last}_n(w) \in L\}.$$

Its *space complexity* is the function $f_{\mathcal{A}} \colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ where $f_{\mathcal{A}}(n)$ is the maximal encoding length of a state in $\mathcal{A}_n$. Note that for every language L and every n the language L_n is regular, which ensures that $\mathcal{A}_n$ can be chosen to be a DFA and hence $f_{\mathcal{A}}(n) < \infty$ for all $n \geq 0$.

A trivial fixed-size sliding window algorithm $\mathcal{B} = (\mathcal{B}_n)_{n \geq 0}$ for L is obtained by taking the DFAs $\mathcal{B}_n = (\Sigma^n, \Sigma, \Box^n, \delta_n, \Sigma^n \cap L)$ with the transition function $\delta_n(au, b) = ub$ for $a, b \in \Sigma$, $u \in \Sigma^{n-1}$. It stores the active window explicitly in the state. States of $\mathcal{B}_n$ can be encoded with $\mathcal{O}(\log |\Sigma| \cdot n)$ bits. By minimizing each $\mathcal{B}_n$, we obtain an *optimal fixed-size sliding window algorithm* $\mathcal{A}_L$ for L. Finally, we define $F_L(n) = f_{\mathcal{A}_L}(n)$. Thus, F_L is the space complexity of an optimal fixed-size sliding window algorithm for L. Notice that F_L is not necessarily monotonic. For instance, take $L = \{au \colon u \in \{a,b\}^*, |u| \text{ odd}\}$. Then, we have $F_L(2n) \in \Theta(n)$ (see Example 5 below) and $F_L(2n+1) \in \mathcal{O}(1)$. The above trivial algorithm $\mathcal{B}$ yields $F_L(n) \in \mathcal{O}(n)$ for every language L.

Note that the fixed-size sliding window model is a *non-uniform* model: for every window size we have a separate streaming algorithm and these algorithms do not have to follow a common pattern. Working with a non-uniform model makes lower bounds stronger. In contrast, the variable-size sliding window model that we discuss next is a uniform model in the sense that there is a single streaming algorithm that works for every window length.

3.2 Variable-Size Sliding Windows

For an alphabet Σ we define the extended alphabet $\overline{\Sigma} = \Sigma \cup \{\downarrow\}$. In the variable-size model the *active window* $\mathrm{wnd}(u) \in \Sigma^*$ for a stream $u \in \overline{\Sigma}^*$ is defined as follows:

- $\mathrm{wnd}(\varepsilon) = \varepsilon$
- $\mathrm{wnd}(ua) = \mathrm{wnd}(u)\, a$ for $a \in \Sigma$
- $\mathrm{wnd}(u\downarrow) = \varepsilon$ if $\mathrm{wnd}(u) = \varepsilon$
- $\mathrm{wnd}(u\downarrow) = v$ if $\mathrm{wnd}(u) = av$ for $a \in \Sigma$

A *variable-size sliding window algorithm* for a language $L \subseteq \Sigma^*$ is a streaming algorithm $\mathcal{A}$ for $\{w \in \overline{\Sigma}^* : \mathrm{wnd}(w) \in L\}$. Its *space complexity* is the function $v_{\mathcal{A}} \colon \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ mapping each window length n to the maximum number of bits used by $\mathcal{A}$ on inputs producing an active window of size at most n. Formally, it is the function

$$v_{\mathcal{A}}(n) = \max\{\mathrm{space}(\mathcal{A}, u) : u \in \overline{\Sigma}^*, |\mathrm{wnd}(v)| \leq n \text{ for all } v \in \mathrm{Pref}(u)\}.$$

Note that $v_{\mathcal{A}}$ is a monotonic function. It is not completely obvious that every language L has an optimal variable-size sliding window algorithm:

Lemma 1. *For every language $L \subseteq \Sigma^*$ there exists a variable-size sliding window algorithm $\mathcal{A}$ such that $v_{\mathcal{A}}(n) \leq v_{\mathcal{B}}(n)$ for every variable-size sliding window algorithm $\mathcal{B}$ for L and every n.*

We define $V_L(n) = v_{\mathcal{A}}(n)$, where $\mathcal{A}$ is a space *optimal variable-size sliding window algorithm* for L from Lemma 1. Since any algorithm in the variable-size model yields an algorithm in the fixed-size model, we have $F_L(n) \leq V_L(n)$.

It is not hard to show that any variable-size sliding window algorithm for a non-trivial language has to store enough information to recover the length of the active window. Hence, we have:

Lemma 2. *For every language $L \subseteq \Sigma^*$ such that $\emptyset \neq L \neq \Sigma^*$ we have $V_L(n) \in \Omega(\log n)$.*

4 Sliding Window Algorithms for Regular Languages

4.1 Space Trichotomy for Regular Languages

The main result from [19] is a space trichotomy for regular languages with respect to the two sliding window models: For every regular language, the space is either constant, logarithmic or linear.

Theorem 3 (space trichotomy [19]). *For every regular language L, exactly one of the following three cases holds:*

1. $F_L(n) \in \mathcal{O}(1)$
2. $F_L(n) \in \mathcal{O}(\log n) \setminus o(\log n)$ *and* $V_L(n) \in \Theta(\log n)$
3. $F_L(n) \in \mathcal{O}(n) \setminus o(n)$ *and* $V_L(n) \in \Theta(n)$

Note in particular that the class of regular languages that need logarithmic space (resp., linear space) is the same for the fixed-size model and the variable-size model. This is not true for constant space: For instance, for the language $L_1 = \{a, b\}^*a$ we have $F_{L_1}(n) \in \mathcal{O}(1)$, whereas Lemma 2 implies that $V_{L_1}(n) \in \Omega(\log n)$. Here are two further examples:

Example 4. For the language $L_2 = \{a, b\}^*a\{a, b\}^*$ we have $F_{L_2}(n) \in \Theta(\log n)$ as well as $V_{L_2}(n) \in \Theta(\log n)$. A variable-size sliding window algorithm for L_2 stores (i) the length of the active window and (ii) the position of the right-most a in the active window (or ∞ if the active window does not contain an a). For this, $\mathcal{O}(\log n)$ bits are sufficient. To see that $V_{L_2}(n) \in \Omega(\log n)$ consider a fixed-size sliding window algorithm $\mathcal{A} = (\mathcal{A}_n)_{n\geq 0}$ for L_2. Consider the window length n and all strings $w_i = b^{i-1}ab^{n-i}$ for $1 \leq i \leq n$. Then a standard fooling argument shows that for $1 \leq i < j \leq n$ the words w_i and w_j must lead to different states in $\mathcal{A}_n$. Hence, $\mathcal{A}_n$ has at least n states, which implies that $f_{\mathcal{A}}(n) \in \Omega(\log n)$.

Example 5. For the language $L_3 = a\{a, b\}^*$ we have $F_{L_3}(n) \in \Theta(n)$ as well as $V_{L_2}(n) \in \Theta(n)$. It suffices to show the lower bound $F_{L_3}(n) \in \Omega(n)$. This follows from a fooling argument similar to the one from Example 4: Consider a fixed-size sliding window algorithm $\mathcal{A} = (\mathcal{A}_n)_{n\geq 0}$ for L_3. Consider the window length n. Then, all words from Σ^n have to lead to different states of $\mathcal{A}_n$, i.e., $\mathcal{A}_n$ has at least $|\Sigma|^n$ states which implies that $f_{\mathcal{A}}(n) \in \Omega(n)$.

The reader might wonder why we write $F_L(n) \in \mathcal{O}(\log n) \setminus o(\log n)$ (resp., $F_L(n) \in \mathcal{O}(n) \setminus o(n)$) instead of $F_L(n) \in \Theta(\log n)$ (resp., $F_L(n) \in \Theta(n)$) in point 2 (resp., point 3) of Theorem 3. To see that this is indeed necessary, consider again the language $L = \{au \colon u \in \{a, b\}^*, |u| \text{ odd}\}$. Then, we have $F_L(2n) \in \mathcal{O}(n) \setminus o(n)$, but $F_L(n) \notin \Omega(n)$, since $F_L(2n + 1) \in \mathcal{O}(1)$. On the other hand, for the variable-size model, we can make the stronger statement $V_L(n) \in \Theta(\log n)$ (resp., $V_L(n) \in \Theta(n)$) due to the monotonicity of $V_L(n)$.

4.2 Characterizations of the Space Classes

We use the following notation for the three classes from Theorem 3:

- Reg(1) is the class of all regular languages for which point 1 from Theorem 3 holds.
- Reg($\log n$) is the class of all regular languages for which point 2 from Theorem 3 holds.
- Reg(n) is the class of all regular languages for which point 3 from Theorem 3 holds.

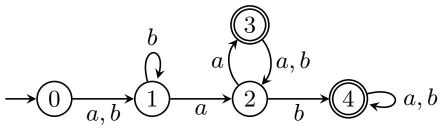

Fig. 1. A well-behaved DFA.

Theorem 3 does not give language theoretical characterizations of the above three classes. Such characterizations were provided in [18]. We need the following definitions.

A language $L \subseteq \Sigma^*$ is called *k-suffix testable* if for all $x, y \in \Sigma^*$ and $z \in \Sigma^k$ we have: $xz \in L$ if and only if $yz \in L$. Equivalently, L is a Boolean combination of languages of the form $\Sigma^* w$ where $w \in \Sigma^{\leq k}$. We call L *suffix testable* if it is k-suffix testable for some $k \geq 0$. Clearly, every finite language is suffix testable: if $L \subseteq \Sigma^{\leq k}$ then L is $(k+1)$-suffix testable. Moreover, every suffix testable language is regular. A language $L \subseteq \Sigma^*$ is called a *length language* if for all $n \in \mathbb{N}$, either $\Sigma^n \subseteq L$ or $L \cap \Sigma^n = \emptyset$.

Theorem 6 ([18]). Reg(1) *is the class of all finite Boolean combination of suffix testable languages and regular length languages.*

In order to characterize the class Reg($\log n$) we need the following definition: A language $L \subseteq \Sigma^*$ is called a *left ideal* if $\Sigma^* L \subseteq L$.

Theorem 7 ([18]). Reg($\log n$) *is the class of all finite Boolean combination of regular left ideals and regular length languages.*

The class Reg($\log n$) has a useful characterization in terms of automata as well. A *strongly connected component* (SCC for short) of a DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ is an inclusion-maximal subset $C \subseteq Q$ such that for all $p, q \in C$ there exist words $u, v \in \Sigma^*$ such that $\delta(p, u) = q$ and $\delta(q, v) = p$. A singleton SCC $\{q\}$ is called trivial if $\delta(q, u) \neq q$ for all non-empty words u (i.e., q is not on a cycle). An SCC $C \subseteq Q$ is *well-behaved* if for all $q \in C$ and $u, v \in \Sigma^*$ with $|u| = |v|$ and $\delta(q, u), \delta(q, v) \in C$ we have: $\delta(q, u) \in F$ if and only if $\delta(q, v) \in F$. Clearly, every trivial SCC is well-behaved. If every SCC in $\mathcal{A}$ which is reachable from q_0 is well-behaved, then $\mathcal{A}$ is called *well-behaved*. Figure 1 shows an example of a well-behaved DFA. Its SCCs are $\{0\}$ (which is trivial), $\{1\}$, $\{2, 3\}$, and $\{4\}$.

For a word $w = a_1 a_2 \cdots a_n$, let $w^{\text{rev}} = a_n \cdots a_2 a_1$ be the reversed word.

Theorem 8 ([18]). *A regular language L belongs to* Reg($\log n$) *if and only if the reversed language $L^{\text{rev}} = \{w^{\text{rev}} \colon w \in L\}$ is accepted by a well-behaved DFA. Moreover, this holds if and only if every DFA for L^{rev} is well-behaved.*

Let us sketch the logspace (variable-size) sliding-window algorithm for a regular language L such that L^{rev} is accepted by a well-behaved DFA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$. Let w be the current active window. Assume that we store for every state $q \in Q$ the run of $\mathcal{A}$ on the word w^{rev} (i.e., the sequence of visited states) that starts in q.

This information would allow to make the necessary updates and queries for the variable-size model (i.e., removing the left-most symbol from the window, adding a symbol on the right, and testing membership in L). But the space needed to store these runs would be linear in the length w. The main observation for the logspace algorithm is that since $\mathcal{A}$ is well-behaved it suffices to store for each of the above runs a so called path summary that is defined as follows: Let $C_1, \dots, C_k$ be the sequence of pairwise different SCCs that are visited by the run in that order. The path summary of the run is the sequence $(q_1, \ell_1, q_2, \ell_2, \dots, q_k, \ell_k)$ where q_i is the first state in C_i visited by ρ, and $\ell_i \geq 0$ is the number of transitions from the first occurrence of q_i until the first state from C_{i+1} (or until the end for q_k).

The lower bound $\Omega(n)$ for the space complexity (with respect to the fixed-size model) in case L^{rev} is accepted by a DFA that is not well-behaved can be shown by a fooling argument similar to the one from Example 5.

4.3 Uniform Space Bounds

In the statements from Sections 4.1 and 4.2 we always assume a fixed regular language. When, e.g., saying that $V_L \in \mathcal{O}(\log n)$ then the $\mathcal{O}$-constant depends on the automaton size. Using the path summaries mentioned in Sections 4.2, one can show:

Theorem 9 ([18]). *Let $\mathcal{A}$ be a DFA or NFA with m states such that $L = L(\mathcal{A}) \in \mathrm{Reg}(\log n)$ is well-behaved. There are constants c_m, d_m that only depend on m such that the following holds:*

- *If $\mathcal{A}$ is a DFA then $V_L(n) \leq (2^m \cdot m + 1) \cdot \log n \; + \; c_m$ for n large enough.*
- *If $\mathcal{A}$ is an NFA then $V_L(n) \leq (4^m + 1) \cdot \log n \; + \; d_m$ for n large enough.*

The following theorem states a lower bound for the fixed-size model (and hence also for the variable-size model) that almost matches the space bound in Theorem 9:

Theorem 10 ([18]). *For all $k \geq 1$ there exists a language $L_k \subseteq \{0, \dots, k\}^*$ recognized by a DFA with $k+3$ states such that $L_k \in \mathrm{Reg}(\log n)$ and $F_{L_k}(n) \geq (2^k - 1) \cdot (\log n - k)$.*

It is open whether in Theorem 10 the alphabet $\{0, \dots, k\}$ can be replaced by a fixed (e.g. binary) alphabet without changing the lower bound.

4.4 Deciding the Space Classes

Theorem 8 leads to a decision algorithm for the class $\mathrm{Reg}(\log n)$: Given a DFA (or NFA) for a regular language L, one first constructs a DFA $\mathcal{A}$ for L^{rev} using standard automata constructions and then checks whether $\mathcal{A}$ is well-behaved. But this algorithm is not very efficient since in general the size of a DFA for L^{rev} is exponential in the size of an automaton for L, even if the latter automaton is deterministic. In [18] we provided a more efficient algorithm for the class $\mathrm{Reg}(\log n)$ as well as $\mathrm{Reg}(1)$ in case the input automaton is deterministic.

Theorem 11 ([18]). *Given a DFA for a regular language L, it is* NL*-complete to check whether $L \in \mathrm{Reg}(\log n)$, respectively $L \in \mathrm{Reg}(1)$.*

As one might expect, if the input automaton is nondeterministic than the complexity increases by one exponent:

Theorem 12 ([18]). *Given an NFA for a regular language L, it is* PSPACE*-complete to check whether $L \in \mathrm{Reg}(\log n)$, respectively $L \in \mathrm{Reg}(1)$.*

5 Future Work

The space trichotomy theorem for regular languages (Theorem 3) leads to several interesting research questions:

- Do similar results hold for context-free languages or subclasses like deterministic context-free languages or visibly pushdown languages?
- Is it possible to generalize Theorem 3 to a randomized setting? In fact, most papers on streaming algorithms deal with randomized streaming algorithms.

These topics will be the content of two forthcoming papers.

References

1. Aggarwal, C.C.: Data Streams - Models and Algorithms. Springer (2007)
2. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. 58(1), 137–147 (1999)
3. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: Proceedings of PODS 2004. pp. 286–296. ACM (2004)
4. Babcock, B., Datar, M., Motwani, R., O'Callaghan, L.: Maintaining variance and k-medians over data stream windows. In: Proceedings of PODS 2003. pp. 234–243. ACM (2003)
5. Babu, A., Limaye, N., Radhakrishnan, J., Varma, G.: Streaming algorithms for language recognition problems. Theor. Comput. Sci. 494, 13–23 (2013)
6. Babu, A., Limaye, N., Varma, G.: Streaming algorithms for some problems in log-space. In: Proceedings of TAMC 2010. Lecture Notes in Computer Science, vol. 6108, pp. 94–104. Springer (2010)
7. Braverman, V.: Sliding window algorithms. In: Encyclopedia of Algorithms, pp. 2006–2011. Springer (2016)
8. Braverman, V., Ostrovsky, R.: Smooth histograms for sliding windows. In: Proceedings of FOCS 2007. pp. 283–293. IEEE Computer Society (2007)
9. Braverman, V., Ostrovsky, R., Zaniolo, C.: Optimal sampling from sliding windows. J. Comput. Syst. Sci. 78(1), 260–272 (2012)
10. Breslauer, D., Galil, Z.: Real-time streaming string-matching. ACM Trans. Algorithms 10(4), 22:1–22:12 (2014)
11. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: Dictionary matching in a stream. In: Proceedings of ESA 2015. Lecture Notes in Computer Science, vol. 9294, pp. 361–372. Springer (2015)

12. Clifford, R., Fontaine, A., Porat, E., Sach, B., Starikovskaya, T.A.: The k-mismatch problem revisited. In: Proceedings of SODA 2016. pp. 2039–2052. SIAM (2016)
13. Clifford, R., Starikovskaya, T.A.: Approximate hamming distance in a stream. In: Proceedings of ICALP 2016. LIPIcs, vol. 55, pp. 20:1–20:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
14. Crouch, M.S., McGregor, A., Stubbs, D.: Dynamic graphs in the sliding-window model. In: Proceedings of ESA 2013. Lecture Notes in Computer Science, vol. 8125, pp. 337–348. Springer (2013)
15. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM J. Comput. 31(6), 1794–1813 (2002)
16. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. J. Comput. Syst. Sci. 31(2), 182–209 (1985)
17. François, N., Magniez, F., de Rougemont, M., Serre, O.: Streaming property testing of visibly pushdown languages. In: Proceedings of ESA 2016. LIPIcs, vol. 57, pp. 43:1–43:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
18. Ganardi, M., Hucke, D., König, D., Lohrey, M., Mamouras, K.: Automata theory on sliding windows. In: Proceedings of STACS 2018. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018), to appear
19. Ganardi, M., Hucke, D., Lohrey, M.: Querying regular languages over sliding windows. In: Proceedings of FSTTCS 2016. LIPIcs, vol. 65, pp. 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
20. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Proceedings of VLDB 2003. pp. 500–511. Morgan Kaufmann (2003)
21. Krebs, A., Limaye, N., Srinivasan, S.: Streaming algorithms for recognizing nearly well-parenthesized expressions. In: Proceedings of MFCS 2011. Lecture Notes in Computer Science, vol. 6907, pp. 412–423. Springer (2011)
22. Magniez, F., Mathieu, C., Nayak, A.: Recognizing well-parenthesized expressions in the streaming model. SIAM J. Comput. 43(6), 1880–1905 (2014)
23. Lewis II, P.M., Stearns, R.E., Hartmanis, J.: Memory bounds for recognition of context-free and context-sensitive languages. In: Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design. pp. 191–202. IEEE Computer Society (1965)
24. Munro, J.I., Paterson, M.: Selection and sorting with limited storage. Theor. Comput. Sci. 12, 315–323 (1980)
25. Segoufin, L., Sirangelo, C.: Constant-memory validation of streaming XML documents against dtds. In: Proceedings of ICDT 2007. Lecture Notes in Computer Science, vol. 4353, pp. 299–313. Springer (2007)
26. Segoufin, L., Vianu, V.: Validating streaming XML documents. In: Proceedings of PODS 2002. pp. 53–64. ACM (2002)
27. Stearns, R.E., Hartmanis, J., Lewis II, P.M.: Hierarchies of memory limited computations. In: Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design. pp. 179–190. IEEE Computer Society (1965)