Smaller and Faster Lempel-Ziv Indices *

Diego Arroyuelo and Gonzalo Navarro

Dept. of Computer Science, Universidad de Chile, Chile.
{darroyue,gnavarro}@dcc.uchile.cl

Abstract. Given a text $T[1..u]$ over an alphabet of size $\sigma = O(\text{polylog}(u))$ and with k-th *order empirical entropy* $H_k(T)$, we propose a new *compressed full-text self-index* based on the Lempel-Ziv (LZ) compression algorithm, which replaces T with a representation requiring about three times the size of the compressed text, i.e $(3+\epsilon)uH_k(T) + o(u \log \sigma)$ bits, for any $\epsilon > 0$ and $k = o(\log_\sigma u)$, and in addition gives indexed access to T: it is able to locate the *occ* occurrences of a pattern $P[1..m]$ in the text in $O((m + occ) \log u)$ time. Our index is smaller than the existing indices that achieve this locating time complexity, and locates the occurrences faster than the smaller indices. Furthermore, our index is able to count the pattern occurrences in $O(m)$ time, and it can extract any text substring of length ℓ in optimal $O(\ell/\log_\sigma u)$ time. Overall, our indices appear as a very attractive alternative for space-efficient indexed text searching.

1 Introduction

With the huge amount of text data available nowadays, the *full-text searching* problem plays a fundamental role in modern computer applications. Full-text searching consists of finding the *occ* occurrences of a given pattern $P[1..m]$ in a text $T[1..u]$, where both P and T are sequences over an alphabet $\Sigma = \{1, 2, \ldots, \sigma\}$. Unlike *word-based* text searching, we wish to find any *text substring*, not only whole words or phrases. This has applications in texts where the concept of *word* is not well defined (e.g., Oriental languages), or texts where words do not exist at all (e.g., DNA, protein, and MIDI pitch sequences).

We assume that the text is large and known in advance to queries, and we need to perform several queries on it. Therefore, we can construct an *index* on the text, which is a data structure allowing efficient access to the pattern occurrences, yet increasing the space requirement. Our main goal is *to provide fast access to the text using as little space as possible*. Classical full-text indices, like *suffix trees* and *suffix arrays*, have the problem of a high space requirement: they require $O(u \log u)$ and $u \log u$ bits respectively, which in practice is about 10 and 4 times the text size, not including the text.

Compressed self-indexing is a recent trend in full-text searching, which consists in developing full-text indices that store enough information so as to search

* Supported in part by CONICYT PhD Fellowship Program (first author), and Fondecyt Grant 1-050493 (second author).

and retrieve any part of the indexed text without storing the text itself, while requiring space proportional to the compressed text size. Because of their compressed nature and since the text is replaced by the index, typical compressed self-indices are much smaller than classical indices, allowing one to store indices of large texts entirely in main memory, in cases where a classical index would have required to access the much slower secondary storage. There exist two classical kind of queries, namely: (*1*) $\mathsf{count}(T, P)$, which counts the number of occurrences of P in T; (*2*) $\mathsf{locate}(T, P)$, which reports the starting positions of the *occ* occurrences of P in T. Self-indices also need operation (*3*) $\mathsf{extract}(T, i, j)$, which decompresses substring $T[i..j]$, for any text positions $i \leqslant j$.

Let $H_k(T)$ denote the k-th order empirical entropy of a sequence of symbols T [12]. The value $uH_k(T)$ provides a lower bound to the number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precede it in T. It holds that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$ (by log we mean $\log_2$ in this paper).

The main types of compressed self-indices [16] are *Compressed Suffix Arrays* (CSA) [8, 19], indices based on *backward search* [6] (which are alternative ways to compress suffix arrays), and the indices based on the *Lempel-Ziv* compression algorithm (LZ-indices for short) [10]. LZ-indices have shown to be very effective in practice for locating occurrences and extracting text, outperforming other compressed indices. Compressed indices based on suffix arrays store extra non-compressible information to carry out these tasks, whereas the extra data stored by LZ-indices is compressible. Therefore, when the texts are highly compressible, LZ-indices can be smaller and faster than alternative indices; and in other cases they offer very attractive space/time trade-offs.

What characterizes the particular niche of LZ-indices is the $O(uH_k(T))$ space combined with $O(\log u)$ time per located occurrence. The smallest LZ-indices available are those by Arroyuelo et al. [1], which require $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, for any constant $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$; however, their time complexity of $O(m^2 \log m + (m + occ) \log u)$ makes them suitable only for short patterns, when the quadratic term is less significant. Other LZ-indices, like a compact version of that by Ferragina and Manzini [6], or the one by Russo and Oliveira [18], achieve $O((m + occ) \log u)$ time. They are, however, relatively large, at least $5uH_k(T) + o(u \log \sigma)$ bits of space.

In this paper we propose a new LZ-index scheme requiring $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for $\sigma = O(\text{polylog}(u))$ and any $k = o(\log_\sigma u)$, with an overall locating time of $O((m + occ) \log u)$, a counting time of $O(m)$, and an extracting time of $O(\ell / \log_\sigma u)$ for a substring of length ℓ. In this way we achieve the same locating complexity of larger LZ-indices [6, 18]. Note that the original index in [6] achieves better locate time, $O(m + occ)$, yet it requires $O(uH_k(T) \log^\gamma)$ bits of space, for any $\gamma > 0$.

The CSA of Sadakane [19] has a locating complexity of $O((m + occ) \log^\epsilon u)$, for any $\epsilon > 0$, however the space requirement is proportional to the zero-th order empirical entropy plus a non-negligible extra space, $\epsilon^{-1}uH_0(T) + O(u \log \log \sigma)$ bits. The *Alphabet-Friendly FM-index* [7], on the other hand, requires $uH_k(T) +$

$o(u \log \sigma)$ bits of space, however its locate complexity is $O(m + occ \log^{1+\epsilon} u)$, which is slower than ours. Finally, the CSA of Grossi, Gupta, and Vitter [8] requires $\epsilon^{-1} u H_k(T) + o(u \log \sigma)$ bits of space, with a locating time of $O((\log u)^{\frac{\epsilon}{1-\epsilon}} (\log \sigma)^{\frac{1-2\epsilon}{1-\epsilon}})$ per occurrence, after a counting time of $O(\frac{m}{\log_\sigma u} + (\log u)^{\frac{1+\epsilon}{1-\epsilon}} (\log \sigma)^{\frac{1-3\epsilon}{1-\epsilon}})$, where $0 < \epsilon < 1/2$. When ϵ approaches $1/3$, the space requirement is about $3uH_k(T) + o(u \log \sigma)$ bits, with a locating time of $O(\frac{m}{\log_\sigma u} + \log^2 u + occ((\log u)^{1/2} (\log \sigma)^{1/2}))$. Thus, using the same space their time per occurrence located is slightly lower, in exchange for an $O(\log^2 u)$ extra additive factor.

In Table 1 we summarize the space and time complexities of some existing compressed self-indices. Locate times in the table are per occurrence reported, after counting the pattern occurrences. In the case of our LZ-index, we have to pay extra $O(m \log u)$ time.

Table 1. Comparison of our LZ-index with alternative compressed self-indices. The result for the GGV-CSA [8] is shown for $\epsilon = 1/3$, and assuming $\sigma = O(\text{polylog}(u))$ in all cases.

Index	Space in bits
SAD-CSA [19]	$\epsilon^{-1} u H_0(T) + O(u \log \log \sigma)$
GGV-CSA [8]	$3uH_k(T) + o(u \log \sigma)$
AF-FMI [7]	$uH_k(T) + o(u \log \sigma)$
ANS-LZI [1]	$(2+\epsilon)uH_k(T) + o(u \log \sigma)$
RO-LZI [18]	$(5+\epsilon)uH_k(T) + o(u \log \sigma)$
Our LZI	$(3+\epsilon)uH_k(T) + o(u \log \sigma)$

Index	`count`	`locate` (per occurrence)	`extract`
[19]	$O(m \log u)$	$O(\log^\epsilon u)$	$O(\ell + \log^\epsilon u)$
[8]	$O(\frac{m}{\log_\sigma u} + \log^2 u)$	$O(\log^{1/2} u \log^{1/2} \sigma)$	$O(\log^{1/2} u \log^{1/2} \sigma + \ell / \log_\sigma u)$
[7]	$O(m)$	$O(\log^{1+\epsilon} u)$	$O(\ell + \log^{1+\epsilon} u)$
[1]	$O((m + occ) \log u)$	free after counting	$O(\ell / \log_\sigma u)$
[18]	$O((m + occ) \log u)$	free after counting	$O(\ell / \log_\sigma u)$
Our LZI	$O(m)$	$O((m + occ) \log u)$	$O(\ell / \log_\sigma u)$

2 Searching in Lempel-Ziv Compressed Texts

Assume that the text $T[1..u]$ has been compressed using the `LZ78` algorithm [21] into $n+1$ phrases, $T = B_0 \dots B_n$, such that $B_0 = \varepsilon$. The search of a pattern $P[1..m]$ in a `LZ78`-compressed text has the additional problem that, as the text is parsed into phrases, a pattern occurrence can span several (two or more) consecutive phrases. We call *occurrences of type 1* those occurrences contained in a single phrase (there are occ_1 occurrences of type 1), and *occurrences of type*

2 those occurrences spanning two or more phrases (there are occ_2 occurrences of this type). Next we review the existing Lempel-Ziv self-indices. The first compressed index based on `LZ78` was that of Kärkkäinen and Ukkonen [10], which has a locating time $O(m^2 + (m + occ) \log u)$ and a space requirement of $O(uH_k(T))$ bits [16], plus the text. However, this is not a self-index.

Ferragina and Manzini [6] define the FM-LZI, a compressed self index based on the `LZ78` compression algorithm, requiring $O(uH_k(T) \log^\gamma u)$ bits of space, for any constant $\gamma > 0$. This index is able to report the *occ* pattern occurrences in optimal $O(m + occ)$ time. This is the *fastest* existing compressed self-index, achieving the same time complexity as suffix trees, yet requiring $o(u \log u)$ bits and without needing the text to operate. However, the extra $O(\log^\gamma u)$ factor makes this index large in practice. However, we can replace their data structure for range queries by that of Chazelle [4], such that the resulting version of FM-LZI requires $(5+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any constant $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$, and is capable of locating the pattern occurrences in $O((m + occ) \log u)$ time.

The LZ-index of Navarro [15] (Nav-LZI for short) has a greater locate time than that of LZ-indices in general, yet the smallest existing LZ-index is a variant of the Nav-LZI: the index defined by Arroyuelo et al. [1] (ANS-LZI for short) requires $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, and its locate time is $O(m^2 \log m + (m + occ) \log u)$. Although the locating time per occurrence is $O(\log u)$ as for other LZ-indices, the $O(m^2 \log m)$ term makes the ANS-LZI attractive only for short patterns.

The key to achieve such a small space requirement is that the Nav-LZI (and therefore the ANS-LZI) do not use the concept of suffix arrays at all. Rather, the search is based only in an implicit representation of the text through the `LZ78` parsing of it: the *LZTrie*, which is the trie representing the `LZ78` phrases of T. As the text is scattered throughout the *LZTrie*, we have to distinguish between occurrences spanning two consecutive phrases (occurrences of type 2) and occurrences spanning more than two phrases (*occurrences of type 3*). For occurrences of type 3 we must consider the $O(m^2)$ possible substrings of P, search for all these strings in *LZTrie*, then form maximal concatenations of consecutive phrases, to finally check every candidate [15]. All of this takes $O(m^2 \log m)$ time.

The space of the ANS-LZI can be reduced to $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits, with a locating time of $O(m^2)$ on average for patterns of length $m \geqslant 2 \log_\sigma u$, yet without worst-case guarantees at search time.

Russo and Oliveira [18] discard the `LZ78` parsing of T and use a so-called maximal parsing instead, which is constructed for the reversed text. In this way they avoid the $O(m^2)$ checks for the different pattern substrings. The resulting LZ-index (the RO-LZI) requires $(5+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any constant $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The locating time of the index is $O((m + occ) \log u)$. As for all the previous LZ-indices, the extract time for any text substring of length ℓ is the optimal $O(\ell / \log_\sigma u)$.

3 Smaller and Faster Lempel-Ziv Indices

In the review of Section 2 we conclude that LZ-indices can be as small as to require $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space yet with a locate time $O(m^2\log m+(m+occ)\log u)$, or we can achieve time $O((m+occ)\log u)$ for locate with a greater index requiring $(5+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space. On the other hand, we can be fast to count only using the FM-LZI: $O(m)$ counting time in the worst case. We show that we can be fast for all these operations with a significantly smaller LZ-index.

3.1 Index Definition

As we aim at a small index, we use the ANS-LZI as a base, specifically the version requiring $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, which is composed of:

- *LZTrie*: is the trie storing the LZ78 phrases $B_0,\ldots,B_n$ of T. As the set of LZ78 phrases is prefix-closed, this trie has exactly $n+1$ nodes. We represent the trie structure using the following data structures: *(1)* $par[0..2n]$: the tree shape of *LZTrie* represented using DFUDS [2], requiring $2n+o(n)$ bits of storage, allowing us to compute operations $parent_{lz}(x)$ (which gets the parent of node x), $child_{lz}(x,i)$ (which gets the i-th child of x), $subtreesize_{lz}(x)$ (which gets the size of the subtree of x, including x itself), $depth_{lz}(x)$ (which gets the depth of x in the trie) and $LA_{lz}(x,d)$ (a *level-ancestor query*, which gets the ancestor at depth d of node x), both of which can be computed on DFUDS by using the idea of Jansson et al. [9], and finally $ancestor_{lz}(x,y)$ (which tells us whether x is an ancestor of node y), all of them in $O(1)$ time. As in [1], we add the data structure of [20] to extract any text substring of length ℓ in optimal $O(\ell/\log_\sigma u)$ time, requiring only $o(u\log\sigma)$ extra bits. *(2)* $ids[0..n]$: is the preorder sequence of LZ78 phrase identifiers. Permutation *ids* is represented using the data structure of Munro et al. [14] such that we can compute *ids* in $O(1)$ time and its inverse permutation ids^{-1} in $O(1/\epsilon)$ time, requiring $(1+\epsilon)n\log n$ bits for any constant $0<\epsilon<1$. *(3)* $letts[1..n]$: the array storing the edge labels of *LZTrie* according to a DFUDS traversal of the trie. We solve operation $child(x,\alpha)$ (which gets the child of node x with label $\alpha\in\{1,\ldots,\sigma\}$) in constant time as follows (this is slightly different to the original approach [2]). Suppose that node x has position p within *par*. Let k be the number of αs up to position $p-1$ in *letts*, and let $p+i$ be the position of the $(k+1)$-th α in *letts*. If $p+i$ lies within positions p and $p+degree(x)$, the child we are looking for is $child(x,i+1)$, which is computed in constant time over *par*; otherwise x has no child labeled α. If $\sigma=O(\text{polylog}(u))$, we represent *letts* using the *wavelet tree* of [7] in order to compute k and $p+i$ in constant time by using $rank_\alpha$ and $select_\alpha$ respectively, and requiring $n\log\sigma+o(n)$ bits of space. We can also retrieve the symbol corresponding to node x (i.e., the symbol by which x descend from its parent) in constant time by $letts[rank_\iota(par,p)-1]$. Sequence *letts* is also used to get the symbols of the text for extract queries.

Overall, *LZTrie* requires $(1+\epsilon)n\log n + 2n + n\log\sigma + o(u\log\sigma)$ bits, which is $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits [11], for any $k = o(\log_\sigma u)$.

- *RevTrie*: is the *PATRICIA* tree [13] of the reversed LZ78 phrases of T. In this trie there could be internal nodes not representing any phrase. We call these nodes *empty*. We compress empty unary nodes, and so we only represent the empty non-unary nodes. As a result, the number of nodes in this trie is $n \leqslant n' \leqslant 2n$. The trie is represented using the DFUDS representation, requiring $2n' + n'\log\sigma + o(n') \leqslant 4n + 2n\log\sigma + o(n)$ bits of space.
- $R[1..n]$: a mapping from *RevTrie* preorder positions (for non-empty nodes) to *LZTrie* preorder positions, requiring $n\log n = uH_k(T) + o(u\log\sigma)$ bits.
- $TPos[1..u]$: a bit vector marking the n phrase beginnings. We represent $TPos$ using the data structure of [17] for *rank* and *select* queries in $O(1)$ time and requiring $uH_0(TPos) + o(u) \leqslant n\log\log n + o(u) = o(u\log\sigma)$ bits of space[1].

We can compress the R mapping [1], so as to require $o(u\log\sigma)$ bits, by adding *suffix links* to *RevTrie*, which are represented by function φ. $R(i)$ (seen as a function) can be computed in constant time by using φ [1]. If we store the values of φ in preorder (according to *RevTrie*), the resulting sequence can be divided into at most σ strictly increasing subsequences, and hence it can be compressed using the δ-code of Elias [5] such that its space requirement is $n\log\sigma$ bits in the worst case, which is $o(u\log\sigma)$. The overall space requirement of the three above data structures is $(1+\epsilon)uH_k + o(u\log\sigma)$ bits.

To avoid the $O(m^2\log m)$ term in the locating complexity, we must avoid occurrences of type 3 (which make the ANS-LZI slower). Hence we add the *alphabet friendly FM-index* [7] of T (AF-FMI(T) for short) to our index. By itself this self-index is able to search for pattern occurrences, requiring $uH_k(T) + o(u\log\sigma)$ bits of space. However, its locate time per occurrence is $O(\log^{1+\epsilon} u)$, for any constant $\epsilon > 0$, which is greater than the $O(\log u)$ time per occurrence of LZ-indices. As AF-FMI(T) is based on the *Burrows-Wheeler transform* (BWT) [3] of T, it can be (conceptually) thought as the suffix array of T.

To find occurrences spanning several phrases we define *Range*, a data structure for 2-dimensional range searching in the grid $[1..u] \times [1..n]$. For each LZ78 phrase with identifier id, for $0 < id \leqslant n$, assume that the *RevTrie* node for id has preorder j, and that phrase $(id+1)$ starts at position p in T. Then we store the point (i, j) in *Range*, where i is the lexicographic order of the suffix of T starting at position p.

Suppose that we search for a given string s_2 in AF-FMI(T) and get the interval $[i_1, i_2]$ in the BWT (equivalently, in the suffix array of T), and that the search for string s_1^r in *RevTrie* yields a node such that the preorder interval for its subtree is $[j_1, j_2]$. Then, a search for $[i_1, i_2] \times [j_1, j_2]$ in *Range* yields all phrases ending with s_1 such that in the next phrase starts an occurrence of s_2.

We transform the grid $[1..u] \times [1..n]$ indexed by *Range* to the equivalent grid $[1..n] \times [1..n]$ by defining a bit vector $\mathcal{V}[1..u]$, indicating (with a 1) which positions of AF-FMI(T) index an LZ78 phrase beginning. We represent $\mathcal{V}$ with the data

[1] $rank_1(TPos, i)$ is the number of 1's in $TPos$ up to position i. $select_1(TPos, j)$ yields the position of the j-th 1 in $TPos$.

structure of [17] allowing *rank* queries, and requiring $uH_0(\mathcal{V}) + o(u) = o(u \log \sigma)$ bits of storage. Thus, instead of storing the point (i, j) as in the previous definition of *Range*, we store the point $(rank_1(\mathcal{V}, i), j)$. The same search of the previous paragraph now becomes $[rank_1(\mathcal{V}, i_1), rank_1(\mathcal{V}, i_2)] \times [j_1, j_2]$.

As there is only one point per row and column of *Range*, we can use the data structure of Chazelle [4], requiring $n \log n(1 + o(1)) = uH_k(T) + o(u \log \sigma)$ bits of space and allowing us to find the K points in a given two-dimensional range in time $O((K + 1) \log n)$. As a result, the overall space requirement of our LZ-index is $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$, for any $k = o(\log_\sigma u)$ and any constant $0 < \epsilon < 1$.

3.2 Search Algorithm

Assume that $P[1..m] = p_1 \dots p_m$, for $p_i \in \Sigma$. We need to consider two types of occurrences of P in T.

Locating Occurrences of Type 1. Assume that phrase B_j contains P. If B_j does not end with P and if $B_j = B_\ell \cdot c$, for $\ell < j$ and $c \in \Sigma$, then by `LZ78` properties B_ℓ contains P as well. Therefore we must find the shortest possible phrases containing P, which according to `LZ78` are all phrases ending with P. This work can be done by searching for P^r in *RevTrie*. Say we arrive at node v. Any node v' in the subtree of v (including v itself) corresponds to a phrase terminated with P. Thus we traverse and report all the subtrees of the *LZTrie* nodes $R(v')$ corresponding to each v'. Total locate time is $O(m + occ_1)$.

Locating Occurrences of Type 2. To find the pattern occurrences spanning two or more consecutive phrases we must consider the $m - 1$ partitions $P[1..i]$ and $P[i + 1..m]$ of P, for $1 \leqslant i < m$. For every partition we must find all phrases terminated with $P[1..i]$ such that the next phrase starts at the same position as an occurrence of $P[i + 1..m]$ in T. Hence, as explained before, we must search for $P^r[1..i]$ in *RevTrie* and for $P[i + 1..m]$ in AF-FMI(T). Thus, every partition produces two one-dimensional intervals, one in each of the above structures.

The $m - 1$ intervals in AF-FMI(T) can be found in $O(m)$ time thanks to the *backward search* concept, since the process to count the number of occurrences of $P[2..m]$ proceeds in $m - 1$ steps, each one taking constant time: in the first step we find the BWT interval for p_m, then we find the interval for occurrences of $p_{m-1}p_m$, and so on to finally find the interval for $p_2 \dots p_m = P[2..m]$. However, the work in *RevTrie* can take time $O(m^2)$ if we search for strings $P^r[1..i]$ separately, as done for the ANS-LZI. Fortunately, some work done to search for a given $P^r[1..i]$ can be reused to search for other strings.

We have to search for strings $p_{m-1}p_{m-2} \dots p_1$; $p_{m-2} \dots p_1$;...; and p_1 in *RevTrie*. Note that every $p_j \dots p_1$ is the longest proper suffix of $p_{j+1}p_j \dots p_1$. Suppose that we successfully search for $P^r[1..m-1] = p_{m-1}p_{m-2} \dots p_1$, reaching the node with preorder i in *RevTrie*, hence finding the corresponding preorder interval in *RevTrie* in $O(m)$ time. Now, to find the node representing suffix $p_{m-2} \dots p_1$ we only need to follow suffix link $\varphi(i)$ (which takes constant time) instead of searching for it from the *RevTrie* root (which would take $O(m)$ time

again). The process of following suffix links can be repeated $m-1$ times up to reaching the node corresponding to string p_1, with total time $O(m)$. This is the main idea to get the $m-1$ preorder intervals in *RevTrie* in time less than quadratic. The general case is slightly more complicated and corresponds to the *descend and suffix walk* method used in the RO-LZI [18]. In the sequel we explain the way we implement descend and suffix walk in our data structure.

We first prove a couple of properties. First, we know that every non-empty node in *RevTrie* has a suffix link [1], yet we need to prove that every *RevTrie* node (including empty-non-unary nodes) has also a suffix link.

Property 1. Every empty non-unary node in *RevTrie* has a suffix link.

Proof. Assume that node with preorder i in *RevTrie* is empty non-unary, and that it represents string $\mathtt{ax}$, for $\mathtt{a} \in \Sigma$ and $\mathtt{x} \in \Sigma^*$. As node i is an empty non-unary node, the node has at least two children. In other words, there exist at least two strings of the form $\mathtt{axy}$ and $\mathtt{axz}$, for $\mathtt{y}, \mathtt{z} \in \Sigma^*$, $\mathtt{y} \neq \mathtt{z}$, both strings corresponding to non-empty nodes, and hence these nodes have a suffix link. These suffix links correspond to strings $\mathtt{xy}$ and $\mathtt{xz}$ in *RevTrie*. Thus, it must exist a non-unary node for string $\mathtt{x}$, so every empty node i has a suffix link. □

We store the $n' \leqslant 2n$ suffix links φ in preorder (as explained before), requiring $2n \log \sigma$ bits of space in the worst case, which is $o(u \log \sigma)$.

The second property is that, although *RevTrie* is a PATRICIA tree and hence we store only the first symbol of each edge label, we can get all of it.

Property 2. Any edge label in *RevTrie* can be extracted in optimal time.

Proof. To extract the label for edge e_{ij} between nodes with preorder i and j in *RevTrie*, note that the length of the edge label can be computed as $depth_{lz}(R[j]) - depth_{lz}(R[i])$. We can access the node from where to start the extraction by $x = LA_{lz}(R[j], depth_{lz}(R[j]) - depth_{lz}(R[i]))$, in constant time. The label of e_{ij} is the label of the root-to-x path (read backwards). □

In this way, every time we arrive to a *RevTrie* node, the string represented by that node will match the corresponding prefix of the pattern.

Previously we show that it is possible to search for all strings $P^r[1..i]$ in time $O(m)$, assuming that $P^r[1..m-1]$ exists in *RevTrie* (therefore all $P^r[1..i]$ exist in *RevTrie*). The general case is as follows. Suppose that, searching for $p_{m-1}p_{m-2}\dots p_1$, we arrive at a node with preorder i in *RevTrie* (hereafter node i), and we try to descend to a child node with preorder j (hereafter node j). Assume that node i represents string $\mathtt{ax}$, for $\mathtt{a} \in \Sigma$ and $\mathtt{x} \in \Sigma^*$. According to Property 2, we are sure that $\mathtt{ax} = p_{m-1}\dots p_t$, for some $1 \leqslant t \leqslant m-1$. Assume also that edge e_{ij} between nodes i and j is labeled $\mathtt{yz}$, for $\mathtt{y}, \mathtt{z} \in \Sigma^*$, $\mathtt{z} = z_1 \dots z_q$. If we discover that $p_{t-1}\dots p_k = \mathtt{y}$ and $p_{k-1} \neq z_1$, then this means that symbol z_1 in the edge label differs from the corresponding symbol p_{k-1} in $P^r[1..m-1]$, and so we cannot descend to node j. This means that there are no phrases ending with $P^r[1..m-1]$, and we go on to consider $P^r[1..m-2]$. To

reuse the work done up to node i, we follow the suffix link to get the node $\varphi(i)$, and from this node we descend using $\mathtt{y} = p_{t-1} \dots p_k$. As this substring of P has been already checked in the previous step, the descent is done checking only the first symbols of the labels, up to a node such that the next node in the path represents a string longer than $|\mathtt{xy}|$. At this point the descent is done as usual, extracting the edge labels and checking with the pattern symbols. In this way the total amortized time is $O(m)$.

If the search in *RevTrie* for $P^r[1..i]$ yields the preorder interval $[x, y]$, and the search for $P[i+1..m]$ in AF-FMI(T) yields interval $[x', y']$, the two-dimensional range $[x', y'] \times [x, y]$ in *Range* yields all pattern occurrences for the given partition of P. For every pattern occurrence we get a point (i', j'). The corresponding phrase identifier can be found as $id = ids(R(j'))$, to finally compute the text position by $select_1(TPos, id+1) - i$. Overall, occurrences of type 2 are found in $O((m + occ_2) \log n)$ time.

For count queries we can achieve $O(m)$ time by just using the AF-FMI(T). For extract queries we use the data structure of Sadakane and Grossi [20] in the *LZTrie* to extract any text substring $T[p..p + \ell]$ in optimal $O(\ell / \log_\sigma u)$ time: the identifier for the phrase containing position p can be computed as $id = rank_1(TPos, p)$. Then, by using ids^{-1} we compute the corresponding *LZTrie* node from where to extract the text.

Theorem 1. *There exists a compressed full-text self-index requiring* $(3+\epsilon)uH_k(T) + o(u \log \sigma)$ *bits of space, for* $\sigma = O(\text{polylog}(u))$*, any* $k = o(\log_\sigma u)$*, and any constant* $0 < \epsilon < 1$*, which is able to: report the occ occurrences of pattern* $P[1..m]$ *in text* $T[1..u]$ *in* $O((m + occ/\epsilon) \log u)$ *worst-case time; count pattern occurrences in* $O(m)$ *time; and extract any text substring of length* ℓ *in time* $O(\ell/(\epsilon \log_\sigma u))$.

References

1. D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. CPM*, pages 319–330, 2006.
2. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S.S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
3. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
4. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
5. P. Elias. Universal codeword sets and representation of integers. *IEEE Trans. Inform. Theory*, 21(2):194–203, 1975.
6. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 54(4):552–581, 2005.
7. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 2007.
8. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

9. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. SODA'07*, pages 575–584, 2007.
10. J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. WSP*, pages 141–155, 1996.
11. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
12. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
13. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
14. I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *Proc. ICALP*, LNCS 2719, pages 345–356, 2003.
15. G. Navarro. Indexing text using the Ziv-Lempel trie. *J. Discrete Algorithms*, 2(1):87–114, 2004.
16. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
17. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.
18. L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. In *Proc. SPIRE*, LNCS 4209, pages 163–180, 2006.
19. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
20. K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. SODA*, pages 1230–1239, 2006.
21. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.