algorithms

MDPI

Article

Reversed Lempel–Ziv Factorization with Suffix Trees †

Dominik Köppl

M&D Data Science Center, Tokyo Medical and Dental University, Tokyo 113-8510, Japan; koeppl.dsc@tmd.ac.jp; Tel.: +81-3-5280-8626

† Parts of this work have been published as part of a Ph.D. Thesis.

Abstract: We present linear-time algorithms computing the reversed Lempel–Ziv factorization [Kolpakov and Kucherov, TCS'09] within the space bounds of two different suffix tree representations. We can adapt these algorithms to compute the longest previous non-overlapping reverse factor table [Crochemore et al., JDA'12] within the same space but pay a multiplicative logarithmic time penalty.

Keywords: longest previous non-overlapping reverse factor table; application of suffix trees; reversed Lempel–Ziv factorization; lossless compression

Citation: Köppl, D. Reversed Lempel–Ziv Factorization with Suffix Trees. *Algorithms* **2021**, *14*, 161. http://doi.org/10.3390/a14060161

Academic Editor: Costas Busch and Shunsuke Inenaga

Received: 17 April 2021
Accepted: 20 May 2021
Published: 21 May 2021

1. Introduction

The non-overlapping reversed Lempel–Ziv (LZ) factorization was introduced by Kolpakov and Kucherov [1] as a helpful tool for detecting gapped palindromes, i.e., substrings of a given text T of the form $S^{\mathsf{R}}GS$ for two strings S and G, where S^{R} denotes the reverse of S. This factorization is defined as follows: Given a factorization $T = F_1 \cdots F_z$ for a string T, it is the non-overlapping reversed LZ factorization of T if each factor F_x, for $x \in [1\,..\,z]$, is either the leftmost occurrence of a character or the longest prefix of $F_x \cdots F_z$ whose reverse has an occurrence in $F_1 \cdots F_{x-1}$. It is a greedy parsing in the sense that it always selects the longest possible such prefix as the candidate for the factor F_x. The factorization can be written like a macro scheme [2], i.e., by a list storing either plain characters or pairs of referred positions and lengths, where a referred position is a previous text position from where the characters of the respective factor can be borrowed. Among all variants of such a left-to-right parsing using the reversed as a reference to the formerly parsed part of the text, the greedy parsing achieves optimality with respect to the number of factors [3] ([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 [3] ([Definition 2.2]).

Kolpakov and Kucherov [1] also gave an algorithm computing the reversed LZ factorization in $\mathcal{O}(n \lg \sigma)$ time using $\mathcal{O}(n \lg n)$ bits of space, by applying Weiner's suffix tree construction algorithm [4] on the reversed text T^{R}. Later, Sugimoto et al. [5] presented an online factorization algorithm running in $\mathcal{O}(n \lg^2 \sigma)$ time using $\mathcal{O}(n \lg \sigma)$ bits of space. We can also compute the reversed LZ factorization with the longest previous non-overlapping reverse factor table LPnrF storing the longest previous non-overlapping reverse factor for each text position. There are algorithms [6–10] computing LPnrF in linear time for strings whose characters are drawn from alphabets with constant sizes; their used data structures include the suffix automaton [11], the suffix tree of T^{R}, the position heap [12], and the suffix heap [13]. Finally, Crochemore et al. [14] presented a linear-time algorithm working with integer alphabets by leveraging the suffix array [15]. To find the longest gapped palindromes of the form $S^{\mathsf{R}}GS$ with the length of G restricted in a given interval $\mathcal{I}$, Dumitran et al. [16] ([Theorem 1]) restricted the distance of the previous reverse occurrence relative to the starting position of the respective factor within $\mathcal{I}$ in their modified definition of LPnrF, and achieved the same time and space bounds of [14]. However, all mentioned linear-time approaches use either pointer-based data structures of $\mathcal{O}(n \lg n)$ bits, or multiple integer arrays of length n to compute LPnrF or the reversed LZ factorization.

1.1. Our Contribution

The aim of this paper is to compute the reversed LZ factorization in less space while retaining the linear time bound. For that, we follow the idea of Crochemore et al. [14] ([Section 4]) who built text indexing data structures on $T \cdot \# \cdot T^{\mathsf{R}}$ to compute LPnrF for an artificial character #. However, they need random access to the suffix array, which makes it hard to achieve linear time for working space bounds within $o(n \lg n)$ bits. We can omit the need for random access to the suffix array by a different approach based on suffix tree traversals. As a precursor of this line of research we can include the work of Gusfield [17] ([APL16]) and Nakashima et al. [18]. The former studies the non-overlapping Lempel–Ziv–Storer–Szymanski (LZSS) factorization [2,19] while the latter the Lempel–Ziv-78 factorization [20]. Although their used techniques are similar to ours, they still need $\mathcal{O}(n \lg n)$ bits of space. To actually improve the space bounds, we follow two approaches: On the one hand, we use the leaf-to-root traversals proposed by Fischer et al. [21] ([Section 3]) for the overlapping LZSS factorization [2] during which they mark visited nodes acting as signposts for candidates for previous occurrences of the factors. On the other hand, we use the root-to-leaf traversals proposed in [22] for the leaves corresponding to the text positions of T to find the lowest marked nodes whose paths to the root constitute the lengths of the non-overlapping LZSS factors. Although we mimic two approaches for computing factorizations different to the reversed LZ factorization, we can show that these traversals on the suffix tree of $T \cdot \# \cdot T^{\mathsf{R}}$ help us to detect the factors of the reversed LZ factorization. Our result is as follows:

Theorem 1. *Given a text T of length $n-1$ whose characters are drawn from an integer alphabet with size $\sigma = n^{\mathcal{O}(1)}$, we can compute its reversed LZ factorization*

- *in $\mathcal{O}(\epsilon^{-1}n)$ time using $(2+\epsilon)n \lg n + \mathcal{O}(n)$ bits (excluding the read-only text T), or*
- *in $\mathcal{O}(\epsilon^{-1}n)$ time using $\mathcal{O}(\epsilon^{-1}n \lg \sigma)$ bits,*

for a selectable parameter $\epsilon \in (0, 1]$.

On the downside, we have to admit that the results are not based on new tools, but rather a combination of already existing data structures with different algorithmic ideas. On the upside, Theorem 1 presents the first linear-time algorithm computing the reversed LZ factorization using a number of bits linear to the input text T, which is $o(n \lg n)$ bits for $\lg \sigma = o(\lg n)$. Interestingly, this has not yet been achieved for the seemingly easier non-overlapping LZSS factorization, for which we have $\mathcal{O}(\epsilon^{-1}n \log_\sigma^\epsilon n)$ time within the same space bound [22] ([Theorem 1]). We can also adapt the algorithm of Theorem 1 to compute LPnrF, but losing the linear time for the $\mathcal{O}(n \lg \sigma)$-bits solution:

Theorem 2. *Given a text T of length $n-1$ whose characters are drawn from an integer alphabet with size $\sigma = n^{\mathcal{O}(1)}$, we can compute a $2n$-bits representation of its longest previous non-overlapping reverse factor table* LPnrF

- *in $\mathcal{O}(\epsilon^{-1}n)$ time using $(2+\epsilon)n \lg n + \mathcal{O}(n)$ bits (excluding the read-only text T), or*
- *in $\mathcal{O}(\epsilon^{-1}n \log_\sigma^\epsilon n)$ time using $\mathcal{O}(\epsilon^{-1}n \lg \sigma)$ bits,*

for a selectable parameter $\epsilon \in (0, 1]$. We can augment our LPnrF *representation with an $o(n)$-bits data structure to provide constant-time random access to* LPnrF *entries.*

We obtain the $2n$-bits representation of LPnrF with the same compression technique used for the permuted longest common prefix array [23] ([Theorem 1]), see [24] ([Definition 4]) for several other examples.

1.2. Related Work

To put the above theorems into the context of space-efficient factorization algorithms that can also compute factor tables like LPnrF, we briefly list some approaches for different variants of the LZ factorization and of LPnrF. We give Table 1 as an overview. We are

aware of approaches to compute the overlapping and non-overlapping LZSS factorization, as well as the longest previous factor (LPF) table LPF [25,26] and the longest previous non-overlapping table LPnF [14]. We can observe in Table 1 that only the overlapping LZSS factorization does not come with a multiplicative $\log_\sigma^\epsilon n$ time penalty when working within $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits. Note that the time and space bounds have an additional multiplicative ϵ^{-1} penalty (unlike described in the references therein) because the currently best construction algorithms of the compressed suffix tree (described later in Section 2) works in $\mathcal{O}(\epsilon^{-1} n)$ time and needs $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits of space [27] ([Section 6.1]).

Regarding space-efficient algorithms computing the LZSS factorization, we are aware of the linear-time algorithm of Goto and Bannai [28] using $n \lg n + \mathcal{O}(\sigma \lg n)$ bits of working space. For ϵn bits of space, Kärkkäinen et al. [29] can compute the factorization in $\mathcal{O}(n \lg n \lg \lg \sigma)$ time, which got improved to $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ by Kosolobov [30]. Finally, the algorithm of Belazzougui and Puglisi [31] uses $\mathcal{O}(n \lg \sigma)$ bits of working space and $\mathcal{O}(n \lg \lg \sigma)$ time.

Another line of research is the online computation of LPF. Here, Okanohara and Sadakane [32] gave a solution that works in $n \lg \sigma + \mathcal{O}(n)$ bits of space and needs $\mathcal{O}(n \lg^3 n)$ time. This time bound got recently improved to $\mathcal{O}(n \lg^2 n)$ by Prezza and Rosone [33].

Table 1. Complexity bounds of related approaches described in Section 1.2 for a selectable parameter $\epsilon \in (0, 1]$.

$(1+\epsilon)n \lg n + \mathcal{O}(n)$ Bits of Working Space (Excluding the Read-Only Text T)		
Reference	**Type**	**Time**
[21] ([Corollary 3.7])	overlapping LZSS	$\mathcal{O}(\epsilon^{-1} n)$
[34] ([Lemma 6])	LPF	$\mathcal{O}(\epsilon^{-1} n)$
[22] ([Theorem 1])	non-overlapping LZSS	$\mathcal{O}(\epsilon^{-1} n)$
[22]([Theorem 3])	LPnF	$\mathcal{O}(\epsilon^{-1} n)$
$\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ Bits of Working Space		
Reference	**Type**	**Time**
[21] ([Corollary 3.4])	overlapping LZSS	$\mathcal{O}(\epsilon^{-1} n)$
[34] ([Lemma 6])	LPF	$\mathcal{O}(\epsilon^{-1} n \log_\sigma^\epsilon n)$
[22] ([Theorem 1])	non-overlapping LZSS	$\mathcal{O}(\epsilon^{-1} n \log_\sigma^\epsilon n)$
[22] ([Theorem 3])	LPnF	$\mathcal{O}(\epsilon^{-1} n \log_\sigma^\epsilon n)$

1.3. Structure of this Article

This article is structured as follows: In Section 2, we start with the introduction of the suffix tree representations we build on the string $T \cdot \# \cdot T^{\mathsf{R}}$, and introduce the reversed LZ factorization in Section 3. We present in Section 3.2 our solution for the claim of Theorem 1 without the referred positions, which we compute subsequently in Section 3.3. Finally, we introduce LPnrF in Section 4, and give two solutions for Theorem 2. One is a derivation of our reversed-LZ factorization algorithm of Section 3.2.2 (cf. Section 4.1), the other is a translation of [14] ([Algorithm 2]) to suffix trees (cf. Section 4.2).

2. Preliminaries

With lg we denote the logarithm $\log_2$ to base two. Our computational model is the word RAM model with machine word size $\Omega(\lg n)$ for a given input size n. Accessing a word costs $\mathcal{O}(1)$ time.

Let T be a text of length $n-1$ whose characters are drawn from an integer alphabet $\Sigma = [1 \,..\, \sigma]$ with $\sigma = n^{\mathcal{O}(1)}$. Given $X, Y, Z \in \Sigma^*$ with $T = XYZ$, then X, Y and Z are called a prefix, substring and suffix of T, respectively. We call $T[i\,..]$ the i-th suffix of T, and denote a substring $T[i]T[i+1] \cdots T[j]$ with $T[i \,..\, j]$. For $i > j$, $[i \,..\, j]$ and $T[i \,..\, j]$ denote the empty set and the empty string, respectively. The reverse T^{R} of T is the concatenation $T^{\mathsf{R}} := T[n-1]T[n-2] \cdots T[1]$. We further write $T[i \,..\, j]^{\mathsf{R}} := T[j]T[j-1] \cdots T[i]$.

Given a character $\mathsf{c} \in \Sigma$, and an integer j, the rank query $T.\mathrm{rank}_{\mathsf{c}}(j)$ counts the occurrences of c in $T[1\,..\,j]$, and the select query $T.\mathrm{select}_{\mathsf{c}}(j)$ gives the position of the j-th c in T, if it exists. We stipulate that $\mathrm{rank}_{\mathsf{c}}(0) = \mathrm{select}_{\mathsf{c}}(0) = 0$. If the alphabet is binary, i.e., when T is a bit vector, there are data structures [35,36] that use $o(|T|)$ extra bits of space, and can compute rank and select in constant time, respectively. There are representations [37] with the same constant-time bounds that can be constructed in time linear in $|T|$. We say that a bit vector has a rank-support and a select-support if it is endowed by data structures providing constant time access to rank and select, respectively.

From now on, we assume that there exist two special characters # and \$ that do not appear in T, with $\$ < \# < c$ for every character $c \in \Sigma$. Under this assumption, none of the suffixes of $T \cdot \#$ and $T^{\mathsf{R}} \cdot \$$ has another suffix as a prefix. Let $R := T \cdot \# \cdot T^{\mathsf{R}} \cdot \$$. R has length $|R| = 2|T| + 2 = 2n$.

The suffix tree ST of R is the tree obtained by compacting the suffix trie, which is the trie of all suffixes of R. ST has $2n$ leaves and at most $2n - 1$ internal nodes. The string stored in a suffix tree edge e is called the label of e. The children of a node v are sorted lexicographically with respect to the labels of the edges connecting the children with v. We identify each node of the suffix tree by its pre-order number. We do so implicitly such that we can say, for instance, that a node v is marked in a bit vector B, i.e., $B[v] = 1$, but actually have $B[i] = 1$, where i is the pre-order number of v. The string label of a node v is defined as the concatenation of all edge labels on the path from the root to v; v's string depth, denoted by $\mathrm{str\_depth}(v)$, is the length of v's string label. The operation $\mathrm{suffixlink}(v)$ returns the node with string label $S[2\,..\,]$ or the root node, given that the string label of v is S with $|S| \geq 2$ or a single character, respectively. suffixlink is undefined for the root node.

The leaf corresponding to the i-th suffix $R[i\,..]$ is labeled with the suffix number $i \in [1\,..\,2n]$. We write $\mathrm{sufnum}(\lambda)$ for the suffix number of a leaf λ. The leaf-rank is the preorder rank ($\in [1\,..\,2n]$) of a leaf among the set of all ST leaves. For instance, the leftmost leaf in ST has leaf-rank 1, while the rightmost leaf has leaf-rank $2n$. To avoid confusing the leaf-rank with the suffix number of a leaf, let us bear in mind that the leaf-ranks correspond to the lexicographical order of the suffixes (represented by the leaves) in R, while the suffix numbers induce a ranking based on the text order of R's suffixes. In this context, the function $\mathrm{suffixlink}(\lambda)$ returns the leaf whose suffix number is $\mathrm{sufnum}(\lambda) + 1$. The reverse function of suffixlink on leaves is $\mathrm{prev\_leaf}(\lambda)$ that returns the leaf whose suffix number is $\mathrm{sufnum}(\lambda) - 1$, or $2n$ if $\mathrm{sufnum}(\lambda) = 1$ (We do not need to compute $\mathrm{suffixlink}(\lambda)$ for a leaf with $\mathrm{sufnum}(\lambda) = 2n$, but want to compute $\mathrm{prev\_leaf}(\lambda)$ for the border case $\mathrm{sufnum}(\lambda) = 1$.).

In this article, we focus on the following two ST representations: the compressed suffix tree (CST) [23,38] and the succinct suffix tree (SST) [21] ([Section 2.2.3]). Both can be computed in $\mathcal{O}(\epsilon^{-1}n)$ time, where the former is due to a construction algorithm given by Belazzougui et al. [27] ([Section 6.1]), and the latter due to [21] ([Theorem 2.8]), see Table 2. These two representations provide some of the above described operations in the time bounds listed in Table 3. Each representation additionally stores the pointer smallest_leaf to the leaf with suffix number 1, and supports the following operations in constant time, independent of ϵ:

leaf_rank(λ) returns the leaf-rank of the leaf λ;

depth(v) returns the depth of the node v, which is the number of nodes on the path between v and the root (exclusive) such that root has depth zero;

level_anc(λ, d) returns the level-ancestor of the λ on depth d; and

lca(u, v) returns the lowest common ancestor (LCA) of u and v.

As previously stated, we implicitly represent nodes by their pre-order numbers such that the above operations actually take pre-order numbers as arguments.

Table 2. Construction time and needed space in bits for the succinct suffix tree (SST) and compressed suffix tree (CST) representations, cf. [21] ([Section 2.2]).

	SST	CST
Time	$\mathcal{O}(n/\epsilon)$	$\mathcal{O}(\epsilon^{-1}n)$
Space	$(2+\epsilon)n\lg n + \mathcal{O}(n)$	$\mathcal{O}(\epsilon^{-1}n\lg\sigma)$

Table 3. Time bounds for certain operations needed by our LZ factorization algorithms. Although not explicitly mentioned in [21], the time for prev_leaf is obtained with the Burrows–Wheeler transform [39] stored in the CST [38] ([A.1]) by constant-time partial rank queries, see [27] ([Section 3.4]) or [38] ([A.4]).

Operation	SST Time	CST Time
sufnum(λ)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(n)$
str_depth(v)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(\text{str\_depth}(v))$
suffixlink(v)	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(1)$
prev_leaf	$\mathcal{O}(1/\epsilon)$	$\mathcal{O}(1)$

3. Reversed LZ Factorization

A factorization of T of size z partitions T into z substrings $F_1 \cdots F_z = T$. Each such substring F_x is called a factor. A factorization is called reversed LZ factorization if each factor F_x is either the leftmost occurrence of a character or the longest prefix of $F_x \cdots F_z$ that occurs at least once in $(F_1 \cdots F_{x-1})^{\mathsf{R}}$, for $x \in [1\,..\,z]$. A similar but much well-studied factorization is the non-overlapping LZSS factorization, where each factor F_x is either the leftmost occurrence of a character or the longest prefix of $F_x \cdots F_z$ that occurs at least once in $F_1 \cdots F_{x-1}$, for $x \in [1\,..\,z]$. See Figure 1 for an example and a comparison of both factorizations. In what follows, let z denote the number of reversed-LZ factors of T.

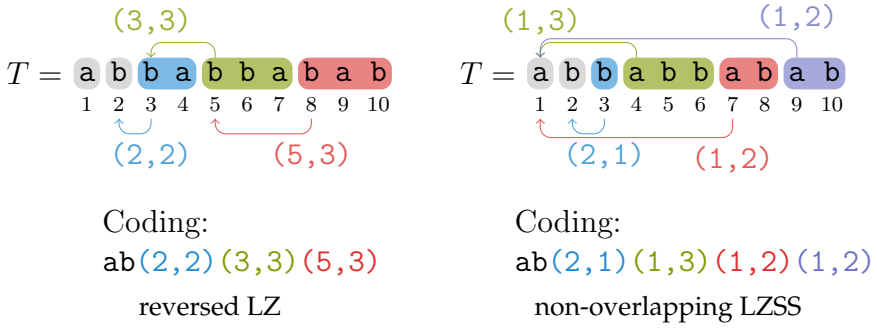


Figure 1. The reversed LZ and the non-overlapping LZSS factorization of the string $T =$ `abbabbabab`. A factor F is visualized by a rounded rectangle. Its coding consists of a mere character if it has no reference; otherwise, its coding consists of its referred position p and its length ℓ such that $F = T[p-\ell+1\,..\,p]^{\mathsf{R}}$ for the reversed LZ factorization, and $F = T[p\,..\,p+\ell-1]$ for the non-overlapping LZSS factorization.

3.1. Coding

We classify factors into fresh and referencing factors: We say that a factor is fresh if it is the leftmost occurrence of a character. We call all other factors referencing. A referencing factor F_x has a reference pointing to the ending position of its longest previous non-overlapping reverse occurrence; as a tie break, we always select the leftmost such ending position. We call this ending position the referred position of F_x. More precisely, the referred position of a factor $F_x = T[i\,..\,i+\ell-1]$ is the smallest text position j with $j \le i-1$ and $T[j-\ell+1\,..\,j]^{\mathsf{R}} = T[i\,..\,i+\ell-1]$. If we represent each referencing factor as a pair consisting of its referred position and its length, we obtain the coding shown in Figure 1.

Although our tie breaking rule selecting the leftmost position among all candidates for the referred position seems up to now arbitrary, it technically simplifies the algorithm in that we only have to index the very first occurrence.

3.2. Factorization Algorithm

In the following, we describe our factorization algorithm working with ST. This algorithm performs traversals on paths connecting leaves with the root, during which it marks certain nodes. One kind of these marked nodes are phrase leaves: A phrase leaf is a leaf whose suffix number is the starting position of a factor. We say that a phrase leaf λ corresponds to a factor F if the suffix number of ℓ is the starting position of F. We call all other leaves non-phrase leaves. Another kind are witnesses, a notion borrowed from [21] ([Section 3]): Witnesses are nodes that create a connection between referencing factors and their referred positions. We formally define them as follows: given λ is the phrase leaf corresponding to a referencing factor F, the witness w of F is the LCA of λ and a leaf with suffix number $2n - j$ (with $j \in [1\,..\,n-1]$) such that $T[j - \text{str\_depth}(w) + 1\,..\,j]^{\mathsf{R}}$ is the longest substring in $T[1\,..\,\text{sufnum}(\lambda) - 1]^{\mathsf{R}}$ that is a prefix of $T[\text{sufnum}(\lambda)\,..]$. The smallest such j is the referred position of λ, which is needed for the coding in Section 3.1. See Figure 2 for a sketch of the setting. In what follows, we show that the witness of a referencing factor F is the node whose string label is F. Generally speaking, for each substring S of T, there is always a node whose string label has S as a prefix, but there maybe no node whose string label is precisely S. This is in particular the case for the non-overlapping LZSS factorization [22] ([Section 3.1]). Here, we can make use of the fact that the suffix number $2n - j$ for a referred position j is always larger than the length of T, which we want to factorize:

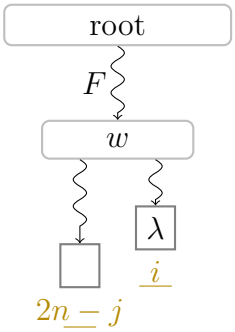


Figure 2. Witness node w of a referencing factor F starting at text position i. Given j is the referred position of F, the witness w of F is the node in the suffix tree having (a) F as a prefix of its string label and (b) the leaves with suffix numbers $2n - j$ and i in its subtree. Lemma 1 shows that w is uniquely defined to be the node whose string label is F.

Lemma 1. *The witness of each referencing factor exists and is well-defined.*

Proof. To show that each referencing factor is indeed the string label of an ST node, we review the definition of right-maximal repeats: A right-maximal repeat is a substring of R having at least two occurrences $R[i_1\,..\,i_1 + \ell - 1]$ and $R[i_2\,..\,i_2 + \ell - 1]$ with $R[i_1 + \ell] \neq R[i_2 + \ell]$. A right-maximal repeat is the string label of an ST node since this node has at least two children; those two children are connected by edges whose labels start with $R[i_1 + \ell]$ and $R[i_2 + \ell]$, respectively. It is therefore sufficient to show that each factor F is a right-maximal repeat. Given j is the referred position of $F = T[i\,..\,i + |F| - 1]$, $F = T[j - |F| + 1\,..\,j]^{\mathsf{R}} = R[2n - j\,..\,2n - j + |F| - 1]$. If $j = |F|$, then $T[i + |F|] \neq R[2n - j + |F|] = \$$, and thus F is a right-maximal repeat. For the other case that $j \geq |F| + 1$, assume that F is not a right-maximal repeat. Then $T[i + |F|] = R[2n - j + |F|] = T[j - |F|]$. However, this means that F is not the longest reversed factor being a prefix of $T[i\,..]$, a contradiction. We visualized the situation in Figure 3. □

Consequently, the referred position of a factor $F_x = T[i\,..\,i+\ell-1]$ is the smallest text position j in T with $j \leq i-1$ and one of the two equivalent conditions hold:

- $T[j-\ell+1\,..\,j]^{\mathsf{R}} = T[i\,..\,i+\ell-1]$; or
- $R[i\,..]$ and $R[2n-j\,..]$ have the longest common prefix of length ℓ.

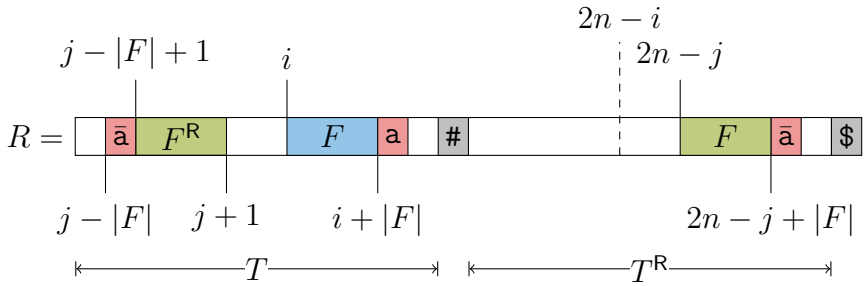


Figure 3. A reversed-LZ factor F starting at position i in R with a referred position $j \geq |F|+1$. If $\mathtt{a} = \bar{\mathtt{a}}$ with $\mathtt{a}, \bar{\mathtt{a}} \in \Sigma$, then we could extend F by one character, contradicting its definition to be the longest prefix of $T[i\,..]$ whose reverse occurs in $T[1\,..\,i-1]$. Hence, $\mathtt{a} \neq \bar{\mathtt{a}}$ and F is a right-maximal repeat.

3.2.1. Overview

We explain our factorization algorithm in terms of a cooperative game with two players (We use this notation only for didactic purposes; the terminology must not be confused with game theory. Here, the notion of player is basically a subroutine of the algorithm having private and shared variables.), whose pseudo code we sketched in Algorithm 1. Player 1 and Player 2 are allowed to access the leaves with suffix numbers in the ranges $[1\,..\,n]$ and $[n\,..\,2n-1]$, respectively. Player 1 (resp. Player 2) starts at the leaf with the smallest (resp. largest) suffix number, and is allowed to access the leaf with the subsequently next (resp. previous) suffix number via suffixlink (resp. prev_leaf). Hence, Player 1 simulates a linear forward scan in the text T, while Player 2 simulates a linear backward scan in T^{R}. Both players take turns at accessing leaves at the same pace. To be more precise, in the i-th turn, Player 1 processes the leaf with suffix number i, whereas Player 2 processes the leaf with suffix number $2n-i$. In one turn, a player accesses a leaf λ and maybe performs a traversal on the path connecting the root with λ. For such a traversal, we use level ancestor queries to traverse each node on the path in constant time. Whenever Player 2 accesses the leaf with suffix number n (shared among both players), the game ends; at that time both players access the same leaf (cf. Line 6 in Algorithm 1). In the following, we call this game a pass (with the meaning that we pass all relevant text positions). Depending on the allowed working space, our algorithm consists of one or two passes (cf. Section 3.3). The goal of Player 2 is to keep track of all nodes she visits. Player 2 does this by maintaining a bit vector B_{V} of length $4n$ such that $B_{\mathrm{V}}[v]$ stores whether a node v has already been visited by Player 2, where we represent a node v by its pre-order number when using it as an index of a bit vector. To keep things simple, we initially mark the root node in B_{V} at the start of each pass. By doing so, after the i-th turn of Player 2 we can read any substring of $T[1\,..\,i]^{\mathsf{R}}$ by a top-down traversal from the suffix tree root, only visiting nodes marked in B_{V}. This is because of the invariant that the set of nodes marked in B_{V} is upper-closed, i.e., if a node v is marked in B_{V}, then all its ancestors are marked in B_{V} as well.

The goal of Player 1 is to find the phrase leaves and the witnesses. For that, she maintains two bit vectors B_{L} and B_{W} of length n and $4n$, respectively, whose entries are marked similarly to B_{V} by using the suffix numbers ($\in [1\,..\,n]$) of the leaves accessed by Player 1 and preorder numbers of the internal nodes. We initially mark smallest_leaf in B_{L} since text position 1 is always the starting position of the fresh factor F_1. By doing so, after the i-th turn of Player 1 we know the ending positions of those factors contained in $T[1\,..\,i]$, which are marked in B_{L}. To sum up, after the i-th turn of both players we know the computed factors starting at text positions up to i thanks to Player 1, and can find the

factor lengths thanks to Player 2, which we explain in detail in Section 3.2.2. There, we will show that the actions of Player 2 allow Player 1 to determine the starting position of the next factor. For that, she computes the string depth of the lowest ancestor marked in B_V of the previously visited phrase leaf. See Appendix A.

As a side note: since we are only interested in the factorization of $T[1..n-1]$ (omitting the appended # at position n), we do not need Player 1 to declare the leaf with suffix number n a phrase leaf. We also terminate the algorithm when both players meet at position n without checking whether we have found a new factor starting at position n.

Algorithm 1: Algorithm of Section 3.2.2 computing the non-overlapping reversed LZ factorization. The function max_sufnum is described in Section 3.3.

```
1  ST ← suffix tree of R = T · # · T^R · $
2  λ^R ← prev_leaf(prev_leaf(smallest_leaf))                ▷ sufnum(λ^R) = 2n − 1
3  λ ← smallest_leaf
4  B_V[root node] ← 1          ▷ at the beginning, only the root node is marked in B_V
5  B_L[1] ← 1                                                ▷ |F_1| starts at text position 1
6  while λ ≠ λ^R do                                  ▷ we stop after having parsed T[1..n]
7      if B_L[sufnum(λ)] = 1 then                                          ▷ turn of Player 1
8          d ← 0
9          while B_V[level_anc(λ, d + 1)] = 1 do d ← d + 1
10         w ← level_anc(λ, d)                          ▷ w is lowest node marked in B_V
11         if w is the root then
12             output fresh factor
13             B_L[sufnum(λ) + 1] ← 1        ▷ next factor starts directly after sufnum(λ)
14         else                                                          ▷ w is the witness of λ
15             output length str_depth(w)
16             B_L[sufnum(λ) + str_depth(w)] ← 1
17             output referred position 2n − max_sufnum(w)          ▷ for the one-pass
                variant
18             B_W[w] ← 1                                          ▷ for the two-pass variant
19     λ ← suffixlink(λ)                                          ▷ end of Player 1's turn
20     foreach node v on the path from λ^R up to the root do                ▷ turn of Player 2
21         if B_V[v] = 1 then break     ▷ end turn on reaching an already marked node
22         B_V[v] ← 1
23     λ^R ← prev_leaf(λ^R)                                       ▷ end of Player 2's turn
```

3.2.2. One-Pass Algorithm in Detail

In detail, a pass works as follows: at the start, Player 1 and Player 2 select smallest_leaf and prev_leaf(prev_leaf(smallest_leaf)), i.e., the leaves with suffix numbers 1 and $2n-1$, respectively. Now the players take action in alternating turns, starting with Player 1. Nevertheless, we first explain the actions of Player 2, since Player 2 acts independently of Player 1, while Player 1's actions depend on Player 2.

Suppose that Player 2 is at a leaf λ^R (cf. Line 20 of Algorithm 1). Player 2 traverses the path from λ^R to the root upwards and marks all visited nodes in B_V until arriving at a node v already marked in B_V (such a node exists since we mark the root in B_V at the beginning of a pass.). When reaching the marked node v, we end the turn of Player 2, and move Player 2 to prev_leaf(λ^R) at Line 23 (and terminate the whole pass in Line 6 when this leaf has suffix number n). The `foreach` loop (Line 20) of the algorithm can be more verbosely expressed with a loop iterating over all depth offsets d in increasing order while computing $v \leftarrow$ level_anc(λ^R, d) until either reaching the root or a node marked in B_V. Subsequently, the turn of Player 1 starts (cf. Line 7). We depict the state after the first turn of Player 2 in Figure 4.

If Player 1 is at a non-phrase leaf λ, we skip the turn of Player 1, move Player 1 to suffixlink(λ) at Line 19, and let Player 2 take action. Now suppose that Player 1 is at a phrase leaf λ corresponding to a factor F. Then we traverse the path from the root to λ downwards to find the lowest ancestor w of λ marked in B_{V}. If w is the root node, then F is a fresh factor (cf. Line 11), and we know that the next factor starts immediately after F (cf. Line 13). Consequently, the leaf suffixlink(λ) is a phrase leaf. Otherwise, w is the witness of λ, and str_depth$(w) = |F|$ (cf. Line 14). Hence, sufnum(λ) + str_depth(w) is the suffix number of the phrase leaf $\tilde{\lambda}$ that Player 1 will subsequently access. We therefore mark w and sufnum$(\tilde{\lambda})$ = sufnum(λ) + str_depth(w) in B_{W} and in B_{L}, respectively (cf. Lines 16 and 18). We depict the fifth turn of our running example in Figure 5, during which Player 1 marks a witness node. Finally, we end the turn of Player 1, move Player 1 to suffixlink(λ) at Line 19, and let Player 2 take action.

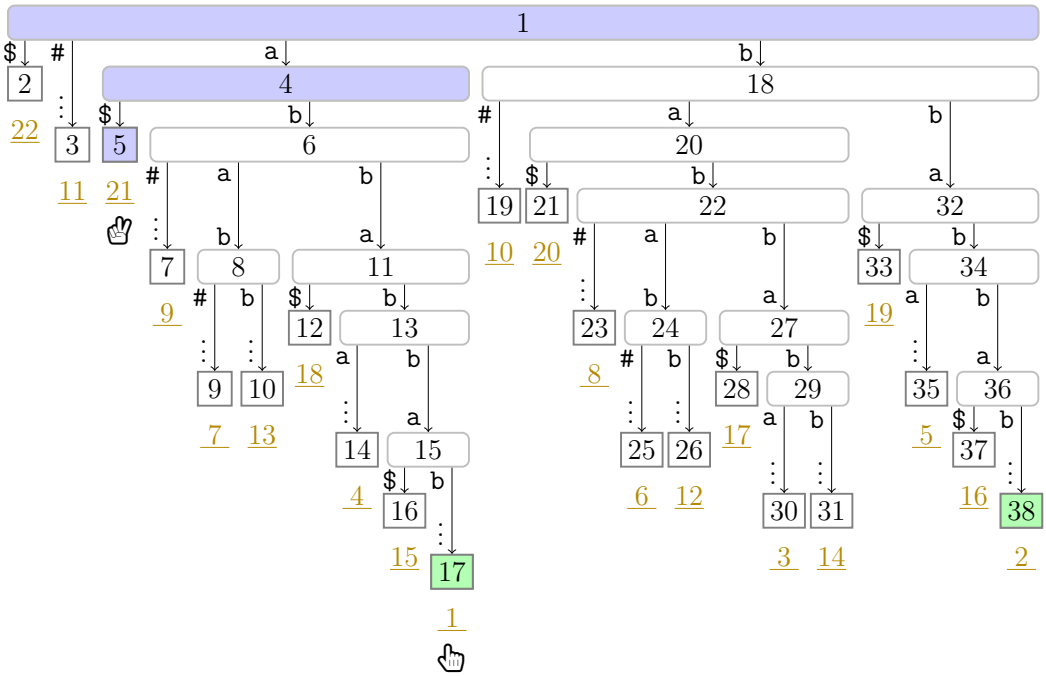


Figure 4. Suffix tree of $T\# \cdot T^{\mathrm{R}} \cdot \$$ used in Section 3.2, where $T =$ `abbabbabab` is our running example. The nodes are labeled by their preorder numbers. The suffix number of each leaf λ is the underlined number drawn in dark yellow below λ. We trimmed the label of each edge to a leaf having more than two characters and display only the first character and the vertical dots ':' as a sign of omission. The tree shows the state of Algorithm 1 after the first turn of both players. The nodes visited by Player 2 are colored in blue ( ), the phrase leaves are colored in green ( ). Player 1 and 2 are represented by the hands and , respectively, pointing to the respective leaves they visited during the first turn.

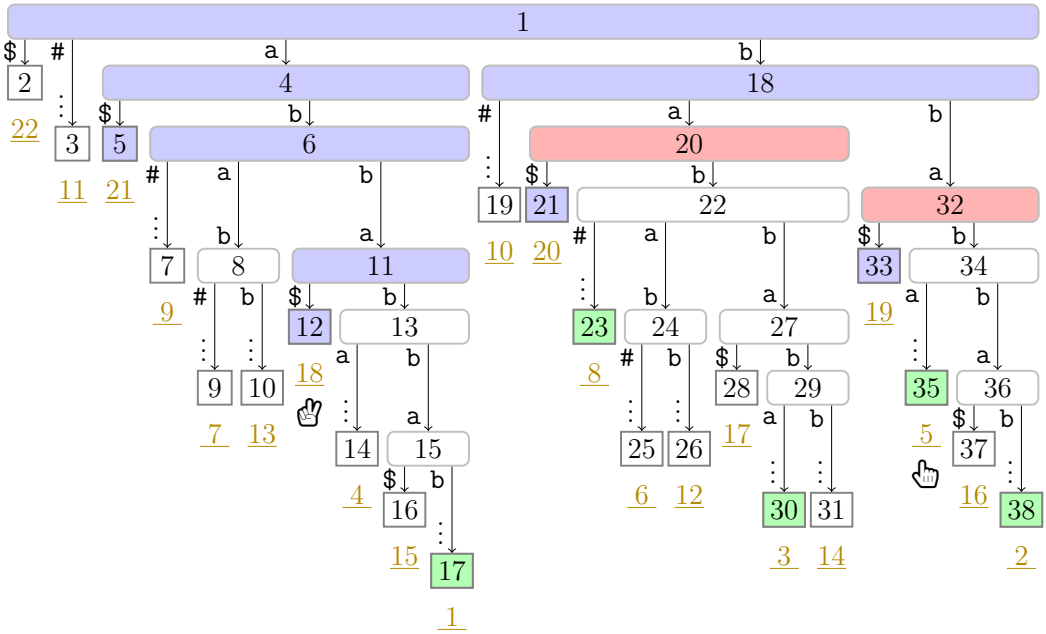


Figure 5. Continuation of Figure 4 with the state at the fifth turn of Player 1. Additionally to the coloring used in Figure 4, witnesses are colored in red ( ). In this figure, Player 1 just finished her turn on making the node with preorder number 32 the witness w of the leaf with suffix number 5. With w we know that the factor starting at text position 5 has the length str_depth(w) and that the next phrase leaf has suffix number 8. For visualization purposes, we left the hand ( ) of Player 2 below the leaf of her last turn.

Correctness. When Player 1 accesses the leaf λ with suffix number i, Player 2 has processed all leaves with suffix numbers $[2n-i+1\,..\,2n-1]$. Due to the leaf-to-root traversals of Player 2, each node marked in B_{V} has a leaf with a suffix number in $[2n-i+1\,..\,2n-1]$ in its subtree. In particular, a node w is marked in B_{V} if and only if the string label of w is a substring of $R[2n-i+1\,..\,2n-1]$. Because $R[2n-i+1\,..\,2n-1]^{\mathsf{R}} = T[1\,..\,i-1]$, the longest prefix of $T[i\,..]$ having a reversed occurrence in $T[1\,..\,i-1]$ is therefore one of the string labels of the nodes marked in B_{V}. In particular, we search the longest string label among those nodes, which we obtain with the lowest ancestor of λ marked in B_{V}.

3.2.3. Time Complexity

First, let us agree on that we never compute the suffix number of a leaf since this is a costly operation for CST (cf. Table 3). Although we need the suffix numbers at multiple occasions, we can infer them if each player maintains a counter for the suffix number of the currently visited leaf. A counter is initialized with 1 (resp. $2n-1$) and becomes incremented (resp. decremented) by one when moving to the succeeding (resp. preceding) leaf in suffix number order. This works since both players traverse the leaves linearly in the order of the suffix numbers (either in ascending or descending order).

Player 2 visits n leaves, and visits only unvisited nodes during a leaf-to-root traversal. Hence, Player 2's actions take $\mathcal{O}(n)$ overall time.

Player 1 also visits n leaves. Since Player 1 has no business with the non-phrase leaves, we only need to analyze the time spent by Player 1 for a phrase leaf corresponding to a factor F: If F is fresh, then the root-to-leaf traversal ends prematurely at the root, and hence we can determine in constant time whether F is fresh or not. If F is referencing, we descend from the root to the lowest ancestor w marked in B_{V}, and compute str_depth(w) to determine the suffix number of the next phrase leaf (cf. Line 15 of Algorithm 1). Since depth$(w) \leq$ str_depth(w), we visit at most $|F|+1$ nodes before reaching w. Computing str_depth(w) takes $\mathcal{O}(1/\epsilon)$ time for the SST, and $\mathcal{O}(|F|)$ time for the CST. This seems costly, but we compute str_depth(w) for each factor only once. Since the sum of all factor lengths

is n, we spend $\mathcal{O}(n + z/\epsilon)$ time or $\mathcal{O}(n)$ time for computing all factor lengths when using the SST or the CST, respectively. We finally obtain the time bounds stated in Theorem 1 for computing the factorization.

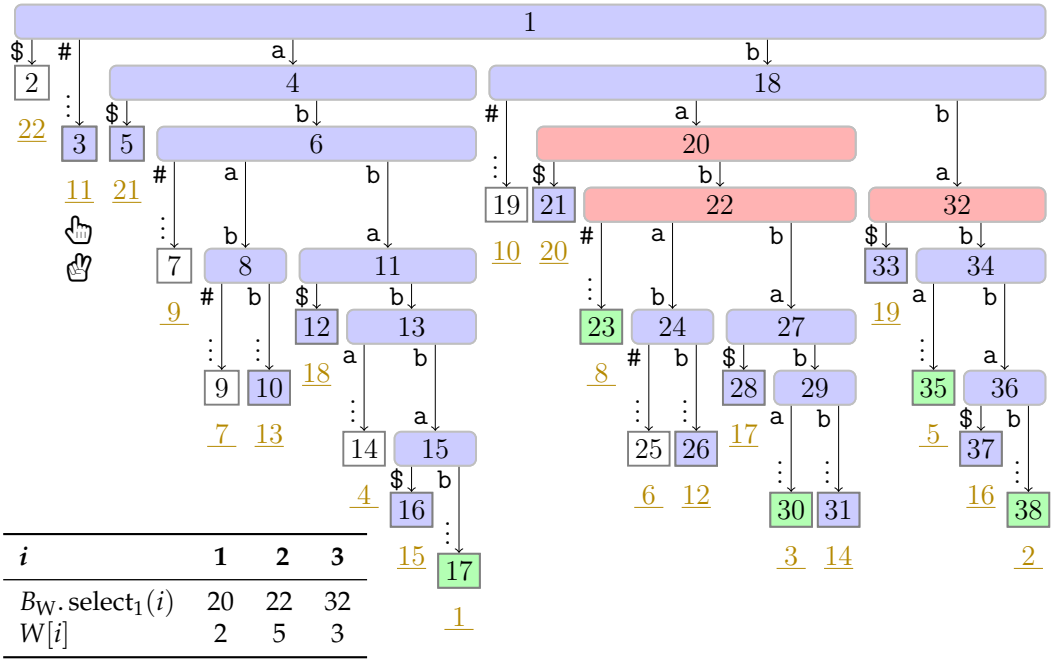
3.3. Determining the Referred Position

Up to now, we can determine the reversed-LZ factors $F_1 \cdots F_z = T$ with B_L marking the starting position of each factor with a one. Yet, we have not the referred positions necessary for the coding of the factors (cf. Section 3.1). To obtain them, we have two options: The first option is easier but comes with the requirement for a support data structure on ST for the operation

max_sufnum(v) returning the maximum among all suffix numbers of the leaves in the subtree rooted in v.

We can build such a support data structure in $\mathcal{O}(\epsilon^{-1}n)$ time (resp. $\mathcal{O}(\epsilon^{-1}n \log_\sigma^\epsilon n)$ time) using $\mathcal{O}(n)$ bits to support max_sufnum in $\mathcal{O}(\epsilon^{-1})$ time (resp. $\mathcal{O}(\epsilon^{-1} \log_\sigma^\epsilon n)$ time) for the SST (resp. CST); see [22] ([Section 3.3]). Being able to query max_sufnum, we can directly compute the referred position of a factor F when discovering its witness w during a turn of Player 1 by max_sufnum(w). max_sufnum(w) gives us the suffix number of a leaf that has already been accessed by Player 2 since Player 2 accesses the leaves in descending order with respect to the suffix numbers, and w must have already been accessed by Player 2 during a leaf-to-root traversal (otherwise w would not have been marked in B_V). Since $R[\text{max\_sufnum}(w) \,..\, \text{max\_sufnum}(w) + \text{str\_depth}(w) - 1] = F^\mathsf{R}$, the referred position of F is $2n - \text{max\_sufnum}(w)$. Consequently, we can compute the coding of the factors during a single pass (cf. Line 17 of Algorithm 1), and are done when the pass finishes.

The second option does not need to compute max_sufnum and retains the linear time bound when using CST. Here, the idea is to run an additional pass, whose pseudo code is given in Algorithm 2. For this additional pass, we do the following preparations: Let z_W be the number of witnesses, which is at most z since there can be multiple factors having the same witness. We keep B_L and B_W marking the phrase leaves and the witnesses, respectively. However, we clear B_V such that Player 2 has again the job to log her visited nodes in B_V. We augment B_W with a rank-support such that we can enumerate the witnesses with ranks from 1 to at most z_W, which we call the witness rank. We additionally create an array W of $z_\mathrm{W} \lg n$ bits. We want $W[B_\mathrm{W}.\mathrm{rank}_1(w)]$ to store the referred position $2n - \text{max\_sufnum}(w) \in [1 \,..\, n-1]$ for each witness w such that we can read the respective referred position from W when Player 1 accesses w. We assign the task for maintaining W to Player 2. Player 2 can handle this task by taking additional action when visiting a witness (i.e., a node marked in B_W) during a leaf-to-root traversal: When visiting a witness node w with witness rank i from a leaf λ, we write $W[i] \leftarrow 2n - \text{sufnum}(\lambda)$ if w is not yet marked in B_V (cf. Line 15 in Algorithm 2). Like before, Player 2 terminates her turn whenever she visits an already visited node. The actions of Player 1 differ in that she no longer needs to compute B_L and B_W: When Player 1 visits a phrase leaf λ, she locates the lowest ancestor w of λ marked in B_V, which has to be marked in B_W, too (as a side note: storing the depth of the witness of each phrase leaf in a list, sorted by the suffix numbers of these leaves, helps us to directly jump to the respective witness in constant time. We can encode this list as a bit vector of length $\mathcal{O}(n)$ by storing each depth in unary coding (cf. [22] ([Section 3.4])). Nevertheless, we can afford the root-to-witness traversals of Player 1 since we visit at most $\sum_{x=1}^{z} |F_x| = n$ nodes in total.). With the rank-support on B_W, we can compute w's witness rank i, and obtain the referred position of λ with $W[i]$ (cf. Line 10 of Algorithm 2). We show the final state after the first pass in Figure 6, together with W computed in the second pass.



i	1	2	3
$B_W.\text{select}_1(i)$	20	22	32
$W[i]$	2	5	3

Figure 6. State of our running example at termination of Algorithm 1. We have computed the bit vector B_L of length $n = 11$ storing a one at the entries $1, 2, 3, 5$, and 8, i.e., the suffix numbers of the phrase leaves, which are marked in green (■), and the bit vector B_W of length 38 (the maximum preorder number of an ST node) storing a one at the entries 20, 22, and 32, i.e., the preorder numbers of the witnesses, which are colored red (■). During the second pass described in Section 3.3, we compute W storing the referred positions in the order of the witness ranks (left table).

Overall, the time complexity is $\mathcal{O}(\epsilon^{-1}n)$ time when working with either the SST or the CST. We use $o(n)$ additional bits of space for the rank-support of B_W, but costly $z_W \lg n$ bits for the array W. However, we can bound z_W by $\mathcal{O}(n \lg \sigma / \lg n)$ since z_W is the number of distinct reversed LZ factors, and by an enumeration argument [40] ([Thm. 2]), a text of length n can be partitioned into at most $\mathcal{O}(n/\log_\sigma n)$ distinct factors. Hence, we can store W in $z_W \lg n = \mathcal{O}(n \lg \sigma)$ bits of space. With that, we finally obtain the working space bound of $\mathcal{O}(\epsilon^{-1} n \lg \sigma)$ bits for the CST solution as claimed in Theorem 1.

4. Computing LPnrF

The longest previous non-overlapping reverse factor table $\mathsf{LPnrF}[1\,..\,n]$ is an array such that $\mathsf{LPnrF}[i]$ is the length of the longest prefix of $T[i\,..] \cdot \#$ occurring as a substring of $T[1\,..\,i-1]^R$. (Appending # at the end is not needed, but simplifies the analysis for $T[1\,..\,n-1] \cdot \#$ having precisely n characters.) Having LPnrF, we can iteratively compute the reversed LZ factorization because $F_x = T[k_x\,..\,k_x + \max(0, \mathsf{LPnrF}[k_x] - 1)]$ with $k_x := 1 + \sum_{y=1}^{x-1} |F_y|$ for $x \in [1\,..\,z]$.

The counterpart of LPnrF for the non-overlapping LZSS factorization is the longest previous non-overlapping factor table $\mathsf{LPnF}[1\,..\,n]$, which is defined similarly, but stores the maximal length of the longest common prefix (LCP) of $T[i\,..]$ with all substrings $T[j\,..\,i-1]$ for $j \in [1\,..\,i-1]$. See Table 4 for a comparison. Analogously to [34] ([Corollary 5]) or [24] ([Definition 4]) for the longest previous factor table LPF [22,26] ([Lemma 1]) for LPnF, LPnrF holds the following property:

Lemma 2 ([14] (Lemma 2)). *$\mathsf{LPnrF}[i-1] - 1 \leq \mathsf{LPnrF}[i] \leq n - i$ for $i \in [2\,..\,n]$.*

Hence, we can encode LPnrF in $2n$ bits by writing the differences $\mathsf{LPnrF}[i] - \mathsf{LPnrF}[i-1] + 1 \geq 0$ in unary, obtaining a bit sequence of (a) n ones for the n entries and (b) $\sum_{i=2}^{n}(\mathsf{LPnrF}[i] - \mathsf{LPnrF}[i-1] + 1) \leq n$ many zeros. We can decode this bit sequence by reading the differences linearly because we know that $\mathsf{LPnrF}[1] = 0$.

Algorithm 2: Determining the referred positions in a second pass described in Section 3.3.

```
1  λ^R ← prev_leaf(prev_leaf(smallest_leaf)) and λ ← smallest_leaf
2  clear B_V and set B_V[root node] ← 1
3  W ← array of size z lg z
4  while λ ≠ λ^R do
5      if B_L[sufnum(λ)] = 1 then                                  ▷ turn of Player 1
6          d ← 0
7          while B_V[level_anc(λ, d + 1)] = 1 do d ← d + 1
8          w ← level_anc(λ, d)
9          if w is the root then output fresh factor
10         else output referred position W[B_W.rank_1(w)]          ▷ invariant: B_W[w] = 1
11     λ ← suffixlink(λ)                                           ▷ end of Player 1's turn
12     foreach node v on the path from λ^R up to the root do       ▷ turn of Player 2
13         if B_V[v] = 1 then break   ▷ end turn on reaching an already marked node
14         B_V[v] ← 1
15         if B_W[v] = 1 then W[B_W.rank_1(v)] = 2n − sufnum(λ^R)
16     λ^R ← prev_leaf(λ^R)                                        ▷ end of Player 2's turn
```

Table 4. LPnrF and LPnF of our running example. Both arrays are defined in Section 4. See Section 5 for the definition of LPrF.

i	1	2	3	4	5	6	7	8	9	10	11
T#	a	b	b	a	b	b	a	b	a	b	#
LPnrF	0	0	2	1	3	3	2	3	2	1	0
LPnF	0	0	1	3	3	3	2	3	2	1	0
LPrF	0	6	5	5	4	3	2	3	2	1	0

4.1. Adaptation of the Single-Pass Algorithm

Having an $\mathcal{O}(n)$-bits representation of LPnrF gives us hope to find an algorithm computing LPnrF in a total workspace space of $o(n \lg n)$ bits. Indeed, we can adapt our algorithm devised for the reversed LZ factorization to compute LPnrF. For that, we just have to promote all leaves to phrase leaves such that the condition in Line 7 of Algorithm 1 is always true. Consequently, Player 1 performs a root-to-leaf traversal for finding the lowest node marked in B_V of each leaf. By doing so, the time complexity becomes $\mathcal{O}(n^2)$, however, since we visit at most $\sum_{i=1}^{n} \mathsf{LPnrF}[i] = \mathcal{O}(n^2)$ many nodes during the root-to-leaf traversals (there are strings like $T = \mathtt{a} \cdots \mathtt{a}$ for which this sum becomes $\Theta(n^2)$).

To lower this time bound, we follow the same strategy as in [22] ([Section 3.5]) or [34] ([Lemma 6]) using suffixlink and Lemma 2: After Player 1 has computed $\text{str\_depth}(w) = \mathsf{LPnrF}[i-1]$ for w being the lowest ancestor marked in B_V of the leaf with suffix number $i-1$, we cache $\tilde{w} := \text{suffixlink}(w)$ for the next turn of Player 1 such that Player 1 can start the root-to-leaf traversal to the leaf $\tilde{\lambda}$ with suffix number i directly from $\tilde{w}$ and thus skips the nodes from the root to $\tilde{w}$. This works because $\tilde{w}$ is the ancestor of $\tilde{\lambda}$ with $\text{str\_depth}(\tilde{w}) = \mathsf{LPnrF}[i-1] - 1$, and $\tilde{w}$ must have been marked in B_V since $\mathsf{LPnrF}[i] \geq \text{str\_depth}(\tilde{w})$. See Figure 7 for a visualization. By skipping the nodes from the root to $\tilde{w}$, we visit only $\mathsf{LPnrF}[i] - \mathsf{LPnrF}[i-1] + 1$ many nodes during the i-th turn of Player 1. A telescoping sum together with Lemma 2 shows that Player 1 visits $\sum_{i=2}^{n}(\mathsf{LPnrF}[i] - \mathsf{LPnrF}[i-1] + 1) = \mathcal{O}(n)$ nodes in total.

The final bottleneck for CST are the n evaluations of $\text{str\_depth}(w)$ to compute the actual values of LPnrF (cf. Line 15 of Algorithm 1). Here, we use a support data structure on CST for str_depth [34] ([Lemma 6]), which can be constructed in $\mathcal{O}(\epsilon^{-1} n \log_\sigma^\epsilon n)$ time, uses $\mathcal{O}(n)$ bits of space, and answers str_depth in $\mathcal{O}(\epsilon^{-1} \log_\sigma^\epsilon n)$ time. This finally gives Theorem 2.

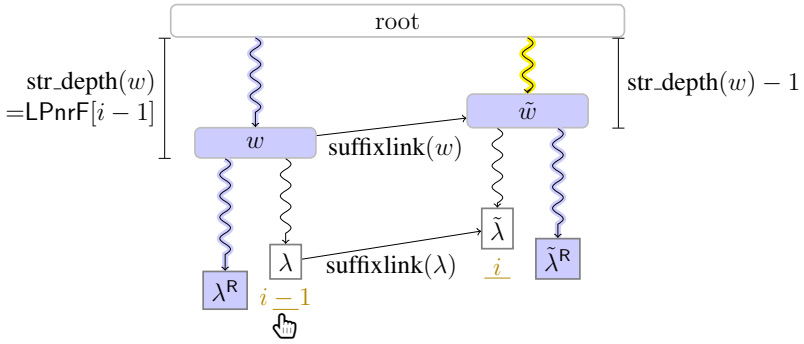


Figure 7. Setting of Section 4.1. Nodes marked in B_{V} are colored in blue ( ). Curly arcs symbolize paths that can visit multiple nodes (which are not visualized). When visiting the lowest ancestor of λ marked in B_{V} for computing $\mathsf{LPnrF}[i-1]$, Player 1 determines $\tilde{w} = \mathrm{suffixlink}(w)$ such that she can skip the nodes on the path from the root to the leaf $\tilde{\lambda}$ for computing $\mathsf{LPnrF}[i]$ (these nodes are symbolized by the curly arc highlighted in yellow ( ) on the right). There are leaves λ^{R} and $\tilde{\lambda}^{\mathsf{R}}$ with suffix numbers of at least $2n-i+2$ and $2n-i+3$, respectively, since otherwise w would not have been marked in B_{V} by Player 2.

4.2. Algorithm of Crochemore et al.

We can also run the algorithm of Crochemore et al. [14] ([Algorithm 2]) with our suffix tree representations to obtain the same space and time bounds as stated in Theorem 2. For that, let us explain this algorithm in suffix tree terminology: For each leaf λ with suffix number i, the idea for computing $\mathsf{LPnrF}[i]$ is to scan the leaves for the leaf $\lambda^\star$ with $2n - \mathrm{sufnum}(\lambda^\star)$ being the referred position, and hence the string depth of $\mathrm{lca}(\lambda, \lambda^\star)$ is $\mathsf{LPnrF}[i]$. To compute $\lambda^\star$, we approach λ from the left and from the right to find λ_{L} (resp. λ_{R}) having the deepest LCA with λ among all leaves to the left (resp. right) side of λ whose suffix numbers are greater than $2n-i$. Then either λ_{L} or λ_{R} is $\lambda^\star$. Let $\ell_{\mathrm{L}}[i] \leftarrow \mathrm{str\_depth}(\mathrm{lca}(\lambda_{\mathrm{L}}, \lambda))$ and $\ell_{\mathrm{R}}[i] \leftarrow \mathrm{str\_depth}(\mathrm{lca}(\lambda_{\mathrm{R}}, \lambda))$. Then $\mathsf{LPnrF}[i] = \max(\ell_{\mathrm{L}}[i], \ell_{\mathrm{R}}[i])$, and the referred position is either $2n - \mathrm{sufnum}(\lambda_{\mathrm{L}})$ or $2n - \mathrm{sufnum}(\lambda_{\mathrm{R}})$, depending on whose respective LCA has the deeper string depth. Note that the referred positions in this algorithm are not necessarily always the leftmost possible ones.

Correctness. Let j be the referred position of the leaf λ with suffix number i such that $R[i\,..]$ and $R[2n-j\,..]$ have the LCP F of length $\mathsf{LPnrF}[i]$. Due to Lemma 1, there is a suffix tree node w whose string label is F. Consequently, λ and the leaf with suffix number $2n-j$ are in the subtree rooted at w. Now suppose that we have computed λ_{L} and λ_{R} according to the above described algorithm. On the one hand, let us first assume that $\ell_{\mathrm{R}}[i] > \mathsf{LPnrF}[i]$ (the case $\ell_{\mathrm{L}}[i] > \mathsf{LPnrF}[i]$ is treated symmetrically). By definition of $\ell_{\mathrm{R}}[i]$, there is a descendant w' of w with the string depth $\ell_{\mathrm{R}}[i]$, and w' has both λ_{R} and λ in its subtree. However, this means that $R[i\,..]$ and $R[\mathrm{sufnum}(\lambda_{\mathrm{R}})\,..]$ have a common prefix longer than $\mathsf{LPnrF}[i]$, a contradiction to $\mathsf{LPnrF}[i]$ storing the length of the longest such LCP. On the other hand, let us assume that $\max(\ell_{\mathrm{L}}[i], \ell_{\mathrm{R}}[i]) < \mathsf{LPnrF}[i]$. Then w is a descendant of the node w' being the LCA of λ and λ_{R}. Without loss of generality, let us stipulate that the leaf $\lambda^\star$ with suffix number $2n-j$ is to the right of λ (the other case to the left of λ works with λ_{L} by symmetry). Then $\lambda^\star$ is to the left of λ_{R}, i.e., $\lambda^\star$ is between λ and λ_{R}. Since $j > 2n-i$, this contradicts the selection of λ_{R} to be the closest leaf on the right hand side of λ with a suffix number larger than $2n-i$.

Finding the Starting Points. Finally, to find the starting points of λ_{L} and λ_{R} being initially the leaves with the maximal suffix number to the left and to the right of λ, respectively, we use a data structure for answering.

maxsuf_leaf(j_1, j_2) returning the leaf with the maximum suffix number among all leaves whose leaf-ranks are in $[j_1\,..\,j_2]$.

We can modify the data structure computing max_sufnum in Section 3.3 to return the leaf-rank instead of the suffix number (the used data structure for max_sufnum first

computes the leaf-rank and then the respective suffix number). Finally, we need to take the border case into account that λ is the leftmost leaf or the rightmost leaf in the suffix tree, in which case we only need to approach λ from the right side or from the left side, respectively.

The algorithm explained up to now already computes LPnrF correctly, but visits $\mathcal{O}(n)$ leaves per LPnrF entry, or $\mathcal{O}(n^2)$ leaves in total. To improve this bound to $\mathcal{O}(n)$ leaves, we apply two tricks. To ease the explanation of these tricks, let us focus on the right-hand side of λ; the left-hand side is treated symmetrically.

Overview for Algorithmic Improvements. Given we want to compute $\ell_{\mathrm{R}}[i]$, we start with a pointer λ'_{R} to a leaf to the right of λ with suffix number larger than $2n-i$, and approach λ with λ'_{R} from the right until there is no leaf closer to λ on its right side with a suffix number larger than $2n-i$. Then λ'_{R} is λ_{R}, and we can compute $\ell_{\mathrm{R}}[i]$ being the string depth of the LCA of λ_{R} and λ. If we scan linearly the suffix tree leaves to reach λ_{R} with the pointer λ'_{R}, this gives us $\mathcal{O}(n)$ leaves to process. Now the first trick lets us reduce the number of these leaves up to $2\ell_{\mathrm{R}}[i]$ many for computing $\ell_{\mathrm{R}}[i]$. The broad idea is that with the max_sufnum operation we can find a leaf closer to λ whose LCA is at least one string depth deeper than the LCA with the previously processed leaf. In total, the first trick helps us to compute LPnrF by processing at most $\sum_{i=1}^{n} \max(\ell_{\mathrm{L}}[i], \ell_{\mathrm{R}}[i]) = \mathcal{O}(n^2)$ many leaves. In the second trick, we show that we can reuse the already computed neighboring leaves λ_{L} and λ_{R} by following their suffix links such we process at most $2(\ell_{\mathrm{R}}[i+1] - \ell_{\mathrm{R}}[i] + 1)$ many leaves (instead of $2\ell_{\mathrm{R}}[i+1]$) for computing $\ell_{\mathrm{R}}[i+1]$. Finally, by a telescoping sum, we obtain a linear number of leaves to process.

First Trick. The first trick is to jump over leaves whose respective suffixes all share the same longest common prefix with $T[i\,..]$. We start with $\lambda_{\mathrm{R}} \leftarrow \mathrm{maxsuf\_leaf}(\mathrm{leaf\_rank}(\lambda)+1, 2n)$ being the leaf on the right-hand side of λ with the largest suffix number. As long as $\mathrm{sufnum}(\lambda_{\mathrm{R}}) > 2n-i$, we search the leftmost leaf λ' between λ and λ_{R} (to be more precise: $\mathrm{leaf\_rank}(\lambda') \in [\mathrm{leaf\_rank}(\lambda)+1\,..\,\mathrm{leaf\_rank}(\lambda_{\mathrm{R}})]$) with $\mathrm{lca}(\lambda', \lambda) = \mathrm{lca}(\lambda_{\mathrm{R}}, \lambda)$. Having λ', we consider:

- If $\mathrm{leaf\_rank}(\lambda') = \mathrm{leaf\_rank}(\lambda)+1$ (meaning λ' is to the right of λ and there is no leaf between λ and λ'), we terminate.
- Otherwise, we set λ'_{R} to the leaf with the largest suffix number among the leaves with leaf-ranks in the range $[\mathrm{leaf\_rank}(\lambda)+1\,..\,\mathrm{leaf\_rank}(\lambda')-1]$. If $\mathrm{sufnum}(\lambda'_{\mathrm{R}}) > 2n-i$, we set $\lambda_{\mathrm{R}} \leftarrow \lambda'_{\mathrm{R}}$ and recurse. Otherwise we terminate.

On termination, $\ell_{\mathrm{R}}[i] = \mathrm{str\_depth}(\mathrm{lca}(\lambda_{\mathrm{R}}, \lambda))$ because there is no leaf λ'' on the right of λ closer to λ than λ_{R} with $\mathrm{str\_depth}(\mathrm{lca}(\lambda'', \lambda)) > \mathrm{str\_depth}(\mathrm{lca}(\lambda_{\mathrm{R}}, \lambda))$ and $\mathrm{sufnum}(\lambda'') > 2n-i$. Hence, $\mathrm{sufnum}(\lambda_{\mathrm{R}})$ is the referred position, and we continue with the computation of $\ell_{\mathrm{R}}[i+1]$. See Figure 8 for a visualization.

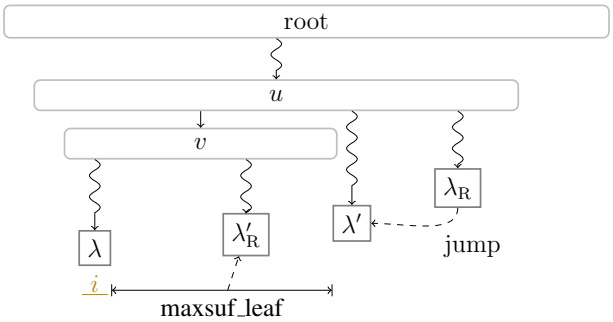


Figure 8. Computing LPnrF with [14] ([Algorithm 2]) as explained in Section 4.2. Starting at the leaf λ_{R}, we jump to the leftmost leaf λ' with $\mathrm{lca}(\lambda', \lambda) = \mathrm{lca}(\lambda_{\mathrm{R}}, \lambda)$. Then, we use the operation $\mathrm{max\_sufnum}(\mathcal{I})$ returning the leaf-rank of the leaf λ'_{R} having the largest suffix number among the query interval $\mathcal{I} = [\mathrm{leaf\_rank}(\lambda)+1\,..\,\mathrm{leaf\_rank}(\lambda')-1]$. If $\mathrm{sufnum}(\lambda'_{\mathrm{R}}) > 2n-i$, we recurse by setting $\lambda_{\mathrm{R}} \leftarrow \lambda'_{\mathrm{R}}$. The LCA of λ'_{R} and λ is at least as deep as the child v of u on the path towards λ (the figure shows the case that $v = \mathrm{lca}(\lambda'_{\mathrm{R}}, \lambda)$), and hence $\ell_{\mathrm{R}}[i]$ is at least $\mathrm{str\_depth}(v)$ if we recurse.

Broadly speaking, the idea is that the closer λ_R gets to λ, the deeper the string depth of $\mathrm{lca}(\lambda_R, \lambda)$ becomes. However, we have to stop when there is no closer leaf with a suffix number larger than $2n - i$. So we first scan until reaching a λ' having the same lowest common ancestor with λ, and then search within the interval of leaves between λ and λ' for the remaining leaf λ'_R with the largest suffix number. We search for λ' because we can jump from λ_R to λ' with a range minimum query on the LCP array returning the index of the leftmost minimum in a given range. We can answer this query with an $\mathcal{O}(n)$-bits data structure in $\mathcal{O}(\epsilon^{-1})$ or $\mathcal{O}(\epsilon^{-1} \log^{\epsilon}_{\sigma} n)$ time for the SST or the CST, respectively, and build it in $\mathcal{O}(\epsilon^{-1} n)$ time or $\mathcal{O}(\epsilon^{-1} n \log^{\epsilon}_{\sigma} n)$ time (cf. [22] ([Section 3.3]) and [41] ([Lemma 3]) for details). However, with this algorithm, we may visit as many leaves as $\sum_{i=1}^{n} 2\ell_R[i] \leq \sum_{i=1}^{n} 2\mathsf{LPnrF}[i]$ since each jump from λ_R to λ'_R via λ' brings us at least one value closer to $\ell_R[i]$. To lower this bound to $\mathcal{O}(n)$ leaf-visits, we again make use of Lemma 2 (cf. Section 4.1), but exchange $\mathsf{LPnrF}[i]$ with $\ell_R[i]$ (or respectively $\ell_L[i]$) in the statement of the lemma.

Second Trick. Assume that we have computed $\ell_R[i-1] = \mathrm{lca}(\lambda_R, \lambda)$ with $j := \mathrm{sufnum}(\lambda_R) > 2n - i$. We subsequently set $\lambda \leftarrow \mathrm{suffixlink}(\lambda)$, but also $\lambda_R \leftarrow \mathrm{suffixlink}(\lambda_R)$. Now λ has suffix number i. If $\ell_R[i-1] \geq 1$, then the string depth of the $\mathrm{lca}(\lambda_R, \lambda)$ is $\ell_R[i-1] - 1$, and $R[\mathrm{sufnum}(\lambda_R)\,..]$ is lexicographically larger than $R[\mathrm{sufnum}(\lambda)\,..]$; hence λ_R is to the right of λ with $\mathrm{sufnum}(\lambda_R) = j + 1$ (generally speaking, given two leaves λ_1 and λ_2 whose LCA is not the root, then $\mathrm{leaf\_rank}(\lambda_1) < \mathrm{leaf\_rank}(\lambda_2)$ if and only if $\mathrm{leaf\_rank}(\mathrm{suffixlink}(\lambda_1)) < \mathrm{leaf\_rank}(\mathrm{suffixlink}(\lambda_2))$.). Otherwise ($\ell_R[i-1] = 0$), we reset $\lambda_R \leftarrow \mathrm{maxsuf\_leaf}(\mathrm{leaf\_rank}(\lambda), 2n)$. By doing so, we assure that λ_R is always a leaf to the right of λ with $\mathrm{sufnum}(\lambda_R) > 2n - i$ (if such a leaf exists), and that we have already skipped $\max(0, \ell_R[i-1] - 1)$ string depths for the search of λ_R with $\mathrm{str\_depth}(\mathrm{lca}(\lambda_R, \lambda)) = \ell_R[i]$. Since $\ell_R[i] \leq \mathsf{LPnrF}[i]$, the telescoping sum $\ell_R[1] + \sum_{i=2}^{n} (\ell_R[i] - \ell_R[i-1] + 1) = \mathcal{O}(n)$ shows that we visit $\mathcal{O}(n)$ leaves in total.

In total, we obtain an algorithm that visits $\mathcal{O}(n)$ leaves, and spends $\mathcal{O}(\epsilon^{-1})$ or $\mathcal{O}(\epsilon^{-1} \log^{\epsilon}_{\sigma} n)$ time per leaf when using the SST or the CST, respectively. We need $\mathcal{O}(n)$ bits of working space on top of ST since we only need the values $\ell_L[i-1\,..\,i]$, $\ell_R[i-1\,..\,i]$, λ_L, and λ_R to compute $\mathsf{LPnrF}[i]$. We note that Crochemore et al. [14] do not need the suffix tree topology, since they only access the suffix array, its inverse, and the LCP array, which we translated to ST leaves and the string depths of their LCAs.

5. Open Problems

There are some problems left open, which we would like to address in what follows:

5.1. Overlapping Reversed LZ Factorization

Crochemore et al. [14] ([Section 5]) gave a variation of LPnrF that supports overlaps, and called the resulting array the longest previous reverse factor table LPrF, where $\mathsf{LPrF}[i]$ is the maximum length ℓ such that $T[i\,..\,i+\ell-1] = T[j\,..\,j+\ell-1]^{\mathsf{R}}$ for a $j < i$. The respective factorization, called the overlapping reversed LZ factorization, was proposed by Sugimoto et al. [5] ([Definition 4]): A factorization $F_1 \cdots F_z = T$ is called the overlapping reversed LZ factorization of T if each factor F_x is either the leftmost occurrence of a character or the longest prefix of $F_x \cdots F_z$ that has at least one reversed occurrence in $F_1 \cdots F_x$ starting before F_x, for $x \in [1\,..\,z]$. We can compute the overlapping reversed LZ factorization with LPrF analogously to computing the (non-overlapping) reversed LZ factorization with LPnrF. As an example, the overlapping reversed LZ factorization of $T = \mathtt{abbababbabab}$ is $\mathtt{a} \cdot \mathtt{bbabba} \cdot \mathtt{bab}$. Table 4 gives an example for LPrF.

Since $\mathsf{LPrF}[i] \geq \mathsf{LPnrF}[i]$ by definition, the overlapping factorization seems more likely to have fewer factors. Unfortunately, this factorization cannot be expressed in a compact coding like Section 3.1 that stores enough information to restore the original text. To see this, take a palindrome P, and compute the overlapping reversed LZ factorization of $\mathtt{a}P\mathtt{a}$. The factorization creates the two factors a and $P\mathtt{a}$. The second factor is $P\mathtt{a}$ since $(P\mathtt{a})^{\mathsf{R}} = \mathtt{a}P$. However, a coding of the second factor needs to store additional information about P to

support restoring the characters of this factor. It seems that we need to store the entire left arm of P, including the middle character for odd palindromes.

Besides searching for an efficient coding for the overlapping reversed LZ factorization, we would like to improve the working space bounds needed for its computation. All algorithms we are aware of [5,14] embrace Manacher's algorithm [42,43] to find the maximal palindromes of each text position. To run in linear time, Manacher stores the arm lengths of these palindromes in a plain array of $n \lg n$ bits. Unfortunately, we are unaware of any time/space trade-offs regarding this array.

5.2. Computing LPF in Linear Time with Compressed Space

Having a $2n$-bit representation for four different kinds of longest previous factor tables (we can exchange LPnrF with LPrF in the proof of Lemma 2), we wonder whether it is possible to compute any of these variants in linear time with $o(n \lg n)$ bits of space. If we want to compute LPF or LPnrF within a working space of $\mathcal{O}(n \lg \sigma)$ bits, it seems hard to achieve linear running time. That is because we need access to the string depth of the suffix tree node w for each entry $\mathsf{LPF}[i]$ (resp. $\mathsf{LPnrF}[i]$), where w is the lowest node having the leaf λ with suffix number i and a leaf with a suffix number less than i (resp. greater than $2n - i$ for LPnrF) in its subtree, cf. [34] ([Lemma 6]) for LPF and the actions of Player 1 in Section 4.1 for LPnrF. While we need to compute $\mathrm{str\_depth}(w)$ for determining the starting position of the subsequent factor (i.e., suffix number of the next phrase leaf, cf. Line 16) for the reversed LZ factorization, the algorithms for computing LPF (cf. [34] ([Lemma 6]) or [44] ([Section 3.4.4])) and LPnrF work independently of the computed factor lengths and therefore can store a batch of str_depth-queries. Our question would be whether there is a $\delta = \mathcal{O}((n \lg \sigma)/\lg n)$ such that we can accesses δ suffix array positions with a $\mathcal{O}(n \lg \sigma)$-bits suffix array representation in $\mathcal{O}(\delta)$ time. (We can afford storing δ integers of $\lg n$ bits in $\mathcal{O}(n \lg \sigma)$ bits.) Grossi and Vitter [45] ([Theorem 3]) have a partial answer for sequential accesses to suffix array regions with large LCP values. Belazzougui and Cunial [24] ([Theorem 1]) experienced the same problem for computing matching statistics, but could evade the evaluation of str_depth with backward search steps on the reversed Burrows–Wheeler transform. Unfortunately, we do not see how to apply their solution here since the referred positions of LPF and LPnrF have to belong to specific text ranges (which is not the case for matching statistics).

5.3. Applications in Compressors

Although it seems appealing to use the reversed LZ factorization for compression, we have to note that the bounds for the number of factors z are not promising:

Lemma 3. *The size of the reversed LZ factorization can be as small as* $\lg n + 1$ *and as large as* n.

Proof. The lower bound is obtained for $T = \mathtt{a} \cdots \mathtt{a}$ with $|T| = 2^{z-1}$ since $|F_1| = |F_2| = 1, |F_x| = 2|F_{x-1}|$ for $x \in [2\,..\,z]$ with $F_1 \cdots F_x = (F_1 \cdots F_x)^{\mathsf{R}}$ being a (not necessarily proper) prefix of $T[|F_1 \cdots F_x|\,..]$. For the upper bound, we consider the ternary string $T = \mathtt{abc} \cdot \mathtt{abc} \cdots \mathtt{abc}$ whose factorization consists only of factors of length one since $T^{\mathsf{R}} = \mathtt{cba} \cdot \mathtt{cba} \cdots \mathtt{cba}$ has no substring of T of length 2 (namely, `ab`, `bc`, or `ca`) as a substring (cf. [46] ([Theorem 5])). □

Even for binary alphabets, there are strings for which $z = \Theta(n)$:

Lemma 4 ([46] (Theorem 9))**.** *There exists an infinite text* T *whose characters are drawn from the binary alphabet such that, for every substring* S *of* T *with* $|S| \geq 5$, S^R *is not a substring of* T.

Funding: This work is funded by the JSPS KAKENHI Grant Numbers `JP18F18120` and `JP21K17701`.

Data Availability Statement: Not applicable.

Acknowledgments: We thank a CPM'2021 reviewer for pointing out that it suffices to store W in $z_W \lg n$ bits of space in Section 3.3, and that the currently best construction algorithm of the compressed suffix tree indeed needs $\mathcal{O}(\epsilon^{-1}n)$ time instead of just $\mathcal{O}(n)$ time.

Conflicts of Interest: The author declares no conflict of interest.

Appendix A. Flip Book

In this appendix, we provide a detailed execution of the algorithm sketched in Figures 4–6 by showing the state per turn and per player in Figures A1–A21. In our running example, each player has 10 turns.

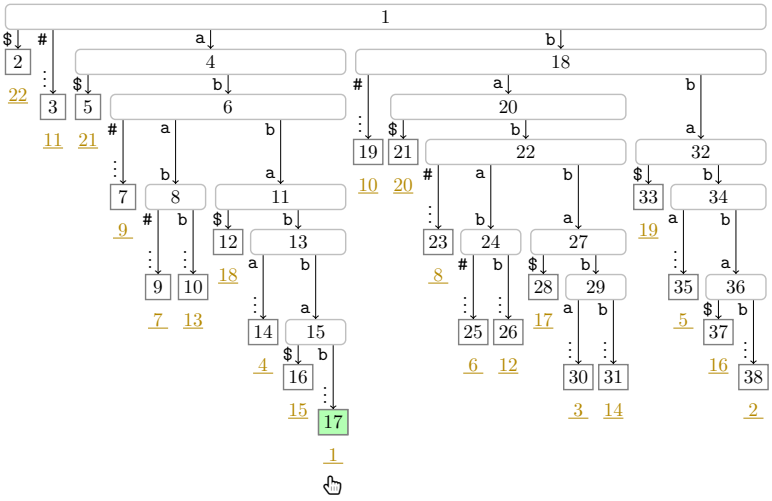


Figure A1. *Flip Book:* Initial State.

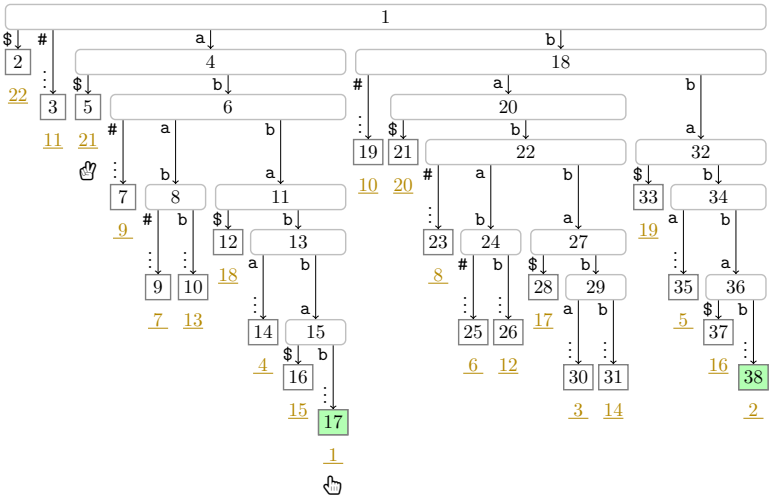


Figure A2. *Flip Book*: End of Turn 1 of Player 1.

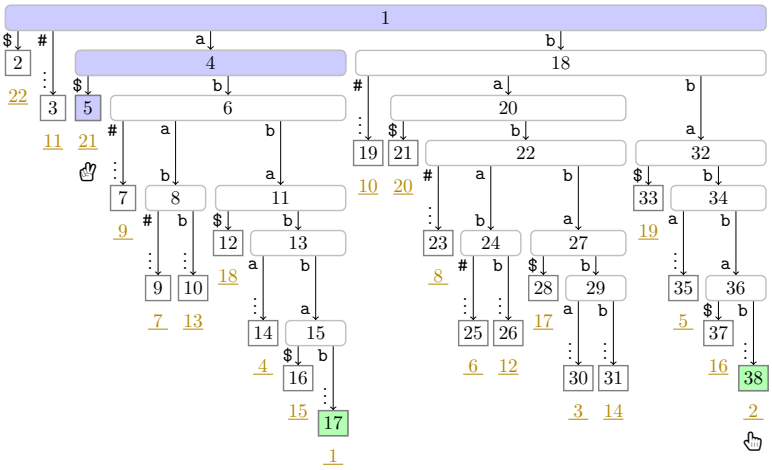


Figure A3. *Flip Book*: End of Turn 1 of Player 2.

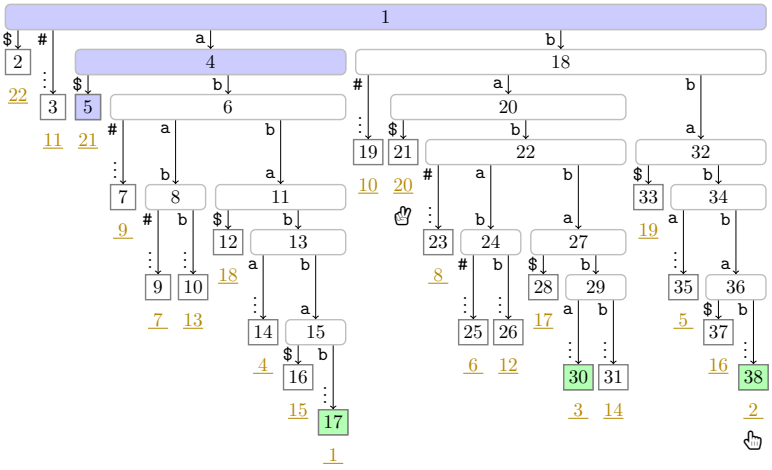


Figure A4. *Flip Book*: End of Turn 2 of Player 1.

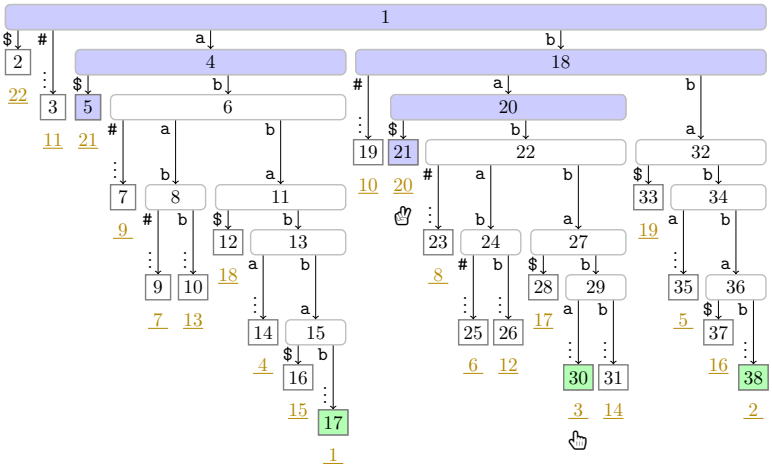


Figure A5. *Flip Book*: End of Turn 2 of Player 2.

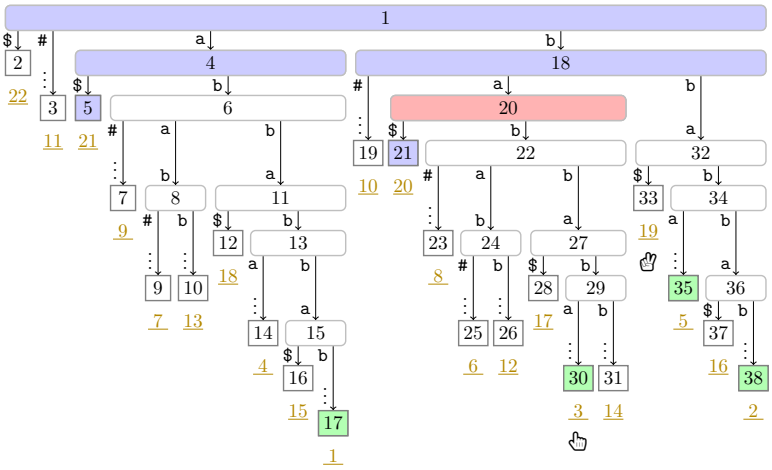


Figure A6. *Flip Book*: End of Turn 3 of Player 1.

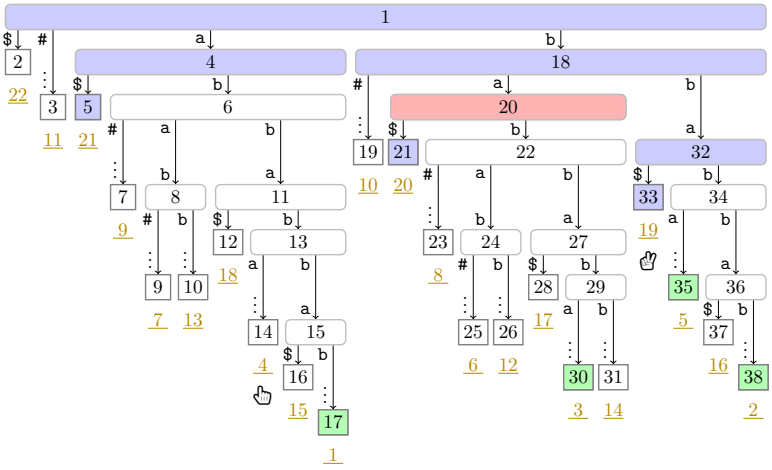


Figure A7. *Flip Book*: End of Turn 3 of Player 2.

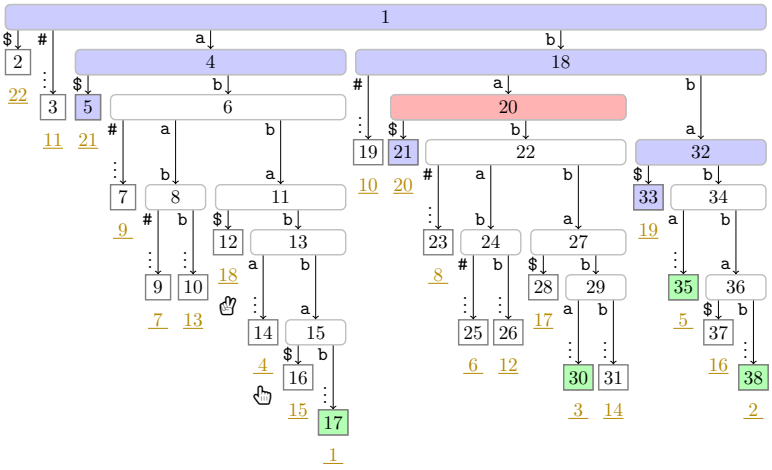


Figure A8. *Flip Book*: End of Turn 4 of Player 1.

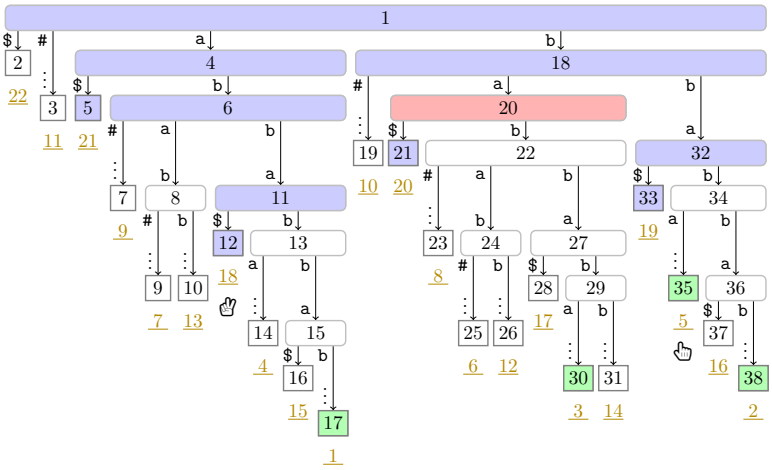


Figure A9. *Flip Book*: End of Turn 4 of Player 2.

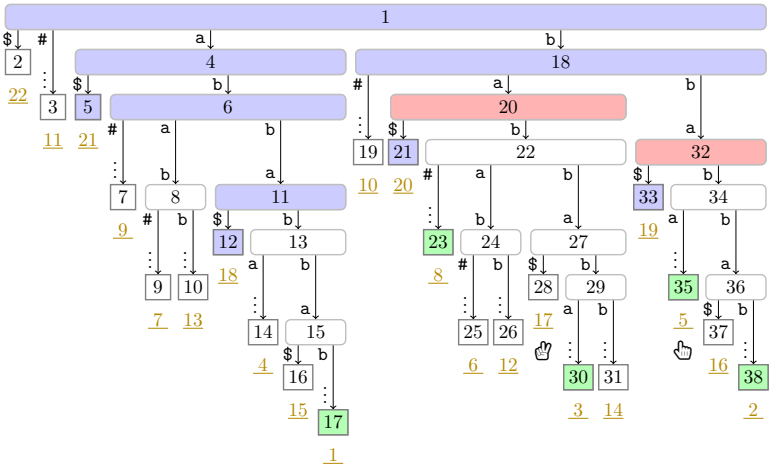


Figure A10. *Flip Book*: End of Turn 5 of Player 1.

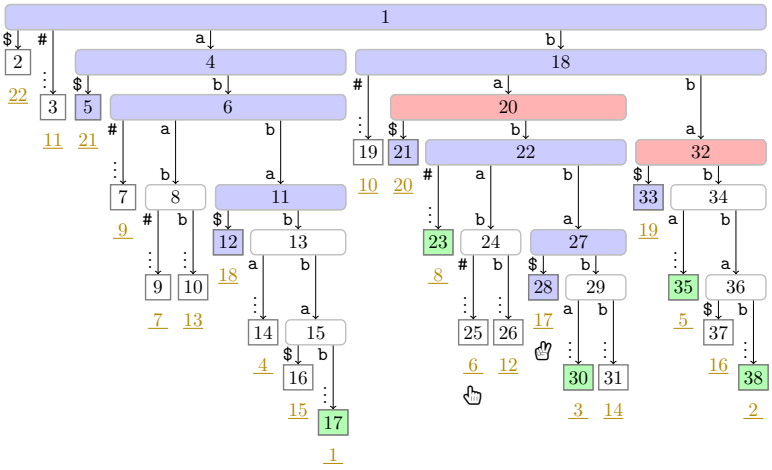


Figure A11. *Flip Book*: End of Turn 5 of Player 2.

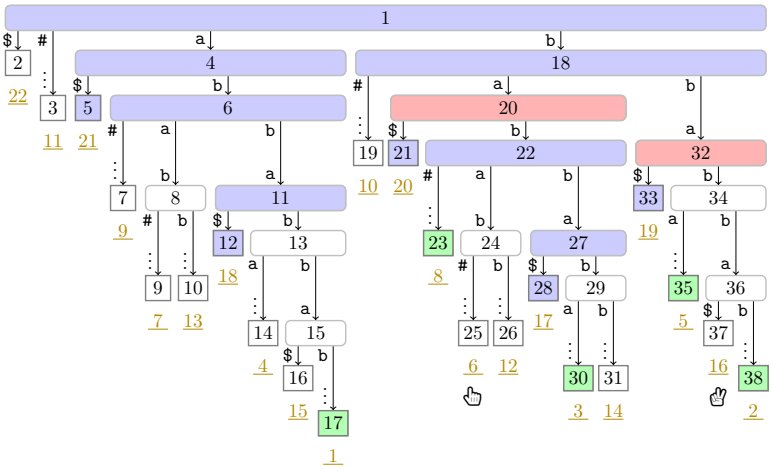


Figure A12. *Flip Book*: End of Turn 6 of Player 1.

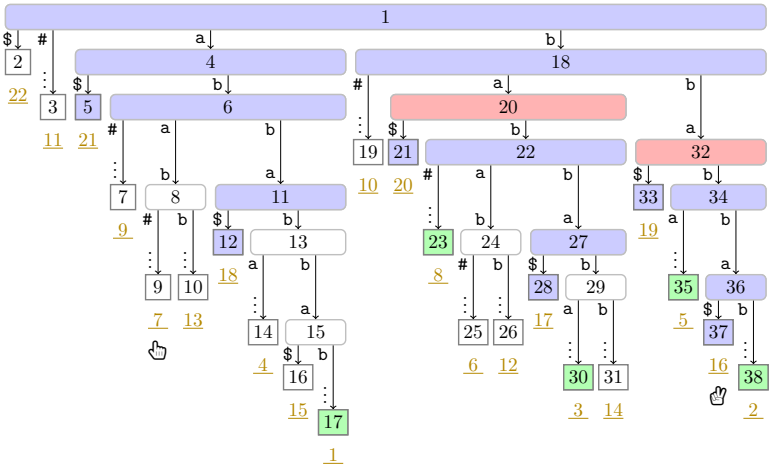


Figure A13. *Flip Book*: End of Turn 6 of Player 2.

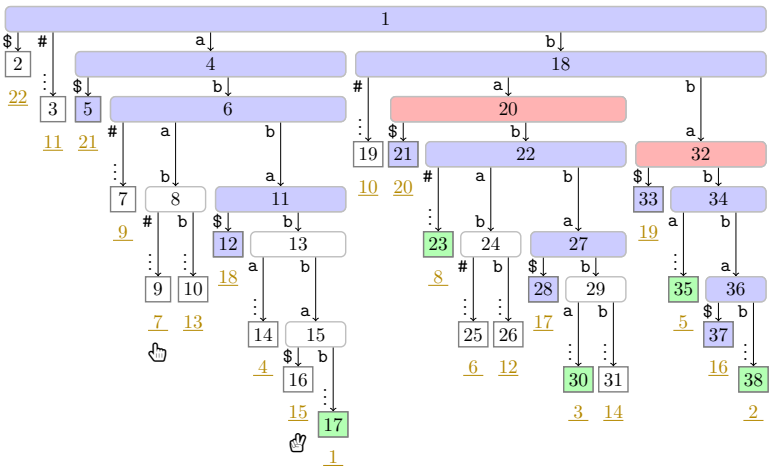


Figure A14. *Flip Book*: End of Turn 7 of Player 1.

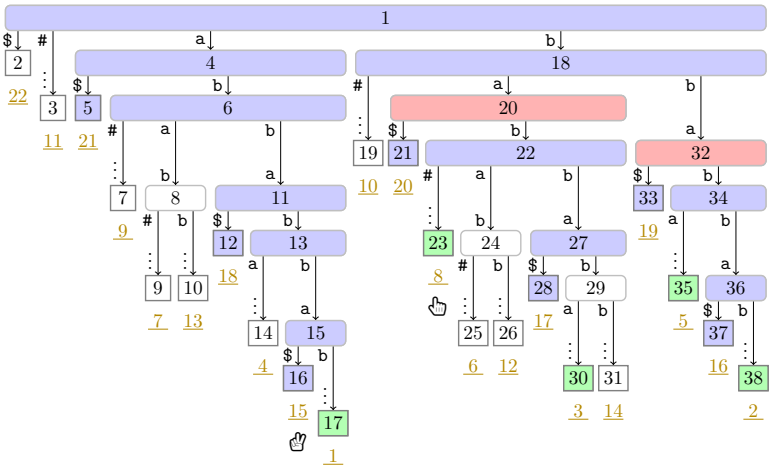


Figure A15. *Flip Book*: End of Turn 7 of Player 2.

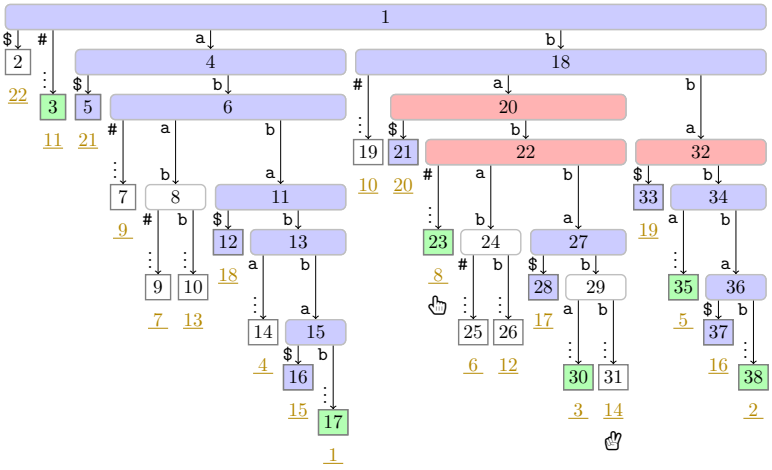


Figure A16. *Flip Book*: End of Turn 8 of Player 1.

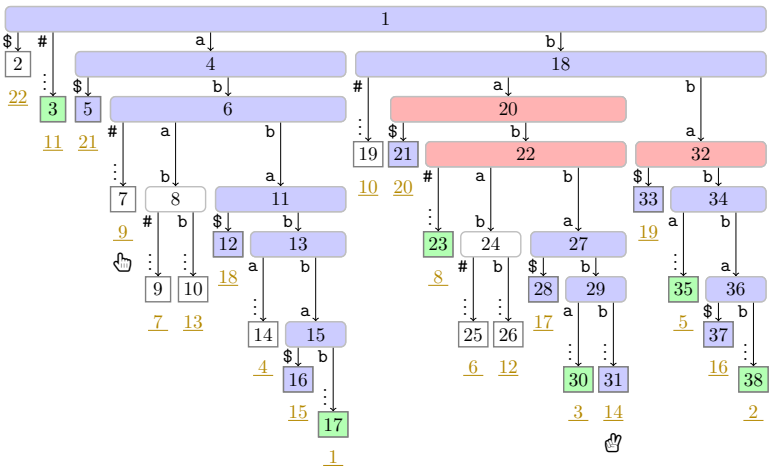


Figure A17. *Flip Book*: End of Turn 8 of Player 2.

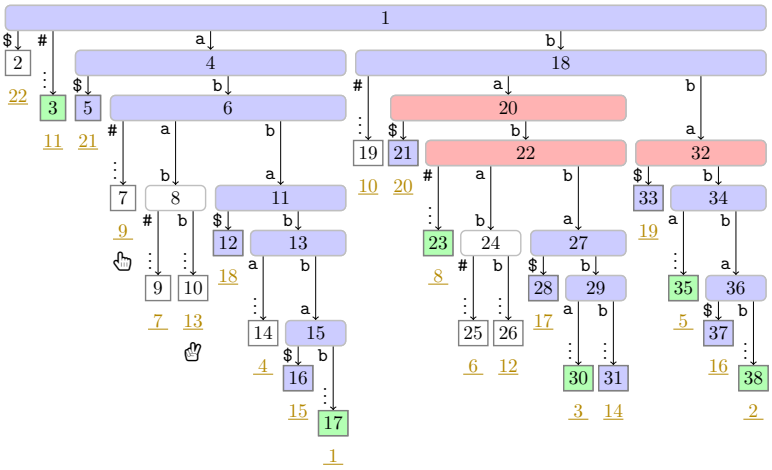


Figure A18. *Flip Book*: End of Turn 9 of Player 1.

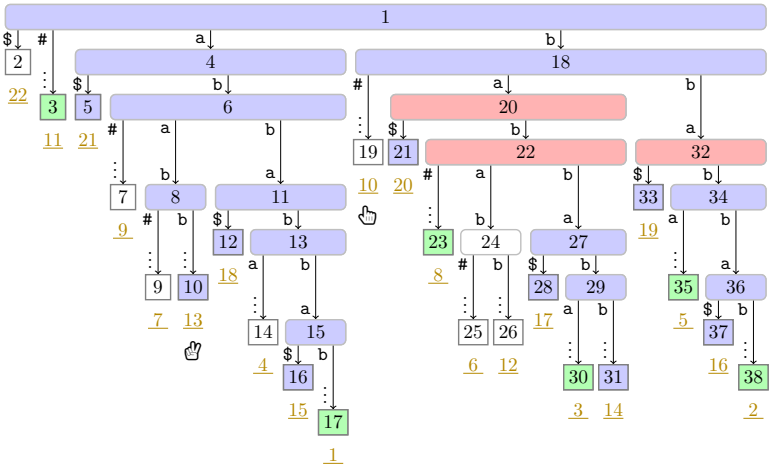


Figure A19. *Flip Book*: End of Turn 9 of Player 2.

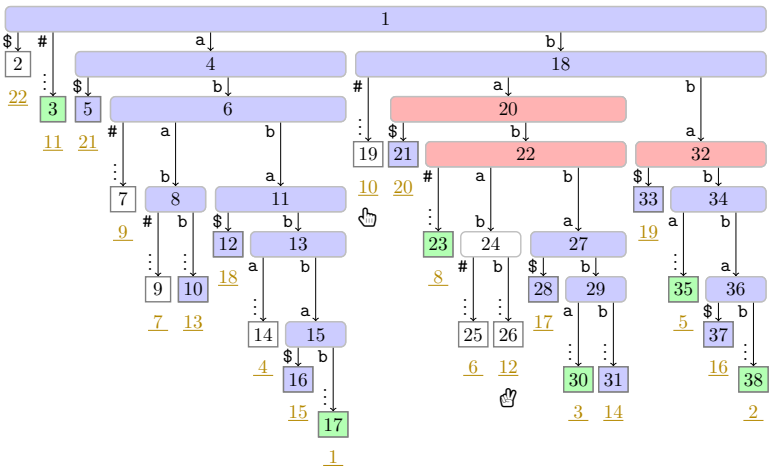


Figure A20. *Flip Book*: End of Turn 10 of Player 1.

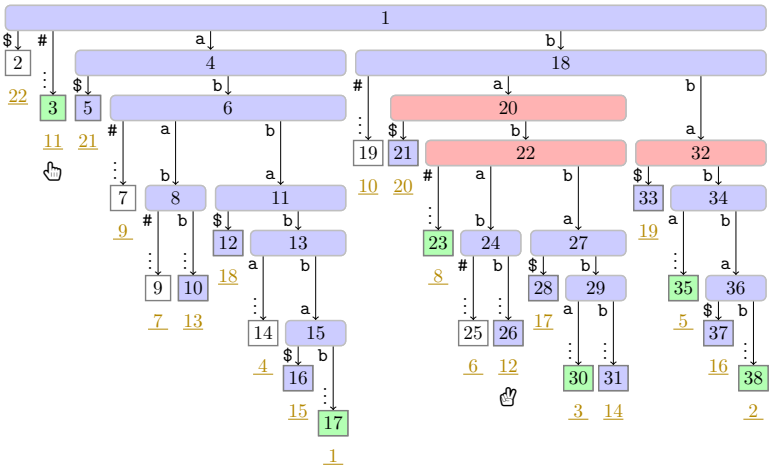


Figure A21. *Flip Book*: End of Turn 10 of Player 2.

References

1. Kolpakov, R.; Kucherov, G. Searching for gapped palindromes. *Theor. Comput. Sci.* **2009**, *410*, 5365–5373.
2. Storer, J.A.; Szymanski, T.G. Data compression via textural substitution. *J. ACM* **1982**, *29*, 928–951.
3. Crochemore, M.; Langiu, A.; Mignosi, F. Note on the greedy parsing optimality for dictionary-based text compression. *Theor. Comput. Sci.* **2014**, *525*, 55–59.
4. Weiner, P. Linear Pattern Matching Algorithms. In Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973) SWAT, Iowa City, IA, USA, 15–17 October 1973; pp. 1–11.
5. Sugimoto, S.; Tomohiro, I.; Inenaga, S.; Bannai, H.; Takeda, M. Computing Reversed Lempel–Ziv Factorization Online. Avaliable online: http://stringology.org/papers/PSC2013.pdf#page=115 (accessed on 15 April 2021).
6. Chairungsee, S.; Crochemore, M. Efficient Computing of Longest Previous Reverse Factors. In Proceedings of the Computer Science and Information Technologies, Yerevan, Armenia, 28 September–2 October 2009; pp. 27–30.
7. Badkobeh, G.; Chairungsee, S.; Crochemore, M. Hunting Redundancies in Strings. In *International Conference on Developments in Language Theory*; LNCS; Springer, Berlin/Heidelberg, Germany, 2011; Volume 6795, pp. 1–14.
8. Chairungsee, S. Searching for Gapped Palindrome. Avaliable online: https://www.sciencedirect.com/science/article/pii/S0304397509006409 (accessed on 15 April 2021).
9. Charoenrak, S.; Chairungsee, S. Palindrome Detection Using On-Line Position. In Proceedings of the 2017 International Conference on Information Technology, Singapore, 27–29 December 2017; pp. 62–65.
10. Charoenrak, S.; Chairungsee, S. Algorithm for Palindrome Detection by Suffix Heap. In Proceedings of the 2019 7th International Conference on Information Technology: IoT and Smart City, Shanghai China, 20–23 December 2019; pp. 85–88.
11. Blumer, A.; Blumer, J.; Ehrenfeucht, A.; Haussler, D.; McConnell, R.M. Building the Minimal DFA for the Set of all Subwords of a Word On-line in Linear Time. In *International Colloquium on Automata, Languages, and Programming*; LNCS; Springer: Berlin/Heidelberg, Germany, 1984; Volume 172, pp. 109–118.
12. Ehrenfeucht, A.; McConnell, R.M.; Osheim, N.; Woo, S. Position heaps: A simple and dynamic text indexing data structure. *J. Discret. Algorithms* **2011**, *9*, 100–121.
13. Gagie, T.; Hon, W.; Ku, T. New Algorithms for Position Heaps. In *Annual Symposium on Combinatorial Pattern Matching*; LNCS; Springer, Berlin/Heidelberg, Germany, 2013; Volume 7922, pp. 95–106.
14. Crochemore, M.; Iliopoulos, C.S.; Kubica, M.; Rytter, W.; Walen, T. Efficient algorithms for three variants of the LPF table. *J. Discret. Algorithms* **2012**, *11*, 51–61.
15. Manber, U.; Myers, E.W. Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.* **1993**, *22*, 935–948.
16. Dumitran, M.; Gawrychowski, P.; Manea, F. Longest Gapped Repeats and Palindromes. *Discret. Math. Theor. Comput. Sci.* **2017**, *19*, 205–217.
17. Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*; Cambridge University Press: Cambridge, UK, 1997.
18. Nakashima, Y.; Tomohiro, I.; Inenaga, S.; Bannai, H.; Takeda, M. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inf. Process. Lett.* **2015**, *115*, 655–659.
19. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343.
20. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536.
21. Fischer, J.; I, T.; Köppl, D.; Sadakane, K. Lempel–Ziv Factorization Powered by Space Efficient Suffix Trees. *Algorithmica* **2018**, *80*, 2048–2081.
22. Köppl, D. Non-Overlapping LZ77 Factorization and LZ78 Substring Compression Queries with Suffix Trees. *Algorithms* **2021**, *14*, 1–21.

23. Sadakane, K. Compressed Suffix Trees with Full Functionality. *Theory Comput. Syst.* **2007**, *41*, 589–607.
24. Belazzougui, D.; Cunial, F. Indexed Matching Statistics and Shortest Unique Substrings. In *International Symposium on String Processing and Information Retrieval*; LNCS; Springer: Cham, Switzerland, 2014; Volume 8799, pp. 179–190.
25. Franek, F.; Holub, J.; Smyth, W.F.; Xiao, X. Computing Quasi Suffix Arrays. *J. Autom. Lang. Comb.* **2003**, *8*, 593–606.
26. Crochemore, M.; Ilie, L. Computing Longest Previous Factor in linear time and applications. *Inf. Process. Lett.* **2008**, *106*, 75–80.
27. Belazzougui, D.; Cunial, F.; Kärkkäinen, J.; Mäkinen, V. Linear-time String Indexing and Analysis in Small Space. *ACM Trans. Algorithms* **2020**, *16*, 17:1–17:54.
28. Goto, K.; Bannai, H. Space Efficient Linear Time Lempel–Ziv Factorization for Small Alphabets. In Proceedings of the 2014 Data Compression Conference, Snowbird, UT, USA, 26–28 March 2014; pp. 163–172.
29. Kärkkäinen, J.; Kempa, D.; Puglisi, S.J. Lightweight Lempel–Ziv Parsing. In *International Symposium on Experimental Algorithms*; LNCS; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7933, pp. 139–150.
30. Kosolobov, D. Faster Lightweight Lempel–Ziv Parsing. In *International Symposium on Mathematical Foundations of Computer Science*; LNCS; Springer: Berlin/Heidelberg, Germany, 2015; Volume 9235, pp. 432–444.
31. Belazzougui, D.; Puglisi, S.J. Range Predecessor and Lempel–Ziv Parsing. In Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, Arlington, VA, USA, 10–12 January 2016; pp. 2053–2071.
32. Okanohara, D.; Sadakane, K. An Online Algorithm for Finding the Longest Previous Factors. In *European Symposium on Algorithms*; LNCS; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5193, pp. 696–707.
33. Prezza, N.; Rosone, G. Faster Online Computation of the Succinct Longest Previous Factor Array. In *Conference on Computability in Europe*; LNCS; Springer: Cham, Switzerland, 2020; Volume 12098, pp. 339–352.
34. Bannai, H.; Inenaga, S.; Köppl, D. Computing All Distinct Squares in Linear Time for Integer Alphabets. Proc. CPM, 2017; Volume 78, LIPIcs, pp. 22:1–22:18. Available online: https://link.springer.com/chapter/10.1007/978-3-662-48057-1_16 (accessed on 16 April 2021).
35. Jacobson, G. Space-efficient Static Trees and Graphs. In Proceedings of the 30th Annual Symposium on Foundations of Computer Science Research, Triangle Park, NC, USA, 30 October–1 November 1989; pp. 549–554.
36. Clark, D.R. Compact Pat Trees. Ph.D. Thesis, University of Waterloo, Waterloo, ON, Canada, 1996.
37. Baumann, T.; Hagerup, T. Rank-Select Indices Without Tears. In Proceedings of the Algorithms and Data Structures—16th International Symposium, WADS 2019, Edmonton, AB, Canada, 5–7 August 2019; LNCS, Volume 11646, pp. 85–98.
38. Munro, J.I.; Navarro, G.; Nekrich, Y. Space-Efficient Construction of Compressed Indexes in Deterministic Linear Time. In Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, Barcelona, Spain, 16–19 January 2017; pp. 408–424.
39. Burrows, M.; Wheeler, D.J. *A Block Sorting Lossless Data Compression Algorithm*; Technical Report 124; Digital Equipment Corporation: Palo Alto, CA, USA, 1994.
40. Lempel, A.; Ziv, J. On the Complexity of Finite Sequences. *IEEE Trans. Inf. Theory* **1976**, *22*, 75–81.
41. Fischer, J.; Mäkinen, V.; Navarro, G. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.* **2009**, *410*, 5354–5364.
42. Manacher, G.K. A New Linear-Time "On-Line" Algorithm for Finding the Smallest Initial Palindrome of a String. *J. ACM* **1975**, *22*, 346–351.
43. Apostolico, A.; Breslauer, D.; Galil, Z. Parallel Detection of all Palindromes in a String. *Theor. Comput. Sci.* **1995**, *141*, 163–173.
44. Köppl, D. Exploring Regular Structures in Strings. Ph.D. Thesis, TU Dortmund, Dortmund, Germany, 2018.
45. Grossi, R.; Vitter, J.S. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM J. Comput.* **2005**, *35*, 378–407.
46. Fleischer, L.; Shallit, J.O. Words Avoiding Reversed Factors, Revisited. *arXiv* **2019**, arXiv:1911.11704.