Faster Online Computation of the Succinct Longest Previous Factor Array

Nicola Prezza[1,2] and Giovanna Rosone[1](✉)

[1] University of Pisa, Pisa, Italy
nicola.prezza@di.unipi.it, giovanna.rosone@unipi.it
[2] LUISS Guido Carli, Rome, Italy
nprezza@luiss.it

Abstract. We consider the problem of computing online the Longest Previous Factor array $LPF[1, n]$ of a text T of length n. For each $1 \leq i \leq n$, $LPF[i]$ stores the length of the longest factor of T with at least two occurrences, one ending at i and the other at a previous position $j < i$. We present an improvement over the previous solution by Okanohara and Sadakane (ESA 2008): our solution uses less space (compressed instead of succinct) and runs in $O(n \log^2 n)$ time, thus being faster by a logarithmic factor. As a by-product, we also obtain the first online algorithm computing the Longest Common Suffix (LCS) array (that is, the LCP array of the reversed text) in $O(n \log^2 n)$ time and compressed space. We also observe that the LPF array can be represented succinctly in $2n$ bits. Our online algorithm computes directly the succinct LPF and LCS arrays.

Keywords: Longest Previous Factor · Online · Compressed data structures

1 Introduction

This paper focuses on the problem of computing the *Longest Previous Factor* (LPF) array which stores, for each position i in a string S, the length of the longest factor (substring) of S that ends both at i and to the left of i in S. While the notion of Longest Previous Factor has been introduced in [10], an array with the same definition already appeared in McCreight's suffix tree construction algorithm [18] (the *head* array) and recently in [12] (the π array).

The concept of LPF array is close to that of *Longest Common Prefix* (LCP) and *Permuted Longest Common Prefix* (PLCP) arrays, structures that are usually associated with the suffix array (SA) data structure to speed up particular queries on strings (for example, pattern matching).

Supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBSI146R5L.

The problem of searching for the longest previous factor is fundamental in many applications [10], including data compression and pattern analysis. For example, the LPF array can be used to derive the Ziv-Lempel factorization [25], a very powerful text compression tool based on longest previous factors [9,10].

Methods to compute the LPF array [6,9,10,21] can be broadly classified into two categories: batch (offline) and online algorithms. For instance, in [10] the authors give two offline linear-time algorithms for computing the Longest Previous Factor (LPF) array. The idea of the first algorithm is that, given SA, for any position i, they only need to consider the suffixes starting to the left of i in S which are closest to the suffix starting at position i in SA. In the second algorithm (see also [7] for a similar in spirit but independent work), the authors use a similar idea, but they take advantage of the fact that this variant processes the suffix array in one pass and requires less memory space.

In [11], the authors show how an algorithm similar to the one of [9,10] can compute the LPF array in linear running time by reading SA left-to-right (that is, online on SA) using a stack that reduces the memory space to $O(\sqrt{n})$ for a string of length n in addition to the SA, LCP and LPF arrays. This algorithm requires less than $2\sqrt{2n}+O(1)$ integer cells in addition to its input and output.

Unlike batch algorithms, an online algorithm for the problem should report the longest match just after reading each character. The online version of the problem can be defined as follows: given a history $T[1, i-1]$, and the next character $c = T[i]$, the goal is to find the longest substring that matches the current suffix: $T[j, \ldots, j+l-1] = T[i-l+1, \ldots, i]$, and report the position and the length of the matched substring. This process must be performed for all $i = 1, \ldots, n$.

Okanohara and Sadakane in [21] propose an online algorithm that relies on the incremental construction of Enhanced Suffix Arrays (ESA) [1] in a similar way to Weiner's suffix tree construction algorithm [24]. They employ compressed full-text indexing methods [20] to represent ESA dynamically in succinct space. Their algorithm requires $n \log \sigma + o(n \log \sigma) + O(n) + \sigma \log n$ bits of working space[1], $O(n \log^3 n)$ total time, and $O(\log^3 n)$ delay per character, where n is the input size and σ is the alphabet size.

Another online construction of the LCP array, in this case of a *string collection*, appears in [8]. In this work, the authors show how to update the LCP of a string collection when all strings are extended by one character.

Our work is a direct improvement over Okanohara and Sadakane's [21] algorithm. The bottleneck in their strategy is the use of a dynamic Range Minimum Query (RMQ) data structure over the (dynamic) LCP array. In this paper, we observe that the RMQ is not needed at all since we can update our structures by computing, with direct character comparisons, just irreducible LCP values. Since it is well-known that the sum of such values amounts to $O(n \log n)$, this yields a logarithmic improvement over the algorithm described in [21]. On the other hand, our strategy offers a worse delay of $O(n \log n)$ per input character.

[1] In their analysis they do not report the term $\sigma \log n$, which however should be included since they use a prefix sum structure over the alphabet's symbols.

2 Definitions

A *string* $S = s_1 s_2 \dots s_n$ is a sequence of $n = |S|$ symbols from alphabet $\Sigma = [1, \sigma]$, with $\sigma \leq n$. A *text* T is a string beginning with special symbol $\# = 1$, not appearing elsewhere in T. A *factor* (or *substring*) of a string S is written as $S[i, j] = s_i \cdots s_j$ with $1 \leq i \leq j \leq n$. When defining an array A, we use the same notation $A[1, k]$ to indicate that A has k entries enumerated from 1 to k.

In this work we use text indices based on the principle of co-lexicographically sorting the prefixes of a text, rather than lexicographically sorting its suffixes. This is the same approach adopted in [21] and is required by the online left-to-right nature of the problem we consider. Given a string $S \in \Sigma^n$, we denote by $<$ the standard co-lexicographic ordering among the prefixes of S.

The *Prefix* array $PA[1, n]$ of a string $S[1, n]$ [23] is an array containing the permutation of the integers $1, 2, \dots, n$ that arranges the ending positions of the prefixes of S into co-lexicographical order, i.e., for all $1 \leq i < j \leq n$, $S[1, PA[i]] < S[1, PA[j]]$. The *Inverse Prefix* array $IPA[1, n]$ is the inverse permutation of PA, i.e., $IPA[i] = j$ if and only if $PA[j] = i$.

The C-array of a string S is an array $C[1, \sigma]$ such that $C[i]$ contains the number of characters lexicographically smaller than i in S, plus one (S will be clear from the context). It is well-known that this array can be kept within $\sigma \log n + o(\sigma \log n)$ bits of space on a dynamic string by using succinct searchable partial sums [3, 14], which support all operations in $O(\log n)$ time.

The *co-lexicographic Burrows-Wheeler Transform* $BWT[1, n]$ of a text T is a reversible transformation that permutes its symbols as $BWT[i] = T[PA[i] + 1]$ if $PA[i] < n$, and $BWT[i] = \#$ otherwise [5].

The *Longest Common Suffix* array $LCS[1, n]$ of a string S [17] is an array storing in $LCS[i]$ the length of the longest common suffix shared by the $(i-1)$-th and i-th co-lexicographically smallest text prefixes if $i > 1$, and $LCS[1] = 0$ otherwise. Function $LCS(i, j)$ generalizes this array: given a string $S[1, n]$, $LCS(i, j)$ denotes the longest common suffix between $S[1, PA[i]]$ and $S[1, PA[j]]$. The *Permuted Longest Common Suffix* array $PLCS[1, n]$ of a string S stores LCS values in string order, rather than co-lexicographic order: $PLCS[i] = LCS[IPA[i]]$.

Function $S.rank_c(i)$, where $c \in \Sigma$, returns the number of characters equal to c in $S[1, i-1]$. Similarly, function $S.select_c(i)$ returns the position of S containing the i-th occurrence of c.

Given $BWT[1, n]$ of a text $T[1, n]$, the *LF mapping* is a function $BWT.LF(i)$ that, given the BWT position containing character $T[j]$ (with $j = PA[i] + 1$), returns the BWT position i' of character $T[j + 1]$. This function can be implemented with a *rank* operation on BWT and one access to the C array. Similarly, the *FL mapping* is the reverse of LF: this is the function $BWT.FL(i)$ that, given the BWT position containing character $T[j]$ (with $j = PA[i] + 1$), returns the BWT position i' of character $T[j - 1]$ (assume for simplicity that $j > 1$; otherwise, $BWT[i] = \#$). This function can be implemented with a *select* operation on BWT and a search on the C array.

3 Succinct PLCS and LPF Arrays

We start by formally introducing the definition of *LPF array.*

Definition 1 (Longest Previous Factor array). *The* Longest Previous Factor array $LPF[1, n]$ *of a string* $S[1, n]$ *is the array containing, at each location* $LPF[i]$*, the largest integer* k *such that there exists* $j < i$ *for which the longest common suffix between* $S[1, j]$ *and* $S[1, i]$ *has length* k*.*

Kasai et al. [16] observe that the Permuted Longest Common Prefix array is almost increasing: $PLCP[i + 1] \geq PLCP[i] - 1$. Of course, this still holds true for the Permuted Longest Common Suffix array that we consider in our work. Specifically, the symmetric relation $PLCS[i + 1] \leq PLCS[i] + 1$ holds. In the next lemma we observe that the same property is true also for the LPF array.

Lemma 1. *For any* $i < n$*, it holds* $LPF[i + 1] \leq LPF[i] + 1$*.*

Proof. Let $LPF[i] = k$. Then, k is the largest integer such that the substring $T[i-k+1, i]$ starts at another position $j < i-k+1$. Assume, for contradiction, that $LPF[i + 1] = k' > k + 1$. Then, this means that $s = T[(i + 1) - k' + 1, i + 1]$ occurs at another position $j' < (i + 1) - k' + 1$. But then, also the prefix $T[(i + 1) - k' + 1, i]$ of s occurs at j'. This is a contradiction, since the length of $T[(i + 1) - k' + 1, i]$ is $k' - 1 > k = LPF[i]$. □

Note that $PLCS[1] = LPF[1] = 0$, thus the two arrays can be encoded and updated succinctly with the same technique, described in Lemma 2.

Lemma 2. *Let* $A[1, n]$ *be a non-negative integer array satisfying properties (a)* $A[1] = 0$ *and (b)* $A[i+1] \leq A[i]+1$ *for* $i < n$*. Then, there is a data structure of* $2n + o(n)$ *bits supporting the following operations in* $O(\log n/\log\log n)$ *time:*

(1) access any $A[i]$*,*
(2) append a new element $A[n + 1]$ *at the end of* A*, and*
(3) update: $A[i] \leftarrow A[i] + \Delta$*,*

provided that operations (2) and (3) do not violate properties (a) and (b). Running time of operation (2) is amortized ($O(n \log n/\log\log n)$ *in the worst case).*

Proof. We encode A as a bitvector A' of length *at most* $2n$ bits, defined as follows. We start with $A' = 01$ and, for $i = 2, \ldots, n$ we append the bit sequence $0^{A[i-1]+1-A[i]}1$ to the end of A'. The intuition is that every bit set increases the previous value $A[i-1]$ by 1, and every bit equal to 0 decreases it by 1. Then, the value $A[i]$ can be retrieved simply as $2i - A'.select_1(i)$. Clearly, A' contains n bits equal to 1 since for each $1 \leq i \leq n$ we insert a bit equal to 1 in A'. Since the total number of 1s is n and A is non-negative, A' contains at most n bits equal to 0 as well. It follows that A' contains at most $2n$ bits in total. In order to support updates, we encode the bitvector with the dynamic string data structure of Munro and Nekrich [19], which takes at most $2n+o(n)$ bits of space and supports queries and updates in $O(\log n/\log\log n)$ worst-case time. We already showed how operation (1) reduces to *select* on A'. Let $\Delta = A[n]+1-A[n+1]$. To support operation (2), we need to append the bit sequence $0^{\Delta}1$ at the end of A'. In the worst case, this operation takes $O(\Delta \log n/\log\log n) = O(n \log n)$ time. However, the sum of all Δ is equal to the total number of 0s in the bitvector; this implies that, over a sequence of n insertions, this operation takes $O(\log n/\log\log n)$ amortized time. Finally, operation $A[i] \leftarrow A[i] + \Delta$ can be implemented by moving the bit at position $A'.select_1(i)$ by Δ positions to the left, which requires just one *delete* and one *insert* operation on A' ($O(\log n/\log\log n)$ time). Note that this is always possible, provided that the update operation does not violate properties (a) and (b) on the underlying array A. □

4 Online Algorithm

We first give a sketch of our idea, and then proceed with the details. Similarly to Okanohara and Sadakane [21], we build online the BWT and the compressed LCS array of the text, and use the latter component to output online array LPF. This is possible by means of a simple observation: after reading character $T[i]$, entry $LPF[i]$ is equal to the maximum between $LCS[IPA[i]]$ and $LCS[IPA[i]+1]$. As in [21], array LCS is represented in compressed form by storing PLCS (in $2n+o(n)$ bits, Lemma 2) and a sampling of the prefix array PA which, together with BWT, allows computing any $PA[i]$ in $O(\log^2 n)$ time. Then, we can retrieve any LCS value in $O(\log^2 n)$ time as $LCS[i] = PLCS[PA[i]]$.

The bottleneck of Okanohara and Sadakane's strategy is the update of LCS. This operation requires being able to compute the longest common suffix between two arbitrary text's prefixes $T[1, PA[i]]$ and $T[1, PA[j]]$ (see [21] for all the details). By a well-known relation, this value is equal to $\min(LCS[i, j])$ (assume $i < j$ w.l.o.g.). In Okanohara and Sadakane's work, this is achieved using a dynamic Range Minimum Query (RMQ) data structure on top of LCS. The RMQ is a balanced tree whose leaves cover $\Theta(\log n)$ LCS values each and therefore requires accessing $O(\log n)$ LCS values in order to compute $\min(LCS[i, j])$, for a total running time of $O(\log^3 n)$. We note that this running time cannot be improved by simply replacing the dynamic RMQ structure of [21] with more recent structures. Brodal et al. in [4] describe a dynamic RMQ structure supporting queries and updates in $O(\log n/\log\log n)$ time, but the required space is $O(n)$ words. Heliou et al. in [13] reduce this space to $O(n)$ bits, but they require, as in [21], $O(\log n)$ accesses to the underlying array.

Our improvement over Okanohara and Sadakane's algorithm stems from the observation that the RMQ structure is not needed at all, as we actually need to compute by direct symbol comparisons just *irreducible* LCS values:

Definition 2. *$LCS[i]$ is said to be* irreducible *if and only if either $i = 0$ or $BWT[i] \neq BWT[i-1]$ hold.*

Irreducible LCS values enjoy the following property:

Lemma 3 ([15], Thm. 1). *The sum of all irreducible LCS values is at most $2n \log n$.*

As a result, we will spend overall just $O(n \log^2 n)$ time to compute all irreducible LCS values. This is less than the time $O(n \log^3 n)$ needed in [21] to compute $O(n)$ minima on the LCS.

4.1 Data Structures

Dynamic BWT. Let $T[1, i]$ be the text prefix seen so far. As in [21], we keep a dynamic BWT data structure to store the BWT of $T[1, i]$. In our case, this structure is represented using Munro and Nekrich's dynamic string [19] and takes $nH_k + o(n \log \sigma) + \sigma \log n + o(\sigma \log n)$ bits of space, for any $k \in o(\log_\sigma n)$. The latter two space components are needed for the C array encoded with succinct searchable partial sums [3,14]. The structure supports *rank*, *select*, and *access* in $O(\log n/ \log \log n)$ time, while *appending* a character at the end of T and computing the LF and FL mappings are supported in $O(\log n)$ time (the bottleneck are succinct searchable partial sums, which cannot support all operations simultaneously in $O(\log n/ \log \log n)$ time by current implementations [3,14]).

Dynamic Sparse Prefix Array. As in Okanohara and Sadakane's solution, we also keep a dynamic Prefix Array sampling. Let $D = \lceil \log n \rceil$ be the sample rate. We store in a dynamic sequence PA' all integers $x_j = j/D$ such that $j \bmod D = 0$, for $j \leq i$ (i.e. we sample one out of D text positions and re-enumerate them starting from 1). Letting $j_1 < \cdots < j_k$ be the co-lexicographic order of the sampled text positions seen so far, the corresponding integers are stored in PA' in the order $x_{j_1}, \ldots, x_{j_k}$. In the next paragraph we describe the structure used to represent PA' (as well as its inverse), which will support queries and updates in $O(\log n)$ time. We use again Munro and Nekrich's dynamic string [19] to keep a dynamic bitvector B_{PA} of length i (i being the length of the current text prefix) that marks with a bit set sampled entries of PA. Since we sample one out of $D = \lceil \log n \rceil$ text's positions and the bitvector is entropy-compressed, its size is $o(n)$ bits. At this point, any $PA[j]$ can be retrieved by returning $D \cdot PA'[B_{PA}.rank_1(j) + 1]$ if $B_{PA}[j] = 1$ or performing at most D LF mapping steps otherwise, for a total running time of $O(\log^2 n)$. Note that, by the way we re-enumerate sampled text positions, the sequence PA' is a permutation.

Dynamic Sparse Inverse Prefix Array. The first difference with Okanohara and Sadakane's solution is that we keep the inverse of PA' as well, that is, a (dynamic) sparse inverse prefix array: we denote this array by IPA' and define it as $IPA'[PA'[j]] = j$, for all $1 \leq j \leq |PA'|$. First, note that we insert integers in PA' in increasing order: $x = 1, 2, 3, \ldots$. Inserting a new integer x at some position t in PA' has the following effect in IPA': first, all elements $IPA'[k] \geq t$ are increased by 1. Then, value t is appended at the end of IPA'.

Example 1. Let PA' and $IPA' = (PA')^{-1}$ be the following permutations: $PA' = \langle 3, 1, 2, 4 \rangle$ and $IPA' = \langle 2, 3, 1, 4 \rangle$. Suppose we insert integer 5 at position 2 in PA'. The updated permutations are: $PA' = \langle 3, 5, 1, 2, 4 \rangle$ and $IPA' = \langle 3, 4, 1, 5, 2 \rangle$.

Policriti and Prezza in [22] show how to represent a permutation of size k and its inverse upon insertions and access queries in $O(\log k)$ time per operation and $O(k)$ words of space. The idea is to store PA' in a self-balancing tree, sorting its elements by the inverse permutation IPA'. Then, IPA' is represented simply as a vector of pointers to the nodes of the tree. By enhancing the tree's nodes with the corresponding sub-tree sizes, the tree can be navigated top-down (to access PA') and bottom-up (to access IPA') in logarithmic time. Since we sample one out of $D = \lceil \log n \rceil$ positions, the structure takes $O(n)$ bits of space.

To compute any $IPA[j]$, we proceed similarly as for PA. We compute the sampled position $j - \delta$ (with $\delta \geq 0$) preceding j in the text, we find the corresponding position t on PA as $t = B_{PA}.select(IPA'[(j-\delta)/D])$, and finally perform $\delta \leq D$ steps of LF mapping to obtain $IPA[j]$. Note that, without loss of generality, we can consider position 1 to be always sampled since $IPA[1] = 1$ is constant. To sum up, computing any $IPA[j]$ requires $O(\log^2 n)$ time, while updating PA' and IPA' takes $O(\log n)$ time.

Dynamic PLCS Vector. We also keep the dynamic PLCS vector, stored using the structure of Lemma 2. When extending the current text prefix $T[1,i]$ by character $T[i+1]$, LCS changes in two locations: first, a new value is inserted at position $IPA[i+1]$. Then, the value $LCS[IPA[i+1]+1]$ (if this cell exists) can possibly increase, due to the insertion of a new text prefix before it in co-lexicographic order. As a consequence, PLCS changes in two places as well: (i) a new value $PLCS[i+1] = LCS[IPA[i+1]]$ is appended at the end, and (ii) value $PLCS[PA[IPA[i+1]+1]]$ (possibly) increases. Both operations are supported in $O(\log n)$ (amortized) time by Lemma 2.

The way these new PLCS values are calculated is where our algorithm differs from Okanohara and Sadakane's [21], and is described in the next section.

4.2 Updating the LCS Array

In this section we show how to update the LCS array (stored in compressed format as described in the previous sections).

Algorithm. We show how to compute the new LCS value to be inserted at position $IPA[i+1]$ (after extending $T[1,i]$ with $T[i+1]$). The other update, to $LCS[IPA[i+1]+1]$, is completely symmetric so we just sketch it. Finally, we analyze the amortized complexity of our algorithm.

Let $a = T[i+1]$ be the new text symbol, and let k be the position such that $BWT[k] = \#$. We recall that the BWT extension algorithm works by replacing $BWT[k]$ with the new character $a = T[i+1]$, and by inserting # at position $C[a] + BWT.rank_a(k) = IPA[i+1]$. We also recall that the location of the first occurrence of a symbol a preceding/following $BWT[k] = \#$ can be easily found with one *rank* and one *select* operations on BWT.

Now, consider the BWT of $T[1,i]$. We distinguish three main cases (in [21], all these cases were treated with a single Range Minimum Query):

(a) $BWT[1,k]$ does not contain occurrences of character a. Then, $T[1,i+1]$ is the co-lexicographically smallest prefix ending with a, therefore the new LCS value to be inserted at position $IPA[i+1]$ is 0.

(b) $BWT[k-1] = a$. Then, prefix $T[1, PA[k-1]+1]$ (ending with $BWT[k-1] = a$) immediately precedes $T[1,i+1]$ in co-lexicographic order. It follows that the LCS between these two prefixes is equal to 1 plus the LCS between $T[1, PA[k-1]]$ and $T[1,i]$, i.e. $1 + LCS[IPA[i]]$. This is the new LCS value to be inserted at position $IPA[i+1]$.

(c) The previous letter equal to a in $BWT[1,k]$ occurs at position $j < k-1$. The goal here is to compute the LCS $\ell = LCS(j,k)$ between prefixes $T[1, PA[k]]$ and $T[1, PA[j]]$. Integer $\ell+1$ is the new LCS value to be inserted at position $IPA[i+1]$. We distinguish two further sub-cases.

(c.1) String $BWT[k+1, i]$ does not contain occurrences of character a. Then, we compare the two prefixes $T[1, PA[k]]$ and $T[1, PA[j]]$ right-to-left simply by repeatedly applying function FL from BWT positions j and k. The number of performed symbol comparisons is $LCS(j,k)$.

(c.2) There is an occurrence of a after position k. Let $q > k$ be the smallest position such that $BWT[q] = a$. Table 1 reports an example of this case. Then, $T[1, PA[j]+1]$ and $T[1, PA[q]+1]$ are adjacent in co-lexicographic order, thus we can compute $\ell' = LCS(j,q)$ as follows. Letting $j' = BWT.LF(j)$ and $q' = BWT.LF(q) = j'+1$ (that is, the co-lexicographic ranks of $T[1, PA[j]+1]$ and $T[1, PA[q]+1]$, respectively), we have $\ell' = LCS[q'] - 1$. Since $j < k < q$, we have $LCS(j,k) = \ell \geq \ell'$. In order to compute $\ell = LCS(j,k)$, the idea is to skip the first ℓ' comparisons (which we know will return a positive result), and only compare the remaining $\ell - \ell'$ characters in the two prefixes, that is, compare the two prefixes $T[1, PA[k] - \ell']$ and $T[1, PA[j] - \ell']$. This can be achieved by finding the co-lexicographic ranks of these two prefixes, that is $IPA[PA[k] - \ell']$ and $IPA[PA[j] - \ell']$ respectively, and applying the FL function from these positions to extract the $\ell - \ell'$ remaining matching characters in the prefixes. The number of performed symbol comparisons is $\ell - \ell' = LCS(j,k) - LCS(j,q)$.

As noted above, the other update to be performed at position $LCS[IPA[i+1]+1]$ is completely symmetric so we just sketch it here. The cases where $BWT[k,i]$ does not contain occurrences of a or where $BWT[k+1] = a$ correspond to cases (a) and (b). If the first occurrence of a following position k appears at position $q > k+1$, on the other hand, we distinguish two further cases. The case where $BWT[1,k]$ does not contain occurrences of a is handled as case (c.1) above (by direct character comparisons between two text prefixes). Otherwise, we find the first occurrence of $a = BWT[j]$ before position k and proceed as in case (c.2), by finding the LCS ℓ' between the suffixes $T[1, PA[q]]$ and $T[1, PA[j]]$, and comparing prefixes $T[1, PA[k] - \ell']$ and $T[1, PA[q] - \ell']$.

Amortized Analysis. In the following, by *symbol comparisons* we indicate the comparisons performed in case (c) to compute LCS values (by means of iterating the FL mapping). For simplicity, we count only comparisons resulting in a match between the two compared characters: every time we encounter a mismatch, the comparison is interrupted; this can happen at most $2n$ times (as we update at most two LCS values per iteration), therefore it adds at most $O(n \log n)$ to our final running time (as every FL step takes $O(\log n)$ time).

We now show that the number of symbol comparisons performed in case (c) is always upper-bounded by the sum of irreducible LCS values.

Definition 3. *A BWT position $k > 1$ is said to be a* relevant run break *if and only if:*

(i) $BWT[k-1] \neq BWT[k]$,
(ii) there exists $j < k-1$ *such that* $BWT[j] = BWT[k]$, *and*
(iii) if $BWT[k-1] = \#$, *then* $k > 2$ *and* $BWT[k-2] \neq BWT[k]$.

Condition (i) requires k to be on the border of an equal-letter BWT run. Condition (ii) requires that there is a character equal to $BWT[k]$ before position $k-1$, and condition (iii) states that # does not contribute in forming relevant run breaks (e.g. in string $a\#a$, the second occurrence of a is not a relevant run break; however, in $ac\#a$ the second occurrence of a is). Intuitively, condition (iii) is required since extending the text by one character might result in two runs of the same letter separated by just # to be merged (e.g. $aaa\#a$ becomes $aaaaa$ after replacing # with a). Without condition (iii), after such a merge we could have characters inside a run that are charged with symbol comparisons.

In Lemma 4 we prove that our algorithm maintains the following invariant:

Invariant 1. *Consider the structures BWT and LCS for $T[1,i]$ at step i. Moreover, let k be a relevant run break, and let $j < k-1$ be the largest position such that $BWT[k] = BWT[j]$. Then:*

1. *Position k is charged with $c_k = LCS(j,k)$ symbol comparisons, and*
2. *Only relevant run breaks are charged with symbol comparisons.*

Lemma 4. *Invariant 1 is true after every step $i = 1, \ldots, n$ of our algorithm.*

Proof. After step $i = 1$, we have processed just $T[1]$ and the property is trivially true as there are no relevant run breaks. Assume by inductive hypothesis that the property holds at step i, i.e. after building all structures (BWT, LCS) for $T[1,i]$. We show that the application of cases (a-c) maintains the invariant true.

Case (a) does not perform symbol comparisons. Moreover, it does not destroy any relevant run break. The only critical case is $BWT[k+1] = a$, since replacing $BWT[k] = \#$ with a destroys the run break at position $k+1$. However, note that $k+1$ cannot be a *relevant* run break, since $BWT[1,k]$ does not contain occurrences of a. It follows that case (a) maintains the invariant.

Also case (b) does not perform symbol comparisons and does not destroy any relevant run break. The only critical case is $BWT[k+1] = a$, since replacing $BWT[k] = \#$ with a destroys the run break at position $k+1$. However, note that $k+1$ cannot be a *relevant* run break, since $BWT[k-1] = a$ and $BWT[k+1] =$

a are separated by $BWT[k] = \#$, which by definition does not contribute in forming relevant run breaks. It follows that case (b) maintains the invariant.

(c.1) Consider the BWT of $T[1, i]$, and let k be the terminator position: $BWT[k] = \#$. Note that, by Definition 3, k is not a relevant run break since no other position contains the terminator, and thus by Invariant 1 it is not charged yet with any symbol comparison. Case (c.1) compares the k-th and j-th co-lexicographically smallest text prefixes, where $j < k-1$ is the previous occurrence of a in the BWT. Clearly, the number of comparisons performed is exactly $c_k = LCS(j, k)$: we charge this quantity to BWT position k. Then, we update the BWT by (i) replacing $BWT[k] = \#$ with a, which makes k a valid relevant run break since $BWT[k-1] \neq a$, $BWT[j] = BWT[k]$, and $j < k-1$ and (ii) inserting $\#$ in some BWT position, which (possibly) shifts position k to $k' \in \{k, k+1\}$ (depending whether $\#$ is inserted before or after k) but does not alter the value of $c_k = LCS(j, k')$, so k' is a relevant run break and is charged correctly as of Invariant 1. Finally, note that (1) the new BWT position containing $\#$ is not charged with any symbol comparison (since we just inserted it), (2) that, if two runs get merged after replacing $\#$ with $T[i+1]$ then, thanks to Condition (iii) of Definition 3 and Invariant 1 at step i, no position inside a equal-letter run is charged with symbol comparisons, and (3) if the new $\#$ is inserted inside a equal-letter run a^t, thus breaking it as $a^{t_1}\#a^{t_2}$ with $t = t_1 + t_2$ and $t_1 > 0$, then the position following $\#$ is not charged with any symbol comparison. (1–3) imply that we still charge only relevant run breaks with symbol comparisons: Invariant 1 is therefore true at step $i+1$.

(c.2) Consider the BWT of $T[1, i]$, and let k, j, q, with $j < k-1 < k < q$, be the terminator position ($BWT[k] = \#$) and the immediately preceding and following positions containing $a = BWT[j] = BWT[q]$. Note that q is a relevant run-break, charged with $c_q = LCS(j, q)$ symbol comparisons by Invariant 1. Assume that $LCS(j, k) \geq LCS(k, q)$: the other case is symmetric and we discuss it below. Then, $LCS(k, q) = LCS(j, q) = c_q$. First, we "lift" the $c_q = LCS(j, q)$ symbol comparisons from position q and re-assign them to position k. By definition, case (c.2) of our algorithm performs $LCS(j, k) - LCS(j, q)$ symbol comparisons; we charge also these symbol comparisons to position k. After replacing $BWT[k]$ with letter a, position k becomes a relevant run break, and is charged with $c_q + (LCS(j, k) - LCS(j, q)) = LCS(j, k) = c_k$ symbol comparisons. Position q, on the other hand, is now charged with 0 symbol comparisons; note that this is required if $q = k+1$ (as in the example of Table 1), since in that case q is no longer a relevant run break (as we replaced $\#$ with a). Finally, we insert $\#$ in some BWT position which, as observed above, does not break Invariant 1.

The other case is $LCS(j, k) < LCS(k, q)$. Then $LCS(j, k) = LCS(j, q)$, and therefore case (c.2) does not perform additional symbol comparisons to compute $LCS[IPA[i+1]]$. On the other hand, the symmetric of case (c.2) (i.e. the case where we update $LCS[IPA[i+1]+1]$) performs $LCS(k, q) - LCS(j, q)$ symbol comparisons if $q > k+1$ (none otherwise); these are all charged to position q and, added to the $LCS(j, q)$ comparisons already charged to q, sum up to $LCS(k, q)$ comparisons. This is correct, since in that case q remains a relevant run break. If, on the other hand, $q = k+1$, then no additional comparisons need to be made to update $LCS[IPA[i+1]+1]$, and we simply lift the $LCS(j,q)$ comparisons from q (which is no longer a relevant run break) and charge them to k (which becomes a relevant run break). This is correct, since k is now charged with $LCS(j,q) = LCS(j,k)$ symbol comparisons. $\square$

Table 1. The example illustrates case (c.2). Column L is the BWT. The other columns contain the sorted text prefixes. Left: structures for $T[1,i] = \#abaaabbaababa$. We are about to extend the text with letter a. Positions k, j, q contain # (to be replaced with a) and the immediately preceding and succeeding BWT positions containing a. To find $LCS(j,q)$, apply LF to j, q, obtaining positions j' and q'. Then, $LCS(j,q) = LCS[q'] - 1 = 2$, emphasized in italic. At this point, $LCS(j,k)$ is computed by comparing the j-th and k-th smallest prefixes outside the italic zone (found using IPA and PA). In the example, we find 1 additional match (underlined). It follows that the new LCS to be inserted between positions j' and q' is $1 + LCS(j,k) = 1 + (LCS(j,q)+1) = 4$. Right: structures updated after appending a to the text. In bold on column LCS: the new LCS value inserted ($LCS[IPA[i+1]] = 4$) and the one updated by the symmetric of case (c.2) ($LCS[q'] = 3$; in this example, the value doesn't change). In bold on column F: last letters of the j'-th and q'-th smallest text's prefixes, interleaved with $T[1, i+1]$.

												F	L	LCS	
b	a	a	a	b	b	a	a	b	a	b	a	#	a	0	
a	a	a	b	b	a	a	b	a	b	a	#	a	b	0	
b	a	a	b	a	b	a	#	a	b	a	a	a	b	1	
b	b	a	a	b	a	b	a	#	a	b	a	**a**	a	2	j'
a	b	a	#	a	b	a	a	a	b	b	a	**a**	b	**3**	q'
a	b	b	a	a	b	a	b	a	#	$\underline{\text{a}}$	*b*	*a*	**a**	1	j
a	#	a	b	a	a	a	b	b	a	a	*b*	*a*	b	3	
a	b	a	a	a	b	b	a	a	b	$\underline{\text{a}}$	*b*	*a*	**#**	3	k
b	a	b	a	#	a	b	a	a	a	b	*b*	*a*	**a**	2	q
a	a	b	b	a	a	b	a	b	a	#	a	b	a	0	
a	a	b	a	b	a	#	a	b	a	a	a	b	b	2	
b	a	#	a	b	a	a	a	b	b	a	a	b	a	3	
#	a	b	a	a	a	b	b	a	a	b	a	b	a	2	
a	b	a	b	a	#	a	b	a	a	a	b	b	a	1	

													F	L	LCS	
b	a	a	a	b	b	a	a	b	a	b	a	a	#	a	0	
a	a	a	b	b	a	a	b	a	b	a	a	#	a	b	0	
b	a	a	b	a	b	a	a	#	a	b	a	a	a	b	1	
b	b	a	a	b	a	b	a	a	#	a	b	a	**a**	a	2	j'
a	b	a	a	a	b	b	a	a	b	a	b	a	a	#	**4**	
a	b	a	a	#	a	b	a	a	a	b	b	a	**a**	b	**3**	q'
a	b	b	a	a	b	a	b	a	a	#	a	b	a	a	1	j
a	a	#	a	b	a	a	a	b	b	a	a	b	a	b	3	
#	a	b	a	a	a	b	b	a	a	b	a	b	a	a	3	k
b	a	b	a	a	#	a	b	a	a	a	b	b	a	a	2	q
a	a	b	b	a	a	b	a	b	a	a	#	a	b	a	0	
a	a	b	a	b	a	a	#	a	b	a	a	a	b	b	2	
b	a	a	#	a	b	a	a	a	b	b	a	a	b	a	3	
a	#	a	b	a	a	a	b	b	a	a	b	a	b	a	2	
a	b	a	b	a	a	#	a	b	a	a	a	b	b	a	1	

Lemma 5. *At any step $i = 1, \ldots, n$, let $k_1, \ldots, k_r$ be the relevant run breaks and $c_{k_1}, \ldots, c_{k_r}$ be the symbol comparisons charged to them, respectively. Then, $\sum_{t=1}^{r} c_{k_t} \leq 2i \log i$.*

Proof. By definition of c_{k_t}, we have $c_{k_t} = LCS(j, k_t) \leq LCS(k_t - 1, k_t) = LCS[k_t]$, where $j < k_t - 1$ is the largest position to the left of $k_t - 1$ containing symbol $BWT[k_t]$. Moreover, note that $\{k_1, \ldots, k_r\}$ is a subset of the BWT run breaks $\{k \ : \ BWT[k-1] \neq BWT[k]\}$, therefore each $LCS[k_t]$ is irreducible. Let $\mathcal{S}$ be the sum of irreducible LCS values. By applying Lemma 3, we obtain:

$$\sum_{j=1}^{r} c_{k_t} \leq \sum_{j=1}^{r} LCS[k_t] \leq \mathcal{S} \leq 2i \log i$$

We obtain our main result:

Theorem 2 (Online succinct LPF and LCS arrays). *The succinct LPF and LCS arrays of a text $T \in [1, \sigma]^n$ can be computed online in $O(n \log^2 n)$ time and $O(n \log n)$ delay per character using $nH_k + o(n \log \sigma) + O(n) + \sigma \log n + o(\sigma \log n)$ bits of working space (including the output), for any $k \in o(\log_\sigma n)$.*

Proof. After LCS and IPA have been updated at step i, we can compute $LPF[i]$ simply as $LPF[i] = \max\{LCS[IPA[i]], LCS[IPA[i] + 1]\}$ in $O(\log^2 n)$ time. This value can be appended at the end of the succinct representation of LPF (Lemma 2) in $O(\log n)$ amortized time (which in the worst case becomes $O(n \log n)$). Updating BWT, PA′, and IPA′ takes $O(\log n)$ time per character. The most expensive part is updating the structures representing LCS: at each step we need to perform a constant number of accesses to arrays PA, IPA, and LCS, which alone takes $O(\log^2 n)$ time per character. Updating PLCS takes $O(\log n)$ amortized time per character (which in the worst case becomes $O(n \log n)$) by Lemma 2. By Lemma 5 we perform overall $O(n \log n)$ symbol comparisons, each requiring two FL steps and two BWT accesses, for a total of $O(n \log^2 n)$ time. Note that a single comparison between two text prefixes cannot extend for more than n characters, therefore in the worst case a single step takes $O(n \log n)$ time. This is our delay per character. To conclude, in Sect. 4.1 we showed that our data structures take at most $nH_k + o(n \log \sigma) + O(n) + \sigma \log n + o(\sigma \log n)$ bits of space. $\square$

Finally we note that, at each step i, we can output also the *location* of the longest previous factor: this requires just one access to the prefix array PA.

5 Conclusions

We improved the state-of-the-art algorithm, from Okanohara and Sadakane [21], computing online the (succinct) LPF and LCS arrays of the text. Our improvement stems from the observation that a dynamic RMQ structure over the LCS array is not needed, as the LCS can be updated by performing a number of character comparisons that is upper-bounded by the sum of irreducible LCS values. Future extensions of this work will include reducing the delay of our algorithm (currently $O(n \log n)$). We observe that it is rather simple to obtain $O(\log^2 n)$ delay at the cost of randomizing the algorithm by employing an online Karp-Rabin fingerprinting structure such as the one described in [2]: once fast fingerprinting is available, one can quickly find the LCS between any two text prefixes by binary search. It would also be interesting to reduce the overall running time of our algorithm. This, however, does not seem straightforward to achieve, as it would require finding a faster implementation of a dynamic compressed prefix array (and its inverse) and finding a faster way of updating LCS values (possibly, with a faster dynamic succinct RMQ structure).

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms **2**(1), 53–86 (2004). https://doi.org/10.1016/S1570-8667(03)00065-0
2. Alzamel, M., et al.: Online algorithms on antipowers and antiperiods. In: Brisaboa, N.R., Puglisi, S.J. (eds.) SPIRE 2019. LNCS, vol. 11811, pp. 175–188. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32686-9_13
3. Bille, P., Christiansen, A.R., Prezza, N., Skjoldjensen, F.R.: Succinct partial sums and fenwick trees. In: Fici, G., Sciortino, M., Venturini, R. (eds.) SPIRE 2017. LNCS, vol. 10508, pp. 91–96. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67428-5_8
4. Brodal, G.S., Davoodi, P., Srinivasa Rao, S.: Path minima queries in dynamic weighted trees. In: Dehne, F., Iacono, J., Sack, J.-R. (eds.) WADS 2011. LNCS, vol. 6844, pp. 290–301. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22300-6_25
5. Burrows, M., Wheeler, D.: A block sorting data compression algorithm. Technical report, DEC Systems Research Center (1994)
6. Chairungsee, S., Charuphanthuset, T.: An approach for LPF table computation. In: Anderst-Kotsis, G., et al. (eds.) DEXA 2019. CCIS, vol. 1062, pp. 3–7. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-27684-3_1
7. Chen, G., Puglisi, S.J., Smyth, W.F.: Fast and practical algorithms for computing all the runs in a string. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 307–315. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73437-6_31
8. Cox, A.J., Garofalo, F., Rosone, G., Sciortino, M.: Lightweight LCP construction for very large collections of strings. J. Discrete Algorithms **37**, 17–33 (2016). https://doi.org/10.1016/j.jda.2016.03.003
9. Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel Ziv factorization. In: Data Compression Conference (DCC 2008), pp. 482–488 (2008). https://doi.org/10.1109/DCC.2008.36
10. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Inf. Process. Lett. **106**(2), 75–80 (2008). https://doi.org/10.1016/j.ipl.2007.10.006
11. Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Waleń, T.: Computing the longest previous factor. Eur. J. Comb. **34**(1), 15–26 (2013). https://doi.org/10.1016/j.ejc.2012.07.011
12. Franěk, F., Holub, J., Smyth, W.F., Xiao, X.: Computing quasi suffix arrays. J. Autom. Lang. Comb. **8**(4), 593–606 (2003)
13. Heliou, A., Léonard, M., Mouchard, L., Salson, M.: Efficient dynamic range minimum query. Theor. Comput. Sci. **656**(PB), 108–117 (2016). https://doi.org/10.1016/j.tcs.2016.07.002
14. Hon, W.K., Sadakane, K., Sung, W.K.: Succinct data structures for searchable partial sums with optimal worst-case performance. Theor. Comput. Sci. **412**(39), 5176–5186 (2011)
15. Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: Kucherov, G., Ukkonen, E. (eds.) CPM 2009. LNCS, vol. 5577, pp. 181–192. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02441-2_17

16. Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Amir, A. (ed.) CPM 2001. LNCS, vol. 2089, pp. 181–192. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48194-X_17
17. Manber, U., Myers, G.: Suffix arrays: a new method for on-line string searches. SIAM J. Comput. **22**(5), 935–948 (1993). https://doi.org/10.1137/0222058
18. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976). https://doi.org/10.1145/321941.321946
19. Munro, J.I., Nekrich, Y.: Compressed data structures for dynamic sequences. In: Bansal, N., Finocchi, I. (eds.) ESA 2015. LNCS, vol. 9294, pp. 891–902. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48350-3_74
20. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Comput. Surv. textbf39(1), 2-es (2007). https://doi.org/10.1145/1216370.1216372
21. Okanohara, D., Sadakane, K.: An online algorithm for finding the longest previous factors. In: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 696–707. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87744-8_58
22. Policriti, A., Prezza, N.: From LZ77 to the run-length encoded Burrows-Wheeler transform, and back. In: 28th Annual Symposium on Combinatorial Pattern Matching. Schloß Dagstuhl (2017)
23. Puglisi, S.J., Turpin, A.: Space-time tradeoffs for longest-common-prefix array computation. In: Hong, S.-H., Nagamochi, H., Fukunaga, T. (eds.) ISAAC 2008. LNCS, vol. 5369, pp. 124–135. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-92182-0_14
24. Weiner, P.: Linear pattern matching algorithms. In: 14th Annual Symposium on Switching and Automata Theory (SWAT), pp. 1–11 (1973). https://doi.org/10.1109/SWAT.1973.13
25. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977). https://doi.org/10.1109/TIT.1977.1055714