Faster Sliding Window String Indexing in Streams

Philip Bille
Technical University of Denmark, Lyngby, Denmark

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Inge Li Gørtz
Technical University of Denmark, Lyngby, Denmark

Simon R. Tarnow
Technical University of Denmark, Lyngby, Denmark

Abstract

The classical string indexing problem asks to preprocess the input string S for efficient pattern matching queries. Bille, Fischer, Gørtz, Pedersen, and Stordalen [CPM 2023] generalized this to the streaming sliding window string indexing problem, where the input string S arrives as a stream, and we are asked to maintain an index of the last w characters, called the window. Further, at any point in time, a pattern P might appear, again given as a stream, and all occurrences of P in the current window must be output. We require that the time to process each character of the text or the pattern is worst-case. It appears that standard string indexing structures, such as suffix trees, do not provide an efficient solution in such a setting, as to obtain a good worst-case bound, they necessarily need to work right-to-left, and we cannot reverse the pattern while keeping a worst-case guarantee on the time to process each of its characters. Nevertheless, it is possible to obtain a bound of $\mathcal{O}(\log w)$ (with high probability) by maintaining a hierarchical structure of multiple suffix trees.

We significantly improve this upper bound by designing a black-box reduction to maintain a suffix tree under prepending characters to the current text. By plugging in the known results, this allows us to obtain a bound of $\mathcal{O}(\log\log w + \log\log\sigma)$ (with high probability), where σ is the size of the alphabet. Further, we introduce an even more general problem, called the streaming dynamic window string indexing, where the goal is to maintain the current text under adding and deleting characters at either end and design a similar black-box reduction.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases data structures, pattern matching, string indexing

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.8

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).
Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFF-9131-00069B and 10.46540/3105-00302B).

1 Introduction

The *string indexing problem* is to preprocess a string S into a compact data structure that supports efficient subsequent pattern matching queries, that is, given a pattern string P, report all occurrences of P within S. Bille, Fischer, Gørtz, Pedersen, and Stordalen [6] introduced a variant of the string indexing problem, called the *streaming sliding window string indexing (SSWSI) problem*, where S arrives as a stream one character at a time. Here, we want to maintain an index of a *window* of the last w character for a specified parameter w. At any point in time, a pattern matching query for a pattern P may arrive also streamed one character at a time, and we need to report the occurrences of P within the current window.

We measure the complexity of the algorithm by the worst-case time it processes a single character of the text or pattern. The goal is to compactly maintain the index while processing the characters arriving from either S or a pattern query efficiently. The SSWSI problem captures scenarios where we want to index recent data in an incoming stream (the window) while supporting fast pattern matching queries. For instance, monitoring a high-speed data stream, where we cannot afford to index the entire stream but still want to support fast queries.

As discussed in Bille, Fischer, Gørtz, Pedersen, and Stordalen [6], the standard string indexing structures, such as *sliding suffix tree* [8,12,20–22] and *online suffix tree* [1–3,7,14,17–19] constructions, do not provide an efficient solution to the SSWSI problem. For instance, efficient online suffix tree constructions require that we process the string (and hence also the pattern) in right-to-left order. In our setting we cannot afford to reverse pattern while keeping a worst-case guarantee on the time to process each of its characters. Bille, Fischer, Gørtz, Pedersen, and Stordalen [6] showed how achieve $\mathcal{O}(\log w)$ (with high probability) time per character by maintaining a hierarchical structure of multiple suffix trees.

In this paper, we present a new black-box reduction to online suffix tree construction algorithms, i.e., algorithms that maintain suffix trees while prepending one character at a time to the current text. By plugging in known results, we obtain solutions using either $\mathcal{O}(\log\log w + \log\log\sigma)$ time (with high probability) or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ (deterministic) time per character. Here, σ is the size of the alphabet. We also consider a generalized version of this problem, called the *streaming dynamic window string indexing (SDWSI) problem.* Here, the window is a dynamic string that can be updated by adding or deleting characters at either end of the string, and we have to support streamed pattern matching queries as above. We show how to extend our reduction and results for this problem, and obtain similar bounds.

1.1 Setup

We now formally define the streaming dynamic window string indexing and streaming sliding window string indexing problems and our main results.

Streaming Dynamic Window String Indexing. Let S be a dynamic string over an alphabet Σ. The *streaming dynamic window string indexing (SDWSI) problem* is to maintain a data structure on S that supports the following operations:

- AddRight(a): add the character a to the right end of S.
- AddLeft(a): add the character a to the left end of S.
- RemoveRight(): remove the last character from S.
- RemoveLeft(): remove the first character from S.
- Report(P) report all the occurrences of P in S.

In the Report(P) query, the pattern string P is streamed one character at a time from left-to-right and the goal is to begin reporting occurrences immediately after receiving the last character. We do not assume that we know the length P before the arrival of its last character.

Streaming Sliding Window String Indexing. Given an integer parameter $w \geq 1$, we define the *streaming sliding window string indexing (SSWSI) problem* as above, except that we support a restricted set of operations:

- Report(P) report all the occurrences of P in S.
- Update(a): AddRight(a). If $|S|$ is now greater than w also perform a RemoveLeft().

Thus, except for the first w Update operations, the window always has size w and changes only by "sliding" one character to the right. This is also called the *timely streaming sliding window string indexing problem* in Bille et al. [6].

Online Suffix Trees and Dynamic Dictionaries. Our main results use online suffix tree construction algorithms and dynamic dictionaries as a black box. We define the precise requirements for these. Let R be a string of length r over an alphabet of size σ and T_i be the suffix tree of $R[i..r]$. An *online suffix tree construction algorithm* processes R from right to left such that at the ith step, the algorithm explicitly constructs T_i and returns a pointer to the new leaf ℓ corresponding to suffix $R[i..r]$, the parent of ℓ, and the edge between ℓ and the parent. We will use the currently best known algorithms for online suffix tree construction due to Kopelowitz [17] and Fischer and Gawrychowski [14].

▶ **Lemma 1** ([14, 17]). *Given a string R of length r over an alphabet of size σ, we can solve online suffix tree in linear space using either $\mathcal{O}(\log\log r + \log\log\sigma)$ time with high probability*[1] *or $\mathcal{O}(\log\log r + \frac{(\log\log\sigma)^2}{\log\log\log\sigma})$ (deterministic) time per character, respectively.*

Let X be a set of x integers from a universe of size u. A *dynamic dictionary structure* on X supports membership (i.e., determine if a given integer is in X or not), insert, and delete on X. We use the following results.

▶ **Lemma 2.** *A set $X \subseteq [U]$ can be maintained in a linear space dynamic dictionary structure that uses either $\mathcal{O}(1)$ time with high probability or $\mathcal{O}(\frac{(\log\log U)^2}{\log\log\log U})$ (deterministic) time.*

Proof. The first bound is obtained by using a dynamic hash table [10]. The second bound follows from a result of Andersson and Thorup [4]. ◀

We will maintain a dynamic dictionary structure $D(v)$ for every explicit node v of the current suffix tree T_i. $D(v)$ maps the first character on an edge to the edge, which allows us to navigate down in T_i to find the (implicit or explicit) node corresponding to $P[1..i]$, for $i = 1, 2, \ldots, m$, in either $\mathcal{O}(1)$ time with high probability or $\mathcal{O}\left(\frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ (deterministic) time per character of P.

1.2 Results

We can now define our main results. Let $t_{\text{suff}}(r, \sigma)$ denote the time per character of a linear space online suffix tree construction algorithm on a string of length r over an alphabet σ. Also, let $t_{\text{dict}}(x, u)$ denote the time per operation of a linear space dynamic dictionary structure. We show the following result for streaming sliding window string indexing.

▶ **Theorem 3.** *Let S be a string over an alphabet of size σ. Given an integer parameter $w \geq 1$ we can solve the streaming sliding window string indexing problem on S for a window of size w with an $\mathcal{O}(w)$ space data structure that supports Update and Report in $\mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ time per character. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

Plugging in Lemmas 1 and 2 in Theorem 3 we obtain the following bounds:

[1] Kopelowitz [17] claims only worst-case expected time, but the expectation is due to hash tables, so one can plug in e.g. the construction of Dietzfelbinger and auf der Heide [10].

▶ **Corollary 4.** *Let S be a string over an alphabet of size σ. Given an integer parameter $w \geq 1$ we can solve the streaming sliding window string indexing problem on S for a window of size w with an $\mathcal{O}(w)$ space data structure that supports* Update *and* Report *in either $\mathcal{O}(\log\log w + \log\log\sigma)$ time per character with high probability or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ deterministic time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

For the streaming dynamic window string indexing we show the following result.

▶ **Theorem 5.** *Let S be a dynamic string of length w over an alphabet σ. We can solve the streaming dynamic window string indexing problem on S with an $\mathcal{O}(w \cdot t_{\text{suff}}(w,\sigma))$ space data structure that supports* AddRight, AddLeft, RemoveRight, RemoveLeft *and* Report *in $\mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

Again, plugging in Lemmas 1 and 2 in Theorem 5 we obtain the following bounds:

▶ **Corollary 6.** *Let S be a dynamic string of length w over an alphabet σ. We can solve the streaming dynamic window string indexing problem on S with an $\mathcal{O}(w \cdot t_{\text{suff}}(w,\sigma))$ space data structure that supports* AddRight, AddLeft, RemoveRight, RemoveLeft *and* Report *in either $\mathcal{O}(\log\log w + \log\log\sigma)$ time per character with high probability or $\mathcal{O}\left(\log\log w + \frac{(\log\log\sigma)^2}{\log\log\log\sigma}\right)$ deterministic time per character. Furthermore,* Report *uses additional worst-case constant time per reported occurrence.*

1.3 Techniques

We first show how to use an online suffix tree construction algorithm to solve the version where the string only changes by appending characters to the right. As noted, existing fast algorithms for this problem work from right to left by prepending characters and then reverse patterns to do a pattern matching query. We cannot do this efficiently in our scenario since we want fast per character processing and we do not know the length of the pattern ahead of time. To overcome this, we construct the online suffix tree over the reverse string and then answer a pattern matching query P by prepending the characters of P to the string as we receive them. Thus after receiving all of P the suffix tree contains all suffixes of $\mathsf{rev}(P)\$\mathsf{rev}(S)$. When we receive the last character from the pattern, we determine if there is an occurrence by checking if the edge in the suffix tree created by prepending the last character starts with a \$. Finally, we return the state of the online suffix tree before the query. To quickly return the state of the online suffix tree to the state before the query, we use techniques from persistent data structures.

To solve the streaming sliding window string indexing problem, we use the above data structure over the last part of the window and a static suffix tree with a range maximum query data structure over the first part of the window. The two data structures always overlap by $w/3$. We can then answer pattern matching queries by querying each of these structures. To find occurrences of long patterns not covered by any of the structures, we give an algorithm that can report all occurrences of a pattern P in a text of length $\mathcal{O}(|P|)$ in constant time per streamed character in P.

We solve the streaming dynamic window string indexing problem by representing S as a concatenation of two shorter strings $S = S_1 \cdot S_2$. We then use online suffix tree data structures on S_1 and S_2. The one on S_1 supports only the updates AddLeft and RemoveLeft, and the one on S_2 supports only the updates AddRight and RemoveRight. To obtain the final result, we use a classic technique to implement a deque with two stacks (see e.g., Hoogerwood [15] combined with a deamortization scheme from Chuang and Goldberg [9].

1.4 Overview

In Section 2 we introduce some notation. In Section 3 we give our algorithm for finding occurrences of a streamed pattern P in strings of length $\mathcal{O}(|P|)$. In Section 4 we show how to solve the version where the string S only grows by appending characters to the right. Section 5 contains our new improved solution to the streaming sliding window string indexing (SSWSI) problem. Finally, in Section 6 we give our solution to the streaming dynamic window string indexing (SDWSI) problem.

2 Preliminaries

Given a string S of length n over an alphabet Σ, the ith character is denoted $S[i]$, and the substring starting at $S[i]$ and ending at $S[j]$ is denoted $S[i..j]$. The substrings of the form $S[i..n]$ are the *suffixes* of S. The reverse of a string S is the string $\mathsf{rev}(S) = S[n]S[n-1]\cdots S[1]$.

The *suffix tree* [23] T over a string $S[1..n]$ is the compact trie of all suffixes of $S\$$, where $\$ \notin \Sigma$ is lexicographically smaller than any letter in the alphabet. Each leaf corresponds to a suffix of S, and the leaves are ordered from left to right in lexicographically increasing order. The suffix tree uses $\mathcal{O}(n)$ space by implicitly representing the string associated with each edge using two indices into S. Farach-Colton, Ferragina, and Muthukrishnan [11] show that the optimal construction time for T is $\text{sort}(n, |\Sigma|)$, i.e., the time it takes to sort n elements from the universe Σ. The *suffix array* L of a string S is the array where $L[i]$ is the starting position of the ith lexicographically smallest suffix of S. Note that $L[i]$ corresponds to the ith leaf of T in left-to-right order. Furthermore, let v be an internal node in T and let s_v be the string spelled out by the root-to-v path. The descendant leaves of v exactly correspond to the suffixes of S that start with s_v, and these leaves correspond to a consecutive range $[\alpha, \beta]_v$ in L. The *locus* of a string P is the minimum depth node v such that P is a prefix of s_v.

▶ **Definition 7** (Periods). *We say that a positive integer p is a* period *of a string S if $S[i] = S[i+p]$ for all $i = 1, \ldots, |S| - p$. A string S is* periodic *if its smallest period is at most $|S|/2$.*

For a periodic pattern P with the smallest period p, we say $a_1, \ldots, a_k$ form a *chain* of occurrences of P in S if $P = S[a_i..a_i + |P| - 1]$ for $i = 1, \ldots, k$ and $a_i - a_{i-1} = p$ for $i = 2, \ldots, k$. The following (known) lemma is an easy consequence of the periodicity lemma [13].

▶ **Lemma 8.** *Let $a_1 < a_2 < \ldots < a_k$ be all occurrences of P in S, where $|S| \leq 2|P|$. If $k \geq 3$ then $a_1, a_2, \ldots, a_k$ form a chain.*

3 Matching Long Streaming Patterns

In this section we describe an algorithm that can find and compactly report all occurrences of a streamed pattern P in a string S of length $\mathcal{O}(|P|)$ in worst-case constant time per streamed character.

We are given an integer m and a string S of length $\mathcal{O}(m)$ supporting random access in constant time. We now receive a streamed pattern P of length between m and $3m$. We want to find all occurrences of P in S using constant time per streamed character. Since we do not know the precise length of P before we receive the last character our algorithm must be able to report all occurrences of the current P in constant time after receiving the first m

characters. Note that, since $|S| = \mathcal{O}(m)$ there is a constant number of occurrences of P in S unless P is periodic. If P is periodic then, by Lemma 8, the occurrences can be described by a constant number of chains.

Algorithm

The algorithm works in three phases. The ith phase starts after $i \cdot \lfloor m/4 \rfloor$ of characters of P have been streamed for $i \in \{1, 2, 3\}$. The third phase has two variants based on the periodicity of $P[0..\lfloor m/4 \rfloor - 1]$. Throughout the phases, we store the streamed characters of P.

Phase 1. Starts after the first $\lfloor m/4 \rfloor$ characters have arrived. We build a KMP automaton [16] of $P[0..\lfloor m/4 \rfloor - 1]$. When we have built the KMP automaton, we check if $P[0..\lfloor m/4 \rfloor - 1]$ is periodic and find its smallest period p.

Phase 2. Starts after $2\lfloor m/4 \rfloor$ characters have arrived. Find all occurrences of $P[0..\lfloor m/4 \rfloor - 1]$ in S using the KMP automaton. If $P[0..\lfloor m/4 \rfloor - 1]$ is not periodic then there is a constant number of occurrences and they are maintained explicitly. If $P[0..\lfloor m/4 \rfloor - 1]$ is periodic then, by Lemma 8, the occurrences can be described by a constant number of chains. In more detail, let $P = S[a_i..a_i + |P| - 1]$ be the previous occurrence and $P = S[a_{i+1}..a_{i+1} + |P| - 1]$ be the next occurrence. If $a_i + p = a_{i+1}$ then we extend the last chain by a_{i+1}, and otherwise we create a new chain initially consisting of only a_{i+1}.

Phase 3. Starts after $3\lfloor m/4 \rfloor$ characters have arrived. There are two cases depending on whether $P[0..\lfloor m/4 \rfloor - 1]$ is periodic or not.

Non-periodic. Extend the match of each occurrence from phase 2 simultaneously by explicitly matching each character of $P[\lfloor m/4 \rfloor..|P|]$. For each streamed character of P, match 4 characters until we have caught up to the stream. When the stream ends, report all occurrences.

Periodic. We match in each chain simultaneously as follows. Let $a_1, \ldots, a_k$ be a chain. We match against one occurrence in the chain, matching 4 characters at a time for each streamed character of P as in the non-periodic case. Let the current occurrence we are checking be occurrence j. Initially, $j = k$ and $i = \lfloor m/4 \rfloor$. As long as $i < |P|$ we do the following: Compare $P[i]$ and $S[a_j + i]$. If we match we set $i = i + 1$ and continue. Otherwise, there are two cases. If the mismatched character $P[i]$ is a continuation of the period and $j > 1$, we set $j = j - 1$, $i = i + 1$, and continue. Otherwise, we stop matching in this chain. While matching, we also check if P still has period p.
When the stream ends there are two cases. If P does not have period p (P is non-periodic or its smallest period is greater than p), then return the occurrence a_j. If P has period p we return the chain $a_1, \ldots, a_j$.

Analysis

Constructing a KMP automaton of $P[0..\lfloor m/4 \rfloor - 1]$ takes $\mathcal{O}(m)$ time. We can find the periodicity through the KMP in $\mathcal{O}(m)$ time. The number of characters streamed in phase 1 is $2\lfloor m/4 \rfloor - \lfloor m/4 \rfloor$, and thus we spent $\mathcal{O}(m)/\lfloor m/4 \rfloor = O(1)$ time per character in phase 1.

In phase 2, we match the KMP automaton of $P[0..\lfloor m/4 \rfloor - 1]$ against S. Since the length of S is $\mathcal{O}(m)$ and the number of characters streamed in phase 2 is $3\lfloor m/4 \rfloor - 2\lfloor m/4 \rfloor$, we spent $\mathcal{O}(m)/\lfloor m/4 \rfloor = \mathcal{O}(1)$ time per character in phase 2.

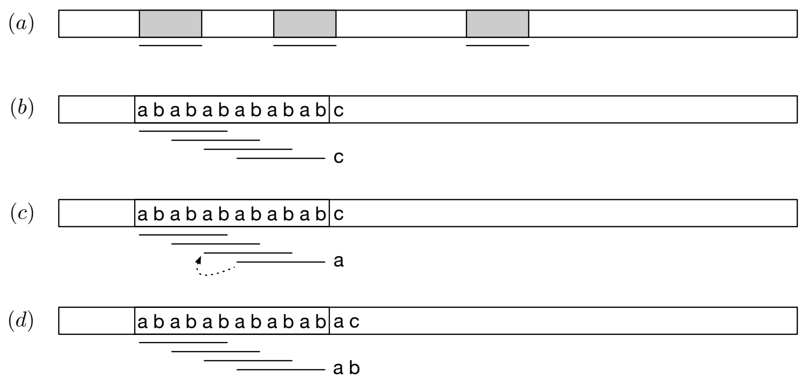
(a)
(b)
a b a b a b a b a b a b c
c
(c)
a b a b a b a b a b a b c
a
(d)
a b a b a b a b a b a b a c
a b

Figure 1 Phase 3. In (a) the pattern is non-periodic and we continue matching from each occurrence of $P[0..\lfloor m/4 \rfloor - 1]$ (marked with gray). (b)-(d) show different cases of the periodic case. Here $P[0..\lfloor m/4 \rfloor - 1] = ababab$. In (b) we match $P[i] = c$ and thus continue matching from this position. Since $P[i] = c$ is not a continuation of the period, we will never shift back to the previous occurrence. In (c) we mismatch and $P[i] = a$ is a continuation of the period, so we shift to the previous occurrence in the chain and keep matching. In (d) we mismatch and the and $P[i] = c$ is not a continuation of the period, so we stop.

In phase 3, we extend each match from phase 2. Since these are occurrences of a pattern of length at least $\lfloor m/4 \rfloor$ and S has length $\mathcal{O}(m)$, by Lemma 8 the number of occurrences or chains from phase 2 is constant. Since we match at most 3 characters at a time per occurrence or chain, we use $\mathcal{O}(1)$ time per streamed character in phase 3.

The space of the KMP automaton is linear, and in addition to that, we only need space for the strings S and P and a constant number of positions (set of possible occurrences) in S. Thus the space is $\mathcal{O}(m)$.

Correctness. For correctness, assume that while matching against chain $a_1, \ldots, a_k$ we have a mismatch $S[a_j + i] \neq P[i]$. If $P[0..i]$ has period p, then since $S[a_j..a_j + i - 1] = P[0..i-1]$ and $S[a_{j'}..a_{j'} + i - 1 + p] = P[0..p] \cdot S[a_{j'+1}..a_{j'+1} + i - 1]$ for $1 \leq j' < j$, then $a_{j'}$ is a starting position of $P[0..i]$. Otherwise, by the same argument, $P[0..i]$ has no starting position in the chain. Let $P[0..\ell]$ be the longest prefix of P such that $P[0..\ell]$ has period p and let $a_1, \ldots, a_j$ be all the starting positions of $P[0..\ell]$ in a chain. If $\ell \neq |P| - 1$, then the remaining part of P is non-periodic, and by the same argument as before, only a_j can be an occurrence of P, which we match explicitly against.

When we enter phase 3 we have matched against $P[0..\lfloor m/4 \rfloor - 1]$ and $3\lfloor m/4 \rfloor$ characters have been streamed. The earliest time the stream can terminate is after m characters. Since we match up to 4 characters at a time, by the time the stream can terminate, we could have matched $\lfloor m/4 \rfloor + 4(m - 3\lfloor m/4 \rfloor) > m$ characters, and thus we catch up to the stream before it can terminate.

In summary, we have shown the following.

▶ **Lemma 9.** *We are given an integer m and a string S of length $\mathcal{O}(m)$ supporting random access in constant time and a streamed pattern P of length between m and $3m$. For each arriving character of P we use constant time, and when the stream ends we output all the occurrences of P in S in constant time. If P is periodic, we output the occurrences as the $\mathcal{O}(1)$ chains describing all occurrences. The algorithm uses $\mathcal{O}(m)$ space.*

Note that we can replace KMP with any real-time pattern matching algorithm.

4 Matching Streaming Patterns with Append

In this section, we show how to maintain the string S under appending characters (adding characters at the right end using $\mathsf{AddRight}(a)$) while supporting $\mathsf{Report}(P)$.

Data structure

The data structure consists of a suffix tree over the reverse string of S, i.e., $\mathsf{rev}(S)$. We utilize the online suffix tree algorithm to maintain the suffix tree. To perform $\mathsf{AddRight}(a)$, we use the online suffix tree algorithm to insert the new suffix $a \cdot \mathsf{rev}(S)$ in the suffix tree by adding a new edge (v, w). Furthermore, if the edge (v, w) splits an edge in the suffix tree, i.e., the node v is a new node, then we store a pointer to the leaf w in node v.

Rollback. When we answer $\mathsf{Report}(P)$ queries, we will modify the data structure, but to quickly return the data structure to the state it had before the query, we do the following. We store three values for each memory cell c used by the data structure:

- The value v_c that cell c has when we are not processing a query.
- The value q_c that cell c has when we are processing a query.
- The timestamp t_c of the last query that modified the cell c. Initially, $t_c = -1$.

When we process the t'th query, if we access cell c, we first check if q_c is outdated by checking t_c. If $t_c < t$ then we access v_c. Otherwise, we access q_c. Whenever we modify a cell c during the query, we set $t_c = t$ and update q_c.

Query

To perform a query $\mathsf{Report}(P)$, we do the following. We prepend $\mathsf{rev}(S)$ with "\$" and then prepend each streamed character from P when we receive it. When we prepend the last character of P, we get the edge (v, w) that is added to insert the new suffix $\mathsf{rev}(P)\$\mathsf{rev}(S)$ in the suffix tree. If the first character of the string on edge (v, w) is "\$" then all other children of v are occurrences of P in S. Otherwise, there are no occurrences of P in S.

To report all occurrences in worst-case constant time per reported occurrence, we do the following. We do a depth first traversal of the subtree rooted at v, visiting four nodes at each time step. We get an occurrence for each node we visit, either by a leaf we visit, or the pointer to a leaf stored in an internal node visited. To avoid reporting the same occurrence twice, we keep an array of size w, storing which indices have been reported in the current $\mathsf{Report}(P)$ query. If we find multiple new occurrences in a single time step, we output one and store the remaining in a buffer. If we do not find any new occurrences in a time step we output an occurrence from the buffer.

Analysis

For each $\mathsf{AddRight}(a)$, we use the online suffix tree algorithm to find the new edge (v, w) in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time. We can identify if v is a new node in constant time by checking the number of children of v and update its stored pointer in constant time. Thus, each $\mathsf{AddRight}(a)$ takes $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time.

For each $\mathsf{Report}(P)$, we spend $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ per character in P that we prepend to the string. When we traverse the subtree rooted at v, we visit at most four nodes per reported occurrence. Thus, each $\mathsf{Report}(P)$ query uses $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ per character in P and an additional $\mathcal{O}(1)$ time per reported occurrence.

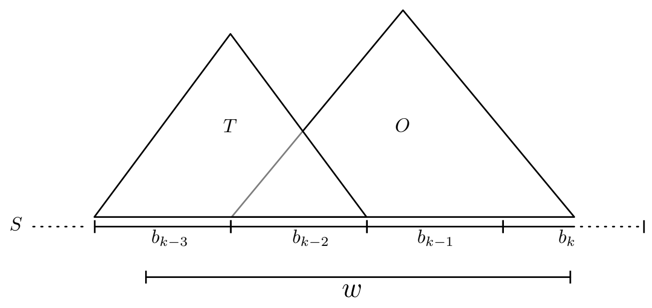

Figure 2 The data structures over the window.

The online suffix tree algorithm uses linear space and both the buffer and the array storing reported indices have size $\mathcal{O}(w)$. Thus the total space is $\mathcal{O}(w)$.

Correctness. Let (v, w) be the edge that is added to insert the new suffix $\mathsf{rev}(P)\$\mathsf{rev}(S)$ in the suffix tree. If the first character on the edge (v, w) is "\$", then the string on the path from the root of the suffix tree to v is $\mathsf{rev}(P)$. Since the suffix tree is built on the string $\mathsf{rev}(P)\$\mathsf{rev}(S)$, then all descendant leaves of v besides w, are occurrences of P in the string S.

Let t be the current time step since we started reporting occurrences. After the t^{th} time step, we have visited at least $2t$ nodes (or all the nodes in the subtree if the size of the subtree is less than $2t$). Since each leaf is stored in at most one internal node and the number of nodes in the subtree rooted at v is $2occ - 1$, then after the t^{th} time step, the number of unique occurrences found is at least t. Thus, at each time step, we either find an occurrence or an occurrence is stored in the buffer.

▶ **Lemma 10.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of streaming dynamic window string indexing operations on S: AddRight and Report in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time per character in $\mathcal{O}(w)$ space. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

5 Streaming Sliding Window String Indexing

In this section, we show Theorem 3. In the SSWSI problem, we can see it as we have a string S that is being streamed, and the string S' that we maintain corresponds to the window, i.e., after the i^{th} update $S' = S[i - w + 1..i]$. We partition the streamed string S into consecutive blocks $b_0, b_1, b_2, \ldots$ of length $B = \lceil w/3 \rceil$ (except possibly the last one), i.e., $b_j = S[j \cdot B..(j + 1) \cdot B - 1]$. Let b_k be the block containing the last streamed character $S[i]$, that is $k = \lfloor i/B \rfloor$. Thus the whole window is contained in $b_{k-3} \cdot b_{k-2} \cdot b_{k-1} \cdot b_k$.

Our data structure for the SSWSI problem consists of two structures. A static data structure T for $b_{k-3} \cdot b_{k-2}$ and an online data structure O for the string $b_{k-2} \cdot b_{k-1} \cdot b_k$. We utilize Lemma 10 for the online data structure O. See Figure 2. Furthermore, we store S' in a rotated array.

Static data structure

The static data structure consists of the following.

- A suffix tree.
- A suffix array A containing the leaves of the suffix tree in left to right order.
- A range maximum query data structure on the array A.
- Furthermore, each node in the suffix tree stores the range of its descendant leaves in the array A.

This is the same as the structure used in Bille et al. [6]. For the suffix tree, we use the same online suffix tree algorithm as used for the online data structure to build it. For the range maximum query, we use a data structure using linear space and preprocessing time and constant query time [5]. To perform a query P in the static data structure, we do the following. We search for P in the suffix tree, reading one character at a time. Let $[\ell, r]$ be the range of leaves stored in the locus of P. We perform a range maximum query in $A[\ell..r]$ to find the rightmost occurrence x. If x is not in the window, then there are no occurrences starting in $b_{k-3}b_{k-2}$ that are in the window. Otherwise, we report x and recurse on $A[\ell..x-1]$ and $A[x+1..r]$. To use worst-case constant time per reported occurrence we do three range maximum queries in one time step and keep a buffer of found but unreported occurrences as in Section 4. It then follows from a similar argument that we can report each occurrence in worst-case constant time.

The combined query

To find all occurrences of a streamed pattern P in S' we do the following. Assume that P has length $m \leq B$. We later show how to handle the case where $m > B$. We query T and O in parallel with the streamed characters of P. When P has arrived, we first report occurrences from the online data structure O. While we report occurrences from the online data structure, we prepare the static data structure to report occurrences that begin in b_{k-3} and are in the window, since occurrences that begin in b_{k-2} are also reported by the online structure. The static data structure does not report the occurrences from the right (even though the first reported occurrence is the rightmost). However, we can modify its reporting procedure so that all the occurrences in b_{k-2} will be reported before b_{k-3} as follows. If the currently considered occurrence falls within b_{k-1} we report it and recurse. Otherwise, we pause the recursion and add the current occurrence to a list. Next, we iterate over the list of paused recursive calls and resume each of them one-by-one. For each reported occurrence in the online data structure, we also process an occurrence in the static data structure until we get to the occurrences in b_{k-3}. When we have finished reporting occurrences in the online data structure O, we resume reporting the occurrences in b_{k-3}.

If $m > B$, we use Lemma 9 to find occurrences of P in S'. We note that, because of how these occurrences are reported (as either a constant number of explicitly given positions or a constant number of arithmetical progressions), it is trivial to filter out the occurrences that are anyway reported by the static or the online data structure. In summary, we have shown that we can report all occurrences of P in S'.

Rebuilding

We rebuild the structures in the background to keep the static and online data structure partially and completely inside the window, respectively. When block b_k begins, we start building the static data structure T' for $b_{k-2}b_{k-1}$ and the online data structure O' for $b_{k-1}b_k$. Since block b_{k-2} and b_{k-1} have been completed, we can use the online suffix tree algorithm to construct a suffix tree for $b_{k-2}b_{k-1}$. We construct T' and O' at a pace such that when the b_{k+1} begins, we have completed the structures and can swap T with T' and O with O'.

Analysis

We update the online data structure for each Update operation in $\mathcal{O}(t_{\text{suff}}(w, \sigma))$ time. We rebuild the static and online suffix trees once per block. It takes $\mathcal{O}(t_{\text{suff}}(w, \sigma) \cdot w)$ time to rebuild the static and online suffix trees. To build the range maximum query data structure and add ranges to the suffix tree, we spent an additional $\mathcal{O}(w)$ time. We augment the suffix tree with a dictionary data structure in each node over the first character of the labels of the outgoing edges. This takes $\mathcal{O}(t_{\text{dict}}(w,\sigma) \cdot w)$ time. Thus we use $\mathcal{O}((t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma)) \cdot w)/B = \mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ time on rebuilding for each Update.

For a query P, we spend $\mathcal{O}(t_{\text{suff}}(w,\sigma))$ time in the online data structure for each character in P by Lemma 10. In the static data structure, we traverse the suffix tree, using $\mathcal{O}(t_{\text{dict}}(w,\sigma))$ time for each character in P. By Lemma 9, we spend $\mathcal{O}(1)$ time per streamed character of P to report long patterns. Thus, in total, we spend $\mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ time per character plus additional constant time per reported occurrence.

Both the online and the static data structures use linear space. We have a constant number of such data structures at a time each over a substring of $b_{k-3}b_{k-2}b_{k-1}b_k$. Since $|b_{k-3}b_{k-2}b_{k-1}b_k| = \mathcal{O}(w)$, this use $\mathcal{O}(w)$ space in total. All other components, i.e. the buffer and the algorithm for the long patterns, use $\mathcal{O}(w)$ space. Thus the total space is $\mathcal{O}(w)$.

Correctness. The static and online data structures cover the entirety of S' and overlap by B characters, and thus they report all occurrences of pattern P in S' if the length of P is no more than B. When we report, we need time to discard the occurrences in b_{k-2} reported by the static data structure. Since the number of occurrences in b_{k-2} is no more than the occurrences in $b_{k-2}b_{k-1}b_k$, we have time to discard the occurrences in the static data structure if we report the occurrences in the online data structure first. If the pattern is longer than B, then the algorithm of Lemma 9 is ready to report occurrences since the sliding window has size $O(w)$ and P has a length between $B = \lceil w/3 \rceil$ and w.

6 Streaming Dynamic Window String Indexing

In this section, we consider the more general case, where we want to maintain the text S under adding and removing characters at either end, while still supporting Report(P) queries. We first show how to maintain S under only prepending/appending characters, and then extend this to the general case.

Directly from our definition of an online suffix tree construction algorithm we have:

▶ **Lemma 11.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of streaming dynamic window string indexing operations on S: AddLeft and Report in $\mathcal{O}(t_{\text{suff}}(w,\sigma) + t_{\text{dict}}(w,\sigma))$ time per character in $\mathcal{O}(w)$ space. Furthermore, Report uses additional worst-case constant time per reported occurrence.*

To get reporting in worst-case constant time per reported occurrence we do as in Section 4.

6.1 Prepend and Append

The current text S is represented as a concatenation $S = S_1S_2$. We store the characters of S_1 and S_2 on a doubly-linked list L_1 and L_2, respectively, and maintain the structure from Lemma 11 for S_1, and the structure from Lemma 10 for S_2. Initially, S_1 and S_2 are empty. The operation S.AddLeft(a) prepends a to L_1 and calls S_1.AddLeft(a), while S.AddRight(a) appends a to L_2 and calls S_2.AddRight(a). The operation S.Report(P) needs to consider occurrences of P in S_1, S_2, and straddling between S_1 and S_2. The first and the second case is implemented by running S_1.Report(P) and S_2.Report(P) in parallel. The third case is implemented by proceeding in phases $0, 1, 2, \ldots$. Phase k corresponds to $m \in (3^{k-1}, 3^k]$. In each phase, we maintain two instances of the procedure from Lemma 9. We maintain an invariant that the text S available to the first instance is $S_1[(|S_1| - 3^k + 1)..|S_1|]S_2[1..3^k]$ (length-3^k suffix of S_1 concatenated with length-3^k prefix of S_2), and after having read $P[i]$, where $i \in (3^{k-1}, 3^k]$, the pattern fed to the first instance is simply the whole $P[1..i]$. Thus, for any $m \in (3^{k-1}, 3^k]$ the first instance allows us to report all occurrences of P that straddle between S_1 and S_2. Meanwhile, we maintain the following invariant concerning the second instance. While reading $P[i]$, for $i \in (3^{k-1}, 2 \cdot 3^{k-1}]$, we create an array storing $S_1[(|S_1| - 3^{k+1} + 1)..|S_1|]S_2[1..3^{k+1}]$ (length-3^{k+1} suffix of S_1 concatenated with length-3^{k+1} prefix of S_2). This can be done by traversing L_1 from the last element and L_2 from the first element, spending constant time for every such i. Thus, after reaching $i = 2 \cdot 3^{k-1}$, the array stores the text that we will need in the next phase. Then, while reading $P[i]$, for $i \in (2 \cdot 3^{k-1}, 3^k]$, we send three characters of the pattern to the second instance for every new character of the pattern. More precisely, after receiving $P[i]$ we send $P[3(i-2 \cdot 3^{k-1})-2], P[3(i-2 \cdot 3^{k-1})-1]$, and $P[3(i-2 \cdot 3^{k-1})]$ to the second instance. Thus, after reaching $i = 3^k$, the second instance has received the whole current pattern $P[1..3^k]$, so we can swap the instances, reset the second instance, and proceed to the next phase.

6.2 General case

We first explain how to extend Lemma 10 to support both $\mathsf{AddLeft}(a)$ and $\mathsf{RemoveLeft}()$, and similarly how to extend Lemma 11 to support both $\mathsf{AddRight}(a)$ and $\mathsf{RemoveRight}()$. In both cases, we use the same simple idea.

▶ **Lemma 12.** *Let S be a dynamic string of length w over an alphabet σ. We can support the following subset of SDWSI operations on S:*

- *either AddLeft, RemoveLeft, and Report in $O(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ per character in $\mathcal{O}(w \cdot (t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma)))$space,*
- *or AddRight, RemoveRight, and Report in $O(t_{\text{suff}}(w, \sigma))$ per character, in $\mathcal{O}(w \cdot t_{\text{suff}}(w, \sigma))$ space.*

Furthermore, Report uses additional worst-case constant time per reported occurrence.

Proof. Supporting AddLeft and RemoveLeft or AddRight and RemoveRight can be seen as providing the possibility of undoing the most recent updates. We consider a structure that can be modified with an Update operation and denote the empty structure by $\perp$. Then, the current structure will be always $S = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_k$. We want to either modify it to obtain $S' = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_k.\mathsf{Update}_{k+1}$, or (if $k \geq 1$) undo the most recent update to obtain $S' = \perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots.\mathsf{Update}_{k-1}$. This can be implemented as follows. We maintain a stack consisting of k records. The i-th record stores (in e.g. a linked list) all modifications made when executing Update_i on $\perp.\mathsf{Update}_1.\mathsf{Update}_2.\ldots, \mathsf{Update}_{i-1}$. Each modification is described by specifying the address of a memory cell, and its value before the update. Let $u(w, \sigma)$ be the time for an update. Because any update modifies only $u(w, \sigma)$ memory cells, the space usage is $\mathcal{O}(w \cdot u(w, \sigma))$. Then, to undo the most recent update we retrieve the top record and revert all memory cells modified by the most recent update to their original values. This takes $\mathcal{O}(u(w, \sigma))$ time. During an update, we push a new record onto the stack and store all modified memory cells there. This also takes $\mathcal{O}(u(w, \sigma))$ time. The results now follow from plugging in the update times from Lemma 10 and Lemma 11. ◀

We are now ready to describe the general case. We use the well-known idea of implementing a deque with two stacks, see e.g. Hoogerwoord [15]. We briefly describe this idea. The current deque is represented as $S = \mathsf{rev}(S_1).S_2$, where S_1 and S_2 are stacks, rev denotes the reversal, and . the concatenation. Then, prepending an element is implemented by pushing it onto S_1 while appending an element is implemented by pushing it onto S_2. Removing the first element is implemented by popping it from S_1, while removing the last element is implemented by popping it from S_2. This works as long as both S_1 and S_2 are non-empty. As soon as one of them, say S_1, becomes empty, we rebuild the structure by distributing the elements stored on S_2 evenly between S_1 and S_2. It is easy to prove that the amortized cost of the rebuilding is constant, by defining the potential of the structure as $||S_1| - |S_2||$.

However, we need a worst-case efficient version. Chuang and Goldberg [9] provide a particularly clean description of how to modify the amortized version to obtain an implementation where every operation takes worst-case constant time. We refer the reader to their original description and only describe what is stored in their implementation. The current deque is represented as $S = \mathsf{rev}(S').\mathsf{rev}(S).B.B'$, where S', S, B and B' are stacks. Additionally, the structure maintains additional stacks $auxS, auxB$, and $extraS, newS, newB, extraB$. The rebuilding is done incrementally, and while this is being done every prepended element is pushed onto both S' and $extraS$, while every appended element is pushed onto both B' and $extraB$. Then, after the rebuilding has finished, the current deque is represented as $S = \mathsf{rev}(extraS).\mathsf{rev}(newS).newB.extraB$.

We built on the worst-case efficient implementation of Chuang and Goldberg [9] to prove Theorem 5. We maintain the current string S in a deque, and represent it as $S = \mathsf{rev}(S').\mathsf{rev}(S).B.B'$. Additionally, for each of the stacks S', S, B, B' and similarly $extraS, newS, newB, extraB$, we maintain a doubly-linked list storing its elements. Note that this would not be allowed in a purely functional implementation, which is the model assumed by Chuang and Goldberg [9], but we are not making any such assumption. Next, for each of the stacks $S', S, extraS, newS$ we maintain an instance of Lemma 12 with AddLeft and RemoveLeft while for each of the stacks $B, B', newB, extraB$ we maintain an instance of Lemma 12 with AddRight and RemoveRight. This makes the update time $\mathcal{O}(t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma))$ and space $\mathcal{O}(w \cdot (t_{\text{suff}}(w, \sigma) + t_{\text{dict}}(w, \sigma)))$. To implement $\mathsf{Report}(P)$, we separately consider occurrences of P inside $\mathsf{rev}(S'), \mathsf{rev}(S), B$ and B' by running $\mathsf{Report}(P)$ in parallel for each of the maintained instances. It remains to consider occurrences of P that straddle between $\mathsf{rev}(S')$ and $\mathsf{rev}(S).B.B'$, or between $\mathsf{rev}(S').\mathsf{rev}(S)$ and $B.B'$. or $\mathsf{rev}(S').\mathsf{rev}(S).B$ and B'. Each of these cases is solved as described in Section 6.1 by observing that we can provide access to the corresponding doubly-linked lists by (temporarily) concatenating some of the maintained doubly-linked lists. All three instances are run in parallel, so the overall additional time per character of P is constant.

References

1 Amihood Amir, Gianni Franceschini, Roberto Grossi, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Managing Unbounded-Length Keys in Comparison-Driven Data Structures with Applications to Online Indexing. *SIAM J. Comput.*, 43(4):1396–1416, 2014. doi:10.1137/110836377.

2 Amihood Amir, Tsvi Kopelowitz, Moshe Lewenstein, and Noa Lewenstein. Towards real-time suffix tree construction. In *Proc. 12th SPIRE*, volume 3772, pages 67–78. Springer, 2005. doi:10.1007/11575832_9.

3 Amihood Amir and Igor Nor. Real-time indexing over fixed finite alphabets. In *Proc. 19th SODA*, pages 1086–1095, 2008. URL: http://dl.acm.org/citation.cfm?id=1347082.1347201.

4 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3):13, 2007.

5 Michael A. Bender and Martín Farach-Colton. The lca problem revisited. In Gaston H. Gonnet and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics*, pages 88–94, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

6 Philip Bille, Johannes Fischer, Inge Li Gørtz, Max Rishøj Pedersen, and Tord Joakim Stordalen. Sliding window string indexing in streams. In *Proc. 34th CPM*, pages 4:1–4:18, 2023. doi:10.4230/LIPICS.CPM.2023.4.

7 Dany Breslauer and Giuseppe F. Italiano. Near real-time suffix tree construction via the fringe marked ancestor problem. *J. Discrete Algorithms*, 18:32–48, 2013. doi:10.1016/j.jda.2012.07.003.

8 Andrej Brodnik and Matevz Jekovec. Sliding suffix tree. *Algorithms*, 11(8):118, 2018. doi:10.3390/a11080118.

9 Tyng-Ruey Chuang and Benjamin Goldberg. Real-time deques, multihead thring machines, and purely functional programming. In *Proc. 6th FPCA*, pages 289–298, 1993.

10 Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th ICALP*, pages 6–19, 1990.

11 Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. ACM*, 47(6):987–1011, 2000.

12 Edward R. Fiala and Daniel H. Greene. Data compression with finite windows. *Commun. ACM*, 32(4):490–505, 1989. doi:10.1145/63334.63341.

13 N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. *Proc. Amer. Math. Soc.*, 16(1):109–114, 1965.

14 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. 26th CPM*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

15 Rob R. Hoogerwoord. A symmetric set of efficient list operations. *J. Funct. Program.*, 2(4):505–513, 1992.

16 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

17 Tsvi Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proc. 53rd FOCS*, pages 283–292, 2012. doi:10.1109/FOCS.2012.79.

18 S. Rao Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix trees (preliminary version). In *Proc. 26th STOC*, pages 310–316, 1994. doi:10.1145/195058.195170.

19 Gregory Kucherov and Yakov Nekrich. Full-Fledged Real-Time Indexing for Constant Size Alphabets. *Algorithmica*, 79(2):387–400, 2017. doi:10.1007/s00453-016-0199-7.

20 N. Jesper Larsson. *Structures of String Matching and Data Compression.* PhD thesis, Lund University, Sweden, 1999. URL: http://lup.lub.lu.se/record/19255.

21 Joong Chae Na, Alberto Apostolico, Costas S. Iliopoulos, and Kunsoo Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003. doi:10.1016/S0304-3975(03)00053-7.

22 Martin Senft and Tomás Dvorák. Sliding CDAWG perfection. In *Proc. 15th SPIRE*, pages 109–120, 2008. doi:10.1007/978-3-540-89097-3_12.

23 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th SWAT*, pages 1–11, 1973.