Lempel-Ziv Computation In Compressed Space (LZ-CICS⚽)

Dominik Köppl[1] and Kunihiko Sadakane[2]

[1]Department of Computer Science, TU Dortmund, Germany
[2]Graduate School of Information Science and Technology, University of Tokyo, Japan

Abstract

We show that both the Lempel-Ziv-77 and the Lempel-Ziv-78 factorization of a text of length n on an integer alphabet of size σ can be computed in $\mathcal{O}(n \lg \lg \sigma)$ time (linear time if we allow randomization) using $\mathcal{O}(n \lg \sigma)$ bits of working space. Given that a compressed representation of the suffix tree is loaded into RAM, we can compute both factorizations in $\mathcal{O}(n)$ time using $z \lg n + \mathcal{O}(n)$ bits of space, where z is the number of factors.

1 Introduction

The Lempel-Ziv-77 (LZ77) [1] and the Lempel-Ziv-78 (LZ78) [2] factorization divide a text into factors that capture repetitions in the text. Although both factorizations are found in major text processing tools like compressors or full text indices, computing any of the two factorizations is a bottleneck in terms of space and time. In practice, compressors still use a sliding window or discarding techniques to avoid high resource consumption. Hence, one might ask whether it is possible to lower the space bound in the light of recent approaches in the field of succinct data structures, while still allowing linear running time.

In this article, we show that the LZ77 and the LZ78 factorization of a text of length n on an integer alphabet of size σ can be computed with $\mathcal{O}(n \lg \sigma)$ bits of working space in either $\mathcal{O}(n)$ randomized or $\mathcal{O}(n \lg \lg \sigma)$ deterministic time.

2 Related Work

We are aware of the following results for LZ77: The currently most space efficient algorithm is due to Kosolobov [3] whose algorithm runs in $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ time and uses only εn bits of working space, provided that we have read-access to the text. A trade-off algorithm is given by Kärkkäinen et al. [4], using $\mathcal{O}(n/d)$ words of working space and $\mathcal{O}(dn)$ time. By setting $d \leftarrow \log_\sigma n$ we get $\mathcal{O}(n \lg \sigma)$ bits of working space and $\mathcal{O}(n \log_\sigma n)$ time. The algorithm of Belazzougui and Puglisi [5] derives its dominant terms in space and time from the same data structure [6] as we do; the LZ77 factorization algorithms of both papers work with the same space and time bounds. Their algorithm uses only the Burrows-Wheeler transform (BWT) [7] construction algorithm from [6]. By exchanging it with an improved version [8], we expect that their algorithm will run in deterministic linear time.

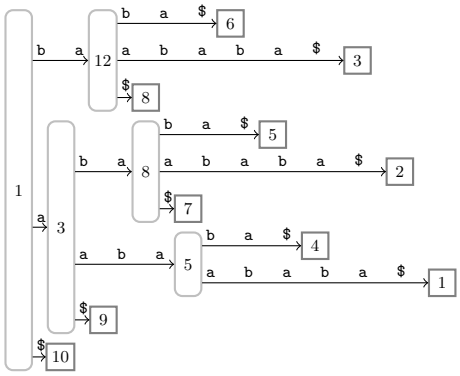
Since LZ78 factors are naturally represented in a trie, the so-called ***LZ trie***, improving LZ78 computation can be done, among others, by using sophisticated trie implementations [9, 10], or by superimposing the suffix tree with the suffix trie [11, 12]. We follow the latter approach. There, both Nakashima et al. [11] and Fischer et al. [12] presented a linear time algorithm, using $\mathcal{O}(n \lg n)$ and $(1 + \varepsilon)n \lg n + \mathcal{O}(n)$ bits of space, respectively.

Based on the data structures of the compressed suffix tree, we derive our techniques from an approach [12] using the suffix tree topology with succinct representations of the suffix array, its inverse and the longest common prefix array. For both factorization variants, Fischer et al. [12] store the inverse suffix array and parts of the enhanced suffix array in $(1 + \varepsilon)n \lg n + \mathcal{O}(n)$ bits of space such that they can access leaves of the suffix tree in text order, and can compute the string depth of internal nodes, both in constant time. Unlike the here presented approach, their algorithms overwrite the working space multiple times, and use a complicated counting for the LZ78 trie nodes.

3 Preliminaries

Our computational model is the word RAM model with word size $\Omega(\lg n)$ for some natural number n. Accessing a word costs $\mathcal{O}(1)$ time. We assume that the function $popcount(w)$, counting the set bits in a word w, can be computed in constant time. Otherwise, we build a lookup-table [13] supporting *popcount* with two rank queries in constant time. The lookup-table fits into our working space.

Let Σ denote an integer alphabet of size $\sigma = |\Sigma| \leq n$. We call an element $T \in \Sigma^*$ a ***string*** or ***text***. Its length is denoted by $|T|$. The empty string is ϵ with $|\epsilon| = 0$. We access the j-th character of T with $T[j]$ for $1 \leq j \leq |T|$. Given



i	1	2	3	4	5	6	7	8	9	10
T	a	a	b	a	a	b	a	b	a	$
SA	10	9	1	4	7	2	5	8	3	6
SA^{-1}	3	6	9	4	7	10	5	8	2	1
ψ	3	1	6	7	8	9	10	2	4	5

BP ((()(()())(()()()))(()()()))
leaves 01001001010001010100001010100 0

Figure 1: The suffix tree of $T =$ aabaababa\$. Internal nodes are labeled by their pre-order numbers, leaves by the text position where their respective suffix starts. The number of letters on an edge e is $c(e)$. Leaves are given in the BP representation by '()'. By creating rank- and select-supports on '()' and '(', we can access internal nodes and leaves separately.

$x, y, z \in \Sigma^*$ with $T = xyz$, we call x, y, and z a ***prefix***, a ***substring***, and a ***suffix*** of T, respectively. In particular, the suffix starting at position j of T is called the ***j-th suffix*** of T.

We call strings on the binary alphabet $\{0,1\}$ ***bit vectors***. For a bit vector B, we are interested in answering the following queries for $c \in \{0,1\} \cup \{0,1\}^2$:

- $B.\mathrm{rank}_c(j)$ counts the number of 'c's in $B[1,j]$, and
- $B.\mathrm{select}_c(j)$ gives the position of the j-th 'c' in B.

We can answer both types of queries due to a result of Raman et al. [14]: There is a data structure taking $o(|B|)$ extra bits of space to answer rank and select queries in constant time. It can be constructed in time linear to $|B|$. We say that a bit vector has a ***rank-support*** and a ***select-support*** if it provides constant time access to rank and select, respectively.

In the rest of this paper, we take a read-only text T of length n, which is subject to the LZ77 or the LZ78 factorization. Let $T[n]$ be a special character appearing nowhere else in T, so that no suffix of T is a prefix of another suffix of T. Without loss of generality, we assume that Σ is the *effective* alphabet of T, i.e., each character of Σ appears in T at least once. Otherwise, we can reduce the alphabet to the effective alphabet by sorting the characters with a linear time integer sorting algorithm using $n \lg \sigma + \mathcal{O}(1)$ working space [15], and an array with $\sigma \lg \sigma$ bits to reconstruct the former alphabet.

3.1 Lempel-Ziv Factorization

A ***factorization*** of T with size z partitions T into z substrings $T = f_1 \cdots f_z$. These substrings are called ***factors***. In particular, we have:

Definition 3.1. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ77 factorization** *of T iff $f_x = \mathrm{argmax}_{S \in S_j(T) \cup \Sigma} |S|$ for all $1 \le x \le z$ with $j = |f_1 \cdots f_{x-1}| + 1$, where $S_j(T)$ denotes the set of substrings of T that start strictly before j (for $1 \le j \le |T|$).*

Definition 3.2. *A factorization $f_1 \cdots f_z = T$ is called the* **LZ78 factorization** *of T iff $f_x = f'_x \cdot c$ with $f'_x = \mathrm{argmax}_{S \in \{f_y : y < x\} \cup \{\epsilon\}} |S|$ and $c \in \Sigma$ for all $1 \le x \le z$.*

3.2 Suffix Tree

The ***suffix trie*** of T is the trie of all suffixes of T. The ***suffix tree (ST)*** of T, denoted by ST, is the tree obtained by compacting the suffix trie of T. We denote the root node of ST by root. Our approach uses a compressed representation of ST, consisting of

- the ψ-array [16] with $\mathsf{SA}[i] = \mathsf{SA}[\psi(i)] - 1$ for $1 \le i \le n$ with $\mathsf{SA}[i] \neq n$ (and $\psi(i) = \mathsf{SA}^{-1}[1]$ for $\mathsf{SA}[i] = n$), and
- a $4n + o(n)$-bit balanced parenthesis representation (BP) of the tree topology [17], equipped with the minmax tree [18] for navigation.

By employing the algorithm of Belazzougui [6] on the text, we can build the compressed suffix tree consuming $\mathcal{O}(n \lg \sigma)$ bits of space in either $\mathcal{O}(n)$ randomized time or $\mathcal{O}(n \lg \lg \sigma)$ deterministic time.

Due to the BP representation, each node of the suffix tree is uniquely identified by its pre-order number. A rank- and a select-support on the BP representation enable us to address a node by its pre-order number in constant time. If the context is clear, we implicitly convert an ST node to its pre-order number, and vice versa.

LZ77	1	2	3	4	5	6
Factor	a	a	b	aaba	ba	$
Coding	a	1,1	b	1,4	3,2	$

LZ78	1	2	3	4	5	6
Factor	a	ab	aa	b	aba	$
Coding	a	1,b	1,a	b	2,a	$

Figure 2: We parse the text aabaababa$ by both factorizations. The coding represents a fresh factor by a single character, and a referencing factor by a tuple with two entries. For LZ77, this tuple consists of the referred position and the number of characters to copy. For LZ78, it consists of the referred index and a new character.

Each leaf is labeled *conceptually* by the text position where its corresponding suffix starts (see Section 3.2). We write $\mathit{label}(\ell)$ for the label of a leaf ℓ. Reading the leaf labels in depth first order returns the suffix array, which we denote by SA. We do neither store SA nor the leaf labels.

For descriptive purposes, we define the conceptional function $c(e)$ returning, for each edge e, the length of e.

We use the following methods on the ST topology that are well known to be computable in constant time after suitable preprocessing (see [18]): $\mathit{parent}(v)$ selects the parent of the node v, $\mathit{depth}(v)$ returns the depth of the node v, $\mathit{level\_anc}(\ell, d)$ selects the ancestor of the leaf ℓ at depth d, $\mathit{leaf\_select}(i)$ selects the i-th leaf, $\mathit{leaf\_rank}(\ell)$ returns the number of preceding leaves of the leaf ℓ, $\mathit{child\_rank}(v)$ returns the number of preceding siblings of the node v, and $v.\mathit{child}(i)$ selects the i-th child of the node v.

Besides those basic tools, we need the following supplementary functions whose implementation details follow their descriptions:

$\mathit{head}(\ell)$ retrieves the first character of the suffix whose starting position coincides with the label of the leaf ℓ. Since Σ is the effective alphabet of T, each character of Σ occurs in T. So root has σ children, each corresponding to a different character. Besides, the order of root's children and of the characters of Σ is the same. Hence, $\mathit{child\_rank}(\mathit{level\_anc}(\ell, \mathit{depth}(\mathsf{root}) + 1)) = \mathit{head}(\ell)$ holds, and the left hand side can be computed in constant time.

smallest_leaf selects the leaf with the label 1. By a linear scan over the ψ-array, we can find the value $\alpha := \mathsf{SA}^{-1}[1]$, so that $\psi^k[\alpha] = \mathsf{SA}^{-1}[k+1]$ for $0 \le k \le n-1$. We store α to answer *smallest_leaf* by $\mathit{leaf\_select}(\alpha)$.

$\mathit{next\_leaf}(\ell)$ selects the leaf labeled with $\mathit{label}(\ell) + 1$. We can compute it in constant time, since $\mathit{next\_leaf}(\ell) = \mathit{leaf\_select}(\psi[\mathit{leaf\_rank}(\ell)])$.

$\mathit{str\_depth}(v)$ returns the string depth of an *internal* node. We use the ψ-array and the *head*-function to compute $\mathit{str\_depth}(v)$ in time proportional to the string depth. Therefore, we take two different children of v (they exist since v is an internal node), and choose an arbitrary leaf in the subtree of each child. So we have two leaves representing two suffixes whose longest common prefix is the string read from the edge labels on the path from the root to the lowest common ancestor (LCA) of both leaves. Our task is to compute the length of this prefix. To this end, we match the first characters of both suffixes by the *head*-function. If they match, we use ψ to move to the next pair of suffixes, and apply the *head*-function again. Informally, applying ψ strips the first character of both suffixes (like taking a suffix link). On a mismatch, we find the first pair of characters that does not belong to the path from the root to v (reading the labels of the edges along this path). We return the number of matched characters as the string depth.

4 Common Settings

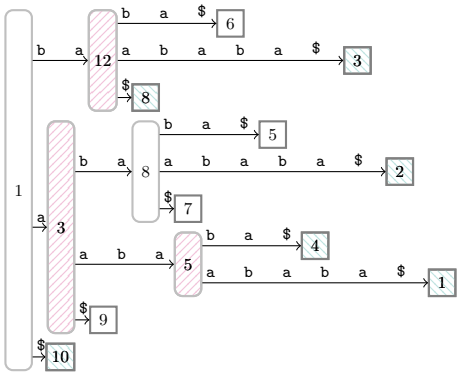
We identify factors by text positions, i.e., we call a text position j the ***factor position*** of f_x ($1 \le x \le z$) iff the factor f_x starts at position j. A factor f_x may refer to either (LZ77) a previous text position j (called f_x's ***referred position***), or (LZ78) to a previous factor f_y (called f_x's ***referred factor***—in this case y is also called the ***referred index*** of f_x). If there is no suitable reference found for a given factor f_x with factor position j, then f_x consists of just the single letter $T[j]$. We call such a factor a ***fresh factor***. The other factors are called ***referencing factors***. Let z_{R} denote the number of referencing factors. An example is given in Section 4. A more sophisticated example can be found in the full version of the paper [19].

Common to our LZ77- and LZ78-factorization algorithms is the traversal of the compressed suffix tree. In more detail, they share a common framework, which we describe in the following by introducing some new keywords:

Witnesses. Witnesses are *internal* nodes that act as signposts for finding (LZ77) the referred position or (LZ78) the referred index of a factor. The *number of witnesses* z_{W} is at most the number of *referencing* factors z_{R}. We will enumerate the witnesses from 1 to z_{W} by a bit vector B_{W} on the BP of ST with a rank_1-support. So each witness has, along with its pre-order number, a so-called ***witness id*** (its B_{W}-rank).

Passes. Like the LZ77 algorithm in [12], we divide our algorithms in several passes. In a pass, we visit the leaves of ST in text position order. This is done by using *smallest_leaf* and then calling *next_leaf* successively. The passes differ in how a leaf is processed. While processing a leaf ℓ, we want to access $\mathit{label}(\ell)$. We can track the label of the current leaf with a counter variable, since we start at the leaf with the label 1.

Corresponding Leaves. We say that a leaf ℓ ***corresponds to*** the factor f if $\mathit{label}(\ell)$ is the factor position of f. During a pass, we keep track of whether a visited leaf corresponds to a factor. To this end, for each leaf ℓ corresponding to a factor f, we compute the length of f while processing ℓ. This length tells us the number of leaves



pre-order number	1	3	5	8	12
B_{W}	0	1	1	0	1

witness id	1	2	3
W	1	1	3

Figure 3: Our LZ77 algorithm determines the witness nodes and the leaves corresponding to factors in Pass (a). Considering our running example $T = \texttt{aabaababa\$}$, the witness nodes are the nodes with the pre-order numbers $3, 5$, and 12, and the leaves corresponding to factors have the labels $1, 2, 3, 4, 8$, and 10. Each witness w is the lowest ancestor of a leaf corresponding to a factor f with the property that the referred position of f is the label of a leaf contained in w's subtree. For instance, the leaf corresponding to the 5-th factor has the label 8. Its witness has pre-order number 12, leading to the leaf with the label 3. So the referred position of the 5-th factor is 3. The length of the 5-th factor is the string depth of its witness. We show W and B_{W} after Pass (b). In this example, $z_{\mathrm{W}} = z_{\mathrm{R}} = 3$.

after ℓ (in *text order*) that do not correspond to a factor. By noting the next corresponding leaf, we know whether the current leaf is corresponding to a factor — remember that a pass selects leaves successively in *text order*, and *smallest_leaf* is always corresponding to the first factor.

Output Space. Given ST and ψ, we analyze our algorithms for both factorizations with respect to time and working space. We assume that the output can be streamed sequentially in the order of the factor indices. Therefore, we do *not* analyze the output space.

Loaded Data Structures in RAM. We need the ψ-array taking $\mathcal{O}(n \lg \sigma)$ bits, and ST's topology plus some rank- and select-support data structures consuming $4n + o(n)$ bits. Our algorithms do *not* access the text T.

5 LZ77

Given that the above stated data structures are loaded into RAM, we show that the LZ77 factorization can be computed with $2n + z \lg n + o(n)$ bits of working space.

LZ77 Passes. Common to all passes is the following procedure: For each visited leaf ℓ, we perform a leaf-to-root traversal, i.e., we visit every node on the path from ℓ to root. But we visit every node at most once, i.e., we stop the leaf-to-root traversal on visiting an already visited node. Therefore, we create a bit vector B_{V} with which we mark a visited node. This bit vector is cleared before a pass starts. Since ST contains at most $n - 1$ internal nodes, a pass can be conducted in linear time.

We perform two passes:

(a) create B_{W} in order to determine the witnesses, and
(b) stream the output by using an array mapping witness ids to text positions.

Pass (a). We follow the approach from [12]. Determining the witnesses is done in the following way: Reaching the root from a leaf corresponding to a factor means that we found a fresh factor. Otherwise, assume that we visit an already visited node $u \neq \mathsf{root}$ from a leaf ℓ. If ℓ corresponds to a factor f, u *witnesses* the referred position of f. This means that there is a suffix starting before $\mathit{label}(\ell)$ having a prefix equal to the string read from the edge labels on the path from the root to u. Moreover, u is the lowest node in the set comprising the lowest common ancestors of ℓ with all already visited leaves. So the factor corresponding to ℓ has to refer to a text position coinciding with the label of a leaf belonging to u's subtree. In order to find the referred position in the next pass, we mark u in B_{W}. Additionally, we compute the length of f with $\mathit{str\_depth}(u)$, and note the next factor position.

After this pass, we have determined the z_{W} witnesses by the '1's stored in B_{W}. We use the witnesses in the next pass to compute the referred positions (see Figure 3).

Pass (b). We clear B_{V}, create a rank-support on B_{W} and allocate an array W consuming $z_{\mathrm{W}} \lg n$ bits. We use W to map a witness id to a text position. Having this array as a working space, $W[w]$ becomes the label of the leaf from which we visited the witness w in the first place. So we find the referred position of a referencing factor f in $W[w]$ when visiting w again from a different leaf corresponding to f. The length of f is the string depth of w. Since fresh factors consist of single characters, we can output a fresh factor by applying the *head*-function to its corresponding leaf.

6 LZ78

A natural representation of the LZ78 factors is a trie, the so-called ***LZ trie***. Each node in the trie represents a factor and is labeled by its index. If the x-th factor refers to the y-th factor, then there is a node u having a child v such that u and v have the unique labels y and x, respectively. The edge (u, v) is labeled by the last character of the x-th factor (the newly introduced character). A node with the label x is the child of the root iff the x-th factor is a fresh factor.

The LZ trie representation is used in the algorithm presented below. By building the LZ trie topology on ST, our streaming algorithm computes the factorization with $5n + z \lg z + o(n)$ additional bits of working space.

Superimposition. The main idea is the superimposition of the suffix trie on the suffix tree, borrowed from Nakashima et al. [11]: The LZ trie is a connected subgraph of the suffix trie containing its root (see Figure 4). Regarding the suffix tree, the LZ nodes are either already represented by an ST node (explicit), or lie on an ST edge (implicit). To ease explanation, we identify each edge $e = (u, v)$ of ST uniquely with its ending node v, i.e., we implicitly convert between the edge e and its in-going node v (each node except root is associated with an edge). In order to address all LZ nodes, we keep track of how far an edge on the suffix tree got explored during the parsing. To this end, for an edge $e = (u, v)$, we define the ***exploration counter*** $0 \leq n_v \leq c(e)$ storing how far e is explored. If $n_v = 0$, then the factorization has not (yet) explored e, whereas $n_v = c(e)$ tells us that we have already reached v. Unfortunately, storing n_v in an integer array for all edges costs us $2n \lg n$ bits.

Our idea is to choose different representations of the exploration counters dependent on the state (not, partially or fully explored) and the number of descendants of a node. First, we mark the fully explored edges in a bit vector B_{V} (dynamically) such that we do not need to store their exploration counters. Further, we do not represent n_v for a node v with parent u until n_u got fully explored. Now let us focus on the rest of the nodes. We classify each node v based on the number of descendants of v, and select an explicit representation if this number is large, otherwise we maintain n_v implicitly. The classification of v is based on the following definitions borrow from [20]: If v's subtree has at most $\lg n$ nodes, we call v ***micro***. If v is not a micro node, but all its children are micro, then we call v a ***jump node***. So a subtree rooted at a jump node contains at least $\lg n$ nodes, and the subtrees rooted at different jump nodes are pairwise disjoint (they do not share a node). This means that there are at most $n/\lg n$ jump nodes. We mark each jump node v in a bit vector B_{J}, and store n_v in an integer array J. The array J has at most $n/\lg n$ entries and therefore consumes at most $(n/\lg n)\lg n = n$ bits. Let us consider a node v that is not micro and whose in-going edge did get partially explored. If v is a jump node, then we look-up n_v in J. Otherwise, v has at least one descendant that is a jump node. Since v's in-going edge is not fully explored, the exploration counters of all edges in the subtree rooted at v are zero. So we can abuse an exploration counter of a jump node belonging to this subtree to represent n_v until n_v gets full. For instance, we can always use the leftmost jump node of v that can be accessed by $B_{\mathrm{J}}.\mathrm{select}_1(B_{\mathrm{J}}.\mathrm{rank}_1(v) + 1)$.

The exploration counters of the micro nodes are maintained implicitly by a bit vector marking visited corresponding leaves: During a pass, when exploring a new factor on the in-going edge of a micro node v, we mark the currently accessed leaf (which will always be a leaf in the subtree rooted at v) in a bit vector B_{C}. By applying *popcount* to B_{C}, we can count how many leaves had been accessed belonging to the subtree rooted at v. This number is exactly n_v. We can compute n_v in constant time, since there are at most $\lg n$ leaves in the subtree rooted at v. After fully exploring the edge of v, we clear the area in B_{C} belonging to the leaves contained in v's subtree. By doing so, the counter n_u of every micro child u of v is reset.

Applying this procedure during a pass, we can determine the fully explored edges and collect n_v of each node v whose in-going edge got partially explored (by the definition of the jump nodes, and since we clear parts of B_{C} after full exploration).

We do two passes:

(a) create B_{W} so we can address the witnesses, and
(b) stream the output by using a helper array mapping witness ids to factor indices.

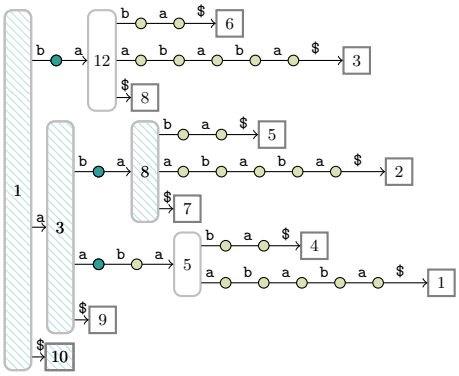
We explain the passes in detail, after introducing their commonality and a helpful lemma:

LZ78 Passes. Since referencing factors address factor indices (z options) instead of text positions (n options), we are only interested in the leaves corresponding to a factor. Starting with *smallest_leaf*, which corresponds to the first factor, we can compute the length of the factor corresponding to the currently accessed leaf so that we know the distance (in text positions) to the next corresponding leaf.

Lemma 6.1 ([12, Lemma 4]). *Let $e = (u, v)$ be an ST edge, and u the parent of the node v. Then $n_v \leq \min(c(e), s)$, where s is the number of leaves of the subtree rooted at v.*

Pass (a). The main goal of this pass is to determine the topology of the LZ trie with respect to the superimposition. Starting with an LZ trie consisting only of the root, we build the LZ trie successively by filling up the exploration counters. If the exploration counter of an edge is filled up, we mark its in-going node in the bit vector B_{V}.

Assume that we visit a leaf ℓ. We want to find the first edge on the path from root to ℓ that is either unexplored or partially explored. By invoking level ancestor queries, we traverse from the root to an edge $e = (u, v)$, where $n_v < c(e)$



SA	10	9	1	4	7	2	5	8	3	6
B_C	0	0	0	1	0	0	0	0	0	1

pre-order number	1	3	5	8	12
B_V	0	1	0	1	0
B_W	0	1	1	1	1

Figure 4: We can get the suffix trie (conceptually) by exchanging every ST edge e with $c(e) - 1$ new suffix trie nodes superimposing e. These new suffix trie nodes are the small rounded nodes in the depicted tree. They represent the implicit suffix trie nodes, while the remaining ST nodes represent the explicit suffix trie nodes. Dark colored and hatched nodes represent the nodes of the LZ78 trie. We show B_C, B_V, and B_W after Pass (a).

and u is (already) represented as a node in the LZ trie. If v is an internal node with $n_v = 0$, we make v a witness by marking v in B_W (the idea is that the edge e is *superimposed* by some LZ nodes).

Regardless that, we add a new factor by incrementing n_v. If the edge e now got fully explored, we additionally mark v in B_V. Whether the edge e got fully explored, can be determined with the *next_leaf* function: First, if v is a leaf, the edge (u, v) can be explored at most once (by Lemma 6.1). Otherwise, we choose a leaf ℓ' such that the LCA of ℓ and ℓ' is v. The idea is that *str_depth*(v) is the length of the longest common prefix of two suffixes corresponding to two leaves (e.g., ℓ and ℓ') having v as their LCA. So we can compare the m-th character of both respective suffixes by applying *next_leaf* m-times on both leaves before using the *head*-function. With $m := \mathit{str\_depth}(u) + n_v + 1$ we can check whether the edge (u, v) got fully explored. Additionally, we can determine the label of the next corresponding factor. Although we apply *next_leaf* as many times as the factor length, we still get linear time overall, because concatenating all factors yields the text T.

Pass (b). This pass is nearly identical to Pass (a). We explore the LZ trie nodes again, but this time we already have the witnesses. So we keep B_W, but reset the exploration counters and B_V.

For finding the referred indices, we create an array W with $z_W \lg z$ bits to store a factor index for each witness id. The witness ids are determined by B_W. The factor indices are given by a counter variable tracking the number of visited corresponding leaves, i.e., the number of processed factors.

Assume that we visit the leaf ℓ corresponding to the x-th factor, i.e., ℓ is the x-th visited corresponding leaf. Again by level ancestor queries, we determine the edge $e = (u, v)$ on the path from the root to ℓ, where u is in B_V and v not.

If v is an internal node, then v is a witness. In this case, we retrieve $y := W[B_W.\operatorname{rank}_1(v)]$. If y is defined, then the x-th factor refers to the y-th factor.

If y is undefined (i.e., its value has not yet been initialized), or if v is a leaf (i.e., $v = \ell$), then the x-th factor is either a fresh factor if v is a child of root, or the x-th factor refers to $W[B_W.\operatorname{rank}_1(\mathit{parent}(v))]$.

If v is an internal node, we set $W[B_w.\operatorname{rank}_1(v)] \leftarrow x$, and increment n_v (thus exploring the LZ trie like before). Like before, when v's in-going edge gets fully explored, we mark v in B_V.

So far, we can output the referred index of the x-th factor, if it exists. We get the new character of the x-th factor (i.e., the last letter of the factor) by accessing the leaf ℓ' that is (with respect to text order) *before* the leaf corresponding to the $(x + 1)$-th factor; then we can output the new character by *head*(ℓ').

Acknowledgements

We thank Veli Mäkinen for outlining us the differences between [6] and [8], and Johannes Fischer for some helpful comments. Most parts of this work was done during a visit at the graduate school of information science and technology of the university of Tokyo, supported by the *Studienwerk für Deutsch-Japanischen Kulturaustausch in NRW e.V.*

References

[1] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

[2] ——, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[3] D. Kosolobov, "Faster Lightweight Lempel-Ziv Parsing," in *MFCS*. Springer Berlin Heidelberg, 2015, vol. 9235, pp. 432–444.

[4] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Lightweight Lempel-Ziv Parsing," in *Experimental Algorithms*, 2013, pp. 139–150.
[5] D. Belazzougui and S. J. Puglisi, "Range Predecessor and Lempel-Ziv Parsing," *SODA*, pp. 2053–2071, 2016.
[6] D. Belazzougui, "Linear Time Construction of Compressed Text Indices in Compact Space," in *STOC*. ACM, 2014, pp. 148–193.
[7] M. Burrows and D. J. Wheeler, "A Block-sorting Lossless Data Compression Algorithm," Digital Equipment Corporation, Tech. Rep., 1994.
[8] D. Belazzougui, "Linear time construction of compressed text indices in compact space," *ArXiv CoRR*, vol. abs/1401.0936, 2015.
[9] J. Fischer and P. Gawrychowski, "Alphabet-Dependent String Searching with Wexponential Search Trees," in *CPM*, 2015, pp. 160–171.
[10] J. Jansson, K. Sadakane, and W. Sung, "Linked Dynamic Tries with Applications to LZ-Compression in Sublinear Time and Space," *Algorithmica*, vol. 71, no. 4, pp. 969–988, 2015.
[11] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda, "Constructing LZ78 Tries and Position Heaps in Linear Time for Large Alphabets," *Inform. Process. Lett.*, vol. 115, no. 9, pp. 655 – 659, 2015.
[12] J. Fischer, T. I, and D. Köppl, "Lempel-Ziv Computation in Small Space (LZ-CISS)," in *CPM*, 2015, pp. 172–184.
[13] J. I. Munro, "Tables," in *Proc. FSTTCS*, ser. LNCS, vol. 1180. Springer, 1996, pp. 37–42.
[14] R. Raman, V. Raman, and S. R. Satti, "Succinct Indexable Dictionaries with Applications to Encoding K-ary Trees, Prefix Sums and Multisets," *ACM Trans. Algorithms*, vol. 3, no. 4, 2007.
[15] G. Franceschini, S. Muthukrishnan, and M. Pătraşcu, "Radix Sorting with No Extra Space," in *ESA*. Springer, 2007, vol. 4698, pp. 194–205.
[16] R. Grossi and J. S. Vitter, "Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.
[17] K. Sadakane, "Compressed Suffix Trees with Full Functionality," *Theory of Computing Systems*, vol. 41, no. 4, pp. 589–607, 2007.
[18] G. Navarro and K. Sadakane, "Fully Functional Static and Dynamic Succinct Trees," *ACM Trans. Algorithms*, vol. 10, no. 3, pp. 16:1–16:39, 2014.
[19] D. Köppl and K. Sadakane, "Lempel-Ziv Computation in Compressed Space (LZ-CICS)," *ArXiv CoRR*, vol. abs/1510.02882, 2015.
[20] K. Sadakane and R. Grossi, "Squeezing succinct data structures into entropy bounds," in *Proc. SODA*. ACM/SIAM, 2006, pp. 1230–1239.