Faster Compact On-Line Lempel-Ziv Factorization

Jun'ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University, Nishiku, Fukuoka, Japan
{tomohiro.i,bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract

We present a new on-line algorithm for computing the Lempel-Ziv factorization of a string that runs in $O(N \log N)$ time and uses only $O(N \log \sigma)$ bits of working space, where N is the length of the string and σ is the size of the alphabet. This is a notable improvement compared to the performance of previous on-line algorithms using the same order of working space but running in either $O(N \log^3 N)$ time (Okanohara & Sadakane 2009) or $O(N \log^2 N)$ time (Starikovskaya 2012). The key to our new algorithm is in the utilization of an elegant but less popular index structure called Directed Acyclic Word Graphs, or DAWGs (Blumer et al. 1985). We also present an opportunistic variant of our algorithm, which, given the run length encoding of size m of a string of length N, computes the Lempel-Ziv factorization of the string on-line, in $O\left(m \cdot \min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right)$ time and $O(m \log N)$ bits of space.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Lempel-Ziv Factorization, String Index

Digital Object Identifier 10.4230/LIPIcs.STACS.2014.675

1 Introduction

The Lempel-Ziv (LZ) factorization of a string [20], discovered over 35 years ago, captures important properties concerning repeated occurrences of substrings in the string, and has numerous applications in the field of data compression, compressed full text indices [12], and is also the key component to various efficient algorithms on strings [11, 6]. Therefore, a large amount of work has been devoted to its efficient computation, especially in the *off-line* setting where the text is static, and the LZ factorization can be computed in as fast as $O(N)$ time assuming an integer alphabet, using $O(N \log N)$ bits of space (see [1] for a survey; more recent results are in [14, 10, 7, 9]). In this paper, we consider the more difficult and challenging *on-line* setting, where new characters may be appended to the end of the string. If we may use $O(N \log N)$ bits of space, the problem can be solved in $O(N \log \sigma)$ time where σ is the size of the alphabet, by use of string indices such as suffix trees [19] and on-line algorithms to construct them [18]. However, when σ is small and N is very large (e.g. DNA), the $O(N \log N)$ bits space complexity is much larger than the $N \log \sigma$ bits of the input text, and can be prohibitive. To solve this problem, space efficient on-line algorithms for LZ factorization based on succinct data structures have been proposed. Okanohara and Sadakane [15] gave an algorithm that runs in $O(N \log^3 N)$ time using $N \log \sigma + o(N \log \sigma) + O(N)$ bits of space. Later Starikovskaya [17], achieved $O(N \log^2 N)$ time using $O(N \log \sigma)$ bits of space, assuming $\log_\sigma N$ characters are packed in a machine word. Kärkkäinen et al. [8] proposed an LZ factorization algorithm that works in $O(Ntd)$ time and $N \log \sigma + O(N \log N/d)$ bits of space with $\lceil N/d \rceil$ delay (i.e., it processes $\lceil N/d \rceil$ characters in a batch) for any $d \geq 1$, where t is the time for a rank query on a string over an alphabet of size σ. When $d = \Theta(\log_\sigma N)$, their algorithm runs in $O(Nt \log N / \log \sigma)$

31st Symposium on Theoretical Aspects of Computer Science (STACS'14).
Editors: Ernst W. Mayr and Natacha Portier; pp. 675–686

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

time and $O(N \log \sigma)$ bits of space. However, the delay then becomes $\Theta(N \log \sigma / \log N)$, which seems to be too large to be called on-line.

In this paper, we propose a new on-line LZ factorization algorithm running in $O(N \log N)$ time using $O(N \log \sigma)$ bits of space, which is a notable improvement compared to the run-times of the previous on-line algorithms while still keeping the working space within a constant factor of the input text. Our algorithm is based on a novel application of a full text index called Directed Acyclic Word Graphs, or DAWGs [4], which, despite its elegance, has not received as much attention as suffix trees. To achieve a more efficient algorithm, we exploit an interesting feature of the DAWG structure that, unlike suffix trees, allows us to collect information concerning the left context of strings into each state in an efficient and on-line manner. We further show that the DAWG allows for an opportunistic variant of the algorithm which is more time and space efficient when the run length encoding (RLE) of the string is small. Given the RLE of size $m \leq N$ of the string, our on-line algorithm runs in $O\left(m \cdot \min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right) = o(m \log m)$ time using $O(m \log N)$ bits of space.

2 Preliminaries

Let $\Sigma = \{1, \ldots, \sigma\}$ be a finite integer *alphabet*. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is the string of length 0. Let $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. The set of prefixes and substrings of S are denoted by $\mathit{Prefix}(S)$ and $\mathit{Substr}(S)$, respectively. The *longest common prefix* (lcp) of strings X, Y is the longest string in $\mathit{Prefix}(X) \cap \mathit{Prefix}(Y)$. The i-th character of a string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a string S that begins at position i and ends at position j is denoted by $S[i..j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i..j] = \varepsilon$ if $j < i$. A position i is called an *occurrence* of X in S if $S[i..i + |X| - 1] = X$. For any string $S = S[1..N]$, let $S^{rev} = S[N] \cdots S[1]$ denote the reversed string. For any character $a \in \Sigma$ and integer $i \geq 0$, let $a^0 = \varepsilon$, $a^i = a^{i-1}a$. We call i the *exponent* of a^i.

The default base of logarithms will be 2. Our model of computation is the unit cost word RAM with the machine word size at least $\log N$ bits. For an input string S of length N, let $r = \log_\sigma N = \frac{\log N}{\log \sigma}$. For simplicity, assume that $\log N$ is divisible by $\log \sigma$, and that N is divisible by r. A string of length r, called a *meta-character*, consists of $\log N$ bits, and therefore fits in a single machine word. Thus, a meta-character can also be transparently regarded as an element in the integer alphabet $\Sigma^r = \{1, \ldots, N\}$. We assume that given $1 \leq i \leq N - r + 1$, any meta-character $A = S[i..i + r - 1]$ can be retrieved in constant time. Also, we can pre-compute an array of size $2^{\frac{\log N}{2}}$ occupying $O(\sqrt{N} \log N) = o(N)$ bits in $o(N)$ time, so A^{rev} can be computed in constant time given A. We call a string on the alphabet Σ^r of meta-characters, a *meta-string*. Any string S whose length is divisible by r can be viewed as a meta-string S of length $n = \frac{|S|}{r}$. We write $\langle S \rangle$ when we explicitly view string S as a meta-string, where $\langle S \rangle[j] = S[(j-1)r + 1..jr]$ for each $j \in [1, n]$. Such range $[(j-1)r + 1, jr]$ of positions will be called *meta-blocks* and the beginning positions $(j-1)r + 1$ of meta-blocks will be called *block borders*. For clarity, the length n of a meta-string $\langle S \rangle$ will be denoted by $\|\langle S \rangle\|$. Meta-strings are sometimes called *packed* strings. Note that $n \log N = N \log \sigma$.

2.1 LZ Factorization

There are several variants of LZ factorization, and as in most recent work, we consider the variant also called s-factorization [5]. The s-factorization of a string S is the factorization

$S = f_1 \cdots f_z$ where each s-factor $f_i \in \Sigma^+$ $(i = 1, \ldots, z)$ is defined as follows: $f_1 = S[1]$. For $i \geq 2$: if $S[|f_1 \cdots f_{i-1}| + 1] = c \in \Sigma$ does not occur in $f_1 \cdots f_{i-1}$, then $f_i = c$. Otherwise, f_i is the longest prefix of $f_i \cdots f_z$ that occurs at least twice in $f_1 \cdots f_i$. Notice that self-referencing is allowed, i.e., the previous occurrence of f_i may overlap with itself. Each s-factor can be represented in a constant number of words, i.e., either as a single character or a pair of integers representing the position of a previous occurrence of the factor and its length.

2.2 Tools

Let B be a bit array of length N. For any position x of B, let $rank(B, x)$ denote the number of 1's in $B[1..x]$. For any integer j, let $select(B, j)$ denote the position of the jth 1 in B. For any pair of positions x, y $(x \leq y)$ of B, the number of 1's in $B[x..y]$ can be expressed as $pc(B, x, y) = rank(B, y) - rank(B, x - 1)$. It is possible to maintain B and support rank/select queries and bit flip operations in $O(\log N)$ time, using $N + o(N)$ bits of space (e.g. Raman et al. [16]).

Directed Acyclic Word Graphs (DAWG) are a variant of suffix indices, similar to suffix trees or suffix arrays. The DAWG of a string S is the smallest partial deterministic finite automaton that accepts all suffixes of S. Thus, an arbitrary string is a substring of S iff it can be traversed from the source of the DAWG. While each edge of the suffix tree corresponds to a substring of S, an edge of a DAWG corresponds to a single character.

▶ **Theorem 1** (Blumer et al. [4]). *The numbers of states, edges and suffix links of the DAWG of string S are $O(|S|)$, independent of the alphabet size σ. The DAWG augmented with the suffix links can be constructed in an on-line manner in $O(|S| \log \sigma)$ time using $O(|S| \log |S|)$ bits of space.*

We give a more formal presentation of DAWGs below. Let $EndPos_S(u) = \{j \mid u = S[i..j], 1 \leq i \leq j \leq N\}$. Define an equivalence relation on $Substr(S)$ such that for any $u, w \in Substr(S)$, $u \equiv_S w \iff EndPos_S(u) = EndPos_S(w)$, and denote the equivalence class of $u \in Substr(S)$ as $[u]_S$. When clear from the context, we abbreviate the above notations as $EndPos$, $\equiv$ and $[u]$, respectively. Note that for any two elements in $[u]$, one is a suffix of the other. We denote by $\overleftarrow{u}$ the longest member of $[u]$. The states V and edges E of the DAWG can be characterized as $V = \{[u] \mid u \in Substr(S)\}$ and $E = \{([u], a, [ua]) \mid u, ua \in Substr(S), u \not\equiv ua\}$. We also define the set G of labeled reversed edges, called *suffix links*, by $G = \{([au], a, [u]) \mid u, au \in Substr(S), u = \overleftarrow{u}\}$. An edge $([u], a, [ua]) \in E$ is called a *primary* edge if $|\overleftarrow{u}| + 1 = |\overleftarrow{ua}|$, and a *secondary* edge otherwise. We call $[ua]$ a primary (resp. secondary) child of $[u]$ if the edge is primary (resp. secondary). By storing $|\overleftarrow{u}|$ at each state $[u]$, we can determine whether an edge $([u], a, [ua])$ is primary or secondary in $O(1)$ time using $O(|S| \log |S|)$ bits of total space.

Whenever a state $[u]$ is created during the on-line construction of the DAWG, it is possible to assign the position $pos_{[u]} = \min EndPos_S(u)$ to that state. If state u is reached by traversing the DAWG from the source with string p, this means that $p = S[pos_{[u]} - |p| + 1..pos_{[u]}]$, and thus the first occurrence $pos_{[u]} - |p| + 1$ of p can be retrieved, using $O(|S| \log |S|)$ bits of total space.

For any set P of points on a 2-D plane, consider query $find\_any(P, I_h, I_t)$ which returns an *arbitrary* element in P that is contained in a given orthogonal range $I_h \times I_t$ if such exists, and returns **nil** otherwise. A simple corollary of the following result by Blelloch [3]:

▶ **Theorem 2** (Blelloch [3]). *The 2D dynamic orthogonal range reporting problem on n elements can be solved using $O(n \log n)$ bits of space so that insertions and deletions take*

$O(\log n)$ amortized time and range reporting queries take $O(\log n + k \log n / \log\log n)$ time, where k is the number of output elements.

is that the query *find_any*(P, I_h, I_t) can be answered in $O(\log n)$ time on a dynamic set P of points. It is also possible to extend the *find_any* query to return, in $O(\log n)$ time, a constant number of elements contained in the range.

3 On-line LZ Factorization with Packed Strings

The problem setting and high-level structure of our algorithm follows that of Starikovskaya [17], but we employ somewhat different tools. The goal of this section is to prove the following theorem.

▶ **Theorem 3.** *The s-factorization of any string $S \in \Sigma^*$ of length N can be computed in an on-line manner in $O(N \log N)$ time and $O(N \log \sigma)$ bits of space.*

By on-line, we assume that the input string S is given r characters at a time, and we are to compute the s-factorization of the string $S[1..jr]$ for all $j = 1, \ldots, n$. Since only the last factor can change for each j, the whole s-factorization need not be re-calculated so we will focus on describing how to compute each s-factor f_i by extending f_i while a previous occurrence exists. We show how to maintain dynamic data structures using $O(N \log \sigma)$ bits in $O(N \log N)$ total time that allow us to (1) determine whether $|f_i| < r$ in $O(1)$ time, and if so, compute f_i in $O(|f_i| \log N)$ time (Lemma 4), (2) compute f_i in $O(|f_i| \log N)$ time when $|f_i| \geq r$ (Lemma 9), and (3) retrieve a previous occurrence of f_i in $O(|f_i| \log N)$ time (Lemma 11). Since $\sum_{i=1}^{z} |f_i| = N$, these three lemmas prove Theorem 3.

The difference between our algorithm and that of Starikovskaya [17] can be summarized as follows: For (1), we show that a dynamic succinct bit-array that supports rank/select queries and flip operations can be used, as opposed to a suffix trie employed in [17]. This allows our algorithm to use a larger meta-character size of $r = \log_\sigma N$ instead of $\frac{\log_\sigma N}{4}$ in [17], where the $1/4$ factor was required to keep the size of the suffix trie within $O(N \log \sigma)$ bits. Hence, our algorithm can pack characters more efficiently into a word. For (2), we show that by using a DAWG on the meta-string of length $n = N/r$ which occupies only $O(N \log \sigma)$ bits, we can reduce the problem of finding valid extensions of a factor to dynamic orthogonal range reporting queries, for which a space efficient dynamic data structure with $O(\log n)$ time query and update exists [3]. In contrast, Starikovskaya's algorithm uses a suffix tree on the meta-string and dynamic wavelet trees requiring $O(\log^2 n)$ time for queries and updates, which is the bottleneck of her algorithm. For (3), we develop a technique for the case $|f_i| < r$, which may be of independent interest.

In what follows, let $l_i = \sum_{k=1}^{i-1} |f_k|$, i.e., l_i is the total length of the first $i-1$ s-factors. Although our presentation assumes that N is known, this can be relaxed at the cost of a constant factor by simply restarting the entire algorithm when the length of the input string doubles.

3.1 Algorithm for $|f_i| < r$

Consider a bit array $M_k[1..N]$. For any meta-character $A \in \Sigma^r$, let $M_k[A] = 1$ iff $S[l{+}1..l{+}r] = A$ for some $0 \leq l \leq k - r$, i.e., $M_k[A]$ indicates whether A occurs as a substring in $S[1..k]$. We will dynamically maintain a single bit array representing M_k, for increasing values of k. For any short string t ($|t| < r$), let D_t and U_t be, respectively, the lexicographically smallest

and largest meta-characters having t as a prefix, namely, the bit-representation[1] of D_t is the concatenation of the bit-representation of t and $0^{(r-|t|)\log\sigma}$, and the bit-representation of U_t is the concatenation of the bit-representation of t and $1^{(r-|t|)\log\sigma}$. These representations can be obtained from t in constant time using standard bit operations. Then, the set of meta-characters that have t as a prefix can be represented by the interval $tr(t) = [D_t, U_t]$. It holds that t occurs in $S[1..k-r+|t|]$ iff some element in $M_k[D_t..U_t]$ is 1, i.e. $pc(M_k, D_t, U_t) > 0$. Therefore, we can check whether or not a string of length up to r occurs at some position $p \leq l_i$ by using M_{l_i+r-1}.

For any $0 \leq m \leq r$, let $t_m = S[l_i + 1..l_i + m]$. We have that $|f_i| < r$ iff $M_{l_i+r-1}[t_r] = 0$, which can be determined in $O(1)$ time. Assume $|f_i| < r$, and let $m_i = \max\{m \mid 0 \leq m < r, pc(M_{l_i+r-1}, D_{t_m}, U_{t_m}) > 0\}$, where $m_i = 0$ indicates that $S[l_i + 1]$ does not occur in $S[1..l_i]$. From the definition of s-factorization, we have that $|f_i| = \max(1, m_i)$. Notice that m_i can be computed by $O(|f_i|)$ rank queries on M_{l_i+r-1}, due to the monotonicity of $pc(M_{l_i+r-1}, D_{t_m}, U_{t_m})$ for increasing values of m. To maintain M_k we can use rank/select dictionaries for a dynamic bit array of length N (e.g. [16]) mentioned in Section 2. Thus we have:

▶ **Lemma 4.** *We can maintain in $O(N \log N)$ total time, a dynamic data structure occupying $N + o(N)$ bits of space that allows whether or not $|f_i| < r$ to be determined in $O(1)$ time, and if so, f_i to be computed in $O(|f_i| \log N)$ time.*

3.2 Algorithm for $|f_i| \geq r$.

To compute f_i when $|f_i| \geq r$, we use the DAWG for the meta-string $\langle S \rangle$ which we call the *packed DAWG*. While the DAWG for S requires $O(N \log N)$ bits, the packed DAWG only requires $O(N \log \sigma)$ bits. However, the complication is that only substrings with occurrences that start at block borders can be traversed from the source of the packed DAWG. In order to overcome this problem, we will augment the packed DAWG and maintain the set $Points_{[u]} = \{(A^{rev}, X) \mid ([u], X, [uX]) \in E, A\overleftarrow{u}X \in Substr(\langle S \rangle)\}$ for all states $[u]$ of the packed DAWG. A pair $(A^{rev}, X) \in Points_{[u]}$ represents that there exists an occurrence of $A\overleftarrow{u}X$ in $\langle S \rangle$, in other words, the longest element $\overleftarrow{u}$ corresponding to the state can be extended by X and still have an occurrence in $\langle S \rangle$ immediately preceded by A.

▶ **Lemma 5.** *For meta-string $\langle S \rangle$ of length n and its packed DAWG (V, E, G), the the total number of elements in $Points_{[u]}$ for all states $[u] \in V$ is $O(n)$.*

Proof. Consider edge $([u], X, [uX]) \in E$. If $\overleftarrow{u}X \neq \overleftarrow{uX}$, i.e., the edge is secondary, it follows that there exists a unique meta-character $A = \langle S \rangle[pos_{[uX]} - \|\overleftarrow{u}X\|]$ such that $A\overleftarrow{u}X \equiv_{\langle S \rangle} \overleftarrow{u}X$, namely, any occurrence of $\overleftarrow{u}X$ is always preceded by A in $\langle S \rangle$. If $\overleftarrow{u}X = \overleftarrow{uX}$, i.e., the edge is primary, then, for each distinct meta-character A preceding an occurrence of $\overleftarrow{u}X = \overleftarrow{uX}$ in $\langle S \rangle$, there exists a suffix link $([A\overleftarrow{uX}], A, [\overleftarrow{uX}]) \in G$. Therefore, each point (A^{rev}, X) in $Points_{[u]}$ can be associated to either a secondary edge from $[u]$ or one of the incoming suffix links to its primary child $[uX]$. Since each state has a unique longest member, each state has exactly one incoming primary edge. Therefore, the total number of elements in $Points_{[u]}$ for *all* states $[u]$ is equal to the total number of secondary edges and suffix links, which is $O(n)$ due to Theorem 1. ◀

[1] Assume that $0^{\log N}$ and $1^{\log N}$ correspond to meta-characters 1 and N, respectively.

▶ **Lemma 6.** *For string $S \in \Sigma^*$ of length N, we can, in $O(N \log \sigma)$ total time and bits of space and in an on-line manner, construct the packed DAWG (V, E, G) of S as well as maintain $Points_{[u]}$ for all states $[u] \in V$ so that find_any$(Points_{[u]}, I_h, I_t)$ for an orthogonal range $I_v \times I_h$ can be answered in $O(\log n)$ time.*

Proof. It follows from Theorem 1 that the packed DAWG can be computed in an on-line manner, in $O(n \log N) = O(N \log \sigma)$ time and bits of space, since the size of the alphabet for meta-strings is $O(N)$ and the length of the meta-string is $n = \frac{N}{r}$. To maintain and support *find_any* queries on *Points* efficiently, we use the dynamic data structure by Blelloch [3] mentioned in Theorem 2. Thus from Lemma 5, the total space requirement is $O(N \log \sigma)$ bits. Since each insert operation can be performed in amortized $O(\log n)$ time (no elements are deleted in our algorithm), what remains is to show that the total number of insert operations to *Points* is $O(n)$. This is shown below by a careful analysis of the on-line DAWG construction algorithm [4].

Assume we have the packed DAWG for a prefix $u = \langle S \rangle[1..\|u\|]$ of meta-string $\langle S \rangle$. Let $B = \langle S \rangle[\|u\| + 1]$ be the meta-character that follows u in $\langle S \rangle$. We group the updates performed on the packed DAWG when adding B, into the following two operations: (a) the new sink state $[uB]$ is created, and (b) a state is split.

First, consider case (a). Let $u_0 = u$, and consider the sequence $[u_1], \ldots, [u_q]$ of states such that the suffix link of $[u_j]$ points to $[u_{j+1}]$ for $0 \leq j < q$, and $[u_q]$ is the first state in the sequence which has an out-going edge labeled by B. As in [4], we use an auxiliary state $\perp$ and assume that for every meta-character $A \in \Sigma^r$ there is an edge $(\perp, A, [\varepsilon])$ leading to the source $[\varepsilon]$, so that there always exists such state $[u_q]$ in any sequence of suffix links. Note that any element of $[u_{j+1}]$ is a suffix of any element of $[u_j]$. The following operations are performed. (a-1) The primary edge from the old sink $[u]$ to the new sink $[uB]$ is created. No insertion is required for this edge since $[uB]$ has no incoming suffix links. (a-2) For each $1 \leq j < q$ a secondary edge $([u_j], B, [uB])$ is created, and the pair (C_j^{rev}, B) is inserted to $Points_{[u_j]}$, where C_j is the unique meta-character that immediately precedes $\overleftarrow{u_j}B$ in uB, i.e., $C_j = \langle uB \rangle[pos_{[uB]} - \|\overleftarrow{u_j}B\|]$. (a-3) Let $([u_q], B, w)$ be the edge with label B from state $[u_q]$. The suffix link of the new sink state $[uB]$ is created and points to w. Let $e = ([v], B, w)$ be the primary incoming edge to w, and A be the meta-character that labels the suffix link (note that $[v]$ is not necessarily equal to $[u_q]$). We then insert a new pair (A^{rev}, B) into $Points_{[v]}$.

Next, consider case (b). After performing (a), node w is split if the edge $([u_q], B, w)$ is secondary. Let $[v_1] = [v]$, and let $[v_1], \ldots, [v_k]$ be the parents of the state w of the packed DAWG for u, sorted in decreasing order of their longest member. Then, it holds that there is a suffix link from $[v_h]$ to $[v_{h+1}]$ and any element of $[v_{h+1}]$ is a suffix of any element of $[v_h]$ for any $1 \leq h < k$. Assume $\overleftarrow{v_i}B$ is the longest suffix of uB that has another (previous) occurrence in uB. (Namely, $[v_i]$ is equal to the state $[u_q]$ of (a-2) above.) If $i > 1$, then the state w is split into two states $[v_1B]$ and $[v_iB]$ such that $[v_1B] \cup [v_iB] = w$ and any element of $[v_iB]$ is a proper suffix of any element of $[v_1B]$. The following operations are performed. (b-1) The secondary edge from $[v_i]$ to w becomes the primary edge to $[v_iB]$, and for all $i < j \leq k$ the secondary edge from $[v_j]$ to w becomes a secondary edge to $[v_jB]$. The primary and secondary edges from $[v_h]$ to w for all $1 \leq h < i$ become the primary and secondary ones from $[v_h]$ to $[v_1B]$, respectively. Clearly the sets $Points_{[v_h]}$ for all $1 \leq h < i$ are unchanged. Also, since edges $([v_j], B, [v_iB])$ are all secondary, the sets $Points_{[v_j]}$ for all $i < j \leq k$ are unchanged. Moreover, the element of $Points_{[v_i]}$ that was associated to the secondary edge to w, is now associated to the suffix link from $[v_1B]$ to $[v_iB]$. Hence, $Points_{[v_i]}$ is also unchanged. Consequently, there are no updates due to edge redirection. (b-2) All outgoing edges of $[v_1B]$ are copied as outgoing edges of $[v_iB]$. Since any element of

$[v_iB]$ is a suffix of any element of $[v_1B]$, the copied edges are all secondary. Hence, we insert a pair to $Points_{[v_iB]}$ for each secondary edge, accordingly.

Thus, the total number of insert operations to *Points* for all states is linear in the number of update operations during the on-line construction of the packed DAWG, which is $O(n)$ due to [4]. This completes the proof. ◀

For any string f and integer $0 \leq m \leq \min(|f|, r-1)$, let strings $\alpha_m(f)$, $\beta_m(f)$, $\gamma_m(f)$ satisfy $f = \alpha_m(f)\beta_m(f)\gamma_m(f)$, $|\alpha_m(f)| = m$, and $|\beta_m(f)| = j'r$ where $j' = \max\{j \geq 0 \mid m + jr \leq |f|\}$. We say that an occurrence of f in S has offset m $(0 \leq m \leq r-1)$, if, in the occurrence, $\alpha_m(f)$ corresponds to a suffix of a meta-block, $\beta_m(f)$ corresponds to a sequence of meta-blocks (i.e. $\beta_m(f) \in Substr(\langle S \rangle)$), and $\gamma_m(f)$ corresponds to a prefix of a meta-block.

Let f_i^m denote the longest prefix of $S[l_i + 1..N]$ which has a previous occurrence in S with offset m. Thus, $|f_i| = \max_{0 \leq m < r} |f_i^m|$. In order to compute f_i^m, the idea is to find the longest prefix u of meta-string $\langle \beta_m(S[l_i + 1..N]) \rangle$ that can be traversed from the source of the packed DAWG while assuring that at least one previous occurrence of u in $\langle S \rangle$ is immediately preceded by a meta-block that has $\alpha_m(S[l_i + 1..N])$ as a suffix. It follows that $u = \beta_m(f_i^m)$.

▶ **Lemma 7.** *Given the augmented packed DAWG (V, E, G) of Lemma 6 of meta-string $\langle S \rangle$, the longest prefix f of any string P that has an occurrence with offset m in S can be computed in $O(\frac{|f|}{r} \log n + r \log n)$ time.*

Proof. We first traverse the packed DAWG for $\langle S \rangle$ to find $\beta_m(f)$. This traversal is trivial for $m = 0$, so we assume $m > 0$. For any string t $(|t| < r)$, let L_t and R_t be, respectively, the lexicographically smallest and largest meta-character which has t as a suffix, namely, the bit-representation of L_t is the concatenation of $0^{(r-|t|)\log\sigma}$ and the bit-representation of t, and the bit-representation of R_t is the concatenation of $1^{(r-|t|)\log\sigma}$ and the bit-representation of t. Then, the set of meta-characters that have t^{rev} as a prefix, (or, t as a suffix when reversed), can be represented by the interval $hr(t) = [L_t^{rev}, R_t^{rev}]$. Suppose we have successfully traversed the packed DAWG with the prefix $u = \langle \beta_m(P) \rangle [1..\|u\|]$ and want to traverse with the next meta-character $X = \langle \beta_m(P) \rangle [\|u\| + 1]$. If $u = \overleftarrow{u}$, i.e. only primary edges were traversed, then there exists an occurrence of $\alpha_m(P)uX$ with offset m in string S iff $find\_any(Points_{[u]}, hr(\alpha_m(P)), [X, X]) \neq \textbf{nil}$. Otherwise, if $u \neq \overleftarrow{u}$, all occurrences of u (and thus all extensions of u that can be traversed) in $\langle S \rangle$ is already guaranteed to be immediately preceded by the unique meta-character $A = \langle S \rangle [pos_{[u]} - \|u\|]$ such that $A^{rev} \in hr(\alpha_m(P))$. Thus, there exists an occurrence of $\alpha_m(P)uX$ with offset m in string S iff $([u], X, [uX]) \in E$. We extend u until *find_any* returns **nil** or no edge is found, at which point we have $\alpha_m(P)u = \alpha_m(f)\beta_m(f)$.

Now, $\gamma_m(f)$ is a prefix of meta-character $B = \langle \beta_m(P) \rangle [\|\langle u \rangle\| + 1]$. When $u = \overleftarrow{u}$, we can compute $\gamma_m(f)$ by asking $find\_any(Points_{[u]}, hr(\alpha_m(P)), tr(B[1..j]))$ for $0 \leq j < r$. The maximum j such that *find_any* does not return **nil** gives $|\gamma_m(f)|$. If $u \neq \overleftarrow{u}$, $\gamma_m(f)$ is the longest lcp between B and any outgoing edge from $[u]$. This can be computed in $O(\log n + |\gamma_m(f)|)$ time by maintaining outgoing edges from $[u]$ in balanced binary search trees, and finding the lexicographic predecessor/successor B^-, B^+ of B in these edges, and computing the lcp between them. The lemma follows since each *find_any* query takes $O(\log n)$ time. ◀

From the proof of Lemma 7, $\beta_m(f_i^m)$ can be computed in $O(\frac{|f_i^m|}{r} \log n)$ time, and for all $0 \leq m < r$, this becomes $O(|f_i| \log n)$ time. However, for computing $\gamma_m(f_i^m)$, if we simply

apply the algorithm and use $O(r \log n)$ time for each f_i^m, the total time for all $0 \leq m < r$ would be $O(r^2 \log n)$ which is too large for our goal. Below, we show that all $\gamma_m(f_i^m)$ are not required for computing $\max_{0 \leq m < r} |f_i^m|$, and this time complexity can be reduced.

Consider computing $F_m = \max_{0 \leq x \leq m} |f_i^x|$ for $m = 0, \ldots, r-1$. We first compute $\hat{f}_i^m = \alpha_m(f_i^m)\beta_m(f_i^m)$ using the first part of the proof of Lemma 7. We shall compute $\gamma_m(f_i^m)$ only when F_m can be larger than F_{m-1} i.e., $|\hat{f}_i^m| + |\gamma_m(f_i^m)| > F_{m-1}$. Since $|\gamma_m(f_i^m)| < r$, this will never be the case if $|\hat{f}_i^m| \leq F_{m-1} - r + 1$, and will always be the case if $|\hat{f}_i^m| > F_{m-1}$. For the remaining case, i.e. $0 \leq F_{m-1} - |f_i^m| < r - 1$, $F_m > F_{m-1}$ iff $|\gamma_m(f_i^m)| > F_{m-1} - |\hat{f}_i^m|$. If $u = \overleftarrow{u}$, this can be determined by a single *find_any* query with $j = F_{m-1} - |\hat{f}_i^m| + 1$ in the last part of the proof of Lemma 7, and if so, the rest of $\gamma_m(f_i^m)$ is computed using the *find_any* query for increasing j. When $u \neq \overleftarrow{u}$, whether or not the lcp between B and B^- or B^+ is greater than $F_{m-1} - |\hat{f}_i^m|$ can be checked in constant time using bit operations.

From the above discussion, each *find_any* or predecessor/successor query for computing $\gamma_m(f_i^m)$ updates F_m, or returns **nil**. Therefore, the total time for computing $F_{r-1} = |f_i|$ is $O((r + |f_i|) \log n) = O(|f_i| \log n)$.

A technicality we have not mentioned yet, is when and to what extent the packed DAWG is updated when computing f_i. Let F be the length of the current longest prefix of $S[l_i+1..N]$ with an occurrence less than $l_i + 1$, found so far while computing f_i. A self-referencing occurrence of $S[l_i + 1..l_i + F]$ can reach up to position $l_i + F - 1$. When computing f_i using the packed DAWG, F is increased by at most r characters at a time. Thus, for our algorithm to successfully detect such self-referencing occurrences, the packed DAWG should be built up to the meta-block that includes position $l_i + F - 1 + r$ and updated when F increases. This causes a slight problem when computing f_i^m for some m; we may detect a substring which only has an occurrence larger than l_i during the traversal of the DAWG. However, from the following lemma, the number of such *future* occurrences that update F can be limited to a constant number, namely two, and hence by reporting up to three elements in each *find_any* query that may update F, we can obtain an occurrence less than $l_i + 1$, if one exists. These occurrences can be retrieved in $O(\log N)$ time in this case, as described in Section 3.3.

▶ **Lemma 8.** *During the computation of f_i^m, there can be at most two future occurrences of f_i^m that will update F.*

Proof. As mentioned above, the packed DAWG is built up to the meta string $\langle S[1..s] \rangle$ where $s = \lceil \frac{l_i+F+r-1}{r} \rceil r$. An occurrence of f_i^m possibly greater than l_i can be written as $p_{m,k} = \lceil \frac{l_i}{r} \rceil r - m + 1 + kr$, where $k = 0, 1, \ldots$. For the occurrence to be able to update F and also be detected in the packed DAWG, it must hold that $s > p_{m,k} + F$. Since $l_i + F + 2r - 2 \geq s > p_{m,k} + F \geq l_i - m + 1 + kr + F$, k should satisfy $(2 - k)r \geq 1 - m$, and thus can only be 0 or 1. ◀

The main result of this subsection is the following:

▶ **Lemma 9.** *We can maintain in a total of $O(N \log N)$ time, a dynamic data structure occupying $O(N \log \sigma)$ bits of space that allows f_i to be computed in $O(|f_i| \log N)$ time, when $|f_i| \geq r$.*

3.3 Retrieving a Previous Occurrence of f_i

If $|f_i| \geq r$, let $f_i = f_i^m$, $A^{rev} \in hr(\alpha_m(f_i))$, $u = \beta_m(f_i)$, and $X \in tr(\gamma_m(f_i))$ where A and X were found during the traversal of the packed DAWG. We can obtain the occurrence of f_i by simple arithmetic on the ending positions stored at each state, i.e., from $pos_{[uX]}$ if $uX \neq \overleftarrow{uX}$

or $m = 0$, from $pos_{[AuX]}$ otherwise. State $[AuX]$ can be reached in $O(\log N)$ time from state $[uX]$, by traversing the suffix link in the reverse direction.

If $|f_i| < r$, then f_i is a substring of a meta-character. Let A_i be any previously occurring meta-character which has f_i as a prefix and satisfy $M_{l_i+r-1}[A_i] = 1$, thus giving a previous occurrence of f_i. Since A_i is any meta-character in the range $tr(f_i) = [D_{t_m}, U_{t_m}]$ with a set bit, A_i can be retrieved in $O(\log N)$ time by $A_i = select(M_{l_i+r-1}, rank(M_{l_i+r-1}, U_{t_m}))$. Unfortunately, we cannot afford to explicitly maintain previous occurrences for all N meta-characters, since this would cost $O(N \log N)$ bits of space. We solve this problem in two steps.

First, consider the case that a previous occurrence of f_i crosses a block border, i.e. has an occurrence with some offset $1 \leq m \leq |f_i| - 1$, and $f_i = \alpha_m(f_i)\gamma_m(f_i)$. For each $m = 1, \ldots, |f_i| - 1$, we ask $find\_any(Points_{[\varepsilon]}, hr(\alpha_m(f_i)), tr(\gamma_m(f_i)))$. If a pair (A^{rev}, X) is returned, this means that AX occurs in $\langle S \rangle$ and $A[r-m+1..r] = \alpha_m(f_i)$ and $X[1..\gamma_m(f_i)] = \gamma_m(f_i)$. Thus, a previous occurrence of f_i can be computed from $pos_{[AX]}$. The total time required is $O(|f_i| \log n)$. If all the *find_any* queries returned **nil**, this implies that no occurrence of f_i crosses a block border and f_i occurs only inside meta-blocks. We develop an interesting technique to deal with this case.

▶ **Lemma 10.** *For string $S[1..k]$ and increasing values of $1 \leq k \leq N$, we can maintain a data structure in $O(N \log N)$ total time and $O(N \log \sigma)$ bits of space that, given any meta-character A, allows us to retrieve a meta-character A' that corresponds to a meta block of S, and some integer d such that $A'[1 + d..r] = A[1..r - d]$ and $0 \leq d \leq d_{A,k}$, in $O(\log N)$ time, where $d_{A,k} = \min\{(l-1) \bmod r \mid 1 \leq l \leq k - r + 1, A = S[l..l + r - 1]\}$.* 7

Proof. Consider a tree T_k where nodes are the set of meta-characters occurring in $S[1..k]$. The root is $\langle S \rangle[1]$. For any meta-character $A \neq \langle S \rangle[1]$, the parent B of A must satisfy $B[2..r] = A[1..r-1]$ and $A \neq B$. Given A, its parent B can be encoded by a single character $B[1] \in \Sigma$ that occupies $\log \sigma$ bits and can be recovered from $B[1]$ and A in constant time by simple bit operations. Thus, together with M_k used in Section 3.1 which indicates which meta-characters are nodes of T_k, the tree can be encoded with $O(N \log \sigma)$ bits of space (recall that there are only N distinct meta-characters). We also maintain another bit vector X_k of length N so that we can determine in constant time, whether a node in T_k corresponds to a meta-block. The lemma can be shown if we can maintain the tree for increasing k so that for any node A in the tree, either A corresponds to a meta-block ($d_{A,k} = 0$), or, A has at least one ancestor at most $d_{A,k}$ nodes above it that corresponds to a meta-block. Assume that we have T_{k-1}, and want to update it to T_k. Let $A = S[k - r + 1..k]$. If A previously corresponded to or the new occurrence corresponds to a meta-block, then, $d_{A,k} = 0$ and we simply set $X_k[A] = 1$ and we are done. Otherwise, let $B = S[k - r..k - 1]$ and denote by C the parent of A in T_{k-1}, if there was a previous occurrence of A. Based on the assumption on T_{k-1}, let $x_B \leq d_{B,k-1} = d_{B,k}$ and x_C be the distance to the closest ancestor of B and C, respectively, that correspond to a meta-block. We also have that $d_{A,k-1} \geq x_C + 1$. If $(k-r) \bmod r \geq x_C + 1$, then $d_{A,k} = \min\{(k-r) \bmod r, d_{A,k-1}\} \geq x_C + 1$, i.e., the constraint is already satisfied and nothing needs to be done. If $(k-r) \bmod r < x_C + 1$ or there was no previous occurrence of A, we have that $d_{A,k} = (k-r) \bmod r$. Notice that in such cases, we cannot have $A = B$ since that would imply $d_{A,k} = d_{A,k-1} \neq (k-r) \bmod r$, and thus by setting the parent of A to B, we have that there exists an ancestor corresponding to a meta-block at distance $x_B + 1 \leq d_{B,k} + 1 \leq (k - r - 1) \bmod r + 1 = d_{A,k}$.

Thus, what remains to be shown is how to compute x_C in order to determine whether $(k-r) \bmod r < x_C + 1$. Explicitly maintaining the distances to the closest ancestor corresponding to a meta-block for all N meta characters will take too much space ($O(N \log \log N)$ bits).

Instead, since the parent of a given meta-character can be obtained in constant time, we calculate x_C by simply going up the tree from C, which takes $O(x_C) = O(\log N)$ time. Thus, the update for each k can be done in $O(\log N)$ time, proving the lemma. ◀

Using Lemma 10, we can retrieve a meta-character A' that corresponds to a meta-block and an integer $0 \leq d \leq d_{A_i,k}$ such that $A'[1+d..r] = A_i[1..r-d]$, in $O(\log N)$ time. Although A' may not actually occur d positions prior to an occurrence of A_i in $S[1..k]$, f_i is guaranteed to be completely contained in A' since it overlaps with A_i, at least as much as any meta-block actually occurring prior to A_i in $S[1..k]$. Thus, $f_i = A_i[1..|f_i|] = A'[1+d..d+|f_i|]$, and $(pos_{[A']} - 1)r + 1 + d$ is a previous occurrence of f_i. The following lemma summarizes this section.

▶ **Lemma 11.** *We can maintain in $O(N \log N)$ total time, a dynamic data structure occupying $O(N \log \sigma)$ bits of space that allows a previous occurrence of f_i to be computed in $O(|f_i| \log N)$ time.*

4 On-line LZ factorization based on RLE

For any string S of length N, let $RLE(S) = a_1^{p_1} a_2^{p_2} \cdots a_m^{p_m}$ denote the *run length encoding* of S. Each $a_k^{p_k}$ is called an RL factor of S, where $a_k \neq a_{k+1}$ for any $1 \leq k < m$, $p_h \geq 1$ for any $1 \leq h \leq m$, and therefore $m \leq N$. Each RL factor can be represented as a pair $(a_k, p_k) \in \Sigma \times [1..N]$, using $O(\log N)$ bits of space. As in the case with packed strings, we consider the on-line LZ factorization problem, where the string is given as a sequence of RL factors and we are to compute the s-factorization of $RLE(S)[1..j] = a_1^{p_1} \cdots a_j^{p_j}$ for all $j = 1, \ldots, m$. Similar to the case of packed strings, we construct the DAWG of $RLE(S)$ of length m, which we will call the RLE-DAWG, in an on-line manner. The RLE-DAWG has $O(m)$ states and edges and each edge label is an RL factor $a_k^{p_k}$, occupying a total of $O(m \log N)$ bits of space. If z is the number of s-factors of string S, then $z \leq 2m$. This allows us to describe the complexity of our algorithm without using z. The main result of this section follows:

▶ **Theorem 12.** *Given $RLE(S) = a_1^{p_1} a_2^{p_2} \cdots a_m^{p_m}$ of size m of a string S of length N, the s-factorization of S can be computed in $O\left(m \cdot \min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right)$ time using $O(m \log N)$ bits of space, in an on-line manner.*

Proof. Let $RLE(S) = a_1^{p_1} a_2^{p_2} \cdots a_m^{p_m}$. For any $1 \leq k \leq h \leq m$, let $RLE(S)[k..h] = a_k^{p_k} a_{k+1}^{p_{k+1}} \cdots a_h^{p_h}$. Let $Substr(RLE(S)) = \{RLE(S)[k..h] \mid 1 \leq k \leq h \leq m\}$.

Assume we have already computed $f_1, \ldots, f_{i-1}$ and we are computing a new s-factor f_i from the $(\ell_i + 1)$th position of S. Let a^d be the RL factor which contains the $(\ell_i + 1)$th position, and let t be the position in the RL factor where f_i begins.

Firstly, consider the case where $2 \leq t \leq d$. Let $p = d - t + 1$, i.e., the remaining suffix of a^d is a^p. It can be shown that a^p is a prefix of f_i. In the sequel, we show how to compute the rest of f_i. For each $j = 1, \ldots, m$ and for any out-going edge $e = ([u], b^q, [ub^q])$ of a state $[u]$ of the RLE-DAWG for $RLE(S)[1..j]$ and each character $a \in \Sigma$, define

$$\mathrm{mpe}_{[u]}(a, b^q) = \max(\{p \mid a^p \overleftarrow{u}\, b^q \in Substr(RLE(S)[1..j])\} \cup \{0\}).$$

That is, $\mathrm{mpe}_{[u]}(a, b^q)$ represents the maximum exponent of the RL factor with character a, that immediately precedes $\overleftarrow{u}\, b^q$ in $RLE(S)[1..j]$. For each pair (a, b) of characters for which there is an out-going edge $([u], b^q, [ub^q])$ from state $[u]$ and $\mathrm{mpe}_{[u]}(a, b^q) > 0$, we insert a

point $(\mathrm{mpe}_{[u]}(a, b^q), q)$ into $Pts_{[u],a,b}$. By similar arguments to the case of packed DAWGs, each point in $Pts_{[u],a,b}$ corresponds to a secondary edge, or a suffix link (labeled with a^p for some p) of a primary child, so the total number of such points is bounded by $O(m)$.

Suppose we have successfully traversed the RLE-DAWG by u with an occurrence that is immediately preceded by a^p (i.e., $a^p u$ is a prefix of s-factor f_i), and we want to traverse with the next RLE factor b^q from state $[u]$.

If $u = \overleftarrow{u}$, i.e., only primary edges were traversed, then we query $Pts_{[u],a,b}$ for a point with maximum x-coordinate in the range $[0, N] \times [q, N]$. Let (x, y) be such a point. If $x \geq p$, then since $y \geq q$, there must be a previous occurrence of $a^p \overleftarrow{u}\, b^q$, and hence $a^p \overleftarrow{u}\, b^q$ is a prefix of f_i. If there is an outgoing edge of $[u]$ labeled by b^q, then we traverse from $[u]$ to $[ub^q]$ and update the RLE-DAWG with the next RL factor b^q, and continue to extend f_i. Otherwise, it turns out that $f_i = a^p \overleftarrow{u}\, b^q$. If $x < p$, or no such point existed, then we query for a point with maximum y-coordinate in the range $[p, N] \times [0, q]$. If (x', y') is such a point, then $f_i = a^p \overleftarrow{u}\, b^{y'}$. If no such point existed, then $f_i = a^p \overleftarrow{u}$.

Otherwise (if $u \neq \overleftarrow{u}$), then all occurrences of u in $S[1..\ell_i]$ is immediately preceded by the unique RL factor $a^{p'}$ with $p' \geq p$. Thus, if $([u], b^q, [ub^q]) \in E$, then $a^p u b^q$ is a prefix of f_i. We update the RLE-DAWG with the next RL factor b^q, and continue to extend f_i. If there is no such edge, then $f_i = a^p u b^y$, where $y = \min(\max(\{k \mid ([u], b^k, [ub^k]) \in E\} \cup \{0\}) \cup \{q\})$.

Secondly, let us consider the case where $t = 1$. Let $([\varepsilon], a^g, [a^g])$ be the edge which has maximum exponent g for the character a from the source state $[\varepsilon]$. If $g < d$, then $f_i = a^g$. Otherwise, a^d is a prefix of f_i, and we traverse the RLE-DAWG in a similar way as above, while checking an immediately preceding occurrence of a^d.

If we use priority search trees of McCreight [13], and balanced binary search trees, the above queries and updates are supported in $O(\log m)$ time using a total of $O(m \log N)$ bits of space. We can do better based on the following observation. For a set T of points in a 2D plane, a point $(p, q) \in T$ is said to be dominant if there is no other point $(p', q') \in T$ satisfying both $p' \geq p$ and $q' \geq q$. Let $Dom_{[u],a,b}$ denote the set of dominant points of $Pts_{[u],a,b}$. Now, a query for a point with maximum x-coordinate in range $[0, N] \times [q, N]$ reduces to a successor query on the y-coordinates of points in $Dom_{[u],a,b}$. On the other hand, a query for a point with maximum y-coordinate in range $[p, N] \times [0, q]$ reduces to a successor query on the x-coordinate of points in $Dom_{[u],a,b}$. Hence, it suffices to maintain only the dominant points.

When a new dominant point is inserted into $Dom_{[u],a,b}$ due to an update of the RLE-DAWG, then all the points that have become non-dominant are deleted from $Dom_{[u],a,b}$. We can find each non-dominant point by a single predecessor/successor query. Once a point is deleted from $Dom_{[u],a,b}$, it will never be re-inserted to $Dom_{[u],a,b}$. Hence, the total number of insert/delete operations is linear in the size of $Dom_{[u],a,b}$, which is $O(m)$ for all the states of the RLE-DAWG. Using the data structure of [2], predecessor/successor queries and insert/delete operations are supported in $O\left(\min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right)$ time, using a total of $O(m \log N)$ bits of space.

Each state of the RLE-DAWG has at most m children and the exponents of the edge labels are in range $[1, N]$. Hence, at each state of the RLE-DAWG we can search branches in $O\left(\min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right)$ time with a total of $O(m \log N)$ bits of space, using the data structure of [2]. A final technicality is how to access the set $Dom_{[u],a,b}$ which is associated with a pair (a, b) of characters. To access $Dom_{[u],a,b}$ at each state $[u]$, we maintain two level search structures, one for the first characters and the other for the second characters of the pairs. At each state $[u]$ we can access $Dom_{[u],a,b}$ in $O\left(\min\left\{\frac{(\log\log m)(\log\log N)}{\log\log\log N}, \sqrt{\frac{\log m}{\log\log m}}\right\}\right)$

time with a total of $O(m \log N)$ bits of space, again using the data structure of [2]. This completes the proof. ◀

References

1 A. Al-Hafeedh, M. Crochemore, L. Ilie, J. Kopylov, W.F. Smyth, G. Tischler, and M. Yusufu. A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. *ACM Computing Surveys*, 45(1):Article 5, 2012.

2 Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *J. Comput. Syst. Sci.*, 65(1):38–72, 2002.

3 Guy E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proc. SODA 2008*, pages 894–903, 2008.

4 Anselm Blumer, Janet Blumer, David Haussler, Andrzej Ehrenfeucht, M. T. Chen, and Joel Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

5 Maxime Crochemore. Linear searching for a square in a word. *Bulletin of the European Association of Theoretical Computer Science*, 24:66–72, 1984.

6 Jean-Pierre Duval, Roman Kolpakov, Gregory Kucherov, Thierry Lecroq, and Arnaud Lefebvre. Linear-time computation of local periods. *Theoretical Computer Science*, 326(1-3):229–240, 2004.

7 Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. DCC 2013*, pages 133–142, 2013.

8 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. SEA 2013*, pages 139–150, 2013.

9 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. CPM 2013*, pages 189–200, 2013.

10 Dominik Kempa and Simon J. Puglisi. Lempel-Ziv factorization: fast, simple, practical. In *Proc. ALENEX 2013*, pages 103–112, 2013.

11 Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *Proc. FOCS 1999*, pages 596–604, 1999.

12 Sebastián Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Proc. CPM 2011*, pages 41–54, 2011.

13 Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.

14 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Proc. CPM 2011*, pages 15–26, 2011.

15 Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Proc. ESA 2008*, pages 696–707, 2008.

16 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct dynamic data structures. In *Proc. WADS 2001*, pages 426–437, 2001.

17 Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Proc. MFCS 2012*, pages 789–799, 2012.

18 E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

19 P. Weiner. Linear pattern-matching algorithms. In *Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory*, pages 1–11, 1973.

20 J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.