Near-Real-Time Solutions for Online String Problems

Dominik Köppl
University of Yamanashi, Kofu, Japan

Gregory Kucherov
LIGM, CNRS and Gustave Eiffel University, Marne-la-Vallée, France

Abstract

Based on the Breslauer–Italiano online suffix tree construction algorithm (2013) with double logarithmic worst-case guarantees on the update time per letter, we develop near-real-time algorithms for several classical problems on strings, including the computation of the longest repeating suffix array, the (reversed) Lempel–Ziv 77 factorization, and the maintenance of minimal unique substrings, all in an online manner. Our solutions improve over the best known running times for these problems in terms of the worst-case time per letter, for which we achieve a poly-*log-logarithmic* time complexity, within a linear space. Best known results for these problems require a poly-*logarithmic* time complexity per letter or only provide amortized complexity bounds. As a result of independent interest, we give conversions between the longest previous factor array and the longest repeating suffix array in space and time bounds based on their irreducible representations, which can have sizes sublinear in the length of the input string.

2012 ACM Subject Classification Information systems → Information retrieval

Keywords and phrases online algorithms, string algorithms, suffix tree, real-time computation, Lempel–Ziv factorization, minimal unique substrings

Digital Object Identifier 10.4230/LIPIcs.CPM.2026.2

Related Version *Full Version*: `https://arxiv.org/abs/2602.15311`

Funding *Dominik Köppl*: This work was supported in part by JSPS KAKENHI Grant Number 25K21150. We thank Fédération de Recherche Bézout for supporting the research visit of the first author to LIGM, where this work was initiated.

1 Introduction

Strings are a fundamental data type in computer science, and many classical problems on strings have been studied extensively in the literature. An important class of those problems concerns *online* processing of large strings when the string is read letter by letter from left to right, while maintaining the desired information about the string read so far. This type of solution is often required in practice. Unfortunately, most online algorithms for strings provide only *amortized* time guarantees per letter, i.e., the average time spent to process each letter is bounded by a function of the input size, but some read letters may take much longer to process. This can be problematic in real-time applications, where it is important to have predictable processing times for each letter. In this paper, we focus on online algorithms with *worst-case* time guarantees per letter. While literature gives real-time algorithms for LZ78-like factorizations [43] with a static dictionary [18] for constant alphabets, palindrome recognition and pattern matching [15], for many classical problems on strings, online algorithms with efficient worst-case time guarantees per letter are not known.

In fact, a majority of string problems can be only solved efficiently if we maintain a full-text indexing data structure of the string read so far. One of the most fundamental such data structures is the suffix tree [39]. The first online construction of the suffix tree

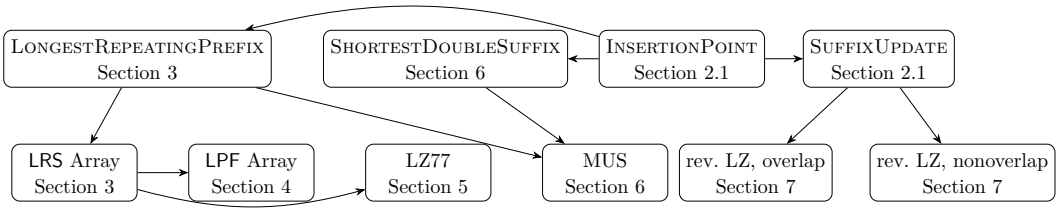

Figure 1 Problems (above) and applications (below) studied in this paper, with their dependencies visualized by arrows. SuffixUpdate is the fundamental problem on which all other problems and applications rely, for which arrows are omitted.

is due to Ukkonen [38], which builds a suffix tree online in time $O(n)$ for a constant-sized alphabet. However Ukkonen's algorithm does not provide any worst-case time guarantee on processing an individual letter, other than a trivial $O(n)$ bound. For certain inputs, this bound is actually tight: Given a string SSc of length n, Ukkonen's algorithm maintains an active node of string depth at least $|S|$ after having parsed the prefix SS, but then needs to add branches for all suffixes of S in case c is a new letter, and thus creates $\Theta(n)$ nodes just for updating the suffix tree for the last letter. (Spending $\Theta(n)$ for the last letter may be a common phenomenon in practice since we usually append a unique delimiter letter \$ at the end of the input string.) To achieve worst-case time guarantees, a line of research is based on the idea to maintain a suffix tree (or similar data structure) for the *inverted* input string. The rationale for this is that appending a new letter to the input string amounts to introducing only one new suffix in the inverted string, which causes less updates to the data structure compared to Ukkonen's algorithm. The starting point of this line of research is Weiner's algorithm [39], historically the first linear-time algorithm for suffix tree construction for constant-sized alphabets. Weiner's algorithm processes the string right-to-left by introducing a new suffix at each step, and is therefore suitable for this task. In its original version, Weiner's algorithm does not provide a worst-case time guarantee on processing a single letter. However, in the following Section 2.1, we summarize modifications that accomplish this. These modifications result in double logarithmic worst-case times per letter, a time complexity referred to as *near-real-time* [6].

Our contributions. Having a near-real-time suffix tree construction algorithm as a subroutine, we can solve various classical problems on strings in near-real-time online manner by relying on the maintained suffix tree. Figure 1 gives an overview of the problems and applications studied in this paper, along with their dependencies. We start with the computation of the *longest repeating suffix array* (LRS) in Section 3 and draw a connection to the *longest previous factor array* (LPF) in Section 4, which serves as the base for computing the Lempel–Ziv 77 (LZ77) [42] factorization in Section 5. In Section 6, we maintain the set of *minimal unique substrings* (MUS) of the string read so far. Finally, in Section 7, we maintain two variants of the LZ77 factorization on the reversed string. Table 1 summarizes our results along with the best known prior results. While some of these obtain $O(\text{polylog}(n))$ time per letter (although with additional constraints on the space), we obtain the first online algorithms with worst-case $O(\text{polyloglog}(n))$ time per letter for all the studied problems.

Table 1 Worst-case time complexities per letter for online algorithms computing various string problems on a string of length n. The shown results are the best known prior to this work, where we need $O(t_{\mathrm{SU}})$ time for each problem. The complexity t_{SU} is the time for solving SuffixUpdate, see Table 2 for possible bounds, which can be in $O(\mathrm{polyloglog}\, n)$.

problem	known	$O(t_{\mathrm{SU}})$-time solution
LRS array	$O(\lg^3 n)$ [28]	Section 3
LZ77	$O(\lg^3 n)$ [28]	Section 5
MUS	$O(\lg \sigma)$ amortized [24]	Section 6
reverse LZ	$O(\lg^2 \sigma)$ amortized [36]	Section 7
overlapping reverse LZ	$O(\lg^2 \sigma)$ amortized [36]	Section 7

2 Preliminaries

We assume the *word RAM* with word size $w \geq \log n$ bits, where n is the length of the input string. Let Σ denote an integer alphabet of size $\sigma = |\Sigma| = n^{O(1)}$. An element of Σ^* is called a *string*. Given a string $S \in \Sigma^*$, we denote its length with $|S|$, its i-th letter with $S[i]$ for $i \in [1..|S|]$. Further, we write $S[i..j] = S[i] \cdots S[j]$. We write $\overleftarrow{S}$ to denote the *inverted* string of S, i.e., $\overleftarrow{S} = S[|S|]S[|S|-1] \cdots S[1]$. The *suffix tree* $\mathsf{ST}(S)$ of a string S is a compacted trie representing all suffixes of S.

In what follows, we fix a string $T[1..n]$ over Σ as the input string to be processed online. Here, online means that the algorithm reads T letter by letter and maintains the result for the processed portion of the string. All presented algorithms in this paper use $O(n)$ words of space, unless otherwise stated. Note that this means that the space is proportional to the size of the currently processed string (as opposed to the size of the entire string).

2.1 Breslauer–Italiano modification of Weiner's algorithm

Weiner's algorithm computes the suffix tree $\mathsf{ST}(T)$ of a string $T[1..n]$ by processing it in backward direction and inserting suffixes from the shortest $T[n..]$ to the longest $T[1..]$. The algorithm augments suffix tree nodes with additional pointers called *Weiner links* (W-links for short): a *W-link* labeled with letter c from a node α (denoted by $\mathsf{W}_c(\alpha)$) points to the locus of string $c \cdot \mathsf{label}(\alpha)$, where $\mathsf{label}(\alpha)$ is the string spelled out by the path from the root to α. $\mathsf{W}_c(\alpha)$ is defined only if $c \cdot \mathsf{label}(\alpha)$ is a substring of the current string. If the locus of $c \cdot \mathsf{label}(\alpha)$ is a node, the W-link is called *hard*, otherwise it is called *soft*. In the Breslauer–Italiano modification of Weiner's algorithm, a soft W-link is represented by a pointer to the closest descendant node of the locus of $c \cdot \mathsf{label}(\alpha)$.

A round of Weiner's algorithm corresponds to the processing of a newly read letter. At round i, the algorithm inserts the new suffix to the suffix tree, solving the following problem.

SuffixUpdate
Input: suffix tree $\mathsf{ST}(T[i+1..])$ and letter $T[i]$.
Output: $\mathsf{ST}(T[i..])$.

To this end, it locates the *insertion point* where a new leaf should be attached. The insertion point is the locus of the longest prefix of $T[i..]$ already present as substring in $T[i+1..]$, it can be either an existing node or a locus on an existing edge. Computing the insertion point is the key operation of Weiner's algorithm, which we formulate as follows.

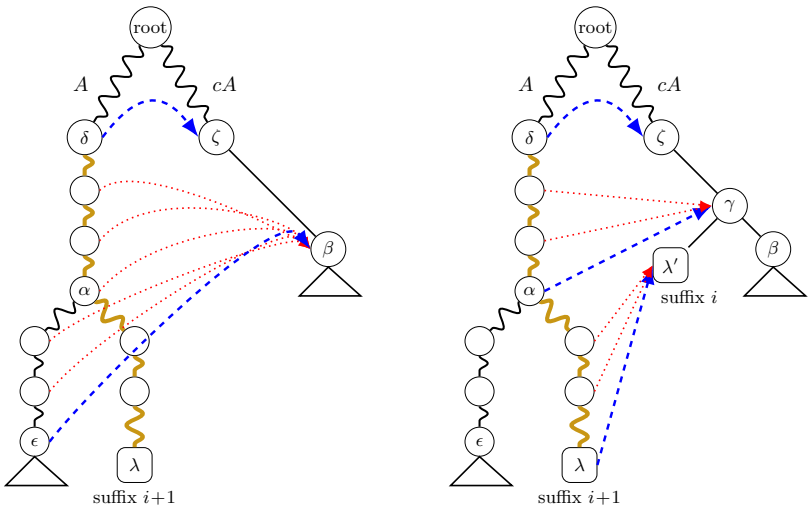

Figure 2 One round of Weiner's suffix tree construction algorithm: updating $\mathsf{ST}(T[i+1..])$ (left) to $\mathsf{ST}(T[i..])$ (right) by inserting the new suffix $T[i..]$. Dashed blue arrows represent hard W-links, dotted red arrows represent soft W-links, both for the letter $c = T[i]$. The W-links on the path from α to ϵ in the right tree are not shown for the sake of clarity. Curly edges represent paths that may contain multiple nodes. Node α is the closest ancestors of λ having a W-link by c. This link can be soft (as in the figure) or hard (in case $\alpha = \delta$). If this link is soft, the algorithm creates a new node γ which is an insertion point. If this link is hard, the insertion point is $W_c(\alpha)$. After creating γ, the W-links on the golden thick curly path from λ to δ need to be updated (Steps 4–5).

InsertionPoint
Input: suffix tree $\mathsf{ST}(T[i+1..])$ and letter $T[i]$.
Output: the locus of the longest prefix of $T[i..]$ appearing in $T[i+1..]$.

Solving InsertionPoint or the whole update process is sometimes called the *suffix tree oracle* [13, 37, 10].

Algorithmic steps of one round. We break down one round of Weiner's solving SuffixUpdate into steps, and explain each step in the context of Breslauer and Italiano's algorithm [6]. Figure 2 shows the suffix tree before and after the update, where the steps are illustrated by the changes from the left tree to the right tree. Let $c = T[i]$.

1. Starting from the leaf λ corresponding to suffix $T[i+1..]$, find its lowest ancestor α such that $\mathsf{W}_c(\alpha)$ is defined.
2. Let $\beta = \mathsf{W}_c(\alpha)$. If $\mathsf{W}_c(\alpha)$ is hard, then β is the insertion point. Otherwise, split the parent edge of β and create a new node γ. Copy all W-links from node β to γ which all become soft.
3. Create a new leaf λ' for suffix $T[i..]$ as a child of the insertion point.
4. Create a hard W-link $\mathsf{W}_c(\lambda) = \lambda'$ and a soft W-link $\mathsf{W}_c(q) = \lambda'$ for each node q on the path between λ (excluded) up to α (excluded).
5. Create a hard W-link $\mathsf{W}_c(\alpha) = \gamma$ and a soft W-link $\mathsf{W}_c(q) = \gamma$ for each node q on the path between α (excluded) up to the first node δ for which $\mathsf{W}_c(\delta) \neq \beta$ (excluded).

To analyze the time complexity of the above steps, we need the following two properties of Weiner's algorithm, which are known results but restated for the sake of completeness.

▶ **Lemma 1.** *The height of* $\mathsf{ST}(T[i..n])$ *is the height of* $\mathsf{ST}(T[i+1..n])$ *increased by at most one.*

Proof. At each round, Weiner's algorithm inserts a new suffix to the suffix tree. This insertion adds one leaf and at most one internal node γ. Given we created this node γ, all nodes in the subtree of γ have their depth increased by one, and the height of the suffix tree increases by at most one. ◀

▶ **Lemma 2.** *Assume an internal node* ζ *in* $\mathsf{ST}(T[i..])$ *is the locus of substring* cX*, i.e.,* $\mathsf{label}(\zeta) = cX$*. Then* X *is the locus of a node* δ *in* $\mathsf{ST}(T[i+1..])$*, in particular,* $\mathsf{W}_c(\delta) = \zeta$ *in* $\mathsf{ST}(T[i..])$ *and the depth of* ζ *is at most the depth of* δ *plus one.*

Proof. If ζ is already an internal node in $\mathsf{ST}(T[i..])$, then the suffix link of ζ points to a node δ with string label X in $\mathsf{ST}(T[i+1..])$.

Now assume that ζ is not an internal node in $\mathsf{ST}(T[i+1..])$ which can only happen when $c = T[i]$. Since ζ is an internal node in $\mathsf{ST}(T[i..])$, there are distinct letters d_1 and d_2 such that Xd_1 and Xd_2 are substrings of $T[i+1..]$. Hence, δ must be an internal node in $\mathsf{ST}(T[i..])$.

To show that the depth of ζ cannot be more than the depth of δ plus one, observe that if λ is an ancestor of ζ with string label cY, then there exists a node in $\mathsf{ST}(T[i+1..])$ with string label Y that is an ancestor of δ. Therefore, every node along the path to ζ maps by the suffix link to a distinct node on the path to δ, except the possible node labeled c which maps to the root. Consequently, the number of ancestors of ζ is bounded by the number of ancestors of δ plus one. ◀

Time complexity analysis. Implemented naively, Step 1 can visit up to $O(n)$ nodes, Step 2 may have to copy $O(\sigma)$ W-links, and Steps 4–5 can update the W-links of $O(n)$ nodes in the worst case. However, assuming a constant-size alphabet, the total time complexity over all n rounds can be shown to be $O(n)$ as follows. Step 1 traverses $(\mathsf{depth}(\lambda_{i+1}) - \mathsf{depth}(\delta))$ nodes, where λ_i stands for the leaf labeling the entire string $T[i..]$. By Lemma 2, we have $\mathsf{depth}(\zeta) \leq \mathsf{depth}(\delta) + 1$, and then $\mathsf{depth}(\lambda_{i+1}) - \mathsf{depth}(\delta) \leq \mathsf{depth}(\lambda_{i+1}) - \mathsf{depth}(\zeta) + 1 = \mathsf{depth}(\lambda_{i+1}) - \mathsf{depth}(\lambda_i) + 3$. Summing up over all rounds, Step 1 visits $O(n)$ nodes altogether. By a similar argument, Steps 4–5 update up to $O(n)$ W-links as well. Finally, Step 2 needs $O(n)$ time in total since there are at most n hard W-links in the entire suffix tree and there are $O(1)$ W-links to copy at each round.

To achieve a nontrivial worst-case time bound per letter, we need to efficiently implement Steps 1 and Steps 4–5. Breslauer and Italiano [6] proposed to implement Step 1 by maintaining, for each alphabet letter a, the (dynamic) Euler tour list of the current suffix tree where nodes are *marked* in color a according to W_a-links defined in them. Then, the ancestor node α is retrieved using two data structures: one supporting queries for the *previous marked element* in a dynamic list, applied to the Euler tour, and the other supporting *dynamic lowest common ancestor* applied to the suffix tree itself. The first is solved using the data structure of Dietz and Raman [12] and the second using the data structure of Cole and Hariharan [9]. Altogether, this results in $O(\log\log n)$ time. We note in passing that for a constant-size alphabet, dynamic lowest common ancestor queries can be easily avoided, therefore only colored predecessor queries are critical.

To efficiently implement updates of Steps 4–5, the authors propose to distribute this work over multiple rounds of the algorithm (i.e. multiple letters of the input string) by updating only a constant number at each round. (For this reason, the algorithm is called *quasi-real time*.) However, the nodes are updated in the order of increasing depth which guarantees no interference with the ongoing process of tree updating. As a result, Breslauer and Italiano modification spends $O(\log\log n)$ worst-case time on each letter.

Table 2 Time complexities t_{SU} for SuffixUpdate.

Time complexity t_{SU}	Reference and comments
$O(\lg n)$	[1]
$O(\sigma \log\log n)$	[6]
$O(\log\log n + \log\log \sigma)$	[20], expected time
$O(\log\log n + \frac{\log^2 \log \sigma}{\log\log\log \sigma})$	[13]
$O(\log\log n)$	[22], for $\sigma = O(\log^{1/4} n)$

The central observation for this paper is that this modification, as well as the improvements in the following subsection, maintain the suffix tree within this worst case-time per letter while keeping nearly complete functionality.

▶ **Observation 3.** *The suffix tree updated by the Breslauer–Italiano algorithm is fully functional except that some W-links may be missing temporarily (due to the deferred updates in Steps 4–5).*

2.2 Improvements of Breslauer–Italiano algorithm

Altogether, the Breslauer–Italiano algorithm achieves worst-case $O(\sigma \log\log n)$ time per read letter, where σ is the alphabet size. For $\sigma = O(1)$ this results in $O(\log\log n)$ time per read letter. This technique was the first to achieve a double logarithmic worst-case bound, as only an $O(\log n)$ bound had been known previously [1].

An obvious weakness of the Breslauer–Italiano approach is the linear dependency on the alphabet size. Subsequent works proposed improvements on this point. Kopelowitz [20] proposed a randomized algorithm for maintaining the suffix tree online in $O(\log\log n + \log\log \sigma)$ worst-case *expected* time per letter. Kucherov and Nekrich [22] generalize the $O(\log\log n)$ bound of [6] to log-sized alphabets, more precisely to $\sigma = O(\log^{1/4} n)$, using the *generalized van Emde Boas data structure* of Giora and Kaplan [16]. An interesting improvement proposed in [22] is that weak W-links are not maintained at all but are computed on the fly in the lazy fashion. This greatly simplifies the steps of the Breslauer–Italiano algorithm: Steps 2, 4 and 5 become constant time and only Step 1 remains to be implemented. The latter is essentially reduced to answering the *colored predecessor queries* on a dynamic colored list, for which the authors borrow an $O(\log\log n)$ solution of [16] for up to $O(\log^{1/4} n)$ colors. Mortensen [25] proposes a $O(\log^2 \log n)$ solution for the dynamic colored predecessor problem for any number of colors, which can be plugged in as an alternative to compute InsertionPoint.

On the other hand, Fischer and Gawrychowski [13] obtain $O(\log\log n + \frac{\log^2 \log \sigma}{\log\log\log \sigma})$ time using a sophisticated technique of *Wexponential search trees*. They store only some of the weak W-links and show how to implement InsertionPoint within this time bound. Steps 4 and 5 can be deamortized as in the Breslauer–Italiano algorithm. A summary of the known time complexities t_{SU} for processing SuffixUpdate is given in Table 2.

Note finally that the idea of maintaining online an indexing data structure for the inverted string has been applied to indexes based on the Burrows–Wheeler transform (BWT) [8] as well. Similar to the suffix tree, it is more convenient to maintain a BWT-index for the inverted string, by adding only one suffix at each round [30, 27]. Updating the BWT-index boils down to maintaining a dynamic string that supports access, rank, and select queries, for which logarithmic solutions exist [26], which, however, are optimal [14]. Thus, there is no great hope for improving the time bounds for online BWT-index maintenance beyond

logarithmic time per letter. In what follows, we focus on a line of research that applies this online BWT-index construction for computing the LRS array, and show that we can obtain better time bounds by relying on the suffix tree construction instead.

3 Longest Repeating Suffix Array

From now on, we will be considering the online setting where the input string T is provided letter by letter from left to right, and we will be maintaining a suffix tree of the *inverted* string $\overleftarrow{T}$, using a variant of Weiner's algorithm. Thus, a *suffix* of a current string will become an *inverted prefix* of the actually indexed string, etc. In our presentation, we will consider both the string and its inverted image and alternate between the two depending on the context which should be clear to the reader.

The *longest repeating suffix array* (LRS) of a string $T[1..n]$ is an array $\mathsf{LRS}[1..n]$ such that for each position $i \in [1..n]$, $\mathsf{LRS}[i]$ is the length of the longest suffix of $T[1..i]$ that occurs at least twice in $T[1..i]$. In other words, $\mathsf{LRS}[i]$ is the length of the longest suffix of $T[1..i]$ occurring also in $T[1..i-1]$.

Okanohara and Sadakane [28] were the first to study computing this array online, for which they obtained $O(\log^3 n)$ worst-case time per letter. Prezza and Rosone [31] obtained $O(\log^2 n)$ *amortized* time, with $O(n \lg n)$ worst-case delay per letter. Importantly, both approaches [28] and [31] solve the problem under additional constraints on the space used by the algorithm: *compact space* for the former and zero-entropy *compressed space* for the latter.

We here show that, without the compact space constraint, we can efficiently compute LRS in near-real-time as a by-product of maintaining ST. Observe that under Weiner's approach described in Section 2.1, the problem becomes the following.

LongestRepeatingPrefix
Input: suffix tree $\mathsf{ST}(T[i+1..])$ and letter $T[i]$.
Output: the longest repeating prefix of $T[i..]$.

We solve LongestRepeatingPrefix by revisiting the methodology of Amir et al. [2]. They reduce the problem to InsertionPoint, since the string label of the insertion point is the longest prefix of $T[i..]$ that already appears in $T[i+1..]$. Thus, updating $\mathsf{ST}(\overleftarrow{T})$ using Weiner's algorithm allows us to compute LRS. When processing letter $T[i]$, we query LongestRepeatingPrefix on $\mathsf{ST}(\overleftarrow{T[1..i-1]})$ with letter $T[i]$. The answer is $\mathsf{LRS}[i]$.

▶ **Theorem 4.** *The longest repeating suffix array* LRS *of a string* $T[1..n]$ *can be computed online in* $O(t_{\mathrm{SU}})$ *worst-case time per letter.*

4 Longest Previous Factor Array

The *longest previous factor array* (LPF) of a string $T[1..n]$ is an array $\mathsf{LPF}[1..n]$ such that for each position $i \in [1..n]$, $\mathsf{LPF}[i]$ is the length of the longest prefix of $T[i..n]$ that has another occurrence starting at a position $j < i$.

We first describe how to convert LRS to LPF and vice versa in linear time *offline*. A similar conversion between border and prefix arrays has been studied in the literature [4]. For that, we observe that we can restate the definitions of LRS and LPF as follows: Let $X_i = \{j \le i \mid j + \mathsf{LPF}[j] - 1 \ge i\}$ be the set of positions j such that the longest previous factor starting at j covers position i. If $X_i \neq \emptyset$, $|X_i| = i - \min(X_i) + 1$. Consequently,

$$\mathsf{LRS}[i] = \begin{cases} |X_i| & \text{if } X_i \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

The problem of computing LRS from LPF is then reduced to finding the minimum in X_i for each i. For that, we focus on *irreducible* LPF values, i.e., positions i such that $\mathsf{LPF}[i] > \mathsf{LPF}[i-1]-1$. The important invariant is that irreducible LPF intervals of the form $[i..i+\mathsf{LPF}[i]-1]$ cannot nest but overlap or are disjoint. Thus, we can determine $\mathsf{LRS}[i]$ by maintaining the leftmost irreducible LPF interval covering position i. To this end, we scan the irreducible LPF values from left to right. More precisely, with a sweep-line $i \in [1..n]$ from left to right we maintain the leftmost irreducible LPF interval $[j..j+\mathsf{LPF}[j]-1]$ such that $i \in [j..j+\mathsf{LPF}[j]-1]$. Then j is the minimum j in X_i.

We can also compute LPF from LRS in linear time by scanning the LRS array left-to-right and computing LPF using the following rule. For each i such that $\mathsf{LRS}[i] \leq \mathsf{LRS}[i-1]$, we set $\mathsf{LPF}[j] = i-j$ for all $j \in [i-\mathsf{LRS}[i-1]..i-\mathsf{LRS}[i]]$. For $i=n$, we complete the computation by setting $\mathsf{LPF}[j] = n-j+1$ for all $j \in [n-\mathsf{LRS}[n]+1..n]$.

▶ **Theorem 5.** *Offline conversions between the longest repeating suffix array* $\mathsf{LRS}[1..n]$ *and the longest previous factor array* $\mathsf{LPF}[1..n]$ *of a string* $T[1..n]$ *can be done in* $O(n)$ *time.*

Like the LCP array [32], the LPF array can be recovered from the list of its r irreducible values in linear time [3], where it has been shown how to store the irreducible values in $2n+o(n)$ bits of space and still support constant-time access to either the i-th irreducible value or $\mathsf{LPF}[i]$, for any given i. By an analogous argument, we can store the irreducible values of LRS within the same space bound with the same query functionality [31]. It is therefore natural to ask whether the above conversions can be done on the irreducible values only. Actually, we can observe that the above algorithm for converting between LRS and LPF can be adapted to work on the irreducible values only, by skipping the assignments for reducible values.

▶ **Corollary 6.** *Offline conversions between the* $O(r)$ *irreducible LRS values and the* $O(r)$ *irreducible LPF values can be done in* $O(r)$ *time.*

The left-to-right conversion of LRS into LPF can also be used to compute the LPF values online by computing LRS values online as described in Section 3, with a delay. By delay we mean that given $T[1..i]$ is the currently processed text, then we have not yet computed the $\mathsf{LPF}[j]$ last values for the positions $j \in [i-\mathsf{LRS}[i]+1..i]$. To turn our offline left-to-right conversion into (near-)real-time, we need to deamortize the sequence of assignments triggered by $\mathsf{LRS}[i] < \mathsf{LRS}[i-1]+1$. This can be done by distributing these assignments over multiple rounds and setting a constant number of values at each round, similarly to Steps 4 and 5 of Breslauer–Italiano algorithm (Section 2.1). The delayed assignments can be stored in a FIFO data structure. By induction we observe that the maximal number of delayed assignments is bounded by $\max_i \mathsf{LRS}[i]$. We thus obtain the following result.

▶ **Theorem 7.** *The longest previous factor array* LPF *of a string* $T[1..n]$ *can be computed online in* $O(t_{\mathrm{SU}})$ *worst-case time per letter, with at most* $\max_i \mathsf{LRS}[i]$ *delayed ending values at each round.*

5 Lempel–Ziv factorization (LZ77)

The LZ77 factorization of a string T is defined as follows: a factorization $T = F_1 \cdots F_z$ it is the *LZ77 factorization* of T if each next factor F_x, for $x \in [1..z]$, is either the first occurrence of a letter or the longest prefix of $F_x \cdots F_z$ that occurs at least twice in $F_1 \cdots F_x$. The factorization can be written like a macro scheme [35], i.e., by a list storing either plain letters or pairs of referred positions and lengths, where a referred position is a previous text position from where the letters of the respective factor can be copied.

While practical compressors based the LZ77 [42] factorization such as gzip and 7zip work with a sliding window over the input text, the original LZ77 factorization considers the entire text as the search buffer. For the online setting, Gusfield's textbook [17, APL 16] presents an algorithm that runs in $O(\log \sigma)$ amortized time per letter. The main idea is to maintain the suffix tree of the text read so far using Ukkonen's algorithm [38], and simultaneously compute each next factor by a top-down traversal of the tree. Since we can easily extract the LZ77 factorization from the LRS or LPF arrays, algorithms of Sections 3,4 can be also used to compute it. Direct online algorithms for computing the LZ77 factorization have also been proposed, additionally focusing on space saving. In this line of research, Starikovskaya [34] proposed an algorithm in $O(\log^2 n)$ amortized time per letter by returning to Ukknonen's suffix tree construction algorithm. Subsequently, Yamamoto et al. [41] obtained $O(\log n)$ amortized time per letter by constructing the directed acyclic word graph [5], and finally Policriti and Prezza [29] achieved the same time bounds with a dynamic FM-index in zero-order entropy compressed space.

Here we diverge from this line of research by allowing $O(n)$ words of space but focusing on the worst-case time per letter. Using the near-real-time computation of LRS (Section 3), it is straightforward to compute the LZ77 factorization within the same time bounds.

Online computation of LZ77 factorization amounts to reporting if the current position i extends the current factor or starts a new one. In the latter case, the previous factor ends at position $i-1$ and i is the first position of a new factor. We can determine this by comparing $i - \mathsf{LRS}[i]$ with the beginning s_x of the current factor F_x. If $i - \mathsf{LRS}[i] + 1 \leq s_x$, then i extends F_x, otherwise F_x ends at $i-1$ and i starts a new factor F_{x+1}. We observe that $\mathsf{LRS}[i] = 0$ if and only if $T[i]$ is the leftmost occurrence of a letter.

▶ **Theorem 8.** *The LZ77 factorization of a string $T[1..n]$ can be computed online in $O(t_{\mathrm{SU}})$ worst-case time per letter.*

6 Minimal Unique Substrings

Given a substring S of a text $T[1..n]$, let $\#occ_T(S)$ denote the number of occurrences of S in T. A substring S of T is called *unique* in T if $\#occ_T(S) = 1$ and *repeating* in T if $\#occ_T(S) \geq 2$. A unique substring S of T is called *a minimal unique substring* of T if S is unique and any proper substring of S is repeating in T. Since a unique substring S of T has exactly one occurrence in T, it can be identified with a unique interval $[\ell..r] \subseteq [1..n]$ with $S = T[\ell..r]$. We denote the set of intervals corresponding to the MUSs of T by $\mathsf{MUS}(T) = \{[s..t] \mid T[s..t] \text{ is a MUS of } T\}$. By the definition of MUSs, $[s..t] \in \mathsf{MUS}(T)$ if and only if (a) $T[s..t]$ is unique in T, (b) $T[s+1..t]$ is repeating in T, and (c) $T[s..t-1]$ is repeating in T.

Mieno et al. [24] present an algorithm for computing MUSs in a sliding window and, in particular, study how $\mathsf{MUS}(T)$ can be updated when a new letter $T[j+1]$ is appended to $T[1..j]$. We summarize these updates here, slightly modifying the description of [24]. Denote $\mathsf{lrs}[j]$ the longest repeating suffix of $T[1..j]$, i.e. $\mathsf{LRS}[j] = |\mathsf{lrs}[j]|$. Denote $\mathsf{sds}[j]$ the shortest suffix of $T[1..j]$ with exactly two occurrences in $T[1..j]$ (that is, having exactly one previous copy). Since $\mathsf{lrs}[j]$ can admit the empty string, $\mathsf{lrs}[j]$ always exists. On the other hand, $\mathsf{sds}[j]$ may not exist when no suffix of $T[1..j]$ occurs exactly twice, i.e., $\#occ_{T[1..j]}(T[i..j]) \neq 2$ for all $i \in [1..j]$.

The size of $\mathsf{lrs}[j]$ and the ending position of the previous copy of $\mathsf{sds}[j]$ (if $\mathsf{sds}[j]$ exists) are the two parameters we need to know in order to specify all possible modifications to $\mathsf{MUS}(T[1..j])$ when $T[j+1]$ is appended to $T[1..j]$. These modifications can consist of one deletion of a MUS and up to three additions of new MUSs, specified below.

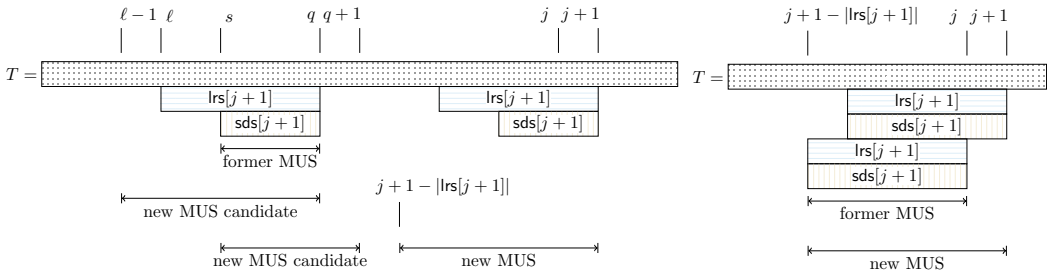

Figure 3 Illustration of Cases (1) – (4). Left: new MUS (Case (1)), former MUS (Case (2)) and two potential new MUSs (Cases (3) and (4)), Right: particular case of Case (2) when $q = j$.

(1): If $\mathsf{LRS}[j+1] \leq \mathsf{LRS}[j]$, then $[j+1-\mathsf{LRS}[j+1]..j+1]$ is a new MUS to be added to $\mathsf{MUS}(T[1..j+1])$.

Proof. By definition of $\mathsf{lrs}[j+1]$, on the one hand, $T[j+1-\mathsf{LRS}[j+1]..j+1]$ is unique. On the other hand, both $T[j+1-\mathsf{LRS}[j+1]..j]$ and $T[j+1-\mathsf{LRS}[j+1]+1..j+1]$ are repeating in $T[1..j+1]$. The former is repeating as it must be a suffix of $T[j-\mathsf{LRS}[j]+1..j]$. ◀

(2): If $\mathsf{sds}[j+1]$ exists, let $T[s..q]$ be its previous copy ($q < j+1$). Then $[s..q]$ is in $\mathsf{MUS}(T[1..j])$ but not in $\mathsf{MUS}(T[1..j+1])$ and therefore should be deleted.

Proof. $T[s..q]$ was unique in $T[1..j]$ as it occurs exactly twice in $T[1..j+1]$. $T[s+1..q]$ must be repeating in $T[1..j]$ as $T[s..q] = \mathsf{sds}[j+1]$ is the *shortest* suffix of $T[1..j+1]$ occurring twice, therefore $T[s+1..q]$ must occur at least three times in $T[1..j+1]$ and therefore at least twice in $T[1..j]$. On the other hand, $T[s..q-1]$ is a repeating suffix of $T[1..j]$. Therefore, $T[s..q]$ was a MUS in $T[1..j]$ that is no longer a MUS in $T[1..j+1]$. Note that $T[s..q]$ may overlap $T[j+1-|\mathsf{sds}[j+1]|+1..j+1]$, in particular, it may happen that $q = j$. In this case, $T[s..j] = \mathsf{sds}[j+1] = T[s+1..j+1]$ and therefore $T[s..j+1] = T[j+1]^k$, $k = |\mathsf{sds}[j+1]|+1$, is the single letter run which is unique in $T[1..j+1]$. Observe that in this case, we have $\mathsf{sds}[j+1] = \mathsf{lrs}[j+1]$. The situation is illustrated on the right of Figure 3. ◀

If $\mathsf{sds}[j+1]$ exists, one or two new MUSs can appear in this case, specified in the following and illustrated on the left in Figure 3.

(3): A new potential MUS to be added to $\mathsf{MUS}(T[1..j+1])$ becomes $[s..q+1]$ provided that no (shorter) MUS in $T[1..j]$ ends at $q+1$.

Proof. Since $T[s..q] = \mathsf{sds}[j+1]$ has a single non-suffix occurrence in $T[1..j+1]$, $T[s..q+1]$ is unique in $T[1..j+1]$. Since $T[s..q]$ is repeating in $T[1..j+1]$, the shortest unique suffix $T[\ell..q+1]$ of $T[s..q+1]$ is a MUS in $T[1..j+1]$. If $\ell = s$, this MUS is a new one, otherwise it already existed in $T[1..j]$. ◀

(4): A new potential MUS to be added to $\mathsf{MUS}(T[1..j+1])$ becomes $[\ell-1..q]$ where $T[\ell..q] = \mathsf{lrs}[j+1]$, provided that $\ell \geq 1$ and no (shorter) MUS in $T[1..j]$ starts at $\ell-1$.

Proof. By definition of $\mathsf{lrs}[j+1]$, $T[\ell-1..q]$ is unique in $T[1..j+1]$ and $T[\ell..q]$ is repeating. Therefore, the shortest unique prefix $T[\ell-1..p]$ of $T[\ell-1..q]$ is a MUS in $T[1..j+1]$. If $p = q$, this MUS is a new one, otherwise it already existed in $T[1..j]$. ◀

We now turn to the implementation of the above updates. First, we need to represent the current set of MUSs. Knowing that MUSs cannot nest, one can bijectively map the starting position of a MUS to its ending position, and vice versa. The authors of [24] used this insight to create two arrays to maintain MUSs via the bijection and its reverse. This allows them to retrieve a MUS by querying one of its end points, and adding/removing a MUS, all in constant time per operation.

To implement the updates online, the authors of [24] maintain $\mathsf{ST}(T[1..j])$ with Ukkonen's suffix tree construction algorithm [38]. While reading a new letter $T[j+1]$ from the text, they update $\mathsf{ST}(T[1..j])$ and compute the locus of $\mathsf{lrs}[j+1]$ and $\mathsf{sds}[j+1]$, which they call *active points*, and their string lengths $\mathsf{LRS}[j+1]$ and $|\mathsf{sds}[j+1]|$ respectively. Computing $\mathsf{lrs}[j+1]$ is a direct by-product of Ukkonen's algorithm, whereas computing $\mathsf{sds}[j+1]$ requires some additional work. Using this information, the set of MUSs is updated using the above description.

We now show how to efficiently implement the updates when we maintain the suffix tree online with Weiner's algorithm applied to the inverted text, as discussed in Section 2. Maintaining $\mathsf{lrs}[i]$ has been studied in Section 3. To maintain $\mathsf{sds}[i]$, we need to solve the following problem.

ShortestDoubleSuffix
Input: suffix tree $\mathsf{ST}(\overleftarrow{T[..j]})$ and letter $T[j+1]$.
Output: the shortest prefix of $\overleftarrow{T[..j+1]}$ that has exactly one occurrence in $\overleftarrow{T[1..j]}$, if such a prefix exists.

Assume a new letter $T[j+1]$ is appended to $T[1..j]$. As explained in Section 3, the locus of $\mathsf{lrs}[j+1]$ corresponds to the insertion point of Weiner's algorithm (see Section 2.1). If the insertion point is an already existing node in $\mathsf{ST}(\overleftarrow{T[1..j]})$, then $\mathsf{lrs}[j+1]$ occurs at least three times in $T[1..j+1]$ and $\mathsf{sds}[j+1]$ does not exist. Otherwise, the locus of $\mathsf{lrs}[j+1]$ becomes a node in $\mathsf{ST}(\overleftarrow{T[1..j+1]})$ with exactly two children (one of them is a leaf). Therefore, in this case, $\mathsf{sds}[j+1]$ exists and its locus is located on the parent edge to the locus of $\mathsf{lrs}[j+1]$ and thus is obtained "for free" in constant time, simplifying the algorithm of [24].

Once $\mathsf{lrs}[j+1]$ and $\mathsf{sds}[j+1]$ are found, we update the set of MUSs implementing the modifications described above:

1. Add a new MUS $[j+1-\mathsf{LRS}[j+1]..j+1]$.
2. If $\mathsf{sds}[j+1]$ exists, find its non-suffix occurrence $[s..q]$: s is the label of the leaf of the locus of $\mathsf{sds}[j+1]$ that already existed in $\mathsf{ST}(T[1..j])$, and $q = s + |\mathsf{sds}[j+1]| - 1$. Delete the MUS $[s..q]$.
3. If there is no MUS ending at position $q+1$, add the MUS $[s..q+1]$.
4. If $q - \mathsf{LRS}[j+1] \geq 1$ and there is no MUS starting at position $q - \mathsf{LRS}[j+1]$, add the MUS $[q-\mathsf{LRS}[j+1]..q]$.

Following [24], we maintain two integer arrays X_1 and X_2 of size n mapping starting positions of the MUSs to their ending positions, and vice versa. In our online setting, arrays X_1 and X_2 are dynamic and grow by one entry to the right at each newly read letter. We implement X_1 and X_2 using the standard doubling technique (e.g. [33, Section 3.2]) which allocates a new array fragment of size n when the size of the current string reaches n, thus doubling the occupied memory. While this implementation maintains the $O(n)$ memory, the waisted memory is also up to $O(n)$. To reduce the waisted memory, one can use the *resizeable array* of Brodnik [7] which supports the growable array operations in $O(1)$ worst-case time while having only up to $O(\sqrt{n})$ of waisted memory. We summarize this section with the following final result.

▶ **Theorem 9.** *We can maintain the set of minimal unique substrings* $\mathsf{MUS}(T[1..n])$ *of a string* $T[1..n]$ *online in* $O(n)$ *space and* $O(t_{\mathrm{SU}})$ *worst-case time per letter.*

7 Reversed Lempel–Ziv

The reversed LZ factorization was introduced by Kolpakov and Kucherov [19] as a helpful tool for detecting *gapped palindromes*, i.e., substrings of a given text T of the form $\overleftarrow{S}GS$ for two strings S and G. The reversed LZ factorization of T is defined recursively as follows: a factorization $T = F_1 \cdots F_z$ is the *reversed LZ factorization* of T if each factor F_x, for $x \in [1..z]$, is either the first occurrence of a letter or the longest prefix of $F_x \cdots F_z$ that has an inverted copy $\overleftarrow{F_x}$ as a substring of $F_1 \cdots F_{x-1}$. Like LZ77, the factorization is a macro scheme [35]. Among all variants of such a left-to-right parsing using the reversed substring as a reference to the formerly parsed part of the text, the greedy parsing achieves optimality with respect to the number of factors [11, Theorem 3.1] since the reversed occurrence of F_x can be the prefix of any suffix in $F_1 \cdots F_{x-1}$, and thus fulfills the suffix-closed property [11, Definition 2.2].

Kolpakov and Kucherov [19] also gave an algorithm computing the reversed LZ factorization in $O(n \lg \sigma)$ time using $O(n \lg n)$ bits of space, by applying Weiner's suffix tree construction algorithm [39] on the reversed text $\overleftarrow{T}$. Later, Sugimoto et al. [36] presented an online factorization algorithm running in $O(n \lg^2 \sigma)$ time using $O(n \lg \sigma)$ bits of space. Recently, Köppl [21] improved the time complexity to $O(n)$ while keeping the space usage of $O(n \lg \sigma)$ bits. While the last approach works offline, the former results are online algorithms with amortized time bounds per letter.

7.1 Time complexities for suffix tree traversal

In the next sections, besides maintaining the suffix tree online using Weiner's algorithm, we will need to navigate in the suffix tree in parallel, performing string matching. Thus, we will need to support the following query.

PatternMatch
Input: suffix tree $\mathsf{ST}(T[1..j-1])$, locus of $P \in \Sigma^*$ in it, letter $T[j]$, and a letter c.
Output: $\mathsf{ST}(T[1..j])$ and the locus of Pc in it.

This problem asks, for a given node v and letter c, to retrieve v's child edge whose first letter matches c. Let t_{trav} denote the time complexity for solving PatternMatch. For all implementations of SuffixUpdate mentioned in Section 2 (see Table 2), a navigation step is subsumed by the tree update step. This is because a tree update step includes creation of a new leaf which, in most implementations, subsumes the parent-to-child access. Table 3 summarizes the time bounds on t_{trav} implied by different implementations of SuffixUpdate. In conclusion, we can solve PatternMatch in $O(t_{\mathrm{trav}}) = O(t_{\mathrm{SU}})$ time per letter, which we use hereafter as a black box to compute the reversed LZ factorizations online in $O(t_{\mathrm{SU}})$ time per letter.

7.2 Reversed LZ factorization without self-references

We follow the ideas of the algorithm of [19] to show how to compute the reversed LZ factorizations without self-references online in $O(t_{\mathrm{SU}})$ worst-case time per processed letter. As in the previous sections, we maintain the suffix tree of the reversed text $\overleftarrow{T}$ online using Weiner's algorithm. The new idea is that we require the capability of pattern matching

Table 3 Time complexities t_{SU} of Table 2 and t_{trav} for solving SuffixUpdate and PatternMatch respectively. We support PatternMatch via a parent-to-child traversal, for which each internal node v is augmented by an associative array that maps the first character of each outgoing edge from v to the child node that edge leads to.

SuffixUpdate		PatternMatch		
t_{SU}	Ref.	t_{trav}	Ref.	caveat
$O(\lg n)$	[1]	$O(\log \sigma)$	binary search	
$O(\sigma \log\log n)$	[6]	—"—	—"—	
$O(\log\log n + \log\log \sigma)$	[20]	$O(\log\log \sigma)$ expected	y-fast trie [40]	randomized
$O(\log\log n + \frac{\log^2 \log \sigma}{\log\log\log \sigma})$	[13]	$O(\frac{\log^2 \log \sigma}{\log\log\log \sigma})$	Wexp trie [13]	
$O(\log\log n)$	[22]	$O(1)$	q-heaps [16]	$\sigma = O(\log^{1/4} n)$

coupled with tree construction. In parallel to extending the suffix tree, we are matching a pattern which is the LZ factor we are currently parsing. The task is to find the locus of the longest prefix of the pattern that can be read by traversing the suffix tree top-down. For that, we maintain the currently computed locus of a matching prefix of the pattern. After reading a new letter of the text, we match this letter by making one traversal step of the suffix tree in time $O(t_{\mathrm{trav}})$. If the letter cannot be matched, the parsing of the current factor terminates and the new one starts. Then the suffix tree is extended with the new letter.

The important point, however, is that we are not allowed to traverse nodes that have been created after we started the pattern matching process. To ensure that, we borrow the idea of timestamping suffix tree nodes [2, Section 3]. Namely, we store a timestamp in each node of $\mathsf{ST}(\overleftarrow{T})$ indicating when it has been created, which is identical to the largest suffix of $\overleftarrow{T}$ at the leaves of its subtree. By doing so, we can abort the pattern matching process if we reach a node whose timestamp is larger than the ending position of the previous factor. The timestamps can be maintained while constructing the suffix tree with Weiner's algorithm in constant additional time since the suffix length of a newly created leaf is known at the time of its creation, and affects only the timestamp of its parent node.

Recall that if we follow the Breslauer-Italiano approach to maintaining weak W-links, we deamortize the update of multiple weak W-links at an individual round by updating a constant number (at least two) of weak W-links at each round in the order of increasing depth. Importantly, this process does not interfere with the pattern matching process we are doing in parallel, as the latter only requires the timely creation of new nodes which in turn is governed by hard W-links alone updated in real time.

We conclude with the following result.

▶ **Theorem 10.** *We can compute the reversed LZ factorization of a string $T[1..n]$ online in $O(n)$ space and $O(t_{\mathrm{SU}})$ worst-case time per letter.*

7.3 Reversed LZ factorization with self-references

Sugimoto et al. [36] presented a variant of the reversed LZ factorization called the *reversed LZ factorization with self-references*. In that version, a factor F_x and its inverted copy $\overleftarrow{F_x}$ are allowed to overlap [36, Definition 4]. In detail, a factorization $F_1 \cdots F_z = T$ is the *overlapping reversed LZ factorization* of T if each next factor F_x is the first occurrence of a letter or the longest prefix of $F_x \cdots F_z$ with the property that each non-empty prefix F of F_x (including

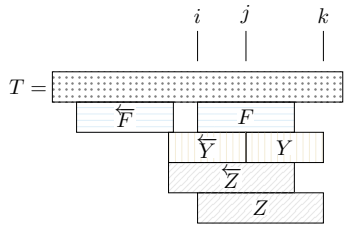

Figure 4 Illustration of the reversed LZ factorization with self-references. We maintain both the longest non-overlapping reversed LZ factor F starting at position i and the longest palindromic suffix $\overleftarrow{Y}Y$ starting before i. The one which extends further right defines the factor. Here the suffix palindrome $\overleftarrow{Y}Y$ defines the new factor Z.

F_x itself) has an inverted copy $\overleftarrow{F}$ occurring in $F_1 \cdots F_{x-1}F$ as a substring[1]. Observe that a factor F_x overlaps its inverted copy $\overleftarrow{F_x}$ if and only if $F_1 \cdots F_x$ has a palindromic suffix of length between $|F_x|$ and $2|F_x| - 1$.

While the reversed LZ factorization with self-references cannot be directly used to encode the string, it can still be computed online using our technique together with Manacher's algorithm [23]. Manacher's algorithm computes, in linear time, all maximal palindromes in a string. It reads the input string online and maintains as an invariant the longest palindromic suffix of the current string. By a standard deamortization argument, Manacher's algorithm can be made real time [15], spending constant worst-case time on each new letter.

The idea is then to run Manacher's algorithm in parallel with the algorithm of Theorem 10. Maintaining the longest palindromic suffix covers the case of self-overlapping factor whereas the algorithm of Theorem 10 covers the case of non-overlapping reversed factor. We then keep at each step the longest factor between those obtained by the two algorithms. See Figure 4 for an illustration.

▶ **Theorem 11.** *We can compute the reversed LZ factorization with self-references of a string $T[1..n]$ online in $O(n)$ space and $O(t_{\mathrm{SU}})$ worst-case time per letter.*

8 Conclusion

We studied applications of the Breslauer–Italiano deamoritization of Weiner's algorithm to obtain near-real-time online algorithms for several string processing problems. The crucial observation was that the deamortization of the suffix tree maintenance algorithm does not interfere with the functionality of the suffix tree needed to solve these problems. We gave online algorithms computing the longest repeating suffixes, the Lempel–Ziv factorization, the set of minimal unique substrings, and the reversed Lempel–Ziv factorization (with and without self-references) in $O(n)$ space and $O(t_{\mathrm{SU}})$ worst-case time per letter, where t_{SU} depends on the implementation of the suffix tree construction algorithm but is at most $O(\log^2 \log n)$. We believe that this approach can be applied to obtain near-real-time online algorithms for other problems on strings, which will be the subject of our forthcoming work.

[1] Our definition slightly differs from that of [36] in that the inverted copy does not have to end *before* the end of F_x, i.e., F_x itself can be a palindrome.

References

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

2 Amihood Amir, Gad M. Landau, and Esko Ukkonen. Online timestamped text indexing. *Inf. Process. Lett.*, 82(5):253–259, 2002. doi:10.1016/S0020-0190(01)00275-7.

3 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. In *Proc. CPM*, volume 78 of *LIPIcs*, pages 22:1–22:18, 2017. doi:10.4230/LIPIcs.CPM.2017.22.

4 Widmer Bland, Gregory Kucherov, and W. F. Smyth. Prefix table construction and conversion. In Thierry Lecroq and Laurent Mouchard, editors, *Combinatorial Algorithms - 24th International Workshop, IWOCA 2013, Rouen, France, July 10-12, 2013, Revised Selected Papers*, volume 8288 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2013. doi:10.1007/978-3-642-45278-9_5.

5 Anselm Blumer, J. Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel I. Seiferas. The smallest automaton recognizing the subwords of a text. *Theor. Comput. Sci.*, 40:31–55, 1985. doi:10.1016/0304-3975(85)90157-4.

6 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.

7 Andrej Brodnik, Svante Carlsson, Erik D. Demaine, J. Ian Munro, and Robert Sedgewick. Resizable arrays in optimal time and space. In *Proc. WADS*, volume 1663 of *LNCS*, pages 37–48, 1999. doi:10.1007/3-540-48447-7_4.

8 Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, 1994.

9 Richard Cole and Ramesh Hariharan. Dynamic LCA queries on trees. *SIAM J. Comput.*, 34(4):894–923, 2005. doi:10.1137/S0097539700370539.

10 Richard Cole, Tsvi Kopelowitz, and Moshe Lewenstein. Suffix trays and suffix trists: Structures for faster text indexing. *Algorithmica*, 72(2):450–466, 2015. doi:10.1007/s00453-013-9860-6.

11 Maxime Crochemore, Alessio Langiu, and Filippo Mignosi. Note on the greedy parsing optimality for dictionary-based text compression. *Theor. Comput. Sci.*, 525:55–59, 2014. doi:10.1016/j.tcs.2014.01.013.

12 Paul F. Dietz and Rajeev Raman. Persistence, amortization and randomization. In *Proc. SODA*, pages 78–88, 1991. URL: http://dl.acm.org/citation.cfm?id=127787.127809.

13 Johannes Fischer and Pawel Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. CPM*, volume 9133 of *LNCS*, pages 160–171, 2015. doi:10.1007/978-3-319-19929-0_14.

14 Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proc. STOC*, pages 345–354, 1989. doi:10.1145/73007.73040.

15 Zvi Galil. Real-time algorithms for string-matching and palindrome recognition. In *Proc. STOC*, pages 161–173, 1976. doi:10.1145/800113.803644.

16 Yoav Giora and Haim Kaplan. Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. *ACM Trans. Algorithms*, 5(3), 2009. doi:10.1145/1541885.1541889.

17 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

18 Alan Hartman and Michael Rodeh. Optimal parsing of strings. In *Combinatorial Algorithms on Words*, pages 155–167, Berlin, Heidelberg, 1985.

19 Roman Kolpakov and Gregory Kucherov. Searching for gapped palindromes. *Theor. Comput. Sci.*, 410(51):5365–5373, 2009. doi:10.1016/j.tcs.2009.09.013.

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

21 Dominik Köppl. Reversed Lempel–Ziv factorization with suffix trees. *Algorithms*, 14(6)(161):1–26, 2021. doi:10.3390/a14060161.

22 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.

23 Glenn K. Manacher. A new linear-time "on-line" algorithm for finding the smallest initial palindrome of a string. *J. ACM*, 22(3):346–351, 1975. doi:10.1145/321892.321896.

24 Takuya Mieno, Yuki Kuhara, Tooru Akagi, Yuta Fujishige, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Minimal unique substrings and minimal absent words in a sliding window. In *Proc. SOFSEM*, volume 12011 of *LNCS*, pages 148–160, 2020. doi:10.1007/978-3-030-38919-2_13.

25 Christian Worm Mortensen. Fully dynamic orthogonal range reporting on RAM. *SIAM J. Comput.*, 35(6):1494–1525, 2006. doi:10.1137/S0097539703436722.

26 Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM J. Comput.*, 43(5):1781–1806, 2014. doi:10.1137/130908245.

27 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows–Wheeler transform. In *Proc. IWOCA*, volume 10765 of *LNCS*, pages 409–419, 2017. doi:10.1007/978-3-319-78825-8_33.

28 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA*, volume 5193 of *LNCS*, pages 696–707, 2008. doi:10.1007/978-3-540-87744-8_58.

29 Alberto Policriti and Nicola Prezza. Fast online Lempel–Ziv factorization in compressed space. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 13–20, 2015. doi:10.1007/978-3-319-23826-5_2.

30 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/s00453-017-0327-z.

31 Nicola Prezza and Giovanna Rosone. Faster online computation of the succinct longest previous factor array. In *Proc. CiE*, volume 12098 of *LNCS*, pages 339–352, 2020. doi:10.1007/978-3-030-51466-2_31.

32 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

33 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Algorithms and Data Structures: The Basic Toolbox*. Springer International Publishing, 2019.

34 Tatiana Starikovskaya. Computing lempel–ziv factorization online. In *Proc. MFCS*, volume 7464 of *LNCS*, pages 789–799, 2012. doi:10.1007/978-3-642-32589-2_68.

35 James A. Storer and Thomas G. Szymanski. Data compression via textural substitution. *J. ACM*, 29(4):928–951, 1982. doi:10.1145/322344.322346.

36 Shiho Sugimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing reversed Lempel–Ziv factorization online. In *Proc. PSC*, pages 107–118, 2013. URL: http://www.stringology.org/event/2013/p10.html.

37 Takuya Takagi, Shunsuke Inenaga, Hiroki Arimura, Dany Breslauer, and Diptarama Hendrian. Fully-online suffix tree and directed acyclic word graph construction for multiple texts. *Algorithmica*, 82(5):1346–1377, 2020. doi:10.1007/s00453-019-00646-w.

38 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

39 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory (SWAT)*, pages 1–11, 1973. doi:10.1109/SWAT.1973.13.

40 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

41 Jun-ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster compact on-line lempel-ziv factorization. In *Proc. STACS*, volume 25 of *LIPIcs*, pages 675–686, 2014. doi:10.4230/LIPIcs.STACS.2014.675.

42 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

43 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.