Stronger Lempel-Ziv Based Compressed Text Indexing[⋆]

Diego Arroyuelo[1] [⋆⋆], Gonzalo Navarro[2] [⋆⋆⋆], and Kunihiko Sadakane[3] [†]

[1] Yahoo! Research Latin America, Chile.
Blanco Encalada 2120, Santiago, Chile.
darroyue@dcc.uchile.cl

[2] Dept. of Computer Science, Universidad de Chile,
Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

[3] Dept. of Computer Science and Communication Engineering, Kyushu University,
Hakozaki 6-10-1, Higashi-ku, Fukuoka 812-8581, Japan.
sada@csce.kyushu-u.ac.jp

Abstract. Given a text $T[1..u]$ over an alphabet of size σ, the *full-text search* problem consists in finding the *occ* occurrences of a given pattern $P[1..m]$ in T. In *indexed* text searching we build an index on T to improve the search time, yet increasing the space requirement. The current trend in indexed text searching is that of *compressed full-text self-indices*, which replace the text with a more space-efficient representation of it, at the same time providing indexed access to the text. Thus, we can provide efficient access within compressed space.

The Lempel-Ziv index (LZ-index) of Navarro is a compressed full-text self-index able to represent T using $4uH_k(T) + o(u \log \sigma)$ bits of space, where $H_k(T)$ denotes the k-th order empirical entropy of T, for any $k = o(\log_\sigma u)$. This space is about four times the compressed text size. The index can locate all the *occ* occurrences of a pattern P in T in $O(m^3 \log \sigma + (m + occ) \log u)$ worst-case time. Although this index has proven very competitive in practice, the $O(m^3 \log \sigma)$ term can be excessive for long patterns. Also, the factor 4 in its space complexity makes it larger than other state-of-the-art alternatives.

In this paper we present stronger Lempel-Ziv based indices (LZ-indices), improving the overall performance of the original LZ-index. We achieve indices requiring $(2+\epsilon)uH_k(T)+o(u \log \sigma)$ bits of space, for any constant $\epsilon > 0$, which makes them the smallest existing LZ-indices. We simultaneously improve the search time to $O(m^2 + (m + occ) \log u)$, which makes our indices very competitive with state-of-the-art alternatives. Our indices support displaying any text substring of length ℓ in optimal $O(\ell/\log_\sigma u)$ time. In addition, we show how the space can be squeezed to $(1+\epsilon)uH_k(T)+o(u \log \sigma)$ to obtain a structure with $O(m^2)$ average search time for $m \geqslant 2\log_\sigma u$. Alternatively, the search time of LZ-indices can be improved to $O((m + occ) \log u)$ with $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, which is much less than the space needed by other Lempel-Ziv-based indices achieving the same search time. Overall our indices stand out as a very attractive alternative for space-efficient indexed text searching.

1 Introduction

Text searching is a classic problem in Computer Science. Given a sequence of symbols $T[1..u]$ (the text) over an alphabet $\Sigma = \{1, \ldots, \sigma\}$, and given another (short) sequence $P[1..m]$ (the *search pattern*) over Σ, the *full-text search problem* consists in finding all the *occ* occurrences of P in T. There exist three typical kinds of queries, namely:

- *Existential queries*: operation `exists`(P) tells us whether pattern P occurs in T or not.

[⋆] A preliminary version of this paper appeared in *Proc. of the 17th Annual Symposium on Combinatorial Pattern Matching* (CPM 2006), Volume 4009 of LNCS, 2006.

[⋆⋆] Supported by CONICYT PhD Fellowship Program.

[⋆⋆⋆] Supported in part by Fondecyt Grant 1-080019.

[†] Supported in part by the Grant-in-Aid of the Ministry of Education, Science, Sports and Culture of Japan.

– *Cardinality queries*: operation `count`(P) counts the number of occurrences of pattern P in T.
– *Locating queries*: operation `locate`(P) reports the starting positions of the *occ* occurrences of pattern P in T.

With the huge amount of text data available nowadays, the full-text search problem plays a fundamental role in modern computer applications, which include text databases in general. 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, program code, etc.). We assume that the text is large and known in advance to queries, and we need to perform many 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.

Classical full-text indices, like *suffix trees* [1] and *suffix arrays* [33], 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 10–20 and 4 times the text size respectively, apart from the text itself. Thus, we can have large texts which fit into main memory, but whose corresponding suffix tree (or array) does not. Using secondary storage for the indices is several orders of magnitude slower, so one looks for ways to reduce their size, with the main motivation of maintaining the indices of very large texts entirely in main memory. Therefore, we seek *to provide fast access to the text using as little space as possible.* The modern trend is to use the compressibility of the text to reduce the space of the index. In recent years there has been much research on *compressed text databases*, focusing on techniques to represent the text and the index using little space, yet permitting efficient text searching [41].

1.1 Compressed Full-Text Self-Indexing

To provide fast access to the text using little space, a current trend is to use *compressed full-text self-indices.*

Definition 1. *A* compressed full-text index *is one whose space requirement is proportional to the compressed text size under some compression model (e.g., $O(uH_k(T))$ bits of space*[4]*).*

Therefore, the space of such an index can be reduced when the text is compressible. This track was started by Kärkkäinen and Ukkonen [27, 26], who studied text indices based on repetitions, and defined the first indices based on the Lempel-Ziv compression algorithms [29]. Later, Grossi and Vitter [22] defined the *Compressed Suffix Arrays*, based on regularities of suffix arrays to reduce their space. However, and just like classical indices, all these indices still need the text to operate. An important space saving can be achieved if we lift this restriction.

Definition 2. *A full-text* self-index *allows one to search and extract any part of the text without storing the text itself.*

A compressed full-text self-index *replaces* the text with a more space-efficient representation of it (profiting from text compressibility to obtain smaller indices), at the same time providing indexed access to the text [41, 16]. Taking space proportional to the compressed text, replacing it, and

[4] $uH_k(T)$, the k-th order empirical entropy of T, is a lower bound to the number of bits used to represent T by any k-th order compressor. See Section 2.2 for more details.

providing efficient indexed access to it is an unprecedented breakthrough in text indexing and compression.

Ferragina and Manzini [14–16], and Sadakane [45] defined the first self-indices. The former is a representation of Compressed Suffix Arrays [22] which does not need the text to operate. The index of Ferragina and Manzini, on the other hand, is based on the close relation between suffix arrays and the *Burrows-Wheeler* transform [9]. Later, many other compressed self-indices were defined, such as the ones by Grossi et al. [21], Navarro [39, 40], Mäkinen and Navarro [30], Ferragina et al. [17], and Russo and Oliveira [44], among others. An important survey about compressed self-indices is given by Navarro and Mäkinen [41].

Compressed full-text self-indices are not only useful to reduce the space requirement of text indices: they also have applications in cases where accessing the text is so expensive that the index must search without having the text at hand, as occurs with most Web search engines.

Extending the Set of Operations. As compressed full-text self-indices replace the text, we are also interested in operations:

- *Displaying contexts around occurrences*: operation `display`(P, ℓ) shows us a context of ℓ symbols surrounding the *occ* occurrences of pattern P in T.
- *Decompressing parts of the text*: operation `extract`(i, j) decompresses the substring $T[i..j]$, for any text positions $i \leqslant j$.

Thus we can see compressed self-indices as full-text indices compressing the text, or as compressors allowing efficient text extraction and indexed full-text searching. For compressed full-text self-indices, which *replace the text*, being able to efficiently `extract` arbitrary text substrings is one of the most basic and important problems that indices must solve efficiently.

Families of Compressed Self-Indices. The main types of compressed self-indices [41] are:

- *Compressed Suffix Arrays* [22] (CSAs for short), as for instance Sadakane's CSA (Sad-CSA) [45], and Grossi et al.'s CSA (GGV-CSA) [21];
- indices based on *backward search* [16] (which are alternative ways to compress suffix arrays, known as the *FM-index* family), as for instance the *Alphabet-Friendly FM-index* (AF-FMI) [17]; and
- indices based on *Lempel-Ziv* compression [29, 48] (LZ-indices for short), as for instance Kärkkäinen and Ukkonen's LZ-index [27], Ferragina and Manzini's LZ-index [16], Navarro's LZ-index (Nav-LZI) [39], and Russo and Oliveira's LZ-index (ILZI) [44].

In Table 1 we show the most efficient existing compressed self-indices, where the different families are separated by horizontal lines. We are particularly interested in LZ-indices, since they have been proven to be very effective in practice for locating occurrences and extracting text [40, 12], outperforming other compressed indices. Also, when the texts are highly compressible, LZ-indices can in practice be smaller and faster than alternative indices, and in other cases they offer very attractive space/time trade-offs [4]. What characterizes the particular niche of LZ-indices is the $O(uH_k(T))$ space combined with $O(\log u)$ time per located occurrence.

Previous Work on LZ-Indices. Historically, the first compressed index based on Lempel-Ziv compression was that of Kärkkäinen and Ukkonen [27, 26] (which is based on a specific version of the Lempel-Ziv parsing algorithm of 1976 [29]). It has a locating time of $O(m^2+(m+occ)\log u)$ and a space requirement of $O(uH_k(T))$ bits, plus the text (as it is needed to operate) [41]. Navarro's LZ-index [39, 40], on the other hand, is a compressed full-text self-index based on the Lempel-Ziv 1978 [48] (LZ78 for short) parsing of the text. See Section 2.3 for a description of the LZ78 compression algorithm. The LZ-index takes about 4 times the size of the compressed text, that is, $4uH_k(T)+o(u\log\sigma)$ bits, for any $k=o(\log_\sigma u)$ [28, 16], and supports `locate` queries in $O(m^3\log\sigma+(m+occ)\log u)$ worst-case time. The index can display a text context of length ℓ around an occurrence found (and in fact any sequence of LZ78 phrases) in $O(\ell\log\sigma)$ time, or obtain the whole text in time $O(u\log\sigma)$. The index is built in $O(u\log\sigma)$ time.

Despite this index having been proven to be very competitive in practice [39, 40], the $O(m^3\log\sigma)$ term in the search time makes it appropriate only for short patterns. Besides, in practice the space requirement of the LZ-index is relatively large compared with competing schemes: 1.2–1.6 times the text size (depending on the compressibility of the text) versus 0.6–0.7 and 0.3–0.8 times the text size of the CSA [45] and the *FM-index* [16], respectively. Yet, the LZ-index is faster to locate and to display the context of an occurrence, which as explained is very important for self-indices.

So the challenge is: can we reduce the space requirement of the LZ-index while retaining its good features? Previous work [4] studies the reduction of the space requirement of LZ-index from a practical approach. The result is an LZ-index requiring $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, and with $O(\frac{m^2}{\epsilon})$ time on average for `locate` queries. However, the space of LZ-index cannot be further reduced by using that approach.

1.2 Our Contribution

In this paper we go one step further on the results of previous work [4], not only reducing by half the space requirement of the LZ-index, but also improving its time complexities. The result is an attractive alternative to the state of the art in compressed self-indexing.

First, in Section 4, we compress one of the data structures composing the original LZ-index by using an approach which is, in some sense, related to the compression of suffix arrays [22, 45]. Second, in Section 5, we combine the balanced parentheses representation of Munro and Raman [38] of the LZ78 trie with the *xbw transform* of Ferragina et al. [13], whose powerful operations are useful for the LZ-index search algorithm.

Although these approaches are very different, when $\sigma=\Theta(\text{polylog}(u))$ (that is, on moderate-size alphabets, which are very common in practice) we achieve in both cases $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any constant $\epsilon>0$, and simultaneously *improve* the original worst-case search time to $O(m^2+(m+occ)\log u)$. Thus we achieve the same search time as the index of Kärkkäinen and Ukkonen [27], yet ours are much smaller and do not need the text to operate. In both cases we also present a version requiring $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ bits, with average search time $O(m^2)$ if $m\geqslant 2\log_\sigma u$. This space can get as close as desired to the optimal $uH_k(T)$ under the k-th order entropy model. The worst-case time for extracting any text substring of length ℓ is also improved to the optimal $O(\ell/\log_\sigma u)$ for all of our indices.

Just as for the original LZ-index, our data structures require $O(uH_k(T))$ bits and spend $O(\log u)$ time per occurrence reported, if $\sigma=\Theta(\text{polylog}(u))$. This fast locating is the strongest point of our structure. Other data structures achieving the same or better complexity for locating occurrences either are of size $O(uH_0(T))$ bits plus a non-negligible extra space of $O(u\log\log\sigma)$ [45], or they achieve this locating time for constant-size alphabets [16]. Finally, the GGV-CSA [21] requires $\epsilon^{-1}uH_k(T)+o(u\log\sigma)$ 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 $0<\epsilon<1/2$ is a constant. When ϵ approaches 1/2, the space requirement approaches (from above) $2uH_k(T)+o(u\log\sigma)$ bits, with a counting time of $O(\frac{m}{\log_\sigma u}+\frac{\log^3 u}{\log\sigma})$ and still a locating time per occurrence of $\omega(\log u)$.

In Table 1 we summarize the space and time complexities of some existing compressed self-indices (other less competitive ones are ignored [41]). Total locate times in the table require counting the pattern occurrences first. For counting the number of occurrences of P in T, our data structures are not competitive with schemes requiring about the same space [21, 17]. Yet, in many practical situations, it is necessary to report the occurrence positions, as well as displaying their contexts and extracting (or uncompressing) any text substring. In this aspect, as explained, our LZ-indices are superior.

A new LZ-index, the *Inverted LZ-index* (ILZI for short) [44], has appeared independently and simultaneously with our work [6]. The ILZI is faster than our data structures since it can report the pattern occurrences in $O((m+occ)\log u)$ time, but at the price of a higher space requirement: $(5+\epsilon)uH_k(T)+o(u\log\sigma)$ bits. However, in this paper we also show that the same reporting time $O((m+occ)\log u)$ can be obtained with a significantly smaller LZ-index requiring $(3+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space. In practice, our LZ-indices and the ILZI are comparable.

Table 1. Comparison of our LZ-index with alternative compressed self-indices. We assume $\sigma = O(\text{polylog}(u))$ in all cases. Note our change of variable in GGV-CSA to make it easier to compare. All indices must count before locating.

Index	Space in bits
SAD-CSA [45]	$(1+\epsilon)uH_0(T)+O(u\log\log\sigma)$
GGV-CSA [21]	$(2+\epsilon)uH_k(T)+o(u\log\sigma)$
AF-FMI [17]	$uH_k(T)+o(u\log\sigma)$
NAV-LZI [39]	$4uH_k(T)+o(u\log\sigma)$
ILZI [44]	$(5+\epsilon)uH_k(T)+o(u\log\sigma)$
Our LZ-index	$(2+\epsilon)uH_k(T)+o(u\log\sigma)$
Our larger LZ-index	$(3+\epsilon)uH_k(T)+o(u\log\sigma)$

Index	`count`	`locate`	`extract`
SAD-CSA	$O(m\log u)$	$O(occ\log^{\frac{1}{1+\epsilon}} u)$	$O(\ell+\log^\epsilon u)$
GGV-CSA	$O(\frac{m}{\log_\sigma u}+\frac{\log^{\frac{3+\epsilon}{1+\epsilon}} u}{\log^{\frac{1-\epsilon}{1+\epsilon}} \sigma})$	$O(occ\log u\log_\sigma^\epsilon u)$	$O(\ell/\log_\sigma u+\log u\log_\sigma^\epsilon u)$
AF-FMI	$O(m)$	$O(occ\log^{1+\epsilon} u)$	$O(\ell+\log^{1+\epsilon} u)$
NAV-LZI	$O(m^3\log\sigma+m\log u+occ)$	$O(occ\log u)$	$O(\ell\log\sigma)$
ILZI	$O(m\log u+occ)$	$O(occ\log u)$	$O(\ell/\log_\sigma u)$
Our LZ-index	$O(m^2+m\log u+occ)$	$O(occ\log u)$	$O(\ell/\log_\sigma u)$
Our larger LZ-index	$O(m)$	$O((m+occ)\log u)$	$O(\ell/\log_\sigma u)$

2 Basic Concepts

2.1 Model of Computation

In this paper we assume the standard *word* RAM model of computation, in which we can access any memory word of length w, such that $w = \Theta(\log u)$, in constant time[5]. Standard arithmetic and logical operations (like additions, bit-wise operations, etc.) are assumed to take constant time in this model. We measure the size of our data structures in bits.

2.2 Empirical Entropy

A concept related to text compression is that of the k-th order empirical entropy of a sequence, T, of symbols over an alphabet of size σ, denoted by $H_k(T)$ [34]. 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.

Formally, the zero-order empirical entropy of T is defined as $H_0(T) = \sum_{c\in\Sigma} \frac{u_c}{u} \log \frac{u}{u_c}$, where u_c is the number of occurrences of symbol c in T. The sum includes only those symbols c that occur in T, so that $u_c > 0$. The k-th order empirical entropy of T is defined as $H_k(T) = \sum_{s\in\Sigma^k} \frac{|T^s|}{u} H_0(T^s)$, where T^s is the subsequence of T formed by all the symbols that occur preceded by the context s. Again, we consider only contexts s that do occur in T.

An important property is that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log\sigma$, for any $k \geqslant 0$. This means that by considering longer contexts we can get more compression.

2.3 Lempel-Ziv Compression

The Lempel-Ziv compression algorithm of 1978 (usually named LZ78 [48]) is based on a *dictionary of phrases*, in which we add every new *phrase* computed. At the beginning of the compression, the dictionary contains a single phrase b_0 of length 0 (i.e., the empty string). The current step of the compression is as follows: If we assume that a prefix $T[1..j]$ of T has been already compressed into a sequence of phrases $Z = b_1 \ldots b_r$, all of them in the dictionary, then we look for the longest prefix of the rest of the text $T[j+1..u]$ which is a phrase of the dictionary. Once we have found this phrase, say b_s of length ℓ_s, we construct a new phrase $b_{r+1} = (s, T[j+\ell_s+1])$, write the pair at the end of the compressed file Z, i.e. $Z = b_1 \ldots b_r b_{r+1}$, and add the phrase to the dictionary.

We will call B_i the string represented by phrase b_i, thus $B_{r+1} = B_s T[j+\ell_s+1]$. In the rest of the paper we assume that the text T has been compressed using the LZ78 algorithm into $n+1$ phrases, $T = B_0 \ldots B_n$, such that $B_0 = \varepsilon$ (the empty string). We say that i is the *phrase identifier* corresponding to B_i, for $0 \leqslant i \leqslant n$.

Property 1. For all $1 \leqslant t \leqslant n$, there exists $\ell < t$ and $c \in \Sigma$ such that $B_t = B_\ell \cdot c$.

That is, every phrase B_t (except B_0) is formed by a previous phrase B_ℓ plus a symbol c at the end. This implies that the set of phrases is *prefix closed*, meaning that any prefix of a phrase B_t is also an element of the dictionary. Therefore, a natural way to represent the set of strings $B_0, \ldots, B_n$ is a trie, which we call *LZTrie*.

Property 2. Every phrase B_i, $0 \leqslant i < n$, represents a different text substring.

[5] $\log x$ means $\lceil \log_2 x \rceil$ in this paper.

This property is used in the LZ-index search algorithm (see Section 3). The only exception to this property is the last phrase B_n. We deal with the exception by appending to T a special symbol "\$" $\notin \Sigma$, assumed to be smaller than any other symbol in the alphabet. The last phrase will contain this symbol and thus will be unique too.

Example 1. In Fig. 1 we show the LZ78 phrase decomposition for our running example text T = "`alabar_a_la_alabarda_para_apalabrarla`", where for clarity we replace blanks by '_', which is assumed to be lexicographically larger than any other symbol in the alphabet. We show the phrase identifiers above each corresponding phrase in the parsing. In Fig. 3(a) we show the corresponding *LZTrie*. Inside each *LZTrie* node we show the corresponding phrase identifier.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`a`	`l`	`ab`	`ar`	`_`	`a_`	`la`	`_a`	`lab`	`ard`	`a_p`	`ara`	`_ap`	`al`	`abr`	`arl`	`a$`

Fig. 1. LZ78 phrase decomposition for the running example text T = "`alabar_a_la_alabarda_para_apalabrarla`", and the corresponding phrase identifiers.

Definition 3. *Let $b_r = (r_1, c_1)$, $b_{r_1} = (r_2, c_2)$, $b_{r_2} = (r_3, c_3)$, and so on until $r_k = 0$ be phrases of the LZ78 parsing of T. The sequence of phrase identifiers $r, r_1, r_2, \ldots$ is called the* referencing chain *starting at phrase r.*

The referencing chain starting at phrase r reproduces the way phrase b_r is formed from previous phrases and it is obtained by successively moving to the parent in the *LZTrie*.

Example 2. The referencing chain of phrase 9 in Fig. 3(a) is $r = 9$, $r_1 = 7$, $r_2 = 2$, and $r_3 = 0$.

The compression algorithm takes $O(u)$ time in the worst case and is efficient in practice provided we use the *LZTrie*, which allows rapid searching of the new text prefix (for each symbol of T we move once in the trie). The decompression needs to build the same dictionary (the pair that defines the phrase r is read at the r-th step of the algorithm).

Property 3 ([48]). It holds that $\sqrt{u} \leqslant n \leqslant \frac{u}{\log_\sigma u}$. Thus, $\log n = \Theta(\log u)$ and $n \log u \leqslant u \log \sigma$ always hold.

Lemma 1 ([28]). *It holds that $n \log n = uH_k(T) + O(u\frac{1+k\log\sigma}{\log_\sigma u})$ for any k.*

In our work we assume $k = o(\log_\sigma u)$ (and hence $\log\sigma = o(\log u)$ to allow for $k > 0$); therefore, $n \log n = uH_k(T) + o(u \log \sigma)$. Also, in the analysis throughout this paper we will assume $\sigma = O(\mathrm{polylog}(u))$; we generalize our results to larger values of σ in Section 8.

2.4 Succinct Representations of Sequences and Permutations

A *succinct data structure* requires space close to the information-theoretic lower bound, while supporting the corresponding operations efficiently. We review here some results on succinct data structures, which are necessary to understand our work.

Data Structures for rank and select. Given a bit vector $\mathcal{B}[1..n]$, we define the operation $\mathsf{rank}_0(\mathcal{B}, i)$ (similarly rank_1) as the number of 0s (1s) occurring up to the i-th position of $\mathcal{B}$. The operation $\mathsf{select}_0(\mathcal{B}, i)$ (similarly select_1) is defined as the position of the i-th 0 (i-th 1) in $\mathcal{B}$. We assume that $\mathsf{select}_0(\mathcal{B}, 0)$ always equals 0 (similarly for select_1). These operations are supported in constant time and using $n + o(n)$ bits [11], or even $nH_0(\mathcal{B}) + o(n)$ bits [42].

Given a sequence $S[1..u]$ over an alphabet Σ, we generalize the above definition for $\mathsf{rank}_c(S, i)$ and $\mathsf{select}_c(S, i)$ for any $c \in \Sigma$. If $\sigma = O(\text{polylog}(u))$, the solution of Ferragina at al. [17] supports both rank_c and select_c, as well as access to $S[i]$ for any i, in constant time and requiring $uH_0(S)+o(u)$ bits of space. Otherwise the time is $O(\frac{\log \sigma}{\log \log u})$ and the space is $uH_0(S) + o(u \log \sigma)$ bits. The representation of Golynski et al. [20] requires $n(\log \sigma + o(\log \sigma)) = O(n \log \sigma)$ bits of space [7], supporting select_c in $O(1)$ time, and rank_c and access to $S[i]$ in $O(\log \log \sigma)$ time.

Succinct Representation of Permutations. The problem here is to represent a permutation π of $\{1, \ldots, n\}$, such that we can compute both $\pi(i)$ and its inverse $\pi^{-1}(j)$ in constant time and using as little space as possible.

Given π represented in plain form, an efficient solution [37] for π^{-1} is based on the cycle notation of a permutation. The cycle for the i-th element of π is formed by elements i, $\pi(i)$, $\pi(\pi(i))$, and so on until the value i is found again, $\pi^k(i) = i$. Then, $\pi^{k-1}(i) = \pi^{-1}(i)$. Every element occurs in one and only one cycle of π. So, to compute $\pi^{-1}(j)$, instead of looking sequentially for j in π, we only need to look for j in its cycle.

To limit the length of cycles, we create subcycles of size $O(1/\epsilon)$ by adding a *backward pointer* out of $O(1/\epsilon)$ elements in each cycle of π, for any $0 < \epsilon < 1$. We store the backward pointers in an array of $\epsilon n \log n$ bits. We mark the elements having a backward pointer using a bit vector supporting rank queries [42], which also helps us to find the backward pointer associated with a given element (see [37] for details). Overall, this solution requires $(1+\epsilon)n \log n + O(\epsilon n \log \frac{1}{\epsilon}) + o(n) \leqslant (1 + \epsilon)n \log n + n + o(n)$ bits of storage.

2.5 Succinct Representation of Trees

Given a tree with n nodes, there exist a number of succinct representations requiring $2n + o(n)$ bits, which is close to the information-theoretic lower bound of $2n - \Theta(\log n)$ bits. We explain the representations that we will need in our work.

Balanced Parentheses. The *balanced parentheses* representation [38] is built from a depth-first preorder traversal of the tree, writing an *opening* parenthesis when arriving at a node for the first time, and a *closing* parenthesis when going up (after traversing the subtree of the node). In this way, each node is represented by a pair of opening and closing parentheses. We identify a tree node x with its opening parenthesis in the representation. The subtree of x contains those nodes (parentheses) enclosed between the opening parenthesis representing x and its matching closing parenthesis.

Let $par[0..2n-1]$ be the balanced parentheses sequence over the alphabet $\{\mathtt{(}, \mathtt{)}\}$. The *preorder position* of a node in this representation can be computed as the number of opening parentheses before the one representing the node. That is, $\mathsf{preorder}(x) \equiv \mathsf{rank}_{\mathtt{(}}(par, x) - 1$. In this way, the preorder of the tree root is always 0. Given a preorder position p, the corresponding node is computed by $\mathsf{selectnode}(p) \equiv \mathsf{select}_{\mathtt{(}}(par, p + 1)$.

This representation requires $2n + o(n)$ bits, supporting operations $\mathsf{parent}(x)$ (which gets the parent of node x), $\mathsf{subtreesize}(x)$ (which gets the size of the subtree rooted at x), $\mathsf{depth}(x)$ (which gets the depth of node x in the tree), and $\mathsf{ancestor}(x, y)$ (which tell us whether node x is an ancestor of node y), all of them in $O(1)$ time. Operation $\mathsf{child}(x, i)$ (which gets the i-th child of node x) can be computed in $O(i)$ time.

Example 3. In Fig. 2(a) we show the balanced parentheses representation for the *LZTrie* of Fig. 3(a), along with the sequence of phrase identifiers (*ids*) in preorder, and the sequence of symbols labeling the edges of the trie (*letts*), also in preorder. As the identifier corresponding to the *LZTrie* root is always 0, we do not store it in *ids*. The data associated with node x is stored at position $\mathsf{preorder}(x)$ both in *ids* and *letts* sequences. Note this information is sufficient to reconstruct *LZTrie*.

```
       0                            10                            20                            30             35
par:   (  (  (  )  (  (  )  )  (  )  (  (  )  (  )  (  )  )  (  (  )  )  )  (  (  (  )  )  )  (  (  (  )  )  )  )
ids:      1  17    3  15       14    4  12    10    16       6  11          2  7  9           5  8  13
letts:    a  $     b  r        l     r  a     d     l        _  p           l  a  b           _  a  p
```

(a) Balanced parentheses representation.

```
       0                            10                            20                            30             35
par:   (  (  (  (  )  (  (  (  (  (  )  )  (  )  )  )  (  (  (  )  )  )  )  (  )  )  (  )  (  )  )  (  )  (  )  )
ids:                  1                 17 3     15 14 4           12 10 16 6     11 2     7     9  5     8     13
letts:    a  l  _     $  b  l  r  _        r           a  d  l              p        a     b        a     p
```

(b) DFUDS representation. The phrase identifiers are stored in preorder, and the symbols labeling the edges of the trie are stored according to DFUDS.

Fig. 2. Succinct representations of *LZTrie* for the running example.

DFUDS **Representation.** To obtain this representation [8] we perform a preorder traversal on the tree, and for every node reached we write its degree in unary using parentheses. For example, 3 reads '((()' under this representation. What we get is almost a balanced parentheses representation: we only need to add a fictitious '(' at the beginning of the sequence. A node of degree d is identified by the position of the first of the $d+1$ parentheses representing the node. Given a node x in this representation, say at position i, its preorder position can be computed by counting the number of closing parentheses before position i; in other words, $\mathsf{preorder}(x) \equiv \mathsf{rank}_{)}(par, x-1)$ where par represents the DFUDS sequence of the tree. Given a preorder position p, the corresponding node is computed by $\mathsf{selectnode}(p) \equiv \mathsf{select}_{)}(par, p) + 1$.

This representation requires also $2n + o(n)$ bits, and we can compute operations $\mathsf{parent}(x)$, $\mathsf{subtreesize}(x)$, $\mathsf{degree}(x)$ (which gets the degree, i.e., the number of children, of node x), $\mathsf{ancestor}(x, y)$,[6] and $\mathsf{child}(x, i)$, all in $O(1)$ time. Operation $\mathsf{depth}(x)$ is also supported in constant time [25].

For cardinal trees (i.e., trees where each node has at most σ children, each child labeled by a symbol in the set $\{1, \ldots, \sigma\}$) we use the DFUDS sequence par plus an array $letts[1..n]$ storing the edge labels according to a DFUDS traversal of the tree: we traverse the tree in depth-first preorder, and every time we reach a node x, we write the symbols labeling the children of x. In this way, the labels of the children of a given node are all stored contiguously in *letts*, which will allow us to support operation $\mathsf{child}(x, \alpha)$ (which gets the child of node x with label $\alpha \in \{1, \ldots, \sigma\}$) efficiently.

[6] As $\mathsf{ancestor}(x, y) \equiv \mathsf{preorder}(x) \leqslant \mathsf{preorder}(y) \leqslant \mathsf{preorder}(x) + \mathsf{subtreesize}(par, x) - 1$.

Example 4. In Fig. 2(b) we show the DFUDS representation of *LZTrie* for our running example.

We support operation $\mathsf{child}(x,\alpha)$ as follows. Suppose that node x has position p within the DFUDS sequence par, and let $p' = \mathsf{rank}_{(}(par, p) - 1$ be the position in *letts* for the symbol of the first child of x. Let $n_\alpha = \mathsf{rank}_\alpha(letts, p'-1)$ be the number of αs up to position $p'-1$ in *letts*, and let $i = \mathsf{select}_\alpha(letts, n_\alpha+1)$ be the position of the $(n_\alpha+1)$-th α in *letts*. If i lies between (and including) positions p' and $p' + \mathsf{degree}(x) - 1$, then the child we are looking for is $\mathsf{child}(x, i-p'+1)$, which, as we said before, is computed in constant time over par; otherwise x does not have a child labeled α. We can also retrieve the symbol by which x descends from its parent with $letts[\mathsf{rank}_{(}(par, \mathsf{parent}(x)) - 1 + \mathsf{childrank}(x) - 1]$, where the first term stands for the position in *letts* corresponding to the first symbol of the parent of node x, and the second term $\mathsf{childrank}(x)$ is the rank of node x within its siblings, which can be computed in constant time [25].

Thus, the time for operation $\mathsf{child}(x,\alpha)$ depends on the representation we use for rank_α and select_α queries. Notice that $\mathsf{child}(x,\alpha)$ could be supported in a straightforward way by binary searching the labels of the children of x, in $O(\log\sigma)$ worst-case time and not needing any extra space on top of array *letts*. The access to $letts[\cdot]$ takes constant time in this case.

Instead, we can represent *letts* with the data structure of Ferragina et al. [17], which requires $n\log\sigma + o(n\log\sigma)$ bits of space, and allows us to compute $\mathsf{child}(x,\alpha)$ in $O(\frac{\log\sigma}{\log\log u})$ time. The access to $letts[\cdot]$ also takes $O(\frac{\log\sigma}{\log\log u})$ time. These times are $O(1)$ whenever $\sigma = O(\mathrm{polylog}(u))$ holds. On the other hand, we can use the data structure of Golynski et al. [20], requiring $O(n\log\sigma)$ bits of space, yet allowing us to compute $\mathsf{child}(x,\alpha)$ in $O(\log\log\sigma)$ time, and access to $letts[\cdot]$ also in $O(\log\log\sigma)$ time. In most of this paper we will use the representation of Ferragina et al., since it is faster for polylog-sized alphabets.

The scheme we have presented to represent *letts* is slightly different from the original one [8], which achieves $O(1)$ time for $\mathsf{child}(x,\alpha)$ for any σ. However, ours is simpler and allows us to efficiently access $letts[\cdot]$, which will be very important in our indices to extract text substrings. We need to store the array of symbols explicitly in case we need to access them (fortunately, this will not asymptotically affect the space requirement of our results).

***xbw* Representation.** The *xbw transform* of Ferragina et al. [13] is a succinct representation for *labeled trees*: Given a labeled tree $\mathcal{T}$, with n nodes and labels taken from an alphabet Σ of size σ, the *xbw* transform of $\mathcal{T}$ is computed by traversing the tree in preorder, and for each node writing a triplet in a table $S_\mathcal{T}$. The first component of each triplet (S_{last}) indicates whether the node is the last child of its parent in the tree, the second component (S_α) is the symbol labeling the edge by which we reach the node, and the third component (S_π) is the string labeling the path from the parent of the node to the root of $\mathcal{T}$. In this way each node is represented by a row in $S_\mathcal{T}$. As a last step we perform an upward-path-sorting of the table by stably sorting the rows of $S_\mathcal{T}$ lexicographically according to the strings in S_π.

Example 5. In Table 2 we show the *xbw* transform for the *LZTrie* of Fig. 3(a).

We have to add a dummy child to each leaf, labeling the dummy edge with a special symbol Δ not in Σ, so that the paths leading to the leaves appear in column S_π and later we can search for them. As we said before, each node in the *LZTrie* is represented by a row in the table, and the row number is called the *xbw position* of the node.

The *xbw* representation supports operations $\mathsf{parent}(x)$, $\mathsf{child}(x,i)$, and $\mathsf{child}(x,\alpha)$, all of them in $O(1)$ time if $\sigma = O(\mathrm{polylog}(u))$, and using $2n\log\sigma + O(n)$ bits of space, because the column S_π of the table is not stored. The representation also allows *subpath queries*, a very powerful operation which, given a string s, returns all the nodes x such that s is a prefix of the string labeling the path from the parent of x to the root. If $\sigma = O(\text{polylog}(n))$, subpath queries can be computed in $O(|s|)$ time [13]. For general σ, the time for all these operations depends on the representation used for S_α (since we need to support rank and select operations on it), which is $O(\frac{\log \sigma}{\log \log u})$ time if we use the representation of [17], and $O(\log \log \sigma)$ time if we use the data structure of [20], in which case the space requirement is $O(n \log \sigma)$.

Because of the upward-path sorting in table $S_{\mathcal{T}}$, the result of a subpath query is a contiguous interval in such table, containing the answers to the query.

Example 6. A subpath query for string '`r`' yields the interval [21..24] in Table 2, corresponding to the nodes with preorders 7, 8, and 9 in Fig. 3(a), plus a fictitious leaf which is a child of node with preorder 4. As another example, a subpath query for string '`ba`' yields the *xbw* interval [13..14], for node with preorder 4 plus a fictitious leaf which is a child of node with preorder 14. In all cases, note that the string s we are looking for is a prefix of the corresponding string in S_π.

Table 2. *xbw* representation for the *LZTrie* of Fig. 3(a).

i	S_{last}	S_α	S_π
1	0	a	empty string
2	0	l	empty string
3	1	_	empty string
4	1	Δ	$a
5	0	$	a
6	0	b	a
7	0	l	a
8	0	r	a
9	1	_	a
10	1	b	al
11	1	Δ	ara
12	1	p	a_
13	1	r	ba
14	1	Δ	bal
15	1	Δ	dra
16	1	a	l
17	1	Δ	la
18	1	Δ	lra
19	1	Δ	pa_
20	1	Δ	p_a
21	0	a	ra
22	0	d	ra
23	1	l	ra
24	1	Δ	rba
25	1	a	_
26	1	p	_a

3 The LZ-index Data Structure

Assume that the text $T[1..u]$ has been compressed using the LZ78 algorithm into $n+1$ phrases $T = B_0 \dots B_n$, as explained in Section 2.3. Next we describe the original LZ-index data structure and search algorithms, and introduce some improvements on them.

Hereafter, given a string $S = s_1 \dots s_i$, we will use $S^r = s_i \dots s_1$ to denote its reverse. Moreover, $S^r[i..j]$ will actually mean $(S[i..j])^r$.

3.1 Original LZ-index Components

The following data structures compose the original LZ-index [39, 40]:

1. *LZTrie*: the trie formed by all the phrases $B_0 \dots B_n$. Given the properties of LZ78 compression, this trie has exactly $n+1$ nodes, each one corresponding to a string.
2. *RevTrie*: the trie formed by all the reverse strings $B_0^r \dots B_n^r$. In this trie there could be internal nodes not representing any phrase. We call these nodes "*empty*".
3. *Node*: a mapping from phrase identifiers to their node in *LZTrie*.
4. *Range*: a data structure for two-dimensional searching in the space $[0..n] \times [0..n]$. We store the points $\{(\mathsf{preorder}_r(t), \mathsf{preorder}_{lz}(t+1)), t \in 0 \dots n-1\}$ in this structure, where $\mathsf{preorder}_r(t)$ is the *RevTrie* preorder of the node for phrase t (considering only the non-empty nodes in the preorder enumeration), and $\mathsf{preorder}_{lz}(t+1)$ is the *LZTrie* preorder of node for phrase $t+1$. For each such point, the corresponding t value is stored.

Fig. 3 shows the *LZTrie*, *RevTrie*, *Range*, and *Node* data structures corresponding to our running example. We show preorder numbers outside each trie node. Empty *RevTrie* nodes are shown in light gray. The next example gives a hint on the usage of those structures for searching, which will be detailed in Section 3.3.

Example 7. To find all phrases ending with substring '`ab`' in the running example, we search for the reversed string '`ba`' in *RevTrie*, reaching the node with preorder 6. The subtree of this *RevTrie* node contains the phrases we are looking for: phrases 3 and 9 (see Fig. 1). As the preorder interval in *RevTrie* defined by this subtree is [6..7], this means that the horizontal semi-infinite range $[-\infty..\infty] \times [6..7]$ in *Range* also contains those phrases. To find all phrases starting with '`ar`', note that the *LZTrie* subtree for node with preorder (incidentally also) 6 (which corresponds to string '`ar`') contains the phrases starting with '`ar`': phrases 4, 12, 10, and 16. The *LZTrie* preorder interval for this subtree is [6..9]. This means that the vertical semi-infinite range $[6..9] \times [-\infty..\infty]$ contains phrases i such that phrase $i+1$ starts with '`ar`': phrases 3, 11, 9, and 15. Finally, the range $[6..9] \times [6..7]$ contains the phrase numbers i such that phrase i ends with '`ab`' followed by phrase $i+1$ starting with '`ar`': phrases 3 and 9, see Fig. 3(c).

3.2 Succinct Representation of the LZ-index Components

In the original work [39], each of the four structures described requires $n \log n + o(u \log \sigma)$ bits of space if they are represented succinctly.

- *LZTrie* is represented using the balanced parentheses representation of Section 2.5 requiring $2n + o(n)$ bits; plus the sequence *letts* of symbols labeling each trie edge, requiring $n \log \sigma$ bits; and the sequence *ids* of $n \log n$ bits storing the LZ78 phrase identifiers. Both *letts* and *ids* are stored in preorder, so we use $\mathsf{preorder}(x)$ to index them. See Fig. 2(a) for an illustration.
- For *RevTrie*, balanced parentheses are also used to represent the *Patricia* tree [35] structure of the trie, compressing empty unary nodes and so ensuring $n' \leqslant 2n$ nodes. This requires at most $4n + o(n)$ bits. The *RevTrie*-preorder sequence of identifiers (*rids*) is stored in $n \log n$ bits (i.e., we only store the identifiers for non-empty nodes). The symbols labeling the edges of the trie and the Patricia-tree skips are not stored in this representation, since they can be retrieved by using the connection with *LZTrie* [39]. Therefore, the navigation on *RevTrie* is more expensive than that on *LZTrie*.
- For *Range*, the data structure of Chazelle [10] permits two-dimensional range searching in a grid of n pairs of integers in the range $[0..n] \times [0..n]$. This data structure supports range queries

in $O((occ + 1)\log n)$ time, where occ is the number of occurrences reported, and requiring $n\log n + O(n\log\log n)$ bits of space [31]. Note that since n is the number of LZ78 phrases of text T, the latter term $O(n\log\log n)$ is $o(u\log\sigma)$. This data structure can count the number of points in a given range in $O(\log n)$ time.

– Finally, *Node* is just a sequence of n pointers to *LZTrie* nodes. As *LZTrie* is implemented using balanced parentheses, *Node*$[i]$ stores the position within the sequence for the opening parenthesis representing the node corresponding to phrase i. As there are $2n$ such positions, we need $n\log 2n = n\log n + n$ bits of storage. See Fig. 3(d) for an illustration.

According to Lemma 1, the final size of the LZ-index is $4uH_k(T) + o(u\log\sigma)$ bits for $k = o(\log_\sigma u)$ (and hence $\log\sigma = o(\log u)$ to achieve more than zero-order compression).

The succinct trie representations used in [39] implement (among others) operations $\mathsf{parent}(x)$ and $\mathsf{child}(x,\alpha)$, both in $O(\log\sigma)$ time for *LZTrie*, and $O(\log\sigma)$ and $O(h\log\sigma)$ time respectively for *RevTrie*, where h is the depth of node x in *RevTrie* (the h in the cost comes from the fact that we must access *LZTrie* to get the label of a *RevTrie* edge). The operation $\mathsf{ancestor}(x,y)$ is implemented in $O(1)$ time both in *LZTrie* and *RevTrie*.

3.3 LZ-index Search Algorithm

Let us consider now the search algorithm for a pattern $P[1..m]$ [39, 40]. For `locate` queries, pattern occurrences are reported in the format $[\![t, \texttt{offset}]\!]$, where t is the phrase where the occurrence starts, and `offset` is the distance between the beginning of the occurrence and the end of the phrase. Later, in Section 7, we will show how to map these two values to a single *text position*. As we deal with an implicit representation of the text (the *LZTrie*), and not the text itself, we distinguish three types of occurrences of P in T, depending on the phrase layout.

Occurrences of Type 1. The occurrence lies inside a single phrase (there are occ_1 occurrences of this type). Given Property 1, every phrase B_t containing P is formed by a shorter phrase B_ℓ followed by a symbol c. If P does not occur at the end of B_t, then B_ℓ contains P as well. We want to find the shortest possible phrase B_i in the LZ78 referencing chain for B_t that contains the occurrence of P.

Note that P^r is a prefix of B_i^r, so B_i can easily be found by searching for P^r in *RevTrie* in $O(m^2\log\sigma)$ time. Say we arrive at node v_r. Any node v_r' descending from v_r in *RevTrie* (including v_r itself) corresponds to a phrase terminated with P. This corresponds with subpath queries (see Section 2.4) in *LZTrie*. For each such v_r', we traverse and report the subtree of the corresponding *LZTrie* node v_{lz} (found using *rids* and *Node*). For any node v_{lz}' in the subtree of v_{lz}, we report an occurrence $[\![t, m + (\mathsf{depth}(v_{lz}') - \mathsf{depth}(v_{lz}))]\!]$, where t is the phrase identifier (*ids*) of node v_{lz}'.

Occurrences of type 1 are located in $O(m^2\log\sigma + occ_1)$ time, since locating each occurrence takes constant time in *LZTrie*. For cardinality queries we just need to compute the subtree size of each v_{lz} in *LZTrie*, as every v_{lz}' in that subtree corresponds to an occurrence of type 1.

Occurrences of Type 2. The occurrence spans two consecutive phrases, B_t and B_{t+1}, such that a prefix $P[1..i]$ matches a suffix of B_t and the suffix $P[i+1..m]$ matches a prefix of B_{t+1} (there are occ_2 occurrences of this type). P can be split at any position, so we have to try them all. The idea is that, for every possible split, we search for the reverse pattern prefix $P^r[1..i]$ in *RevTrie* (obtaining node v_r) and for the pattern suffix $P[i+1\ldots m]$ in *LZTrie* (obtaining node v_{lz}).

As in a trie all the strings represented in a subtree form a preorder interval, we have two preorder intervals: one in the space of reversed phrases (phrases finishing with $P[1..i]$) and one in that of the normal phrases (phrases starting with $P[i+1..m]$). We need to find the phrase pairs $(t, t+1)$ such that t is in the *RevTrie* preorder interval and $t+1$ is in the *LZTrie* preorder interval. As we have seen in Example 7, this is what the range searching data structure (*Range*) is for. If we denote $p_{lz} = \mathsf{preorder}(v_{lz})$ and $p_r = \mathsf{preorder}(v_r)$, we must search *Range* for $[p_{lz}..p_{lz}+\mathsf{subtreesize}(v_{lz})-1] \times [p_r..p_r+\mathsf{subtreesize}(v_r)-1]$. For every pair $(t, t+1)$ found, we report occurrence $[\![t, i]\!]$. Occurrences of type 2 are located in $O(m^3 \log \sigma + (m + occ_2) \log n)$ time, where the first term comes from searching the tries (in particular, searching for the $O(m)$ partitions of P in the *RevTrie*), and the second one is for the $m-1$ range searches on *RevTrie*.

Occurrences of Type 3. The occurrence spans three or more phrases, $B_{t-1}, \ldots, B_{\ell+1}$, such that $P[i..j] = B_t \ldots B_\ell$, $P[1..i-1]$ matches a suffix of B_{t-1} and $P[j+1..m]$ matches a prefix of $B_{\ell+1}$ (there are occ_3 occurrences of this type). We need one more observation for this part: Since the LZ78 algorithm guarantees that every phrase represents a different string (Property 2), there is at most one phrase matching $P[i..j]$ for each choice of i and j. Therefore, if we partition P into more than two consecutive substrings, there is at most one pattern occurrence for such partition, which severely limits occ_3 to $O(m^2)$, since this is the number of different partitions of P.

Let us define matrix $C_{lz}[1..m, 1..m]$ and arrays A_i, for $1 \leqslant i \leqslant m$, which store information about the search. We first identify the only possible phrase matching each substring $P[i..j]$. This is done by searching for every pattern substring $P[i..j]$ in *LZTrie*, for $i = 1, \ldots, m$ and $j = i, \ldots, m$. Thus, we perform a single search in the trie for each i. We record in $C_{lz}[i,j]$ the *LZTrie* node corresponding to $P[i..j]$, and store the pair (id, j) at the end of A_i, such that id is the phrase identifier of the node corresponding to $P[i..j]$. Note that since we search for $P[i..j]$ for increasing j, we get the values of id in increasing order, as the phrase identifier of a node is always larger than that of the parent node. Therefore, the corresponding pairs in A_i are stored by increasing value of id. This process takes $O(m^2 \log \sigma)$ time.

Then we find the $O(m^2)$ maximal concatenations of successive phrases that match contiguous pattern substrings. For $1 \leqslant i \leqslant j \leqslant m$, for increasing j, we try to extend the match of $P[i..j]$ to the right. If id is the phrase identifier for node $C_{lz}[i,j]$, then we have to search for $(id+1, r)$ in array A_{j+1}, for some r. Array A_{j+1} can be binary searched because it is sorted. If we find $(id+1, r)$ in A_{j+1}, this means that $B_{id} = P[i..j]$ and $B_{id+1} = P[j+1..r]$, which also means that the concatenation of phrases $B_{id}B_{id+1}$ equals $P[i..r]$. We repeat the process from $j = r$, and stop when the pair $(id+1, r)$ is not found in the corresponding array (this means that a concatenation of phrases cannot be extended further, so the current concatenation is maximal). See [39] for further details. As we have to perform $O(m^2)$ binary searches in arrays of size $O(m)$, this procedure takes $O(m^2 \log m)$ worst-case time.

Let $P[i..j] = B_t \ldots B_\ell$ be a maximal concatenation. Then we check whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$, that is, we check whether $Node[\ell+1]$ is a descendant of node $C_{lz}[j+1, m]$, in constant time per maximal concatenation. Finally we check whether phrase B_{t-1} ends with $P[1..i-1]$, by starting from $Node[i-1]$ in *LZTrie* and successively going to the parent to check whether the last $i-1$ symbols, read upwards, equal $P^r[1..i-1]$, in $O(m \log \sigma)$ time per maximal concatenation. If all these conditions hold, we report an occurrence $[\![t-1, i-1]\!]$. Overall, occurrences of type 3 are located in $O(m^3 \log \sigma)$ time.

Overall Query Time. Note that each of the $occ = occ_1 + occ_2 + occ_3$ possible occurrences of P lies exactly in one of the three cases above. Overall, the total search time to report the occ occurrences of P in T is $O(m^3 \log \sigma + (m + occ) \log u)$.

Extracting Text Substrings. The original LZ-index is able to extract text substrings, yet not in the way we have defined before: we have to provide an LZ78 phrase number from where to start the extraction. We assume also that the ℓ symbols we want to extract correspond to whole phrases (in Section 7 we shall avoid these restrictions). Given phrase i, we follow the upward path from $Node[i]$ up to the $LZTrie$ root, outputting the symbols labeling the upward path. Then we perform the same procedure but now starting from $Node[i+1]$ in $LZTrie$, and so on until we extract the ℓ desired symbols, taking overall $O(\ell \log \sigma)$ time, because operation parent is implemented in $O(\log \sigma)$ time [39]. Finally, we can uncompress the whole text T in $O(u \log \sigma)$ time using the same idea, starting the procedure from the first LZ78 phrase.

Improving the Algorithm for Finding Maximal Concatenations. In the case of occurrences of type 3, we now improve the algorithm for finding maximal concatenations of phrases, replacing the binary searches on arrays A_i by an access to the correct position in matrix C_{lz}.

When computing maximal concatenations of phrases, for each $1 \leqslant i \leqslant j \leqslant m$, for increasing j, we try to extend the match of $P[i..j]$ to the right. Let id be the phrase identifier for node $C_{lz}[i, j]$. Then $P[i..j] = B_{id}$ holds. Then, to check whether $P[j+1..r] = B_{id+1}$ holds, instead of searching for $(id+1, r)$ in A_{j+1} as before, we note that $r = j + l$, where l is the length of phrase B_{id+1}, which in turn is computed as $l = \mathsf{depth}(Node[id + 1])$ in $LZTrie$. Then we check, in constant time, whether the node $C_{lz}[j + 1, j + l]$ corresponds to identifier $id + 1$. If so, this means that $P[j + 1..r] = B_{id+1}$ holds, for $r = j + l$, and hence we can extend the concatenation of phrases to $P[i..r] = B_{id}B_{id+1}$. We repeat the process for $j = r$ and stop the procedure when the above condition does not hold, or r becomes greater than m.

Note that by using this algorithm to find maximal concatenations we reduce the time to $O(m^2)$, which does not improve the total performance of the algorithm for finding occurrences of type 3. However, the reduction will be relevant in Sections 4 and 5 for improved versions of the LZ-index.

Lemma 2. *Given the LZ78 parsing of text $T\$ = B_0 \dots B_n$, and given a pattern $P[1..m]$, we can compute the maximal concatenation of successive phrases $B_t \dots B_\ell$ that match contiguous pattern substrings $P[i..j]$, for any $0 \leqslant i \leqslant j \leqslant m$, in $O(m^2)$ time overall.*

3.4 A More Compact Version of the LZ-index

In the practical implementation of LZ-index [39, 40], the $Range$ data structure defined in Section 3.1 is replaced by $RNode$, which is a mapping from phrase identifiers to their node in $RevTrie$. The $RNode$ data structure requires $n \log n$ bits, so this practical version of LZ-index also requires $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

The original search algorithm is modified as follows (occurrences of type 1 are found as for the original LZ-index).

Occurrences of Type 2. For every possible split $P[1..i]$ and $P[i + 1..m]$ of P, assume the search for $P^r[1..i]$ in $RevTrie$ yields node v_r, and the search for $P[i + 1..m]$ in $LZTrie$ yields node v_{lz}.

Then, one checks each phrase t in the subtree of v_r and reports it if $Node[t+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has fewer elements, one does the opposite: check phrases from v_{lz} in v_r, using $RNode[t-1]$.

Unlike when using *Range*, now the time to find occurrences of type 2 is proportional to the size of the smallest subtree among those of v_r and v_{lz}, which can be arbitrarily larger than the number of occurrences reported. That is, by using *RNode* we have no worst-case guarantees at search time. However, the average search time for occurrences of type 2 is $O(n/\sigma^{m/2})$ [39, 40] [7]. This is $O(1)$ for long patterns, $m \geqslant 2\log_\sigma n$.

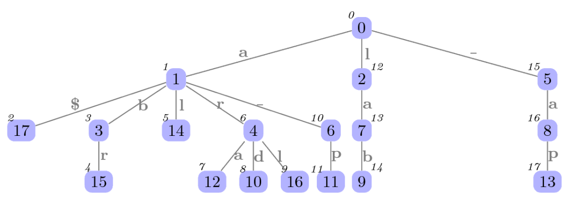
Occurrences of Type 3. For occurrences of type 3, after finding that $P[i..j] = B_t \dots B_\ell$ is a maximal concatenation, one checks whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$ by using operation $\mathsf{ancestor}(C_{lz}[j+1,m], Node[\ell+1])$, just as in Section 3.3. Instead of checking symbol by symbol in the *LZTrie* to determine whether phrase B_{t-1} ends with $P[1..i-1]$, as is done with the original LZ-index, one simply checks whether $\mathsf{ancestor}(C_{lz}[1,i-1], RNode[t-1])$ holds in *RevTrie*.

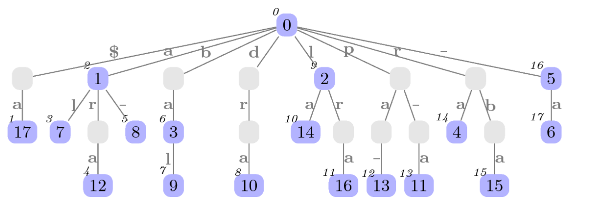
LZ-index as a Navigation Scheme. This version of the LZ-index can be seen as a *navigation scheme*, as shown in Fig. 4, where solid arrows represent the main data structures of the index. Dashed arrows are asymptotically "for free" in terms of space requirement, since they are followed by applying rank on the corresponding parentheses structure (see Section 2.4). The four solid arrows are in fact the four main components in the space usage of the index: array of phrase identifiers in *LZTrie* (*ids*) and in *RevTrie* (*rids*), and mapping from phrase identifiers to tree nodes in *LZTrie* (*Node*) and in *RevTrie* (*RNode*).

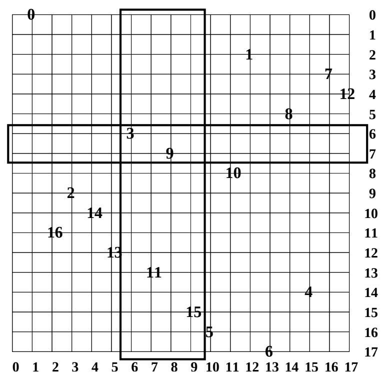
However, we can provide the same navigation functionality needed by the index with a reduced scheme, which we can represent efficiently in the following way:

- *LZTrie*: the Lempel-Ziv trie, which is implemented with the following data structures:
  - $par[0..2n-1]$: the tree shape of *LZTrie* represented with DFUDS [8], requiring $2n+o(n)$ bits.
  - $letts[1..n]$: the array of symbols labeling the edges of *LZTrie*, represented as explained in Section 2.4 so as to allow operation $\mathsf{child}(x,\alpha)$ in constant time, requiring $n\log\sigma + o(n)$ bits of space. We can get the i-th symbol in preorder by first finding the i-th node in preorder in *par*, and then retrieving its symbol as explained in Section 2.4.
  - $ids[1..n]$: the array of LZ78 phrase identifiers in preorder. $ids[0] = 0$ always holds, so we do not store this value. Note that *ids* is a permutation of $\{1,\dots,n\}$, and hence we use the representation [37] given in Section 2.4, such that the inverse permutation $ids^{-1}(j)$ can be computed in $O(1/\epsilon)$ time, requiring $(1+\epsilon)n\log n + n + o(n)$ bits, for any $0 < \epsilon < 1$.
- *RevTrie*: the *Patricia* tree [35] of the reversed LZ78 phrases, which is implemented with the following data structures:
  - $rpar[0..2n'-1]$: the *RevTrie* structure, compressing empty unary paths and represented with DFUDS, thus ensuring $n' \leqslant 2n$ nodes, because empty non-unary nodes still exist. The space requirement is $2n' + o(n')$ bits to support the same functionalities as *LZTrie*.
  - $rletts[1..n']$: the array storing the first symbol of each edge label in *RevTrie*, represented as for *LZTrie* and requiring $n'\log\sigma + o(n')$ bits of space.

[7] The average is taken over the distribution of P, which is assumed to be statistically independent of T, that is, the probability of $P[i..i+\ell] = T[k..k+\ell]$ is $1/\sigma^{\ell+1}$. The text T can be arbitrary.


(a) Lempel-Ziv Trie (*LZTrie*).


(b) *RevTrie* data structure.


(c) *Range* data structure. Horizontal coordinates are for *LZTrie* preorders, and vertical coordinates are for *RevTrie* preorders.

i :	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
Node$[i]$:	0	1	23	4	10	29	18	24	30	25	13	19	11	31	8	5	15	2

(d) *Node* data structure, assuming that the parentheses sequence starts from position zero, cf. Fig. 2(a).

Fig. 3. LZ-index components for the running example.

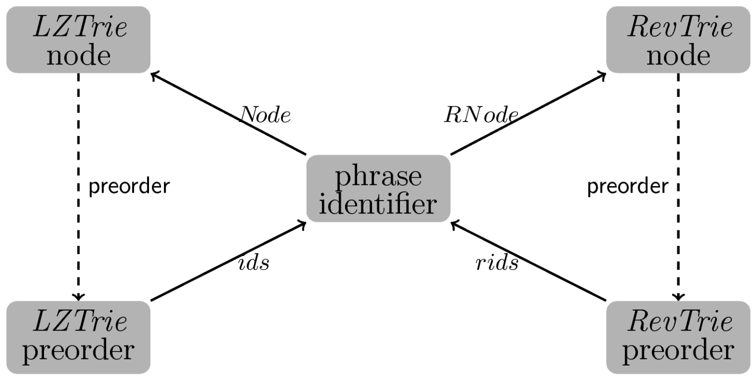


Fig. 4. The original LZ-index navigation structures over index components.

  - $B[1..n']$: a bit vector supporting rank and select queries, and requiring $n'(1 + o(1))$ bits [36]. This bit vector marks the non-empty nodes: the j-th bit of B is 1 iff the node with preorder position j in $rpar$ is not empty, otherwise the bit is 0. Given a position p in $rpar$ corresponding to a *RevTrie* node, the associated bit in B is $B[\mathsf{rank}_{)}(rpar, p-1)]$.
  - $skips[1..n']$: the *Patricia tree* skip values of the nodes in preorder, using $\log\log u$ bits per node and inserting empty unary nodes when the skip exceeds $\log u$. In this way, one out of $\log u$ empty unary nodes could be explicitly represented. In the worst case, there are $O(u)$ empty unary nodes, of which $O(u/\log u)$ are explicitly represented. This adds $O(u/\log u)$ nodes to n', which translates into $O((n' + \frac{u}{\log u})(3 + \log\sigma + \log\log u)) = o(u\log\sigma)$ bits overall for the *RevTrie* nodes, symbols, and skips.
- $R[1..n]$: a mapping from *RevTrie* preorder positions to *LZTrie* preorder positions. Given a non-empty *RevTrie* node with preorder i (just counting non-empty nodes), we define the corresponding *LZTrie* preorder as $R[i] = ids^{-1}(rids[i])$. This is a permutation and is represented using again the succinct data structure for permutations, requiring $(1+\epsilon)n\log n + n + o(n)$ bits to represent R and compute R^{-1} in $O(1/\epsilon)$ worst-case time. Given a position p in $rpar$ corresponding to a non-empty *RevTrie* node, the associated R value (i.e., preorder in *LZTrie*) can be computed as $R[\mathsf{rank}_1(B, \mathsf{rank}_{)}(rpar, p-1))]$.

In Fig. 5 we draw the navigation scheme. The search algorithm remains the same since we can map preorder positions to nodes in the DFUDS representation of the tries and vice versa (see Section 2.4), and also we can simulate the missing arrays: $rids(i) \equiv ids[R[i]]$, $RNode(i) \equiv \mathsf{selectnode}(R^{-1}(ids^{-1}(i)))$, and $Node(i) \equiv \mathsf{selectnode}(ids^{-1}(i))$, all of which take $O(1/\epsilon)$ time.

Space and Time Analysis. The space requirement is $(2+\epsilon)n\log n + 3n\log\sigma + 2n\log\log u + 10n + o(u) = (2+\epsilon)n\log n + o(u\log\sigma)$ bits, which according to Lemma 1 is $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$.

The child operation on *RevTrie* can now be computed in $O(1)$ time thanks to DFUDS, to *rletts*, and to the skips, versus the $O(h\log\sigma)$ time of the original LZ-index [39]. Now, because *RevTrie* is a Patricia tree and the underlying strings are not readily available, it is not obvious how to traverse it. The next lemma addresses this issue.

Lemma 3. *Given a string $s \in \Sigma^*$, we can determine whether it is represented in RevTrie or not (and finding the corresponding node in the affirmative case) in $O(|s|)$ time.*

Proof. To find the node corresponding to string s, we descend from the *RevTrie* root, using operation $\mathsf{child}(x, \alpha)$ on the first symbol of each edge label, which is stored in *rletts*, and using the skips to compute the next symbol of s to use in the descent. If some symbol of s cannot be matched while descending, then we determine that it is not represented in *RevTrie* in $O(|s|)$ time. Otherwise, assume that after consuming string s in this way, we arrive at node v_r with preorder j in *RevTrie*. The string labeling the root-to-v_r path in *RevTrie* can be computed by accessing the node v_{lz} with preorder $R[j]$ in *LZTrie*, and then extracting the string labeling the v_{lz}-to-root path in *LZTrie*. Then we compare that string against s to verify that the node we arrived at corresponds to s, or otherwise that s does not occur in *RevTrie*.

In the case where node v_r in *RevTrie* is empty, $R[j]$ is undefined. Notice, however, that there must be at least one non-empty node descending from this empty node, since leaves in *RevTrie* cannot be empty as they always correspond to an LZ78 phrase. Given that the string represented by every non-empty node in the subtree of node v_r has the string s as a prefix, the corresponding strings in *LZTrie* have s^r as a suffix. So we can use any R value within the subtree of node v_r in order to map to the *LZTrie* and then extract the string it represents. We can compute the preorder j' of the next non-empty node within that subtree by $j' = \mathsf{select}_1(B, \mathsf{rank}_1(B, j) + 1)$, where $\mathsf{rank}_1(B, j)$ represents the number of non-empty nodes up to node v_r. Thus, we use $R[j']$ to access the *LZTrie* and extract the corresponding string. We know when to stop extracting, since the length of the string represented by v_r matches the length of the string we are looking for.

The overall cost for the descending process is therefore $O(|s|)$. □

Operations child and parent on *LZTrie* can be also computed in $O(1)$ time, versus the $O(\log \sigma)$ time of the original LZ-index. Hence, occurrences of type 1 are found in $O(m + occ)$ time, by using the original algorithm of Section 3.3; occurrences of type 2 are found by using the algorithm explained at the beginning of Section 3.4, in $O(\frac{n}{\epsilon \sigma^{m/2}})$ average time, where the $O(1/\epsilon)$ factor comes from simulating *Node* and *RNode* (by using ids^{-1} and R^{-1} respectively) for the checks of type 2; occurrences of type 3 are found in $O(\frac{m^2}{\epsilon})$ worst-case time, since we use the method of Lemma 2 to find the maximal concatenations of phrases in $O(\frac{m^2}{\epsilon})$ time (because *Node* is simulated), and then we need to use *Node* and *RNode* to check every possible candidate, in $O(\frac{m^2}{\epsilon})$ worst-case time as well (as explained at the beginning of Section 3.4). Therefore, the *occ* occurrences of P in T can be located in $O(\frac{m^2}{\epsilon} + \frac{n}{\epsilon \sigma^{m/2}})$ average time, for $0 < \epsilon < 1$. Since the term $O(\frac{n}{\epsilon \sigma^{m/2}})$ is $O(\frac{1}{\epsilon})$ for $m \geqslant 2 \log_\sigma u$, the time is $O(\frac{m^2}{\epsilon})$ on average for $m \geqslant 2 \log_\sigma u$.

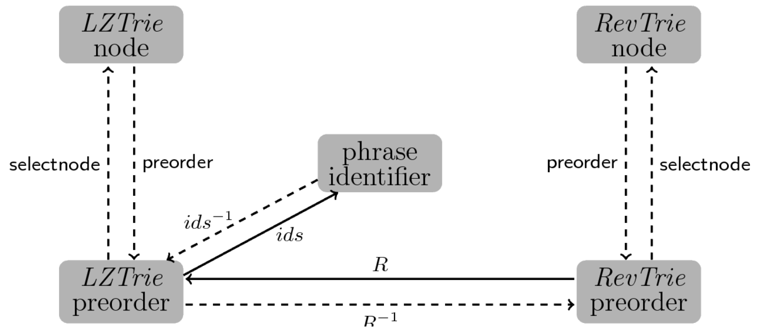


Fig. 5. A more compact navigation scheme over LZ-index components, requiring $(2 + \epsilon)uH_k + o(u \log \sigma)$ bits.

4 Suffix Links in *RevTrie*

As we have seen in Section 3.4, for the LZ-index we can achieve $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(m^2/\epsilon)$ average search time for patterns of length $m \geqslant 2 \log_\sigma u$ [4]. Hence, two questions may arise:

Question 1. Can we reduce the space requirement of LZ-index to $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits, that is, to almost optimal in terms of H_k?

Question 2. Can we retain worst-case guarantees at search time (as for the original LZ-index), while still using at most $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of storage (as for the scheme of Fig. 5)?

In this section (and in the next one), we will find affirmative answers to these questions. Specifically, Theorem 1 shall answer Question 1, and Theorem 2 shall answer Question 2.

We will build on the more compact scheme described in Section 3.4 and illustrated in Fig. 5. Given a *LZTrie* node with preorder position i, we define $\mathsf{parent}_{lz}(i) \equiv \mathsf{preorder}(\mathsf{parent}(\mathsf{selectnode}(i)))$. That is, parent_{lz} is the parent operation working on preorders rather than on DFUDS numbers. Let $\mathsf{child}_{lz}(i,\alpha)$ be defined similarly as $\mathsf{child}_{lz}(i,\alpha) \equiv \mathsf{preorder}(\mathsf{child}(\mathsf{selectnode}(i),\alpha))$. Also, let us define $letts_{lz}(i) \equiv letts[\mathsf{rank}_((par, \mathsf{parent}(x))-1+\mathsf{childrank}(x)-1]$, which yields the symbol by which the node with preorder i descends from its parent, where node x is computed as $\mathsf{selectnode}(i)$. Let $str_{lz}(i)$ denote the string represented by the node with preorder i in *LZTrie*. In the same way, we define $str_r(j)$ for the node with preorder j in *RevTrie*.

The idea is that we are going to compress the R mapping defined for the compact LZ-index of Fig. 5. Let us see this array as a kind of suffix array which, instead of storing text positions, stores *LZTrie* preorder positions. R is a lexicographically sorted array of the reversed LZ78 phases (because it is sorted according to *RevTrie* preorders). Given a reversed phrase with preorder i in *RevTrie* (and preorder $R[i]$ in *LZTrie*), its longest proper suffix has position $\mathsf{parent}_{lz}(R[i])$ in *LZTrie* (as this corresponds to the longest proper prefix in *LZTrie*). Given a reverse phrase with position j in *LZTrie*, its lexicographic rank is $R^{-1}[j]$.

Given this analogy, the question is: can we compress the R mapping just as we can compress a suffix array [22, 45]? We define now the analogue in LZ-index to function Ψ of Compressed Suffix Arrays [22, 45].

Definition 4. *For every RevTrie preorder* $1 \leqslant i \leqslant n$ *we define function* φ *such that* $\varphi(i) = R^{-1}(\mathsf{parent}_{lz}(R[i]))$, *and* $\varphi(0) = 0$.

We have the following properties for function φ.

Property 4. Given a non-empty node with preorder i in *RevTrie*, such that $str_r(i) = ax$, for some $a \in \Sigma$, $x \in \Sigma^*$, then

1. $str_r(\varphi(i)) = x$,
2. $R[\varphi(i)] = \mathsf{parent}_{lz}(R[i])$, and
3. $letts_{lz}(R[i]) = a$.

Point 1 means that φ acts as a *suffix link* in *RevTrie*: since $str_r(i) = ax$, we have that $str_{lz}(R[i]) = x^r a$ (recall that node i in *RevTrie* corresponds to node $R[i]$ in *LZTrie*). Therefore, $str_{lz}(\mathsf{parent}_{lz}(R[i])) = x^r$, which finally means $str_r(R^{-1}(\mathsf{parent}_{lz}(R[i]))) = str_r(\varphi(i)) = x$. Point 2 implies that by following a suffix link in *RevTrie*, we are "going to the parent" in *LZTrie*, and it follows from applying R to both sides of the equation in Definition 4, as Fig. 6 illustrates these facts. Note that the edge connecting the *LZTrie* nodes with preorders $R[\varphi(i)]$ and $R[i]$ is labelled a (as stated by point 3 in Property 4), which is the same symbol we are missing when following the suffix link $\varphi(i)$ in *RevTrie*.

We can prove that *RevTrie* is suffix closed since *LZTrie* is prefix closed, hence suffix links are well defined.

Lemma 4. *Every non-empty node in RevTrie has a suffix link.*

Proof. Let us consider any non-empty node in *RevTrie* with preorder i, such that $str_r(i) = ax$, for $a \in \Sigma$ and $x \in \Sigma^*$. As ax is a *RevTrie* phrase (with preorder i), then $x^r a$ must be a *LZTrie* phrase (with preorder $R[i]$). By Property 1 of the LZ78 parsing it follows that x^r is also a *LZTrie* phrase and thus x must be a *RevTrie* phrase. Hence, every non-empty node in *RevTrie* (i.e., every *RevTrie* node belonging to a reverse LZ78 phrase) has a suffix link. □

We will use Property 4 to reduce the space requirement of the R mapping: suppose that we do not store $R[i]$, for the *RevTrie* node with preorder i in Fig. 6, but we store $R[\varphi(i)]$; then note that $R[i]$ can be computed as $\mathsf{child}_{lz}(R[\varphi(i)], a)$. Russo and Oliveira [44] also use properties of suffix links in their index; however, their main objective is to reduce the locating complexity of their LZ-index to $O((m + occ) \log u)$, not to reduce the space requirement as we do. In Section 6, however, we will show how to use suffix links to reduce the time complexity of our indices, achieving their same locating complexity while requiring less space.

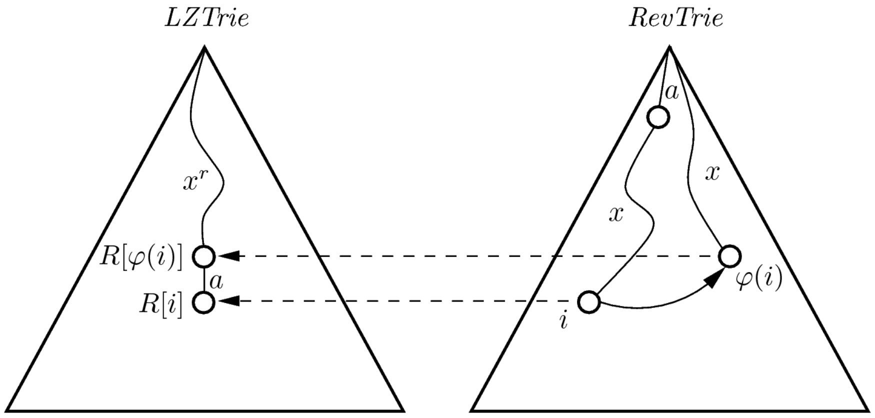


Fig. 6. Illustration of Property 4. Preorder numbers, both in *LZTrie* and *RevTrie*, are shown outside each node. Dashed arrows associate a *RevTrie* node with its corresponding node in *LZTrie*. This association is given by the R mapping.

Example 8. In Table 3 we show some arrays composing the reduced version of LZ-index for our running example, including the φ function and the set of reversed LZ78 phrases in *RevTrie* (in preorder, i.e., lexicographically sorted).

4.1 Using Suffix Links to Compute R

Let us show how to compute $R[i]$ using function φ. We define array $L[1..n]$, which for each non-empty node with preorder i in *RevTrie* stores the first symbol of the string $str_r(i)$.

Example 9. In the *RevTrie* of Fig. 3(b), it holds that $L[i] =$ '`a`' for every i in the preorder interval $[2, 5]$. In the same example, note that if we follow the suffix link $\varphi(i)$, for $2 \leqslant i \leqslant 5$, we discard the symbol '`a`'.

In the example of Fig. 6 we have $L[i] = a$; as we said before, this also means that $L[i]$ is the label of the edge connecting *LZTrie* nodes with preorders $R[\varphi(i)]$ and $R[i]$. In other words, $L[i] = letts_{lz}(R[i])$.

Table 3. Illustration of the different components of our index for the running example. In the case of *RevTrie*, array *rids* is shown just for simplicity, yet this is not explicitly stored. In each case, i indicates the preorders in each trie.

LZTrie components

i	$ids[i]$	$letts_{lz}(i)$	$R^{-1}(i)$
0	0		0
1	1	`a`	2
2	17	`$`	1
3	3	`b`	6
4	15	`r`	15
5	14	`l`	10
6	4	`r`	14
7	12	`a`	4
8	10	`d`	8
9	16	`l`	11
10	6	`_`	17
11	11	`p`	13
12	2	`l`	9
13	7	`a`	3
14	9	`b`	7
15	5	`_`	16
16	8	`a`	5
17	13	`p`	12

RevTrie components

i	$rids[i]$	$R[i]$	$\varphi(i)$	$L[i]$	$L_B[i]$	string in *RevTrie*
0	0	0	0			`(empty string)`
1	17	2	2	`$`	0	`$a`
2	1	1	0	`a`	1	`a`
3	7	13	9	`a`	0	`al`
4	12	7	14	`a`	0	`ara`
5	8	16	16	`a`	0	`a_`
6	3	3	2	`b`	1	`ba`
7	9	14	3	`b`	0	`bal`
8	10	8	14	`d`	1	`dra`
9	2	12	0	`l`	1	`l`
10	14	5	2	`l`	0	`la`
11	16	9	14	`l`	0	`lra`
12	13	17	5	`p`	1	`pa_`
13	11	11	17	`p`	0	`p_a`
14	4	6	2	`r`	1	`ra`
15	15	4	6	`r`	0	`rba`
16	5	15	0	`_`	1	`_`
17	6	10	2	`_`	0	`_a`

Example 10. In Table 3 we show the values of L for our running example.

It is not hard to prove that $L[i] \leqslant L[j]$ whenever $i \leqslant j$: let i and j be preorder numbers in *RevTrie*, such that $i \leqslant j$. Therefore, for the strings corresponding to these preorders it holds that $str_r(i) \leqslant str_r(j)$. As $L[i]$ and $L[j]$ store the first symbol of $str_r(i)$ and $str_r(j)$ respectively, then it holds that $L[i] \leqslant L[j]$. Thus, L can be divided into σ runs of equal symbols. In this way L can be represented by an array L' of at most $\sigma \log \sigma$ bits and a bit vector L_B of $n + o(n)$ bits, such that $L_B[i] = 1$ iff $L[i] \neq L[i-1]$, for $i = 2 \ldots n$, and $L_B[1] = 0$ (this position belongs to the text terminator "\$", which is not in the alphabet). For every i such that $L_B[i] = 1$, we store $L'[\mathsf{rank}_1(L_B, i)] = L[i]$. Hence, $L[i]$ can be computed as $L'[\mathsf{rank}_1(L_B, i)]$ in $O(1)$ time.

Given a *RevTrie* preorder position i, in order to compute $R[i]$ we could follow suffix links in *RevTrie* starting from node with preorder i, until we reach the *RevTrie* root. At this point we could apply, starting from the root of *LZTrie*, child operations using the first symbol of each *RevTrie* string we obtained while following suffix links, in reverse order. This procedure is formalized in the following lemma.

Lemma 5. *Given a RevTrie preorder position $0 \leqslant i \leqslant n$, the corresponding LZTrie preorder position $R[i]$ can be computed by the following recurrence:*

$$R[i] = \begin{cases} \mathsf{child}_{lz}(R[\varphi(i)], L[i]) & \text{if } i \neq 0 \\ 0 & \text{if } i = 0 \end{cases}$$

Proof. $R[0] = 0$ holds from the fact that the preorder position corresponding to the empty string, both in *LZTrie* and *RevTrie*, is 0. To prove the other part, we note that if x is the parent in *LZTrie* of node y with preorder position $R[i]$, then the symbol labeling the edge connecting x to y is stored in $L[i] = letts_{lz}(R[i])$. That is, $\mathsf{child}_{lz}(\mathsf{parent}_{lz}(R[i]), L[i]) = R[i]$. The lemma follows from this fact and replacing $\varphi(i)$ by Definition 4 in the recurrence. $\square$

Example 11. To compute $R[13]$ for the example of Table 3, which corresponds to string '$\mathsf{p\_a}$' in *RevTrie*, we need to compute $\mathsf{child}_{lz}(R[17], L[13])$, where $L[13] = $ 'p' and $\varphi(13) = 17$ corresponds to the *RevTrie* preorder position of string '$\mathsf{\_a}$'. Now, to compute $R[17]$ we need to compute $\mathsf{child}_{lz}(R[2], L[17])$, where $L[17] = $ '$\mathsf{\_}$'. Then, $R[2]$ is computed as $\mathsf{child}_{lz}(R[0], L[2])$, which is just $\mathsf{child}(0, \text{'}\mathsf{a}\text{'})$. At this point we must perform the child operations from the *LZTrie* root. Recall that we assume the child_{lz} operation works on preorder positions, so we must compute $\mathsf{child}_{lz}(\mathsf{child}_{lz}(\mathsf{child}_{lz}(0, \text{'}\mathsf{a}\text{'}), \text{'}\mathsf{\_}\text{'}), \text{'}\mathsf{p}\text{'})$ in *LZTrie*, which is the same as $\mathsf{child}_{lz}(\mathsf{child}_{lz}(1, \text{'}\mathsf{\_}\text{'}), \text{'}\mathsf{p}\text{'})$, which in turn is $\mathsf{child}_{lz}(10, \text{'}\mathsf{p}\text{'})$, which finally yields the node with preorder position 11. Hence, we conclude that $R[13] = 11$.

4.2 Compressing the R Mapping

As in the case of the Ψ function of *Compressed Suffix Arrays* [22, 45], we can prove the following lemma for the φ function, which is the key to compressing the R mapping.

Lemma 6. *For every $i < j$, if $L[i] = L[j]$, then $\varphi(i) < \varphi(j)$.*

Proof. Let $str_r(i)$ denote the i-th string in the lexicographically sorted set of reversed strings. Note that $str_r(i) < str_r(j)$ iff $i < j$. If $i < j$ and $L[i] = L[j]$ (i.e., $str_r(i)$ and $str_r(j)$ start with the same symbol), then $str_r(\varphi(i)) < str_r(\varphi(j))$ (as $str_r(\varphi(i))$ is $str_r(i)$ without its first symbol, recall Property 4, point 1), and thus $\varphi(i) < \varphi(j)$. □

Corollary 1. *Array φ can be partitioned into at most σ strictly increasing sequences.*

This fact is illustrated in Table 3, where the increasing runs of φ, corresponding to runs of equal symbols in L, are separated by horizontal lines.

As a result, we replace R by φ, L', and L_B, and use them to compute a given value $R[i]$. According to Corollary 1, we can represent φ using the idea of Sadakane [45] to represent Ψ. Thus, φ can be encoded with $nH_0(letts) + O(n \log\log\sigma)$ bits, and hence we replace the $n\log n$-bits representation of R by the $nH_0(letts) + O(n\log\log\sigma) + n + o(n) = O(n\log\sigma) = o(u\log\sigma)$ bits of the representation of φ, L', and L_B.

4.3 Computing R and R^{-1} in $O(1/\epsilon)$ Time

According to Lemma 5, the time to compute $R[i]$ is $O(|str_r(i)|)$, which actually corresponds to traversing *LZTrie* from the root with the symbols of $str_r(i)$ in reverse order. However, the procedure of Lemma 5 can be adapted to allow constant-time computation of $R[i]$. We store ϵn values of R in an array R', plus a bit vector R_B of $n + o(n)$ bits indicating which values of R have been stored, ensuring that $R[i]$ can be computed in $O(1/\epsilon)$ time while requiring $\epsilon n \log n$ extra bits.

To determine the R values to be explicitly stored, we fix $l = \Theta(1/\epsilon)$ and carry out a preorder traversal on *LZTrie* to mark the nodes *(1)* whose *depth* is $j \cdot l$, for some $j > 0$, and such that *(2)* the corresponding node *height* is greater or equal to l. Since for every such marked node we have at least l non-marked nodes descending from it, we mark $O(\epsilon n)$ nodes overall. We also ensure that, if we start at an arbitrary node in *LZTrie* and go successively to the parent, in the worst case we must apply $O(1/\epsilon)$ parent operations to find a marked node. It can be the case that near the leaves of the trie we must follow a longer path to get a marked node, because of condition (2) above. However, notice that this path is never longer than $2l$, which still is $O(1/\epsilon)$. On the *RevTrie* side, this means that in the worst case we must follow $O(1/\epsilon)$ suffix links to find a node whose R value has been stored.

If the node to mark is at preorder position j, then we set $R_B[R^{-1}(j)] = 1$ (note that R_B is indexed by *RevTrie* preorder). After we mark the positions of R to be stored, we scan R_B sequentially from left to right, and for every i such that $R_B[i] = 1$, we set $R'[\mathsf{rank}_1(R_B, i)] = R[i]$. Then, we free R since $R[i]$ can be computed in $O(1/\epsilon)$ worst-case time, as stated by the following lemma.

Lemma 7. *Given a RevTrie preorder position $0 \leqslant i \leqslant n$ and given any $0 < \epsilon < 1$, the corresponding LZTrie preorder position $R[i]$ can be computed in constant $O(1/\epsilon)$ worst-case time by the following recurrence:*

$$R[i] = \begin{cases} \mathsf{child}(R[\varphi(i)], L'[\mathsf{rank}_1(L_B, i)]) & \text{if } R_B[i] = 0 \\ R'[\mathsf{rank}_1(R_B, i)] & \text{if } R_B[i] = 1. \end{cases}$$

This structure requires $\epsilon n \log n + O(n \log \sigma) = \epsilon H_k(T) + o(u \log \sigma)$ bits of space.

Note that the same structure used to compute R^{-1} using the explicit representation of R, can be used under this reduced-space representation of R, with cost $O(1/\epsilon^2)$ to compute $R^{-1}(j)$ (as we have to access $O(1/\epsilon)$ positions in R). However, we show now how to compute $R^{-1}(j)$ in $O(1/\epsilon)$ time, using a novel approach which basically consists in reverting the process used to compute R.

Definition 5. *For every RevTrie preorder $0 \leqslant i \leqslant n$ and every symbol $a \in \Sigma$, we define function φ' such that $\varphi'(i, a) = R^{-1}(\mathsf{child}_{lz}(R[i], a))$.*

We have the following immediate properties for function φ'.

Property 5. Given a non-empty node with preorder i in *RevTrie*, such that $str_r(i) = x$, for $x \in \Sigma^*$, then for $a \in \Sigma$ it holds that

1. $str_r(\varphi'(i, a)) = ax$,
2. $R[\varphi'(i, a)] = \mathsf{child}_{lz}(R[i], a)$.

Point (1) means that φ' acts as a *Weiner link* [47] in *RevTrie*. Point (2) means that by following a Weiner link by symbol a from node with preorder i, we are "going to a child by symbol a" in *LZTrie*. See Fig. 7 for an illustration.

Next we show how to efficiently compute φ' while requiring little space (since the obvious way to represent it requires basically $n \log n$ bits). Let $S_W[1..n]$ be an array of $n \log \sigma$ bits storing, for every *RevTrie* node, in preorder, the symbols by which the node has Weiner links defined, and let V_W be a bit vector. Because of Property 5 (2) (i.e., following a Weiner link by a symbol in *RevTrie* means going to the child by the same symbol in *LZTrie*), we can use the *LZTrie* as an aid to construct S_W: We perform a preorder traversal on *RevTrie*, and for every non-empty node with preorder i, let d be the degree of the corresponding *LZTrie* node $R[i]$. Then, we write the degree d in unary in V_W, in the format $\mathtt{1}\mathtt{0}^d$. Thus, the $\mathtt{1}$s in V_W will be used to locate the position of a node within the data structure (via operation select_1), while the $\mathtt{0}$s in V_W shall be used to locate the position for the symbols of the links of a given node, as we will see soon. In the same traversal we also store in S_W the symbols labeling the children of node $R[i]$ in *LZTrie*. We represent arrays V_W and S_W with data structures for rank and select queries, requiring $o(u \log \sigma)$ bits overall.

In order to understand how Weiner links can be represented in a compact way and computed efficiently, we shall store them (conceptually) in such a way that we can divide the resulting array

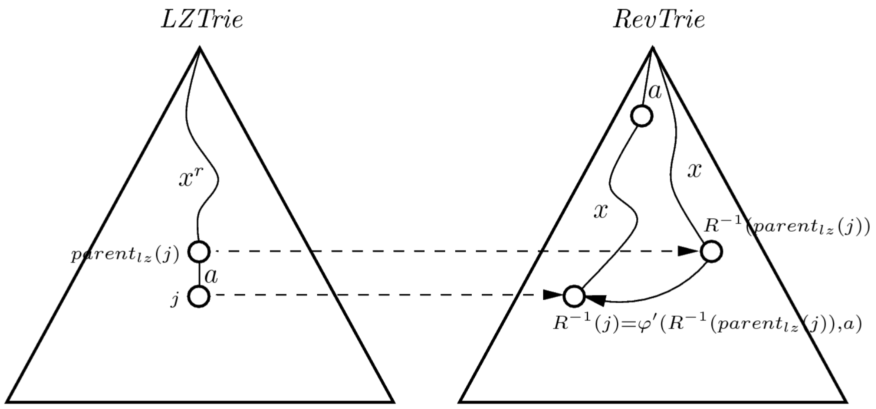


Fig. 7. Illustration of Property 5 for function φ'. Dashed arrows associate an *LZTrie* node with its corresponding node in *RevTrie*. This association is given by R^{-1}.

into at most σ strictly increasing subsequences (note that this cannot be ensured if we simply store the links in preorder). Let $W[1..n]$ be the (conceptual) array storing the sequence of Weiner links. We will have an increasing subsequence in W for every symbol in the alphabet; every such subsequence stores the links going out by that symbol. Let $C_W[1..\sigma]$ be an array storing the starting position for the subsequence corresponding to every alphabet symbol. We then carry out a new preorder traversal on *RevTrie*. For every non-empty node with preorder i (counting just non-empty nodes), let $R[i]$ be the corresponding *LZTrie* node, of degree d. Let $i_1 \leftarrow \mathsf{select}_1(V_W, i+1)$ be the position in V_W corresponding to the current *RevTrie* node. Let $i_2 \leftarrow \mathsf{rank}_0(V_W, i_1) + 1$ be the starting position in S_W corresponding to the current node. Then, for every child $j = 1, \ldots, d$ of node $R[i]$, which is labeled by symbol $s \leftarrow \mathit{letts}_{lz}(\mathsf{child}_{lz}(R[i], j))$ in *LZTrie*, we store $W[C_W[s] + \mathsf{rank}_s(S_W, i_2 - 1)] \leftarrow R^{-1}(\mathsf{child}_{lz}(R[i], j))$.

Given this representation, we can compute, for any non-empty node with preorder i in *RevTrie* and a symbol $a \in \Sigma$:

$$\varphi'(i, a) \equiv W[C_W[a] + \mathsf{rank}_a(S_W, \mathsf{rank}_0(V_W, \mathsf{select}_1(V_W, i + 1)) + 1)].$$

Now it remains to show that this representation can be compressed.

Lemma 8. *Array W can be partitioned into at most σ strictly increasing sequences.*

Proof. Let positions i and j in W, for $i < j$, correspond to Weiner links going out by the same symbol $a \in \Sigma$. Assume that position i corresponds to the node for string $x \in \Sigma^*$ in *RevTrie*, and position j corresponds to string $y \in \Sigma^*$. Since $i < j$ and given the way in which W is constructed, it follows that the preorder of the node for x is smaller than the preorder of the node representing y. This also means that $x < y$. Then, $ax < ay$ also holds. Therefore the preorder stored at $W[i]$ (i.e., the one pointing to the node for string ax) is smaller than the preorder stored at $W[j]$ (which points to the node for string ay). $\square$

Thus, we could represent W in the same way as array φ, requiring overall $O(n \log \sigma) = o(u \log \sigma)$ bits of space. However, we can do better. Let $j \leftarrow \mathsf{child}_r(0, a)$ be the preorder of the child of the *RevTrie* root by symbol a. Notice that all Weiner links going out by a given symbol, say symbol a, point to a node within the subtree of the node with preorder j. Since Lemma 8 states that the

Weiner links for symbol a appear in increasing order within the corresponding subsequence of W, this means that the first link for a points to the first node in preorder within the subtree of the node with preorder j, the second link points to the second node in preorder within the subtree of the node with preorder j, and so on. This means that just performing a rank on S_W allows us to compute the corresponding link, so we do not need to store array W. Formally, we have:

$$\varphi'(i,a) \equiv \mathsf{child}_r(0,a) + \mathsf{rank}_a(S_W, \mathsf{rank}_0(V_W, \mathsf{select}_1(V_W, i+1)) + 1) - 1. \qquad (1)$$

(Note this mechanism can be regarded as extending the LF-mapping of the Burrows-Wheeler transform [9, 16] to tries).

We will use function φ' and its properties in order to compute R': suppose that we do not store $R^{-1}(j)$ for the *LZTrie* node with preorder j in Fig. 7, but we store $R^{-1}(\mathsf{parent}_{lz}(j))$. Then $R^{-1}(j)$ can be computed as $\varphi'(R^{-1}(\mathsf{parent}_{lz}(j)), a)$. For every *LZTrie* node that has been marked to store an R value, as explained above, we also store the corresponding value of R^{-1} in array R''. We mark in a bit vector R^{-1}_B (according to preorder in *LZTrie*) the nodes whose R^{-1} value has been stored. This ensures that, starting at an arbitrary node in *LZTrie*, we shall find a sampled node after performing at most $O(1/\epsilon)$ parent operations. Then we can conclude:

Lemma 9. *Given an LZTrie preorder position $0 \leqslant j \leqslant n$ and given any $0 < \epsilon < 1$, the corresponding RevTrie preorder position $R^{-1}(j)$ can be computed in $O(1/\epsilon)$ worst-case time by the following recurrence:*

$$R^{-1}(j) = \begin{cases} \varphi'(R^{-1}(\mathsf{parent}_{lz}(j)), letts_{lz}(j)) & \text{if } R^{-1}_B[j] = 0 \\ R''[\mathsf{rank}_1(R^{-1}_B, j)] & \text{if } R^{-1}_B[j] = 1. \end{cases}$$

This structure requires $\epsilon n \log n + O(n\log\sigma) = \epsilon H_k(T) + o(u\log\sigma)$ bits of space.

4.4 Space and Time Analysis

As now we store *ids* in $n\log n$ bits, ids^{-1}, R', and R'' in $\epsilon n\log n$ bits each, and φ, φ', *letts*, and *rletts* in $O(n\log\sigma) = o(u\log\sigma)$ bits, the total space requirement is $(1+\epsilon)n\log n + o(u\log\sigma)$ bits (renaming $4\epsilon = \epsilon$), and we provide the same navigation scheme as in Fig. 5. Occurrences of type 1 are found as usual, in $O(m + \frac{occ_1}{\epsilon})$ time, where the extra $O(\frac{occ_1}{\epsilon})$ term appears because we have to use R to map from *RevTrie* to *LZTrie*, which takes $O(1/\epsilon)$ each time. Occurrences of type 2 are found as explained in Section 3.4, in $O(\frac{n}{\epsilon\sigma^{m/2}})$ average time since now the access between tries is provided by R and R^{-1}. For solving occurrences of type 3, we first search for all the pattern substrings in *LZTrie* in $O(m^2)$ time, and then compute the maximal concatenations of phrases, in $O(\frac{m^2}{\epsilon})$ time by using the improved algorithm of Lemma 2 (the $O(1/\epsilon)$ factor comes from the fact that we use ids^{-1} to simulate *Node*). Finally, for each of the $O(m^2)$ maximal concatenations found, we carry out the tests as explained in Section 3.4, with cost $O(\frac{m^2}{\epsilon})$ because *RNode* is implemented by using R^{-1}. We have proved:

Theorem 1. *Given a text $T[1..u]$ over an alphabet of size σ and let n be the number of phrases in the LZ78 parsing of T, there exists a compressed full-text self-index requiring $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for $\sigma = O(\mathrm{polylog}(u))$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a pattern $P[1..m]$, this index is able to locate (and count) the occ occurrences of P in T in $O(\frac{m^2}{\epsilon} + \frac{u}{\epsilon\sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon})$ if $m \geqslant 2\log_\sigma n$.*

Now we can get worst-case guarantees in the search process by adding *Range*, the two-dimensional range search data structure defined in Section 3 for the original LZ-index, requiring $n \log n + o(u \log \sigma)$ extra bits [31]. Occurrences of type 2 can now be found in $O((m + occ_2) \log n)$ worst-case time by using *Range*. Occurrences of type 1 and type 3 are found as for the index of Theorem 1. Existential queries, on the other hand, can be supported by first looking whether there is any occurrence of type 1 (i.e., by looking for P^r in *RevTrie* and then checking whether the corresponding subtree is empty or not) in $O(m)$ time. If there are no occurrences of type 1, we check whether there is any occurrence of type 2 by partitioning the pattern and using *Range* to count the number of occurrences for each partition. This takes $O(m \log n)$ time overall, since we use *Range* just to count. Finally, if there are no occurrences of type 2, we look for occurrences of type 3 in $O(m^2/\epsilon)$ time. Hence, we have the following theorem.

Theorem 2. *Given a text $T[1..u]$ over an alphabet of size σ, there exists a compressed full-text self-index requiring $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for $\sigma = O(\text{polylog}(u))$, any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

1. *locate the occ occurrences of P in T in $O(\frac{m^2}{\epsilon} + (m + occ)\log u + \frac{occ}{\epsilon})$ worst-case time;*
2. *count the number of pattern occurrences in $O(\frac{m^2}{\epsilon} + m \log u + \frac{occ}{\epsilon})$ worst-case time; and*
3. *determine whether P exists in T in $O(\frac{m^2}{\epsilon} + m \log u)$ worst-case time.*

In Section 8 we show that the theorem is valid for the more general case $\log \sigma = o(\log u)$. We leave for Section 7 the study of `display` and `extract` queries on our indices.

5 Using the *xbw* Transform to Represent the *LZTrie*

A different idea to reduce the space requirement of LZ-index is to use the *xbw transform* of Ferragina et al. [13] to represent the *LZTrie*. We show that subpath queries, which are efficiently supported by the *xbw* transform (see Section 2.5), are so powerful that we can carry out the work of both *LZTrie* and *RevTrie* with only the *xbw* representation of *LZTrie*, thus achieving the same result as in Section 4 (always assuming $\sigma = O(\text{polylog}(u))$), yet by very different means. Ferragina et al. [13] have shown how the *xbw* representation can be compressed in order to take advantage of the tree regularities, which can be very important in practice and adds extra value to this representation.

5.1 Index Definition

We represent the LZ-index with the following data structures:

- *xbw LZTrie*: the *xbw* representation [13] of *LZTrie*, where the nodes are lexicographically sorted according to their upward paths in the trie. We store,
  - S_α: the array of symbols labeling the edges of the trie, in the order defined in Section 2.5. In the worst case *LZTrie* has $2n$ nodes (because of the dummy leaves we add, recall Section 2.5). We represent this array by using a data structure for rank and select [17], which are needed to compute the operations on *xbw*. The space requirement is $2n \log \sigma + o(n \log \sigma)$ bits.
  - S_{last}: a bit array such that $S_{last}[i] = 1$ iff the corresponding node in *LZTrie* is the last child of its parent. We represent this array with a data structure for rank and select [36]. The space requirement is at most $2n + o(n)$ bits.
- Balanced parentheses *LZTrie*: the trie of the Lempel-Ziv phrases, implemented by,

- *par*: the balanced parentheses representation [38] of *LZTrie*. In order to index the *LZTrie* leaves with *xbw*, we have to add a dummy child to each, as was explained in Section 2.4. In this way, the trie has $n' \leqslant 2n$ nodes. Non-dummy nodes are marked in a bit vector $B[1..n']$ in the same way as empty nodes are marked in *RevTrie* (see Section 3.4). We represent array B with a data structure for rank and select queries [36]. The space requirement is $2n' + n' + o(n)$ bits, which is $6n + o(n)$ bits in the worst case. This sequence *par* is needed to support some operations which are not supported by the *xbw*, such as $\mathsf{ancestor}(x, y)$ and $\mathsf{depth}(x)$.
- *ids*: the array of LZ78 phrase identifiers in preorder, only for non-dummy nodes (we find the phrase identifier for a given node by using rank_1 on B). This array is represented with the data structure for permutations [37], so that we can compute the inverse permutation ids^{-1} in $O(1/\epsilon)$ time, requiring $(1+\epsilon)n \log n + n + o(n)$ bits (recall Section 2.4).

– *Pos*: a mapping from *xbw* positions to the corresponding *LZTrie* preorder positions (this is a permutation of *LZTrie* preorders). In the worst case there are $2n$ such positions, so the space requirement is $2n \log(2n)$ bits. We can reduce this space to $\epsilon n \log(2n)$ bits by storing in an array Pos' one out of $O(1/\epsilon)$ values of *Pos*, so that $Pos[i]$ can be computed in $O(1/\epsilon)$ time. We need a bit vector Pos_B of $2n + o(n)$ bits indicating which values of *Pos* have been stored. Assume we need to compute the preorder position $Pos[i]$, for a given *xbw* position i. If $Pos_B[i] = 1$, then such preorder position is stored explicitly at $Pos'[\mathsf{rank}_1(Pos_B, i)]$. Otherwise, we simulate a preorder traversal in *xbw* from the node at *xbw* position i, until $Pos_B[j] = 1$, for an *xbw* position j. Each preorder step we perform in *xbw* corresponds to moving to the next opening parenthesis in *par*. Once this j is found, we map to the preorder position $j' = Pos'[\mathsf{rank}_1(Pos_B, j)]$. If d is the number of nodes in preorder traversal from *xbw* position i to *xbw* position j, then $j' - d$ is the preorder position corresponding to the node at *xbw* position i.
We also need to compute Pos^{-1}, which can be done in $O(1/\epsilon^2)$ time under this scheme, requiring $\epsilon n \log(2n)$ extra bits if we use the representation [37] for inverse permutations. However, we can support the computation of Pos^{-1} in $O(1/\epsilon)$ time as follows. For every node such that its *Pos* value has been stored in Pos', we also store the corresponding value of Pos^{-1} in array Pos''. If we want to compute $Pos^{-1}[i]$, we first compute the preorder of the previous node that has been sampled in Pos'', at $j = l\lfloor \frac{i}{l} \rfloor$, where $l = \Theta(1/\epsilon)$. Then, we use the sample value stored at $Pos''[\lfloor \frac{i}{l} \rfloor]$ to map to the *xbw*, and then carry out $i - j$ preorder steps in *xbw*, to find the node corresponding to $Pos^{-1}[i]$. This takes $O(1/\epsilon)$ time in the worst case.

– *Range*: a *range search* data structure in which we store the point k (belonging to phrase identifier k) at coordinate (x, y), where x is the *xbw position* of node for phrase k and y is the *LZTrie preorder position* of node for phrase $k+1$. We use the data structure of Chazelle [10], as for the original LZ-index. The space requirement is $n \log n + O(n \log\log n) = n \log n + o(u \log \sigma)$ bits.

Example 12. See Table 2 for an illustration of the *xbw* of *LZTrie* for the running example, and Fig. 8 for an illustration of the balanced parentheses *LZTrie* and *Pos*.

In Fig. 9 we show the basic resulting navigation scheme following the notation of Section 3.4. The total space requirement is $(2+\epsilon)n \log n + 2n \log \sigma + 11n + O(n \log\log n) + o(n)$ bits, which is $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$.

5.2 Search Algorithm

We depict now the search algorithm for a pattern P of length m.

	0										10										20										30										40										50			
par:	(	(	(	(	)	)	(	(	(	)	)	)	(	(	)	)	(	(	(	)	)	(	(	)	)	(	(	)	)	)	(	(	(	)	)	)	)	(	(	(	(	)	)	)	)	(	(	(	(	)	)	)	)	)
B:	1	1	1	0			1	1	0				1	0			1	1	0			1	0			1	0				1	1	0					1	1	1	0					1	1	1	0					
ids:	0	1	17				3	15					14				4	12				10				16					6	11						2	7	9						5	8	13						
letts:		a	$	Δ			b	r	Δ				l	Δ			r	a	Δ			d	Δ			l	Δ				_	p	Δ					l	a	b	Δ					_	a	p	Δ					
Pos^{-1}:		1	5	4			6	13	24				7	17			8	21	11			22	15			23	18				9	26	20					2	16	10	14					3	25	12	19					

i :	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*	*25*	*26*
$Pos[i]$:	1	19	23	3	2	4	7	9	16	21	11	25	5	22	13	20	8	15	26	18	10	12	14	6	24	17

Fig. 8. Balanced-parentheses representation of *LZTrie* for the running example, with dummy leaves added in order to index the (original) leaves with the *xbw* representation. Bit vector B marks each dummy node with a 0. Given node x, the corresponding phrase identifier can be computed as $ids[\mathsf{rank}_1(B, \mathsf{preorder}(x))]$. We also show the *Pos* mapping (from *xbw* positions, see Table 2, to *LZTrie* preorders), and the Pos^{-1} mapping (from *LZTrie* preorders to *xbw* positions).

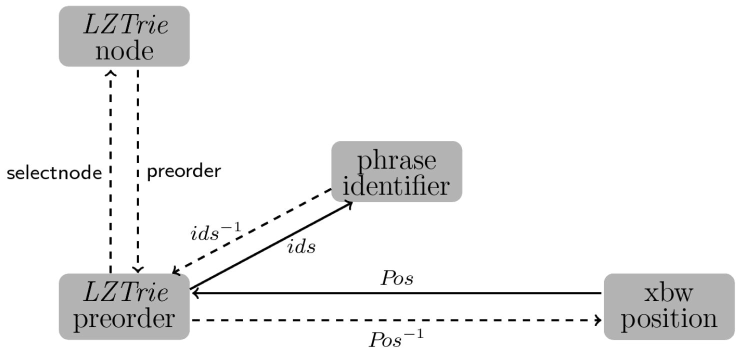


Fig. 9. Basic navigation scheme using the *xbw* representation of the *LZTrie*.

Occurrences of Type 1. Recall from Section 3.3 that first we need to find all the phrases having P as a suffix. To do this we perform a subpath query with P^r on the *xbw* representation of *LZTrie*, simulating in this way the work done on *RevTrie* in the original scheme, in $O(m)$ time. Suppose that we obtain the interval $[x_1..x_2]$ in the *xbw* of *LZTrie*, corresponding to all the nodes whose phrase ends with P. In other words, the interval $[x_1..x_2]$ contains the roots of the subtrees containing the nodes we are looking for to find occurrences of type 1. For each position $i \in [x_1..x_2]$, we can get the corresponding preorder in the parentheses representation using $Pos(i)$, which takes $O(1/\epsilon)$ time, and then $\mathsf{selectnode}(Pos(i))$ over *par* yields the node position. As in the worst case this mapping is carried out occ_1 times, the overall time is $O(\frac{occ_1}{\epsilon})$. Finally, we traverse the subtrees of these nodes in *par* and report all the identifiers found, in constant time per occurrence as done with the usual LZ-index.

Occurrences of Type 2. To find occurrences of type 2, for every possible partition $P[1..i]$ and $P[i+1..m]$ of P, we traverse the *xbw* from the root, using operation $\mathsf{child}(x, \alpha)$ with the symbols of $P[i+1..m]$. This takes $O(m^2)$ time overall for the $m-1$ partitions of P. In this way we are simulating the work done on *LZTrie* when searching for occurrences of type 2 in the original scheme. Once this is found, say at *xbw* position j, we switch to the preorder tree (parentheses) using $\mathsf{selectnode}(Pos(j))$ over *par*, to get the node v_{lz} whose subtree has preorder interval $[y_1..y_2]$ of all the nodes whose strings start with $P[i+1..m]$. This takes overall $O(\frac{m}{\epsilon})$ time, for the $m-1$ partitions of P. Next we perform a subpath query for $P[1..i]$ in *xbw*, and get the *xbw* interval $[x_1..x_2]$

of all the nodes whose strings finish with $P[1..i]$ (actually we have to perform $x_1 \leftarrow \mathsf{rank}_1(S_{last}, x_1)$ and $x_2 \leftarrow \mathsf{rank}_1(S_{last}, x_2)$ to avoid counting the same node multiple times, see [13]). This also takes $O(m^2)$ time overall. Finally, we search the *Range* data structure for $[x_1..x_2] \times [y_1..y_2]$ to obtain all phrase identifiers t such that phrase B_t finishes with $P[1..i]$ and phrase B_{t+1} starts with $P[i+1..m]$, in $O((m + occ)\log n)$ time overall.

Occurrences of Type 3. For occurrences of type 3, one proceeds mostly as with the original *LZTrie* (navigating the *xbw* instead), so as to find all the nodes equal to substrings of P in $O(m^2)$ time. Then, for each maximal concatenation of phrases $P[i..j] = B_t \ldots B_\ell$, we must check whether phrase $B_{\ell+1}$ starts with $P[j + 1..m]$ and whether phrase B_{t-1} finishes with $P[1..i - 1]$. The first check can be done in $O(1/\epsilon)$ time by using ids^{-1}: as we have searched for all substrings of P in the trie, we know the *LZTrie* preorder interval of the descendants of $P[j + 1..m]$, and thus we check whether the node at preorder position $ids^{-1}(\ell + 1)$ belongs to that interval. The second check can be done in $O(1/\epsilon)$ time, by determining whether $t - 1$ lies in the *xbw* interval of $P[1..i - 1]$ (that is, B_{t-1} finishes with $P[1..i - 1]$). For this, we need Pos^{-1}, so that the position is $Pos^{-1}(ids^{-1}(t - 1))$.

Summarizing, occurrences of type 1 cost $O(m + \frac{occ}{\epsilon})$ time, occurrences of type 2 cost $O(m^2 + \frac{m}{\epsilon} + (m + occ)\log n)$ time, and of type 3 cost $O(\frac{m^2}{\epsilon})$ time. Thus, we have achieved Theorem 2 again with radically different means. The same complexities are also achieved for `count` and `exists` queries. We can also obtain a version requiring $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits and $O(m^2)$ average reporting time if $m \geqslant 2\log_\sigma n$ (as in Theorem 1), if we search for the occurrences of type 2 by using a checking procedure similar to that used to check phrases $t - 1$ and $\ell + 1$ for the occurrences of type 3.

6 Faster and Still Small LZ-indices

In Section 4, we have shown how to use suffix links in *RevTrie* to reduce the space requirement of the LZ-index. Russo and Oliveira [44] show how to use suffix links to reduce the locating time of their LZ-index to $O((m + occ)\log u)$, but not to reduce the space of their index. On the other hand, Ferragina and Manzini [16] combine the backward-search concept with a Lempel-Ziv-based scheme to achieve optimal $O(m + occ)$ locating time, without restrictions on m or occ. Yet, their index is even larger, requiring $O(uH_k(T)\log^\gamma u)$ bits of space, for any constant $\gamma > 0$.

In this section, we use suffix links to speed up occurrences of type 2, using an idea similar to that of [44], and we find occurrences of type 3 as a particular case of occurrences of type 2, using a similar idea to that of [16]. In this way we manage to avoid the $O(m^2)$ term in the locating complexity of the LZ-index, achieving the same locating time as [44], while reducing their space requirement of $(5 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits.

6.1 Index Definition

We build basically on the LZ-index of Theorem 1, composed of *LZTrie*, *RevTrie*, and the R mapping (compressed using suffix links φ). We add to *LZTrie* the data structure of Jansson et al. [25] to compute *level ancestor queries*, $\mathsf{LA}(x, d)$, which returns the ancestor at depth d of node x. This requires $o(n)$ extra bits and supports LA queries in constant time. Therefore, the overall space requirement of the three above data structures is $(1 + \epsilon)uH_k + o(u \log \sigma)$ bits.

To avoid the $O(m^2)$ term in the locating complexity, we should avoid occurrences of type 3, since they make us check the $O(m^2)$ possible candidates. We cannot use the same procedure as for occurrences of type 2 (using the *Range* data structure) because *LZTrie* is only able to index whole phrases, not text suffixes. Then, by using *LZTrie* to query the *Range* data structure, we are only able to return the phrases starting with a given suffix $P[i+1..m]$ of the pattern, and therefore we can find only occurrences spanning two consecutive phrases (i.e., occurrences of type 2).

Hence we add the *alphabet friendly FM-index* [17] 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 $O(\log^{1+\epsilon} u \, \frac{\log \sigma}{\log \log u})$, 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* [9] of T ($bwt(T)$ for short), it can be (conceptually) thought of as the suffix array SA_T of T. Recall that, in a suffix array, a given interval corresponds to a lexicographic interval of the text suffixes. The AF-FMI(T) indexes text suffixes. In particular, we will be interested in those suffixes that are aligned with the LZ78 phrase beginnings. By using this structure to query the *Range* data structure (instead of using *LZTrie*) we will be able to find those text suffixes that are aligned with LZ78 phrases and have $P[i+1..m]$ as a prefix. Thus, $P[i+1..m]$ can span more than two consecutive phrases, and therefore we will consider occurrences of type 3 as a special case of occurrences of type 2.

To find occurrences spanning several phrases we re-define *Range*, the data structure for 2-dimensional range searching. Now it will operate on 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, i.e. $SA_T[i'] = p$ holds.

Suppose that we search for a given string s_2 in AF-FMI(T) and get the interval $[i_1, i_2]$ in the $bwt(T)$ (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 the next phrase is aligned with an occurrence of s_2 in T.

We transform the grid $[1..u] \times [1..n]$ indexed by *Range* into an equivalent grid $[1..n] \times [1..n]$ by defining a bit vector $V[1..u]$, which indicates (with a 1) which positions of AF-FMI(T) point to the beginning of an LZ78 phrase. We represent V with the data structure of [42] supporting rank queries, and using $uH_0(V) + o(u) \leqslant n \log \frac{u}{n} + o(u) \leqslant \frac{u \log \log u}{\log_\sigma u} + o(u) = o(u \log \sigma)$ bits of storage (recall $n \leqslant u/\log_\sigma u$). Thus, instead of storing the point (i', j') as in the previous definition of *Range*, we store the point $(\mathsf{rank}_1(V, i'), j')$. The search of the previous paragraph now becomes $[\mathsf{rank}_1(V, i_1), \mathsf{rank}_1(V, i_2)] \times [j_1, j_2]$.

As there is only one point per row and column of *Range*, we can use the data structure of Chazelle [10], which can be implemented by using $n \log n + O(n \log \log n) = uH_k(T) + o(u \log \sigma)$ bits [31]. 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 $0 < \epsilon < 1$.

6.2 Search Algorithm

For `exists` and `count` queries we can achieve $O(m)$ time by just using the AF-FMI(T). We focus now on `locate` queries. Assume that $P[1..m] = p_1 \ldots p_m$, for $p_i \in \Sigma$. As explained, we need to consider only occurrences of P in T of type 1 and 2. Those of type 1 are found just as for the original LZ-index, in $O(m + \frac{occ_1}{\epsilon})$ time. The rest of the section is devoted to those 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.

If the search in *RevTrie* for $P^r[1..i]$ yields the preorder interval $[j_1, j_2]$, and the search for $P[i+1..m]$ in AF-FMI(T) yields interval $[i_1, i_2]$, the two-dimensional range $[\mathsf{rank}_1(V, i_1), \mathsf{rank}_1(V, i_2)] \times [j_1, j_2]$ in *Range* yields all pattern occurrences for the given partition of P. For every pattern occurrence we get a point (i', j') from *Range*. The corresponding phrase identifier can be found as $t = ids(R(j'))$, to finally report a pattern occurrence $[\![t, i]\!]$.

Overall, occurrences of type 2 are found in $O((m + occ_2)\log n)$ time. Yet, we still have to show how to find efficiently the intervals in AF-FMI(T) and in *RevTrie*.

The $m-1$ intervals for $P[i+1..m]$ 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 if $\sigma = O(\text{polylog}(u))$ [16]: in the first step we find the BWT interval for p_m, then we find the interval for occurrences of $p_{m-1}p_m$, then $p_{m-2}p_{m-1}p_m$, and so on to finally find the interval for $p_2 \ldots p_m = P[2..m]$.

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 indices of Section 4. 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 $p_{m-1}p_{m-2}\ldots p_1$, $p_{m-2}\ldots p_1, \ldots,$ and p_1 in *RevTrie*. Note that every $p_j \ldots p_1$ is the longest proper suffix of $p_{j+1}p_j \ldots p_1$. Suppose that we successfully search for $P^r[1..m-1] = p_{m-1}p_{m-2}\ldots p_1$, 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 $p_{m-2}\ldots p_1$ we only need to follow suffix link $\varphi(i')$ (which takes $O(1)$ 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 [44].

In what follows, we explain the way we implement descend and suffix walk in our data structure. However, we must prove a couple of properties for *RevTrie* in order to be able to apply this method. First, we know that every non-empty node in *RevTrie* has a suffix link (see Lemma 4), yet we need to prove that every *RevTrie* node (including empty-non-unary nodes) has also a suffix link.

Lemma 10. *Every empty non-unary node in RevTrie has a suffix link.*

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

The descent process in *RevTrie* will be a little bit different from the one described in the proof of Lemma 3. This time, we are going to reuse the work done for a string already searched for in *RevTrie*, so we have to be sure that every time we arrive at a *RevTrie* node, the string represented by that node matches the corresponding pattern prefix (the usual skipping process of a Patricia tree does not ensure that). Thus, 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 the whole label in time linear in its length.

Lemma 11. *Any edge label of length l in RevTrie can be extracted in $O(l)$ time.*

Proof. Assume that we are at node v_r in *RevTrie*, and want to extract the label for edge $e_{v_r v'_r}$ between nodes v_r and v'_r in *RevTrie*. Since we arrive at a node in *RevTrie* by descending from the root, the length of the string represented by a given node can be computed by summing up the skips we have seen in the descent. Let l_{v_r} and $l_{v'_r}$ be the length of strings represented by nodes v_r and v'_r, respectively. Then, $l_{v'_r} - l_{v_r}$ is the length of the label of edge $e_{v_r v'_r}$.

If we assume that node v'_r has preorder j_1 in *RevTrie*, we can access the *LZTrie* node from which to start the extraction of the label by $v'_{lz} = \mathsf{LA}(R[j_1], l_{v'_r} - l_{v_r})$, in constant time [25]. The label of $e_{v_r v'_r}$ is the label of the v'_{lz}-to-root path. Notice that with the level-ancestor query on *LZTrie*, we avoid extracting the string represented by node v_r in *RevTrie*, as it has been already extracted before descending to v_r.

In the case where v'_r is an empty node, recall that the corresponding value $R[j_1]$ is undefined. However, just as in the proof of Lemma 3, we can use any non-empty node within the subtree of v'_r to map to the *LZTrie*. For instance, we can use the next non-empty node within the subtree of v'_r: let $j_2 = \mathsf{rank}_1(B, j_1) + 1$, then the length of the corresponding string can be computed as $\mathsf{depth}(R[j_2])$ in *LZTrie*, and we compute $v'_{lz} = \mathsf{LA}(R[j_2], \mathsf{depth}(R[j_2]) - l_{v_r})$, to finally extract the edge label by moving to the parent $l_{v'_r} - l_{v_r}$ times. □

Thus, we search *RevTrie* as in a normal trie, comparing *every* symbol as we descend, without skipping as is done in Lemma 3. In this way, every time we arrive at a *RevTrie* node, the string represented by that node will match the corresponding prefix of the pattern.

Previously we showed that it is possible to search for all strings $P^r[1..i]$ in $O(m)$ time, assuming that $P^r[1..m-1]$ exists in *RevTrie* (therefore all the $P^r[1..i]$'s exist in *RevTrie*). The general case is as follows. Let $P^r[1..m-1] = p_{m-1} \dots p_1$ be the longest string that we need to search for in *RevTrie*. We define three integer indices on $P^r[1..m-1]$, which guide the search:

i_1, which marks the beginning of the pattern suffix we are currently searching for. It is initialized at 1 since we start searching for $p_{m-1}p_{m-2} \dots p_1$;

i_2, which indicates the current symbol in the pattern, which is being compared with a symbol in an edge label, with the aim of descending to a child of the current node. Notice that $(P^r)[i_1..i_2-1]$ is the part of the current pattern that has been matched with the edge labels of *RevTrie*; and

i_3, which delimits the string corresponding to the current node, which represents string $(P^r)[i_1..i_3]$ in *RevTrie*. Thus $(P^r)[i_3+1..i_2-1]$ will be the part of the pattern that has been compared with the label of the edge leading to the node we are trying to descend to.

Our descend and suffix walk will be composed of three basic operations: *descend*, *suffix*, and *retraverse*.

Descend. We start searching for $p_{m-1}p_{m-2} \dots p_1$ from the *RevTrie* root, using the method of Lemma 11 and using i_2 to indicate the current symbol being compared in the descent. Every time we descend to a non-empty-unary child node (after matching all the characters of an edge), we set $i_3 \leftarrow i_2$ and continue descending in the same way from this node. If, when trying to descend to a child node, we find an empty-unary node (which was added because of the skips in *RevTrie*, see

Section 3.4), the index i_3 is not updated as explained before. In this case, we continue the descent with i_2 from the empty-unary node, using Lemma 11.

Suffix. Now assume that, being at current node v_r (with preorder j_1 in *RevTrie* and representing string ax, for $a \in \Sigma$, $x \in \Sigma^*$), we cannot descend to a child node v'_r (with preorder j_2 in *RevTrie* and representing string $axyz$, for $y, z \in \Sigma^*$, such that $|yz| > 0$). Let $e_{v_r v'_r}$ be the edge between nodes v_r and v'_r, with label yz, where $y = (P^r)[i_3+1..i_2-1]$ and $(P^r)[i_2] \neq z_1$. Hence there are no phrases ending with $P^r[1..m-i_1]$.

Then, we go on to consider the next suffix $P^r[1..m-i_1-1]$. To reuse the work done up to node v_r (i.e. $(P^r)[i_1..i_3] = ax$), we follow the suffix link to get the node $\varphi(j_1)$ representing string x, setting $i_1 \leftarrow i_1 + 1$.

Retraverse. We have reused the work up to x, but we had actually worked up to xy. Notice that suffix xy certainly exists in *RevTrie*, yet it could be represented by an empty unary node which has been compressed in an edge. Therefore, from node $\varphi(j_1)$ we descend using $y = (P^r)[i_3+1..i_2-1]$. The edge $e_{v_r v'_r} = yz$ could be split into a path of several nodes between nodes $\varphi(j_1)$ and $\varphi(j_2)$. As substring y has been already checked in the previous step, the descent from node $\varphi(j_1)$ is done by skipping and checking only the first symbols of the edge labels (advancing i_3 accordingly as we reach new nodes). If we find an empty unary node when trying to descend from node v''_r to the next node, we jump directly to the position of the next non-empty unary node (with preorder j_3) and then compute the length l of the string represented by that node.

For this direct jump we need a bit vector E marking the empty unary nodes, in preorder. We preprocess E with a data structure supporting rank and select, so this requires $n' + o(n')$ extra bits. The node with preorder j_3 can be found by using rank and select on E. The length l can be computed as the sum of the length of the current node plus $n_e \cdot \log u$, where n_e is the number of empty unary nodes between the current node and the one with preorder j_3 (which can be computed as the number of 1s between the corresponding positions in E), and $\log u$ comes from the skips of empty unary nodes (recall Section 3.4). If $l > |xy|$, we resume the *suffix* mode from v''_r. Otherwise, we stay in the *retraverse* mode from the node with preorder j_3. This process is carried out till string y is fully consumed, and then we resume the *descend* mode from the corresponding node.

After we find the first suffix $P^r[1..i]$ in *RevTrie* (if any), we are sure that every suffix of it also exists in *RevTrie* (because this trie is suffix closed). The nodes corresponding to these suffixes are found by following suffix links.

Lemma 12. *Given a string P of length m, we can search for strings $P^r[1..i]$, for $1 \leqslant i < m$, in RevTrie in $O(m)$ time.*

Proof. Consider the method just described. Indices i_1, i_2, and i_3 grow from 1 to at most m. For every constant-time action we carry out, at least one of those indices increases. Thus the total work is $O(m)$. □

Therefore, we have proved:

Theorem 3. *Given a text $T[1..u]$ over an alphabet of size σ, 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 $0 < \epsilon < 1$. Given a search pattern $P[1..m]$, this index is able to:*

1. *locate the occ occurrences of P in T in $O((m + \frac{occ}{\epsilon}) \log u)$ worst-case time;*

2. count pattern occurrences in $O(m)$ worst-case time; and
3. determine whether P exists in T in $O(m)$ worst-case time.

7 Optimal Displaying of Text Substrings

7.1 Reporting Text Positions with LZ-index

As we said before, the original LZ-index is able to report occurrences in the format $[\![t, \texttt{offset}]\!]$, where t is the phrase in which the occurrence starts and `offset` is the distance between the beginning of the occurrence and the end of the phrase. The same happens for our indices of Sections 4, 5, and 6.

However, we can report occurrences as *text positions* by adding a bit vector $TPos[1..u]$ that marks with a `1` the text positions corresponding to the phrase beginnings. Thus, there are n `1`s in $TPos$. Given a text position i, $\mathsf{rank}_1(TPos, i)$ is the phrase number i belongs to. Given a phrase identifier j, $\mathsf{select}_1(TPos, j)$ yields the text position at which the j-th phrase starts. Therefore, given an occurrence in the format $[\![t, \texttt{offset}]\!]$, the text position for that occurrence can be computed as $\mathsf{select}_1(TPos, t+1) - \texttt{offset}$. Such $TPos$ can be represented with $uH_0(TPos) + o(u) \leqslant n \log \frac{u}{n} + o(u) \leqslant \frac{u \log\log u}{\log_\sigma u} + o(u) = o(u \log \sigma)$ bits [42] (recall $n \leqslant u/\log_\sigma u$).

The algorithm for `extract` queries (of whole LZ78 phrases) described in Section 3.3 can also be used on the indices of Theorems 1, 2 and 3, yet this time providing the text positions from which to extract (rather than the phrase identifiers), since these positions can be transformed into phrase identifiers by using data structure $TPos$. As the *Node* data structure is simulated by using ids^{-1}, it takes $O(\ell(1 + \frac{1}{\epsilon \log_\sigma \ell}))$ time to extract any text substring of length ℓ. This is because we perform ℓ parent operations to get the ℓ symbols we want to display, and we must pay $O(1/\epsilon)$ to use ids^{-1} each time we go on to extract the next phrase, which in the (very) worst case is done $O(\ell/\log_\sigma \ell)$ times.

To extract the text with *xbw*-based LZ-index of Section 5, we use $TPos$ to transform the text positions into phrase identifiers, and then we use ids^{-1} to find the preorder position of the corresponding phrase, to finally map to the *xbw* representation of *LZTrie* by using Pos^{-1} in $O(1/\epsilon)$ time. Then we move to the parent in the *xbw*, displaying the corresponding symbol stored in S_α. When we reach the tree root, we use ids^{-1} again to consider the next phrase, and map to the *xbw* again. The time is therefore $O(\ell(1 + \frac{1}{\epsilon \log_\sigma \ell}))$.

We can avoid the restriction of displaying only whole phrases by adding a data structure for *level-ancestor* (LA) queries on *LZTrie*. The data structure [25] builds on DFUDS, allows constant time computation of operation LA, and requires $o(n)$ extra bits of space. Thus, the part of a phrase that we do not need to display is skipped by using the appropriate LA query. Yet, the displaying time is not optimal, since we work $O(1)$ per extracted symbol and on a RAM we are able to handle $\Theta(\log u)$ bits per access, which means $\Theta(\log u/\log \sigma) = \Theta(\log_\sigma u)$ symbols per access.

7.2 Achieving Optimal Extracting Time

We describe a technique that can be plugged to any of the indices proposed in Sections 4, 5 and 6, for displaying any text substring $T[i..i+\ell-1]$, in optimal $O(1 + \ell/\log_\sigma u)$ time. A compressed data structure [46] to display any text substring of length $\Theta(\log_\sigma u)$ in constant time, turns out to have similarities to the LZ-index. We take advantage of this similarity to plug it into our indices, with some modifications, and obtain improved time to display text substrings. In [46], the authors added auxiliary data structures of $o(u \log \sigma)$ bits to *LZTrie* to support this operation efficiently. Given a position i in the text, we first find the phrase including the position i by using $\mathsf{rank}_1(\mathit{TPos}, i)$, and then find the node of *LZTrie* that corresponds to the phrase using *Node* (that is, the corresponding implementation of it). Then displaying a phrase is equivalent to outputting the path going from the node to the root of *LZTrie*. The auxiliary data structure, of size $O(n \log \sigma) = o(u \log \sigma)$ bits, permits outputting the path by chunks of $\Theta(\log_\sigma u)$ symbols in $O(1)$ time per chunk. As explained before, we can also display not only whole phrases, but any text substring within this complexity. Thus the displaying can start backwards from anywhere in a phrase and, of course, it can stop at any point as well.

We modify this method to plug it into our indices. In their original method [46], if more than one consecutive phrases has length less than $(\log_\sigma u)/2$ each, their phrase identifiers are not stored. Instead the substring of the text including those phrases is stored without compression. This guarantees efficient displaying without increasing the space requirement. However this will cause the problem that we cannot find patterns including those phrases. Therefore in our modification we store, for these short phrases, both the phrases themselves and their phrase identifiers. The search algorithm remains as before. To decode short phrases we can just output the explicitly stored substring including the phrases. For each phrase with length at most $(\log_\sigma u)/2$, we store a substring of length $\log u$ containing the phrase. Because there are at most $O(\sqrt{u})$ such phrases in the text (recall that all LZ78 phrases are different), we can store all these substrings in $O(\sqrt{u} \log u) = o(u)$ bits. These auxiliary structures work as long as we can convert a phrase identifier into a preorder position in *LZtrie* (that is, compute ids^{-1}) . Hence they can be applied to all the data structures in Sections 4, 5, and 6.

Theorem 4. *The indices of Theorem 1 and Theorem 2 (and also those of Sections 5 and 6) can be adapted to extract a text substring of length* ℓ *surrounding any text position in optimal* $O(1+\frac{\ell}{\epsilon \log_\sigma u})$ *worst-case time, using only* $o(u \log \sigma)$ *extra bits of space, for any* $0 < \epsilon < 1$.

8 Handling Larger Alphabets

For simplicity, throughout this paper we have assumed $\sigma = O(\mathrm{polylog}(u))$, or equivalently $\log \sigma = O(\log \log u)$. Here we study the cases $\log \sigma = o(\log u)$ and $\log \sigma = \Theta(\log u)$.

8.1 The Case $\log \sigma = o(\log u)$

As long as $\log \sigma = o(\log u)$ holds, we can still have $k = o(\log_\sigma u) > 0$, while it also holds that $n \log n = uH_k(T) + o(u \log \sigma)$ [28]. Therefore, the space requirements of the indices of Theorems 1 to 3 stay the same.

Index of Section 4. The data structure of [17], which we use to represent *letts* and array S_W, has a time complexity of $O(\frac{\log \sigma}{\log \log u})$ for rank and select queries; thus, we lose the constant time for operations $\mathsf{child}(x, \alpha)$ and $\varphi'(x, \alpha)$ on the tries, which would increase the time complexity of the whole index. Nevertheless, we can represent *letts* with the (more complicated) data structure used in [8], thus ensuring constant time for $\mathsf{child}(x, \alpha)$ for any σ, and retaining the same time complexity in our theorems. In the case of S_W we can use the following scheme, which is a variant of that used in [13] to achieve constant-time rank over the sequence, requiring $n \log \sigma + o(u \log \sigma)$ bits of space.

Let S' be a $\sigma \times n$ binary matrix, such that $S'[a, i] = \mathtt{1}$ if and only if $S_W[i] = a$ holds. Thus, $\mathsf{rank}_a(S_W, i)$ can be computed as the number of $\mathtt{1}$s within the a-th row of S', up to column i. We represent S' as a linear bit vector S'' of σn bits. Since there are n $\mathtt{1}$s in this bit vector, we use the data structure of [42] that requires $\log \binom{\sigma n}{n} + o(n) + O(\log\log \sigma n)$ bits of space, which is $n \log \sigma + O(n)$ bits. We add an array counting the number of $\mathtt{1}$s up to the beginning of each row in S', using $\sigma \log n$ extra bits. By using this representation, we are able to compute $\mathsf{rank}_{\mathtt{1}}(S'', i)$ in constant time, but only if we know that $S''[i] = \mathtt{1}$ holds [42]. In our representation, this means that we can compute $\mathsf{rank}_a(S_W, i)$ only if we know that $S_W[i] = a$ holds. However, this is not the case, since the rank_a in Eq. (1) is carried out over the first position of S_W corresponding to the Weiner-link symbols of a node, which can store any possible symbol, not necessarily a.

We devise the following data structure in order to find the position of symbol a within the Weiner-link symbols of a *RevTrie* node v with preorder i. Given the portion of array S_W corresponding to the symbols for the Weiner links of node v, we can construct a DFUDS [8] directory on these symbols in order to compute, in $O(1)$ time, the position of any symbol within this segment. The space for the directory is $d \log \sigma$ bits, where d is the number of Weiner links defined for node v. We define array D of $n \log \sigma$ bits, storing the DFUDS directory D_v for every *RevTrie* node in preorder. Directory D_v is aligned with the positions in array S_W for node v. Let $i_2 \leftarrow \mathsf{rank}_0(V_W, \mathsf{select}_1(V_W, i+1)) + 1$ be the starting position in S_W for the Weiner-link symbols of v. Let j be the position of symbol a within the symbols of v, yielded by D_v in $O(1)$ time (D_v also allows us to know whether or not symbol a exists within the symbols of node v). Then, we know that $S_W[i_2 + j] = a$ holds, hence the rank_a in Eq. (1) must be computed up to position $i_2 + j$, intead of just i_2.

None of the remaining data structures of the index are affected by the alphabet size. As a result, Theorem 2 can be extended for the case $\log \sigma = o(\log u)$, rather than only for $\sigma = O(\mathrm{polylog}(u))$.

Index of Section 5. The times for the operations on the *xbw* representation of *LZTrie* are affected by the alphabet size, depending on the representation used for S_α. If we use the data structure of Golynski et al. [20], occurrences of type 1 are found in $O(m \log\log \sigma + \frac{occ}{\epsilon})$ time, because of the subpath query we perform on *LZTrie*; occurrences of type 2 are found in $O(m^2 \log\log \sigma + \frac{m}{\epsilon} + (m + occ)\log n)$ time, where the first term comes from searching for the $m-1$ partitions of P in *xbw*; and occurrences of type 3 are found in $O(m^2 \log\log \sigma + \frac{m^2}{\epsilon})$, where the first term comes from searching for the $O(m^2)$ pattern substrings in the *xbw* representation of *LZTrie*. Overall, the time for `locate` is $O(m^2(\frac{1}{\epsilon} + \log\log \sigma) + (m + occ)\log u)$. We can also replace $O(\log\log \sigma)$ for $O(\frac{\log \sigma}{\log\log u})$ in all these figures.

Index of Section 6. For this index the only affected part is the Alphabet-Friendly FM-index, AF-FMI(T), which still has a space requirement of $uH_k(T) + o(u \log \sigma)$ bits of space. The counting time is increased to $O(m(1 + \frac{\log \sigma}{\log\log u}))$. Thus, the time for `locate` of this version of LZ-index now becomes $O(m(1 + \frac{\log \sigma}{\log\log u}) + (m + \frac{occ}{\epsilon})\log u)$, which is still $O((m + \frac{occ}{\epsilon})\log u)$, the same as stated by Theorem 3, since $m\frac{\log \sigma}{\log\log u} = O(m \log u)$. The counting time, on the other hand, now becomes $O(m(1 + \frac{\log \sigma}{\log\log u}))$. Thus, we have a more general version of Theorem 3:

Theorem 5. *Given a text* $T[1..u]$ *over an alphabet of size* σ*, there exists a compressed full-text self-index requiring* $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ *bits of space, for any* $k = o(\log_\sigma u)$*, any* $0 < \epsilon < 1$*, and such that* $\log \sigma = o(\log u)$*. Given a search pattern* $P[1..m]$*, this index is able to:*

1. *locate the occ occurrences of P in T in $O((m + \frac{occ}{\epsilon}) \log u)$ worst-case time;*
2. *count pattern occurrences in $O(m(1 + \frac{\log \sigma}{\log \log u}))$ worst-case time;*
3. *determine whether P exists in T in $O(m(1 + \frac{\log \sigma}{\log \log u}))$ worst-case time; and*
4. *extract any text substring of length ℓ in $O(\ell/(\epsilon \log_\sigma u))$ worst-case time.*

8.2 The Case $\log \sigma = \Theta(\log u)$

For the case $\log \sigma = \Theta(\log u)$, because of Lemma 1 we have that $n \log n = uH_k(T) + O(u(1+k \log \sigma))$ bits of space, which is $\Theta(u \log \sigma)$ even for $k = 1$. Thus, high-order compression is lost. For $k = 0$ the space is $uH_0(T) + o(u \log \sigma)$ bits of space, so zero-order compression is retained. On the other hand, all the time complexities obtained for the case $\log \sigma = o(\log u)$ are valid for this larger σ. However, it has been shown that the empirical-entropy model is not adequate for such a large alphabet [19].

9 Conclusions and Future Work

We have improved the overall performance of LZ-indices, achieving stronger compressed self-indices based on the Lempel-Ziv compression algorithm [48]. We have reduced the space of Navarro's LZ-index [39] to about a half, achieving $(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space to index a text $T[1..u]$ with k-th order empirical entropy H_k, for any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$. Our indices are able to search for the *occ* occurrences of a pattern $P[1..m]$ in T in $O(\frac{m^2}{\epsilon} + (m + occ) \log u)$ worst-case time, as well as extracting any text substring of length ℓ in optimal $O(\frac{\ell}{\epsilon \log_\sigma u})$ time. Thus, we achieve the same locating time as the index of Kärkkäinen and Ukkonen [27], yet with a much smaller index which does not need the text to operate. We also showed how the space can be squeezed to $(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits, with $O(m^2)$ average-case search time if $m \geqslant 2 \log_\sigma n$. This space approaches, as closely as desired, the optimal $uH_k(T)$ under the k-th order empirical entropy model for all k. However, this index does not provide worst-case guarantees at search time. Thus, ours are the smallest existing compressed self-indices based on Lempel-Ziv compression.

We also showed how to use an LZ-index to achieve $O((m+occ) \log u)$ time to locate the pattern occurrences, requiring $(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ bits of space. This is much less than the space required by other LZ-indices having the same search time.

Thus, we have achieved LZ-indices with space requirements ranging from $(1+\epsilon)$ to $(3+\epsilon)$ times the empirical entropy of the text (plus lower-order terms), with different achievements in the time complexities according with the space requirement of the index. These indices are very competitive with state-of-the-art indices, both in time and space requirement.

The most basic problems for compressed self-indices are that of searching and reproducing the text. However, there are many other functionalities that a self-index must provide in order to be fully useful, as for example the space-efficient construction of the indices, secondary-memory capabilities (in cases where the text is so huge that the corresponding compressed self-index does not fit in main memory), dynamic capabilities, and allowing more complex queries on the text (such as regular-expression and approximate searching).

Constructing the indices with little space is an important research topic regarding their practicality [24, 23, 2, 32]. It has been shown [5] that all the indices defined in this paper can be constructed without requiring any extra space on top of the space of the index itself, which adds extra value to our results. Also, it has been shown that the LZ-index can be efficiently handled on secondary storage [3], by means of adding redundancy to the index to avoid most random accesses. This provides a very promising alternative, yet an interesting question is whether we can use techniques similar to those of this paper to reduce the added redundancy. It has also been shown that the LZ-indices (in particular the ILZI of [44]) are adequate for approximate string matching [43].

A very important aspect is that of the practical implementations of compressed indices, as many theoretical indices are proposed but never implemented. The *Pizza&Chili Corpus* [18] provides practical implementations of compressed indices, as well as some example texts. To show the practicality of our approach, there are currently in the site some implementations of reduced schemes of LZ-index, based on ideas which are similar to the ones described in this paper. These indices have shown to be very competitive against others [4, 12], specifically for `locate` and `extract` queries. We hope to achieve further results along this line.

Finally, the results obtained about succinct representation of suffix and Weiner links are of independent interest and could find applications in other cases, such as compressed suffix trees.

References

1. A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
2. D. Arroyuelo and G. Navarro. Space-efficient construction of LZ-index. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 3827, pages 1143–1152, 2005.
3. D. Arroyuelo and G. Navarro. A Lempel-Ziv text index on secondary storage. In *Proc. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 83–94, 2007.
4. D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices. Technical Report TR/DCC-2008-9, Department of Computer Science, University of Chile, 2008. `http://www.dcc.uchile.cl/TR/2008/TR_DCC-2008-009.pdf`. Submitted.
5. D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. Technical Report TR/DCC-2009-2, Department of Computer Science, University of Chile, 2009. `http://www.dcc.uchile.cl/TR/2009/TR_DCC-20090313-002.pdf`. Submitted.
6. D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 319–330, 2006.
7. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.
8. D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
9. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
10. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
11. D. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 383–391, 1996.
12. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice! *ACM Journal of Experimental Algorithmics*, 13:article 12, 2009. 30 pages.
13. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc. 46th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.
14. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, 2000.
15. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In *Proc. 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 269–278, 2001.
16. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.
17. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):article 20, 2007.
18. P. Ferragina and G. Navarro. Pizza&Chili Corpus — Compressed indexes and their testbeds, 2005. `http://pizzachili.dcc.uchile.cl`.

19. T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.
20. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.
21. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.
22. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
23. W.-K. Hon, T. W. Lam, K. Sadakane, W.-K. Sung, and M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.
24. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 251–260, 2003.
25. J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.
26. J. Kärkkäinen. *Repetition-based text indexes*. PhD thesis, Dept. of CS, University of Helsinki, Finland, 1999.
27. J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*, pages 141–155, 1996.
28. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
29. A. Lempel and J. Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.
30. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
31. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
32. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008. 38 pages.
33. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
34. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
35. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
36. J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
37. J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 2719, pages 345–356, 2003.
38. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
39. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
40. G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13:article 2, 2009. 49 pages.
41. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
42. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.
43. L. Russo, G. Navarro, and A. Oliveira. Approximate string matching with Lempel-Ziv compressed indexes. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 264–275, 2007.
44. L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 5(3):501–513, 2007.
45. K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
46. K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239, 2006.
47. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.
48. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.