Faster Lightweight Lempel-Ziv Parsing

Dmitry Kosolobov

Ural Federal University, Ekaterinburg, Russia
dkosolobov@mail.ru

Abstract. We present an algorithm that computes the Lempel-Ziv decomposition in $O(n(\log\sigma + \log\log n))$ time and $n\log\sigma + \epsilon n$ bits of space, where ϵ is a constant rational parameter, n is the length of the input string, and σ is the alphabet size. The $n\log\sigma$ bits in the space bound are for the input string itself which is treated as read-only.

1 Introduction

The Lempel-Ziv decomposition [15] is a basic technique for data compression and plays an important role in string processing. It has several modifications used in various compression schemes. The decomposition considered in this paper is used in LZ77-based compression methods and in several compressed text indexes designed to efficiently store and search massive highly-repetitive data sets.

The standard algorithms computing the Lempel-Ziv decomposition work in $O(n\log\sigma)$[1] time and $O(n\log n)$ bits of space, where n is the length of the input string and σ is the alphabet size. It is known that this is the best possible time for the general alphabets [14]. However, for the most important case of integer alphabet, there exist algorithms working in $O(n)$ time and $O(n\log n)$ bits (see [8] for references). When σ is small, this number of bits is too big compared to the $n\log\sigma$ bits of the input string and can be prohibitive. To address this issue, several algorithms using $O(n\log\sigma)$ bits were designed.

The main contribution of this paper is a new algorithm computing the Lempel-Ziv decomposition in $O(n(\log\sigma + \log\log n))$ time and $n\log\sigma + \epsilon n$ bits of space, where ϵ is a constant rational parameter. The $n\log\sigma$ bits in the space bound are for the input string itself which is treated as read-only. The following table lists the time and space required by existing approaches to the Lempel-Ziv parsing in $O(n\log\sigma)$ bits of space.

Time	Bits of space	Note	Author(s)
$O(n\log\sigma)$	$O(n\log\sigma)$		Ohlebusch and Gog [17]
$O(n\log^3 n)$	$n\log\sigma + O(n)$	online	Okanohara and Sadakane [18]
$O(n\log^2 n)$	$O(n\log\sigma)$	online	Starikovskaya [20]
$O(n\log n)$	$O(n\log\sigma)$	online	Yamamoto et al. [21]
$O(n\log n\log\log\sigma)$	$n\log\sigma + \epsilon n$		Kärkkäinen et al. [12]
$O(n(\log\sigma + \log\log n))$	$n\log\sigma + \epsilon n$		this paper

By a more careful analysis, one can show that when ϵ is not a constant, the running time of our algorithm is $O(\frac{n}{\epsilon}(\log\sigma + \log\frac{\log n}{\epsilon}))$; we omit the details here.

[1] Throughout the paper, log denotes the logarithm with the base 2.

Preliminaries. Let w be a string of length n. Denote $|w| = n$. We write $w[0], w[1], \dots, w[n{-}1]$ for the letters of w and $w[i..j]$ for $w[i]w[i{+}1]\cdots w[j]$. A string can be *reversed* to get $\overleftarrow{w} = w[n{-}1]\cdots w[1]w[0]$ called the *reversed* w. A string u is a *substring* (or *factor*) of w if $u = w[i..j]$ for some i and j. The pair (i,j) is not necessarily unique; we say that i specifies an *occurrence* of u in w. A string can have many occurrences in another string. For $i, j \in \mathbb{Z}$, the set $\{k \in \mathbb{Z}\colon i \le k \le j\}$ is denoted by $[i..j]$; $[i..j)$ denotes $[i..j{-}1]$.

Throughout the paper, s denotes the input string of length n over the integer alphabet $[0..\sigma)$. Without loss of generality, we assume that $\sigma \le n$ and σ is a power of two. Thus, s occupies $n \log \sigma$ bits. Simplifying the presentation, we suppose that $s[0]$ is a special letter that is smaller than any letter in $s[1..n{-}1]$.

Our model of computation is the unit cost word RAM with the machine word size at least $\log n$ bits. Denote $r = \log_\sigma n = \frac{\log n}{\log \sigma}$. For simplicity, we assume that $\log n$ is divisible by $\log \sigma$. Thus, one machine word can contain a string of length $\le r$; we say that it is a *packed string*. Any substring of s of length r can be packed in a machine word in constant time by standard bitwise operations. Therefore, one can compare any two substrings of s of length k in $O(k/r + 1)$ time.

The *Lempel-Ziv decomposition of* s is the decomposition $s = z_1 z_2 \cdots z_l$ such that each z_i is either a letter that does not occur in $z_1 z_2 \cdots z_{i-1}$ or the longest substring that occurs at least twice in $z_1 z_2 \cdots z_i$ (e.g., $s = a{\cdot}b{\cdot}b{\cdot}abbabb{\cdot}c{\cdot}ab{\cdot}ab$). The substrings $z_1, z_2, \dots, z_l$ are called the *Lempel-Ziv factors*. Our algorithm consecutively reports the factors in the form of pairs $(|z_i|, p_i)$, where p_i is either the position of a nontrivial occurrence of z_i in $z_1 z_2 \cdots z_i$ (it is called an *earlier occurrence of* z_i) or z_i itself if z_i is a letter that does not occur in $z_1 z_2 \cdots z_{i-1}$. The reported pairs are not stored in main memory.

Fix a rational constant $\epsilon > 0$. It suffices to prove that our algorithm works in $O(n(\log \sigma + \log\log n))$ time and $n \log \sigma + O(\epsilon n)$ bits: the substitution $\epsilon' = c\epsilon$, where c is the constant under the bit-O, gives the required $n \log \sigma + \epsilon' n$ bits with the same working time. We use different approaches to process the Lempel-Ziv factors of different lengths. In Section 2 we show how to process "short" factors of length $<r/2$. In Section 3 we describe new compact data structures that allow us to find all "medium" factors of length $<(\log n/\epsilon)^2$. In Section 4 we apply the clever technique of [6] for the analysis of all other "long" factors.

2 Short Factors

In this section we consider the Lempel-Ziv factors of length $< r/2$, so we assume $r \ge 2$. Suppose the algorithm has reported the factors $z_1, z_2, \dots, z_{k-1}$ and now we process z_k. Denote $p = |z_1 z_2 \cdots z_{k-1}|$. We maintain arrays $H_1, H_2, \dots, H_{\lceil r/2 \rceil}$ defined as follows: for $i \in [1..\lceil \frac{r}{2} \rceil]$, the array H_i contains σ^i integers such that for any $x \in [0..\sigma^i)$, either $H_i[x]$ equals the position from $[0..p)$ of an occurrence in s of the packed string x of length i or $H_i[x] = -1$ if there are no such positions.

For each $i \in [1..r]$ and $j \in [0..n]$, denote by x_i^j the packed string $s[j..j{+}i{-}1]$. We have $H_1[x_1^p] = -1$ iff z_k is a letter that does not appear in $s[0..p{-}1]$; in this case the algorithm reports z_k immediately. Further, we have $H_{\lceil r/2 \rceil}[x^p_{\lceil r/2 \rceil}] \ne -1$

iff $|z_k| \geq \frac{r}{2}$; this case is considered in Sections 3, 4. Suppose $H_1[x_1^p] \neq -1$ and $H_{\lceil r/2 \rceil}[x^p_{\lceil r/2 \rceil}] = -1$. Our algorithm finds the minimal $q \in [2..\lceil \frac{r}{2} \rceil]$ such that $H_q[x_q^p] = -1$. Then we obviously have $|z_k| = q{-}1$ and $H_{|z_k|}[x^p_{|z_k|}]$ is the position of an earlier occurrence of z_k. Clearly, the algorithm works in $O(|z_k|)$ time.

The inequality $r = \log n/\log \sigma \geq 2$ implies $\sigma \leq \sqrt{n}$. Thus, $H_1, H_2, \ldots, H_{\lceil r/2 \rceil}$ altogether occupy at most $\sigma^{\lceil r/2 \rceil} r \log n \leq \sigma^{\frac{r}{2}} \sigma^{\frac{1}{2}} r \log n \leq n^{\frac{3}{4}} r \log n = o(n)$ bits.

To maintain $H_1, \ldots, H_{\lceil r/2 \rceil}$, we consecutively examine the positions $j = 0, 1, \ldots, p{-}1$ and for those positions, for which $H_{\lceil r/2 \rceil}[x^j_{\lceil r/2 \rceil}] = -1$, we perform the assignments $H_1[x_1^j] \leftarrow j, H_2[x_2^j] \leftarrow j, \ldots, H_{\lceil r/2 \rceil}[x^j_{\lceil r/2 \rceil}] \leftarrow j$. Hence, we execute these assignments for at most $\sigma^{\lceil r/2 \rceil}$ positions and the overall time required for the maintenance of $H_1, \ldots, H_{\lceil r/2 \rceil}$ is $O(n + r\sigma^{\lceil r/2 \rceil}) = O(n)$.

3 Medium Factors

Suppose the algorithm has reported the Lempel-Ziv factors $z_1, z_2, \ldots, z_{k-1}$ and already decided that $|z_k| \geq \frac{r}{2}$ applying the procedure of Section 2. Denote $p = |z_1 z_2 \cdots z_{k-1}|$, $\tau = \lceil \frac{\log n}{\epsilon} \rceil$, and $b = \lceil \epsilon n/(\log \sigma + \log\log n) \rceil$. We assume $p{+}b{+}\tau^2 < n$; the case $p{+}b{+}\tau^2 \geq n$ is analogous. Our algorithm processes $s[0..p{+}b]$ and reports not only z_k but also all Lempel-Ziv factors starting in positions $[p..p{+}b]$.

The algorithm consists of three phases: the first one builds for other phases an indexing data structure on the string $s[p..p{+}b]$ in $O(b \log \sigma)$ time and $O(b(\log \sigma + \log\log n)) = O(\epsilon n)$ bits; the second phase scans $s[0..p{+}b]$ in $O(n)$ time and fills a bit array $lz[0..b]$ so that for any $i \in [0..b]$, $lz[i] = 1$ iff there is a Lempel-Ziv factor starting in the position $p{+}i$; finally, the last phase scans $s[0..p{+}b]$ in $O(n)$ time and reports earlier occurrences of the found Lempel-Ziv factors. Thus, the overall time required by this algorithm is $O((n + b\log \sigma)\frac{n}{b}) = O(n(\log \sigma + \log\log n))$.

The data structures we use can search only the Lempel-Ziv factors of length $< \tau^2$; we delegate the longer factors to the procedure of Section 4. This restriction allows us to make our structures fast and compact. More precisely, our algorithm consecutively computes the lengths of the Lempel-Ziv factors starting in $[p..p{+}b]$ and once we have found a factor of length $\geq \tau^2$, we invoke the procedure of Section 4 to compute the length and an earlier occurrence of this factor.

3.1 Main Tools

Let x be a string of length $d{+}1$. Denote $\overleftarrow{x}_i = \overleftarrow{x[0..i]}$. The *suffix array of* $\overleftarrow{x}$ is the permutation $SA[0..d]$ of the integers $[0..d]$ such that $\overleftarrow{x}_{SA[0]} < \overleftarrow{x}_{SA[1]} < \ldots < \overleftarrow{x}_{SA[d]}$ in the lexicographical order. The *Burrows-Wheeler transform* [7] of $\overleftarrow{x}$ is the string $BWT[0..d]$ such that $BWT[i] = x[SA[i]{+}1]$ if $SA[i] < d$ and $BWT[i] = x[0]$ otherwise. We equip BWT with the function Ψ defined as follows: $\Psi(i) = SA^{-1}[SA[i] + 1]$ if $SA[i] < d$ and $\Psi(i) = 0$ otherwise.

Lemma 1 (see [11]). *The string BWT and the function Ψ for a string $\overleftarrow{x}$ of length $d{+}1$ over the alphabet $[0..\sigma)$ can be constructed in $O(d \log\log \sigma)$ time and $O(d \log \sigma)$ bits of space; Ψ is encoded in $O(d \log \sigma)$ bits with $O(1)$ access time.*

Example 1. Consider the string $x = \$aabadcaababadcaaba$.

$x[0..SA[i]]$	$BWT[i]$	$SA[i]$	$\Psi(i)$	i
$\$$	a	0	1	0
$\$a$	a	1	2	1
$\$aa$	b	2	11	2
$\$aabadcaa$	b	8	12	3
$\$aabadcaababadcaa$	b	16	13	4
$\$aaba$	d	4	17	5
$\$aabadcaaba$	b	10	14	6
$\$aabadcaababadcaaba$	$\$$	18	0	7
$\$aabadcaababa$	d	12	18	8
$\$aabadca$	a	7	3	9
$\$aabadcaababadca$	a	15	4	10
$\$aab$	a	3	5	11
$\$aabadcaab$	a	9	6	12
$\$aabadcaababadcaab$	a	17	7	13
$\$aabadcaabab$	a	11	8	14
$\$aabadc$	a	6	9	15
$\$aabadcaababadc$	a	14	10	16
$\$aabad$	c	5	15	17
$\$aabadcaababad$	c	13	16	18

In the *dynamic weighted ancestor (WA for short) problem* one has 1) a weighted tree, where the weight of each vertex is greater than the weight of parent, 2) the queries finding for a vertex v and number i the ancestor of v with the minimal weight $\geq i$, 3) the updates inserting new vertices. Let v be a vertex of a trie T ($v \in T$ for short). Denote by $lab(v)$ the string written on the path from the root to v. We treat tries as weighted trees: $|lab(v)|$ is the weight of v.

Lemma 2 (see [13]). *For a weighted tree with at most k vertices, the dynamic WA problem can be solved in $O(k \log k)$ bits of space with queries and updates working in $O(\log k)$ amortized time.*

One can easily modify the proof of [13] for a special case of this problem when the weights are integers $[0..\tau^2]$ and the height of the tree is bounded by τ^2.

Lemma 3. *Let T be a weighted tree with at most $m \leq n$ vertices, the weights $[0..\tau^2]$, and the height $\leq \tau^2$. The dynamic WA problem for T can be solved in $O(m(\log m + \log\log n))$ bits of space with queries and updates working in $O(1)$ amortized time using a shared table of size $o(n)$ bits.*

Proof. In [13], using $O(m \log m)$ additional bits of space, the general problem for a tree with m vertices, the weights $[0..\tau^2]$, and the height $\leq \tau^2$ is reduced to the same problem for subtrees with at most $\log\log m$ vertices and the problem of the maintenance of a set of dynamic predecessor data structures on the weights $[0..\tau^2]$ so that each of these predecessor structures contains at most τ^2 weights and all they contain $O(m)$ weights in total. Each query or update on the tree requires a constant number of queries/updates on the subtrees of size $\leq \log\log m$ and on the predecessor structures.

Since the weights are bounded by τ^2, a subtree with at most $\log\log m$ vertices fits in $O(\log\log m \log \tau) = O((\log\log n)^2)$ bits. So, we can perform queries and updates on these trees in $O(1)$ time using a shared table of size $O(2^{O((\log\log n)^2)} \log^{O(1)} n) = o(n)$ bits. Further, one can organize a dynamic predecessor data structure with at most τ^2 elements as a B-tree of a constant depth with $O(\sqrt{\tau})$-element predecessor structures on each level. Any predecessor structure with $O(\sqrt{\tau})$ weights fits in $O(\sqrt{\tau}\log\log n)$ bits and therefore, one can perform all operations on these small structures with the aid of a shared table of size $O(2^{\sqrt{\tau}\log\log n}\log^{O(1)} n) = o(n)$ bits. Thus we can perform all operations on the source predecessor structure in $O(1)$ time. $\square$

Denote by $lcp(t_1, t_2)$ the length of the longest common prefix of the strings t_1 and t_2. Denote $rlcp(i, j) = \min\{\tau^2, lcp(x'_{SA[i]}, x'_{SA[j]})\}$.

Lemma 4 (see [2]). *For a string x of length $d{+}1$, using BWT of $\overleftarrow{x}$, one can compute an array $rlcp[0..d{-}1]$ such that $rlcp[i] = rlcp(i, i{+}1)$, for $i \in [0..d)$, in $O(d\log\sigma)$ time and $O(d\log\sigma)$ bits; the array occupies $O(d\log\log n)$ bits.*

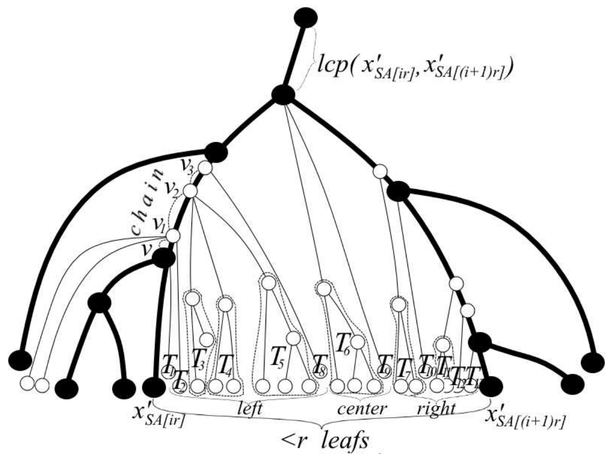
3.2 Indexing Data Structure

Trie. Denote $d = 1{+}b{+}\tau^2$. The algorithm creates a string x of length $d{+}1$ and copies the string $s[p..p{+}b{+}\tau^2]$ in $x[1..d]$; $x[0]$ is set to a special letter less than any letter in $x[1..d]$. Let SA be the suffix array of $\overleftarrow{x}$ (we use it only conceptually). Denote $x'_i = \overleftarrow{x[i{-}\tau^2{+}1..i]}$ (we assume that $x[-1], x[-2], \ldots$ are equal to $x[0]$). Here we discuss the design of our indexing data structure, a carefully packed in $O(d(\log\sigma + \log\log n))$ bits augmented compact trie of the strings $x'_0, x'_1, \ldots, x'_d$.

For simplicity, suppose d is a multiple of r. The skeleton of our structure is a compact trie Q_0 of the strings $\{x'_{SA[jr]} \colon j \in [0..d/r]\}$. We augment Q_0 with the WA structure of Lemma 3. Each vertex $v \in Q_0$ contains the following fields: 1) the pointer to the parent of v (if any); 2) the pointers to the children of v in the lexicographical order; 3) the length of $lab(v)$; 4) the length of the string written on the edge connecting v to its parent (if any).

Notice that the fields 3)–4) fit in $O(\log\log n)$ bits. Clearly, Q_0 occupies $O((d/r)\log n) = O(d\log\sigma)$ bits of space. The pointers to the substrings of x written on the edges of Q_0 are not stored, so, one cannot use Q_0 for searching.

We create an array $L[0..d/r]$ such that for $i \in [0..d/r]$, $L[i]$ is the pointer to the leaf of Q_0 corresponding to $x'_{SA[ir]}$. Now we build a compact trie Q inserting the strings $\{x'_{SA[ir+j]}\}_{j=1}^{r-1}$ in Q_0 for each $i \in [0..d/r)$ as follows. For a fixed i, these strings add to Q_0 trees $T_1, \ldots, T_l$ attached to the branches $x'_{SA[ir]}$ and $x'_{SA[(i+1)r]}$ in Q_0 (see Fig. 1). We store $T_1, \ldots, T_l$ in a contiguous memory block F_i. The pointer to F_i is stored in the leaf of Q_0 corresponding to $x'_{SA[ir]}$, so, one can find F_i in $O(1)$ time using L. Since $T_1, \ldots, T_l$ have at most $2r$ vertices in total, $O(\log\log n)$ bits per vertex suffice for the fields 1)–4). Now we discuss how $T_1, \ldots, T_l$ are attached to Q_0. Consider $v \in Q_0$ and the vertices $v_1, \ldots, v_h$ splitting the edge connecting v to its parent in Q_0. Let $T_{i_1}, \ldots, T_{i_g}$ be the trees that must be attached to $v, v_1, \ldots, v_h$ (see Fig. 1). We add to v a memory block N_v containing the WA structure of Lemma 3 for the chain $v, v_1, \ldots, v_h$ with the weights $|lab(v)|, |lab(v_1)|, \ldots, |lab(v_h)|$. Each of the vertices $v, v_1, \ldots, v_h$ in this chain contains the $O(\log\log n)$-bit pointers (inside F_i) to the roots of $T_{i_1}, \ldots, T_{i_g}$ attached to this vertex. Hence, N_v occupies $O((h+g)\log\log n)$ bits. One can



$lcp(x'_{SA[ir]}, x'_{SA[(i+1)r]})$
chain
v_3
v_2
v_1
v
T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9 T_{10} T_{11} T_{12} T_{13} T_{14}
$x'_{SA[ir]}$
left
center
right
$x'_{SA[(i+1)r]}$
$<r$ leafs

Fig. 1. Solid vertices and edges are from Q_0.

find the children for each of the vertices $v, v_1, \ldots, v_h$ in $O(1)$ time using Q_0 and the chain in the block N_v. Further, one can find, for any $j \in [1..g]$, the parent of the root of T_{i_j} in $O(1)$ time by a WA query on Q_0 to find a suitable v and a WA query on the chain in N_v. Finally, we augment each T_i with the WA structure of Lemma 3. Thus, by Lemma 3, $T_1, \ldots, T_l$ add at most $O(r \log\log n)$ bits to Q.

For each $i \in [0..d/r)$, we augment the leaf referred by $L[i]$ with an array $L_i[0..r{-}2]$ such that for $j \in [0..r{-}2]$, $L_i[j]$ is the $O(\log\log n)$-bit pointer (inside F_i) to the leaf of Q corresponding to $x'_{SA[ir+1+j]}$. So, for any $j \in [0..d]$, one can easily find the leaf of Q corresponding to $x'_{SA[j]}$ in $O(1)$ time via L and $L_{\lfloor j/r \rfloor}$. Finally, the whole described structure Q occupies $O(d(\log\sigma + \log\log n))$ bits.

Prefix links. Consider $v \in Q$. Denote by $[i_v..j_v]$ the longest segment such that for each $i \in [i_v..j_v]$, $x'_{SA[i]}$ starts with $lab(v)$ (see Fig. 2). Let BWT be the Burrows-Wheeler transform of $\overleftarrow{x}$. Denote the set of the letters of $BWT[i_v..j_v]$ by P_v. We associate with v the *prefix links* mapping each $c \in P_v$ to an integer $p_v(c) \in [i_v..j_v]$ such that $x[SA[p_v(c)]{+}1] = c$ (there might be many such $p_v(c)$; we choose any). The prefix links correspond to the well-known *Weiner-links*. Hence, Q has at most $O(d)$ prefix links. Observe that $P_u \supset P_v$ for any ancestor u of v. The problem is to store the prefix links in $O(d(\log\sigma + \log\log n))$ bits.

Fix $i \in [0..d)$. Denote by V_i the set of the vertices $v \notin Q_0$ such that v does not have descendants from Q_0 and lies between branches $x'_{SA[ir]}$ and $x'_{SA[(i+1)r]}$. We associate with each $v \in V_i$ a dictionary D_v mapping each $c \in P_v$ to $p_v(c){-}ir$ and store all D_v, for $v \in V_i$, in a contiguous memory block H_i. Since $|V_i| < r$ and P_v is a subset of $BWT[ir..(i{+}1)r]$, we have $p_v(c){-}ir \in [1..r)$ and all D_v, for $v \in V_i$, occupy overall $O(\sum_{v \in V_i} |P_v|(\log\sigma + \log\log n)) = O(r^2(\log\sigma + \log\log n))$ bits of space. Therefore, we can store in each $v \in V_i$ the $O(\log\log n)$-bit pointer to D_v (inside H_i). The pointer to H_i itself is stored in the leaf referred by $L[i]$.

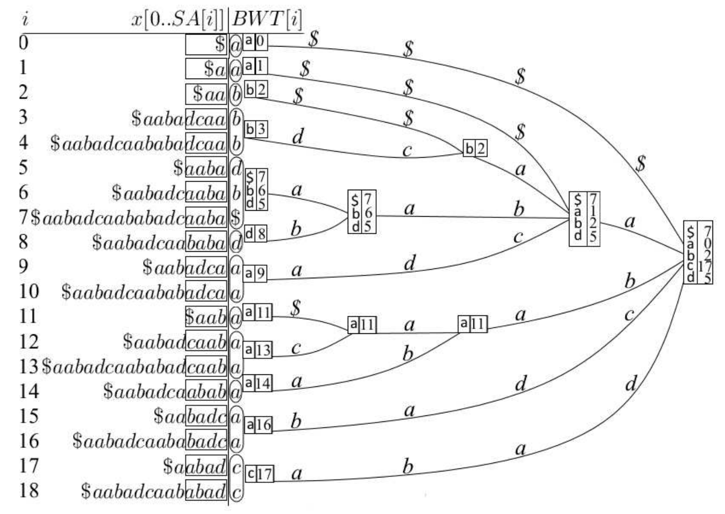


Fig. 2. $\tau^2 = 4$, the prefix links associated with vertices are in squares.

Consider $v \notin Q_0$ such that v lies on an edge connecting a vertex $w \in Q_0$ to its parent in Q_0. Let $x'_{SA[j_1r]}$ and $x'_{SA[j_2r]}$ be the strings corresponding to the leftmost and rightmost descendant leaves of w contained in Q_0. We split P_v on three subsets: $P_1 = \{c \in P_v \colon p_v(c) < j_1r\}$, $P_2 = \{c \in P_v \colon p_v(c) > j_2r\}$, $P_3 = P_v \setminus (P_1 \cup P_2)$. Clearly $P_3 \subset P_w \subset P_v$. Hence, we can use P_w instead of P_3 and store only the sets P_1 and P_2 in a way similar to that discussed above.

Suppose $v \in Q_0$. Let for $c \in P_v$, $j_c \in [i_v..j_v]$ be the position of the first occurrence of c in $BWT[i_v..j_v]$. Clearly, we can set $p_v(c) = j_c$. We add to v a dictionary mapping each $c \in P_v$ to $h_c = |\{c' \in P_v \colon j_{c'} < j_c\}|$. Denote $q = |P_v|$. Since $q \le \sigma$, the dictionary occupies $O(q \log \sigma)$ bits. Now it suffices to map h_c to j_c. Let $j'_0, \ldots, j'_{q-1}$ denote all j_c, for $c \in P_v$, in increasing order. Obviously $j'_{h_c} = j_c$. The idea is to sample each $(\tau^2 \log n)$th position in BWT. We add to v a bit array $A_v[0..q{-}1]$ indicating the sampled $j'_0, \ldots, j'_{q-1}$: $A_v[0] = 1$ and for $h \in [1..q)$, $A_v[h] = 1$ iff $j'_{h-1} < l\tau^2 \log n \le j'_h$ for an integer l; A_v is equipped with the structure of [19] supporting the queries $\mathrm{r}_{A_v}(h) = \sum_{i=0}^{h} A_v[i]$ in $O(1)$ time and $o(q)$ additional bits. The sampled sequence $\{j'_h \colon A_v[h] = 1\}$ is stored in an array B_v. Finally, we add an array $C_v[0..q{-}1]$ such that $C_v[h] = j'_h - B_v[\mathrm{r}_{A_v}(h){-}1]$. Now we map h to j'_h as follows: $j'_h = B_v[\mathrm{r}_{A_v}(h){-}1] + C_v[h]$. Clearly, each value of C_v is in the range $[0..\tau^2 \log n]$ and hence, C_v occupies $O(q \log(\tau^2 \log n)) = O(q \log\log n)$ bits. It suffices to estimate the space consumed by B_v. Since the number of the vertices in Q_0 is $O(d/r)$ and the height of Q is at most τ^2, all B_v arrays occupy at most $O((d/r) \log n + \frac{d}{\tau^2 \log n} \tau^2 \log n) = O(d \log \sigma)$ bits in total.

Construction of Q**.** Initially, Q contains one leaf corresponding to $x'_{SA[0]}$. We consecutively insert $x'_{SA[1]}, \ldots, x'_{SA[d]}$ in Q in groups of r elements. During the construction, we maintain on Q a set of the dynamic WA structures of Lemma 3 in such a way that one can answer any WA query on Q in $O(1)$ time.

Suppose we have inserted $x'_{SA[0]}, \ldots, x'_{SA[ir]}$ in Q and now we are to insert $x'_{SA[ir+1]}, \ldots, x'_{SA[(i+1)r]}$. We first allocate the memory block F_i required for new vertices. Using Lemma 4, we compute $rlcp(j{-}1, j)$ for all $j \in [ir{+}1..(i{+}1)r]$. Since $rlcp(j_1, j_2) = \min\{rlcp(j_1, j_2{-}1), rlcp(j_2{-}1, j_2)\}$, the algorithm can compute $rlcp(ir, ir{+}j)$ for all $j \in [1..r]$ in $O(r)$ time. Using the WA query on the leaf $x'_{SA[ir]}$ and the value $rlcp(ir, (i{+}1)r)$, we find the position where we insert a new leaf $x'_{SA[(i+1)r]}$. Similarly, using the WA queries, we consecutively insert $x'_{SA[ir+j]}$ for $j = 1, 2, \ldots$ as long as $rlcp(ir, ir{+}j) > rlcp(ir, (i{+}1)r)$ and then all other $x'_{SA[(i+1)r-j]}$ for $j = 1, 2, \ldots$ (Fig. 1). All related WA structures, the arrays L, L_i, the pointers, and the fields for the vertices are built in an obvious way.

One can construct the prefix links of a vertex from those of its children in $O(q \log \sigma)$ time, where q is the number of the links in the children. As there are at most $O(d)$ prefix links, one DFS traverse of Q builds them in $O(d \log \sigma)$ time.

Finally, using the result of [10], the algorithm converts in $O(d \log \sigma)$ time all dictionaries in the prefix links of the resulting trie Q in the perfect hashes with $O(1)$ access time. So, one can access any prefix link in $O(1)$ time.

3.3 Algorithm for Medium Factors

In the *dynamic marked descendant problem* one has a tree, a set of marked vertices, the queries asking whether there is a marked descendant of a given vertex, and the updates marking a given vertex. We assume that each vertex is a descendant of itself. We solve this problem on Q as follows.

Lemma 5. *In $O(d(\log \sigma + \log\log n))$ bits one can solve the dynamic marked descendant problem on Q so that any k queries and updates take $O(k + d)$ time.*

Proof. Let q be the number of the vertices in Q. Obviously $q = O(d)$. We perform a DFS traverse of Q in the lexicographical order and assign the indices $0, 1, \ldots, q{-}1$ to the vertices of Q in the order of their appearance in the traverse. Denote by $idx(v)$ the index of a vertex v. We add to our structure a bit array $M[0..q{-}1]$ initially filled with zeros. A vertex v is marked iff $M[idx(v)] = 1$. It is easy to see that the indices of the descendants of v form a contiguous segment $[idx(v)..j]$ for some $j \geq idx(v)$. So, the problem is to find for each vertex the segment of the descendant indices and then test whether there is an index k in this segment such that $M[k] = 1$.

For each $v \in Q_0$, we store $idx(v)$ and the segment of the descendant indices explicitly using $O(\log n)$ bits. Consider a vertex $v \notin Q_0$. Let the leftmost descendant leaf of v corresponds to a string $x'_{SA[j]}$, where $j = ir{-}k$ for some $i \in [0..d/r]$ and $k \in [0..r)$. Denote by u the leaf corresponding to $x'_{SA[ir]}$. Since there are at most $2r$ vertices inserted between the leaves corresponding to $x'_{SA[(i-1)r]}$ and $x'_{SA[ir]}$ and the height of Q is at most τ^2, we have $0 < idx(u) - idx(v) \leq 2r + \tau^2$. So, we store in v the value $idx(u) - idx(v)$ using $O(\log\log n)$ bits. Obviously, one can compute $idx(v)$ in $O(1)$ time using $idx(u)$ stored explicitly. The structure occupies $O((d/r)\log n + d\log\log n) = O(d(\log\sigma + \log\log n))$ bits.

Now it is sufficient to describe how to answer the queries on the segments of the dynamic bit array M. We can answer the queries on the segments of length $\le \frac{\log n}{2}$ using a shared table occupying $O(2^{\log n/2}\log^3 n) = o(n)$ bits. So, the problem is reduced to the queries on the segments of the form $[i\log n..j\log n)$. We build a perfect binary tree T with leaves corresponding to the segments $[i\log n..(i{+}1)\log n)$ for $i \in [0..q/\log n)$ (without loss of generality, we assume that q is a multiple of $\log n$ and $q/\log n$ is a power of 2). Each internal vertex v of T naturally corresponds to a segment $[i2^j\log n..(i{+}1)2^j\log n)$ for some i and $j > 0$. Denote $c = i2^j + 2^{j-1}$. We associate with v bit arrays D_v and E_v of lengths 2^{j-1} such that for any $k \in [1..2^{j-1}]$, $D_v[k{-}1] = 1$ iff there are ones in the segment $M[(c{-}k)\log n..c\log n{-}1]$ and, similarly, $E_v[k{-}1] = 1$ iff there are ones in $M[c\log n..(c{+}k)\log n{-}1]$. We construct on T the least common ancestor structure (in the case of the perfect binary tree with $O(q/\log n)$ vertices, this can be simply done in $O(q)$ bits). Then, to answer the query on a segment $[i\log n..j\log n)$, we first find in $O(1)$ time the least common ancestor v of the leaves of T corresponding to the segments $[i\log n..(i{+}1)\log n)$ and $[(j{-}1)\log n..j\log n)$ and then test appropriate bits of D_v and E_v. All in $O(1)$ time. The structure occupies $O(\frac{q}{\log n}\log\frac{q}{\log n}) = O(d)$ bits.

When we set $M[i] = 1$ for some $i \in [0..q)$, the modifications are straightforward: if the segment $[\lfloor i/\log n\rfloor..\lfloor i/\log n\rfloor{+}\log n))$ already has ones, then we are done; otherwise, for each ancestor v of the leaf of T corresponding to $[\lfloor i/\log n\rfloor..\lfloor i/\log n\rfloor{+}\log n)$, we scan the array D_v $[E_v]$ from left to right [right to left] from the appropriate position and flip all zero bits. Since there are only $O(d)$ bits in the structure, the height of T is $O(\log q) = O(\log n)$, and the updates are initiated at most $q/\log n$ times, k updates run in $O(d{+}(q/\log n)\log n{+}k) = O(d + k)$ time. □

Filling *lz***.** Denote $s_i = s[0..i]$. Let for $i \in [0..p{+}d)$, t_i denotes the longest prefix of $\overleftarrow{s_i}$ presented in Q. We add to each $v \in Q$ an $O(\log\log n)$-bit field $v.mlen$ initialized to τ^2. Also, we use an integer variable f that initially equals 0.

The algorithm increases f computing $|t_f|$ in each step and augments Q as follows. Suppose $v \in Q$ is such that t_{f-1} is a prefix of $lab(v)$ and other vertices with this property are descendants of v. We say that v *corresponds to* t_{f-1}. We are to find the vertex of Q corresponding to t_f. Suppose $p_v(s[f])$ is defined. By Lemma 1, one can compute $i = \Psi(p_v(s[f]))$ in $O(1)$ time. Obviously, $x'_{SA[i]}$ starts with $s[f]t_{f-1}$. We obtain the leaf corresponding to $x'_{SA[i]}$ in $O(1)$ time via L and $L_{\lfloor i/r\rfloor}$ and then find $w \in Q$ corresponding to t_f by the WA query on the obtained leaf and the number $\min\{\tau^2, |t_{f-1}|{+}1\}$. Suppose $p_v(s[f])$ is undefined. If v is the root of Q, then we have $|t_f| = 0$. Otherwise, we recursively process the parent u of v in the same way as v assuming $t_{f-1} = lab(u)$. Finally, once we have found $w \in Q$ corresponding to t_f, we mark the parent of w using the structure of Lemma 5 and assign $w.mlen \leftarrow \min\{w.mlen, |lab(w)|{-}|t_f|\}$.

Let $i \in [p..f{+}1]$ such that $|s[i..f{+}1]| \le \tau^2$. Suppose all positions $[0..f]$ are processed as described above. It is easy to verify that the string $s[i..f{+}1]$ has an occurrence in $s[0..f]$ iff either the vertex $v \in Q$ corresponding to $\overleftarrow{s[i..f{+}1]}$ has a marked descendant or the parent of v is marked and $|lab(v)| - v.mlen \geq |s[i..f{+}1]|$. Based on this observation, the algorithm computes lz as follows.

```
1: for ($t \gets p$; $t \leq p + b$; $t \gets t + \max\{1, z\}$) do
2:     for ($z \gets 0, v \gets$ the root of $Q$; true; $v \gets w, z \gets z + 1$) do
3:         increase $f$ processing $Q$ accordingly until $f = t + z - 1$
4:         if $z \geq \tau^2$ then invoke the procedure of Section 4 to find $z$ and break;
5:         find $w \in Q$ corresp. to $\overleftarrow{s[t..t{+}z]}$ using $v$, prefix links, WA queries
6:         if $w$ is undefined then break;
7:         if $w$ do not have marked descendants then
8:             if $parent(w)$ is not marked or $|lab(w)| - w.mlen \leq z$ then break;
9:     $lz[t{-}p] \gets 1$;
```

The lengths of the Lempel-Ziv factors are accumulated in z. The above observation implies the correctness. Line 5 is similar to the procedure described above. Since $O(n)$ queries to the prefix links and $O(n)$ markings of vertices take $O(n)$ time, by standard arguments, one can show that the algorithm takes $O(n)$ time.

Searching of occurrences. Denote by Z the set of all Lempel-Ziv factors of lengths $[r/2..\tau^2)$ starting in $[p..p{+}b]$. Obviously $|Z| = O(d/r)$. Using lz, we build in $O(d \log \sigma)$ time a compact trie R of the strings $\{\overleftarrow{z} : z \in Z\}$. We add to each $v \in R$ such that $z_v = \overleftarrow{lab(v)} \in Z$ the list of all starting positions of the Lempel-Ziv factors z_v in $[p..p{+}b]$. Obviously, R occupies $O((d/r) \log n) = O(d \log \sigma)$ bits. We construct for the strings Z a succinct Aho-Corasick automaton of [1] occupying $O((d/r) \log n) = O(d \log \sigma)$ bits. In [1] it is shown that the reporting states of the automaton can be associated with vertices of R, so that we can scan $s[0..p{+}d{-}1]$ in $O(n)$ time and store the found positions of the first occurrences of the strings Z in R. Finally, by a DFS traverse on R, we obtain for each string of Z the position of its first occurrence in $s[0..p{+}d{-}1]$. To find earlier occurrences of other Lempel-Ziv factors starting in $[p..p{+}b]$, we use the algorithms of Sections 2, 4.

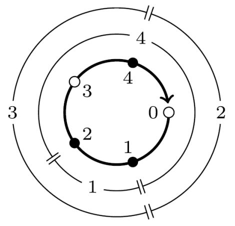
4 Long Factors

4.1 Main Tools

Let $k \in \mathbb{N}$. A set $D \subset [0..k)$ is called a *difference cover of* $[0..k)$ if for any $x \in [0..k)$, there exist $y, z \in D$ such that $y - z \equiv x \pmod{k}$. Obviously $|D| \geq \sqrt{k}$. Conversely, for any $k \in \mathbb{N}$, there is a difference cover of $[0..k)$ with $O(\sqrt{k})$ elements and it can be constructed in $O(k)$ time (see [6]).

Example 2. The set $D = \{1, 2, 4\}$ is a difference cover of $[0..5)$.

x	0	1	2	3	4
y, z	1, 1	2, 1	1, 4	4, 1	1, 2



(the figure is from [4].)

Lemma 6 (see [6]). *Let D be a difference cover of $[0..k)$. For any integers i, j, there exists $d \in [0..k)$ such that $(i - d) \bmod k \in D$ and $(j - d) \bmod k \in D$.*

An *ordered tree* is a tree whose leaves are totally ordered (e.g, a trie).

Lemma 7 (see [16]). *In $O(k \log k)$ bits of space we can maintain an ordered tree with at most k vertices under the following operations:*
1.insertion of a new leaf (possibly splitting an edge) in $O(\log k)$ time;
2.searching of the leftmost/rightmost descendant leaf of a vertex in $O(\log k)$ time.

Lemma 8 (see [3]). *A linked list can be designed to support the following operations: 1. insertion of a new element in $O(1)$ amortized time; 2. determine whether x precedes y for given elements x and y in $O(1)$ time.*

To support fast navigation in tries, we associate with each vertex v a dictionary mapping the first letters in the labels written on the outgoing edges of v to the corresponding children of v. So, whether a trie contains a string with a prefix w can be checked in $O(|w| \log \rho)$ time, where ρ is the alphabet size. Notice that a compact trie for a set of k substrings of the string s can be stored in $O(k \log n)$ bits using pointers for the edge labels. But the described searching time is too slow for our purposes, so, using packed strings and fast string dictionaries, we improve our tries with the operations provided in the following lemma.

Lemma 9. *In $O(k \log n)$ bits of space we can maintain a compact trie for at most k substrings of s under the following operations:*
1. insertion of a string w in $O(|w|/r + \log n)$ amortized time;
2. searching of a string w in $O(|u|/r + \log n)$ time, where u is the longest prefix of w present in the trie; we scan w from left to right r letters at a time and report the vertices of the trie corresponding to the prefixes of lengths $r, 2r, \ldots, \lfloor |u|/r \rfloor r$, and $|u|$ immediately after reading these prefixes.

Proof. Denote by S the set of all strings stored in T. For a substring t of the string s, denote by t' a string of length $\lfloor |t|/r \rfloor$ such that for any $i \in [0..|t'|)$, $t'[i]$ is equal to the packed string $t[ri..r(i{+}1){-}1]$. We maintain a special compact trie T' containing the set of strings $\{t' \colon t \in S\}$: the dictionaries associated with the vertices of T' are organized in such a way that the searching and insertion of a string w' both work in $O(|w'| + \log k)$ amortized time; such tries are called *dynamic ternary trees* (see [9] for a comprehensive list of references). For each $v \in T$, we insert in T' a vertex corresponding to the string t' (if there is no such

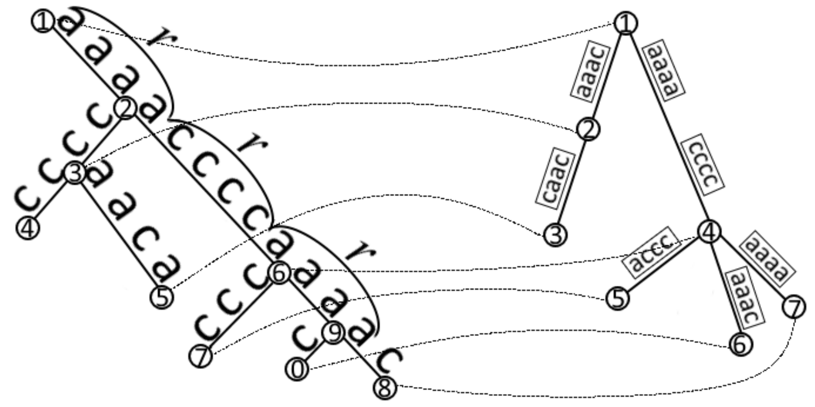


Fig. 3. A compact trie T is on the left; the corresponding ternary tree T' is on the right. If $r = 4$, the searching of $w = aaaaccccaaaac$ reports the vertices 6,6,8,8 corresponding to the prefixes of lengths r, $2r$, $3r$, and $|w|$, respectively.

vertex), where $t = lab(v)$ (consider the vertices 3 on the left and 2 on the right of Fig. 3). All vertices of T' are augmented with the pointers to the corresponding vertices of T (depicted as dashed lines in Fig. 3).

Let w be a string to be searched in T. Using the pointers of T', we can report vertices corresponding to the prefixes $w[0..r{-}1]$, $w[0..2r{-}1], \ldots, w[0..|w'|r{-}1]$ while traverse T'. Denote by u the longest prefix of w presented in T. Once u' is found in T' in $O(|u'| + \log k)$ time, we start to traverse T reading the string $u[|u'|r..|u|{-}1]$ from the position corresponding to $u[0..|u'|r{-}1]$. This operation requires additional $O(r \log \sigma) = O(\log n)$ time. The insertion is analogous. $\square$

In the *dynamic tree range reporting problem* one has ordered trees T_1 and T_2 and a set of pairs $Z = \{(x_1^i, x_2^i)\}$, where x_1^i and x_2^i are leaves of T_1 and T_2, respectively (see Fig. 4); the query asks, for given vertices $v_1 \in T_1$ and $v_2 \in T_2$, to find a pair $(x_1, x_2) \in Z$ such that x_1 and x_2 are descendants of v_1 and v_2, respectively; the update inserts new pairs in Z or new vertices in T_1 and T_2. To solve this problem, we apply the structure of [5] and Lemmas 7 and 8.

Lemma 10. *The dynamic tree range reporting problem with $|Z| \le k$ can be solved in $O(k \log k)$ bits of space with updates and queries working in $O(\log k)$ amortized time.*

Proof. To prove this Lemma, we need an additional tool. In the *dynamic orthogonal range reporting problem* one has two linked lists X and Y, and a set of pairs $Z = \{(x_i, y_i)\}$, where $x_i \in X$ and $y_i \in Y$; the query asks to report for given elements $x_1, x_2 \in X$ and $y_1, y_2 \in Y$, a pair $(x, y) \in Z$ such that x lies between x_1 and x_2 in X, and y lies between y_1 and y_2 in Y; the update inserts new pairs in Z or new elements in X or Y.

Lemma 11 (see [5]). *The dynamic orthogonal range reporting problem on at most k pairs can be solved in $O(k \log k)$ bits of space with updates and queries working in $O(\log k)$ amortized time.*

We maintain the ordered tree structure of Lemma 7 on T_1 and T_2. The order on the lists of leaves of T_1 and T_2 is maintained with the aid of enhanced linked lists of Lemma 8. To process queries efficiently, we build the dynamic orthogonal range reporting structure of Lemma 11 on these lists and the set of pairs Z. These structures take overall $O(k \log k)$ bits of space. By Lemmas 8, 7, 11, the update of T_1, T_2, or Z requires $O(\log k)$ amortized time.

Suppose we process a query for vertices $v_1 \in T_1$ and $v_2 \in T_2$. We obtain the leftmost and rightmost descendant leaves of v_1 and v_2 using Lemma 7. Then we report a desired pair from Z (or decide that there are no such pairs) using Lemma 11. By Lemmas 7 and 11, the query takes $O(\log k)$ amortized time. $\square$

4.2 Algorithm for Long Factors

Data structures. At the beginning, using the algorithm of [6], our algorithm constructs a difference cover D of $[0..\tau^2)$ such that $|D| = \Theta(\tau)$. Denote $M = \{i \in [0..n) \colon i \bmod \tau^2 \in D\}$. The set M is the basic component in our constructions.

Suppose the algorithm has reported the Lempel-Ziv factors $z_1, z_2, \ldots, z_{k-1}$ and already decided that $|z_k| \geq \tau^2$ applying the procedure of Section 3. Denote $p = |z_1 z_2 \cdots z_{k-1}|$. We use an integer variable z to compute the length of $|z_k|$ and z is initially equal to τ^2. Let us first discuss the related data structures.

We use an auxiliary variable t such that $p \leq t < p+z$ at any time of the work; initially $t = p$. Denote $s_i = s[0..i]$. Our main data structures are compact tries S and T: S contains the strings $\overleftarrow{s_i}$ and T contains the strings $s[i{+}1..i{+}\tau^2]$ for all $i \in [0..t) \cap M$ (we append τ^2 letters $s[0]$ to the right of s so that $s[i{+}1..i{+}\tau^2]$ is always defined). Both S and T are augmented with the structures supporting the searching of Lemma 9 and the tree range queries of Lemma 10 on pairs of leaves of S and T. Since $s[0]$ is a sentinel letter, each $\overleftarrow{s_i}$, for $i \in [0..t) \cap M$, is represented in S by a leaf. The set of pairs for our tree range reporting structure contains the pairs of leaves corresponding to $\overleftarrow{s_i}$ in S and $s[i{+}1..i{+}\tau^2]$ in T for all $i \in [0..t) \cap M$ (see Fig. 4). Also, we add to S the WA structure of Lemma 2.

Let us consider vertices $v \in S$ and $v' \in T$ corresponding to strings $\overleftarrow{t}_v$ and $t_{v'}$, respectively. Denote by $\mathrm{treeRng}(v, v')$ the tree range query that returns either **nil** or a suitable pair of descendant leaves of v and v'. We have $\mathrm{treeRng}(v, v') \neq$ **nil** iff there is $i \in [0..t) \cap M$ such that $s[i{-}|t_v|{+}1..i]s[i{+}1..i{+}|t_{v'}|] = \overleftarrow{t}_v t_{v'}$.

Since $|M| \leq \frac{n}{\tau^2}|D| = O(\frac{n}{\tau})$, it follows from Lemmas 2, 9, 10 that S and T with all related structures occupy at most $O(\frac{n}{\tau} \log n) = O(\epsilon n)$ bits.

The algorithm. Suppose the factor z_k occurs in a position $x \in [0..p)$; then, by Lemma 6, there is a $d \in [0..\tau^2)$ such that $x + |z_k| - d \in M$ and $p + |z_k| - d \in M$. Based on this observation, our algorithm, for each $t \in M \cap [p..z)$, finds the vertex $v \in S$ corresponding to $\overleftarrow{s[p..t]}$ and the vertex $v' \in T$ corresponding to as long as possible prefix of $s[t{+}1..n{+}\tau^2]$ such that $\mathrm{treeRng}(v, v') \neq$ **nil** and with the aid of this bidirectional search, we further increase z if it is possible.

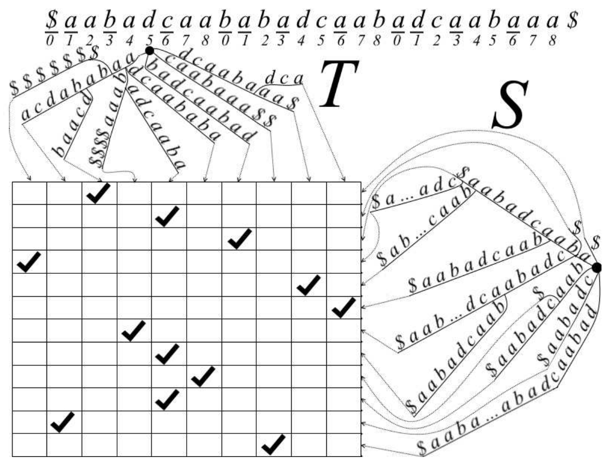


Fig. 4. $\tau = 3$, $D = \{0, 1, 3, 6\}$ is a diff. cover of $[0..\tau^2)$, positions in M are underlined.

```
1: **for** $(t \leftarrow \min\{i \geq p\colon i \in M\};\ t < p + z;\ t \leftarrow \min\{i > t\colon i \in M\})$ **do**
2:     $x \leftarrow$ the length of the longest prefix of $s[t{+}1..t{+}\tau^2]$ present in $T$
3:     $y \leftarrow$ the length of the longest prefix of $\overleftarrow{s}_t$ present in $S$
4:     **if** $y < t - p + 1$ **then** go to line 13
5:     $v \leftarrow$ the vertex corresp. to the longest prefix of $\overleftarrow{s}_t$ present in $S$
6:     $v \leftarrow \mathrm{weiAnc}(v, t - p + 1)$;
7:     **for** $j = t, t{+}r, t{+}2r, \ldots, t{+}\lfloor x/r \rfloor r, x$ and $v' \in T$ corresp. to $s[t{+}1..j]$ **do**
8:         **if** $j \geq p + z$ **then**                    $\triangleright\ |s[p..j]| > |s[p..p{+}z{-}1]|$
9:             **if** $\mathrm{treeRng}(v, v') = $ **nil then**
10:                $j \leftarrow \max\{j'\colon \mathrm{treeRng}(v, u) {\neq}$ **nil** for $u \in T$ corresp. $s[t{+}1..j']\}$;
11:            $z \leftarrow \max\{z, j - p + 1\}$;
12:            **if** $\mathrm{treeRng}(v, v') = $ **nil then break**;
13:    insert $s[t{+}1..t{+}\tau^2]$ in $T$, $\overleftarrow{s}_t$ in $S$; process the pair of the corresp. leaves
```

Some lines need further clarification. Here $\mathrm{weiAnc}(v, i)$ denotes the WA query that returns either the ancestor of v with the minimal weight $\geq i$ or **nil** if there is no such ancestor; we assume that any vertex is an ancestor of itself. Since M has period τ^2, one can compute, for any t, $\min\{i > t\colon i \in M\}$ in $O(1)$ time using an array of length τ^2 for example. The operations on T in lines 2, 13 take, by Lemma 9, $O(\tau^2/r + \log n)$ time. To perform the similar operations on S in lines 3, 5, 13, we use other techniques (discussed below) working in the same time. The loop in line 7 executes exactly the procedure described in Lemma 9. To compute j in line 10, we perform the binary search on at most r ancestors of the vertex v'; thus, we invoke treeRng $O(\log r)$ times in line 10.

Let us prove the correctness. Suppose we have $\tau^2 \leq z < |z_k|$ in some iteration. It suffices to show that the algorithm cannot terminate with this value of z. Let z_k occur in a position $x \in [0..p)$. By Lemma 6, there is a $d \in [0..\tau^2)$ such that $x + z - d \in M$ and $p + z - d \in M$. Thus, the string $s[p..p{+}z{-}d]$ is presented in S

when $t = p{+}z{-}d$ and we find the corresponding vertex v in line 6. Moreover, the string $s[p{+}z{-}d{+}1..p{+}z]$ is presented in T and we find the vertex corresponding to this or a longer string in the loop 7–12. Denote this vertex by w; w is either v' or u in line 10. Obviously, $\mathrm{treeRng}(v, w) \neq \mathbf{nil}$, so, we increase z in line 11.

Let us estimate the running time. The main loop performs $O(|z_k|/\tau)$ iterations. The operations in lines 2, 3, 5, 13 require, as mentioned above, $O(\tau^2/r + \log n)$ time (some of them will be discussed in the sequel). One WA query and one modification of the tree range reporting structure take, by Lemmas 2 and 10, $O(\log n)$ time. By Lemma 9, the traverse of T in line 7 requires $O(\tau^2/r + \log n)$ time. For each fixed t, every time we perform treeRng query in line 9, except probably for the first and last queries, we increase z by r. Hence, the algorithm executes at most $O(|z_k|/\tau + |z_k|/r)$ such queries in total. Finally, in line 10 we invoke treeRng at most $O(\log r)$ times for every fixed t. Putting everything together, we obtain $O(\frac{|z_k|}{\tau}(\tau^2/r{+}\log n){+}\frac{|z_k|}{r}\log n{+}\frac{|z_k|}{\tau}\log r \log n) = O(|z_k| \log\sigma + |z_k| \log r) = O(|z_k|(\log\sigma + \log\log n))$ overall time.

One can find the position of an early occurrence of z_k from the pairs of leaves reported in lines 9, 10. Now let us discuss how to insert and search strings in S.

Operations on S. The operations on S are based on the fact that for any $i \in [\tau^2..n) \cap M$, $i - \tau^2 \in M$. Let u and v be leaves of S corresponding to some $\overleftarrow{s_j}$ and $\overleftarrow{s_k}$. To compare $\overleftarrow{s_j}$ and $\overleftarrow{s_k}$ in $O(1)$ time via u and v, we store all leaves of S in a linked list K of Lemma 8 in the lexicographical order. To calculate $lcp(\overleftarrow{s_j}, \overleftarrow{s_k})$ in $O(\log n)$ time via u and v, we put all leaves of S in an augmented search tree B. Finally, we augment S with the ordered tree structure of Lemma 7.

Denote $s'_i = \overleftarrow{s[i{-}\tau^2{+}1..i]}$. We add to S a compact trie S' containing s'_i for all $i \in [0..t) \cap M$ (we assume $s[0]{=}s[-1]{=}\ldots$, so, S' is well-defined). The vertices of S' are linked to the respective vertices of S. Let w be a leaf of S' corresponding to a string s'_i. We add to w the set $H_w = \{(p_1^j, p_2^j)\colon j \in [0..t) \cap M \text{ and } s'_j = s'_i\}$, where p_1^j and p_2^j are the pointers to the leaves of S corresponding to $\overleftarrow{s}_{j-\tau^2}$ and $\overleftarrow{s_j}$, respectively; H_w is stored in a search tree in the lexicographical order of the strings $\overleftarrow{s}_{j-\tau^2}$ referred by p_1^j, so, one can find, for any $k \in [0..t{+}\tau^2) \cap M$, the predecessor or successor of the string $\overleftarrow{s}_{k-\tau^2}$ in H_w in $O(\log n)$ time. It is straightforward that all these structures occupy $O(\frac{n}{\tau}\log n) = O(\epsilon n)$ bits.

Suppose S contains $\overleftarrow{s_i}$ for all $i \in [0..t) \cap M$ and we insert $\overleftarrow{s_t}$. We first search s'_t in S'. Suppose S' does not contain s'_t. We insert s'_t in S' in $O(\tau^2/r + \log n)$ time, by Lemma 9, then add to S the vertices corresponding to the new vertices of S' and link them to each other. Using the structure of Lemma 7 on S, we find the position of $\overleftarrow{s_t}$ in K in $O(\log n)$ time. All other structures are easily modified in $O(\log n)$ time. Now suppose S' has a vertex w corresponding to s'_t. In $O(\log n)$ time we find in H_w the pairs (p_1^j, p_2^k) and (p_1^j, p_2^k) such that p_1^j points to the predecessor $\overleftarrow{s}_{j-\tau^2}$ of $\overleftarrow{s}_{t-\tau^2}$ in H_w and p_1^k points to the successor $\overleftarrow{s}_{k-\tau^2}$. So, the leaf corresponding to $\overleftarrow{s_t}$ must be between $\overleftarrow{s}_j$ and $\overleftarrow{s}_k$. Using B, we calculate $lcp(\overleftarrow{s}_j, \overleftarrow{s_t}) = lcp(\overleftarrow{s}_{j-\tau^2}, \overleftarrow{s}_{t-\tau^2}) + \tau^2$ and, similarly, $lcp(\overleftarrow{s}_k, \overleftarrow{s_t})$ in $O(\log n)$ time and then find the position where to insert the new leaf by WA queries on S. All other structures are simply modified in $O(\log n)$ time. Thus, the insertion takes $O(\tau^2/r + \log n)$ time. One can use a similar algorithm for the searching of $\overleftarrow{s_t}$.

References

1. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: CPM 2010. LNCS, vol. 6129, pp. 88–100. Springer (2010)
2. Beller, T., Gog, S., Ohlebusch, E., Schnattinger, T.: Computing the longest common prefix array based on the Burrows–Wheeler transform. J. of Discrete Algorithms 18, 22–31 (2013)
3. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: Algorithms-ESA 2002, LNCS, vol. 2461, pp. 152–164. Springer (2002)
4. Bille, P., Gørtz, I.L., Sach, B., Vildhøj, H.W.: Time-space trade-offs for longest common extensions. J. of Discrete Algorithms 25, 42–50 (2014)
5. Blelloch, G.E.: Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In: SODA 2008. pp. 894–903. SIAM (2008)
6. Burkhardt, S., Kärkkäinen, J.: Fast lightweight suffix array construction and checking. In: CPM 2003. LNCS, vol. 2676, pp. 55–69. Springer (2003)
7. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124 (1994)
8. Fischer, J., I, T., Köppl, D.: Lempel-Ziv computation in small space (LZ-CISS). arXiv preprint arXiv:1504.02605 (accepted to CPM 2015) (2015)
9. Franceschini, G., Grossi, R.: A general technique for managing strings in comparison-driven data structures. In: ICALP 2004. LNCS, vol. 3142, pp. 606–617. Springer (2004)
10. Hagerup, T., Miltersen, P.B., Pagh, R.: Deterministic dictionaries. J. of Algorithms 41(1), 69–85 (2001)
11. Hon, W.K., Sadakane, K., Sung, W.K.: Breaking a time-and-space barrier in constructing full-text indices. In: FOCS 2003. pp. 251–260. IEEE (2003)
12. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight Lempel-Ziv parsing. In: SEA 2013. LNCS, vol. 7933, pp. 139–150. Springer (2013)
13. Kopelowitz, T., Lewenstein, M.: Dynamic weighted ancestors. In: SODA 2007. pp. 565–574. SIAM (2007)
14. Kosolobov, D.: Lempel-Ziv factorization may be harder than computing all runs. In: STACS 2015. LIPICS, vol. 30, pp. 582–593 (2015)
15. Lempel, A., Ziv, J.: On the complexity of finite sequences. IEEE Transactions on Information Theory 22(1), 75–81 (1976)
16. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Transactions on Algorithms (TALG) 10(3), 16 (2014)
17. Ohlebusch, E., Gog, S.: Lempel-Ziv factorization revisited. In: CPM 2011. LNCS, vol. 6661, pp. 15–26. Springer (2011)
18. Okanohara, D., Sadakane, K.: An online algorithm for finding the longest previous factors. In: Algorithms-ESA 2008, LNCS, vol. 5193, pp. 696–707. Springer (2008)
19. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: SODA 2002. pp. 233–242. SIAM (2002)
20. Starikovskaya, T.: Computing Lempel-Ziv factorization online. In: MFCS 2012, LNCS, vol. 7464, pp. 789–799. Springer (2012)
21. Yamamoto, J., I, T., Bannai, H., Inenaga, S., Takeda, M.: Faster compact on-line Lempel-Ziv factorization. In: STACS 2014. LIPICS, vol. 25, pp. 675–686 (2014)