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

Diego Arroyuelo ⋆⋆ and Gonzalo Navarro ⋆⋆⋆

Dept. of Computer Science, Universidad 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 $4uH_k(T) + o(u \log \sigma)$ bits of space, where $H_k(T)$ is the k-th order empirical entropy of T. Although in practice the LZ-index needs 1.0-1.5 times the text size, its construction requires much more main memory (around 5 times the text size), which limits its applicability only to not so large texts. In this paper we present an space-efficient algorithm to construct the LZ-index in $O(u(\log \sigma + \log \log u))$ time and requiring $4uH_k(T)+o(u \log \sigma)$ bits of space. Our experimental results show that our method is efficient in practice, needing an amount of memory close to that of the final index, and outperforming by far the construction time of other compressed indexes. We also adapt our algorithm to construct some recent reduced versions of the LZ-index, showing that these can also be built without using extra space on top of that required by the final index.
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). We show how to build all the LZ-index alternatives in $O(u(\log \sigma + \log \log u))$ time, and within $uH_k(T) + o(u \log \sigma)$ bits of space.

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 [67, 58] (able of finding whole words and phrases), until one decade ago it was 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 indexes available were the suffix arrays [47], requiring $u \log u$ bits[1] to index a text of u symbols. Since the text requires $u \log \sigma$ bits to be represented, this index is usually much larger than the text (typically 4 times the text size). With the huge texts available nowadays (e.g., the Human Genome consists of about 3×10^9 base pairs), one solution is to store the indexes on

⋆ A preliminary partial version of this paper appeared in *Proc. ISAAC 2005*, pp. 1143–1152.
⋆⋆ Funded by CONICYT PhD Fellowship Program, Chile. Most of this work was done while the author was in 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, Grant ICM P05-001-F, Mideplan, Chile.
[1] $\log x$ means $\lceil \log_2 x \rceil$ in this paper.

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 trees [2] or arrays have been made [36, 40, 1], focusing on good engineering to reduce the space. A parallel track [37, 29, 43, 63, 17, 26, 55, 44, 18, 62] focused on *compressed indexing*, which takes advantage of the regularities of the text to operate in space proportional to that of the compressed text (e.g., 3 times the zero-order entropy of the text). Especially, in some of those works [63, 17, 26, 27, 55, 44, 18] the indexes *replace* the text and, using little space (sometimes even less than 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 unprecedented breakthrough.

As with text compression, using compressed indexes increases processing time. However, given the relation between main and secondary memory access times, it is preferable to handle compressed indexes entirely in main memory, to handling them in uncompressed form in secondary storage.

The main families of compressed self-indexes [57] are *Compressed Suffix Arrays* (CSA for short) [29, 63, 26], indexes based on *backward search* [17, 44, 18] (which are alternative ways to compress suffix arrays, and known as the *FM-index* family), and the indexes based on the *Lempel-Ziv* compression algorithm [68] (LZ-indexes for short) [37, 55, 17, 62].

We are particularly interested in LZ-indexes, since they have shown to be effective in practice for extracting text, displaying occurrence contexts, and locating the occurrences, outperforming other compressed indexes at these tasks [55, 56]. What characterizes the particular niche of LZ-indexes is the $O(uH_k(T))$ space combined with $O(\log u)$ theoretical worst-case time per located occurrence. Moreover, in practice many pattern occurrences can be actually found in constant time per occurrence, which makes the LZ-indexes competitive, for example, to search for short patterns. In particular, we will be interested in Navarro's LZ-index [55, 56].

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 *Compressed Suffix Arrays* (CSA) [29, 63] and *FM-index* [17] involves building first the suffix array of the text, using for example the algorithm of Larsson and Sadakane [42] or the one by Manzini and Ferragina [49]. 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 CSA and the FM-index, by using the deep-shallow algorithm [49]). For example, the Human Genome may fit in less than 1 GB of main memory using these indexes (and thus it can be operated entirely in RAM on a 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. [33] 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 $O(u \log^{\epsilon} u)$ time for suffix trees, where $0 < \epsilon < 1$. Thus, we have 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, in the case of FM-index assuming $\log \sigma = o(\log u)$. However, the space requirement to construct the CSA is still bigger than that needed by the final index.

The works of Lam et al. [41] and Hon et al. [31, 32] deal with the space (and time) efficient construction of CSA. The former presents an algorithm that 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 may be impractical for larger alphabets such as Oriental languages.

The second work [32] 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 CSA using negligible extra space in $O(u)$ time. In practice they are able to build the CSA for the Human Genome in about 24 hours and requiring about 3.6 GB of main memory [30], 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.

Finally, Na and Park [54] construct the CSA in $O(u\log\sigma\log_\sigma^{\log_3 2} u)$ 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.

Thus, 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 in the literature. Since LZ-indexes are competitive in practice for locating pattern occurrences and extracting text substrings [56, 5] (which is very important for self-indexes), the space-efficient construction of LZ-indexes is also 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 $O(u\log\sigma)$ time, while requiring $4uH_k(T)+o(u\log\sigma)$ bits of space. This is the same space the final LZ-index requires to operate. 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) [18] has appeared, requiring $uH_k(T)+o(u\log\sigma)$ bits of space, and $O(u\log u\log\sigma)$ time [46], and even $O(u\log u\frac{\log\sigma}{\log\log u})$ time [25]. The time obtained in the present paper also improves upon the $O(\sigma u)$ worst-case time of [4].

We show how the reduced versions of LZ-index [6, 5, 7] can be constructed within little space. 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. This model has applications in cases where the indexing process must be carried out in a computer that is not so powerful so as 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 $(1+\epsilon)uH_k(T)+o(u\log\sigma)$ bits of space, for any $0<\epsilon<1$, in $O(u(\log\sigma+\log\log u))$ time. This means that the LZ-indexes can be built within slightly more space (in some cases the same) than that required by the compressed text.

We implement and test in practice a simplification of our algorithm, and demonstrate that in many practical scenarios the indexing space requirement is also the same as that of the final index. Thus, we conclude that wherever the LZ-index can be used, we can build it. We show how 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, showing that this work can be carried out in a commodity PC. Notice that our algorithm is many times faster than the algorithm for constructing the CSA [30]. 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 symbols are represented by bytes).

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

Table 1. Comparison of different algorithms for constructing text indexes. The reduced LZ-index versions can be constructed within the same space required by the final indexes.

Index	Indexing space (in bits)	Indexing time
Suffix Arrays (SA) [33]	$O(u \log \sigma)$ (*)	$O(u \log \log \sigma)$
SA [21]	$u \log u$	$O(u \log u)$
CSA [32]	$u(H_0(T) + 2 + \epsilon) + o(u \log \sigma)$ (¶)	$O(u \log u)$
CSA [54]	$O(u \log \sigma \log^{\epsilon}_{\sigma} u)$ (†)	$O(u)$
AF-FMI [25]	$uH_k(T) + o(u \log \sigma)$ (§)	$O(u \log u(1 + \frac{\log \sigma}{\log \log u}))$
LZ-index [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 for $\log \sigma = o(\log u)$. (¶) this is $O(u \log \sigma)$ bits of space, in the worst case. (†) for $\epsilon = \log_3 2$. Again, the space is $o(u \log u)$ bits for $\log \sigma = o(\log u)$. (‡) for any $0 < \epsilon < 1$ and $k = o(\log_{\sigma} u)$, applies to all LZ-index variants. (§) for any $k \leqslant \alpha \log_{\sigma} u$ and any constant $0 < \alpha < 1$.

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, we will assume that there is enough secondary memory to hold the index we build.

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 [65] 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 over an alphabet of size σ, denoted by $H_k(T)$ [48]. 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 [68]) is based on a *dictionary of phrases*, in which we add every new *phrase* computed. At the beginning of the compression, the dictionary contains a single phrase b_0 of length 0 (i.e., the empty string). The current step of the compression is as follows: If we assume that a prefix $T[1..j]$ of T has been already compressed into a sequence of phrases $Z = b_1 \dots b_r$, all of them in the dictionary, then we look for the longest prefix of the rest of the text $T[j+1..u]$ which is a phrase of the dictionary. Once we have found this phrase, say b_s of length ℓ_s, we construct a new phrase $b_{r+1} = (s, T[j+\ell_s+1])$, write the pair at the end of the compressed file Z, i.e. $Z = b_1 \dots b_r b_{r+1}$, 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 $0 \leqslant i \leqslant n$.

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 is assumed to be lexicographically larger than any other symbol in the alphabet. We show the phrase identifiers above each corresponding phrase in the parsing. In Fig. 3(a) we show the corresponding *LZTrie*. Inside each *LZTrie* node we show the corresponding phrase identifier.

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

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

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 ([68]). 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 [39] to bound the output of the LZ78 parsing of text T in terms of the k-th order empirical entropy of T.

Lemma 1 ([39]). *It holds that* $n \log n = uH_k(T) + O(u\frac{1+k\log\sigma}{\log_\sigma u})$ *for any* k.

In our work we assume $k = o(\log_\sigma u)$ (and hence $\log\sigma = o(\log u)$ to allow for $k > 0$, i.e. higher order compression); so that $n \log n = uH_k(T) + o(u \log\sigma)$.

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 [60].

There exist a number of practical data structures supporting $rank$ and $select$, like the one by González et al. [24], Kim et al. [38], Okanohara and Sadakane [59], etc. Among these, the first [24] 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 [18] 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. The representation of Golynski et al. [23] requires $n(\log\sigma + o(\log\sigma)) = O(n \log\sigma)$ bits of space [8], allowing us to compute $select_c$ in $O(1)$ time, and $rank_c$ and access to $S[i]$ in $O(\log\log\sigma)$ time.

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 $A[j]$.

The data structure of [46] supports all these operations in $O(\log n)$ worst-case time, and requires $nk' + o(nk')$ bits of space. For us, it is interesting that the space can be made $nk' + O(n)$ bits.

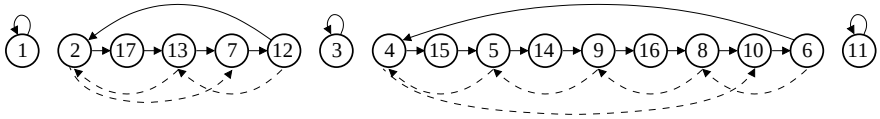
Succinct Representation of Permutations The problem here is to represent a permutation π of $\{1, \ldots, n\}$, such that we can compute both $\pi(i)$ and its inverse $\pi^{-1}(j)$ in constant time and using as little space as possible. A natural representation for π is to store the values $\pi(i)$, $i = 1, \ldots, n$, in an array of $n \log n$ bits. The brute-force solution to the problem computes $\pi^{-1}(j)$ looking for j sequentially in the array representing π. If j is stored at position i, i.e. $\pi(i) = j$, then $\pi^{-1}(j) = i$. Although this solution does not require any extra space to compute π^{-1}, it takes $O(n)$ time in the worst case.

A more efficient solution is based on the *cycle notation of a permutation*. The cycle for the i-th element of π is formed by elements i, $\pi(i)$, $\pi(\pi(i))$, and so on until i is found again. Notice that every element occurs in one and only one cycle of π. For example, the cycle notation for permutation ids 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 $ids^{-1}(13)$ in our example, we start at position 13, then move to position $ids(13) = 7$, then to position $ids(7) = 12$, then to $ids(12) = 2$, then to $ids(2) = 17$, and as $ids(17) = 13$ we conclude that $ids^{-1}(13) = 17$. 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
$ids[i]$	1	17	3	15	14	4	12	10	16	6	11	2	7	9	5	8	13

(a) An example of permutation ids.



(b) Cycle notation of permutation ids.

Fig. 2. Cycle representation for a given permutation ids. Each solid arrow $i \rightarrow j$ in the diagram means $ids(i) = j$. 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 as soon as possible. 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 to find the backward pointer corresponding to a given element (see [52] for details). Overall, this solution requires $(1 + \epsilon)n \log n + n + o(n)$ bits.

Next we present a result which 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 2. *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., "$i \rightarrow j$" 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 tree with n nodes, there exist a number of succinct representations requiring $2n + o(n)$ bits, which is close to the information-theoretic lower bound of at least $2n - \Theta(\log n)$ 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] and the one by Navarro [55, Section 6.1]. The latter has shown to be very effective for representing LZ-indexes [56].

DFUDS **Tree Representation** To get this representation [9] 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 [35]), and $ancestor(x, y)$ (which tell 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 be also computed in constant time on DFUDS by using the approach of Jansson et al. [35], 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., trees where each node has at most σ children, each child labeled by a symbol in the set $\{1, \ldots, \sigma\}$) we use the DFUDS sequence par plus an array $letts[1..n]$ storing the edge labels according to a DFUDS traversal of the tree: we traverse the tree in depth-first preorder, and every time we reach a node x we write the symbols labeling the children of x. In this way,

the labels of the children of a given node are all stored contiguously in $letts$, which will allow us to compute operation $child(x, \alpha)$ (which gets the child of node x with label $\alpha \in \{1, \ldots, \sigma\}$) efficiently. In Fig. 3(c) we show the DFUDS representation of *LZTrie* for our running example.

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 [18, 23]. 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 to the original one [9], 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 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. 3(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 succinct representation of *LZTrie*, essentially using the parentheses representation of Munro and Raman [53], yet newer versions of the LZ-index [6] use the DFUDS representation [9]. Arrays ids and $letts$ are also created at this stage.

$Node$. Once the *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 $n \log 2n$ 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 and represent the trie topology of *RevTrie* and the phrase identifiers in array *rids*. Empty unary nodes are removed only at this step, and so the final number of nodes in *RevTrie* is $n \leqslant n' \leqslant 2n$.

Notice that if we use $n' \log n$ bits for the *rids* array, then in the worst case *RevTrie* requires $2uH_k(T) + o(u \log \sigma)$ bits of storage, and the whole index requires $5uH_k(T) + o(u \log \sigma)$ bits. 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 $rank(j)$. Using this representation, *RevTrie* requires $uH_k(T) + o(u \log \sigma)$ bits of storage. This was unclear in the original LZ-index paper [55, 56].

Range. For $Range$, the data structure of Chazelle [12] permits two-dimensional range searching in a grid of n pairs of integers in the range $[0..n] \times [0..n]$. This data structure can be represented with $n \log n + O(n \log \log n)$ bits of space [45] as follows. We assume first that the points are obtained from pairing two permutations of $\{1, \ldots, n\}$, i.e., there is exactly one point with first coordinate i for any $0 \leqslant i \leqslant n$, and one point with second coordinate j for any $0 \leqslant j \leqslant n$.

To construct *Range*, we sort the set by the second coordinate j, and then we 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 [45] for more details). Thus, we only need $O(\log n)$ pointers to represent the levels of the tree, avoiding in this way to store the pointers that represent the balanced tree.

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 [45]. Since in our case n is the number of LZ78 phrases of text T, the $O(n \log \log n)$ term in the space requirement of the data structure is just $o(u \log \sigma)$ bits, and thus the space requirement is $uH_k(T) + o(u \log \sigma)$ bits.

$RNode$. In the practical implementation of 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 that $Node$ 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, bust just average-case guarantees. However, this approach has shown to be effective in practice since it has a good average-case search time [56].

Overall Space Requirement. Using those succinct representations, each of the four structures that conform LZ-index requires $n \log n + o(u \log \sigma)$ bits of space, which according to Lemma 1 is $uH_k(T) + o(u \log \sigma)$ bits, for $k = o(\log_\sigma u)$. Hence, the final size of the LZ-index is $4uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$. The LZ-index can be built in $O(u \log \sigma)$ time [55].

3.3 Experimental Indexing Space

A large amount of storage is needed to construct the LZ-index [56], mainly because of the pointer representation of the tries used at construction time. In the experiments of the original LZ-index [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 [56].

On the other hand, the indexing space for the pointer-based *reverse* trie is, in some cases, 4 times the text size. This is, mainly, because of the empty unary nodes. This space dictates the maximum indexing space of the algorithm. The overall indexing space was 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 [49]

3.4 Reduced 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 the search performance of the original LZ-index. One of the approaches introduced to reduce the space is the so-called *navigational-scheme* approach, which consists in regarding the original LZ-index (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).

Thus, the original LZ-index has the scheme: $Node$: phrase identifier $\mapsto$ *LZTrie* node; $RNode$: phrase identifier $\mapsto$ *RevTrie* node; ids : *LZTrie* preorder $\mapsto$ phrase identifier; and $rids$: *RevTrie* preorder $\mapsto$ phraseidentifier. 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.

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

Scheme 2 The so-called Scheme 2 of LZ-index [5] is as follows: ids : *LZTrie* preorder $\mapsto$ phrase identifier; $rids^{-1}$: phrase identifier $\mapsto$ *RevTrie* preorder; and R : *RevTrie* preorder $\mapsto$ *LZTrie* preorder. Thus, the space requirement is $3uH_k(T) + o(u \log \sigma)$ bits of space. Though 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, called Scheme 1. However, Scheme 2 outperforms it in most practical cases [5], and thus we disregard Scheme 1 in this paper.

Scheme 3 This LZ-index variant has the following scheme: 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 space requirement is $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, since arrays ids and $rids$ are represented with the data structure for permutations of Munro et al. [52]. 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 outperforms Scheme 3 in many cases).

Scheme 4 This scheme of LZ-index 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 space requirement is also $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, since the inverse permutations are represented by the data structure of [52]. Though 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.

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). This function works as a suffix link in *RevTrie* [6]: 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 [6]. 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 [28, 63], thus this can be also compressed to $O(n \log \sigma) = o(u \log \sigma)$ bits of space. 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* [66] in *RevTrie*. The space requirement is $\epsilon n \log n + o(u \log \sigma)$ bits. Thus we have:

Lemma 3 ([6, 7]). *There exists a Lempel-Ziv compressed full-text self-index requiring $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$ and any $0 < \epsilon < 1$, which is able to locate (and count) the occ occurrences of a pattern $P[1..m]$ in text $T[1..u]$ in $O(\frac{m^2}{\epsilon} + \frac{u}{\epsilon \sigma^{m/2}})$ average time, which is $O(\frac{m^2}{\epsilon})$ if $m \geqslant 2 \log_\sigma u$.*

Thus the LZ-index can be represented with almost optimal space (under the model of the empirical entropy $H_k(T)$), 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 the *Range* data structure, the two-dimensional range search data structure as defined for the original LZ-index. This requires $n \log n + o(u \log \sigma)$ extra bits of space, and thus we get:

Lemma 4 ([6, 7]). *There exists a Lempel-Ziv compressed full-text self-index requiring $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ 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 text $T[1..u]$ in $O(\frac{m^2}{\epsilon} + (m + occ) \log u)$ worst-case time; count the number of pattern occurrences in time $O(\frac{m^2}{\epsilon} + m \log u + occ)$; and determine whether pattern P exists in P in $O(\frac{m^2}{\epsilon} + m \log u)$ time.*

Finally, we add the *Alphabet-Friendly FM-index* [18] of text T to this index, to get:

Lemma 5 ([7]). *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 text $T[1..u]$ in $O((m + \frac{occ}{\epsilon}) \log u)$ worst-case time; count the number of pattern occurrences in $O(m)$ time; and determine whether pattern P exists in P in $O(m)$ time.

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.

The space-efficient construction algorithm for LZ-index presented in [4] has a construction time of the form $O(\sigma u)$. This makes the construction algorithm impractical for moderately-large alphabets. In the sequel we shall achieve $O(u(\log \sigma + \log \log u))$ time by using an improved dynamic representation.

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 fot dynamic data structures, using a standard model [61] of memory allocation. In Section 4.7, we shall adapt our algorithm to the cases in which there is no 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 and 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 is pointing 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.

To summarize, every such 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), and also that we are enforcing that every node is stored in the same block of its children, which also means that sibling nodes are all stored in the same block.

Rather than using a static representation for the trie blocks [4], which are rebuilt from scratch upon insertions, this time 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 in [3] 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 the *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 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 short enough pointers within blocks.

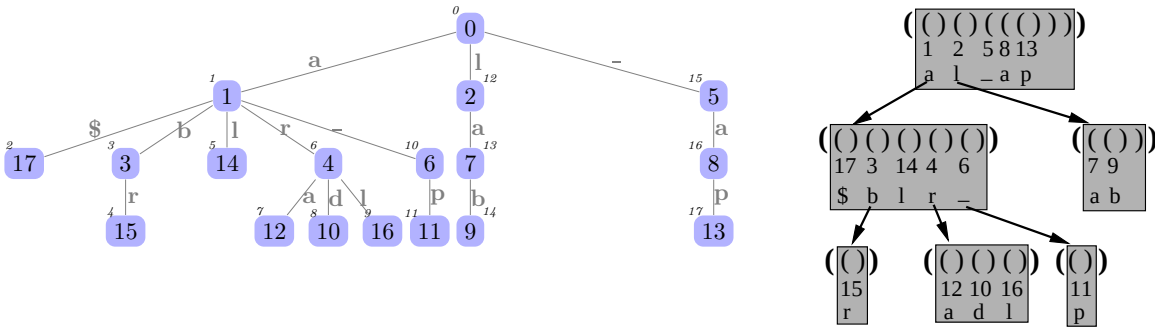
Defining the Block Layout Each block p of N nodes consists of:

- The representation T_p of the topology of the block, using any suitable tree representation.
- 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 associated an 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. 3(b) we show an example of hierarchical representation of *LZTrie* for the running example text, assuming that BP is used to represent the trie topology of each block. 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.



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

(b) Hierarchical representation of *LZTrie*, using BP inside the blocks. Duplicate nodes are shown outside the blocks.

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

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 [11] to represent the DFUDS [9] 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, which in our case also means $O(\log N)$ nodes per segment (by identifying each node with its first parenthesis). 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 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 [46], 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 [11]. Deletions of leaves are handled in a similar way. The space requirement is $O(N)$ bits per block, which adds up to $O(n) = o(u)$ bits overall[2].

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 [46]. This data structure 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 $selectnode_p(j)$.

[2] The space requirement of the trie topology can be reduced to $2n + o(n)$ bits overall, see [3]. However, $O(n)$ bits is sufficient for our purposes.

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 $n + o(n) = o(u)$ extra bits to our representation. Arroyuelo [3] gives a more involved representation for F_p, requiring $o(n)$ bits, yet the one we are using here is simpler yet 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 [46], which supports all the operations (including insertions and deletions) in $O(\log N)$ time, and requiring $Nk' + O(N) = N \log \sigma + O(N)$ bits of space, adding overall $n \log \sigma + O(n) = o(u \log \sigma)$ bits of space.

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 *linked 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 linked-list functionality is easily achieved by simplifying, for example, a dynamic partial sums data structure [46], so that only accesses and insertions are permitted. For a list of N elements, this data structure is a balanced tree storing circular arrays of $\Theta(\log N)$ list elements at the leaves, and subtree sizes at internal nodes. It carries out all the operations in $O(\log N)$ time and poses an extra space overhead of $O(N)$ bits.

We need $N \log u + O(N)$ bits of space to maintain the identifiers, which adds up to $n \log u + O(n)$ bits overall. This is $uH_k(T) + o(u \log \sigma)$ bits of space according to Lemma 1. Recall that $N = O(N_M)$ in our case, and therefore the time to manipulate the list is $O(\log \sigma + \log \log u)$ per operation.

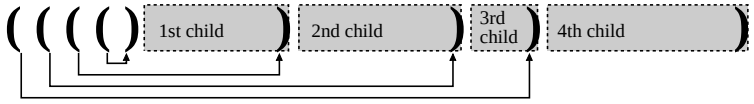
Representation of the Inter-Block Pointers, ptr_p For the inter-block pointers, we use also a linked list L_{ptr_p}, managed in a similar way as for L_{ids_p}. Since blocks have at least N_m nodes, we have a pointer out of (at least) $\Theta(\log^2 u)$ nodes, which adds $O(n/\log n) = o(u/\log u)$ bits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

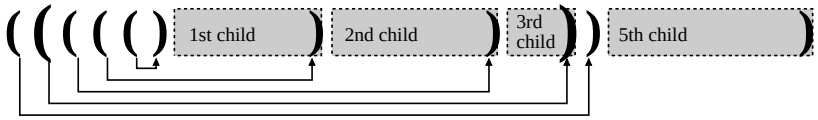
The insertion of a new node for the LZ78 phrase B_t in the trie implies to update only the block p in which the insertion is carried out. Assume that the new leaf must become the j-th node (in preorder) within the block p, and that the new leaf is a new child of node x in block p (i.e., node x represents 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 [11]. See Fig. 4 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 takes $O(\log N_M) = O(\log \sigma + \log \log u)$ time.



(a) A node x of degree 4 and its corresponding subtree in the DFUDS representation of the *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. 4. Illustration of the insertion of a new leaf node in the DFUDS representation of *LZTrie*.

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.

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 opening parenthesis 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, which can be performed in two different ways: upon a block overflow, we traverse block p to select node z, which takes $O(N_M)$ time in the worst case, or looking for z in advance to overflows, as we perform the insertion of new nodes (using the insertion path to look for possible candidates). We choose the latter option, since in this way we can obtain a good amortized cost for updates, as we will see later in our analysis.

To quickly select node z, we maintain in each block p a *candidate list* C_p [3], 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 [46]. 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 finding 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 6. *For every candidate node x in block p, it holds that $subtreesize_p(x) < \sigma N_m$.*

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 been already paid by some node at insertion time.[3] Thus, the insertion cost is $O(\log N_M)$ amortized. After n insertions, the overall cost amortizes to $O(n \log N_M) = O(n(\log \sigma + \log \log u))$.

Once we solved the overflow, the insertion of the new node is carried out either in p' or in p, depending 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 $O(n) + (n + o(n)) + (n \log \sigma + O(n)) + (n \log u + O(n)) + o(n/\log n) + o(n/\log n)$ bits of storage to represent the trie topology, flags, symbols, identifiers, inter-block pointers, and candidate lists, respectively. Because of Lemma 1, the space requirement is $uH_k(T) + o(u \log \sigma)$ bits, for any $k = o(\log_\sigma u)$.

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 cost of rebuilding blocks after an insertion is $O(\log N_M)$ amortized, and therefore the total cost amortizes to $O(n(\log \sigma + \log \log u)) = o(u(\log \sigma + \log \log u))$. Therefore, the total construction time is $O(u(\log \sigma + \log \log u))$.

Representing the Final *LZTrie* Once we construct the same hierarchical representation for *LZTrie*, we delete the text since this is not anymore necessary, and then use the hierarchical *LZTrie* to build the final version of *LZTrie* in $O(n(\log \sigma + \log \log u))$ time. We perform a preorder traversal on the hierarchical tree,

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

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. We also allocate $n \log \sigma = o(u \log \sigma)$ bits of space for the final array $letts$, and $n \log n$ bits for array ids.

Thus, the maximum amount of space used is $2uH_k(T) + o(u \log \sigma)$, since at some point we store both the hierarchical and final versions of *LZTrie*. We then free the hierarchical *LZTrie*, thus we end up with a representation requiring $uH_k(T) + o(u \log \sigma)$ bits. Thus, we have proved:

Lemma 7. *There exists an algorithm to construct the LZTrie for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, in $O(u(\log \sigma + \log \log u))$ time and using $2uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$.*

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* [50], 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 block identifiers $rids$ stored by the final *RevTrie*. Each pointer uses $\log 2n$ bits, since the *LZTrie* parentheses representation has $2n$ positions (either in BP or DFUDS representations, recall that *LZTrie* is already in final static form). We store these pointers to *LZTrie* in the same way as for array ids_p in Section 4.1, in preorder according to *RevTrie* and spending $O(1)$ extra bits per element for the linked 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 maximum 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 data dynamic 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 versions of LZ-index, yet introducing some additional problems in our representation, as we shall see below.

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, using the same partial sum approach as for *LZTrie*. We store the Patricia-tree skips of every trie node in a linked list $skips_p$, in preorder and using a linked list as for ids_p in *LZTrie*, using $\log \log u$ bits per node. To enforce this limit, we insert *empty* unary nodes when the skip exceeds $\log u$. Hence, one out of $\log u$ empty unary nodes could be explicitly represented. In the worst case there are $O(u)$ empty unary nodes, of which $O(\frac{u}{\log u})$ can be explicitly represented. This means $O(\frac{u}{\log u}(O(1) + \log \log u + \log \sigma)) = o(u \log \sigma)$ extra bits overall in the hierarchical representation (this is for the space of T_p, F_p, B_p, $skips_p$, and $rletts_p$, plus their overheads). Since we use a linked list for $skips_p$, it takes $O(\log N_M)$ time to find the skip corresponding to a given node.

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.

When searching for a given string s in *RevTrie*, we descend in the trie checking only the first symbols stored in the trie edges, using the skips to know which symbol of s to use at each node. When the longest possible prefix of string s is thus consumed, say upon arriving at node v_r of *RevTrie*, we must compare the

string represented by v_r against s, in order to determine whether the prefix is actually present in *RevTrie* or not. To compute the string corresponding to node v_r we use the connection with the *LZTrie*: we follow the pointer to the corresponding *LZTrie* node, and follow the upward path in *LZTrie* to extract the symbols. However, we are storing some empty nodes in *RevTrie*, for which we do not store pointers to *LZTrie*.

Assume that node v_r in block p is empty, and represents string s'. Since every descendant of v_r has s' as a suffix, if we map to *LZTrie* from any of these descendants we would find string s' also by reading the upward path in *LZTrie* (we know the length of the string we are looking for, so we know when to stop going up in *LZTrie*). Notice that there exists at least one non-empty descendant v'_r of v_r since *RevTrie* leaves cannot be empty (because they always correspond to an LZ78 phrase). So we can use the *LZTrie* pointer of v'_r to find s'. Since we only store pointers for non-empty nodes, the pointer 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, in order to compute string s'.

An important difference with the *LZTrie* construction is that in *RevTrie* we do not necessarily 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 [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 $O(n') + (n' + o(n')) + (n' + o(n')) + (n \log 2n + O(n)) + (n' \log \sigma + O(n')) + (n' \log \log u + O(n')) + o(n'/\log n') + o(n'/\log n')$ bits of storage to represent the trie topology, flags, bit vector of empty nodes, pointers to *LZTrie* stored in the nodes, symbols, skips, pointers (both inter-block and extra *LZTrie* pointers associated to each block), and candidates, respectively. As we compress empty unary paths, $n \leqslant n' \leqslant 2n$ holds, and thus, the space is upper bounded by $n \log n + o(u \log \sigma)$, which according to Lemma 1 is $uH_k(T) + o(u \log \sigma)$ bits of space, for any $k = o(\log_\sigma u)$.

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 extract the string from *LZTrie*, in order to do the final check in the Patricia tree). 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 $O(n' \log N_M) = o(u \log \sigma)$ time, replacing the pointers to *LZTrie* by the corresponding

phrase identifiers ($rids$). This raises the space to $3uH_k(T) + o(u \log \sigma)$ bits. We then free the hierarchical trie, dropping the space to $2uH_k(T) + o(u \log \sigma)$ bits. Thus, we have proved:

Lemma 8. *Given the LZTrie for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, 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 $2uH_k(T) + o(u \log \sigma)$ bits of space on top of the space required by the final LZTrie, for any $k = o(\log_\sigma u)$.*

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 $\{0, \ldots, n\}$.

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

Therefore, we start by computing ids^{-1} on the same space of ids, using the algorithm of Lemma 2, 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 $3uH_k(T) + o(u \log \sigma)$ bits. Next, we recover ids from ids^{-1}, using again Lemma 2.

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 [4]. 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 2 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}. We allocate space for $\log n$ bit vectors of n bits each, requiring $n \log n$ extra bits, thus raising the space usage to $4uH_k(T) + o(u \log \sigma)$ bits. Then, we construct $Range$ just as explained in Section 3.2 and using the points represented by RQ^{-1}. This takes $O(n \log n)$ time, which in the worst case is $O(\frac{u \log u}{\log_\sigma u}) = O(u \log \sigma)$. We then free RQ^{-1}, dropping the space to $3uH_k(T) + o(u \log \sigma)$ bits.

Lemma 9. *Given a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, and given the corresponding $LZTrie$ and $RevTrie$ data structures, there exists an algorithm to construct*

[4] 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 R itself. The importance of this fact shall be made clear later in this section.

the $Range$ data structure requiring a maximum total space of $2uH_k(T) + o(u \log \sigma)$ extra bits on top of the space for $LZTrie$ and $RevTrie$, and takes $O(u \log \sigma)$ time in the worst case.

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

After building *RevTrie*, 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 $4uH_k(T) + o(u \log \sigma)$ bits, which is the final space required by the LZ-index. The process can be carried out in $O(n)$ 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. [6] add a bit vector $TPos$ marking the phrase beginnings, which is then represented with a data structure for $rank$ and $select$ and requiring $o(u \log \sigma)$ bits of space [60], see [7] for details. 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 of space by properly choosing the sample rates. See [5] for details. 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 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. The meaning of the third column in the table shall be made clear later in Section 4.7.

Table 2. Space and time requirements of each step in the whole compressed indexing process. We assume $k = o(\log_\sigma u)$, and that the tree topology of blocks is represented with DFUDS.

Indexing step	Maximum total space	Maximum main-memory space	Indexing time
1	$uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
2	$2uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
3	$2uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
4	$3uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u(\log \sigma + \log \log u))$
5	$4uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u \log \sigma)$
6	$4uH_k(T) + o(u \log \sigma)$	$uH_k(T) + o(u \log \sigma)$	$O(u/\log_\sigma u)$

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 [61]. 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 [61], which is the most restrictive one. Note $w = \log n + o(\log n)$, as we need $\Theta(n \log n)$ bits of space to build our index[5].

We manage the memory of every trie block separately, each in a "contiguous" memory space. However, trie blocks are dynamic as we insert of new nodes, hence the memory space for trie blocks must grow accordingly. If we use an *Extendible Array* (EA) [10] 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; shrink(A), which shrinks the size of array A by one; and access(A, i), which access the i-th item in array A.

Raman and Rao [61] 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 [46].

Thus, we use operation grow on the corresponding EA 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 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 [61]. A similar analysis can 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 $n\log u + O(n) + O(\frac{nw}{N_m} + n\sqrt{\frac{w\log u}{N_m}}) = n\log n + O(n)$ 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 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 $s = n\log u + n\log\sigma + O(n)$ bits (where the $O(n)$ term includes the space for the collections of T_p, F_p, etc., as well as the space overhead due to the EA memory management of these collections). Hence, the total space overhead is $O(w + \sqrt{wn\log u})$ bits, which is $O(\sqrt{n}\log u) = O(\sqrt{n}\log n) = o(n)$ bits.

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

[5] Note this is consistent with our earlier $w = \Theta(\log u)$ assumption for the RAM model, as $\log u = \Theta(\log n)$.

Theorem 1. *There exists an algorithm to construct the LZ-index for a text $T[1..u]$ over an alphabet of size σ and with k-th order empirical entropy $H_k(T)$, using $4uH_k(T) + o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $k = o(\log_\sigma u)$. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

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 $uH_k(T) + o(u \log \sigma)$ bits (i.e., 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. This does not mean that the index is built on secondary storage, but that in certain cases we use the available secondary memory to store an index component which is not currently needed, thus reducing the peak of main memory usage. However, and as we have seen before 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 show the space requirement in two ways: the *maximum amount of main memory* used at that step and the *total amount of memory* used at that step (main-memory plus secondary-memory space). The latter corresponds to the amount of main memory used at every step if we do not use the disk along the construction process.

Step (1) We build the hierarchical $LZTrie$ from the text. We can then erase the text. The total and main-memory space is $uH_k(T) + o(u \log \sigma)$ 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 indexed by 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 $(n \log n)/B$ (sequential) disk accesses. The symbols ($letts$) and the trie topology are maintained in main memory for the next step, requiring $2n + n \log \sigma + o(n \log \sigma) = o(u \log \sigma)$ bits of space.

Thus, the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits, while the maximum total amount of space is $2uH_k(T) + o(u \log \sigma)$ bits, since we store the hierarchical *LZTrie* in main memory and array ids on disk. We then free the hierarchical $LZTrie$, ending up with a representation requiring $o(u \log \sigma)$ bits of main-memory space, and a total of $uH_k(T) + o(u \log \sigma)$ 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. This raises the total space requirement to $2uH_k(T) + o(u \log \sigma)$ bits of space. The maximum main-memory usage is $uH_k(T) + o(u \log \sigma)$ bits of space (recall that array ids is on disk).

Step (4) We build $RevTrie$ from its hierarchical representation as follows. 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 and representing the remaining components of *RevTrie*, the total space is raised to $3uH_k(T) + o(u \log \sigma)$ bits, since we have at the same time the final *LZTrie* (array ids is on disk), the hierarchical *RevTrie* (in main memory), and the final *RevTrie* (pointers to *LZTrie* are on disk). Then, we free the hierarchical *RevTrie*, thus reducing the total and main-memory space.

Then, we proceed to replace the pointers by the corresponding phrase identifiers. We first load array ids to main memory (leaving a copy of it on disk, for further use). Then, 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. Finally, we leave a 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 $uH_k(T) + o(u \log \sigma)$ bits, which corresponds to the space of the hierarchical *RevTrie*, and we end up with a representation requiring $uH_k(T) + o(u \log \sigma)$ bits of main memory, and $3uH_k(T) + o(u \log \sigma)$ bits overall. The number of disk accesses performed is $(4n \log n)/B$.

Step (5) We build $Range$, basically using the procedure of Section 4.3, yet with some changes in the memory management in order to reduce the peak of memory usage. Therefore, we compute ids^{-1} on the same space required by ids, using the algorithm of Lemma 2, requiring $O(n)$ time and n extra bits of space. Then, we traverse *rids* in preorder and for every non-empty node with preorder i we set $RQ[i] \leftarrow ids^{-1}[rids[i]+1]$. 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 $uH_k(T) + o(u \log \sigma)$ bits.

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

After building RQ, to construct $Range$ we must sort the points in RQ by the second coordinate, by means of constructing RQ^{-1}. Thus, we bring RQ to main memory (and delete it on disk), and use the algorithm of Lemma 2 to construct RQ^{-1} on top of the space for RQ, in $O(n)$ time and requiring n extra bits of space on top of RQ^{-1}. To build $Range$ from RQ^{-1}, instead of allocating memory for the $\log n$ bit vectors of n bits each, which would require $n \log n$ extra bits of space on top of RQ^{-1}, we just allocate memory level per level (i.e., we allocate just n bits per level), construct that level from RQ^{-1}, just as explained in Section 3.2, and then we save that level to disk. Thus, the maximum main-memory space requirement to construct $Range$ is $n \log n + o(u \log \sigma) = uH_k(T) + o(u \log \sigma)$ bits of space. The maximum total space is $2n \log n + o(u \log \sigma) = 2uH_k(T) + o(u \log \sigma)$ extra bits on top of the space for $LZTrie$ and *RevTrie*, which means a total space of $4uH_k(T) + o(u \log \sigma)$ bits. The construction process takes $O(n \log n)$ time, which in the worst case is $O(\frac{u \log u}{\log_\sigma u}) = O(u \log \sigma)$. After getting $Range$, we free array RQ^{-1} and we are done in this step with a partial representation of LZ-index requiring $3uH_k(T) + o(u \log \sigma)$ bits. The number of disk accesses is $(4n \log n)/B$.

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 number of disk accesses is $(2n \log n)/B$.

Thus, we need only $uH_k(T) + o(u \log \sigma)$ bits of main-memory space to construct $Node$, and this increases the total space requirement to $4uH_k(T) + o(u \log \sigma)$ bits, which is the final space required by the LZ-index. The process can be carried out in $O(n)$ time. We use the same procedure in case of using the *RNode* data structure instead of $Range$.

In the third column of Table 2 we show the maximum main-memory space requirement at each step. The overall number of disk accesses is $(11n \log n)/B = (11uH_k(T) + o(u \log \sigma))/B$. Thus, we have proved:

Theorem 2. *There exists an algorithm to construct the LZ-index for a text $T[1..u]$ over an alphabet of size σ, using a maximum main-memory space of $H_k(T) + o(u \log \sigma)$ bits and $O(u(\log \sigma + \log \log u))$ time. The algorithm performs $(7uH_k(T) + o(u \log \sigma))/B$ disk accesses, plus those to write the final index. This holds for any $k = o(\log_\sigma u)$. The total space used by the algorithm is $4uH_k(T) + o(u \log \sigma)$ bits. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

5 Space-Efficient Construction of Reduced LZ-indexes

There exist new reduced versions of LZ-index, some of which are able to replace the original LZ-index in many practical scenarios [6, 5]. Henceforth, in this section we show how to adapt our space-efficient algorithm to build these new indexes.

Throughout this section we assume that the final tries are represented with DFUDS, just as in [6, 7]. We also assume the reduced-memory scenario as in Section 4.7. Recall that we 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.

5.1 Space-Efficient Construction of Scheme 2

We perform the following steps to build Scheme 2 of 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 $uH_k(T) + o(u \log \sigma)$ bits.
2. We derive the final *LZTrie* from the hierarchical one, which is then freed. The *LZTrie* stores the trie topology par, the symbols $letts$, and the phrase identifiers ids, requiring $uH_k(T) + o(u \log \sigma)$ extra bits. This takes $O(u(\log \sigma + \log \log u))$ time because of the traversals on the hierarchical *LZTrie*. We use the approach of Section 4.7 to construct ids, without requiring extra asymptotic space. The total space usage is $2uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory usage is $uH_k(T) + o(u \log \sigma)$ bits. The main-memory space after freeing the hierarchical trie is $o(u \log \sigma)$ bits. The resulting number of I/Os is $(uH_k(T) + o(u \log \sigma))/B$, because of the construction of array ids.
3. We build the hierarchical *RevTrie* from the *LZTrie*, as in Section 4.2. This takes $O(u(\log \sigma + \log \log u))$ time. The total space usage is raised to $2uH_k(T) + o(u \log \sigma)$ bits. The maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits.
4. We build the final *RevTrie* from the hierarchical one, storing the trie topology $rpar$, the Patricia-tree $skips$, the symbols $rletts$, and bit vector B marking the empty nodes, requiring $(n' + \frac{u}{\log u})(3 + \log \log u + \log \sigma) = o(u \log \sigma)$ extra bits of space. In order to reduce the indexing space, array $rids^{-1}$ is built later. Array R is built from the pointers to *LZTrie*, replacing them by 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 $(uH_k(T)+o(u \log \sigma))/B$ extra I/Os. The total time is $O(u(\log \sigma + \log \log u))$. We then free the space of the hierarchical *RevTrie*. The maximum total

space is $3uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits. We end up using $o(u \log \sigma)$ bits of main-memory space.

5. To space-efficiently construct array $rids^{-1}$, we first construct $rids$ in the following way: we start by loading array ids to main memory and erasing it from disk. Then, for every non-empty *RevTrie* node with preorder j we store $rids[j] \leftarrow ids[R[j]]$. In this way, arrays $rids$ and R are traversed sequentially, for increasing values of j. Then, 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 $3uH_k(T) + o(u \log \sigma)$ bits. We then store array ids to disk, and free its main-memory space (hence dropping the total space). Finally, we load $rids$ to main memory, and use the procedure of Lemma 2 to construct $rids^{-1}$ on top of $rids$, to finally store $rids^{-1}$ on disk. The overall time is $O(n)$. The maximum total space is $3uH_k(T) + o(u \log \sigma)$ bits, while the maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits. The total number of disk accesses performed by this process is $(6uH_k(T) + o(u \log \sigma))/B$.

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

Theorem 3. *There exists an algorithm to construct the Scheme 2 of the LZ-index for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using a total space of $3uH_k(T) + o(u \log \sigma)$ bits and $O(u(\log \sigma + \log \log u))$ time, for any $k = o(\log_\sigma u)$. The maximum main-memory space used at any time to construct Scheme 2 can be reduced to $uH_k(T)+o(u \log \sigma)$ bits, in such a case performing $(5uH_k(T) + o(u \log \sigma))/B$ disk accesses, plus those 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 LZ-index, we first build the *LZTrie* in $O(u(\log \sigma + \log \log u))$ time, storing par, $letts$, and ids. This requires a maximum of $2uH_k(T) + o(u \log \sigma)$ bits of space, and ends up with a representation requiring $uH_k(T) + o(u \log \sigma)$ bits. The maximum main-memory space is $uH_k(T) + o(u \log \sigma)$ bits, using the same procedure as in Section 4.7, Step (2). This requires $(uH_k(T) + o(u \log \sigma))/B$ disk accesses.

We then construct the hierarchical *RevTrie*, storing pointers to *LZTrie* nodes for connectivity among tries. Thus, the space requirement raises to $2uH_k(T) + o(u \log \sigma)$ bits. We build the final *RevTrie* storing just $rpar$, $skips$, and $rletts$, and discard the pointers to *LZTrie*, temporarily losing the connectivity between tries. We then free the hierarchical *RevTrie*, which drops the space used to $uH_k(T) + o(u \log \sigma)$ bits.

Next we allocate memory space for array $rids[1..n]$, requiring $n \log n$ extra bits. We traverse the *LZTrie* in preorder, and generate every phrase B_t stored in it (assuming that i is the preorder of the corresponding *LZTrie* node). We then look for B_t^r in the *RevTrie*. Recall that at this point we do not have the connectivity between tries, which is generally used to search in the *RevTrie*. However, since this string exists for sure in *RevTrie* (because it exists as an LZ78 phrase in *LZTrie*), we only need to descend in the *RevTrie* using the skips, up to consuming B_t^r. At this point we have arrived at the node for B_t^r, which has preorder j in *RevTrie*, without the need of accessing the *LZTrie* to extract the string. Then we set $rids[j] \leftarrow ids[i]$ (notice the sequential scan on ids, which is brought to main memory by parts). Then, we store array $rids$ on disk, and free its main memory space. This requires $(2uH_k(T) + o(u \log \sigma))/B$ extra disk accesses.

Now, we go on to compute the inverse permutations for ids and $rids$ arrays. We first load ids from disk, performing $(uH_k(T) + o(u \log \sigma))/B$ extra disk acceses, and construct on it the data structure of [52], in order to support the computation of ids^{-1}. This requires $\epsilon n \log n + O(n)$ extra space, for $0 < \epsilon < 1$, and takes $O(n)$ time if we use the following procedure.

Let $A_{ids}[1..n]$ be an auxiliary bit vector, and let $B_{ids}[1..n]$ be a bit vector marking which elements of ids have an associated backward pointer. Both bit vectors 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 $A_{ids}[i] \leftarrow \texttt{1}$. We also mark one out of $1/\epsilon$ elements when following the cycles, by setting to $\texttt{1}$ the appropriate position in B_{ids}. We stop following the current cycle upon arriving to a position j such that $A_{ids}[j] = \texttt{1}$; then, we move sequentially from position j to the next position j' such that $A_{ids}[j'] = \texttt{0}$, and repeat the previous process.

Each element in ids is visited twice in this process (this is similar to the process done in the proof of Lemma 2), 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 operation $rank$ [51]. 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 $B_{ids}[i] = \texttt{1}$, we store a backward pointer to the previously visited position j in the cycle, such that $B_{ids}[j] = \texttt{1}$ (this means that there are $1/\epsilon$ elements between these two positions within the cycle). In other words, we set $Bwd[rank_1(B_{ids}, i)] \leftarrow j$.

This second scan takes also $O(n)$ time, thus the overall process takes $O(n)$ 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.

Then, we store ids and the data structure for ids^{-1} on disk, and free its main-memory space. This yields $((1+\epsilon)uH_k(T) + o(u \log \sigma))/B$ disk accesses. Finally, we build on $rids$ the data structure of [52], to support the efficient computation of $rids^{-1}$, with $((2+\epsilon)uH_k(T) + o(u \log \sigma))/B$ extra disk accesses. Thus, we conclude:

Theorem 4. *There exists an algorithm to construct the Scheme 3 of the LZ-index for a text $T[1..u]$ over an alphabet of size σ, and k-th order empirical entropy $H_k(T)$, using $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The main-memory space used at any time to construct Scheme 3 can be reduced to $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits, in such a case performing $(5uH_k(T) + o(u \log \sigma))/B$ disk accesses, plus those to write the final index. The space and time bounds are valid in the standard model $\mathcal{M}_B$ of memory allocation.*

5.3 Space-Efficient Construction of Index of Lemma 3 and Relatives

To construct the LZ-index of Lemma 3 without (asymptotically) requiring extra space, we will need two passes over the text, and several traversals over the *LZTrie* and *RevTrie* (yet the number of traversals is a constant). This is because we must be careful not to surpass the reduced space requirement of this index, $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits. We carry out the following steps in order:

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 $O(n \log \sigma) = o(u \log \sigma)$ 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 extra $O(n \log \sigma)$ bits of space. Recall that we do not store the phrase identifiers ids. We then free the hierarchical *LZTrie*.
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 $uH_k(T) + o(u \log \sigma)$ 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 just the tree topology $rpar$, the Patricia-tree skips, and the symbols $rletts$, requiring $o(u \log \sigma)$ extra bits of space. The pointers to

LZTrie nodes are not stored, but these were used just to provide the connectivity between tries while constructing *RevTrie*. We then free the hierarchical *RevTrie*. This takes $O(u(\log\sigma + \log\log u))$ time. The maximum space requirement is $uH_k(T) + o(u\log\sigma)$ bits (before freeing the hierarchical *RevTrie*), and we end up with a representation using just $o(u\log\sigma)$ bits.

5. We allocate memory for array $R[1..n]$, of $n\log n$ bits of space, which is constructed as follows. We traverse the *LZTrie* in preorder, and for every phrase B_i, we look for B_i^r in *RevTrie*, which exists for sure and therefore we do not need the connection between tries in order to search. This takes $O(|B_i^r|\log\sigma)$ time. Let v_{lz} be the *LZTrie* node corresponding to B_i. Then we store $R[preorder(v_r)] \leftarrow preorder(v_{lz})$. 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 the *RevTrie*, on the other hand, we have that $\sum_{i=1}^{n}|B_i^r| = u$, and thus the overall time is $O(u\log\sigma)$. We then sample ϵn values of R, as explained in [7].
6. We allocate space for arrays V_W and S_W [7], which are used to compute function φ' in *RevTrie*. This adds $O(n\log\sigma) = o(u\log\sigma)$ extra bits. We traverse the *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. This takes $O(n)$ time overall. Then we preprocess V_W and S_W with data structures to support $rank$ and $select$ on them.
7. We build on R the data structure for inverse permutations of [52], using the same procedure as in Section 5.2, raising the overall space requirement to $(1+\epsilon)uH_k(T) + o(u\log\sigma)$ bits. This takes $O(n)$ time. We then sample ϵn values of R^{-1}, as explained in [7].
8. We reuse the space allocated for array R to build the uncompressed representation of function φ. Just as in Step (5), we do not need the connection between tries in order to navigate the *RevTrie*, and hence we do not need the information of array R. Recall from [6] that φ acts as a suffix link in *RevTrie*, and we only store suffix links for the n non-empty nodes. Henceforth, we traverse again the *LZTrie* in preorder, and generate each phrase $B_i = xa$ in $O(\log\sigma)$ time, for $x \in \Sigma^*$, and $a \in \Sigma$. Then we search for ax^r and x^r in *RevTrie*, obtaining non-empty nodes v_r and v'_r respectively. Thus, we store $\varphi[preorder(v_r)] \leftarrow preorder(v'_r)$, and go on with the next phrase in *LZTrie*.
   Thus, the work for phrase $B_i^r = xa$ takes $O((|ax^r| + |x^r|)\log\sigma) = O(|B_i^r|\log\sigma)$ time, and thus the overall time is $O(\sum_{i=1}^{n}|B_i^r|\log\sigma) = O(u\log\sigma)$.
9. We build the compressed version of φ, requiring only extra $O(n\log\sigma) = o(u\log\sigma)$ bits for the final compressed representation of φ. The representation of φ is as follows, profiting from the fact that φ can be divided into (up to) σ strictly increasing subsequences. Rather than storing $\varphi[i]$, we store the δ-code [16] of the differences $\varphi[i] - \varphi[i-1]$ whenever the i-th string of *RevTrie* (in preorder, i.e., in lexicographic order) starts with the same symbol as that of the $(i-1)$-th string. Otherwise, we store the δ-code of $\varphi[i]$. In order to access $\varphi[i]$ in constant time, absolute values of φ are inserted every $O(\log n)$ bits, which adds $O(n)$ extra bits. See [6, 7] for more details. We then free the uncompressed φ.
   We could alternatively use the approach of [11] to construct φ, which is originally defined to construct function Ψ of Compressed Suffix Arrays [29, 63] in $O(u\log u)$ time and requiring only $O(u\log\sigma)$ bits of space. In the case of constructing $\varphi = R^{-1}(parent_{lz}(R[i]))$, for every *RevTrie* preorder $i = 1,\ldots,n$, with this alternative approach this would take $O(n\log n + \frac{n}{\epsilon}) = O(u\log\sigma + \frac{u\log\sigma}{\epsilon\log u})$ time, for any $0 < \epsilon < 1$, requiring no asymptotic extra space (just the $o(u\log\sigma)$ bits for φ). In our case, however, we have previously allocated space for array R, which we use to construct φ much faster. At the end of this step we drop the space requirement to $\epsilon uH_k(T) + o(u\log\sigma)$ bits.
10. We finally allocate memory for array $ids[1..n]$, 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 $(u\log\sigma)/B$ extra disk accesses in case the text is stored on disk), descending in the *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 $O(n \log \sigma)$ time provided the *LZTrie* is represented with DFUDS. Finally, we can erase the text.

Theorem 5. *There exists an algorithm to construct the LZ-index of Lemma 3 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm performs two passes over text T, thus requiring $(u \log \sigma)/B$ disk accesses 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 4, which only adds the *Range* data structure, which in turn can be constructed with the same procedure used in Section 4.7, Step (5). Since this requires $2uH_k(T) + o(u \log \sigma)$ bits of space to be constructed, we build *Range* before Step (5) of the previous algorithm. Thus we conclude:

Corollary 1. *There exists an algorithm to construct the LZ-index of Lemma 4 for a text $T[1..u]$ over an alphabet of size σ, and with k-th order empirical entropy $H_k(T)$, using $(2+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space and $O(u(\log \sigma + \log\log u))$ time. This holds for any $0 < \epsilon < 1$ and any $k = o(\log_\sigma u)$. The algorithm requires $(u \log \sigma + 2uH_k(T) + o(u \log \sigma))/B$ disk accesses in addition to those 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 5 adds the *Alphabet-Friendly FM-index* [18], which according to [25] 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 5 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 $(u \log \sigma + uH_k(T) + o(u \log \sigma))/B$ disk accesses, in addition to those 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. Times were obtained using 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 [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. 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 by Navarro [56]). 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 this is a very efficient representation for our intermediate tries, achieving 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 [4]. We assume 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[6] 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 [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* [19], 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. 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 ($1/\epsilon = 15$). As it can be seen, the sample rate of the inverse permutations in Scheme 3 does not affect much the indexing speed.

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. This is 0.59 times the space 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 in practice the final *RevTrie* does not. Table 4(a) summarizes.

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

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
	Total	574,548,639	965,869,767	3,549.20
Scheme 3	1	411,928,076	411,928,076	898.40
$1/\epsilon = 1$	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
	Total	574,548,639	1,204,608,375	3,583.56
Scheme 3	1	411,928,076	411,928,076	896.88
$1/\epsilon = 15$	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
	Total	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
	Total	1,677,938,853	2,985,958,274	16,685.28
Scheme 3	1	1,233,336,206	1,233,336,206	2,443.83
$1/\epsilon = 1$	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
	Total	1,677,938,853	3,775,475,122	16,543.63
Scheme 3	1	1,233,336,206	1,233,336,206	2,445.02
$1/\epsilon = 15$	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
	Total	1,677,938,853	2,665,257,752	16,719.02

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 [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) [7], with an indexing rate of about 1.29 MB/sec (see Table 4(b)). This means that our indexing algorithm is 4.60 times slower than the original indexing algorithm (see column "Slowdown" in Table 4(b)), yet we require 5.80 times less memory (see column "Space reduction" in Table 4(b)). The intermediate *LZTrie* required 751,817,455 bytes (extrapolating, this is 3.66 times the size of our hierarchical *LZTrie*, see column "Intermediate *LZTrie*" in Table 4(b)), while the intermediate *RevTrie* required 1,185,969,250 bytes (extrapolating, this is 5.79 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[8], 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*

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

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

(b) Main statistics for the construction of Scheme 2 versus the non-space-efficient original algorithm.

Text	Main-memory peak	Indexing rate (MB/secs)	Slowdown	Space reduction	Intermediate *LZTrie*	Intermediate *RevTrie*
English (500 MB)	1,566 MB ($3.13 \times$ text)	1.29	4.60	5.80	3.66	5.74
Genome (500 MB)	1,275 MB ($2.55 \times$ text)	1.86	9.78	5.10	3.22	5.95
XML	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	9.58	3.39	2.41	3.04

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 and 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.

Table 5 shows the practical results for the best indexing algorithms we know of. The results have been taken from the original papers indicated in the table. As a comparison, W.-K. Hon et al. [31, 30] index 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. Despite the difference in CPU rate of our machine compared to Hon et al.'s, the difference in indexing time suggests us that the LZ-index can be space-efficiently constructed in much less time than CSAs. Hon et al. also construct the FM-index in about 4 extra hours, for a total of about 28 hours. The algorithm of [15], on the other hand, indexes the Human Genome in about 8.52 hours, using secondary storage and just a constant amount of main memory.

Ours is a relevant practical 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 indexing algorithms to construct an index for the Human Genome. For suffix trees, Kurtz estimated the indexing time on his machine, whose CPU is 10 times slower than ours. In case of suffix arrays, we estimate the indexing space according to the space used with other texts; we do not have time estimations for these. In both cases the indexing algorithms are probably faster than our algorithms for the LZ-index (provided they have the given amount of main memory available).

Index	Construction algorithm	Indexing time	Maximum indexing space (RAM)
Suffix tree	[40]	$<$ 9 hours (*)	45.31 GB
Suffix array	[42]	—	27.96 GB
Suffix array	[49]	—	18.64 GB
Suffix array	[15]	8.52 hours	sec. storage
CSA	[30]	24 hours	3.60 GB (¶)
FM-index	[30]	28 hours	3.60 GB
Scheme 2 of LZ-index	This paper	4.63 hours	2.78 GB
Scheme 2 (reduced-memoy model)	This paper	4.63 hours	1.56 GB (†)

(*) On a Sun-UltraSparc 300 MHz, 192 MB of main memory, under Solaris 2. (¶) Hon [30] reported a size of 2.88 GB for the Human Genome, whereas ours is of size 3.11 GB. They use a 1.7 GHz CPU. (†) Just regarding main-memory space.

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 the 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.

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.

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
	Total	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
	Total	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
	Total	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
	Total	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
	Total	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
	Total	1,133,180,292	1,766,181,601	5,508.14

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 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, i.e. $4uH_k(T) + o(u \log \sigma)$ bits, to construct the LZ-index for a text $T[1..u]$ in $O(u \log \sigma)$ time, being σ the alphabet size and $H_k(T)$ the k-th order empirical entropy of T. We also show that all LZ-index variants presented in [7, 5] can be constructed within the same space needed by the final index. 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 the LZ-indexes can be constructed within $uH_k(T) + o(u \log \sigma)$ bits of space, in $O(u(\log \sigma + \log \log u))$ time. This means that the LZ-indexes can be constructed within asymptotically 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 the same amount of memory as needed by the final index. Under the reduced-memory scenario, we have that the LZ-index versions can be constructed requiring 0.40 – 1.05 times the size of the original text, depending on the compressibility of the text. This means about 3.39 – 7.50 times less space as 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 4.60 – 9.58 times slower than the original construction algorithm. In conclusion, our algorithm requires much less memory than the original

one, in exchange for a slower construction algorithm. However, our indexing algorithm is still competitive with existing indexing technologies. For example, we are able to construct the LZ-index for the Human Genome in less than 5 hours, while Dementiev et al. [15] and Hon et al. [32] require 8.5 and 24 hours to construct the suffix array and Compressed Suffix Array for the Human Genome, respectively.

An interesting application of our indexing algorithm is in the construction of the LZ78 parsing of a text T. Grossi and Sadakane [64] define an alternative representation for the LZ78 parsing, which has the nice property of supporting optimal time to access any text substring. The parsing consists basically of the *LZTrie* (the trie topology and 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. [34] 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 means $(u\log\sigma)/B$ extra disk accesses 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 it was explained in Section 4.7. This takes extra $O(u(\log\sigma + \log\log u))$ time, and performs $(uH_k(T) + o(u\log\sigma))/B$ extra disk accesses.
- We then free the hierarchical *LZTrie* and load array ids back to main memory, performing $(uH_k(T) + o(u\log\sigma))/B$ extra disk accesses.
- We compute ids^{-1} in place, using the algorithm of Lemma 2, and this way we complete the representation for the LZ78 parsing of text T.

As seen, we exchange the $(u\log\sigma)/B$ extra disk accesses of [34] by $(2uH_k(T) + o(u\log\sigma))/B$. This can be much better, specifically 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 $uH_k(T) + o(u\log\sigma)$ bits.

We think that our methods could be extended to build related LZ-indexes [17, 62] within limited space.

References

1. M. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In *Proc. 9th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2476, pages 31–43, 2002.
2. A Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.
3. 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.
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. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/smallerlzpract.ps.gz`. 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. Technical Report TR/DCC-2008-2, Dept. of Computer Science, University of Chile, 2008. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/smallerlzindex.ps.gz`. Submitted.
8. J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.
9. 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.

10. 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.
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. Elias. Universal codeword sets and representation of integers. *IEEE Trans. on Information Theory*, 21(2):194–203, 1975.
17. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.
18. 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.
19. P. Ferragina and G Navarro. Pizza&Chili Corpus — Compressed indexes and their testbeds, 2005. `http://pizzachili.dcc.uchile.cl`.
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. A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.
24. 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.
25. R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 374–386, 2008.
26. 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.
27. R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.
28. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.
29. 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.
30. W.-K. Hon. *On the construction and application of Compressed text indexes*. PhD thesis, University of Hong Kong, 2004.
31. 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.
32. 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.
33. W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 251–260, 2003.
34. 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.
35. 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.
36. J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 937, pages 191–204, 1995.
37. 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.
38. 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.
39. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

40. S. Kurtz. Reducing the space requeriments of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.
41. T. W. Lam, K. Sadakane, W. K. Sung, and S. M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics (COCOON)*, pages 401–410, 2002.
42. J. Larsson and K. Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.
43. V. Mäkinen. Compact suffix array - a space-efficient full-text index. *Fundamenta Informaticae*, 56(1–2):191–210, 2003.
44. V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic J. of Computing*, 12(1):40–66, 2005.
45. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.
46. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):article 32, 2008.
47. U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.
48. G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.
49. G. Manzini and P. Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.
50. D. R. Morrison. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.
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)*, 2008. To appear. Also as Technical Report TR/DCC-2003-0, Dept. of Computer Science, University of Chile. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/jlzindex.ps.gz`.
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. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.
60. 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.
61. 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.
62. L. Russo and A. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 5(3):501–513, 2007.
63. K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *J. of Algorithms*, 48(2):294–313, 2003.
64. 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.
65. J. S. Vitter. *Algorithms and Data Structures for External Memory*. Series on Foundations and Trends in Theoretical Computer Science, now Publishers, 2008.
66. P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1–11, 1973.
67. I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, second edition, 1999.
68. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.