Reducing the Space Requirement of LZ-index*

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

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

[2] 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. The LZ-index is a *compressed full-text self-index* able to represent a text $T_{1\dots u}$, over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, using $4uH_k(T)+o(u\log\sigma)$ bits for any $k=o(\log_\sigma u)$. It can report all the *occ* occurrences of a pattern $P_{1\dots m}$ in T in $O(m^3\log\sigma+(m+occ)\log u)$ worst case time. This is the only existing data structure of size $O(uH_k(T))$ able of spending $O(\log u)$ time per occurrence reported. Its main drawback is the factor 4 in its space complexity, which makes it larger than other state-of-the-art alternatives. In this paper we present two different approaches to reduce the space requirement of LZ-index. In both cases we achieve $(2+\varepsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any constant $\varepsilon>0$, and we simultaneously improve the search time to $O(m^2\log m+(m+occ)\log u)$. Both indexes support displaying any subtext of length ℓ in optimal $O(\ell/\log_\sigma u)$ time. In addition, we show how the space can be squeezed to $(1+\varepsilon)uH_k(T)+o(u\log\sigma)$ to obtain a structure with $O(m^2)$ average search time for $m\geqslant 2\log_\sigma u$.

1 Introduction and Previous Work

Text searching is a classical problem in Computer Science. Given a sequence of symbols $T_{1\dots u}$ (the text) over an alphabet Σ of size σ, and given another (short) sequence $P_{1\dots m}$ (the *search pattern*) over Σ, the *full-text search problem* consists in finding all the *occ* occurrences of P in T.

Applications of full-text searching include text databases in general, which typically contain natural language texts, DNA or protein sequences, MIDI pitch sequences, program code, etc. A central goal of modern text databases is *to provide fast access to the text using as little space as possible*. Yet, these goals are opposed: to provide fast access we must build an index on the text, increasing the space requirement. The main motivation of using little space is to store the indexes of very large texts entirely in main memory. This can compensate for significant CPU time to access them. 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.

A concept related to text compression is that of the k-th order empirical entropy of a sequence of symbols T, denoted by $H_k(T)$ [10]. The value $uH_k(T)$ provides a lower bound to the number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precede it in T. It holds that $0\leqslant H_k(T)\leqslant H_{k-1}(T)\leqslant\cdots\leqslant H_0(T)\leqslant\log\sigma$ (log means $\log_2$ in this paper).

To provide fast access to the text using little space, the current trend is to use *compressed full-text self-indexes*. A *full-text self-index* allows one to search and retrieve any part of the text without storing the text itself. A *full-text compressed index* is one whose space requirement is proportional to the compressed text size. Then a compressed full-text self-index *replaces* the text with a more space-efficient representation of it, which at the same time provides indexed access to the text. Some works on compressed self-indexes are [19, 4, 6, 9, 5]. Taking space proportional to the compressed text, replacing it, and providing efficient indexed access to it is an unprecedented breakthrough in text indexing and compression.

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

The LZ-index [17, 16] is a compressed full-text self-index on these lines, based on the Ziv-Lempel [21] parsing of the text. If the text is parsed into n phrases by the LZ78 algorithm, then the LZ-index takes $4n\log_2 n(1+o(1))$ bits of space, which is 4 times the size of the compressed text, i.e. $4uH_k(T)+o(u\log\sigma)$ bits, for any $k=o(\log_\sigma u)$ [8, 4]. The LZ-index answers queries in $O(m^3\log\sigma+(m+occ)\log n)$ worst case time. The index also replaces the text (that is, is a *self-index*): it can reproduce a text context of length ℓ around an occurrence found (and in fact any sequence of 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.

However, in practice the space requirement of LZ-index is relatively large compared with competing schemes, such as *CS-Array* [19] and *FM-index* [4], which require 0.6–0.7 and 0.3–0.8 times the text size respectively, versus 1.2–1.6 times the text size of LZ-index. Yet, the LZ-index is faster to report and to display the context of an occurrence. Fast displaying of text substrings is very important in self-indexes, as the text is not available otherwise.

In this paper we study how to reduce the space requirement of LZ-index, using two different approaches. The first one, a *navigational scheme* approach, consists in reducing the redundancy among the different data structures that conform the LZ-index. These data structures allow us moving among data representations. In this part we define new data structures allowing the same navigation, yet reducing the original redundancy. In the second approach we combine the balanced parentheses representation of Munro and Raman [14] of the LZ78 trie with the *xbw transform* of Ferragina et al. [3], whose powerful operations are useful for the LZ-index search algorithm.

Despite these approaches are very different, in both cases we achieve $(2+\varepsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any constant $\varepsilon>0$, and we simultaneously *improve* the search time to $O(m^2\log m+(m+occ)\log n)$ (worst case). In both cases we also present a version requiring $(1+\varepsilon)uH_k(T)+o(u\log\sigma)$ bits, and whose search time is, on average, $O(m^2)$ if $m\geqslant 2\log_\sigma n$. In all cases, the worst case time to display a context of length ℓ around any occurrence found is optimal $O(\ell/\log_\sigma u)$. Note that, just as LZ-index, our data structures are the only compressed full-text self-indexes of size $O(uH_k(T))$ able of spending $O(\log n)$ time per occurrence reported. Our data structures are not competitive against schemes requiring about the same space [6, 5] for counting the number of occurrences of P in T. Yet, in many practical situations, it is necessary to report the occurrence positions as well as displaying their contexts. In this aspect, LZ-index is superior.

2 The LZ-index Data Structure

Assume that the text $T_{1\ldots u}$ has been compressed using the LZ78 [21] algorithm into $n+1$ phrases[3] $T=B_0\ldots B_n$, such that $B_0=\varepsilon$ (the empty string); $\forall k\neq\ell, B_k\neq B_\ell$; and $\forall k\geqslant 1, \exists\ell<k, c\in\Sigma, B_k=B_\ell\cdot c$. To ensure that B_n is not a prefix of another B_i, we append to T a special symbol "\$" $\notin\Sigma$, assumed to be smaller than any other symbol. We say that i is the *phrase identifier* corresponding to B_i, $0\leqslant i\leqslant n$.

The following data structures conform the LZ-index [17, 16].

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

[3] According to [21], $\sqrt{u}\leqslant n\leqslant\frac{u}{\log_\sigma u}$; thus, $n\log u\leqslant u\log\sigma$ always holds.

Each of these four structures requires $n \log n(1+o(1))$ bits of space if they are represented succinctly, for example, using the balanced parenthesis representation [14] for the tries. For *Range*, the data structure of Chazelle [2] permits two-dimensional range searching in a grid of n pairs of integers in the range $[0 \ldots n] \times [0 \ldots n]$, answering queries in $O((occ+1)\log n)$ time, where occ is the number of occurrences reported, and requiring $n \log n(1+o(1))$ bits. As $n \log u = uH_k(T) + O(kn \log \sigma) \leqslant u \log \sigma$ for any k [8], the final size of the LZ-index is $4uH_k(T) + o(u \log \sigma)$ bits for $k = o(\log_\sigma u)$. The succinct representation given in the original work [17] implements (among others) the operations $parent(x)$ (which gets the parent of node x) and $child(x, \alpha)$ (which gets the child of node x with label $\alpha \in \Sigma$) 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 operation $ancestor(x, y)$, which is used to ask if node x is an ancestor of node y, is implemented in $O(1)$ time both in *LZTrie* and *RevTrie*. These operations are basically based on rank/select operations on bit vectors. Given a bit vector $\mathcal{B}_{1\ldots n}$, we define the function $rank_0(\mathcal{B}, i)$ (similarly $rank_1$) as the number of 0s (1s) occurring up to the i-th position of $\mathcal{B}$. The function $select_0(\mathcal{B}, i)$ (similarly $select_1$) is defined as the position of the i-th 0 (1) in $\mathcal{B}$. These operations ca be supported in constant time and requiring $n + o(n)$ bits [12], or $H_0(\mathcal{B}) + o(n)$ bits [18].

Let us consider now the search algorithm for a pattern $P_{1\ldots m}$ [16, 17]. We distinguish three types of occurrences of P in T, depending on the phrase layout:

1. **the occurrence lies inside a single phrase** (there are occ_1 occurrences of this type): given the properties of LZ78, every phrase B_k containing P is formed by a shorter phrase B_ℓ concatenated to a symbol c. If P does not occur at the end of B_k, then B_ℓ contains P as well. We want to find the shortest possible phrase B in the LZ78 referencing chain for B_k that contains the occurrence of P. This phrase B finishes with the string P, hence it can be easily found by searching for P^r in *RevTrie* in $O(m^2 \log \sigma)$ time. Say we arrive at node v. Any node v' descending from v (including v itself) corresponds to a phrase terminated with P. Thus we traverse and report all subtrees of the *LZTrie* node corresponding to each v'. Occurrences of type 1 are located in $O(m^2 \log \sigma + occ_1)$ time;
2. **the occurrence spans two consecutive phrases**, B_k and B_{k+1}, such that a prefix $P_{1\ldots i}$ matches a suffix of B_k and the suffix $P_{i+1\ldots m}$ matches a prefix of B_{k+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 in *RevTrie* and for the pattern suffix in *LZTrie*. Now we have two ranges, one in the space of reversed strings (phrases finishing with the first part of P) and one in that of the normal strings (phrases starting with the second part of P), and need to find the phrase pairs $(k, k+1)$ such that k is in the first range and $k+1$ is in the second range. This is what the range searching data structure is for. Occurrences of type 2 are located in $O(m^3 \log \sigma + (m + occ_2)\log n)$ time; and
3. **the occurrence spans three or more phrases**, $B_k \ldots B_\ell$, such that $P_{i\ldots j} = B_{k+1} \ldots B_{\ell-1}$, $P_{1\ldots i-1}$ matches a suffix of B_k and $P_{j+1\ldots m}$ matches a prefix of B_ℓ (there are occ_3 occurrences of this type): we need one more observation for this part. The LZ78 algorithm guarantees that every phrase represents a different string. Hence, there is at most one phrase matching $P_{i\ldots j}$ for each choice of i and j. This fact severely limits the number of occurrences of this class that may exist, $occ_3 = O(m^2)$. The idea is to identify maximal concatenations of phrases $P_{i\ldots j} = B_k \ldots B_\ell$ contained in the pattern, and thus determine whether B_{k-1} finishes with $P_{1\ldots i-1}$ and $B_{\ell+1}$ starts with $P_{j+1\ldots m}$. If this is the case we can report an occurrence. We first search for every pattern substring in *LZTrie*, in $O(m^2 \log \sigma)$ time. Then, the $O(m^2)$ maximal concatenations of phrases are obtained in $O(m^2 \log m)$ worst case time and $O(m^2)$ time on average. Finally, each of those maximal concatenations is verified in $O(m \log \sigma)$ time using operation *parent* for B_k. Overall, occurrences of type 3 are located in $O(m^3 \log \sigma)$ 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 $O(m^3 \log \sigma + (m + occ)\log n)$. Finally, we can uncompress and display the text of length ℓ surrounding any occurrence reported in $O(\ell \log \sigma)$ time, and uncompress the whole text T in $O(u \log \sigma)$ time.

3 LZ-index as a Navigation Scheme

In the practical implementation of LZ-index [17, 16], the *Range* data structure is replaced by *RNode*, which is a mapping from phrase identifiers to their node in *RevTrie*. Now occurrences of type 2 are found as follows: 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_{rev}, and the search for $P_{i+1...m}$ in *LZTrie* yields node v_{lz}. Then, we check each phrase k in the subtree of v_{rev} and report it if $Node[k+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has less elements, we do the opposite: check phrases from v_{lz} in v_{rev}, using *RNode*. Unlike when using *Range*, now the time to solve occurrences of type 2 is proportional to the smallest subtree size among v_{rev} 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 $O(n/\sigma^{m/2})$. This is $O(1)$ for long patterns, $m \geqslant 2\log_\sigma n$. The *RNode* data structure requires $uH_k(T) + o(u\log\sigma)$ bits, and so this version of LZ-index also requires $4uH_k(T) + o(u\log\sigma)$ bits, $k = o(\log_\sigma u)$.

Both *LZTrie* and *RevTrie* use originally the *balanced parenthesis* representation [14], in which every node, represented by a pair of opening and closing parentheses, encloses its subtree. When we replace *Range* by *RNode* structure, the result is actually a "navigation" scheme that permits us moving back and forth from trie nodes to the corresponding *preorder positions*[4], both in *LZTrie* and *RevTrie*. The phrase identifiers are common to both tries and permit moving from one trie to the other.

Figure 1 (left) shows the navigation scheme. Dashed arrows are asymptotically "for free" in terms of memory, since they are followed by applying *rank* on the corresponding parenthesis structure. The other four 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 array of tree nodes in *LZTrie* (*Node*) and in *RevTrie* (*RNode*). The structure is symmetric and we can move from any point to any other.

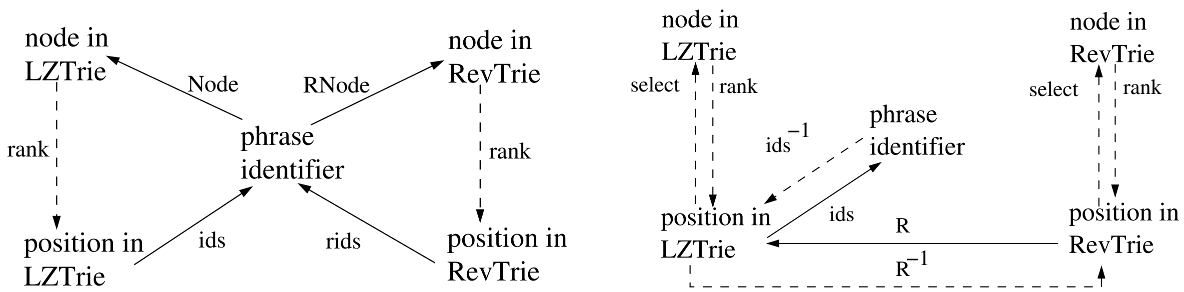


Fig. 1. The original (left) and the reduced (right) navigation structure over index components.

The structure, however, is redundant, in the sense that the number of arrows is not minimal. We start by defining the following reduced scheme for LZ-index:

- *LZTrie*: the Lempel-Ziv trie, which is implemented with the following data structures
  - $par_{0...2n}$: the tree shape of *LZTrie* according to the representation of [1], which requires $2n + n\lceil\log\sigma\rceil + o(n) + O(\log\log\sigma)$ bits to support the operations $parent(x)$, $child(x,\alpha)$, *subtree size* (including the root of the subtree), and *node degree*, all of them in $O(1)$ time. It also supports the operation $child(x,i)$ in constant time, which gets the i-th child of node x. To get this representation, we perform a preorder traversal on the trie, and for every node touched we write its

[4] In the parenthesis representation [14], the preorder position of a node is the number of opening parenthesis before the one representing the node. This is the $rank_0$ of the node position in the bit sequence representing the parentheses, if bit 1 represents ')'.

degree in unary using parentheses (for example, 3 reads ‘`((()`’ and it is writen ‘`0001`’), 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 represented by the position of the first of the $(d+1)$ parentheses representing the node. Given a node in this representation, say at position i, its preorder position can be computed by $rank_1(par,i)$. Given a preorder position p, the corresponding node is computed by $select_1(par,p)+1$. With this representation we can compute all the operations required by *LZTrie* [17] in $O(1)$ time, including $ancestor(x,y)$ [5]. The symbols labeling the arcs of the trie are represented implicitly. We denote by $lets$ the array of such symbols, such that $lets[i]$ is the symbol corresponding to the node at position $select_0(par,i)+1$ (i.e., the symbol with preorder position i).

  - $ids_{0\ldots n}$: the array of LZ78 phrase identifiers in preorder. We use the representation of Munro et al. [13] for ids such that the inverse permutation ids^{-1} can be computed in $O(1/\varepsilon)$ time, requiring $(1+\varepsilon)n\log n$ bits[6].

- *RevTrie*: the *PATRICIA* tree [11] of the reversed LZ78 phrases, which is implemented with the following data structures
  - $rpar_{0\ldots 2n'}$: the *RevTrie* structure according to [1], compressing empty unary paths and so ensuring $n' \leqslant 2n$ nodes, because empty non-unary nodes still exist. The space requirement is $2n' + n'\lceil\log\sigma\rceil + o(n') + O(\log\log\sigma)$ bits to support the same functionalities as *LZTrie*.
  - $B_{0\ldots n'}$: a bit vector supporting *rank* and *select* queries, and requiring $n'(1+o(1))$ bits [12]. The j-th bit of B is 1 iff the node with preorder position j in $rpar$ is not an empty node, otherwise the bit is 0. Given a position p in $rpar$ corresponding to a *RevTrie* node, the corresponding bit in B is $B[rank_1(rpar,p)]$.
  - $R_{0\ldots n}$: a mapping from *RevTrie* preorder positions to *LZTrie* preorder positions defined as $R[i] = ids^{-1}(rids[i])$. R is implemented using the succinct data structure for permutations of Munro et al. [13], requiring $(1+\varepsilon)n\log n$ bits to represent R and compute R^{-1} in $O(1/\varepsilon)$ worst case time. Given a position i in $rpar$ corresponding to a *RevTrie* node, the corresponding R value is $R[rank_1(B, rank_1(rpar,i))]$.
  - $skips_{0\ldots n'}$: the skips of the nodes in preorder, and using $\log\log u$ bits per node, 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 $O(u/\log u)$ to n', which becomes $O(\frac{u\log\log u}{\log u}) = o(u)$ bits overall.

In Fig. 1 (right) we show the corresponding navigation scheme. The search algorithm remains the same since we can map preorder positions to nodes in the new representation of the tries (and vice versa), and we can simulate $rids[i] = ids[R[i]]$, $RNode[i] = select_1(rpar, R^{-1}(ids^{-1}(i))) + 1$, and $Node[i] = select_1(par, ids^{-1}(i)) + 1$, all of which take constant time.

The space requirement is $(2+\varepsilon)n\log n + 3n\log\sigma + 2n\log\log u + 8n + o(u) = (2+\varepsilon)n\log n + o(u\log\sigma)$ bits if $\log\sigma = o(\log u)$. As $n\log u = uH_k(T) + O(kn\log\sigma)$ for any k [8], the space requirement is $(2+\varepsilon)uH_k(T) + o(u\log\sigma)$ bits, for any $k = o(\log_\sigma u)$. The *child* operation on *RevTrie* can now be computed in $O(1)$ time, versus the $O(h\log\sigma)$ time of the original LZ-index [17]. Hence, the *occ* occurrences of P can be reported in $O(\frac{m^2}{\varepsilon} + \frac{n}{\varepsilon\sigma^{m/2}})$ average time, for $0 < \varepsilon < 1$.

Reducing Further. To simplify notation, given a *LZTrie* node with preorder position $R[i]$, suppose that operation $parent(R[i])$ gives the preorder position of its parent. We define function φ such that

$$\varphi[i] = R^{-1}(parent(R[i])) \tag{1}$$

[5] As $ancestor(x,y) \equiv rank_1(par,x) \leqslant rank_1(par,y) \leqslant rank_1(par,x) + subtreesize(par,x) - 1$.

[6] This data structure ensures that one finds the inverse after following the permutation $O(1/\varepsilon)$ times.

That is, in $\varphi[i]$ we store the *RevTrie* preorder position corresponding to the maximal proper suffix of the i-th string in *RevTrie* (i.e., the string whose node has preorder position i in *RevTrie*). Note this is a kind of suffix link in *RevTrie*.

Let us show how to compute R using only φ. We define array $L_{1\ldots n}$ such that $L[i] = lets[R[i]]$. It is not hard to prove that $L[i] \leqslant L[j]$ whenever $i \leqslant j$, and thus L can be divided into σ runs of equal symbols. In this way L can be represented by an array L' of $\sigma \log \sigma$ 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 "\$"). For every i such that $L_B[i] = 1$, we store $L'[rank_1(L_B, i)] = L[i]$. Hence, $L[i]$ can be computed as $L'[rank_1(L_B, i)]$ in $O(1)$ time. To simplify the notation assume that, given a *LZTrie* position $R[i]$, operation $child(R[i], \alpha)$ yields the *LZTrie* preorder position belonging to the child (by symbol α) of the node corresponding to $R[i]$.

Lemma 1. *Given $0 \leqslant i \leqslant n$, the value $R[i]$ can be computed by the following recurrence:*

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

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 arc connecting x to y is stored in $L[i]$. That is, $child(parent(R[i]), L[i]) = R[i]$. The lemma follows from this fact and replacing $\varphi[i]$ by its definition (Eq. (1)) in the recurrence. □

As in the case of the Ψ function of *Compressed Suffix Arrays* [7, 19], we can prove the following lemmas for the φ function, which are the key to compress the mapping R.

Lemma 2. *For every $i < j$, if $lets[R[i]] = lets[R[j]]$, 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 $lets[R[i]] = lets[R[j]]$ (i.e., the $str_r(i)$ and $str_r(j)$ start with the same symbol, as their reverses end 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), and thus $\varphi[i] < \varphi[j]$. □

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

As a result, we replace R by φ and L_B, and use them to compute a given value $R[i]$. According to Lemma 1, we can represent φ using the idea of Sadakane [19], requiring $nH_0(lets) + O(n \log\log \sigma)$ bits and allowing to access $\varphi[i]$ in constant time, and hence we replace the $n \log n$ bits representation of R by the $nH_0(lets) + O(n \log\log \sigma) + n + o(n)$ bits representation of φ and L_B.

The time to compute $R[i]$ is now $O(|str_r(i)|)$, which actually corresponds to traversing *LZTrie* from the root with the symbols of $str_r(i)$ in reverse order. But we can 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 $O(1/\varepsilon)$ time and requiring $\varepsilon n \log n$ extra bits. To determine the R values to be explicitly stored, for each *LZTrie* leaf we traverse the upward path to the root, marking one out of $O(1/\varepsilon)$ nodes, and stopping the procedure for the current leaf when we reach the root or when we reach an already marked node. If the node to mark is at preorder position j, then we set $R_B[R^{-1}(j)] = 1$. 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 $R'[rank_1(R_B, i)] = R[i]$. Then, we free R since $R[i]$ can be computed by:

$$R[i] = \begin{cases} child(R[\varphi[i]], L'[rank_1(L_B, i)]) & \text{if } R_B[i] = 0 \\ R'[rank_1(R_B, i)] & \text{if } R_B[i] = 1 \end{cases}$$

Note that the same structure used to compute R^{-1} before freeing R can be used under this scheme, with cost $O(1/\varepsilon^2)$ (recall footnote 6).

Theorem 1. *There exists a compressed full-text self-index requiring* $(1+\varepsilon)uH_k(T)+o(u\log\sigma)$ *bits of space, for* $k=o(\log_\sigma u)$ *and any constant* $0<\varepsilon<1$*, and able to report the occ occurrences of a pattern* $P_{1\ldots m}$ *in a text* $T_{1\ldots u}$ *in* $O(\frac{m^2}{\varepsilon^2}+\frac{n}{\varepsilon^2\sigma^{m/2}})$ *average time, which is* $O(m^2)$ *if* $m\geqslant 2\log_\sigma n$*. Also, it is able to display a text substring of length* ℓ *in* $O(\ell(1+\frac{1}{\varepsilon\log_\sigma\ell}))$ *worst case time.*

The bound $O(\ell(1+\frac{1}{\varepsilon\log_\sigma\ell})$ in the displaying time holds from the fact that we perform ℓ *parent* operations, and we must pay $O(1/\varepsilon)$ to use ids^{-1} each time we pass to display the next (previous) phrase, which in the (very) worst case is done $O(\ell/\log_\sigma\ell)$ times. Note that we assume that the ℓ symbols displayed correspond to whole phrases.

We can get worst case guarantees in the search process by adding *Range*, the two dimensional range search data structure defined in Section 2 for the original LZ-index. Occurrences of type 2 can now be solved in $O(m^2+(m+occ)\log n)$ time. Hence, we have the following theorem.

Theorem 2. *There exists a compressed full-text self-index requiring* $(2+\varepsilon)uH_k(T)+o(u\log\sigma)$ *bits of space, for* $k=o(\log_\sigma u)$ *and any constant* $0<\varepsilon<1$*, and able to report the occ occurrences of a pattern* $P_{1\ldots m}$ *in a text* $T_{1\ldots u}$ *in* $O(m^2(\log m+\frac{1}{\varepsilon^2})+(m+occ)\log n+\frac{occ}{\varepsilon})=O(m^2\log m+(m+occ)\log n)$ *worst case time. Also, it is able to display a text substring of length* ℓ *in* $O(\ell(1+\frac{1}{\varepsilon\log_\sigma\ell}))$ *worst case time.*

It is important to remark that, as for the original LZ-index, our indexes are the only existing compressed full-text self-indexes of size $O(uH_k(T))$ bits able of spending $O(\log n)$ time per occurrence reported.

4 Using the *xbw* Transform to Represent LZTrie

A different idea to reduce the space requirement of LZ-index is to use the *xbw transform* of Ferragina et al. [3] to represent the *LZTrie*. This succinct representation supports the operations $parent(x)$, $child(x,i)$, and $child(x,\alpha)$, all of them in $O(1)$ time and using $2n\log\sigma+O(n)$ bits of space. 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 suffix of the string represented by x. We represent LZ-index with the following data structures:

- Balanced parentheses *LZTrie*: the trie of the Lempel-Ziv phrases, implemented by
  - *par*: the balanced parentheses representation [14] of *LZTrie*. In order to index the *LZTrie* leaves with *xbw*, we have to add a dummy child to each. In this way, the trie has $n'\leqslant 2n$ nodes. The space requirement is $4n+o(n)$ bits in the worst case if we use the Munro and Raman representation [14]. We use the bit 0 to represent '`(`' and 1 to represent '`)`'. In this way, the preorder position of a node is computed by a $rank_0$ query, and the node corresponding to a preorder position by a $select_0$ query, both in $O(1)$ time.
  - *ids*: the array of LZ78 phrase identifiers in preorder, represented by the data structure of Munro et al. [13], such that we can compute the inverse permutation ids^{-1} in $O(1/\varepsilon)$ time, requiring $(1+\varepsilon)n\log n$ bits.
- *xbw LZTrie*: the *xbw* representation [3] of the *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 arcs of the trie. In the worst case *LZTrie* has $2n$ nodes (because of the dummy leaves we add), and then this array requires $2n\log\sigma$ 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. The space requirement is $n(2+o(1))$ bits (worst case).
- *Pos*: a mapping from *xbw* positions to the corresponding preorder positions. In the worst case there are $2n$ such positions, and so the space requirement is $2n\log(2n)$ bits. We can reduce this space to $\varepsilon n\log(2n)$ bits by storing in an array Pos' one out of $O(1/\varepsilon)$ values of *Pos*, such that $Pos[i]$ can be computed in $O(1/\varepsilon)$ time. We need a bit vector Pos_B of $n(2+o(1))$ bits indicating which values of

Pos have been stored. Assume we need compute $Pos[i]$, for a given *xbw* position i. If $Pos_B[i] = 1$, then such value is stored at $Pos'[rank_1(Pos_B, i)]$. Otherwise, we simulate a preorder traversal in *xbw* from node at *xbw* position i, until $Pos_B[j] = 1$, for a *xbw* position j. Once this j is found, we map to the preorder position $j' = Pos'[rank_1(Pos_B, j)]$. If d is the number of nodes in preorder 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 $O(1/\varepsilon^2)$ time under this scheme, requiring $\varepsilon n \log(2n)$ extra bits if we use the representation of [13].
– *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 phrase* k and y is the *preorder position of phrase* $k+1$. We use the data structure of Chazelle [2], as for the original LZ-index. The space requirement is $n \log n(1 + o(1))$ bits.

The total space requirement is $(2+\varepsilon)n\log n(1+o(1)) + 2n\log\sigma + 8n + o(n)$ bits, which is $(2+\varepsilon)uH_k(T) + o(u\log\sigma)$ bits for $k = o(\log_\sigma u)$.

We depict now the search algorithm for a pattern P of length m. For occurrences of type 1, we perform a subpath query for P to obtain the interval $[x_1, x_2]$ in the *xbw* of *LZTrie* corresponding to all the nodes whose phrase ends with P. For each position $i \in [x_1, x_2]$, we can get the corresponding node in the parenthesis representation using $select_0(par, Pos[i])$, and then we traverse the subtrees of these nodes and report all the identifiers found, as done with the usual LZ-index.

To solve occurrences of type 2, for every possible partition $P_{1\ldots i}$ and $P_{i+1\ldots m}$ of P, we traverse the *xbw* from the root, using operation $child(x, \alpha)$ with the symbols of $P_{i+1\ldots m}$. Once this is found, say at *xbw* position i, we switch to the preorder tree (parentheses) using $select_0(par, Pos[i])$, to get the node v_{lz} whose subtree has preorder interval $[y_1, y_2]$. This is the preorder interval of all the nodes that start with $P_{i+1\ldots m}$. Next we perform a subpath query for $P_{1\ldots i}$ in *xbw*, and get the *xbw* interval $[x_1, x_2]$ of all the nodes that finish with $P_{1\ldots i}$ (actually we have to perform $rank_1(S_{last}, i)$ to avoid counting the same node multiple times, see [3]). Then, we search the *Range* data structure for $[x_1 \ldots x_2] \times [y_1 \ldots y_2]$ to get all phrase identifiers k such that k finishes with $P_{1\ldots i}$ and $k+1$ starts with $P_{i+1\ldots m}$.

For occurrences of type 3, one could do 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 $P_{i\ldots j} = B_{k+1}\ldots B_{\ell-1}$ we must check that phrase B_ℓ starts with $P_{j+1\ldots m}$ and that phrase B_k finishes with $P_{1\ldots i-1}$. The first check can be done in constant time using ids^{-1}. As we have searched for all substrings of P in the trie, we know the preorder interval of descendants of $P_{j+1\ldots m}$, thus we check whether the node at preorder position $ids^{-1}(\ell)$ belongs to that interval. The second check can also be done in constant time, by determining whether k is in the *xbw* interval of $P_{1\ldots i-1}$ (that is, B_k finishes with $P_{1\ldots i-1}$). For this, we need Pos^{-1}, so that the position is $Pos^{-1}(ids^{-1}(k))$.

To display the text around an occurrence, we use ids^{-1} to find the preorder position of the corresponding phrase, and then we use *parent* on the parentheses to find the symbols in the upward path. To know the symbol, we have to use Pos^{-1} to go to the *xbw* position and read S_α.

For the search time, occurrences of type 1 cost $O(m + occ/\varepsilon)$, type 2 cost $O(m^2 + m/\varepsilon + m(occ + \log n))$, and type 3 cost $O(m^2(\log m + \frac{1}{\varepsilon^2}))$. Thus, we have achieved Theorem 2 again with radically different means. The displaying time is $O(\ell/\varepsilon)$, but it can also become $O(\ell(1 + \frac{1}{\varepsilon \log_\sigma \ell}))$ if we store the array of symbols in the balanced parentheses *LZTrie*, which adds $o(u\log\sigma)$ bits of space. We can get a version requiring $(1+\varepsilon)uH_k(T) + o(u\log\sigma)$ bits and $O(m^2)$ average reporting time (if $m \geqslant 2\log_\sigma n$) (as in Theorem 1) if we solve occurrences of type 2 by using a procedure similar to that used to solve occurrences of type 3.

5 Displaying Text Substrings

LZ-index is able of reporting occurrences in the format (k, offset), where k 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. However, we can report occurrences as *text positions* by adding a bit vector $V_{1\dots u}$ that marks the n phrase beginnings. Then $rank_1(V,i)$ is the phrase number i belongs to, and $select_1(V,j)$ is the text position of the j-th phrase. Such V can be represented with $H_0(V)+o(u) \leqslant n\log u/n + o(u) \leqslant n\log\log u + o(u) = o(u\log\sigma)$ bits [18].

We can also add an operation for displaying a subtext $T_{i\dots i+\ell-1}$ for any given position i, to both proposed indexes. The time complexity is the optimal $O(\ell/\log_\sigma u)$.

A compressed data structure [20] to display any text substring of length $\Theta(\log_\sigma u)$ in constant time, turns out to have similarities with LZ-index. We take advantage of this similarity to plug it within our index, with some modifications, and obtain improved time to display text substrings.

They proposed auxiliary data structures of $o(u\log\sigma)$ bits to LZ78 to support this operation efficiently. Given a position i of the text, we first find the phrase including the position i by using rank and select of the above vector V, then find the node of *LZTrie* that corresponds to the phrase by ids^{-1}. Then displaying a phrase is equivalent to outputting the path going from the node to the root of *LZTrie*. The size of the auxiliary data structure is $O(n\log\sigma) = o(u\log\sigma)$. This time we can display not only whole phrases, but any text substring within this complexity. The reason is that any prefix of a phrase is also a phrase, and it can be found in constant time by using a level-ancestor query [15] on the *LZTrie*.

We modify this method to plug into our indexes. In their original method, if more than one consecutive phrases have length less than $1/2\log_\sigma u$ each, their phrase identifiers are not stored. Instead the substring of the text including those phrases are stored without compression. This guarantees efficient displaying operation without increasing the space requirement. However this will cause a problem that we cannot find patterns including those phrases. Therefore in our modification we store 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 $1/2\log_\sigma u$, we store a substring of length $\log u$ containing the phrase. Because there are at most $O(\sqrt{u})$ such phrases, we can store the substrings in $O(\sqrt{u}\log u) = o(u)$ bits. These auxiliary data structures work if we can convert each phrase identifier into the preorder of the corresponding node in *LZtrie* in constant time. Therefore they can be applied to all the data structures in Sections 3 and 4.

6 Conclusions

In this paper we have studied the reduction of the space requirement of LZ-index, using two different approaches: the *navigational scheme* approach, and the *xbw representation of the LZ78 trie* approach. In the first approach, we replace the original LZ-index data structures by new ones allowing the same functionalities, but with reduced redundancy. In the second approach, the main idea is to combine the balanced parentheses [14] representation of the LZ78 trie with the *xbw* transform [3]. With either approach we achieve $(2+\varepsilon)uH_k(T) + o(u\log\sigma)$ bits of space to index to text $T_{1\dots u}$, for $k = o(\log_\sigma u)$ and any constant $\varepsilon > 0$, and the original time to report the *occ* occurrences of a pattern $P_{1\dots m}$ in T is *improved* to $O(m^2\log\sigma + (m+occ)\log n)$. We also define an index requiring $(1+\varepsilon)uH_k(T) + o(u\log\sigma)$ bits and with $O(m^2)$ average case time if $m \geqslant 2\log_\sigma n$. The time to display a context of length ℓ around any occurrence reported is the optimal $O(\ell/\log_\sigma n)$.

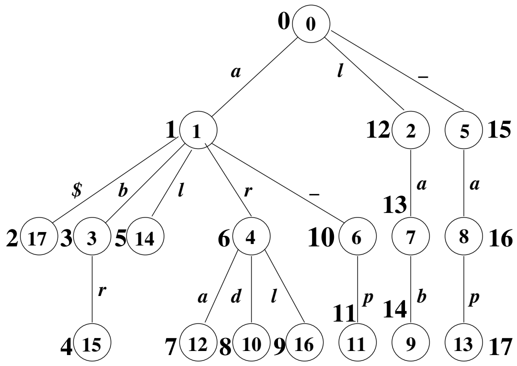
References

1. D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S.S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.
2. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.
3. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *Proc FOCS'05*, pages 184–196, 2005.
4. P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of the ACM*, 54(4):552–581, 2005.

5. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proc.SPIRE'04*, LNCS 3246, pages 150–160. Springer, 2004. Extended version to appear in *ACM TALG*.
6. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA'03*, pages 841–850. SIAM, 2003.
7. 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.
8. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
9. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proc. CPM'05*, LNCS 3537, pages 45–56, 2005.
10. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
11. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
12. I. Munro. Tables. In *Proc. FSTTCS'96*, LNCS 1180, pages 37–42. Springer, 1996.
13. I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *ICALP*, LNCS 2719, pages 345–356, 2003.
14. J. I. Munro and V. Raman. Succinct Representation of Balanced Parentheses and Static Trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
15. J. I. Munro and S. S. Rao. Succinct Representations of Functions. In *Proceedings of ICALP*, LNCS 3142, pages 1006–1015, 2004.
16. G. Navarro. Indexing text using the Ziv-Lempel trie. Technical Report TR/DCC-2002-2, Dept. of Computer Science, Univ. of Chile, 2002. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/lzindex.ps.gz`.
17. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
18. R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA'02*, pages 233–242, 2002.
19. K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
20. K. Sadakane and R. Grossi. Squeezing Succinct Data Structures into Entropy Bounds. In *Proc. ACM-SIAM SODA*, pages 1230–1239, 2006.
21. J. Ziv and A. Lempel. Compression of individual sequences via variable–rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

A Example

We illustrate the definition of our data structure of Theorem 1 with an example. In Fig 2 we show the data structure for the text '`alabar_a_la_alabarda_para_apalabrarla$`'. We show the corresponding *LZTrie* (with preorder numbers outside each node), the LZ78 parsing of the text, the set of string in *RevTrie* (in preorder, i.e., lexicographically sorted) and some of the arrays that conform the index. Given the *RevTrie* preorder position 13, for example, which corresponds to string '`p_a`' in *RevTrie*, to compute the *LZTrie* preorder position $R[13]$ we need to compute $child(R[\varphi[13]], L[13])$, where $\varphi[13] = 17$ corresponds to the *RevTrie* preorder position of string '`_a`'. Note that by using φ we are following '*suffix links*' in *RevTrie* (which corresponds to *parent* operations in *LZTrie*) until we reach the root node. At this point we must perform the *child* operations from the *LZTrie* root. For our example, we must compute $child(child(child(0,\mathrm{a}),\_),\mathrm{p})$ in *LZTrie*, which is the same as $child(child(1,\_),\mathrm{p})$, which in turn is $child(10,\mathrm{p})$, and finally we get the node with preorder position 11. Hence, $R[13] = 11$. By storing some values of R we can stop traversing *RevTrie* as soon as we reach a node whose R value has been stored. Then, we use this R value to get the corresponding *LZTrie* node from where to apply the *child* operations.



alabar a la alabarda para apalabrarla

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$*

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

Fig. 2. Illustration of the data structure of Theorem 1.