Space-Efficient Construction of Lempel-Ziv Compressed Text Indexes ⋆

Diego Arroyuelo[1] ⋆⋆ and Gonzalo Navarro[2] ⋆⋆⋆

[1] Yahoo! Research Latin America, Blanco Encalada 2120, Santiago, Chile.
[2] Dept. of Computer Science, University of de Chile, Blanco Encalada 2120, Santiago, Chile.
{darroyue, gnavarro}@dcc.uchile.cl

Abstract. A *compressed full-text self-index* is a data structure that replaces a text and in addition gives indexed access to it, while taking space proportional to the compressed text size. This is very important nowadays, since one can accommodate the index of very large texts entirely in main memory, avoiding the slower access to secondary storage. In particular, the LZ-index [G. Navarro, Journal of Discrete Algorithms, 2004] stands out for its good performance at extracting text passages and locating pattern occurrences. Given a text $T[1..u]$ over an alphabet of size σ, the LZ-index requires $4|LZ|(1 + o(1))$ bits of space, where $|LZ|$ is the size of the LZ78-compression of T. This can be bounded by $|LZ| = uH_k(T) + o(u \log \sigma)$, where $H_k(T)$ is the k-th order empirical entropy of T, for any $k = o(\log_\sigma u)$. The LZ-index is built in $O(u \log \sigma)$ time, yet requiring $O(u \log u)$ bits of main memory in the worst case. In practice, the LZ-index occupies 1.0-1.5 times the text size (and replaces the text), but its construction requires around 5 times the text size. This limits its applicability to medium-sized texts. In this paper we present a space-efficient algorithm to construct the LZ-index in $O(u(\log \sigma + \log \log u))$ time and requiring $4|LZ|(1 + o(1))$ bits of main memory, that is, asymptotically the same space of the final index. We also adapt our algorithm to construct more recent reduced versions of the LZ-index, which occupy from 1 to 3 times $|LZ|(1 + o(1))$ bits, and show that these can also be built using asymptotically the same space of the final index. Finally, we study an alternative model in which we are given only a limited amount of main memory to carry out the indexing process (less than that required by the final index), and must use the disk for the rest. We show how to build all the LZ-index variants in $O(u(\log \sigma + \log \log u))$ time, and within $|LZ|(1 + o(1))$ bits of main memory, that is, asymptotically just the space to hold the LZ78-compressed text. Our experimental results show that our method is efficient in practice, needing an amount of memory close to that of the final index, and being competitive with the best construction times of other compressed indexes.

1 Introduction and Previous Work

Text searching is a classical problem in Computer Science. Given a sequence of symbols $T[1..u]$ (the text) over an alphabet Σ of size σ, and given another (short) sequence $P[1..m]$ (the *search pattern*) over Σ, the *full-text search problem* consists of finding (counting or reporting) all the occ occurrences of P in T. Nowadays, much information is stored in the form of (usually large) texts, e.g. biological sequences such as DNA and proteins, XML data, MIDI pitch sequences, digital libraries, program code, etc. Usually, these texts need to be searched for patterns of interest, and therefore the full-text search problem plays a fundamental role in modern computer applications.

Text Compression and Indexing. Despite that there has been some work on space-efficient inverted indexes for natural language texts [71, 58] (able to find whole words and phrases), until one decade ago it was

⋆ A preliminary partial version of this paper appeared in *Proc. ISAAC 2005*, pp. 1143–1152.
⋆⋆ Funded by CONICYT PhD Fellowship Program, Chile. Part of this work was done while the author was in the Department of Computer Science, Univesity of Chile, and later visiting the David Cheriton School of Computer Science, University of Waterloo.
⋆⋆⋆ Funded by Fondecyt Grant 1-080019 and by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

believed that any general index for text searching (such as those that we are considering in this paper) would need much more space. In practice, the smallest index available was the suffix array [46], a compact version of suffix trees [1] requiring $u \log u$ bits[3] to index a text of u symbols. Since the text requires $u \log \sigma$ bits to be represented, the suffix array is usually much larger than the text (typically 4 times the text size). With the large texts available nowadays (e.g., the Human Genome consists of about 3×10^9 base pairs), one solution is to store the indexes on secondary memory. However, this has a significant impact on the running time of an application, as accesses to secondary memory are orders of magnitude slower.

Several attempts to reduce the space of the suffix arrays have been made [41, 26, 65, 18, 25, 42, 19]. They aim at *compressed indexing*, which takes advantage of the regularities of the text to operate in space proportional to that of the compressed text (i.e., c times the size of the text compressed under some model, for some constant c). Especially, in some of those works [65, 18, 25, 55, 42, 19, 64, 7] the indexes *replace* the text and, using little space (sometimes even less than that of the original text), provide indexed access. This feature is known as *self-indexing*, since the index allows one to search and retrieve any part of the text without storing the text itself. Taking space proportional to the compressed text, replacing it, and providing efficient indexed access to it, is an important breakthrough.

The main families of self-indexes based on suffix arrays [57] are the *Compressed Suffix Arrays* (CSAs for short) [65, 25] and FM-indexes (for "Full-text index in Minute space") [18, 42, 19]. The latter compress suffix arrays via the Burrows-Wheeler Transform [10]. The compressibility in both families is usually measured in terms of the *k-th order empirical entropy*, H_k, which is a lower bound on the performance of statistical compressors based on predicting the next text symbol as a function of the k preceding ones.

A separate track of indexes based on Lempel-Ziv compression [72, 73] was pursued in parallel to the research on compressing suffix arrays. These are generally called LZ-indexes [36, 55, 18, 64, 7]. Except for the first pioneering work [36], all the rest are self-indexes and based on the Lempel-Ziv compression algorithm of 1978 (LZ78) [73]. Their space performance is measured in terms of the output size of Lempel-Ziv compressors, which are based on exploiting the repetitions that arise in the text. This can be upper-bounded by the k-th order empirical entropy of the text, but it can be smaller when the text is repetitive.

Handling compressed indexes certainly requires more operations than classical indexes. However, given the relation between main and secondary memory access times, handling compressed indexes entirely in main memory is much faster than handling them in uncompressed form in secondary storage.

We are particularly interested in LZ-indexes, since they have shown to be very effective in practice for extracting text, displaying occurrence contexts, and locating many occurrences, outperforming suffix-array-based self-indexes at these tasks [56, 64, 5, 17]. In theory, only LZ-indexes achieve high-order entropy space together with $O(\log u)$ worst-case time per located occurrence. Moreover, in practice many pattern occurrences can be actually found in constant time. In particular, we will be interested in Navarro's LZ-index [55, 56] and its more recent variants [6, 7, 5].

Compressed Construction of Self-Indexes. Many works on compressed full-text self-indexes do not consider the space-efficient construction of the indexes. Yet, this aspect becomes crucial when implementing the index in practice. For example, the original construction of the CSA [26, 65] and *FM-index* [18] involves building first the suffix array of the text, using for example the algorithm of Larsson and Sadakane [40] or the one by Manzini and Ferragina [48]. Similarly, Navarro's LZ-index is constructed over a non-compressed intermediate representation [55]. In both cases one needs in practice about 5 times the text size (in the case of the CSA and the FM-index, by using the deep-shallow algorithm [48]). For example, the Human Genome

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

may fit in less than 1 GB of main memory using these indexes (and thus it can be operated entirely in RAM on a modest desktop computer), but 15 GB of main memory are needed to build the indexes! Using secondary memory for the construction is nowadays the most practical alternative [15].

Another research path is to try building the suffix array directly in compressed space in main memory. Hon et al. [31] present an algorithm to construct suffix arrays (and also suffix trees) using $O(u \log \sigma)$ bits of storage, in $O(u \log \log \sigma) = o(u \log u)$ time for suffix arrays, and $O(u(\log^{\epsilon} u + \log \sigma))$ time for suffix trees, for any fixed $0 < \epsilon < 1$. This gives an alternative algorithm to construct the CSA and the FM-index using $O(u \log \sigma)$ bits of storage and $O(u \log \log \sigma)$ time. For sufficiently small alphabets, i.e., $\log \sigma = O((\log \log u)^{1-\epsilon})$, the construction time can be made optimal, $O(u)$. However, the space requirement to construct the CSA is still bigger than that needed by the final index.

The work of Hon et al. [29, 30] deal with the space (and time) efficient construction of the CSA. The former uses $(2H_0(T) + 1 + \epsilon)u + o(u \log \sigma)$ bits of space to build the CSA, where ϵ is any positive constant. The construction time is $O(\sigma u \log u)$, which is good enough for small alphabets (as for DNA sequences), but impractical for larger alphabets such as those of Oriental languages.

The second work [30] addresses this problem by requiring $(H_0(T) + 2 + \epsilon)u + o(u \log \sigma)$ bits of space and $O(u \log u)$ time to build the CSA. Also, they show how to build the FM-index from the CSA using negligible extra space in $O(u)$ time. In practice they were able to build the CSA for the Human Genome in about 24 hours and requiring about 3.6 GB of main memory [28], on a 1.7 GHz CPU. The FM-index can be built from the CSA in about 4 extra hours, for a total of about 28 hours.

Na and Park [54] construct the CSA in $O(u \log \sigma (\log_{\sigma} u)^{\log_3 2})$ bits of space and $O(u)$ time. This is the most space-efficient linear-time algorithm for constructing the CSA. They leave open, however, the question of whether the CSA can be constructed in linear time and requiring $O(u \log \sigma)$ bits of space.

Kärkkäinen [35] introduces an algorithm to construct the Burrows-Wheeler transform of a text T (and hence its FM-index) in $O(u \log u + vu)$ worst-case time and using $O(u \log u / \sqrt{v})$ bits of working space, where $v \in [3, u^{2/3}]$. Sirén [32] introduces a space-efficient algorithm to construct CSAs in $O(u \log u)$ worst-case time and using $O(u)$ bits of space on top of the CSA itself. Ferragina et al. [16] present an algorithm for building the Burrows-Wheeler transform of a text T (and also for building compressed indexes) in $O(u \log^{1+\epsilon} u)$ time, for any $\epsilon > 0$, which uses $o(u)$ bits of working space if the alphabet size is a constant. If we make the algorithm from Kärkkäinen [35] use $o(u)$ bits of working space,the construction time becomes $\omega(u \log^2 u)$. However, that complexity holds for any alphabet size, not only for constant-size alphabets [16].

As seen, many works study the space-efficient construction of the CSA and the FM-index. However, the space-efficient construction of LZ-indexes has not been addressed. The original construction algorithm requires $O(u \log \sigma)$ time, but $O(u \log u)$ bits of main memory in the worst case, just as the uncompressed construction of CSAs and FM-indexes. Since LZ-indexes are competitive in practice for locating pattern occurrences and extracting text substrings [56, 5, 17] (which is very important for self-indexes), their space-efficient construction is certainly an important issue.

Our Contribution. We present a practical and efficient algorithm to construct Navarro's LZ-index [55, 56] using little space. Our idea is to replace, at construction time, the (space-inefficient) intermediate representations of the tries that conform the index by space-efficient counterparts. Basically, we define an intermediate representation for the tries, supporting fast incremental construction directly from the text and requiring little space compared with the traditional (pointer-based) representation. The resulting intermediate data structure consists of a tree whose nodes are small connected components of the original trie, or *blocks*. These small tries are represented succinctly in order to require little space. Notice also that the blocks are easier and

cheaper to update, since they are small. The idea is inspired in the work of Clark and Munro [13], yet ours differs in numerous aspects (structuring inside the blocks, overflow management policies, etc.).

Our algorithm builds the LZ-index in $O(u(\log\sigma+\log\log u))$ time, while requiring $4|LZ|(1+o(1))$ bits of space, where $|LZ|$ is the bit-size of the output of the LZ78-compression of T. This is the same asymptotic space the final LZ-index requires to operate. This size can be compared with that of compressed suffix array via the (not always tight) upper bound $|LZ| \leqslant uH_k(T) + o(u\log\sigma)$. At the time of the preliminary version of this work [4], this was the *first* construction algorithm for a compressed self-index requiring space proportional to $H_k(T)$ instead of $H_0(T)$. Recently, however, a construction algorithm for the so-called *Alphabet Friendly* FM-index (AF-FMI) [19] has appeared, requiring $uH_k(T) + o(u\log\sigma)$ bits of space, and $O(u\log u\log\sigma)$ time [44], and even $O(u\frac{\log u\log\sigma}{\log\log u})$ [24]. Yet, the time obtained in the present paper is far better, and it also improves significantly upon the $O(\sigma u)$ worst-case time of our early result [4].

We show how the reduced-space versions of the LZ-index [6, 5, 7] can similarly be constructed within asymptotically the space required by the final index. We also present an alternative model to construct the indexes, in which we assume that the available main memory to carry out the indexing process is smaller than the space required by the final index, and we must use the disk for the rest. This model has applications in cases where the indexing process must be carried out in a computer that is not so powerful to maintain the whole index in main memory, leaving a more powerful equipment exclusively to answer user queries. We show that, under this model, the LZ-indexes can be constructed within $|LZ|(1+o(1))$ bits of space, for any $0<\epsilon<1$, in $O(u(\log\sigma+\log\log u))$ time and $O(|LZ|)$ I/O cost. This means that the LZ-indexes can be built within asymptotically the same space than that required by the LZ78-compressed text.

We implement and empirically test a simplification of our algorithm, and demonstrate that in many practical scenarios the indexing space requirement is almost the same as that of the final index. Thus, we conclude that *wherever the LZ-index can be used, we can build it*. For example, we show that our algorithm is able to build the LZ-index for the Human Genome in less than 5 hours on a 3 Ghz CPU, and requiring 3.5 GB of main memory, so that this work can be carried out in a commodity PC. Under the reduced-memory scenario, our experimental results show that the LZ-index for the Human Genome can be constructed within 1.6 GB of main memory, which is about half of the space required by the uncompressed genome (assuming the base pairs are represented by bytes), and also in less than 5 hours. This is competitive with the best current algorithms to build suffix arrays [15].

Table 1 summarizes the results obtained in this paper and compares with existing approaches.

2 Preliminary Concepts

2.1 Model of Computation

We assume the standard *word* RAM model of computation, in which we can access any memory word of w bits, such that $w=\Theta(\log u)$, in constant time. Standard arithmetic and logical operations are assumed to take constant time under this model. We measure the size of our data structures in bits.

Usually, after an indexing algorithm builds a text index in main memory, the index is stored on disk along with the text database, for persistence purposes. In the case of compressed self-indexes, the index by itself represents the database. At query time, the index is loaded into main memory in order to answer (many) user queries. Thus, by saving the index the (usually costly) indexing process is amortized over several queries. Yet, in other scenarios, one builds the index in main memory and answers queries on the fly.

We will initially assume that there is enough main memory to hold the final index. Later we will consider reduced-main-memory scenarios, where we will resort to secondary memory to hold the intermediate results. In this case, as the final index must reside on disk, we will assume that there is enough secondary memory to hold the index we build.

Table 1. Comparison of different algorithms for constructing text indexes. The reduced-space LZ-index versions can be constructed within the same space required by the final indexes. In all cases ϵ stands for any positive (and usually small) value.

Index	Indexing space (in bits)	Indexing time
Suffix Array (SA) [21]	$u \log u$	$O(u \log u)$
SA [31]	$O(u \log \sigma)$ (*)	$O(u \log \log \sigma)$
CSA [30]	$u(H_0(T) + 2 + \epsilon) + o(u \log \sigma)$ (†)	$O(u \log u)$
CSA [54]	$O(u \log \sigma (\log_\sigma u)^{\log_3 2})$ (*)	$O(u)$
AF-FMI [24]	$uH_k(T) + o(u \log \sigma)$ (§)	$O(u \log u(1 + \frac{\log \sigma}{\log \log u}))$
LZ-index (original) [55, 56]	$O(u \log u)$	$O(u \log \sigma)$
LZ-index (our early result) [4]	$(4 + \epsilon)uH_k(T) + o(u \log \sigma)$ (‡)	$O(\sigma u)$
LZ-index (**this paper**)	$4uH_k(T) + o(u \log \sigma)$ (‡)	$O(u(\log \sigma + \log \log u))$
Reduced LZ-index **a** (**this paper**)	$(1 + \epsilon)uH_k(T) + o(u \log \sigma)$ (‡)	$O(u(\log \sigma + \log \log u))$
Reduced LZ-index **b** (**this paper**)	$(2 + \epsilon)uH_k(T) + o(u \log \sigma)$ (‡)	$O(u(\log \sigma + \log \log u))$
Reduced LZ-index **c** (**this paper**)	$(3 + \epsilon)uH_k(T) + o(u \log \sigma)$ (‡)	$O(u(\log \sigma + \log \log u))$

(*) This is $o(u \log u)$ bits if $\log \sigma = o(\log u)$. (†) This is $O(u \log \sigma)$ in the worst case. (§) For any $k \leqslant \alpha \log_\sigma u$ and any constant $0 < \alpha < 1$. (‡) For any $k = o(\log_\sigma u)$. In fact this is an upper bound, as the real space is $c|LZ|(1 + o(1))$, for various constants c.

Since, depending on the scenario, we might or might not have to read the text from disk, and we might or might not have to write the final index to disk, and because those costs are fixed, we will not mention them. Yet, in the reduced-main-memory scenarios we will use the disk to read/write intermediate results, and in this case we will also consider the amount of extra I/O performed. When accessing the disk, we assume the standard model [69] where a disk page of B bits is transferred to/from secondary storage with each access. Finally, the space required by the text is not accounted for in the space required by the indexing algorithms. If it resides on disk one can process it sequentially so it does not require any significant main memory. Moreover, in most of our algorithms one could erase the text at an early stage of the construction.

2.2 Empirical Entropy

A concept related to text compression is that of the k-th order empirical entropy of a sequence of symbols $T[1..u]$ over an alphabet of size σ, denoted by $H_k(T)$ [47]. 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.

2.3 Lempel-Ziv Compression

The Lempel-Ziv compression algorithm of 1978 (usually named LZ78 [73]) 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 $LZ = b_1 \ldots b_r$, 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 LZ, i.e. $LZ = b_1 \ldots b_r b_{r+1}$, 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, $T = B_0 \dots B_n$, such that $B_0 = \varepsilon$ (the empty string). We say that i is the *phrase identifier* corresponding to B_i, for $0 \leqslant i \leqslant n$.

Therefore the output size of the LZ78 compression algorithm is $|LZ| = n(\log n + \log \sigma)$. Although we will usually give detailed space results, when we summarize we will assume $\log \sigma = o(\log n)$, and thus $|LZ| = n \log n(1 + o(1))$. We now point out some useful properties.

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. Hence, a natural way to represent the set of strings $B_0, \dots, B_n$ is a trie, which we call *LZTrie*.

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

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.

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 will be 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. 4(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.

The compression algorithm is $O(u)$ time in the worst case and 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).

Property 3 ([73]). It holds that $\sqrt{u} \leqslant n \leqslant \frac{u}{\log_\sigma u}$. This implies $\log n = \Theta(\log u)$ and $n \log u \leqslant u \log \sigma$.

We shall use the following result of Kosaraju and Manzini [38] to bound the size of the output of the LZ78 parsing of text T in terms of the k-th order empirical entropy of T.

Lemma 1 ([38]). *It holds that* $n \log n \leqslant uH_k(T) + O(u \frac{1 + k \log \sigma}{\log_\sigma u})$ *for any* k.

In particular, for $k = o(\log_\sigma u)$, we have that in the worst case $n \log n = uH_k(T) + o(u \log \sigma)$. This requires assuming $\log \sigma = o(\log u)$ to allow for $k > 0$, i.e., high-order compression. Note this is equivalent to the $\log \sigma = o(\log n)$ simplifying assumption we have mentioned above.

We also prove the following result, which is related to Lemma 1 and shall be useful in our work.

Lemma 2. *It holds that* $n \log u \leqslant uH_k(T) + o(u \log \sigma)$ *for any* $k = o(\log_\sigma u)$.

Proof. Note that $n \log u = n \log n + n \log \frac{u}{n}$. By Lemma 1, the former term is at most $uH_k(T) + o(u \log \sigma)$, for any $k = o(\log_\sigma u)$. The latter term is increasing in n for $n \leqslant u/e$, so for $n = o(u)$ we can pessimistically replace n by $\frac{u}{\log_\sigma u}$ due to Property 3. This yields $n \log \frac{u}{n} \leqslant \frac{u}{\log_\sigma u} \log \log_\sigma u = o(u \log \sigma)$. If, instead, $n = \Theta(u)$ then, again by Property 3, we have that $\log \sigma = \Theta(\log u)$ and the latter term is $O(n) = o(u \log \sigma)$. □

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 some results on succinct data structures, which are needed in our work.

Data Structures for $rank$ and $select$ Given a bit vector $\mathcal{B}[1..n]$, we define the operation $rank_0(\mathcal{B}, i)$ (similarly $rank_1$) as the number of 0s (1s) occurring up to the i-th position of $\mathcal{B}$. The operation $select_0(\mathcal{B}, i)$ (similarly $select_1$) is defined as the position of the i-th 0 (i-th 1) in $\mathcal{B}$. We assume that $select_0(\mathcal{B}, 0)$ always equals 0 (similarly for $select_1$). These operations can be supported in constant time and requiring $n + o(n)$ bits [51], or even $nH_0(\mathcal{B}) + o(n)$ bits [62]. The $o(n)$ overhead can be made as small as $O(n/\log^c n)$ for any constant c [61].

There exist a number of practical data structures supporting $rank$ and $select$, like the one by González et al. [23], Kim et al. [37], Okanohara and Sadakane [60], etc. Among these, the first [23] is very (perhaps the most) efficient in practice to compute $rank$, requiring little space on top of the sequence itself. Operation $select$ is implemented by binary searching the directory built for operation $rank$, and thus without requiring any extra space for that operation (yet, the time for $select$ becomes $O(\log n)$).

Given a sequence $S[1..u]$ over an alphabet Σ, we generalize the above definition to $rank_c(S, i)$ and $select_c(S, i)$ for any $c \in \Sigma$. If $\sigma = O(\text{polylog}(u))$, the solution of Ferragina et al. [19] allows one to compute both $rank_c$ and $select_c$ 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. Mäkinen and Navarro [44] showed how to handle in addition insertions and deletions on bitmaps and sequences, achieving $O(\log u \log \sigma)$ time for all operations. This was later improved [24] to $O(\log u(1 + \frac{\log \sigma}{\log \log u}))$, always within the same space bounds.

Data Structures for Searchable Partial Sums Given an array $A[1..n]$ of n integers of k' bits each, a data structure for searchable partial sums allows one to retrieve $A[i]$ and supports operations $Sum(A, i)$, which computes $\sum_{j=1}^{i} A[j]$; $Search(A, i)$, which finds the smallest j' such that $Sum(A, j') \geqslant i$; $Update(A, i, \delta)$, which sets $A[i] \leftarrow A[i] + \delta$; $Insert(A, i, e)$, which adds a new element e to the set between elements $A[i-1]$ and $A[i]$; and $Delete(A, j)$, which deletes $A[j]$.

A simple data structure [44] supports all these operations in $O(\log n)$ worst-case time, and requires $nk' + o(nk')$ bits of space. For this work, it is interesting that the space can be made $nk' + O(n)$ bits.

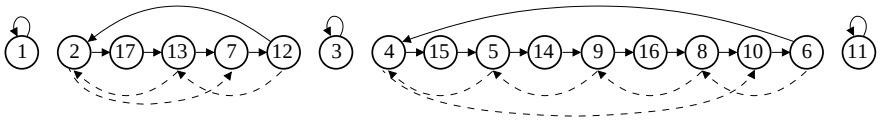
Succinct Representation of Permutations The problem here is to represent a permutation π of $\{1, \ldots, n\}$, such that we can compute both $\pi(i)$ and its inverse $\pi^{-1}(j)$ in constant time and using as little space as possible. A natural representation for π is to store the values $\pi(i)$, $i = 1, \ldots, n$, in an array of $n \log n$ bits.

An efficient solution to computing $\pi^{-1}(j)$ within little extra space [52] is based on the *cycle notation* of a permutation. We explain it in some detail, as this will be necessary later in this work. The cycle for the i-th element of π is formed by elements i, $\pi(i)$, $\pi(\pi(i))$, and so on until i is found again. Notice that every

element occurs in exactly one cycle of π. For example, the cycle notation for permutation π of Fig. 2(a) is shown in Fig. 2(b). So, we compute $\pi^{-1}(j)$ looking for j only in its cycle: $\pi^{-1}(j)$ is just the value "pointing" to j in the diagram. To compute $\pi^{-1}(13)$ in our example, we start at position 13, then move to position $\pi(13) = 7$, then to $\pi(7) = 12$, then to $\pi(12) = 2$, then to $\pi(2) = 17$, and as $\pi(17) = 13$ we conclude that $\pi^{-1}(13) = 17$. Since there are no bounds for the size of a cycle, this takes $O(n)$ time in the worst case. Yet, it can be improved for a more efficient computation of $\pi^{-1}(j)$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\pi[i]$	1	17	3	15	14	4	12	10	16	6	11	2	7	9	5	8	13

(a) An example of permutation π.



(b) Cycle notation of permutation π.

Fig. 2. Cycle representation for a given permutation π. Each solid arrow $i \to j$ in the diagram means $\pi(i) = j$. Dashed arrows represent backward pointers.

Given $0 < \epsilon < 1$, we create subcycles of size $O(1/\epsilon)$ by adding a *backward pointer* out of $O(1/\epsilon)$ elements in each cycle of π. Dashed arrows in Fig. 2(b) show backward pointers for $1/\epsilon = 2$. To compute π^{-1} we follow the cycles as before, yet now we follow a backward pointer if we reach it. We store the backward pointers compactly in an array of $\epsilon n \log n$ bits. We mark the elements having a backward pointer by using a bit vector supporting $rank$ queries, which also help us find the backward pointer corresponding to an element (see Munro et al. [52] for details). The whole solution uses $(1+\epsilon)n \log n + n + o(n)$ bits.

Next we present a result that shall be useful later for our purposes of constructing the LZ-index for a text T. Our result states that any permutation π can be inverted in-place in linear time and using only n extra bits of space. This can be seen as a particular case of *rearranging a permutation* [20], where we are given an array and a permutation, and want to rearrange the array according to the permutation.

Lemma 3. *Given a permutation π of $\{1, \ldots, n\}$ represented by an array using $n \log n$ bits of space, we can compute on the same array the inverse permutation π^{-1} in $O(n)$ time and requiring n bits of extra space.*

Proof. Let $A_\pi[1..n]$ be an auxiliary bit vector requiring n bits of storage, which is initialized with all zeros (this is just the raw bit vector, no additional data structure for $rank$ and $select$ is added). Let π be the array representing the permutation, using $n \log n$ bits of space. The idea to construct π^{-1} is to use the cycle structure of π to reverse the "arrows" conforming the cycles (i.e., "$i \to j$" in a cycle of π, which means $\pi[i] = j$, now becomes "$i \leftarrow j$", which means $\pi^{-1}[j] = i$). So, the main idea is to regard the cycles of π as "linked lists". Thus, constructing π^{-1} is a matter of reversing the pointers in the lists, and therefore we shall need three auxiliary pointers to do that job. We follow the cycles of π, using A_π to mark with a `1` those positions which have been already visited during this process.

We start with the cycle at position $a \leftarrow 1$, and traverse it from position $p \leftarrow \pi[a]$. We then set $b \leftarrow \pi[p]$, $\pi[p] \leftarrow a$ (i.e., we store the position a which brings us to the current one), and $A_\pi[p] \leftarrow$ `1`. Then we move to position $a \leftarrow p$, set $p \leftarrow b$, and repeat the process again, stopping as soon as we find a `1` in A_π. Then we

try with the cycle starting at position $p+1$, which is the next one after the position that started the previous cycle, and follow it just if the corresponding bit in A_π is `0`.

Thus, each element in the permutation is visited twice: elements starting a cycle are visited at the beginning and at the end of the cycle, while elements in the middle of a cycle are visited when traversing the cycle to which they belong, and when trying to start a cycle from them. Thus, the overall time is $O(n)$, and we use n extra bits on top of the space of π, and the lemma follows. □

2.5 Succinct Representation of Trees

Given a (general and unlabeled) tree with n nodes, there exist a number of succinct representations requiring $2n + o(n)$ bits. Since the number of distinct trees of n nodes is $C_n = \frac{1}{n+1}\binom{2n}{n} = \Theta(4^n/n^{3/2})$, this is close to the information-theoretic lower bound of at least $\log C_n = 2n - \Theta(\log n)$ bits.

Balanced Parentheses The problem of representing a sequence of balanced parentheses is highly related to the succinct representation of trees [53]. Given a sequence par of $2n$ balanced parentheses, we want to support the following operations on par: $findclose(par, i)$, which given an opening parenthesis at position i, finds the position of the matching closing parenthesis; $findopen(par, j)$, which given a closing parenthesis at position j, finds the position of the matching opening parenthesis; $excess(par, i)$, which yields the difference between the number of opening and closing parentheses up to position i; and $enclose(par, i)$, which given a parentheses pair whose opening parenthesis is at position i, yields the position of the opening parenthesis corresponding to the closest matching parentheses pair enclosing the one at position i.

Munro and Raman [53] show how to compute all these operations in constant time and requiring $2n + o(n)$ bits of space. They also show one of the main applications of maintaining a sequence of balanced parentheses: the succinct representation of general trees, with the so-called BP representation. Among the practical alternatives, we have the representation of Geary et al. [22], the one of Sadakane and Navarro [67], and the one by Navarro [55, Section 6.1]. The latter has shown to be very effective for representing LZ-indexes [56, 3].

DFUDS **Tree Representation** To get this representation, named after Depth-First Unary Degree Sequence [8], we perform a preorder traversal on the tree, and for every node reached we write its degree in unary using parentheses. For example, a node of degree 3 reads '`(((`' under this representation. Notice that a leaf is represented by '`)`'. 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.

This representation requires $2n+o(n)$ bits, and supports operations $parent(x)$ (which gets the parent of node x), $child(x, i)$ (which gets the i-th child of node x), $subtreesize(x)$ (which gets the size of the subtree of node x, including x itself), $degree(x)$ (which gets the degree, i.e., the number of children, of node x), $childrank(x)$ (which gets the rank of node x within its siblings [34]), and $ancestor(x, y)$ (which tells us whether node x is an ancestor of node y), all in $O(1)$ time. If we assume that par represents the DFUDS sequence of the tree, then we have:

$$parent(x) \equiv select_)(par, rank_)(par, findopen(par, x-1))) + 1$$
$$child(x, i) \equiv findclose(par, select_)(par, rank_)(par, x) + 1) - i) + 1$$

Operation $depth(x)$ (which gets the depth of node x in the tree) can also be computed in constant time on DFUDS by using the approach of Jansson et al. [34], requiring $o(n)$ extra bits.

Given a node 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, $preorder(x) \equiv rank_)(par, x-1)$. Given a preorder position p, the corresponding node is computed by $selectnode(p) \equiv select_)(par, p) + 1$.

Representing σ-ary Trees with DFUDS For cardinal trees (i.e., where each node has at most σ children, labeled by distinct symbols 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 compute operation $child(x, \alpha)$ (which gets the child of node x with label $\alpha \in \{1, \ldots, \sigma\}$) efficiently. In Fig. 4(c) we show the DFUDS representation of *LZTrie* for our running example (plus an array ids with the phrase identifiers).

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

Thus, the time for operation $child(x, \alpha)$ depends on the representation we use for $rank_\alpha$ and $select_\alpha$ queries (see Section 2.4). Notice that $child(x, \alpha)$ 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 using any extra space on top of array $letts$. The scheme we have presented to represent $letts$ is slightly different from the original one [8], which achieves $O(1)$ time for $child(x, \alpha)$ for any σ. However, our method is simpler to build, since the original one is based on perfect hashing, which is expensive to construct.

3 The LZ-index Data Structure

3.1 Definition of the Data Structures

Assume that the text $T[1..u]$ has been compressed using the LZ78 algorithm into $n + 1$ phrases $T = B_0 \ldots B_n$, as explained in Section 2.3. The data structures that conform the LZ-index are [55, 56]:

1. *LZTrie*: is the trie formed by all phrases $B_0 \ldots B_n$. Given the properties of LZ78 compression, this trie has exactly $n + 1$ nodes, each one corresponding to a phrase B_i.
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 $\{(revpreorder(t), preorder(t+1)), t \in 0 \ldots n-1\}$ in this structure, where $revpreorder(t)$ is the *RevTrie* preorder of node for phrase t (considering only non-empty nodes in the preorder enumeration), and $preorder(t+1)$ is the *LZTrie* preorder for phrase $t+1$. For each such point, the corresponding t value is stored.

3.2 Succinct Representation of the Data Structures

The data structures that compose the LZ-index are built and represented as follows.

LZTrie. For the construction of *LZTrie* we traverse the text and at the same time build a trie representing the Lempel-Ziv phrases, spending (as usual) one pointer per parent-child relation. At step t (assume $B_t = B_\ell \cdot c$), we read the text that follows and step down the trie until we cannot continue. At this point we create a new trie leaf (child of the trie node of phrase ℓ, by symbol c, and assigning the leaf phrase number t), go to the root again, and go on with step $t+1$ to read the rest of the text. The process completes when the last phrase finishes with the text terminator "\$". In Fig. 4(a) we show the Lempel-Ziv trie for the running example, using pointers. After we build the trie, we can erase the text as it is not anymore necessary, since we have now enough information to build the remaining index components.

Then we build the final representation of the topology of *LZTrie*, bitmap par, using the parentheses representation of Munro and Raman [53], yet newer versions of the LZ-index [7] use the DFUDS representation [8]. We also create the array $ids[1..n]$, storing the LZ78 phrase identifiers in preorder, and $letts[1..n]$, storing the symbols that label the trie edges, in preorder. The final size is $n \log n + n \log \sigma + O(n)$ bits.

Node. Once *LZTrie* is built, we free the space of the pointer-based trie and build $Node$. This is just an array with the n nodes of *LZTrie*. If the i-th position of the ids array corresponds to the j-th phrase identifier (i.e., $ids[i] = j$), then the j-th position of $Node$ stores the position of the i-th node within the balanced parentheses. As there are $2n$ parentheses, $Node$ requires $n \log 2n = n \log n + O(n)$ bits.

RevTrie. To construct *RevTrie* we traverse *LZTrie* in preorder, generating each LZ78 phrase B_i stored in *LZTrie* in constant time, and then inserting it into a *trie of reversed strings* (represented with pointers). For simplicity, empty unary paths are not compressed in the pointer-based trie. When we finish, we traverse the trie in preorder and represent the trie topology of *RevTrie* in bitmap $rpar$, the phrase identifiers in array $rids$, and the labels of the edges in array $rletts$. Empty unary nodes are removed only at this step, and so the final number n' of nodes in *RevTrie* satisfies $n \leqslant n' \leqslant 2n$.

Notice that if we use $n' \log n$ bits for the $rids$ array, then in the worst case *RevTrie* requires $2n \log n + O(n \log \sigma)$ bits of storage, which would increase the space usage of the index. Instead, we can represent the $rids$ array with $n \log n$ bits (i.e., only for the non-empty nodes), plus a bitmap of $2n + o(n)$ bits supporting $rank$ queries in $O(1)$ time [51]. The j-th bit of the bitmap is `1` if the node represented by the j-th opening parenthesis is not an empty node, otherwise the bit is `0`. The $rids$ index corresponding to the j-th opening parenthesis is $rank_1(j)$. Using this representation, *RevTrie* requires at most $n \log n + 2n \log \sigma + O(n)$ bits of storage. This was unclear in the original LZ-index paper [55, 56].

Range. The data structure of Chazelle [12] permits two-dimensional range searching in a grid of n pairs of integers in the range $[1..n] \times [1..n]$. This structure can be represented with $n \log n + O(n)$ bits of space. We explain the simpler case, which is the one that arises in our work, where the points represent a permutation of $\{1, \ldots, n\}$ [43], i.e., there is exactly one point with first coordinate i for any $1 \leqslant i \leqslant n$, and one point with second coordinate j for any $1 \leqslant j \leqslant n$.

To construct *Range*, we sort the set by the second coordinate j, and then divide the set according to the first coordinate i, to form a perfect binary tree where each node handles an interval of the first coordinate i, and thus knows only the points whose first coordinate falls in that interval. The root handles the points with first coordinate within $[1..n]$ (i.e., all), and the children of a node handling the interval $[i..i']$ are associated to $[i..\lfloor (i+i')/2 \rfloor]$ and $[\lfloor (i+i')/2 \rfloor + 1..i']$. Leaves handle intervals of the form $[i..i]$.

Every tree node v is then represented with a bit vector B_v indicating for each point handled by v whether the point belongs to the left or right child. In other words, $B_v[r] = 0$ iff the r-th point handled by node v (in the order given by the second coordinate j) belongs to the left child. Every level of the tree is represented as a single bit vector of n bits, using data structures for constant-time $rank$ and $select$ [51], which are needed to support the search (as well as, given a node, finding the corresponding starting position within the level, see Mäkinen and Navarro [43] for more details). Thus, we only need $O(\log n)$ pointers to represent the levels of the tree, avoiding in this way the need to store the pointers that represent the balanced tree. The total $o(n \log n)$ extra space for supporting $rank$ and $select$ over all the bitmaps can be made $O(n)$ by using Pătraşcu's representation [61].

This data structure supports counting the number of points that lie within a two-dimensional range in $O(\log n)$ time, as well as reporting the occ points inside the search range in $O((1 + occ) \log n)$ time [43].

$RNode$. In the practical implementation of the LZ-index [55, 56], the ***Range*** data structure is replaced by $RNode$, which is a mapping from phrase identifiers to their node in ***RevTrie***. After we free the space of the pointer-based reverse trie, we build $RNode$ from $rids$ in the same way as $Node$ is built from ids. It is important to note that, by using $RNode$ instead of $Range$, the LZ-index cannot provide worst-case guarantees at search time, but just average-case guarantees. However, this approach has shown to be effective in practice since it has a good average-case search time [56].

Time Performance. The original LZ-index locates the occ occurrences of a pattern of length m in worst-case time $O(m^3 \log \sigma + (m + occ) \log n)$. The practical variant using $RNode$ instead of $Range$ requires average time $O(m^2(1 + \frac{\log \sigma}{\log \log u}) + \frac{u}{\sigma^{m/2}})$, which is $O(m^2(1 + \frac{\log \sigma}{\log \log u}))$ for $m \geqslant 2 \log_\sigma u$, if we assume the representation for $letts$ given in Section 2.4.

3.3 Indexing and Final Space

Using the succinct representations, the four structures that conform the LZ-index add up to at most $4n \log n + 3n \log \sigma + O(n)$ bits of space. According to Lemma 1, this is at most $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

The LZ-index can be built in $O(u \log \sigma)$ time [55]. However, a large amount of storage is needed to construct it [56], mainly because of the pointer representation of the tries used at construction time. In theory, representing empty unary nodes in ***RevTrie*** requires worst-case space $O(u \log u)$ bits. By compacting it, the space would become $O(n \log u)$, yet still with a large constant due to the use of pointers.

In the original experiments [56], the largest extra space needed to build ***LZTrie*** is that of the pointer-based trie, which is 1.7–2.0 times the text size. However, as expected, the peak space usage is that of building the pointer-based reverse trie, which is in some cases 4 times the text size. In practice representing the empty unary nodes does not add much to the space, but the reverse trie has a number of empty non-unary nodes, which cannot be compacted and sharply increase the space usage. The overall indexing space is 4.8–5.8 times the text size for English text, and 3.4–3.7 times the text size for DNA. As a comparison, the construction of a plain suffix array without any extra data structure requires 5 times the text size [48].

3.4 Reduced Space Versions of the LZ-index

New versions of the LZ-index have been introduced recently [6, 7, 5], which require less space than the original LZ-index, in some cases also improving its search performance. The approach introduced to reduce

the space is the so-called *navigational-scheme* approach, which consists in regarding the original LZ-index (in particular, the version using $RNode$ instead of $Range$, see Section 3.2) as a navigation structure which allows us moving among the LZ-index components (i.e., *LZTrie* nodes, *LZTrie* preorders, phrase identifiers, *RevTrie* nodes, and *RevTrie* preorders). All searches are carried out by navigating among these components.

In Fig. 3 we illustrate the original LZ-index navigation scheme, where the four main structures of the index are shown as solid arrows:

$$\begin{aligned} Node:&\ \text{phrase identifier} \mapsto \textit{LZTrie}\ \text{node};\\ RNode:&\ \text{phrase identifier} \mapsto \textit{RevTrie}\ \text{node};\\ ids:&\ \textit{LZTrie}\ \text{preorder} \mapsto \text{phrase identifier};\ \text{and}\\ rids:&\ \textit{RevTrie}\ \text{preorder} \mapsto \text{phrase}. \end{aligned}$$

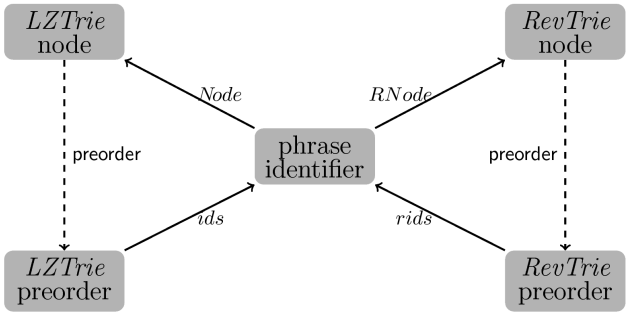


Fig. 3. The original LZ-index navigation structures over index components.

As we have seen in Section 2.5 for the DFUDS representation, trie nodes and the corresponding preorders are "connected" by means of $preorder$ and $selectnode$ operations, so we have a navigation scheme that allows us moving back and forth from any index component to any other. We will subindicate these operations with lz if they refer to *LZTrie* and with r if they refer to *RevTrie*.

This approach allows us to study the redundancy introduced by the original index. As a result, several new reduced space schemes have been introduced [5], allowing the same navigation yet requiring less space.

Scheme 2 The so-called Scheme 2 of the LZ-index [5] represents the components ids : *LZTrie* preorder $\mapsto$ phrase identifier; $rids^{-1}$: phrase identifier $\mapsto$ *RevTrie* preorder; and R : *RevTrie* preorder $\mapsto$ *LZTrie* preorder. The original search algorithm remains the same, since we can simulate the missing data structures: $rids(i) \equiv ids[R[i]]$, $\textit{RNode}(i) \equiv selectnode_r(rids^{-1}[i])$, and $\textit{Node}(i) \equiv selectnode_{lz}(R[rids^{-1}[i]])$, all in constant time. The space requirement [5] is $3n \log n + 3n \log \sigma + 2n \log\log u + O(n) + o(u)$ bits. According to Lemma 1, this is $3uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$. Although this scheme does not provide worst-case guarantees at search time, it has shown to be efficient in practice, outperforming competing indexes in many real-life scenarios [5]. Thus, we are also interested in its space-efficient construction in order to extend its applicability. There exists another alternative requiring the same space as Scheme 2, which shall be disregarded in this paper, since Scheme 2 outperforms it in most practical cases [5].

Scheme 3 This LZ-index variant represents ids : *LZTrie* preorder $\mapsto$ phrase identifier; ids^{-1} : phrase identifier $\mapsto$ *LZTrie* preorder; $rids$: *RevTrie* preorder $\mapsto$ phrase identifier; and $rids^{-1}$: phrase identifier $\mapsto$ *RevTrie* preorder. The missing data structures can be simulated as: *Node*$(i) \equiv selectnode_{lz}(ids^{-1}(i))$ and *RNode*$(i) \equiv selectnode_r(rids^{-1}(i))$, all in $O(1/\epsilon)$ time. Since arrays ids and $rids$ are represented with the data structure for permutations of Munro et al. [52], they require a total space of $(2+\epsilon)n\log n+2n+o(n)$ bits, for any $0<\epsilon<1$. The overall space requirement is $(2+\epsilon)n\log n+3n\log\sigma+2n\log\log u+O(n)+o(u)$ bits, which according to Lemma 1 is $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits, for any $k=o(\log_\sigma u)$. This scheme has also shown to be efficient in practice, outperforming competing indexes in many real-life scenarios and being able to require less space than Scheme 2 (yet, when requiring the same space, Scheme 2 usually outperforms Scheme 3).

Scheme 4 This variant represents the following data: ids : *LZTrie* preorder $\mapsto$ phrase identifier; ids^{-1} : phrase identifier $\mapsto$ *LZTrie* preorder; R : *RevTrie* preorder $\mapsto$ *LZTrie* preorder; and R^{-1} : *LZTrie*preorder $\mapsto$ *RevTrie* preorder. The missing arrays are simulated as $rids(i) \equiv ids[R[i]]$, *Node*$(i) \equiv selectnode_{lz}(ids^{-1}(i))$, and *RNode*$(i) \equiv selectnode_r(R^{-1}(ids^{-1}(i)))$, all of which take $O(1/\epsilon)$ time. The inverse permutations are also represented by the data structure of Munro et al. [52]. Hence, the space requirement is $(2+\epsilon)n\log n+3n\log\sigma+2n\log\log u+O(n)+o(u)$, which according to Lemma 1 is $(2+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any $k=o(\log_\sigma u)$.

Although Scheme 3 outperforms Scheme 4 in most practical scenarios [5], Scheme 4 is interesting by itself since its space can be reduced even more, achieving interesting theoretical results [7]. The idea is to replace array R by a data structure allowing us to compute any $R[i]$, yet requiring less than the $n\log n$ bits required by the original array. Thus, for every *RevTrie* preorder $1 \leqslant i \leqslant n$ we define function φ such that $\varphi(i)=R^{-1}(parent_{lz}(R[i]))$, and $\varphi(0)=0$ (operation $parent_{lz}$ is the parent operation in *LZTrie*, yet working on preorders instead of on nodes as originally defined). Function φ works as a *suffix link* in *RevTrie*: given a *RevTrie* node with preorder i representing string ax (for $a\in\Sigma, x\in\Sigma^*$), the *RevTrie* node with preorder $\varphi(i)$ represents string x. An important result is that $R[i]$ can be computed by means of function φ [7]. We also sample ϵn values of R in such a way that the computation of $R[i]$ (by means of φ) takes $O(1/\epsilon)$ time in the worst case.

Function φ has the same properties as function Ψ of Compressed Suffix Arrays [26, 65], thus this can be also compressed to $n\log\sigma+O(n\log\log\sigma)$ bits of space (in this paper we show how to compress it to $n\log\sigma+O(n)$ bits and still compute any entry in constant time). The computation of R^{-1} is supported also in $O(1/\epsilon)$ time, by reverting the process used to compute R. For this, function φ' is defined as *Weiner links* [70] in *RevTrie* [4]. Function φ' is supported by two arrays, $S_W[1..n]$ (of $n\log\sigma$ bits storing, for every *RevTrie* node, in preorder, the symbols by which the node has Weiner links defined), and $V_W[1..2n]$ (a bit vector storing, for every *RevTrie* node, in preorder, the bit sequence 10^d such that d is the number of Weiner links defined for the node). The space requirement is $\epsilon n\log n+n\log\sigma+O(n)$ bits. By rewriting 2ϵ as ϵ, which does not change time complexities, we have:

Lemma 4 ([7]). *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists a Lempel-Ziv compressed full-text self-index requiring $(1+\epsilon)n\log n+5n\log\sigma+O(n)$ bits of space, for any $0<\epsilon<1$. This is $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ for any $k=o(\log_\sigma u)$. The index is*

[4] Given a *RevTrie* node v representing string $x\in\Sigma^*$, the Weiner link for v and symbol $a\in\Sigma$ is a pointer to the *RevTrie* node representing string ax.

able to locate (and count) the occ occurrences of a pattern $P[1..m]$ in text T in $O(\frac{m^2}{\epsilon}(1+\frac{\log\sigma}{\log\log u})+\frac{u}{\epsilon\sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon}(1+\frac{\log\sigma}{\log\log u}))$ if $m \geqslant 2\log_\sigma u$.

Thus the LZ-index can be represented with almost optimal space under the LZ78 compression model (recall that $|LZ| = n\log n + n\log\sigma$), and also under the empirical entropy model $H_k(T)$ in the (usual) case $H_k(T) = \Theta(\log\sigma)$). Yet, we cannot provide worst-case guarantees at search time within this space.

We can get such worst-case guarantees at search time by adding *Range*, the two-dimensional range search data structure, as defined for the original LZ-index. This requires $n\log n + O(n)$ extra bits of space.

Lemma 5 ([7]). *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists a Lempel-Ziv compressed full-text self-index requiring $(2+\epsilon)n\log n + 5n\log\sigma + O(n)$ bits of space, for any $0 < \epsilon < 1$. This is $(2+\epsilon)uH_k(T) + o(u\log\sigma)$ bits for any $k = o(\log_\sigma u)$. The index is able to locate the occ occurrences of a pattern $P[1..m]$ in T in $O(\frac{m^2}{\epsilon}(1+\frac{\log\sigma}{\log\log u})+(m+occ)\log u)$ worst-case time; count the number of pattern occurrences in time $O(\frac{m^2}{\epsilon}(1+\frac{\log\sigma}{\log\log u})+m\log u+occ)$; and determine whether pattern P exists in T in $O(\frac{m^2}{\epsilon}(1+\frac{\log\sigma}{\log\log u})+m\log u)$ time.*

Finally, we can add the *Alphabet-Friendly FM-index* [19] of text T to this index, to get:

Lemma 6 ([7]). *Let text $T[1..u]$ be a text over an alphabet of size σ. Then there exists a Lempel-Ziv compressed full-text self-index requiring $(3+\epsilon)uH_k(T) + o(u\log\sigma)$ bits of space, for any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$, which is able to locate the occ occurrences of a pattern $P[1..m]$ in T in $O((m(1+\frac{\log\sigma}{\log\log u})+\frac{occ}{\epsilon})\log u)$ worst-case time; and count the number of pattern occurrences (or determine if the pattern exists or not in the text) in $O(m(1+\frac{\log\sigma}{\log\log u}))$ time.*

Note we have used only the H_k-related notation in this latter lemma as it contains an FM-index, whose space is not related to $|LZ|$. Note also that the practical Schemes 2–4 contain a term space of the form $O(n\log\log u) + o(u)$, the latter of which is not always $o(|LZ|)$. These terms owe to the representation of the Patricia skips [49] in the reverse trie. While not strictly necessary in theory (and hence not present in the theoretical Lemmas 4 to 6), in practice these skips speed up the index considerably and do not increase much the space. Similarly, the $5n\log\sigma$ space of the last lemmas can be reduced to $3n\log\sigma$ by not storing the letters of *RevTrie*. These can in theory be obtained from *LZTrie*, but the $1+\frac{\log\sigma}{\log\log u}$ time factor (coming from the representation of $rletts$ using the structure of Section 2.4) worsens to $\log\sigma$. Also, in practice it is a good idea to maintain the symbols explicitly.

4 Space-Efficient Construction of the LZ-index

The LZ-index is a compressed full-text self-index, and as such it allows large texts to be indexed and stored in main memory. However, the construction process requires a large amount of main memory, mainly to support the pointer-based tries used to build the final versions of *LZTrie* and *RevTrie* (recall Section 3.3). So our problem is: given a text $T[1..u]$ over an alphabet of size σ, we want to construct the LZ-index for T using as little space as possible and within reasonable time. We aim at an efficient algorithm to build those tries in little memory, by replacing the pointer-based tries with space-efficient data structures that support insertions. These can be seen as hybrids between pointer-based tries and the final succinct representations.

Note that we could simply use succinct dynamic trees [11] and dynamic sequences [44] to create the tries. However, the construction time would become at best $O(u\log n(1+\frac{\log\sigma}{\log\log n}))$.

Our early space-efficient construction algorithm for the LZ-index [4] partitions the tree into moderately-sized connected components, which are updated in naive form. As a result, it has a construction time of the form $O(\sigma u)$, which is impractical for moderately-large alphabets.

In the sequel we shall achieve $O(u(\log \sigma + \log\log u))$ time by combining the best from both ideas, i.e., using advanced succinct dynamic representations on moderately-sized connected components of the tries.

In Sections 4.1 to 4.5 we assume that we have enough main memory to store the final LZ-index. In Section 4.6 we study how to manage the memory dynamically, which is an important aspect of dynamic data structures, using a standard model [63] of memory allocation. In Section 4.7, we shall adapt our algorithm to the cases in which there is not enough space to store the whole final index in main memory.

We show next how to space-efficiently construct the LZ-index components. From now on we assume $\sigma \geqslant 2$, as otherwise the whole indexing problem is trivial.

4.1 Space-Efficient Construction of *LZTrie*

The space-efficient construction of *LZTrie* is based on a compact representation supporting a fast incremental construction as we traverse the text. In either the BP or DFUDS representations, the insertion of a new node at any position of the sequence implies to rebuild the sequence from scratch, which is expensive. To avoid this we define a *hierarchical representation*, such that we rebuild only a small part of the entire original sequence upon the insertion of a new node.

We incrementally cut the trie into disjoint *blocks* such that every block stores a subset of nodes representing a connected component of the whole trie. We arrange these blocks in a tree by adding some *inter-block* pointers, and thus the entire trie is represented by a tree of blocks.

If a node x is a leaf of a block p, but is not a leaf of the whole trie, then node x stores an inter-block pointer to the representation of its subtree. Let us say that this pointer points to block q. We say that q is a *child block* of p. In our representation, node x is also stored in block q, as a fictitious root node. Thus, every block is a tree by itself, which shall simplify the navigation as well as the management of each block. Thus every such fictitious node x has two representations: (1) as a leaf in block p; (2) as the root node of block q. Note that the number of extra nodes introduced by duplicating nodes equals the number of blocks in the representation (minus one). We not only enforce that the parent of any (non-fictitious) node is stored in the same block of the node, but also that all its sibling nodes are stored in the same block.

Rather than using a static representation for the trie blocks [4], which are rebuilt from scratch upon insertions, we represent each block by using dynamic data structures, which can be updated in time less than linear in the block size. We adapt the approach used by Arroyuelo [2] to represent succinct dynamic σ-ary trees: We first reduce the size of the problem by dividing the trie into small blocks, and then represent every block (i.e., smaller trie) with a dynamic data structure to avoid the total rebuilding of blocks upon updates.

Defining Block Sizes We divide *LZTrie* into blocks of N nodes each, where $N_m \leqslant N \leqslant N_M$, for minimum block size $N_m = \Theta(\log^2 u)$ nodes and maximum block size $N_M \geqslant 2\sigma N_m$ nodes. We also need $N_M = (\sigma \log u)^{O(1)}$, for example $N_M = \Theta(\sigma \log^3 u)$ (we do not show the roundings, but it should be clear that these values must be integers). Hence, notice that we shall have one inter-block pointer out of at least N_m nodes. Since each pointer is represented with $\log u$ bits, and since we have n nodes in the tree, we have $\frac{n}{N_m} \log u = O(n/\log u)$ bits overall for inter-block pointers. The definition of N_M, on the other hand, is such that it ensures that a block p has room to store at least the potential σ children of the block root (recall that sibling nodes must be stored all in the same block). Also, when a block overflows we should be able to

split the block into two blocks, each of size at least N_m. By defining N_M as we do, in the worst case (i.e., the case where the overflown block has the smallest possible size) the root of the block has some child with at least N_m nodes, as $N_M \geqslant 1 + \sigma N_m$. Thus, upon an overflow, we can create a new block of size at least N_m from such subtree, requiring little space for inter-block pointers and maintaining the properties of our data structure. The stricter factor 2 shall be useful for our amortized analysis of block partitioning, whereas the polylog upper bound is necessary to ensure that pointers within blocks are short enough.

Defining the Block Layout Each block p of N nodes consists of:

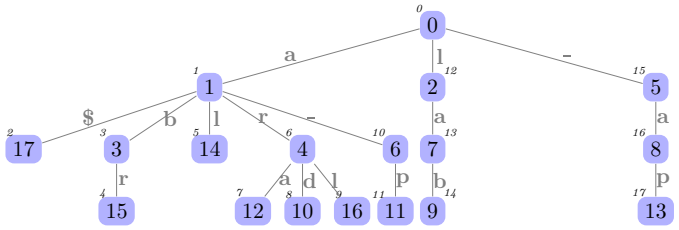
- The representation T_p of the topology of the block, using any suitable tree representation. In particular, we will use the DFUDS [8], which is particularly well suited for our goals.
- A bit-vector $F_p[1..N]$ (the *flags*) such that $F_p[j] = \texttt{1}$ iff the j-th node of T_p (in preorder) has an associated inter-block pointer. We shall represent F_p with a data structure for $rank$ and $select$ queries.
- $\log N_M$ bits to count the current number N of nodes stored in the block.
- The sequence $ids_p[1..N]$ of LZ78 phrase identifiers for the nodes of T_p, in preorder. Except for the *LZTrie* root, every block root is replicated as a leaf in its parent block, as explained. In that case we store the corresponding phrase identifier only in the leaf of the parent block. That is, fictitious roots in each block do not store phrase identifiers. We use $\log u$ bits per phrase identifier, instead of using $\log n$ bits as in the final representation of ids. This is because before constructing the LZ78 parsing of the text we do not know n, the number of phrase identifiers.
- The symbols ($letts_p$) labeling the edges in the block (the order of the symbols depends on the representation used for T_p, recall Section 2.5). Each symbol uses $\log \sigma$ bits of space.
- A variable number of inter-block pointers, stored in data structure ptr_p. The number of inter-block pointers varies from 0 to N, and it corresponds to the number `1`s in F_p.

In Fig. 4(b) we show an example of hierarchical representation of *LZTrie* for the running example text. If the subtree of the j-th node (in preorder) of block p is stored in block q, then q is a child block of p and the j-th flag in p has the value `1`. If the number of flags with value `1` before the j-th flag in p is h, then the h-th inter-block pointer of p points to q. Note that h can be computed as $rank_{\texttt{1}}(F_p, j)$.

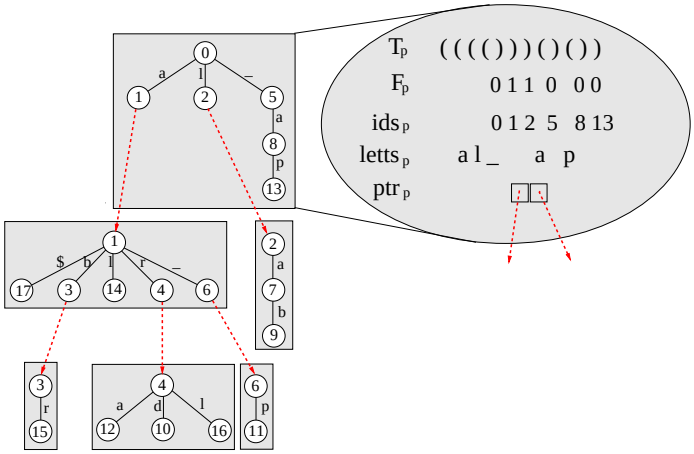
Since blocks are tries by themselves, inside a block p we use the traditional trie-like descent process, using operation $child_p(x, \alpha)$ on T_p. From now on we use the subscript p with the trie operations, to indicate operations which are local to a block p, i.e., disregarding the inter-block structure (e.g., $preorder_p$ computes the preorder of a node within block p, and not within the whole trie, and so on). When we reach a block leaf (with preorder j inside the block), we check the j-th flag in p. If $F_p[j] = \texttt{1}$ holds in that block, then we compute $h = rank_{\texttt{1}}(F_p, j)$ and follow the h-th inter-block pointer in p to reach the corresponding child block q. Then we follow the descent inside q as before. Otherwise, if $F_p[j] = \texttt{0}$, then we are in a leaf of the whole trie, and we cannot descend anymore.

We represent the above components for block p in the following way.

Representation of the Trie Topology, T_p To represent the trie topology of block p we use the data structure for dynamic balanced parentheses of Chan et al. [11] to represent the DFUDS [8] of the block. The main idea of Chan et al. is to divide the original parentheses sequence into segments S_i of $O(\log N)$ bits. Every segment S_i is stored in the leaves of a balanced binary tree T'_p, such that concatenating the leaves from left to right gives us back the original sequence T_p. Some information is stored in the internal nodes of T'_p in order to support the operations on the parentheses sequence, as well as support insertions and deletions of *pairs of*



(a) Lempel-Ziv Trie (*LZTrie*), represented in the traditional (pointer-based) way.



(b) Hierarchical representation of *LZTrie*, Nodes having an inter-block pointer are duplicated as the root of the child block. We show the actual succinct representation for the root block.

	0	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	27	28	29	30	31	32	33	34	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	

(c) DFUDS representation of *LZTrie*. The phrase identifiers are stored in preorder, and the symbols labeling the edges of the trie are stored according to a DFUDS traversal.

Fig. 4. Different representations of the Lempel-Ziv trie and related data structures for the running example.

matching parentheses. All the operations of Section 2.5 are supported in $O(\log N)$ time by navigating T'_p. In addition, we store in every internal node of T'_p the number of opening parentheses within the left subtree, as well as the total number of parentheses within the left subtree, such as in Mäkinen and Navarro [44], in order to support operations $rank_($, $rank_)$, $select_($, and $select_)$ over T_p in $O(\log N)$ time.

All these operations on the sequence of parentheses allow us to support the DFUDS operations (recall Section 2.5): $parent_p$, $child_p(x, i)$, $subtreesize_p$, $degree_p$, $preorder_p$, $selectnode_p$, etc., all of them in $O(\log N) = O(\log N_M)$ time. As we shall explain later in this section, the insertion of a new node in DFUDS can be simulated by inserting a new pair of matching parentheses in T_p, and thus we can handle it in a straightforward way with the data structure of Chan et al. [11]. Deletions of leaves are handled in a similar way. The space requirement is $O(N)$ bits per block, which adds up to $O(n)$ bits overall[5].

Representation of the Flags, F_p We represent the flags of block p in preorder and using a dynamic data structure for $rank$ and $select$ over a binary sequence [44]. It supports $rank$, $select$, and updates on F_p in $O(\log N)$ worst-case time, and requires $N+o(N)$ bits of space. This data structure can be connected with T_p via operations $preorder_p$ and $selectnode_p$: Given a node x in p, the corresponding flag is $F_p[preorder_p(x)]$. Given $F_p[j]$, on the other hand, the corresponding node in T_p is $selectnode_p(j)$. When we insert a new node in T_p, we insert a new flag (with value `0` because the new node is inserted with no related inter-block pointer) at the corresponding position (given by $preorder_p$). This data structure adds $n + o(n)$ extra bits to our representation. Arroyuelo [2] gives a more involved representation for F_p, requiring $o(n)$ bits, yet the one we are using here is simpler and still adequate for our purposes.

Representation of the Symbols, $letts_p$ We represent the symbols labeling the edges of the block according to a DFUDS traversal on T_p (see Section 2.5), yet this time we store them in differential form, except for the symbol of the first child of every node, which is represented in absolute form. We then represent this sequence of N integers of $k' = \log \sigma$ bits each with the dynamic data structure for searchable partial sums of Mäkinen and Navarro [44], which supports all the operations (including insertions and deletions) in time $O(\log N)$, and requires $Nk' + O(N) = N \log \sigma + O(N)$ bits of space. These add up to $n \log \sigma + O(n)$ bits.

We can connect $letts_p$ with T_p by using $rank_($ over T_p. Given a node x in T_p, the subsequence $letts_p$ $[rank_(T_p, x)..rank_(T_p, x) + degree_p(x) - 1]$ stores the symbols labeling the children of x. To support operation $child_p(x, \alpha)$, which shall be used to descend in the trie at construction time, we first compute $i \leftarrow rank_(T_p, x)$ to obtain the position in $letts_p$ for the first child of x. We then compute $s \leftarrow Sum(letts_p, i - 1)$, which is the sum of the symbols in $letts_p$ up to position $i - 1$ (i.e., the sum before the first child of x). To compute the position of symbol α within the symbols of the children of node x, we perform $j \leftarrow Search(letts_p, s + \alpha)$. Thus, the node we are looking for is the $(j - i + 1)$-th child of x, which can be computed by $child_p(x, j - i + 1)$, in $O(\log N)$ time overall. To make sure j is a valid answer, we use operation $degree_p(x)$ to check whether $j - i + 1$ is smaller or equal to the degree of x, and then we check whether $Sum(letts_p, j - i + 1) - s = \alpha$ actually holds.

Representation of the Phrase Identifiers, ids_p To store the phrase identifiers of the trie nodes, we define a list L_{ids_p} for block p, storing the identifiers in preorder. Given a new inserted node x in T_p, we must insert the corresponding phrase identifier at position $preorder_p(x)$ within L_{ids_p}, so we must support the efficient search of this position. The required functionality is easily achieved by regarding the vector of N

[5] The space requirement of the trie topology can be reduced to $2n + o(n)$ bits overall [2, 67]. However, $O(n)$ bits is sufficient for our purposes.

ids_p values, each of width t, as a bitmap of length tN. The dynamic data structure for bitmaps of Mäkinen and Navarro [44] would easily permit inserting, deleting, and accessing any identifier (i.e., t-bit chunk) in time $O(\log N)$ provided $t = O(\log u)$, which is the case. Its space overhead would be $O(N)$ bits.[6]

These identifiers will ultimately require $t = \lceil \log n \rceil$ bits, but we do not know n at this time. Therefore, we will use an amortized scheme as follows. All the identifiers ids_p of a block p will use the same t_p value. At step r of the parsing process, where r phrases have been identified, this value will be $t_p \leqslant \lceil \log r \rceil$. Every time an insertion arrives at block p with a value of r larger than 2^{t_p}, we will increase t_p to $\lceil \log r \rceil$, and make a pass over the whole list L_{ids_p} adding the new highest 0-bits to each number. This work amortizes over the whole construction process, as at most $n/2^t$ identifiers are modified t times.

Therefore, we need $N \log n + O(N)$ bits of space to maintain the identifiers, which adds up to $n \log n + O(n)$ bits overall.

Representation of the Inter-Block Pointers, ptr_p For the inter-block pointers, we use also a list L_{ptr_p}, managed in a similar way as for L_{ids_p} (this time the pointers always use $\lceil \log u \rceil$ bits). Since blocks have at least N_m nodes, we have a pointer out of (at least) $\Theta(\log^2 u)$ nodes, which adds $O(n/\log u)$ bits overall.

Construction Process The construction of *LZTrie* proceeds as explained in Section 3.2, using the symbols in the text to descend in the trie, until we cannot descend anymore. This indicates that we have found the longest prefix of the rest of the text that equals a phrase B_ℓ already in the LZ78 dictionary. Thus, we form a new phrase $B_t = B_\ell \cdot c$, where c is the next symbol in the text, and then insert a new leaf representing this phrase. However, this time the nodes are inserted in our hierarchical *LZTrie*, instead of a pointer-based trie.

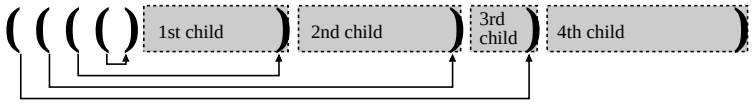
The insertion of a new node for the LZ78 phrase B_t in the trie implies to update only the block p in which the insertion is carried out. Assume that the new leaf must become the j-th node (in preorder) within the block p, and that the new leaf is a new child of node x in block p (i.e., node x represents phrase B_ℓ). We explain next how to carry out the insertion of the new leaf within the DFUDS of T_p.

We must insert a new '`(`' within the representation of x (which simulates the increase of the degree of node x, because of the insertion of the new child), and inserting also a new '`)`' to represent the new leaf we are inserting. Assume that the new leaf will become the new i-th child of node x. Therefore the new '`(`' must be inserted to the right of the opening parenthesis already at position $i' = x + degree(x) - i$ (recall from Section 2.5 how operation $child(x, i)$ uses the opening parentheses defining node x to descend to the i-th child). Then, the new '`)`' must be inserted at position $i'' = findclose(T_p, i' + 1)$, shifting to the right the last '`)`' in the subtree of the $(i-1)$-th child of x, which now becomes the new leaf. As a result, the two inserted parentheses form a matching pair, which can be handled in a straightforward way with the data structure of Chan et al. [11]. See Fig. 5 for an illustration.

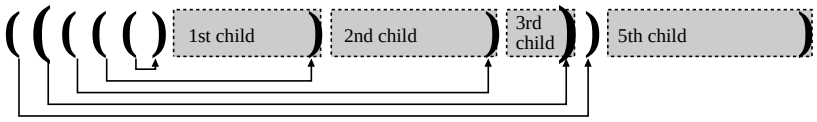
Then, we add a new flag `0` at position j in F_p. Also, c is inserted at the corresponding position within $letts_p$, and t is inserted at position j within the identifiers of block p (since these are stored in preorder). All this process takes $O(\log N)$ time.

Managing Block Overflows A *block overflow* occurs when, at construction time, the insertion of a new node must be carried out within a block p of N_M nodes. In such a case, we need to make room in p for the new node by selecting a subset of nodes to be copied to a new child block (of p) and then will be deleted from p. We explain this procedure in detail.

[6] To achieve this time and space, the balanced tree used by the structure must be modified to use leaves of $\Theta(\log N_M \log u)$ bits, instead of $\Theta(\log^2 N_M)$. For the purists: this may require using universal tables of size $O(u^\epsilon)$, for some constant $0 < \epsilon < 1$, but this is $o(n)$ if we choose $\epsilon < 1/2$, given that $n \geqslant \sqrt{u}$ by Property 3.



(a) A node x of degree 4 and its corresponding subtree in the DFUDS representation of *LZTrie*. Notice the relation among the four opening parentheses in the definition of x and the subtrees of the children of node x.



(b) Insertion of a new child of node x. The new leaf is inserted as the new fourth child of x, and thus it is represented by the new bold pair of matching parentheses. Notice how the degree of x is increased to 5 with the new opening parenthesis. The last closing parenthesis in the subtree of the third child of x is shifted to the right and now represents the new inserted leaf.

Fig. 5. Illustration of the insertion of a new leaf node in the DFUDS representation of *LZTrie*.

First we select a node z in p whose local subtree (along with z itself) will be copied to a new child block. In this way we ensure that a node and its children (and therefore all sibling nodes) are always stored in the same block (recall that a copy of z, as a leaf, will be kept in p).

Suppose that we have selected in this way the subtree of the j-th node (in preorder) in the block. Both the selected node z and its subtree are copied to a new block p', via insertions in $T_{p'}$. We must also copy to p' the flags $F_p[preorder_p(z)+1..preorder_p(z)+subtreesize_p(z)-1]$ (via insertions in $F_{p'}$) as well as the corresponding inter-block pointers within the subtree of the selected node z, which are stored in array ptr_p from position $rank_1(F_p, preorder_p(z))+1$ up to $rank_1(F_p, preorder_p(z)+subtreesize_p(z)-1)$.

Next we add in p a pointer to p'. The new pointer belongs to z, the j-th node in preorder in p (because we selected its subtree). We compute the position for the new pointer as $rank_1(F_p, j)$, adding the pointer at this position in L_{ptr_p}, and then we set to `1` the j-th flag in F_p, updating accordingly the $rank/select$ data structure for F_p (the portion copied to $F_{p'}$ must be deleted from F_p). Finally, we delete in p the subtree of z (via deletions in T_p), leaving z as a leaf in p.

Thus, the reinsertion process can be performed in time proportional to the size of the reinserted subtree (times $O(\log N_M)$), by using the insert and delete operations on the corresponding dynamic data structures that conform a block. However, we must be careful with the selection of node z. If, upon a block overflow, we traverse block p to select node z, we will take $O(N_M)$ time, which is too long. Instead, we will look for z in advance to overflows, by looking for possible candidates in the insertion path of new nodes.

To quickly select node z, we maintain in each block p a *candidate list* C_p [2], storing the local preorders of the nodes that can be copied to a new child block p' upon block overflow. With $selectnode$ we can obtain the candidate node corresponding to such a preorder. A subtree must have size at least N_m to be considered a candidate. Thus, after a number of insertions we will find that a node (within the insertion path) becomes a candidate. Let us think for a moment that we only maintain a candidate per block, and not a list of them. It can be the case that a few children of the block root have received (almost) all the insertions, so we have a few large subtrees within the block. When block p overflows, we reinsert the only candidate to a new child block, so we have no candidate anymore for p. We have to use the next insertions in order to find a new one. However, it can be also the case that different children of the root of p receive the new insertions, and

hence block p could overflow again within a few insertions, without finding a new subtree large enough so as to be considered a candidate (recall that we just use the insertion path to look for candidates). Thus, by maintaining a list of candidates in each block, instead of a unique candidate per block, we can keep track of all the nodes in p whose subtree is large enough, avoiding this problem.

Since the preorder of a node within a block p can change after the insertion of a new node in p, we must update C_p in order to reflect these changes. In particular, we must update the preorders stored in C_p for all candidate nodes whose preorder is greater than that of the new inserted node. To perform these updates efficiently, we represent C_p using a searchable partial sum data structure [44]. Thus, the original preorder $C_p[i]$ is obtained by performing $Sum(C_p, i)$ in $O(\log N)$ time. Let x be the new inserted node. Then, with $j = Search(C_p, preorder_p(x))$ we find the first candidate (in preorder) whose preorder must be updated, and we perform operation $Update(C_p, j, 1)$. In this way, we are increasing $C_p[j]$ by 1, automatically updating all the preorders in C_p that have changed after the insertion of x, in $O(\log N)$ time overall.

If we keep track of every candidate of size at least N_m, then every time p overflows there will be already candidate blocks. The reason is, again, that $N_M \geqslant 1 + \sigma N_m$, and thus that at least one of the children of the root must have size at least N_m. Since we use the descent process to look for candidates, we will find them as soon as their subtrees become large enough. In other words, the subtree of a node becomes larger as we descend through the node many times to insert new nodes, until eventually becoming a candidate.

We must also ensure that C_p requires little space (so we cannot have too many candidates). The size of the local subtree (i.e., only considering the descendant nodes stored in block p) of every candidate must be at least N_m. Also, we enforce that no candidate node descends from another candidate, in order to bound the number of candidates. To maintain C_p, every time we descend in the trie to insert a new LZ78 phrase, we maintain the last node z in the path such that $subtreesize_p(z) \geqslant N_m$. When we find the insertion point of the new node x, say at block p, before adding z to C_p we first perform $p_1 = Search(C_p, preorder_p(z))$, and then $p_2 = Search(C_p, preorder_p(z) + subtreesize_p(z))$. Then, z is added to C_p whenever: (1) z is not the root of block p, and (2) there is no other candidate in the subtree of z (that is, $p_1 = p_2$ holds).

If we find a candidate node z' which is an ancestor of the prospective candidate z, then after inserting z to C_p we delete z' from C_p. Thus, we keep the lowest possible candidates, avoiding that the subtree of a candidate becomes too large after inserting it in C_p, which would not guarantee a fair partition into two blocks of size between N_m and N_M. Because of Condition (2) above, there are one candidate out of (at least) N_m nodes; thus, the total space for C_p is $\frac{n}{N_m} \log N_M + O(n)$ bits, which is $o(n/\log u)$.

The reinsertion cost is in this way proportional to the size of p', since finding node z now takes $O(\log N_M)$ time (because of the partial-sum data structure used to represent C_p). Notice that the first time a node is reinserted, the reinsertion cost amortizes with the cost of the original insertion. Unfortunately, there are no bounds on the number of reinsertions for a given node. However, we shall show that multiple reinsertions of a node over time amortize with the insertion of other nodes. We use the following *accounting argument* [14] to prove the amortized cost of insertions. Let $\hat{c} = 2$ be the amortized cost of normal insertions (without overflows), being $c = 1$ the actual cost of an insertion. Therefore, every insertion spends one unit for the insertion itself, and reserves the remaining unit for future (more costly) operations. Let us think that we have separate reserves, one per block of the data structure. We shall prove that every time a block overflows, it has enough reserves so as to pay for the costly operation of reinserting a set of nodes.

In particular, every time a block overflows, its reserve is $N_M - I$, where I was the initial number of nodes for the block ($I = 0$ holds *only* for the root block). Let I' be the number of nodes of the new block p'. Then we must prove that $N_M - I \geqslant I'$ always holds, that is, $N_M \geqslant I + I'$. We need to prove:

Lemma 7. *For every candidate node z in block p, it holds that $subtreesize_p(z) < \sigma N_m$.*

Proof. By maintaining the lowest possible candidates, we find the smallest possible ones. If a node cannot be chosen as a candidate, this means that its subtree size is smaller than N_m nodes (another possibility is that there is another candidate within the subtree, yet this case is not interesting here). Therefore, the smallest subtree that can be chosen as a candidate may have up to $N_m - 1$ nodes in each children, and hence its total size is at most $1 + \sigma(N_m - 1) < \sigma N_m$. □

Because of this, blocks are created with $I', I < \sigma N_m$ nodes. As we have chosen $N_M \geqslant 2\sigma N_m$, it follows that $N_M \geqslant I + I'$. This means that every reinsertion of a node has already been paid for by some node at insertion time.[7] Thus, the insertion cost is $O(\log N_M)$ amortized. After n insertions, the overall cost amortizes to $O(n \log N_M)$.

Once we solved the overflow, the insertion of the new node is carried out either in p' or in p, depending on whether the insertion point lies within the moved subtree or not, respectively. Notice that there is room for the new node in either block.

Hierarchical *LZTrie* Construction Analysis As the trie has n nodes, we need $O(n) + (n + o(n)) + (n \log \sigma + O(n)) + (n \log n + O(n)) + O(n/\log u) + o(n/\log u) = n \log n + n \log \sigma + O(n)$ bits to represent the trie topology, flags, symbols, identifiers, inter-block pointers, and candidate lists, respectively.

When constructing *LZTrie*, the *navigational cost* per symbol of the text is $O(\log N_M) = O(\log \sigma + \log \log u)$, for a total worst-case time $O(u(\log \sigma + \log \log u))$. On the other hand, the amortized cost of updating blocks after an insertion is $O(\log N_M)$ per node, and therefore the total update cost adds up to $O(n(\log \sigma + \log \log u))$. Therefore, the total *LZTrie* construction time is $O(u(\log \sigma + \log \log u))$.

Representing the Final *LZTrie* Once we construct the hierarchical representation for *LZTrie*, we delete the text since it is not anymore necessary,[8] and then use the hierarchical *LZTrie* to build the final version of *LZTrie* in $O(n(\log \sigma + \log \log u))$ time. We allocate $n \log \sigma$ bits of space for the final array $letts$, $n \log n$ bits for array ids, and $O(n)$ for par. Then we perform a preorder traversal on the hierarchical tree, transcribing the nodes to a linear representation. Every time we copy a node, we check the corresponding flag, and then decide whether to descend to the corresponding child block or not.

Thus, the maximum amount of space used is $2n \log n + n \log \sigma + O(n)$ bits, since at some point we store both the hierarchical and final versions of ids (this is also true for $letts$, but we can first convert ids, then delete all the ids_p structures, and only then allocate $letts$, filling it in a second pass over the hierarchy). We then free the hierarchical *LZTrie*, and end up with a representation requiring $n \log n + n \log \sigma + O(n)$ bits. Thus, we have proved:

Lemma 8. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the LZTrie for T in $O(u(\log \sigma + \log \log u))$ time and using $2n \log n + n \log \sigma + O(n) = 2|LZ|(1 + o(1))$ bits of space.*

[7] More generally we could have set $N_M \geqslant (1 + \alpha)\sigma N_m$ for any constant $\alpha > 0$, and the analysis would have worked with $\hat{c} = 1 + 1/\alpha$.

[8] If allowed, we can even reuse the space occupied by the text as we parse it. Even if the text T is uncompressible (i.e., $|LZ| = |T|$), the extra space required to build *LZTrie* under this model would be $O(n) = O(|T|/\log u)$ bits.

4.2 Space-Efficient Construction of *RevTrie*

For the space-efficient construction of *RevTrie*, we use the technique of Section 4.1, to represent not the original reverse trie but its *Patricia tree* [49], which compresses *empty* unary paths, yielding an important saving of space. As we still maintain empty non-unary nodes, the number of nodes in *RevTrie* is $n' \leqslant 2n$.

Throughout the construction process we store in the nodes of the reverse trie "pointers" to *LZTrie* nodes, instead of the corresponding phrase identifiers $rids$ stored by the final *RevTrie*. Each such "pointer" is an offset into the *LZTrie* topology sequence of $2n$ bits (recall that *LZTrie* is already in final static form), and thus it uses $\log 2n$ bits. We store these pointers to *LZTrie* in the same way as for array ids_p in Section 4.1 (with fixed width t), in preorder according to *RevTrie* and spending $O(n)$ extra bits for the list functionality. The aim is to obtain the text of the phrase represented by a *RevTrie* node, since we are compressing empty-unary paths and the string represented by a node is not available otherwise (unlike what happens with the traditional Patricia trees). This connection is given by $Node$ in the final LZ-index. However, at construction time we avoid accessing $Node$ when building the reverse trie, so we can build $Node$ after both tries have been built, thus reducing the peak indexing space.

Empty non-unary nodes are marked by storing in each block p a bit vector B_p (represented in the same way as F_p, with a dynamic data structure supporting $rank$ and $select$ queries). We store pointers to *LZTrie* nodes only for non-empty *RevTrie* nodes, so we store n of them. This shall reduce the indexing space of the preliminary definition of the algorithm [4], which shall be useful later when constructing reduced-space versions of the LZ-index.

As we compress empty-unary paths, the trie edges are labeled with strings instead of single symbols. The Patricia tree stores only the first symbol of the edge labels. We do the same in our reverse trie, using the same partial sum approach as for *LZTrie*, on array $rletts$. However, we do not store the Patricia-tree skips, as their space consumption is problematic. Instead, we use the following procedure to find out in $O(\ell)$ time the skip value ℓ of the edge leading to a node y from its parent x [50]. Let X and Y be the strings labeling the paths from the root of the reverse trie to nodes x and y, respectively, then $\ell = |Y| - |X|$. We find the leftmost and rightmost leaves v_r^1 and v_r^2 descending from y, and map them to *LZTrie* nodes v_{lz}^1 and v_{lz}^2 using the reverse trie pointers. Since v_r^1 and v_r^2 are labeled by strings that start with Y and differ in the next character, v_{lz}^1 and v_{lz}^2 are labeled by strings ending at Y^r, and that differ in the previous character. Therefore, we carry out $parent$ in *LZTrie* consecutively $|X|$ times, starting from v_{lz}^1 and from v_{lz}^2, and then continue moving to the parents in synchronization until the characters leading to both nodes differ. The total number of $parent$ operations executed is $2|Y|$, from what we can infer ℓ. The first $|X|$ $parent$ operations can be executed with a single operation called *level-ancestor*, which can be executed in constant time using $o(n)$ extra bits on top of the *LZTrie* topology representation [34]. Thus the overall time is $O(|Y| - |X|) = O(\ell)$. Since the total amount of skips traversed along the construction process is u, computing the skips in this way adds $O(u(1 + \frac{\log \sigma}{\log \log u}))$ to the overall time (the $1 + \frac{\log \sigma}{\log \log u}$) factor is due to our representation of $letts$).

Note, additionally, that with this process we do compare all the characters of a string as we descend in the reverse trie, so we do not need to carry out the final Patricia tree check that is necessary in the classical implementation.

Construction Process To construct the reverse trie we traverse the final *LZTrie* in depth-first order, generating each LZ78 phrase B_i stored in *LZTrie*, and then inserting its reverse B_i^r into the reverse trie.

Note that our proposed scheme to compute skips can be simplified when the node y corresponds to a phrase. In this case it is sufficient to map y itself to *LZTrie*, as the depth of the mapped node will be $|Y|$.

For the case of empty nodes, we note that the general scheme we described above works equally well if we choose any descendant of the first and second children of y, as they will also differ at the next character. Such children will exist because empty nodes cannot be unary. Therefore, it is sufficient to obtain any non-empty descendant of a node v_r, where v_r is the first or second child of y. For example, the *LZTrie* pointer corresponding to the first non-empty descendant v'_r of v_r can be found at position $rank_1(B_p, preorder_p(v_r)) + 1$ within the pointer array.

However, there exists an additional problem: the *local* subtree of node v_r can be exclusively formed by empty nodes, in which case finding the non-empty node v'_r is not as straightforward as explained, since v'_r is stored in a descendant block. This problem comes from the fact that, upon a block overflow in the past, we might have chosen empty nodes z descending from v_r, whose subtrees were reinserted into new blocks.

To solve this problem, we store in every block p a pointer to *LZTrie*, which is representative for the nodes stored in the block p. If a block is created from a non-empty node, then we can store the pointer of that node. In case of creating a new block p' from an empty node, if the new block p' is going to be a leaf in the tree of blocks, then it will contain at least a non-empty node. Thus, we associate with p' the pointer to *LZTrie* of this non-empty node. If, otherwise, p' is created as an internal node in the tree of blocks, then it can be the case that all of the nodes in p' are empty. In this case, we choose any of the descendants blocks of p' and copy its pointer to p'. This pointer has been "inherited" (in one or several steps) from a leaf block, thus this corresponds to a non-empty *RevTrie* node. Thus, in case that the local subtree of v_r is formed only by empty nodes, we take one of the blocks descending from v_r (say the first in preorder) and use the *LZTrie* pointer associated to that block.

An important difference with the *LZTrie* construction is that in *RevTrie* we do not only insert new leaves: there are cases where we insert a new non-empty *unary* internal node (corresponding to the phrase we are inserting in *RevTrie*). A unary node is represented as '`()`' in DFUDS, which is a matching pair and hence the insertion can be handled by the data structure of Chan et al. [11]. If we insert the new node as the parent of an existing node x, then the insertion point is just before the representation of x in the DFUDS sequence.

Hierarchical *RevTrie* Construction Analysis The hierarchical representation of the reverse trie requires $O(n')+(n'+o(n'))+(n'+o(n'))+(n\log 2n+O(n))+(n'\log\sigma+O(n'))+O(n'/\log u)+o(n'/\log u) \leqslant n\log n+2n\log\sigma+O(n)$ bits of storage to represent the trie topology, flags, bit vector of empty nodes, pointers to *LZTrie* stored in the nodes, symbols, pointers (both inter-block and extra *LZTrie* pointers associated to each block), and candidates, respectively.

For each reverse phrase B_i^r to be inserted in the reverse trie, $1 \leqslant i \leqslant n$, the navigational cost is $O(|B_i^r|\log N_M)$ (this subsumes the $O(|B_i^r|)$ time needed to compute the skips). Since $\sum_{i=1}^{n}|B_i^r| = u$, the total navigational cost to construct the hierarchical *RevTrie* is $O(u\log N_M)$. Since the number of node insertions is $n' = O(n)$, the total cost is $O(u(\log\sigma + \log\log u))$, just as for *LZTrie*.

Constructing the Final *RevTrie* After we construct the hierarchical reverse trie, we construct *RevTrie* directly from it in $O(n'\log N_M)$ time, replacing the pointers to *LZTrie* by the corresponding phrase identifiers ($rids$). Since we have to preallocate $rids[1..n]$, the space is raised to $3n\log n + 3n\log\sigma + O(n)$ bits. We avoid a similar blowup for $rletts$ by deleting all the $rletts_p$ structures once the hierarchical *RevTrie* is built, and only then allocating the static $rletts$. It is still possible to find each letter value along a preorder traversal of *RevTrie* by mapping to *LZTrie* as done for computing the skips. This must be done before the pointers to *LZTrie* are converted into $rids$. Finally, we free the hierarchical trie, dropping the space to $2n\log n + 3n\log\sigma + O(n)$ bits.

Lemma 9. *Given the LZTrie of n nodes for a text $T[1..u]$ over an alphabet of size σ, there exists an algorithm to construct the corresponding RevTrie in $O(u(\log \sigma + \log\log u))$ worst-case time and using a total space of $2n \log n + 2n \log \sigma + O(n)$ bits on top of the space required by the final LZTrie.*

4.3 Space-Efficient Construction of $Range$

To construct the *Range* data structure, recall that for every LZ78 phrase B_t of T we must store the point $(preorder_r(v_r), preorder_{lz}(v_{lz}))$, where v_r is the *RevTrie* node corresponding to B_t^r, and v_{lz} is the *LZTrie* node corresponding to phrase B_{t+1}. We allocate memory space for a temporary array $RQ[1..n]$ of $n \log n$ bits, storing the points to be represented by *Range*. Array RQ is initially sorted by the first coordinates of the points. Notice that since there is a point for every first coordinate $1 \leqslant i \leqslant n$, the first coordinate of every point is represented simply by the index of array RQ, thus saving space. In other words, $RQ[i] = j$ represents the point (i, j). Notice also that RQ is a permutation of $\{1, \ldots, n\}$. (In fact, the $preorder_r$ values that participate in $Range$ are $\{0, \ldots, n-1\}$, so we must shift them by one.)

To generate the points, we first notice that for a *RevTrie* preorder $i = 0, \ldots, n-1$ (corresponding only to non-empty nodes) representing the reverse phrase B_t^r, we can obtain the corresponding phrase identifier $t = rids[i+1]$, and then with the inverse permutation $ids^{-1}[t+1]$ we obtain the *LZTrie* preorder for the node corresponding to phrase B_{t+1}. Thus, we define $RQ[i+1] = ids^{-1}[rids[i+1]+1]$.

Therefore, we start by computing ids^{-1} on the same space of ids, using the algorithm of Lemma 3, requiring $O(n)$ time and n extra bits of space. Then, we allocate $n \log n$ bits for array RQ, and traverse *RevTrie* in preorder. For every non-empty node with preorder i we set RQ as defined above. The total space is thus raised to $3n \log n + 3n \log \sigma + O(n)$ bits. Next, we recover ids from ids^{-1}, using again Lemma 3.

After building RQ, to construct $Range$ we must sort the points in RQ by the second coordinate (recall Section 3.2), which in our space-efficient representation of the points means using the second coordinates as array indexes, and storing the first coordinates as array values [9]. This means sorting the current values stored in array RQ. However, since these values along with the corresponding array indexes represent points, after sorting the points we must recall the original array index for every value, so as to store that value in the array. This is straightforward if we store both coordinates of the points, requiring $2n \log n$ bits of space. However, we are trying to reduce the indexing space, and therefore use an alternative approach.

Notice that since $RQ[i] = j$ represents the point (i, j), $RQ^{-1}[j] = i$ shall also represent the point (i, j), yet the points in the inverse permutation RQ^{-1} are sorted by their second coordinate (i.e., in RQ^{-1} the second coordinates are used as array indexes). Thus, we use the algorithm of Lemma 3 to construct RQ^{-1} on top of the space for RQ, in $O(n)$ time and requiring n extra bits of space. Now, we can finally build $Range$ from RQ^{-1} just as explained in Section 3.2.

However, to save space, we will not allocate space for the $\log n$ bit vectors of n bits in advance. Rather, we will allocate the n bits for the top-level bitmap, fill it, and then *compact* array RQ^{-1} so that the most significant bits of all the elements are dropped. This can be done in-place and will save n bits. Only then we will allocate the n bits of the second-level bitmap, fill it, then compact RQ^{-1} once again, and so on. Notice that for this to work we must decide whether a value goes left or right in the $Range$ structure by considering its highest bit and not whether its value belongs to the left or right half of the interval. This may at worst

[9] We could choose to define RQ in a different way, storing the first coordinate of the points and using the second coordinate as array index. However, by using our approach we can construct array RQ with a sequential scan over arrays $rids$ and RQ itself. The importance of this fact shall be made clear later in this section.

yield a $Range$ structure that has one more level than the original one, thus wasting $O(n)$ bits. In exchange, we build $Range$ from RQ^{-1} using only $O(n)$ extra space.[10]

Lemma 10. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then, given the LZTrie and RevTrie data structures for T, there exists an algorithm to construct the $Range$ data structure requiring a maximum space of $n \log n + O(n)$ further bits, and takes $O(n \log n) = O(u \log \sigma)$ time.*

4.4 Construction of the $Node$ Mapping and Remaining Data Structures

Finally, we proceed to construct the $Node$ mapping as follows: we traverse *LZTrie* in preorder, and for every node x with LZ78 identifier i, we store in $Node[i]$ the node position within the corresponding parentheses sequence. This increases the total space requirement to $4n \log n + 3n \log \sigma + O(n)$ bits, which is the final space required by the LZ-index. The process can be carried out in $O(n) = O(u/\log_\sigma u)$ time.

As we said in Section 3.2, in a practical implementation the *Range* data structure is replaced by the $RNode$ mapping [56]. This is built from $rids$ in the same way as $Node$ is built from ids. The process explained in Section 4.3 is not carried out in such a case.

The original LZ-index is able to report the pattern occurrences in the format $[\![t, offset]\!]$, where t is the phrase number where the occurrence starts, and $offset$ is the distance between the beginning of the occurrence and the end of the phrase. To map these occurrences into text positions, Arroyuelo et al. [7] add a bit vector $TPos$ marking the phrase beginnings, which is then represented with a data structure for $rank$ and $select$ and requiring $n \log \frac{u}{n} + O(n) + o(u) = o(u \log \sigma)$ bits of space [62]. A more practical approach [5] consists in sampling the starting positions of some phrases, and then representing the starting position of every other phrase as an offset from the previous sampled phrase (thus saving space). With high probability, the space requirement of this alternative approach is $n + O(n \log \log u) = o(u \log \sigma)$ bits if sample rates are properly chosen. Both data structures can be constructed without requiring any extra space, and thus to simplify we omit them in this paper.

4.5 The Whole Compressed Indexing Process

The whole compressed construction of the LZ-index is summarized in the following steps:

1. We build the hierarchical *LZTrie* from the text. We can then erase the text.
2. We build *LZTrie* from its hierarchical representation. We then free the hierarchical *LZTrie*.
3. We build the hierarchical representation of the reverse trie from *LZTrie*.
4. We build *RevTrie* from its hierarchical representation, and then free the hierarchical *RevTrie*.
5. We build $Range$.
6. We build $Node$ from ids.

In Table 2 we show the total space and time requeriment at each step.

[10] Another slight complication is that the recursive procedure cannot be used, but we must proceed levelwise. This is not really problematic because the tree is perfectly balanced.

Table 2. Space and time requirements of each step in the whole compressed indexing process.

Indexing step	Maximum total space	Space after step	Indexing time
1	$n \log n + n \log \sigma + O(n)$	$n \log n + n \log \sigma + O(n)$	$O(u(\log \sigma + \log \log u))$
2	$2n \log n + n \log \sigma + O(n)$	$n \log n + n \log \sigma + O(n)$	$O(u(\log \sigma + \log \log u))$
3	$2n \log n + 3n \log \sigma + O(n)$	$2n \log n + 3n \log \sigma + O(n)$	$O(u(\log \sigma + \log \log u))$
4	$3n \log n + 3n \log \sigma + O(n)$	$2n \log n + 3n \log \sigma + O(n)$	$O(u(\log \sigma + \log \log u))$
5	$3n \log n + 3n \log \sigma + O(n)$	$3n \log n + 3n \log \sigma + O(n)$	$O(u \log \sigma)$
6	$4n \log n + 3n \log \sigma + O(n)$	$4n \log n + 3n \log \sigma + O(n)$	$O(u/\log_\sigma u)$

4.6 Managing Dynamic Memory

The model of memory allocation is a fundamental issue of succinct dynamic data structures, since we must be able to manage the dynamic memory fast and without requiring much extra memory space due to memory fragmentation [63]. We assume a standard model where the memory is regarded as an array, with words numbered 0 up to $2^w - 1$. The space usage of an algorithm at a given time is the highest memory word currently in use by the algorithm. This corresponds to the so-called $\mathcal{M}_B$ memory model [63], which is the standard on the RAM model and assumes the least from the system: in model $\mathcal{M}_B$ there are no system calls for allocation and deallocation of memory, but the program must handle memory by itself. We set $w = \Theta(\log u)$, as we need $\Theta(n \log n)$ bits of space to build our index but we do not know n in advance.

We manage the memory of every trie block separately, each in a "contiguous" memory space. However, trie blocks are dynamic as we insert new nodes, hence the memory space for trie blocks must grow accordingly. If we use an *Extendible Array* (EA) [9] to manage the memory of a given block, we end up with a collection of at most $O(n/N_m) = O(n/\log^2 u)$ EAs, which must be maintained under the operations: create, which creates a new empty EA in the collection; destroy, which destroys an EA from the collection; grow(A), which increases the size of array A by one cell; shrink(A), which shrinks the size of array A by one cell; and access(A, i), which access the i-th item in array A.

Raman and Rao [63] show how operation access can be supported in $O(1)$ worst-case time, create, grow and shrink in $O(1)$ amortized time, and destroy in $O(s'/w)$ time, where s' is the nominal size (in bits) of array A to be destroyed. The whole space requirement is $s + O(a^* w + \sqrt{sa^* w})$ bits, where a^* is the maximum number of EAs that ever existed simultaneously, and s is the nominal size of the collection.

To simplify the analysis we store every component of a block in different EA collections (i.e., we have a collection for T_ps, a collection for $letts_p$s, and so on). The memory for $letts_p$, F_p, C_p, T_p, L_{ids_p}, etc. inside the corresponding EAs is managed as in the original work [44].

Thus, we use operation grow on the corresponding EAs every time we insert a node in the tree, and operation create to create a new block upon block overflows, both in $O(1)$ amortized time. Operation shrink, on the other hand, is used by our representation after we reinsert the subtree upon a block overflow, in $O(1)$ amortized time. Finally, operation destroy over the blocks is used when destroying the whole hierarchical trie. As the cost to build the trie is $O(\log N_M)$ per element inserted, which adds $\Theta(\log u)$ bits to the data structure, the cost per bit inserted is $O(\frac{\log \sigma + \log \log u}{\log u})$. The cost for destroy is just $O(1/w) = O(\frac{1}{\log u})$ per bit, which is subsumed by the earlier construction cost.

Let us analyze the space overhead due to EAs for the case of T_p. Since we only insert nodes into our tries, we have that the maximum number of blocks that we ever have is $a^* = O(n/N_m)$. As the nominal size of the EA collection for T_p is $O(n)$ bits, the EA requires $O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w}{N_m}}) = O(n)$ bits of space [63].

A similar analysis can be done for the collections supporting F_p and C_p. The nominal size of the collection for $letts_p$ is $n \log \sigma + O(n)$, and thus we have $n \log \sigma + O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w \log \sigma}{N_m}}) = n \log \sigma + O(n)$ bits overall. For the collection supporting ids_p we obtain $n \log n + O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w \log u}{N_m}}) = n \log n + O(n)$ bits of space. In general, the whole space overhead due to memory management is $O(n)$ bits.

To complete the definition of our memory allocation model, it remains to say that we can store the EAs representing the block components within a unique global EA. In this case, the number of EAs in the collection is $a^* = O(1)$, since we have a constant number of block components. The nominal size of the whole collection is $s = n \log n + n \log \sigma + O(n)$ bits (where the $O(n)$ term includes the space for the EA memory management of these collections). Hence, the space overhead of this global EA is $O(w + \sqrt{wn \log u}) = o(n)$ bits.

Now that we have defined our memory allocation model, we can conclude:

Theorem 1. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the LZ-index for T using $4n \log n + 3n \log \sigma + O(n)$ bits of space and $O(u(\log \sigma + \log \log u))$ time. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

Note that this construction space may differ from the final LZ-index space only in the $O(n)$ extra-bit space, which is $O(|T|/\log u)$. The total space can also be written as $4uH_k(T) + o(n \log \sigma)$ for any $k = o(\log_\sigma u)$.

4.7 Constructing the LZ-index in Reduced-Memory Scenarios

We assume next a model where we have restrictions in the amount of main memory available, such that we cannot maintain the whole index in main memory. So, we aim at reducing as much as possible the main memory usage of our algorithms. We shall prove that the LZ-index can be constructed as long as the available memory is $n \log n + 3n \log \sigma + O(n)$ bits (i.e., essentially, the compressed text can be stored in main memory). This has applications, for instance, in text search engines, where we can use a less powerful computer to carry out the indexing process, devoting a more powerful one to answer user queries.

Since we have assumed that we have enough secondary storage space so as to store the final index (see Section 2.1), we will use that space to temporarily store on disk certain LZ-index components which will not be needed in the next indexing step, and then possibly loading them back to main memory when needed. However, and as we have seen throughout Section 4, our indexing algorithm is independent of this fact, and we can choose not to use the disk at all when enough main memory is available.

In the following, we show how to adapt our original algorithm to this scenario. At every step we will analyze the maximum and final amount of main memory required at that step. The total amount of memory (main plus secondary) and time complexities will be omitted as they are always as in Section 4, that is, as if we did not use the disk along the construction process. We will only mention them in special cases. Instead, we consider the amount of I/O carried out, in bits.

Step (1) We build the hierarchical *LZTrie* from the text. We can then erase the text. The maximum and final main-memory space is $n \log n + n \log \sigma + O(n)$ bits.

Step (2) We build *LZTrie* from its hierarchical representation. To construct the final ids array while trying to reduce the maximum main-memory space, we do not allocate space for it at once. Since this array is stored in preorder, and since we perform a preorder traversal on the trie, the values in array ids are produced by a linear scan. Thus, we only allocate main-memory space for a constant number of components of the array (e.g., a constant number of disk pages), which are stored on disk upon filling them. This process performs $n \log n + O(n)$ bits of I/O, and at the end we free all the hierarchical ids_p components.

Then the symbols ($letts$) and the trie topology (par) are converted into static form in memory, and their hierarchical versions are freed. The static versions are maintained in main memory for the next step, requiring only $n \log \sigma + O(n)$ bits. The maximum main-memory space used along this step is $n \log n + n \log \sigma + O(n)$ bits.

Step (3) We build the hierarchical representation of the reverse trie from *LZTrie*. Recall that every non-empty *RevTrie* node stores a pointer to the corresponding *LZTrie* node. These pointers, par and $letts$ are necessary to obtain the skips for navigating *RevTrie*. The maximum and final main-memory usage is $n \log n + 3n \log \sigma + O(n)$ bits (recall that array ids is on disk).

Step (4) We build *RevTrie* from its hierarchical representation as follows. We first erase the hierarchical $rletts_p$ components and recompute them using level-ancestor queries on *LZTrie*, as in Section 4.2. In this way the static array $rletts$ is generated in preorder directly on disk. After this the already static array $letts$ is also moved to disk (that is, progressively written as it is deleted from main memory).

Now we generate $rids$. We store the pointers to *LZTrie* associated with *RevTrie* nodes in a linear array, in the same way as done in Step (2) for array ids in *LZTrie*. In this way we do not need extra main-memory space on top of the hierarchical *RevTrie*. After storing the pointers on disk, the total space is raised to $3n \log n + 3n \log \sigma + O(n)$ bits, since we have at the same time the final *LZTrie* (array ids is on disk), the hierarchical *RevTrie* pointers (in main memory), and the static *RevTrie* pointers (on disk). Now we free the hierarchical *RevTrie* pointers, thus reducing the main-memory space to $O(n)$ bits.

Then, we proceed to replace the pointers by the corresponding phrase identifiers ($rids$). We first load array ids into main memory (leaving a copy of it on disk, for further use). Now we perform a sequential scan on the array of pointers, bringing to main memory just a constant number of disk pages, then following these pointers to *LZTrie* to get the phrase identifier stored in ids (note this means that the accesses to ids are at random, hence we need ids in main memory) and storing these identifiers in the same space of the pointers, writing them to disk and loading the next portion of the pointer array. We leave the copy of array ids in main memory (this shall be useful for the next step).

The maximum main-memory space needed along this step is $n \log n + 3n \log \sigma + O(n)$ bits, and we finish with $n \log n + O(n)$ bits in use. The amount of I/O performed is $4n \log n + 3n \log \sigma + O(n)$ bits.

Step (5) We build $Range$ as in Section 4.3, yet with some care for the peak of main memory usage. We compute ids^{-1} on the same space required by ids, using the algorithm of Lemma 3. Then, we traverse $rids$ in preorder, creating array $RQ[i+1] \leftarrow ids^{-1}[rids[i+1]+1]$. Notice that both arrays $rids$ and RQ are accessed sequentially, which means that we can maintain just a constant number of components of these arrays in main memory. Array ids^{-1}, on the other hand, is accessed randomly, so we maintain it in main memory. In this way, the maximum main-memory space needed along this process is $n \log n + O(n)$ bits.

When this process finishes, the total space is raised to $4n \log n + 3n \log \sigma + O(n)$ bits, and then we free array ids^{-1} (recall that we still have a copy of the original array ids on disk), dropping the main-memory space to $O(n)$ bits, since we maintain just the trie topologies of *LZTrie* and *RevTrie*.

After building RQ on disk, we move it to main memory and construct $Range$ within $O(n)$ extra bits of space using the algorithm of Section 4.3. Then $Range$ is moved to disk. Thus, the maximum main-memory space requirement to construct $Range$ is $n \log n + O(n)$ bits. At the end we have only $O(n)$ bits of main memory in use. The amount of I/O is $4n \log n + O(n)$ bits.

Step (6) We build $Node$ from ids, by traversing *LZTrie* in preorder. In this way, array ids is sequentially traversed, while $Node$ is randomly accessed. Thus, we allocate $n \log 2n$ bits of space for $Node$, and maintain it in main memory. Array ids, on the other hand, is brought by parts to main memory, according to a sequential scan. Finally, we save $Node$ to disk. The amount of I/O is $2n \log n + O(n)$ bits. The amount of main memory used is $n \log n + O(n)$ bits. We use the same procedure in case of using the *RNode* data structure instead of $Range$. At the end we move to disk both trie topologies.

The overall amount of I/O is $11n \log n + 3n \log \sigma + O(n)$ bits. Thus, we have proved:

Theorem 2. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there is an algorithm to build the LZ-index of T using a maximum main-memory space of $n \log n + 3n \log \sigma + O(n)$ bits and $O(u(\log \sigma + \log \log u))$ time. The algorithm requires $7n \log n + O(n)$ bits of I/O, plus those needed to write the final index. The total space used by the algorithm is $4n \log n + 3n \log \sigma + O(n)$ bits. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

Note that the total I/O is less than 3 times the one required if we can build the whole index in main memory and then store the final result on disk.

5 Space-Efficient Construction of Reduced-Space LZ-indexes

There exist new reduced versions of the LZ-index, some of which are able to replace the original LZ-index in many practical scenarios [5]. Henceforth, in this section we show how to adapt our space-efficient algorithm to build these new indexes. The result is, again, that we can build the indexes within the space of the final index except for a lower-order term of $O(n)$ bits, and that we can build them using just the main memory required for storing the LZ78-compressed text, plus $O(n \log \sigma)$ bits. In the latter case, an amount of I/O is required that varies depending on the variant we build.

Throughout this section we assume the reduced-memory scenario as in Section 4.7. We will present the space usage of our algorithms in two ways: the total maximum main-memory space and the maximum total space (main-memory plus secondary-memory space) at every step. The latter is also the maximum space usage of the algorithm if we build it entirely in main memory.

5.1 Space-Efficient Construction of Scheme 2

We perform the following steps to build Scheme 2 of the LZ-index (recall its definition in Section 3.4).

1. We build the hierarchical *LZTrie* from the text. This takes $O(u(\log \sigma + \log \log u))$ time, and the maximum space requirement is $n \log n + n \log \sigma + O(n)$ bits.
2. We derive the final *LZTrie* from the hierarchical one, which is then freed. The conversion takes $O(u(\log \sigma + \log \log u))$ time because of the traversals on the hierarchical *LZTrie*. It creates the static trie topology par, the symbols $letts$, and the phrase identifiers ids, and requires $n \log n$ extra bits. We use the approach of Section 4.7 to construct ids on disk, without requiring extra main-memory space. Thus the

total space usage is again $2n \log n + n \log \sigma + O(n)$ bits, while the maximum main-memory usage is $n \log n + n \log \sigma + O(n)$ bits. Arrays par and $letts$ are kept in main memory for the next steps, so the main-memory space after this step is $n \log \sigma + O(n)$ bits. The resulting amount of I/O is $n \log n + O(n)$ bits, for the construction of array ids.

3. We build the hierarchical *RevTrie* from *LZTrie*, as in Section 4.2. This takes $O(u(\log \sigma + \log \log u))$ time. The total space usage is raised to $2n \log n + 3n \log \sigma + O(n)$ bits. The maximum main-memory space is $n \log n + 3n \log \sigma + O(n)$ bits.
4. We build the final *RevTrie* from the hierarchical one, storing the trie topology $rpar$, the symbols $rletts$, and bit vector B marking the empty nodes. As before, we can erase the $rletts_p$ structures and re-create the static array $rletts$ directly on disk, from the topology $rpar$ and the static $LZTrie$, so that no extra space is required. Array R is now built from the pointers to *LZTrie*, by replacing the pointers with the corresponding *LZTrie* preorder (recall that we apply $rank$ on par to get the *LZTrie* preorder of a node). We construct R by using the same approach as for array ids in Step (2), performing $n \log n + O(n)$ bits of extra I/Os. The total time is $O(u(\log \sigma + \log \log u))$. We then free the space of the hierarchical *RevTrie* pointers. The maximum total space is $3n \log n + 3n \log \sigma + O(n)$ bits, while the maximum main-memory space is $n \log n + 3n \log \sigma + O(n)$ bits. At this point we can move $letts$ and both tries topologies definitely to disk, and leave the main memory empty.
5. To space-efficiently construct array $rids^{-1}$, we first construct $rids$ in the following way: we start by moving array ids to main memory. Then we compute $rids[j] \leftarrow ids[R[j]]$ for increasing values of j. As arrays $rids$ and R are traversed sequentially, we can store/load them to/from disk by parts (respectively), without requiring extra main-memory space. After we build $rids$, the total space has raised to $3n \log n + 3n \log \sigma + O(n)$ bits. We then move array ids back to disk. Finally, we load $rids$ to main memory, and use the procedure of Lemma 3 to construct $rids^{-1}$ on top of $rids$, to finally store $rids^{-1}$ to disk. The overall time is $O(n)$. The maximum total space is $3n \log n + 3n \log \sigma + O(n)$ bits, while the maximum main-memory space is $n \log n + 3n \log \sigma + O(n)$ bits. The total number of disk I/O performed by this process is $6n \log n + O(n)$ bits.

This is a practical version of the LZ-index, and thus we do not store $Range$. Thus, we conclude:

Theorem 3. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the Scheme 2 of the LZ-index for T using a total space of $3n \log n + 3n \log \sigma + O(n)$ bits and $O(u(\log \sigma + \log \log u))$ time. The maximum main-memory space used at any time to construct Scheme 2 can be reduced to $n \log n + 3n \log \sigma + O(n)$ bits, in such a case performing $5n \log n + O(n)$ bits of I/O, plus those needed to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

5.2 Space-Efficient Construction of Scheme 3

To build Scheme 3 of the LZ-index, we first build *LZTrie* in $O(u(\log \sigma + \log \log u))$ time, storing par, $letts$, and ids, the latter directly on disk using the procedure of Section 4.7, Step (2). This requires a maximum of $2n \log n + n \log \sigma + O(n)$ bits of total space, $n \log n + n \log \sigma + O(n)$ bits of main memory, and $n \log n + O(n)$ bits of I/O. It ends up using $n \log \sigma + O(n)$ bits in main memory.

We then construct the hierarchical *RevTrie*. The space requirement raises to $2n \log n + 3n \log \sigma + O(n)$ bits. We build the final *RevTrie* storing just $rpar$ and $rletts$ in main memory, and discard the pointers to

LZTrie, temporarily losing the connectivity between tries. We use the method of Section 4.7 to generate $rletts$, i.e., we erase the hierarchical $rletts_p$ arrays and then re-create the static $rletts$ from the static *LZTrie* par and $letts$ components. This time, before discarding the pointers, we will create explicitly the static $skips[1..n']$ array, so that $skips[i]$ is the skip by which one arrives at the i-th node of *RevTrie* in preorder.

Array $skips$ is created together with $rletts$ and in similar fashion, by traversing *RevTrie* and using the information of *LZTrie*. The total time is $O(u(1+\frac{\log\sigma}{\log\log u}))$ because, as explained in Section 4.2, all the skips add up at most to u. We reduce the number of skips stored to at most $n'/2 \leqslant n$, by not storing the skips of the leaf nodes. As we see soon, these will not be necessary. The topology representation $rpar$ allows one to count the number of leaves to the left of a node [8], so that we can index into the reduced array $skips$.

Note that each skip may be as large as u. However, as they are at most n and add up to at most u, we can set up a bitmap $S[1..u]$ where we write each skip as $skip[i]-1$ 0s followed by a 1. Hence later we can recover $skip[i] = select_1(S,i) - select_1(S,i-1)$. By choosing a suitable static bitmap encoding method for S [60], the structure requires at most $n\log\frac{u}{n} + O(n)$ bits, and answers $select$ queries in constant time[11].

The peak of memory usage right after freeing the pointers is thus $n\log n + n\log u + 3n\log\sigma + O(n)$, of which all but the first $n\log n$ term is in main memory. After freeing the pointers, the main memory space becomes $n\log\frac{u}{n} + 3n\log\sigma + O(n)$ bits.

Next we allocate main memory space for array $rids$, requiring $n\log n + O(n)$ extra bits. We traverse *LZTrie* in preorder, and generate every phrase B_i stored in it (where i is the preorder of the *LZTrie* node). We then look for B_i^r in *RevTrie*. Recall that at this point we do not have the connectivity between tries, which is generally used to search in *RevTrie*, but we have $rletts$ and $skips$. Moreover, since string B_i^r exists for sure in *RevTrie* (because it exists as an LZ78 phrase in *LZTrie*), we only need to descend in *RevTrie* without the Patricia-tree verifications, up to consuming B_i^r. For this reason we do not need the skips at the leaves either: when we arrive at a leaf we must have consumed B_i^r. When we arrive at the (leaf or internal) node for B_i^r, which has preorder j in *RevTrie*, we set $rids[j] \leftarrow ids[i]$. Notice the sequential scan on ids, which is brought to main memory by parts. The overall work on *LZTrie* is $O(n\log\sigma)$, since each string is generated in $O(\log\sigma)$ time (because of the data structure used to represent $letts$). For *RevTrie*, on the other hand, we have that $\sum_{i=1}^{n}|B_i^r| \leqslant u$, and thus the overall time is $O(u\log\sigma)$.

Finally we move $rletts$ and trie topologies to disk. The skips can be erased or moved to disk, as desired for the final representation. Note that array $rids$ is still in main memory. Before moving it to disk, we create $rids^{-1}$ within $\epsilon n\log n + O(n)$ extra main-memory bits, and then move both $rids$ and $rids^{-1}$ to disk. Finally, we move ids to main memory, create ids^{-1} in the same way, and move it back to disk, writing ids^{-1} as well. The whole process of creating $rids$, $rids^{-1}$ and ids^{-1}, requires $(4+2\epsilon)n\log n + O(n)$ extra I/O bits.

For creating ids^{-1} (the process for $rids^{-1}$ is identical) we build on ids the data structure of Munro et al. [52] (Section 2.4), as follows. Let $A_{ids}[1..n]$ be an auxiliary bitmap, and $B_{ids}[1..n]$ be a bitmap marking which elements of ids have an associated backward pointer. Both bitmaps are initialized to all zeros.

We start from the first position of ids, and follow the cycles of the permutation. We mark every visited position i of the permutation as $A_{ids}[i] \leftarrow \mathtt{1}$. We also mark one out of $1/\epsilon$ elements when following the cycles, by setting to $\mathtt{1}$ the appropriate position in B_{ids}. We stop following the current cycle upon arriving to a position j such that $A_{ids}[j] = \mathtt{1}$; then, we move sequentially from position j to the next position j' such that $A_{ids}[j'] = \mathtt{0}$, and repeat the previous process.

[11] Although Okanohara and Sadakane report a non-constant time in their article [60], this is easily converted into constant by using a constant-time $select$ implementation for their internal dense array of $O(n)$ bits. Note also that this space is preferable to the $O(n\log\log u) + o(u)$ used in previous static versions [5] when n/u is sufficiently small. We prefer it in this paper to free us from any super-logarithmic dependence on u

Each element in ids is visited twice in this process (this is similar to the process done in the proof of Lemma 3), thus this first scan takes $O(n)$ time.

Then, we go on a second scan on the cycles of ids. We set A_{ids} to all zeros again, and allocate array Bwd of $\epsilon n \log n$ bits of space, which shall store the backward pointers of the permutation. We preprocess array B_{ids} with data structures to support $rank$. We start from the first element and follow the cycles once again. Visited elements are marked in A_{ids}, as before. Every time we reach a position i in the permutation such that $B_{ids}[i] = \mathtt{1}$, we store a backward pointer to the previously visited position j in the cycle, such that $B_{ids}[j] = \mathtt{1}$ (this means that there are $1/\epsilon$ elements between these two positions within the cycle). In other words, we set $Bwd[rank_{\mathtt{1}}(B_{ids}, i)] \leftarrow j$.

This second scan takes also $O(n)$ time. We finally free the space of A_{ids} and maintain bit vector B_{ids} as a marker of the positions storing the backward pointers. By adjusting ϵ to $\epsilon/2$ as in the static case we obtain:

Theorem 4. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the Scheme 3 of the LZ-index for T using $n(\log n + \max((1+\epsilon)\log n, \log u)) + 3n\log\sigma + O(n)$ bits of space and $O(u(\log\sigma + \log\log u))$ time, for any $0 < \epsilon < 1$. The main-memory space used at any time to construct Scheme 3 can be reduced to $n\max((1+\epsilon)\log n, \log u) + 3n\log\sigma + O(n)$ bits, in such a case performing $3n\log n + O(n)$ bits of I/O, plus those needed to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

Note that, by virtue of Lemma 2, the total space can be upper bounded by $(2+\epsilon)uH_k(T) + o(u\log\sigma)$, which is asymptotically the same space of the final index under this weaker model. Similarly, the main-memory space is at most $(1+\epsilon)uH_k + o(u\log\sigma)$, the same of the compressed text.

5.3 Space-Efficient Construction of Index of Lemma 4 and Relatives

The LZ-index of Lemma 4 is the smallest variant, requiring just $(1+\epsilon)n\log n + 3n\log\sigma + O(n)$ bits (plus the space for the skips, if desired). Recall from Section 3.4 that this LZ-index is a reduced-space version of Scheme 4. Hence, the procedure below

To construct it using the minimum possible extra space, we will need two passes over the text, and several traversals over *LZTrie* and *RevTrie* (yet the number of traversals is a constant). We carry out the following steps:

1. We build the hierarchical *LZTrie*, just storing the trie topology T_p and the symbols $letts_p$, without storing the phrase identifiers ids_p in each trie block p. This requires $n\log\sigma + O(n)$ bits of space, and takes $O(u(\log\sigma + \log\log u))$ time. We cannot yet erase the text, as we need it at a later step.
2. We build the final *LZTrie* from its hierarchical representation, in $O(u(\log\sigma + \log\log u))$ time and requiring $2n\log\sigma + O(n)$ bits of space. Recall that we do not store the phrase identifiers ids. We then free the hierarchical *LZTrie*, leaving $n\log\sigma + O(n)$ bits in use.
3. We traverse *LZTrie* in preorder, generating each LZ78 phrase B_i in constant time per string, and insert B_i^r into a hierarchical *RevTrie*. We store pointers to *LZTrie* nodes in the *RevTrie* nodes, just as in Section 4. This requires a maximum of $n\log n + 3n\log\sigma + O(n)$ bits of space after the hierarchical *RevTrie* is built, and takes $O(u(\log\sigma + \log\log u))$ time.
4. We build the final *RevTrie* from its hierarchical representation, storing the tree topology $rpar$ and recreating the $skips$ and $rletts$ arrays, which requires $n\log u + 3n\log\sigma + O(n)$ bits. The pointers to *LZTrie* nodes are now deleted, as these were used just to provide the connectivity between tries while

constructing *RevTrie*. This takes $O(u(\log\sigma + \log\log u))$ time. After freeing the hierarchical *RevTrie* we end up using just $n\log\frac{u}{n} + 3n\log\sigma + O(n)$ bits.

5. We allocate memory for array $R[1..n]$, of $n\log n + O(n)$ bits, which is constructed as follows. We traverse *LZTrie* in preorder, and for every phrase B_i corresponding to node v_{lz}, we look for B_i^r in *RevTrie*, obtaining node v_r as in Section 5.2. Then we store $R[preorder(v_r)] \leftarrow preorder(v_{lz})$. The overall work is $O(u\log\sigma)$. At this point we free the skip information (or we could retain it if desired for the final structure). Array R will be represented more space-efficiently (using function φ, which represents suffix links in *RevTrie*, see below). We then sample ϵn values of R, as explained in Arroyuelo et al. [7], ensuring that at most $O(1/\epsilon)$ suffix links are followed in order to compute a given $R[i]$.
6. We allocate space for arrays V_W and S_W [7], which are used to compute function φ' in *RevTrie*. This adds $n\log\sigma + O(n)$ extra bits. We traverse *RevTrie* in preorder, and for every non-empty node with preorder i we map to *LZTrie* using $R[i]$, and then write sequentially the degree of $R[i]$ in unary in V_W, and the symbols labeling the children of $R[i]$ in S_W. Then we preprocess V_W and S_W with data structures to support $rank$ and $select$ on them. This takes $O(n\log\sigma)$ time overall.
7. We build on R the data structure for inverse permutations of Munro et al. [52], using the same procedure as in Section 5.2. This takes $O(n)$ time. In a similar way as done for array R in Step 5 above, we sample ϵn values of R^{-1}, as explained in Arroyuelo et al. [7]. The overall space requirement raises to $(1+3\epsilon)n\log n + 4n\log\sigma + O(n)$ bits.
8. We use the approach of Chan et al. [11] to construct φ, which is originally defined for building function Ψ of Compressed Suffix Arrays [26, 65] requiring only $O(u\log\sigma)$ bits of space. In our case we compute $\varphi[i] = R^{-1}(parent_{lz}(R[i]))$ for consecutive i values, each in time $O(1/\epsilon)$ as we have R stored in plain form and R^{-1} represented with the structure of Munro et al. [52]. Since there is no point in using $\epsilon = o(\frac{1}{\log n})$ (as by then $\epsilon n\log n = o(n)$, so the times would increase without any asymptotic space gain), the overall time is $O(n/\epsilon) = O(n\log n) = O(u\log\sigma)$. We produce φ left-to-right, and thus we can directly generate it in compressed form: The $\varphi[i]$ values for all the preorders i of *RevTrie* nodes that descend from the same child of the root form an increasing sequence of values up to n [7]. So each of the (at most σ) increasing sequences can be represented using Okanohara and Sadakane's bitmaps [60], for a total space of $n\log\sigma + O(n)$ bits. Each $\varphi[i]$ value is then retrieved in constant time using $select_1$. The space has reached $(1+3\epsilon)n\log n + 5n\log\sigma + O(n)$ bits. We free R now.
9. We finally allocate memory for array ids, and set it with all zeros. We also set $i \leftarrow 1$. We perform a second pass on T to enumerate the LZ78 phrases (this yields $u\log\sigma$ extra I/O bits in case the text is stored on disk), descending in *LZTrie* with the symbols of T. Every time we reach a node v_{lz} in *LZTrie*, we check whether $ids[preorder(v_{lz})]$ is `0` or not. In the affirmative case, this means that the corresponding phrase has not yet been enumerated, and thus we store $ids[preorder(v_{lz})] \leftarrow i$ and set $i \leftarrow i+1$. We go back to the *LZTrie* root and go on with the next symbol of T. In case we arrive at a node v_{lz} with $ids[preorder(v_{lz})] \neq 0$, then we continue the descent from this node, since its phrase has been already enumerated. This takes $O(u\log\sigma)$ time. Finally, we can erase the text.

By rewriting 3ϵ as ϵ, which does not change time complexities, we obtain:

Theorem 5. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the LZ-index of Lemma 4 for T using $n\max((1+\epsilon)\log n, \log u) + 5n\log\sigma + O(n)$ bits of space and $O(u(\log\sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$. The algorithm performs two passes over text T, thus requiring $u\log\sigma$ I/O bits in addition*

to those for writing the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.

We can use this algorithm to construct the LZ-index of Lemma 5, which only adds the *Range* data structure. For this sake, we do not delete R at the end of Step (8) of the previous algorithm, but rather move it to disk and then execute Step (9), after which we have ids in main memory. Now we read R sequentially from disk and compose it with ids, progressively replacing R by $rids$ on disk. Now we invert ids in main memory (using $O(n)$ extra bits, Lemma 3), and read $rids$ sequentially from disk, progressively replacing it by array $RQ[i+1] = ids^{-1}[rids[i+1]+1]$. Now we invert again ids^{-1} to obtain ids, which is swapped with the RQ array that is on disk. Finally, $Range$ is built from RQ as explained in Section 4.7, Step (5), and the result written to disk.

Corollary 1. *Let text $T[1..u]$, over an alphabet of size σ, be parseable into n phrases by the LZ78 algorithm. Then there exists an algorithm to construct the LZ-index of Lemma 5 for T using $n \log n + n \max((1+\epsilon)\log n, \log u) + 5n \log \sigma + O(n)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$. The algorithm requires $u \log \sigma + 5n \log n + O(n)$ bits of I/O in addition to those needed to write the final index to disk. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

Finally, the LZ-index of Lemma 6 adds the *Alphabet-Friendly FM-index* [19], which according to González and Navarro [24] can be constructed with $uH_k(T) + o(u \log \sigma)$ bits of space in $O(u \log u(1 + \frac{\log \sigma}{\log\log u}))$ time. Then, we have:

Corollary 2. *There exists an algorithm to construct the LZ-index of Lemma 6 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(3+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(u \log u(1 + \frac{\log \sigma}{\log\log u}))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm requires $u \log \sigma + 5uH_k(T) + o(u \log \sigma)$ I/O bits, in addition to those needed to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

6 Experimental Results

We implemented a simplification of the algorithm presented in Section 4, which shall be tested in this section. We run our experiments on an Intel(R) Pentium(R) 4 processor at 3 GHz, 4 GB of RAM and 1MB of L2 cache, running version 2.6.13-gentoo of Linux kernel. We compiled the code with `gcc 3.3.6` using full optimization. The disk is a Maxtor DiamondMax Plus 9 of 120GB and 7,000 rpm, with interface DMA/ATA-133 (Ultra) Fast Drives, buffer of 2MB, average seek time of 9 ms, and transfer rate of 133 MB/sec (yet we will soon show that the influence of the disks in our performance is very slight). Construction times were averaged over 10 repetitions.

6.1 A Practical Implementation of Hierarchical Tries

We implement our construction algorithms for Scheme 2 and Scheme 3, and use a simpler representation for the hierarchical trie, just as defined in our original work [4]. In this simpler representation, every block in the tree uses contiguous memory space, which stores all the block components. We define different block capacities $N_m < N_2 \ldots < N_M$, and say that a block of size N_i is able to store up to N_i nodes. When we want to insert a node in a block p of size $N_i < N_M$ which is already full, we first create a new block of size

N_{i+1}, copy the content of p to the new one, and then insert the new node within this block. This is called a grow operation. If the full block p is of size N_M, we say that p overflows. In such a case we proceed as explained in Section 4.1, with the only difference that the subtree to be reinserted is searched by traversing the whole block (we choose the subtree of maximum size not exceeding $N_M/2$ nodes, just as in our previous work [4]).

To ensure a minimum fill ratio $0 < \alpha < 1$ in the trie blocks, thus controlling the wasted space, we define $N_i = N_{i-1}/\alpha$, for $i = 2, \ldots, M$, and $1 \leqslant N_m \leqslant 1/\alpha$. Notice that parameter α allows us for time/space trade-offs: smaller values of α yield a poor utilization of blocks, yet they trigger a smaller number of grow operations (which are expensive) as we insert new nodes. The opposite occurs for large values of α.

The block representation is completely static: the whole block is rebuilt from scratch upon insertions, or upon block overflows. Each block is allocated as a single chunk of main memory, using the standard function `malloc`. We represent the trie topologies with balanced parentheses rather than with DFUDS. We do not store information to quickly navigate the parentheses within each block. So, we navigate them by brute force (using precomputed tables to avoid a bit-per-bit scan, just as for the balanced parentheses data structure of Navarro [56]). In the case of *Revtrie*, we store $rletts$ in each block, yet the skips value are not stored, but computed by successively going to the parent in *LZTrie* (which is by then already in static form). In this way, navigations can be a little bit slower, yet we save space and time reconstructing these data structures after every insertion. We will show, however, that we achieve competitive results in practice.

We use the following parameters throughout our experiments: $N_m = 2$, $N_M = 1024$, and $\alpha = 0.95$, according to the preliminary results obtained in our previous work [4]. We implement the reduced-memory model presented in Section 4.7. We also show the results for the model in which only main memory is used, where in most cases the maximum total space coincides with the size of the final LZ-index. We use the `memusage` application by Ulrich Drepper[12] to measure the peaks of main memory usage. Since our algorithms need to use the disk to store intermediate partial results, we measure the user time plus the system time of our algorithms.

We show the results only for Scheme 2 and Scheme 3, since these are the most competitive in practice [5], and also because the most critical points along the indexing algorithm (i.e., the construction of the hierarchical tries) is the same for all schemes (including the original LZ-index). For Scheme 3, we choose parameters $1/\epsilon = 1$ and $1/\epsilon = 15$ for the inverse-permutation data structures. These represent the extreme cases (both for time and space requirements) tested in Arroyuelo and Navarro [5]; intermediate values offer interesting results as well. Note that when $1/\epsilon = 1$ the space requirement of Scheme 3 is the same as that of the original LZ-index.

6.2 Indexing English Texts

For the experiments with English texts we use the 1-GB file provided in the *Pizza&Chili Corpus*, downloadable from `http://pizzachili.dcc.uchile.cl/texts/nlang/english.1024MB.gz`.

In Table 3(a) we show the results for English text. As it can be seen, the most time-consuming tasks along the construction process are that of building the hierarchical representations of the tries, taking up 96–98% of the time. For *LZTrie*, the construction rate is about 1.01 MB/sec, while for *RevTrie* the result is about 0.39 MB/sec. Thus, *RevTrie* is much slower than *LZTrie* to be built. The overall average indexing rate is 0.29 MB/sec for Scheme 2, 0.29 MB/sec for Scheme 3 ($1/\epsilon = 1$), and 0.28 MB/sec for Scheme 3

[12] `http://pizzachili.dcc.uchile.cl/utils/memusage/memusage-2.2.2.tar.gz`

($1/\epsilon = 15$). As it can be seen, the sample rate of the inverse permutations in Scheme 3 does not affect much the indexing speed. Furthermore, because the construction of tries is an in-memory task, one can see that the impact of moving data from/to disk is very low, thus the practical performance of our reduced-memory schemes is almost the same as those using all the main memory they need.

Table 3. Experimental results for English text and Human Genome. Numbers in boldface indicate the final index size in every case.

(a) English Text.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time secs
Scheme 2	1	411,928,076	411,928,076	909.37
	2	505,729,592	822,801,159	17.55
	3	574,548,639	819,749,431	2,554.07
	4	454,026,216	883,576,755	15.01
	5 & 6	491,169,360	**965,869,767**	52.19
	Peak	574,548,639	965,869,767	3,549.20
Scheme 3 $1/\epsilon = 1$	1	411,928,076	411,928,076	898.40
	2	505,729,592	822,801,159	17.51
	3	574,548,639	819,749,431	2,590.78
	4	454,026,216	883,576,755	14.86
	5 & 6	491,169,360	**1,204,608,375**	62.00
	Peak	574,548,639	1,204,608,375	3,583.56
Scheme 3 $1/\epsilon = 15$	1	411,928,076	411,928,076	896.88
	2	505,729,592	822,801,159	17.46
	3	574,548,639	819,749,431	2,588.83
	4	454,026,216	883,576,755	14.81
	5 & 6	274,463,684	**771,197,007**	102.80
	Peak	574,548,639	883,576,755	3,620.87

(b) Human Genome.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time secs
Scheme 2	1	1,233,336,206	1,233,336,206	2,440.33
	2	1,428,595,278	2,442,409,424	51.73
	3	1,677,938,853	2,467,406,392	13,966.22
	4	1,405,350,330	2,665,257,752	45.00
	5 & 6	1,579,033,696	**2,985,958,274**	181.96
	Peak	1,677,938,853	2,985,958,274	16,685.28
Scheme 3 $1/\epsilon = 1$	1	1,233,336,206	1,233,336,206	2,443.83
	2	1,428,595,278	2,442,409,424	51.98
	3	1,677,938,853	2,467,406,392	13,791.08
	4	1,405,350,330	2,665,257,752	44.93
	5 & 6	1,579,033,696	**3,775,475,122**	211.81
	Peak	1,677,938,853	3,775,475,122	16,543.63
Scheme 3 $1/\epsilon = 15$	1	1,233,336,206	1,233,336,206	2,445.02
	2	1,428,595,278	2,442,409,424	51.61
	3	1,677,938,853	2,467,406,392	13,812.29
	4	1,405,350,330	2,665,257,752	44.92
	5 & 6	841,516,932	**2,300,440,426**	365.18
	Peak	1,677,938,853	2,665,257,752	16,719.02

For Scheme 2, the maximum main-memory peak is reached at Step 3, and it is of about 548 MB. This means about 0.54 times the size of the original text needed to construct the Scheme 2 for the English text. Also, this space is 0.59 times that of the final Scheme 2. When comparing the space required by the hierarchical trie representations with that required by the final trie representations, we have 411,928,076 bytes for the hierarchical *LZTrie* and 408,876,348 bytes for the hierarchical *RevTrie*, versus 410,873,083 bytes for *LZTrie* and 309,412,004 bytes for *RevTrie*. This means that the hierarchical *LZTrie* requires about 1.01 times the size of the final *LZTrie*, while the hierarchical *RevTrie* requires about 1.32 times the size of the final *RevTrie*. The bigger difference between *RevTrie* representations comes from the fact that the hierarchical *RevTrie* stores the symbols labeling the arcs, while the final *RevTrie* does not. Table 4(a) summarizes.

The results are very similar for Scheme 3 and $1/\epsilon = 1$. For $1/\epsilon = 15$, however, the peak of memory usage when considering the total indexing space at each step is reached at Step 4, and it is slightly greater than the space needed by the final Scheme 3 (more precisely, 1.15 times the size of the final Scheme 3).

As a comparison, we indexed a 500-MB prefix of this text with the original construction algorithm of Scheme 2, using an approach similar to that used in Navarro [56], with non-space-efficient intermediate representation for the tries. The peak of main memory is 1,566 MB (this means 3.13 times the size of the

original text)[13], with an indexing rate of about 1.29 MB/sec (see Table 4(b)). Applied on this same prefix, our new indexing algorithm is 6.45 times slower than the original one (see column "Slowdown" in Table 4(b)), yet it requires 4.29 times less memory than the original (see column "Space reduction" in Table 4(b)). The intermediate *LZTrie* of the original algorithm required 751,817,455 bytes (this is 2.65 times the size of our hierarchical *LZTrie* on this same prefix, see column "Intermediate *LZTrie*" in Table 4(b)), while the intermediate *RevTrie* required 1,185,969,250 bytes (this is 4.31 times the size of our hierarchical *RevTrie*, see column "Intermediate *RevTrie*" in Table 4(b)). Note the bigger difference among *RevTrie* representations. This is because we are not only using a space-efficient representation, but also because we are compressing empty unary paths at reverse-trie construction time. Thus, we can conclude that our space-efficient trie representations are effective to reduce the indexing space of LZ-index schemes. The price is, on the other hand, a slower construction.

6.3 Indexing the Human Genome

For the test on DNA data we indexed the Human Genome[14], whose size is about 3,182MB. In Table 3(b) we show the results obtained with our construction algorithm. The indexing rate for the hierarchical *LZTrie* is about 1.30 MB/sec, while for *RevTrie* it is about 0.23 MB/sec. The total indexing time (user time plus system time) is about 4.63 hours, which means an overall indexing rate of about 0.19 MB/sec.

See Table 4(a) for the statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 4(b) for a comparison with the original construction algorithm for Scheme 2, indexing a 500-MB prefix of the Human Genome.

Now we show that the running times of our algorithms are comparable to those of state-of-the-art methods. Hence, we test the practical indexing times for the best indexing algorithms we know of:

- The space-efficient algorithm from Sirén [32] to build the Burrows-Wheeler transform of a text collection. In particular, the algorithm is used to build the Run-length Compressed Suffix Array [45] (RL-CSA for short). We divided the Human Genome into several equal-size files. To obtain different space/time trade-offs, we used 25 (which was the value tested by Sirén [32]), 50, 100 and 500 files. We used the same construction parameters as in the original article [32]. The program was run in our machine.
- The algorithm for constructing suffix arrays from Dementiev et al. [15]. Most of the work of this algorithm is carried out on secondary storage, using just a constant amount of main memory. Therefore its performance depends basically on the speed of the disk used, whereas ours depend mostly on the CPU speed. As the disks they used are similar or faster than ours, we directly report their experimental results [15] instead of rerunning them in our machine. They report results in two scenarios: (*i*) a 2.0GHz Intel Xeon processor, 1GB of RAM and eight 80GB ATA IBM 120GXP disks (these are similar to those in our machine: 7,200 rpm, 8.5 ms seek time, 2MB buffer, 100 MB/sec transfer rate); and (*ii*) a more powerful SMP system with four 64-bit AMD Opteron 1.8 GHz processors (just one processor was used), 8GB of RAM (just 1GB was used by the algorithms) and eight 73GB SCSI Seagate ST373453LC disks (these spin at 15,000 rpm, have 8 MB buffers and 3.6 ms seek time; their transfer rate is 320 MB/sec).

Table 5 shows the results. As can be seen, for 25 files the indexing time for the RL-CSA is 4.33 hours, with a memory peak of 2,299 MB. Thus, the construction time is sligthly better than ours, though using more

[13] It is important to note that the original algorithm uses just main memory to construct Scheme 2

[14] `http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/est.fa.gz`.

Table 4. Some statistics for our construction algorithms.

(a) Statistics for our space-efficient indexing algorithm for Scheme 2. The results for Scheme 3 are similar.

Text	Main-memory peak	Size hierarchical *LZTrie*	Size hierarchical *RevTrie*
English (1 GB)	0.54 times text size. 0.59 times size of final Scheme 2.	411,928,076 bytes (1.01 times size of final *LZTrie*)	309,412,004 bytes (1.32 times size of final *RevTrie*)
Human Genome (3.11 GB)	0.50 times text size. 0.44 times size of final Scheme 2.	1,233,336,206 bytes (1.02 times size of final *LZTrie*)	1,209,073,218 bytes (1.27 times size of final *RevTrie*)
XML (285 MB)	0.40 times text size. 0.61 times size of final Scheme 2.	90,563,835 bytes (1.07 times size of final *LZTrie*)	84,591,900 bytes (1.29 times size of final *RevTrie*)
Proteins (1 GB)	1.05 times text size. 0.51 times size of final Scheme 2.	839,446,471 bytes (0.99 times size of final *LZTrie*)	807,660,745 bytes (1.28 times size of final *RevTrie*)

(b) Statistics for the construction of Scheme 2 versus the non-space-efficient original algorithm. The first two columns refer to the latter. Column "Slowdown" shows the slowdown experienced by using our space-efficient algorithm instead of the original one. "Space reduction" indicates the factor of space reduction gained by using our algorithm instead of the original one. Finally, columns "Intermediate *LZTrie*" and "Intermediate *RevTrie*" show the size of the (non-space-efficient) intermediate data structures used to build the final tries, as a fraction of the size of the final trie representations.

Text	Main-memory peak	Indexing rate (MB/sec)	Slowdown	Space reduction	Intermediate *LZTrie*	Intermediate *RevTrie*
English (500 MB)	1,566 MB ($3.13 \times$ text)	1.29	6.45	4.29	2.65	4.31
Genome (500 MB)	1,275 MB ($2.55 \times$ text)	1.86	8.86	4.46	2.74	3.47
XML (285 MB)	862 MB ($3.02 \times$ text)	2.31	5.25	7.50	2.68	9.02
Proteins (500 MB)	1,781 MB ($3.56 \times$ text)	1.82	8.27	3.63	2.68	3.41

main memory. For 500 files, the indexing time raises to 18.79 hours, whereas the memory peak decreases to 1,799 MB. This is closer but still higher than our memory usage. Hence, the indexing space can be reduced to approach ours, yet at the price of degrading much the indexing time.

Using computer (*i*) above, the algorithm of Dementiev et al. [15] indexes the Human Genome in about 8.52 hours, using secondary storage and just a constant amount of main memory. By using computer (*ii*), on the other hand, the indexing times are reduced to 5.11 hours. This is comparable to our results (yet, remember that different structures are being built, so this is not a direct competition but rather tries to put the practicality of our LZ-index construction in context).

The comparison shows that our LZ-index construction is at least as practical as the best constructions of suffix-array-based indexes. This is a very relevant result, specifically for biological research, since it demonstrates that it is feasible to index the Human Genome within less than 5 hours and in the main memory of a desktop computer.

Table 5. Comparison of the best indexing algorithms to construct an index for the Human Genome.

Index	Construction algorithm	Indexing time	Maximum indexing space (RAM)
Run-length Compressed Suffix Arrays – 25 files	[32]	4.33 hours	2,299 MB
Run-length Compressed Suffix Arrays – 50 files	[32]	4.98 hours	2,038 MB
Run-length Compressed Suffix Arrays – 100 files	[32]	6.33 hours	1,904 MB
Run-length Compressed Suffix Arrays – 500 files	[32]	18.79 hours	1,799 MB
Suffix array – on computer (i)	[15]	8.52 hours	1,024 MB
Suffix array – on computer (ii)	[15]	5.11 hours	1,024 MB
Scheme 2 of LZ-index	This paper	4.63 hours	2,847 MB
Scheme 2 – reduced-memory model	This paper	4.63 hours	1,597 MB

As a historical note to illustrate the evolution of text indexing technologies, there are several results on indexing the Human Genome in the literature:

- Kurtz [39] indexed this text in less than 9 hours on a Sun-UltraSparc 300 MHz, 192 MB of main memory, under Solaris 2. The main-memory usage was of about 45.31 GB.
- Sadakane and Shibuya [68] constructed the suffix array for the Human Genome, and used it to construct the Compressed Suffix Array. They used an IBM SP-2 (450MHz CPU) with 64GB of RAM to achieve 7 hours of indexing time. The indexing space was about 12GB.
- Hon et al. [29, 28] indexed the Human Genome with the CSA in about 24 hours, using a Pentium IV processor at 1.7 GHz with 512 KB of L2 cache, and 4 GB of main memory, running Solaris 9 operating system. They also constructed the FM-index in about 4 extra hours, for a total of about 28 hours.

6.4 Indexing XML Data

Another relevant application is that of compressing and searching XML texts. Nowadays many applications handle text data in XML format, which are automatically generated in large amounts. It is interesting therefore to be able to compress such data, while at the same time being able to search and extract any

part of the text, since XML data is usually queried and navigated by other applications. We indexed the file `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.gz` of about 285 MB provided in the *Pizza&Chili* Corpus. This text is highly compressible.

In Table 6(a) we show the results for XML text. The indexing rate for *LZTrie* is about 1.43 MB/sec, while for *RevTrie* it is about 0.65 MB/sec. The overall indexing rate is about 0.44 MB/sec. See Table 4(a) for statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 4(b) for a comparison with the original construction algorithm.

Table 6. Experimental results for XML text and proteins. Numbers in boldface indicate the final index size in every case.

(a) XML text.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time secs
Scheme 2	1	90,563,835	90,563,835	199.74
	2	111,467,467	175,009,211	3.82
	3	120,592,538	169,037,276	435.20
	4	98,337,536	185,878,936	3.23
	5 & 6	97,231,032	**198,518,068**	9.29
	Peak	120,592,538	198,518,068	651.28
Scheme 3 $1/\epsilon = 1$	1	90,563,835	90,563,835	201.43
	2	111,467,467	175,009,211	3.88
	3	120,592,538	169,037,276	441.91
	4	98,337,536	185,878,936	3.24
	5 & 6	97,231,032	**245,871,260**	11.02
	Peak	120,592,538	245,871,260	661.41
Scheme 3 $1/\epsilon = 15$	1	90,563,835	90,563,835	200.91
	2	111,467,467	175,009,211	3.79
	3	120,592,538	169,037,276	441.34
	4	98,337,536	185,878,936	3.20
	5 & 6	54,641,864	**160,692,920**	18.66
	Peak	120,592,538	185,878,936	667.91

(b) Proteins.

Index	Indexing step	Main-memory space (bytes)	Total space (bytes)	Time secs
Scheme 2	1	839,446,471	839,446,471	1,087.58
	2	1,018,660,027	1,681,050,175	33.82
	3	1,133,180,292	1,649,264,449	4,105.11
	4	895,675,465	1,766,181,601	27.83
	5 & 6	1,032,374,144	**1,990,895,000**	112.75
	Peak	1,133,180,292	1,990,895,000	5,374.88
Scheme 3 $1/\epsilon = 1$	1	839,446,471	839,446,471	1,095.56
	2	1,018,660,027	1,681,050,175	33.49
	3	1,133,180,292	1,649,264,449	4,113.27
	4	895,675,465	1,766,181,601	27.55
	5 & 6	1,032,374,144	**2,502,718,500**	134.72
	Peak	1,133,180,292	2,502,718,500	5,404.62
Scheme 3 $1/\epsilon = 15$	1	839,446,471	839,446,471	1,097.09
	2	1,018,660,027	1,681,050,175	33.86
	3	1,133,180,292	1,649,264,449	4,117.30
	4	895,675,465	1,766,181,601	27.62
	5 & 6	575,948,072	**1,589,866,364**	232.25
	Peak	1,133,180,292	1,766,181,601	5,508.14

6.5 Indexing Proteins

Another interesting application of text-indexing tools in biological research is that of indexing proteins. We indexed the text `http://pizzachili.dcc.uchile.cl/texts/protein/proteins.gz` of about 1 GB provided in the *Pizza&Chili* Corpus. This is a not so compressible text.

In Table 6(b) we show the results for proteins. The indexing rate for the hierarchical *LZTrie* is about 0.92 MB/sec, while for *RevTrie* it is about 0.24 MB/sec. The indexing rate for *RevTrie* is much slower than for other texts. This could be mainly because proteins are not so compressible, and then the tries have a greater number of nodes to be inserted, making the process slower. The overall indexing rate is about 0.19 MB/sec.

See Table 4(a) for the statistics regarding the memory peak of the algorithm, as well as a comparison between intermediate and final trie representations. See Table 4(b) for a comparison with the original construction algorithm for Scheme 2, indexing a 500-MB prefix of Proteins.

7 Conclusions and Future Work

The space-efficient construction of compressed full-text self-indexes is a very important aspect regarding their practicality. In this paper we proposed a space-efficient algorithm to construct Navarro's LZ-index [55]. Given the data structures that conform the LZ-index, this problem is highly related to the representation of succinct dynamic σ-ary trees. Thus, the basic idea is to construct the tries of the LZ-index using space-efficient intermediate representations supporting fast incremental insertion of nodes. Our algorithm requires asymptotically the same space as the final LZ-index. Let a text $T[1..u]$ over an alphabet of size σ be compressed by the LZ78 algorithm into a representation LZ. Then the size of Navarro's LZ-index is $4|LZ|(1 + o(1))$ bits, and this is also the space needed by the algorithm introduced in this article to build such index, within $O(u(\log \sigma + \log\log u))$ time. We also show that all LZ-index variants presented in previous work [7, 5], requiring from $(1 + \epsilon)|LZ|(1 + o(1))$ to $3|LZ|(1 + o(1))$ bits, can be constructed within the same asymptotic space needed by the final index (the $o(1)$-factor is small, $O(\frac{1}{\log |LZ|})$) and within the same time as before. These smaller indexes are able to replace the original LZ-index in many practical scenarios [5], hence the importance to space-efficiently construct them.

We defined an alternative model in which we have a reduced amount of main memory to perform the indexing process (perhaps less memory than that needed to accommodate the whole index). We show that several LZ-indexes can be constructed within $|LZ|(1 + o(1))$ bits of main memory space, in $O(u(\log \sigma + \log\log u))$ time and with $O(|LZ|)$ I/Os. Others need slightly more space, $\epsilon|LZ|$ for a small value $0 < \epsilon < 1$ or $|LZ|\frac{\log u}{\log |LZ|}$. This means that the LZ-indexes can be constructed essentially within the same space than that required to store the compressed text.

Our experimental results indicate that all LZ-index versions can be constructed in practice within almost the same amount of memory than needed by the final index. Under the reduced-memory scenario, we have that the LZ-index versions can be constructed requiring main memory to hold 0.40 – 1.05 times, and using overall space 0.66 – 1.84 times, the size of the original text, depending on its compressibility. This means about 3.39 – 7.50 times less space than that needed by the original construction algorithm (which works assuming that there is enough memory to store the whole index in main memory). Our indexing rate is about 0.19 – 0.44 MB/sec., which is 5.25 – 8.86 times slower than the original construction algorithm. In conclusion, our algorithm requires much less memory than the original one, in exchange for a higher construction time. Still, our indexing algorithm is competitive with existing indexing technologies. For example, we are able to construct the LZ-index for the Human Genome in less than 5 hours, indexing algorithms in the literature for constructing other indexes like suffix arrays [15] and Compressed Suffix Arrays [32].

An interesting application of our indexing algorithm is in the construction of the LZ78 parsing of a text T. Grossi and Sadakane [66] define an alternative representation for the LZ78 parsing, which has the nice property of supporting optimal-time access to any text substring. The parsing consists basically of *LZTrie* (the trie topology and the array of edge symbols), plus an array that, for any phrase identifier i, stores the preorder of the corresponding *LZTrie* node. Using our notation, the latter is just array ids^{-1}. Jansson et al. [33] propose an algorithm to construct the parsing in $O(\frac{u}{\log_\sigma u}\frac{(\log\log u)^2}{\log\log\log u})$ time and requiring $uH_k(T) + o(u \log \sigma)$ bits of space. The algorithm, however, needs two passes over the text, which involves $|T| = u \log \sigma$ extra bits of I/O if it is stored on disk, which can be expensive. We can reduce the number of disk accesses as follows, mainly when the text is compressible:

– We construct the hierarchical *LZTrie* for T, storing the phrase identifier for each node. We can erase T since it is not anymore necessary. This takes $O(u(\log \sigma + \log\log u))$ time.

– We build the final *LZTrie*, storing array ids on disk, as explained in Section 4.7. This takes $O(u(\log\sigma + \log\log u))$ further time, and carries out $|LZ|$ extra bits of I/O.
– We then free the hierarchical *LZTrie* and load array ids back to main memory, performing $|LZ|$ bits of further I/O.
– We compute ids^{-1} in place, using the algorithm of Lemma 3, and this way we complete the representation for the LZ78 parsing of text T.

As seen, we exchange the $|T|$ bits of extra I/O of Jansson et al. [33] by $2|LZ|$. This can be much better in the case of large compressible texts. The total time is $O(u(\log\sigma + \log\log u))$, and the maximum main-memory space used is $|LZ|(1 + o(1))$ bits. We think that our methods could be extended to build related LZ-indexes [18, 64] within limited space.

Finally, recent advances [27, 59] (not all refereed yet) seem to indicate that it is possible to handle all the classical operations on a tree of n nodes within $2n + o(n)$ bits and $O(\frac{\log n}{\log\log n})$ time; and that a dynamic sequence of length n over an alphabet of size σ can be handled within $n\log\sigma(1 + o(1))$ bits and $O(\frac{\log n}{\log\log n}(1+\frac{\log\sigma}{\log\log n}))$ time per operation, which also may extend to partial sums. In such a case, we would be able to handle the operations of our tree blocks of size N within time $O(\frac{\log N}{\log\log u})$, and as a consequence all our construction times would drop from $O(u(\log\sigma + \log\log u))$ to the familiar $O(u(1 + \frac{\log\sigma}{\log\log u}))$. This is the time for carrying out u operations on a *static* FM-index, whereas a dynamic FM-index construction would pose an extra $O(\frac{\log u}{\log\log u})$ factor in the time complexity.

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. An improved succinct representation for dynamic k-ary trees. In *Proc. 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 277–289, 2008.
3. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 84–97, 2010.
4. 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. Springer, 2005.
5. 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, Dept. of Computer Science, University of Chile, 2008. `http://www.dcc.uchile.cl/TR/2008/TR_DCC-2008-009.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. D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. To appear in *Algorithmica*, DOI 10.1007/s00453-010-9443-8. See also `http://www.dcc.uchile.cl/~darroyue/papers/algor2010.pdf`., 2010.
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. A. Brodnik, S. Carlsson, E. Demaine, J. I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. In *Proc. WADS*, LNCS 1663, pages 37–48. Springer, 1999.
10. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
11. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):article 21, 2007.
12. B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17(3):427–462, 1988.

13. 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.
14. T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. Prentice–Hall, second edition, 2001.
15. R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics (JEA)*, 12:1–24, article 3.4, 2008.
16. P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 697–710, 2010.
17. P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009.
18. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.
19. 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.
20. F. Fich, J. I. Munro, and P. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.
21. G. Franceschini and S. Muthukrishnan. In-place suffix sorting. In *Proc. of 34th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 4596, pages 533–546, 2007.
22. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science*, 368(3):231–246, 2006.
23. R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Vol. of 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 27–38. CTI Press and Ellinika Grammata, 2005.
24. R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410:4414–4422, 2008.
25. 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.
26. 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.
27. M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2010. To appear.
28. W.-K. Hon. *On the Construction and Application of Compressed Text Indexes*. PhD thesis, University of Hong Kong, 2004.
29. W. K. Hon, T. W. Lam, K. Sadakane, and W. K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS 2906, pages 240–249, 2003.
30. 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.
31. W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM J. Comput.*, 38(6):2162–2178, 2009.
32. Sirén J. Compressed suffix arrays for massive data. In *Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 63–74, 2009.
33. J. Jansson, K. Sadakane, and W.-K. Sung. Compressed dynamic tries with applications to LZ-compression in sublinear time and space. In *27th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 424–435, 2007.
34. 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.
35. J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.
36. 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.
37. D. Kim, J. Na, J. Kim, and K. Park. Efficient implementation of rank and select functions for succinct representation. In *Proc. 4th Workshop on Experimental and Efficient Algorithms (WEA)*, pages 315–327. LNCS 3503, 2005.
38. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
39. S. Kurtz. Reducing the space requeriments of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
40. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
41. V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.
42. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.
43. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

44. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
45. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
46. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
47. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
48. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.
49. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
50. I. Munro, V. Raman, and S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2):205–222, 2001.
51. J. I. Munro. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS 1180, pages 37–42, 1996.
52. 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.
53. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
54. J. Na and K. Park. Alphabet-independent linear-time construction of compressed suffix arrays using $o(n \log n)$-bit working space. *Theoretical Computer Science*, 385:127–136, 2007.
55. G. Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.
56. G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13(article 2), 2009.
57. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.
58. G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.
59. G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. Technical Report arXiv:0905.0768v4, ArXiv, 2010.
60. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.
61. M. Pătraşcu. Succincter. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 305–313, 2008.
62. 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.
63. R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP)*, LNCS 2719, pages 357–368, 2003.
64. L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 5(3):501–513, 2007.
65. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.
66. 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.
67. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 134–149, 2010.
68. K. Sadakane and T. Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.
69. J. S. Vitter. *Algorithms and Data Structures for External Memory*. Series on Foundations and Trends in Theoretical Computer Science. Now Publishers, 2008.
70. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.
71. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, second edition, 1999.
72. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.
73. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.