Practical Approaches to Reduce the Space Requirement of Lempel-Ziv-Based Compressed Text Indices

DIEGO ARROYUELO
Yahoo! Research Chile
and
GONZALO NAVARRO
University of Chile

Given a text $T[1..n]$ over an alphabet of size σ, the *full-text search* problem consists in locating the *occ* occurrences of a given pattern $P[1..m]$ in T. *Compressed full-text self-indices* are space-efficient representations of the text that provide direct access to and indexed search on it.

The LZ-index of Navarro is a compressed full-text self-index based on the LZ78 compression algorithm. This index requires about 5 times the size of the compressed text (in theory, $4nH_k(T) + o(n \log \sigma)$ bits of space, where $H_k(T)$ is the k-th order empirical entropy of T). In practice the average locating complexity of the LZ-index is $O(\sigma m \log_\sigma n + occ\, \sigma^{m/2})$, where *occ* is the number of occurrences of P. It can extract text substrings of length ℓ in $O(\ell)$ time. This index outperforms competing schemes both to locate short patterns and to extract text snippets. However, the LZ-index can be up to 4 times larger than the smallest existing indices (which use $nH_k(T) + o(n \log \sigma)$ bits in theory), and it does not offer space/time tuning options. This limits its applicability.

In this paper we study practical ways to reduce the space of the LZ-index. We obtain new LZ-index variants that require $2(1+\epsilon)nH_k(T) + o(n \log \sigma)$ bits of space, for any $0 < \epsilon < 1$. They have an average locating time of $O(\frac{1}{\epsilon}(m \log n + occ\, \sigma^{m/2}))$, while extracting takes $O(\ell)$ time.

We perform extensive experimentation and conclude that our schemes are able to reduce the space of the original LZ-index by a factor of 2/3, that is, around 3 times the compressed text size. Our schemes are able to extract about 1–2 megabytes of the text per second, being twice as fast as the most competitive alternatives. Pattern occurrences are located at a rate of up to 1–4 million per second. This constitutes the best space/time trade-off when indices are allowed to use 4 times the size of the compressed text or more.

Categories and Subject Descriptors: E.2 [**Data storage representations**]: ; E.4 [**Coding and information theory**]: Data compaction and compression; H.2.4 [**Systems**]: Textual databases

General Terms: Algorithms, experimentation

Additional Key Words and Phrases: Lempel-Ziv compression, Compressed data structures, Indexed text search

Second author supported in part by Fondecyt Grant 1-080019.
Author's address: Diego Arroyuelo, Yahoo! Research Chile. Blanco Encalada 2120, Santiago, Chile. darroyue@dcc.uchile.cl. Gonzalo Navarro, Department of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl

1. INTRODUCTION

Many modern applications require searching, mining, and analyzing sequences of diverse kinds. The basic problem of finding the occurrences of string patterns in a text is fundamental both as a kernel routine of more complex processing and for final users of text retrieval systems.

Unlike *word-based* text searching, which is the typical scenario in Information Retrieval, we wish to find any *text substring*, not only whole words or phrases. This has applications in texts where the concept of *word* is not clearly defined (e.g., Oriental languages, program code, etc.), or texts where words do not exist at all (e.g., DNA, protein, MIDI pitch sequences, etc.). It also offers more flexible searching on natural language text.

Given a sequence of symbols $T[1..n]$ (the text) over an alphabet $\Sigma = \{1, \ldots, \sigma\}$, and given another (short) sequence $P[1..m]$ (the *search pattern*) over Σ, the *full-text search problem* consists in finding all the *occ* occurrences of P in T. There exist three typical kinds of queries, which arise in different types of applications:

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

—*Cardinality queries*: Operation `count`(P) counts the number of occurrences of pattern P in T.

—*Locating queries*: Operation `locate`(P) reports the starting position of the *occ* occurrences of pattern P in T.

While locating all the occurrences can be useful for further processing, end users rarely wish to see more than a few occurrences at once. *Partial* `locate` queries find a fixed number K of occurrences. Different applications might have different requirements on which occurrences are preferred. In this paper we consider the simplest case of retrieving K *arbitrary* occurrences. An example where this turns out to be natural is on Web search engines, which display pointers to the pages found plus a short context where the pattern occurs within each such page. This context is displayed in order to help users decide whether a page is relevant or not without actually retrieving the page. In this example we must be able to quickly find just one arbitrary pattern occurrence in the text and display its context.

We assume that the text is large and known in advance to queries, and we need to perform several queries on it. Therefore, we can preprocess the text to construct an *index* on it, which is a data structure allowing efficient access to the pattern occurrences, yet increasing the space requirement.

Though classical full-text indices, like *suffix trees* [Apostolico 1985] and *suffix arrays* [Manber and Myers 1993], are very efficient at search time, they have the problem of a high space requirement: they require $O(n \log n)$ and $n \log n$ bits respectively, which in practice is about 10–20 and 4 times the text size respectively, apart from the text itself. Thus, we can have large texts which fit into main memory, but whose corresponding suffix tree (or array) does not. Using secondary storage for the indices is several orders of magnitude slower, so one looks for ways to reduce their size, aiming to maintain the indices of relatively large texts *entirely in main memory.* The modern trend is to use the compressibility of the text to reduce the space of the index, focusing on techniques to represent the text and the index using little space, yet permitting efficient text searching [Navarro and Mäkinen 2007].

1.1 Compressed Full-Text Self-Indexing

The track of *compressed full-text self-indices* started a decade ago [Grossi and Vitter 2000; Ferragina and Manzini 2000; Sadakane 2000]. A *full-text self-index* allows one to retrieve any part of the text without storing the text itself, and in addition provides search capabilities on the text. A *compressed* full-text self-index is one whose space requirement is proportional to the compressed text size, for example $O(nH_k(T))$ bits, where $H_k(T)$ denotes the k-th order empirical entropy of T [Manzini 2001], a standard measure of compressibility (see Section 2.1 for more details). Then a compressed full-text self-index *replaces* the text with a more space-efficient representation of it, which at the same time provides indexed access to the text [Navarro and Mäkinen 2007; Ferragina and Manzini 2005]. Thanks to the advances in this technology, it is quite common nowadays that the indices of large texts can be accommodated in the main memory of a desktop computer (e.g., an index for the Human Genome, which has about 3 billion bases, fits comfortably in a 1 GB desktop PC).

As compressed full-text self-indices replace the text, we are also interested in the following operations:

—`display`(P, ℓ), which displays a context of ℓ symbols sorrounding the *occ* occurrences of pattern P in T, and

—`extract`(i, j), which decompresses the substring $T[i..j]$, for any text positions $i \leqslant j$.

Being able to efficiently `extract` arbitrary text substrings is one of the most basic and important problems these indices must solve efficiently, since the text is not available otherwise.

Similarly, although `locate` queries are important in classical full-text indexing (since we have the text at hand to access the occurrences and their contexts), they are usually not enough for compressed self-indices, since they give just text positions. In many applications, such as showing snippets of Web pages, the context surrounding an occurrence is more important than the occurrence position itself. Indeed, the widely-used Unix search tool `grep` by default shows the text lines containing the occurrences. Therefore, in our scenario `display` queries are usually more important than `locate` queries. The latter can be interesting in specific cases, for example if one wants to take statistics about the positions of the occurrences for linguistic or data mining applications.

Finally, `count` and `exists` queries have much more specific applications, and they usually form the internal machinery of more complex tasks, such as *approximate pattern matching* and *text categorization* (where a document is assigned a class or category depending on the frequency of appearance of given keywords). Some *pattern discovery* tasks may use the frequency of certain strings to decide that they are important patterns. Another example is *selective dissemination of information*, where user profiles are formed by keywords of interest and the system considers the presence or absence of those keywords to send or not the document to the user.

Table I. Practical average complexities achieved by the main families of compressed self-indexes and our contribution.

Index	Space in bits	Extraction time for ℓ symbols	Search time (related to m)	Locating time per occurrence
CSA	$nH_0(T) + O(n \log\log\sigma)$	$O(\ell + \log^{1+\epsilon} n)$	$O(m \log n)$	$O(\log^{1+\epsilon} n)$
FM-index	$nH_k(T) + o(n \log\sigma)$	$O(\ell \log\sigma + \log^{1+\epsilon} n)$	$O(m \log\sigma)$	$O(\log^{1+\epsilon} n)$
LZ-index	$4nH_k(T) + o(n \log\sigma)$	$O(\ell)$	$O(\sigma m \log_\sigma n)$	$O(\sigma^{m/2})$
Our LZ-index	$2(1+\epsilon)nH_k(T) + o(n \log\sigma)$	$O(\ell)$	$O(\frac{1}{\epsilon} m \log n)$	$O(\frac{1}{\epsilon}\sigma^{m/2})$

1.2 Families of Compressed Full-Text Self-Indices

The main families of compressed self-indices [Navarro and Mäkinen 2007] are *Compressed Suffix Arrays (CSA)* [Grossi and Vitter 2005; Sadakane 2003; Grossi et al. 2003], indices based on *backward search* [Ferragina and Manzini 2005; Mäkinen and Navarro 2005; Ferragina et al. 2007] (which are alternative ways to compress suffix arrays, and known as the *FM-index* family), and indices based on *Lempel-Ziv* compression [Ziv and Lempel 1977; 1978] (LZ-indices for short) [Kärkkäinen and Ukkonen 1996; Navarro 2004; Ferragina and Manzini 2005; Arroyuelo et al. 2006; Russo and Oliveira 2008]. Table I summarizes the average complexities of their best practical implementations.

In this paper we are interested in LZ-indices since they proved to be effective in practice, outperforming the other families of compressed indices [Navarro 2004; 2009], for the tasks we consider most important: extracting text, displaying occurrence contexts, and locating the occurrences. They are weaker for counting and for searching for long patterns, instead they are fast on short patterns, as these usually have many occurrences to locate.

We focus on the LZ-index of Navarro [2004; 2009], a compressed full-text self-index based on the LZ78 parsing of the text [Ziv and Lempel 1978] (see Section 2.2 for details). The LZ-index takes about 5 times the size of the compressed text (in theory, $4nH_k(T) + o(n \log\sigma)$ bits, for any $k = o(\log_\sigma n)$ [Kosaraju and Manzini 1999; Ferragina and Manzini 2005]), and locates the *occ* occurrences of a pattern in average time $O(\sigma m \log_\sigma n + occ\, \sigma^{m/2})$ in practice. The index also replaces the text (i.e., it is a *self-index*): it can display a text context of length ℓ around an occurrence found (and in fact any sequence of LZ78 phrases) in $O(\ell)$ time, or obtain the whole text in time $O(n)$. The index is built in $O(n)$ time.

However, the space requirement of the LZ-index is relatively large compared with competing schemes: 1.2–1.6 times the text size versus 0.6–0.7 times the text size achieved by the *CSA* [Sadakane 2003], and 0.3–0.8 times the text size reached by the *FM-index* [Ferragina et al. 2007]. In addition, the LZ-index does not offer space/time trade-offs, which limits its applicability.

1.3 Our Contribution

In this paper we study how to reduce the space requirement of Navarro's LZ-index. We reduce the redundancy among its different data structures, while retaining fast locating and text extraction. We also provide space/time trade-offs. Our developments are, in some extent, based on the theoretical ideas of Arroyuelo et al.

Table II. Rough spaces and times achieved by the main compressed self-indexes and ours. Our ranges include the most typical values obtained, disregarding some extremes. Locating times are obtained from the data for $m = 5$. For more precise data see Section 6.

Index	Space times $nHk(T)$	Million symbols extracted per second	Million occurrences located per second	Microseconds to locate first occ.
CSA	2–6	<0.5	0.2–1.0	40–70
FM-index	2–5	0.8–1.0	0.2–0.5	15–30
LZ-index	5	1.5–2.0	2–4	10–20
Our LZ-index	3–5	1.0–2.0	1–4	5–20

[2006], yet in this paper we consider more practical solutions and perform extensive experimentation. We remark that all the compressed full-text indexes considered in this paper operate in main memory.

We define several reduced-space alternatives requiring $3nH_k(T) + o(n \log \sigma)$ bits of space. We also add a space/time tuning parameter to the index, achieving $2(1 + \epsilon)nH_k(T) + o(n \log \sigma)$ bits of space, for $0 < \epsilon < 1$. Our indices do not provide worst-case guarantees at search time, yet they support `locate` queries in $O(\frac{1}{\epsilon}(m \log n + occ\, \sigma^{m/2}))$ average time, and extract text substrings in $O(\ell)$ time. Table I puts these results in context, highlighting the fact that LZ-indexes are larger, but faster at extracting/displaying text, and at locating for short patterns.

We implement and test our indices in several real-life scenarios, and conclude that the space requirement of the original LZ-index can be reduced up to about 2/3, i.e., to about 3 times the size of the compressed text (while replacing it). The $o(n \log \sigma)$ space redundancy turns out to add 25%–40% in practice, depending on σ. For the key operations of `extract` and `display`, our schemes are able to extract about 1–2 MB of text per second in a commodity PC, being about twice as fast as the most competitive space-efficient alternatives. For partial `locate` queries we develop a heuristic to get fast access to the first occurrences, avoiding as much as possible the navigation on the tries that compose the LZ-index; this navigation becomes expensive if we report only a few occurrences. Our indices outperform the alternatives particularly when searching for short patterns (say $m \leqslant 10$), small alphabets (such as DNA data), and when retrieving a few ocurrences ($K \leqslant 5$). For full `locate` queries, for short patterns of length 5, in most scenarios our schemes offer the best space/time trade-offs when the indices are allowed to use 4 times the compressed text size or more. In general, our indices locate up to 1–4 million occurrences per second. The original LZ-index becomes now just an extreme of the trade-off we offer. Table II roughly summarizes actual spaces and times (we emphasize pure locating times by considering the times for short patterns, for longer patterns our times degrade faster than those of the other indices).

We also demonstrate a very important aspect of our indices: compressed indices based on suffix arrays store extra non-compressible information to efficiently carry out the `locate` and `display` tasks, whereas the extra data stored by LZ-indices is largely compressible. Therefore, when the texts are highly compressible, LZ-indices can be smaller and faster than alternative indices; in other cases they offer attractive space/time trade-offs.

2. BASIC CONCEPTS

2.1 Empirical Entropy

Text compression is a technique to represent a text using less space, profiting from the regularities of non-random texts. 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 $H_k(T)$ [Manzini 2001]. The value $nH_k(T)$ provides a lower bound to the number of bits needed to compress T using any compressor that encodes each symbol considering only the context of k symbols that precede it in T. It holds that $0 \leqslant H_k(T) \leqslant H_{k-1}(T) \leqslant \cdots \leqslant H_0(T) \leqslant \log \sigma$ ($\log x$ means $\lceil \log_2 x \rceil$ in this paper). Formally, we have

Definition 2.1. Given a text $T[1..n]$ over an alphabet Σ, the zero-order empirical entropy of T is defined as

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \frac{n}{n_c}$$

where n_c is the number of occurrences of symbol c in T. The sum includes only those symbols c that occur in T, so that $n_c > 0$.

Definition 2.2. Given a text $T[1..n]$ over an alphabet Σ, the k-th order empirical entropy of T is defined as

$$H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{n} H_0(T^s)$$

where T^s is the subsequence of T formed by all the symbols that occur preceded by the context s. Again, we consider only contexts s that do occur in T.

2.2 Ziv-Lempel Compression

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

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

Property 2.3. For all $1 \leqslant t \leqslant d$, there exists $\ell < t$ and $c \in \Sigma$ such that $B_t = B_\ell \cdot c$.

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

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.

we call *LZTrie*. In Fig. 1 we show the LZ78 phrase decomposition for the text $T =$“alabar_a_la_alabarda_para_apalabrarla”, which will be our running example. We show phrase identifiers above the corresponding phrase in the parsing.

In Fig. 4(a) we show the corresponding *LZTrie*. Inside each *LZTrie* node we show the corresponding phrase identifier.

Property 2.4. Every phrase B_i, $0 \leqslant i < d$, represents a different text substring.

This property is used in the LZ-index search algorithm (see Section 3). The only exception to this property is the last phrase B_d. 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.

Definition 2.5. Let $b_r = (r_1, c_1)$, $b_{r_1} = (r_2, c_2)$, $b_{r_2} = (r_3, c_3)$, and so on until $r_k = 0$ be phrases of the LZ78 parsing of T. The sequence of phrase identifiers $r, r_1, r_2, \ldots$ is called the *referencing chain* starting at phrase r.

The referencing chain starting at phrase r reproduces the way phrase b_r is formed from previous phrases and it is obtained by successively moving to the parent in the *LZTrie*. For example, the referencing chain of phrase 9 in Fig. 4(a) is $r = 9$, $r_1 = 7$, $r_2 = 2$, and $r_3 = 0$.

The compression algorithm is $O(n)$ 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). The decompression needs to build the same dictionary (the pair that defines the phrase r is read at the r-th step of the algorithm).

Property 2.6 *[Ziv and Lempel 1978]*. It holds that $\sqrt{n} \leqslant d \leqslant \frac{n}{\log_\sigma n}$. Thus, $d \log n \leqslant n \log \sigma$ always holds.

Lemma 2.7 [Kosaraju and Manzini 1999]. *It holds that* $d \log d \leqslant nH_k(T) + O(n\frac{1+k\log\sigma}{\log_\sigma n})$ *for any* k.

In our work we assume $k = o(\log_\sigma n)$ (and hence $\log \sigma = o(\log n)$ to allow for $k > 0$); therefore, in the worst case $d \log d = nH_k(T) + o(n \log \sigma)$.

2.3 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. In this section and the next we review some results on succinct data structures, which are necessary to follow our work.

2.3.1 *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 [Munro 1996], or even $nH_0(\mathcal{B}) + o(n)$ bits (including $\mathcal{B}$ itself) [Raman et al. 2002].

There exist a number of practical data structures supporting *rank* and *select*, like the one by González et al. [2005], Kim et al. [2005], Okanohara and Sadakane [2007], etc. Among these, the index of González et al. [2005] is very (perhaps the most) efficient in practice to compute *rank*, requiring little space on top of the sequence itself. *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 $O(\log n)$ in the worst case).

Given a sequence $S[1..n]$ over an alphabet Σ, we generalize the above definition for $rank_c(S, i)$ and $select_c(S, i)$ for any $c \in \Sigma$. If $\sigma = O(\text{polylog}(n))$, the solution of Ferragina et al. [2007] allows one to compute both $rank_c$ and $select_c$ in constant time and requiring $nH_0(S) + o(n)$ bits of space. Otherwise the time is $O(\frac{\log \sigma}{\log \log n})$ and the space is $nH_0(S)+o(n \log \sigma)$ bits. The representation of Golynski et al. [2006] requires $n(\log \sigma + o(\log \sigma)) = O(n \log \sigma)$ bits of space [Barbay et al. 2007], allowing us to compute $select_c$ in $O(1)$ time, and $rank_c$ and access to $S[i]$ in $O(\log \log \sigma)$ time.

2.3.2 *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 much 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 the value i is found again. It is important to note that every element occurs in one and only one cycle of π. For example, the cycle notation for permutation *ids* of Fig. 3(a) is shown in Fig. 2. So, to compute $\pi^{-1}(j)$, instead of looking sequentially for j in π, we only need to look for j in its cycle: $\pi^{-1}(j)$ is just the value "pointing" to j in the diagram of Fig. 2. To compute $ids^{-1}(13)$ in the previous 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 $ids^{-1}(13) = 17$. The only problem here is that there are no bounds for the size of a cycle, hence this algorithm takes also $O(n)$ time in the worst case. However, it can be improved for a more efficient computation of $\pi^{-1}(j)$.

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 show backward pointers for $1/\epsilon = 2$. To compute $ids^{-1}(17)$, we first move to $ids(17) = 13$; as 13 has a backward pointer we follow it and hence we move to position 2. Then, as $ids(2) = 17$ we conclude that $ids^{-1}(17) = 2$, in $O(1/\epsilon)$ worst-case time. We store the backward pointers compactly in an array of $\epsilon n \log n$ bits. We mark the

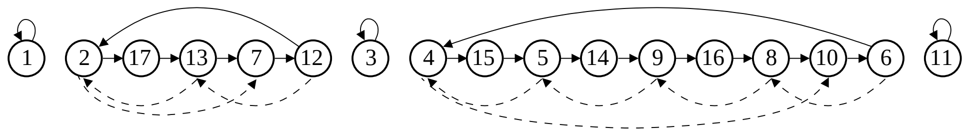


Fig. 2. Cycle representation of permutation *ids* of Fig. 3(a). Each solid arrow $i \rightarrow j$ in the diagram means $ids(i) = j$. Dashed arrows represent backward pointers.

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 the article by Munro et al. [2003] for details). Overall, this solution requires $(1+\epsilon)n\log n + n + o(n)$ bits of storage. In a practical implementation of bit vectors with *rank* and *select* we use, the $n + o(n)$ terms in the space become $1.375n$ bits [González et al. 2005].

2.4 Succinct Representation of Trees

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

2.4.1 *Balanced Parentheses.* The problem of representing a sequence of balanced parentheses is highly related to the succinct representation of trees [Munro and Raman 2001]. 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 in the parentheses sequence; 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 [2001] 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. Among the practical alternatives, we have the representation of Geary et al. [2006] and the one by Navarro [2004, Section 6.1]. The latter has shown to be very effective for representing LZ-indices, and therefore we briefly review it in which follows.

Navarro's Practical Representation of Balanced Parentheses. To support the operations we could simply precompute and store all the possible answers, requiring $O(n \log n)$ bits overall. However, in many applications (e.g., the representation of trees) matching opening and closing parentheses tend to be close to each other. Profiting from this property, and for instance to support $findclose$, Navarro uses a brute-force approach for these parentheses, sequentially looking for the closing parenthesis within the next few, say 32, parentheses. Actually, this search is performed by using precomputed tables to avoid a bit-per-bit scan.

If the answer cannot be found in this way, Navarro searches a hash table storing the answers for parentheses that are not so close, though not so far away from each other. Say, for example, matching parentheses with a difference of up to 256 positions (parentheses). Instead of storing absolute positions, the difference between positions is stored, and thus we can use 8 bits to code these numbers, which saves space. Finally, if the answer cannot be found in the previous hash table, another table is searched for matching parentheses that are far away from each other (here full numbers are stored, but there are hopefully few entries). A similar approach is used to compute *enclose* and $findopen$ operations.

The parentheses operations are supported in $O(\log\log n)$ average time [Navarro 2009]. However, this representation does not provide theoretical worst-case guarantees in the space requirement, since in the worst case almost every opening parenthesis has its matching parenthesis far away, so we have to store its information in the tables. Fortunately these cases are not common in practice.

To compute operation $excess(i)$, we need to support operation *rank* over the binary sequence of parentheses, since $excess(i) \equiv rank_{(}(par, i) - rank_{)}(par, i)$. We use the representation of González et al. [2005] to efficiently support *rank* and *select* (which will be needed later) on *par*.

2.4.2 BP *Representation of Trees.* The *balanced parentheses* (BP) representation of a tree defined by Munro and Raman [2001] is built from a depth-first preorder traversal of the tree, writing an *opening* parenthesis when arriving to a node for the first time, and a *closing* parenthesis when going up (after traversing the subtree of the node). In this way, we get a sequence of balanced parentheses, where each node is represented by a pair of opening and closing parentheses. We identify a tree node x with its opening parenthesis in the representation. The subtree of x contains those nodes (parentheses) enclosed between the opening parenthesis representing x and its matching closing parenthesis.

This representation requires $2n + o(n)$ bits, and supports operations $parent(x)$ (which gets the parent of node x), $subtreesize(x)$ (which gets the size of the subtree of node x, including x itself), $depth(x)$ (which gets the depth of node x in the tree), $nextsibling(x)$ (which gets the next sibling of node x), and $ancestor(x, y)$ (which tells us whether node x is an ancestor of node y), all of them in $O(1)$ time, in the following way (let *par* be the sequence of balanced parentheses representing the tree):

$$\begin{aligned}
parent(x) &\equiv enclose(par, x)\\
subtreesize(x) &\equiv (findclose(par, x) - x + 1)/2\\
depth(x) &\equiv excess(par, x)\\
nextsibling(x) &\equiv findclose(par, x) + 1\\
ancestor(x, y) &\equiv x \leqslant y \leqslant findclose(par, x)
\end{aligned}$$

Operation $child(x, i)$ (which gets the i-th child of node x) can be computed in $O(i)$ time by repeatedly applying operation *nextsibling*. This takes, in the worst case,

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

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
par:	(	(	)	)	)	(	(	(	)	)	)	(	(	(	)	)	)	)
ids:	6	11				2	7	9				5	8	13				
letts:	_	p				l	a	b				_	a	p				

(a) Balanced parentheses representation of *LZTrie* for the running example.

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

	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
par:	(	)	)	)	)	(	)	)	(	)	(	)	)	(	)	(	)	)
ids:			12	10	16	6		11	2		7		9	5		8		13
letts:	l					p			a		b			a		p		

(b) DFUDS representation of *LZTrie* for the running example. The phrase identifiers are stored in preorder, and the symbols labeling the arcs of the trie are stored according to DFUDS.

Fig. 3. Succinct representations of *LZTrie* for the running example.

linear time on the maximum arity of the tree.

The *preorder position* of a node can be computed in this representation as the number of opening parentheses before the one representing the node. That is, $preorder(x) \equiv rank_{(}(par, x) - 1$. Notice that in this way we assume that the preorder of the tree root is always 0. Given a preorder position p, the corresponding node is computed by $selectnode(p) \equiv select_{(}(par, p+1)$.

In Fig. 3(a) we show the balanced parentheses representation for the *LZTrie* of Fig. 4(a), along with the sequence of LZ78 phrase identifiers sequence (*ids*) in preorder, and the sequence of symbols labeling the arcs of the trie (*letts*), also in preorder. As the identifier corresponding to the *LZTrie* root is always 0, we do not store it in *ids*. The data associated with node x is stored at position $preorder(x)$ both in *ids* and *letts* sequences. Note this information is sufficient to reconstruct *LZTrie*.

2.4.3 DFUDS *Representation of Trees.* To get this representation [Benoit et al. 2005] 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. 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 also $2n+o(n)$ bits, and supports operations $parent(x)$, $subtreesize(x)$, $degree(x)$ (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 [Jansson et al. 2007]), $ancestor(x, y)$, and $child(x, i)$, all in $O(1)$ time in the following way, assuming that *par* represents now the DFUDS sequence of the tree:

$$\begin{aligned}
parent(x) &\equiv select_{)}(par, rank_{)}(par, findopen(par, x-1))) + 1\\
child(x,i) &\equiv findclose(par, select_{)}(par, rank_{)}(par, x) + 1) - i) + 1\\
subtreesize(x) &\equiv (findclose(par, enclose(par, x)) - x)/2 + 1\\
degree(x) &\equiv select_{)}(par, rank_{)}(par, x) + 1) - x\\
childrank(x) &\equiv select_{)}(par, rank_{)}(par, findopen(par, x-1)) + 1)\\
&\quad - findopen(par, x-1)
\end{aligned}$$

1

$$ancestor(x,y) \equiv x \leqslant y \leqslant findclose(par, enclose(par,x))$$

Operation $depth(x)$ can be also computed in constant time on DFUDS by using the approach of Jansson et al. [2007], requiring $o(n)$ extra bits. It is important to note that, unlike the BP representation, DFUDS needs operation $findopen$ on the parentheses in order to compute operation $parent$ on the tree. In practice, if we build on Navarro's parentheses data structure, this implies that DFUDS needs more space than BP since we need additional hash tables to support $findopen$.

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 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(b) we show the DFUDS representation of *LZTrie* for our running example. Notice the inverse relation between the d opening parentheses defining x and the symbols of the children of x: the label of the i-th child is at position i within the symbols of the children of x, while the corresponding opening parenthesis is at position $(d - i + 1)$ within the definition of x. This shall mean extra work when retrieving the symbol by which a given node descends from its parent.

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 within 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. The second term, $childrank(x)$, comes from the inverse relation between symbols and opening parentheses representing a node.

Thus, the time for operation $child(x, \alpha)$ depends on the representation we use for $rank_\alpha$ and $select_\alpha$ queries. 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 needing any extra space on top of array $letts$. The access to $letts[\cdot]$ takes constant time.

Alternatively, we can represent $letts$ with the data structure of Ferragina et al. [2007], which requires $n \log \sigma + o(n \log \sigma)$ bits of space, and allows us to compute $child(x, \alpha)$ in $O(1 + \frac{\log \sigma}{\log \log n})$ time. The access to $letts[\cdot]$ also takes $O(1 + \frac{\log \sigma}{\log \log n})$

time. These times are $O(1)$ whenever $\sigma = O(\text{polylog}(n))$ holds. On the other hand, we can use the data structure of Golynski et al. [2006], requiring $O(n \log \sigma)$ bits of space, yet allowing us to compute $child(x, \alpha)$ in $O(\log\log\sigma)$ time, and access to $letts[\cdot]$ also in $O(\log\log\sigma)$ time. We will prefer the representation of Ferragina et al., since it is able to improve its time complexity to $O(1)$ for smaller alphabets.

The scheme we have presented to represent *letts* is slightly different to the original one [Benoit et al. 2005], which achieves $O(1)$ time for $child(x, \alpha)$ for any σ. However, ours is simpler and allows us to efficiently access $letts[\cdot]$, which will be very important in our indices to extract text substrings.

3. THE LZ-INDEX DATA STRUCTURE

Assume that the text $T[1..n]$ has been compressed using the LZ78 algorithm into $d+1$ phrases $T = B_0 \dots B_d$, as explained in Section 2.2. As we are mainly interested in practical performance in this paper, we describe next a practical representation of the LZ-index and its corresponding search algorithm [Navarro 2009, Section 9].

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

3.1 Original LZ-index Components

The following data structures conform the practical version of LZ-index [Navarro 2004; 2009]:

(1) *LZTrie*: is the trie formed by all the phrases $B_0, \dots, B_d$. Given the properties of LZ78 compression, this trie has exactly $d+1$ nodes, each one corresponding to a phrase.

(2) *RevTrie*: is the trie formed by all the reverse strings $B_0^r, \dots, B_d^r$. In this trie there could be internal nodes not representing any phrase. We call these nodes "*empty*". Empty unary paths are compressed.

(3) *Node*: is a mapping from phrase identifiers to their node in *LZTrie*.

(4) *RNode*: is a mapping from phrase identifiers to their node in *RevTrie*.

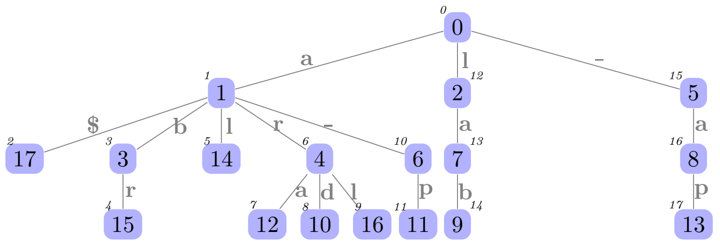
Fig. 4 shows the *LZTrie*, *RevTrie*, *Node*, and *RNode* data structures corresponding to our running example. We show preorder numbers, both in *LZTrie* and *RevTrie* (in the latter case only counting non-empty nodes), outside each trie node. In the case of *RevTrie*, empty nodes are shown in light gray.

3.2 Succinct Representation of the LZ-index Components

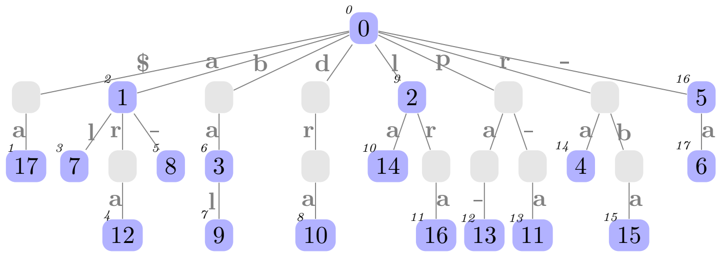
In the original work [Navarro 2004; 2009], each of the four structures described requires $d \log d + o(n \log \sigma)$ bits of space if they are represented succinctly.

—*LZTrie* is represented using the balanced parentheses representation [Munro and Raman 2001] requiring $2d + o(d)$ bits; plus the sequence *letts* of symbols labeling each trie edge, requiring $d \log \sigma$ bits; and the sequence *ids* of $d \log d$ bits storing the LZ78 phrase identifiers. Both *letts* and *ids* are stored in preorder, so we use $preorder(x)$ to index them. See Fig. 3(a) for an illustration.

—For *RevTrie*, balanced parentheses are also used to represent the *Patricia* tree [Morrison 1968] structure of the trie, compressing empty unary nodes and so



(a) Lempel-Ziv Trie (*LZTrie*) for the running example.



(b) *RevTrie* data structure.

```
        0              5              10             15             20             25             30             35          39
   par: (  (  )  (  (  )  (  )  (  )  )  (  (  )  )  (  )  (  (  )  (  )  )  (  (  )  (  )  )  (  (  )  (  )  )  (  (  )  )  )
     B: 1  1     1  1     1     1        1  1        1     1  1     1        0  1     1        0  1     1        1  1
rletts:    $     a  l     r     _        b  l        d     l  a     r        p  a     _        r  a     b        _  a
  rids: 0  17    1  7     12    8        3  9        10    2  14    16          13    11          4     15       5  6
```

(c) Balanced parentheses representation of *RevTrie*, compressing empty unary paths. The bitmap B marks with a 0 the empty non-unary nodes. Notice that array *rids* stores phrase identifiers only for non-empty nodes.

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

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

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$RNode[i]$	0	3	17	11	30	35	36	4	8	12	15	26	6	24	18	32	20	1

(e) *RNode* data structure, assuming that the parentheses sequence starts from position zero.

Fig. 4. LZ-index components for the running example.

ensuring $d' \leqslant 2d$ nodes. This requires at most $4d + o(d)$ bits. The *RevTrie*-preorder sequence of identifiers (*rids*) is stored in $d \log d$ bits (i.e., we only store the identifiers for non-empty nodes). Non-empty nodes are marked with bit vector $B[1..d']$, such that $B[j] = 0$ iff node x with preorder j is empty. Thus, the phrase identifier for node x is $rids[rank_1(B, j)]$. The symbols labeling the arcs of the trie and the Patricia-tree skips are not stored in this representation, since they can be retrieved by using the connection with *LZTrie*. Therefore, the navigation on *RevTrie* is more expensive than that on *LZTrie*.

1

—*Node* is just a sequence of d pointers to *LZTrie* nodes. As *LZTrie* is implemented using balanced parentheses, *Node*$[i]$ stores the position within the sequence for the opening parenthesis representing the node corresponding to phrase i. As there are $2d$ such positions, we need $d \log 2d = d \log d + d$ bits of storage. See Fig. 4(d) for an illustration.

—Finally, *RNode* is also represented as a sequence of d pointers to *RevTrie* nodes. As *RevTrie* has $d' \leqslant 2d$ nodes and since we use balanced parentheses for this trie, we need $d \log 4d = d \log d + 2d$ bits of space. See Fig. 4(e) for an illustration.

According to Lemma 2.7, the final size of the LZ-index is $4nH_k(T) + o(n \log \sigma)$ bits for $k = o(\log_\sigma n)$ (and hence $\log \sigma = o(\log n)$ if we want to allow for $k > 0$).

In theory, the succinct trie representations used [Navarro 2004] implement (among others) operations $parent(x)$ and $child(x, \alpha)$, both in $O(\log \sigma)$ time for *LZTrie*, and $O(\log \sigma)$ and $O(h \log \sigma)$ time respectively for *RevTrie*, where h is the depth of node x in *RevTrie* (the h in the cost comes from the fact that we must access *LZTrie* to get the label of a *RevTrie* edge). The operation $ancestor(x, y)$ is implemented in $O(1)$ time both in *LZTrie* and *RevTrie*.

In practice, however, the BP representation [Munro and Raman 2001] for general trees is used. Despite that under this representation operation $child(x, \alpha)$ is implemented by using operation $child(x, i)$ in $O(\sigma)$ worst-case time, this has shown to be very effective in practice [Navarro 2004; 2009]. Operation *parent* is supported in $O(1)$ time under this representation.

3.3 LZ-index Search Algorithm

Let us consider now the search algorithm for a pattern $P[1..m]$ [Navarro 2004; 2009]. For `locate` queries, pattern occurrences are originally reported in the format $\langle t, offset \rangle$, where t is the phrase where the occurrence starts, and $offset$ is the distance between the beginning of the occurrence and the end of the phrase. Later, in Section 5.2, we will show how to map these two values into a simple *text position*. As we deal with an implicit representation of the text (the *LZTrie*), and not the text itself, we distinguish three types of occurrences of P in T, depending on the phrase layout.

3.3.1 *Occurrences of Type 1.* The occurrence lies inside a single phrase (there are occ_1 occurrences of this type). Given the properties of LZ78, every phrase B_t containing P is formed by a shorter phrase B_ℓ concatenated to a symbol c (Property 2.3). If P does not occur at the end of B_t, then B_ℓ contains P as well. We want to find the shortest possible phrase B_i in the LZ78 referencing chain for B_t that contains the occurrence of P, that is, P is a suffix of B_i. But then P^r is a prefix of B_i^r, and it can be easily found by searching for P^r in *RevTrie*. Say we arrive at node v_r. Any node v_r' descending from v_r in *RevTrie* (including v_r itself) corresponds to a phrase terminated with P. For each such v_r', we traverse and report the subtree of the corresponding *LZTrie* node v_{lz} (found using *rids* and *Node*). For any node v_{lz}' in the subtree of v_{lz}, we report an occurrence $\langle t, m + (depth(v_{lz}') - depth(v_{lz})) \rangle$, where t is the phrase identifier (*ids*) of node v_{lz}'. Occurrences of type 1 are located in constant time each. For `count` queries we just need to compute the subtree size of each v_{lz} in *LZTrie*.

3.3.2 *Occurrences of Type 2.* The occurrence spans two consecutive phrases, B_t and B_{t+1}, such that a prefix $P[1..i]$ matches a suffix of B_t and the suffix $P[i+1..m]$ matches a prefix of B_{t+1} (there are occ_2 occurrences of this type). P can be split at any position, so we have to try them all. For every possible split, we search for the reverse pattern prefix $P^r[1..i]$ in *RevTrie* (getting node v_r) and for the pattern suffix $P[i+1 \ldots m]$ in *LZTrie* (getting node v_{lz}). The *RevTrie* node v_r for $P^r[1..i]$ is stored in array $C_r[i]$, since it shall be needed later. As in a trie all the strings represented in a subtree form a preorder interval, we have two preorder intervals: one in the space of reversed phrases (phrases finishing with $P[1..i]$) and one in that of the normal phrases (phrases starting with $P[i+1..m]$), and need to find the phrase pairs $(t, t+1)$ such that t is in the *RevTrie* interval and $t+1$ is in the *LZTrie* interval. Then, we check each phrase t in the subtree of v_r and report it if $Node[t+1]$ descends from v_{lz}. Each such check takes constant time. Yet, if the subtree of v_{lz} has fewer elements, one does the opposite: check phrases from v_{lz} in v_r, using $RNode[t-1]$. For every pair of consecutive phrases that passes this test, we report an occurrence Dt, iT.

The time to solve occurrences of type 2 is proportional to the smallest subtree size among v_r and v_{lz}, which can be arbitrarily larger than the number of occurrences reported. That is, we have no worst-case guarantees at search time[1]. However, the average number of candidates per occurrence of type 2 is $O(\sigma^{m/2})$ [Navarro 2009][2].

3.3.3 *Occurrences of Type 3.* The occurrence spans three or more phrases, B_{t-1}, $\ldots, B_{\ell+1}$, such that $P[i..j] = B_t \ldots B_\ell$, $P[1..i-1]$ matches a suffix of B_{t-1} and $P[j+1..m]$ matches a prefix of $B_{\ell+1}$ (there are occ_3 occurrences of this type). Since the LZ78 algorithm guarantees that every phrase represents a different string (Property 2.4), there is at most one phrase matching $P[i..j]$ for each choice of i and j. Therefore, if we partition P into more than two consecutive substrings, there is at most one pattern occurrence for such partition, which severely limits occ_3 to $O(m^2)$, the number of different partitions of P.

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

Then we find the $O(m^2)$ maximal concatenations of successive phrases that match contiguous pattern substrings. For $1 \leqslant i \leqslant j \leqslant m$, for increasing j, we try to extend the match of $P[i..j]$ to the right. If id is the phrase identifier for node $C_{lz}[i,j]$, then we have to search for $(id+1, r)$ in array A_{j+1}, for some r. Array A_{j+1} can be binary searched because it is sorted. If we find $(id+1, r)$ in A_{j+1}, this means that $B_{id} = P[i..j]$ and $B_{id+1} = P[j+1..r]$, i.e. the concatenation $B_{id}B_{id+1}$ equals

[1]The theoretical version [Navarro 2004] uses different structures which do offer such guarantees.
[2]Navarro [2009] chooses to express this as $O(\sqrt{n/occ})$.

$P[i..r]$. We repeat the process from $j = r$, and stop when the pair $(id + 1, r)$ is not found in the corresponding array (this means that the current concatenation is maximal). In practice, the binary search is replaced by closed hashing schemes, taking constant time on average per search [Navarro 2004, Section 6.5].

Once $P[i..j] = B_t \dots B_\ell$ is a maximal concatenation, we check whether phrase $B_{\ell+1}$ starts with $P[j+1..m]$ by using operation $\mathit{ancestor}(C_{lz}[j+1, m], \mathit{Node}[\ell+1])$, in constant time per maximal concatenation. Finally we check whether phrase B_{t-1} ends with $P[1..i-1]$ by checking whether $\mathit{ancestor}(C_r[i-1], \mathit{RNode}[t-1])$ holds in *RevTrie*, in constant time per maximal concatenation. If all these conditions hold, we report an occurrence $\mathrm{D}t - 1, i - 1\mathrm{T}$.

3.3.4 *The Search Algorithm in Practice.* In practice, the search algorithm proceeds as follows. We first search for every pattern substring $P[i..j]$ in *LZTrie*, and store the corresponding node in $C_{lz}[i, j]$. The search proceeds looking first for $P[1]$, then for $P[1..2]$, then for $P[1..3]$, and so on until we cannot find $P[1..j]$, for some j, or until $j = m$. Then we do the same starting from $P[2]$, and so on. We also store a matrix $C_{id}[1..m, 1..m]$, where $C_{id}[i, j]$ is the LZ78 phrase identifier for phrase $P[i..j]$; if $P[i..j]$ does not exist as an LZ78 phrase, then we store a null value.

The second step consists of searching for every prefix of the reversed pattern $P^r[1..j]$ in *RevTrie*. Recall that we need this for occurrences of type 1 and type 2. Recall also that searching in *RevTrie* is much slower that searching in *LZTrie*, so we try to reduce this work as much as possible. The results already obtained in C_{id} are useful. If we search for $P^r[1..j]$ in *RevTrie*, and $P[1..j]$ exists as a phrase in *LZTrie*, then $\mathit{RNode}[C_{id}[1, j]]$ is the *RevTrie* node we are looking for. Otherwise, $P^r[1..j]$ corresponds to an empty node in *RevTrie*, or to a position in a label between two nodes, and cannot be found with the *LZTrie*. Yet, we can reduce the cost as follows. Let i be the minimum value such that $C_{lz}[i, j]$ is defined, i.e. $P[i..j]$ exists as a phrase in *LZTrie* (and hence $P^r[i..j]$ exists as a reverse phrase in *RevTrie*). Then, we map to *RevTrie* with $\mathit{RNode}[C_{id}[i, j]]$, which corresponds to string $P^r[i..j]$, and from this node try to descend with the string $P^r[1..i-1]$. This final search has to be done using operation *child* on *RevTrie*. The *RevTrie* node corresponding to $P^r[1..i]$ is stored in $C_r[i]$.

Then we use these results to search for occurrences of type 1, type 2, and type 3 respectively. The overall time is shown to be $O(\sigma m \log_\sigma n + \mathit{occ}\, \sigma^{m/2})$ on average [Navarro 2009].

3.3.5 *Displaying Occurrence Contexts.* To display a context of length ℓ surrounding any occurrence reported, if an occurrence starts at phrase i, then we follow the upward path from $\mathit{Node}[i]$ up to the *LZTrie* root, outputting the symbols labeling the upward path. Then we perform the same procedure but now starting from $\mathit{Node}[i + 1]$ (alternatively $\mathit{Node}[i - 1]$) in *LZTrie*, and so on until we display the ℓ desired symbols, taking overall $O(\ell \log \sigma)$ time, because operation *parent* is supported in $O(\log \sigma)$ time in theory [Navarro 2004]. In practice, and using the BP representation for the tries, the time is $O(\ell)$. Finally, we can uncompress the whole text T in $O(n \log \sigma)$ time using the same idea, starting the procedure from the first LZ78 phrase (the time is $O(n)$ in practice).

4. THE LZ-INDEX AS A NAVIGATION SCHEME

4.1 The Original Navigation Scheme

The LZ-index structure can be regarded as a *navigation scheme* that permits us moving back and forth from trie nodes to the corresponding *preorder positions*, both in *LZTrie* and *RevTrie*. The phrase identifiers are common to both tries (arrays *ids* and *rids*) and permit moving from one trie to the other by using *Node* and *RNode* mappings.

Fig. 5 shows the navigation scheme, where solid arrows represent the main data structures of the index. Dashed arrows are asymptotically "for free" in terms of space requirement, since they are followed by applying *preorder* on the corresponding parentheses structure (see Section 2.4). From now on we use the subscript "*lz*" for the operations on *LZTrie*, and subscript "*r*" for *RevTrie*. The four solid arrows in the diagram are in fact the four main components in the space usage of the LZ-index: array of phrase identifiers in *LZTrie* (*ids*) and in *RevTrie* (*rids*), and mapping from phrase identifiers to tree nodes in *LZTrie* (*Node*) and in *RevTrie* (*RNode*). The structure is symmetric and we can move from any point to any other.

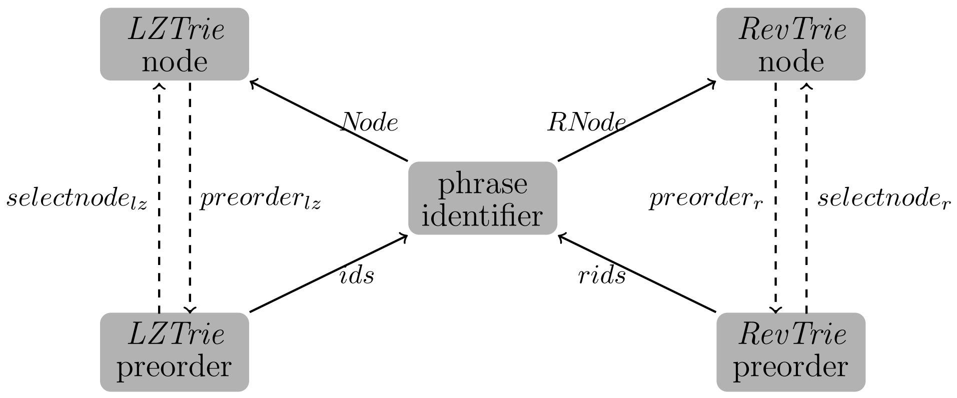


Fig. 5. The original LZ-index navigation structures over index components.

The structure, however, is redundant in the sense that the number of arrows is not minimal. Given a graph with t nodes (in our case t index components), t arrows are sufficient to connect them in both directions (actually forming a ring structure). Since nodes and preorder positions in the tries are "connected" using operations *preorder* and *selectnode* over the trie representations (see Section 2.4), we can think that there are only three main index components to connect: *LZTrie* (either nodes or preorder positions), phrase identifiers, and *RevTrie* (either nodes or preorder positions). Next we define more space-efficient representations for LZ-index, trying to reduce the number of arrows in the scheme. Note that, because of Lemma 2.7, we are interested in reducing the number of the index components that require $d \log d = nH_k(T) + o(n \log \sigma)$ bits of storage.

4.2 Schemes Requiring $3nH_k + o(n \log \sigma)$ bits

In this section we present schemes requiring only three solid arrows to connect the LZ-index components, thus forming a ring structure that still allows the same navigation as in the original LZ-index. Different choices yield different efficiencies depending on how often is each type of navigation used during the search.

4.2.1 *Scheme 1.* The following data structures conform this version of LZ-index:

(1) *LZTrie*: The Lempel-Ziv trie, which is implemented with the following data structures
  —$par[0..2d-1]$: The tree shape of *LZTrie* represented either with balanced parentheses [Munro and Raman 2001] or with DFUDS [Benoit et al. 2005], requiring in any case $2d + o(d)$ bits.
  —$letts[1..d]$: The array of symbols labeling the arcs of *LZTrie*, represented as explained in Section 2.3, depending on the representation used for *par*.
  —$ids[1..d]$: The array of LZ78 phrase identifiers in preorder. Since $ids[0] = 0$, we do not store this value. Note that *ids* is a permutation of $\{1, \ldots, d\}$. The space requirement is $d \log d$ bits.

(2) *RevTrie*: The *Patricia* tree [Morrison 1968] of the reversed LZ78 phrases, which is implemented with the following data structures
  —$rpar[0..2d'-1]$: The *RevTrie* structure, represented either with BP or with DFUDS, compressing empty unary paths and thus ensuring $d' \leqslant 2d$ nodes, because empty non-unary nodes still exist. Thus, the space requirement is $2d' + o(d')$ bits.
  —$rletts[1..d']$: the array storing the first symbol of each edge label in *RevTrie*, represented as for *LZTrie* and requiring $d' \log \sigma + o(d')$ bits of space.
  —$skips[1..d']$: the *Patricia tree* skips of the nodes in preorder, using $\log \log n$ bits per node and inserting empty unary nodes when the skip exceeds $\log n$. In this way, one out of $\log n$ empty unary nodes could be explicitly represented. In the worst case there are $O(n)$ empty unary nodes, of which $O(n/\log n)$ are explicitly represented. This adds $O(n/\log n)$ nodes to d', which translates into $O((d' + \frac{n}{\log n})(3 + \log \sigma + \log \log n)) = o(n \log \sigma)$ bits overall for the *RevTrie* nodes, symbols, and skips.
  —$B[1..d']$: A bit vector supporting *rank* and *select* queries, and requiring $d'(1 + o(1))$ bits of space [Munro 1996]. This bit vector marks the non-empty nodes: The j-th bit of B is 1 iff the node with preorder position j in *rpar* is not empty, otherwise the bit is 0. Given a position i in *rpar* corresponding to a *RevTrie* node, the corresponding bit in B is $B[preorder_r(i)]$. The preorder of a node p counting only non-empty nodes can be computed as $rank_1(B, preorder_r(i))$.

(3) $RNode[1..d]$: The mapping from phrase identifiers to the corresponding *RevTrie* node. Since we represent nodes as the positions of opening parentheses, and since there are $2d' \leqslant 4d$ such positions in *RevTrie*, this mapping needs $d \log 4d = d \log d + 2d$ bits. We only store pointers to non-empty nodes.

(4) $Rev[1..d]$: A mapping from a *RevTrie* preorder position to the corresponding *LZTrie* node, defined as $Rev[i] = Node[rids[i]]$. Given a position i in *rpar* corresponding to a non-empty *RevTrie* node, the corresponding *Rev* value

(i.e., *LZTrie* node) is $Rev[rank_1(B, preorder_r(i))]$. The space requirement is $d \log d + d$ bits.

The resulting navigation scheme is shown in Fig. 6(a). The search algorithm remains the same since we can map preorder positions to nodes in the tries and vice versa (see Section 2.3), and also we can simulate the missing arrays $rids(i) \equiv ids[preorder_{lz}(Rev[i])]$ and $Node(i) \equiv Rev[rank_1(B, preorder_r(RNode[i]))]$, all of which take constant time.

We have reduced the space requirement to $3d \log d + 3d \log \sigma + 2d \log \log n + 11d + o(n) = 3d \log d + o(n \log \sigma)$ bits if $\log \sigma = o(\log n)$, which according to Lemma 2.7 is $3nH_k(T) + o(n \log \sigma)$ bits, for any $k = o(\log_\sigma n)$.

The *child* operation on *RevTrie* can now be computed in $O(1)$ time if we use DFUDS, and because we store *rletts* and the skips. Compare to the $O(h \log \sigma)$ time of the original LZ-index [Navarro 2004]. Now, because *RevTrie* is a Patricia tree and the underlying strings are not readily available, it is not obvious how to traverse it. The next lemma addresses this issue.

LEMMA 4.1. *Given a string $s \in \Sigma^*$, we can determine whether it is represented in RevTrie or not (finding the corresponding node in the affirmative case) in $O(|s|)$ time.*

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

In case node v_r in *RevTrie* is empty, $Rev[j]$ is undefined. Notice, however, that there must be at least one non-empty node descending from this empty node, since leaves in *RevTrie* cannot be empty as they always correspond to an LZ78 phrase. Given that the string represented by every non-empty node in the subtree of node v_r has the string s as a prefix, the corresponding strings in *LZTrie* have s^r as a suffix. So we can use any *Rev* value within the subtree of node v_r in order to map to the *LZTrie* and then extract the string it represents. We can use, for example, the value $Rev[rank_1(B, j) + 1]$, which corresponds to the next non-empty node within the subtree of node v_r. We know when to stop extracting, since we know the length of the string we are looking for.

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

Operations *child* and *parent* on *LZTrie* can be also computed in $O(1)$ time if we use DFUDS on this trie, versus the $O(\log \sigma)$ time (in theory) of the original LZ-index. In practice we use binary search on the children, so our *child* time is $O(\log \sigma)$ (versus the $O(\sigma)$ time in practice of the original LZ-index). Hence, the original practical LZ-index complexities [Navarro 2009] become $O((\log \sigma) m \log_\sigma n +$

$occ\,\sigma^{m/2}) = O(m \log n + occ\,\sigma^{m/2})$. We have shown elsewhere how to achieve worst-case guarantees with reduced schemes of the LZ-index [Arroyuelo et al. 2006], but we are focusing in practical performance here.

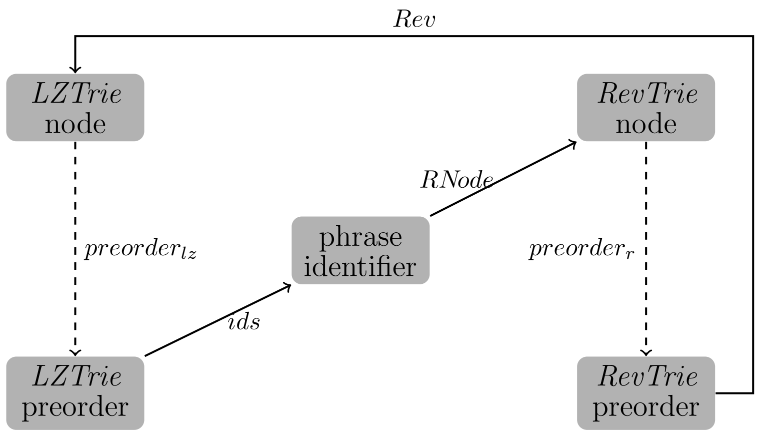
Practical Issues. On the practical side, the access from *RevTrie* nodes to the corresponding *LZTrie* node is faster under this scheme, since the direct link *Rev* is faster than the composition of *rids* and *Node* of the original scheme. This is good, for example, for finding occurrences of type 1, which can be dominant for short patterns, as there is a high probability that an occurrence is contained in a single phrase. However, sometimes we must follow longer navigation paths in the search process: for example, when finding occurrences of type 2, we can choose to traverse the subtree in *RevTrie*, and for each phrase identifier *id* in such subtree apply $Node(id+1)$ to check whether it descends from the appropriate subtree in *LZTrie*. As now we have to simulate *Node*, this is more expensive (in practice, even if not asymptotically) than in the original scheme. Even worse, since array *rids* is not stored in *RevTrie*, we must simulate $rids(i)$ to get the phrase identifier *id*. Therefore, the search time could be increased in practice, depending on the number of each type of occurrence.

Let us study occurrences of type 2 in more detail, since they seem to be critical under this scheme. Suppose that for a given partition $P[1..i]$ and $P[i+1..m]$ of P we get nodes v_{lz} and v_r in *LZTrie* and *RevTrie* respectively. If we choose to traverse the subtree of v_r in *RevTrie*, then for each node v_r' in this subtree we get the corresponding phrase identifier $id = ids[preorder_{lz}(Rev[p_r])]$, where p_r is set initially as $p_r = rank_1(B, preorder_r(v_r))$, and it is incremented by one with each node in a preorder traversal of the subtree. We then check whether the node $Rev[rank_1(B, preorder_r(RNode[id+1]))]$ descends from v_{lz} in *LZTrie*. If, on the other hand, we choose to traverse the subtree of v_{lz} in *LZTrie*, then for each node v_{lz}' in this subtree we get the phrase identifier as $id = ids[p_{lz}]$, where p_{lz} is set initially as $p_{lz} = preorder_{lz}(v_{lz})$, and it is incremented by one with each node in a preorder traversal. We then check whether the node $RNode[id-1]$ descends from v_r in *RevTrie*. Empirically, a check from *RevTrie* to *LZTrie* is about 3 times as expensive as in the opposite direction, and thus we choose to traverse the subtree of v_r whenever its size is less than 1/3 the size of the subtree of v_{lz}.

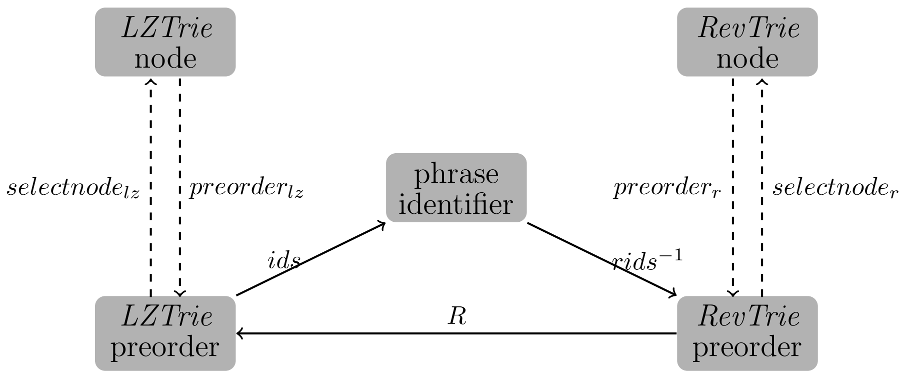
4.2.2 *Scheme 2.* This scheme tries to reduce the space requirement while also reducing the average path length in the navigation scheme:

(1) *LZTrie* The Lempel-Ziv trie, defined just as for Scheme 1.
(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined just as for Scheme 1, but now we add
—$rids^{-1}[1..d]$: The explicit representation of the inverse of permutation *rids* of the original LZ-index definition, requiring $d \log d$ bits.
(3) $R[1..d]$: A mapping from *RevTrie* preorder positions to *LZTrie* preorder positions defined as $R[i] = ids^{-1}(rids[i])$ and requiring $d \log d$ bits. Given a position i in *rpar* corresponding to a non-empty *RevTrie* node, the corresponding R value (i.e., preorder in *LZTrie*) can be computed as $R[rank_1(B, preorder_r(i))]$.

The resulting navigation scheme is shown in Fig. 6(b). We can compute $rids(i) \equiv ids[R[i]]$, $RNode(i) \equiv selectnode_r(rids^{-1}[i])$, and $Node(i) \equiv selectnode_{lz}(R[rids^{-1}$



(a) Scheme 1.



(b) Scheme 2.

Fig. 6. Reduced navigation schemes over LZ-index components, requiring $3uH_k + o(u \log \sigma)$ bits.

$[i]]$), all in constant time.

The space requirement is $3d \log d + 3d \log \sigma + 2d \log \log n + 8d + o(n) = 3d \log d + o(n \log \sigma)$ bits if $\log \sigma = o(\log n)$, which according to Lemma 2.7 is $3nH_k(T) + o(n \log \sigma)$ bits, for any $k = o(\log_\sigma n)$.

If we use DFUDS to represent both *LZTrie* and *RevTrie*, then we can locate the *occ* occurrences of pattern P in $O(m \log n + occ\, \sigma^{m/2})$ average time.

Practical Issues. It is interesting to note that the average path length of this scheme is shorter than that of Scheme 1, which can translate into a more efficient navigation among index components. In this scheme, for occurrences of type 1 we have direct access to a *LZTrie* preorder by using R, and then we have to apply *selectnode* to get the node whose subtree contains the pattern occurrences. This can be slightly slower than for Scheme 1, where we have direct access to the corresponding node.

However, for occurrences of type 2 we have to follow shorter paths than for Scheme 1. Suppose that for a given partition $P[1..i]$ and $P[i+1..m]$ of P we get nodes v_{lz} and v_r in *LZTrie* and *RevTrie* respectively. If we choose to traverse the subtree of v_r in *RevTrie*, for each node in this subtree we get the corresponding phrase identifier *id* by using both the R mapping and then *ids*. Then we use $R[rids^{-1}[id+1]]$ to get the corresponding *LZTrie* preorder, and then we check whether this preorder lies within the subtree of v_{lz}. If, on the other hand, we choose to traverse the subtree of v_{lz} in *LZTrie*, for every node in this subtree we get the corresponding phrase identifier *id* using *ids*, and then we check whether the preorder $rids^{-1}[id-1]$ lies within the subtree of v_r in *RevTrie*. Thus, a check from *RevTrie* to *LZTrie* is twice as expensive as in the opposite direction, and thus we choose to traverse the subtree of v_r whenever its size is less than half the size of the subtree of v_{lz}.

Fortunately, the checks of type 2 can be carried out directly on the preorders of both tries, avoiding the use of (the usually expensive) *selectnode* to get the corresponding trie node: if we choose to traverse the subtree of v_r, for example, we compute the preorder interval for the subtree of v_{lz} as $[preorder_{lz}(v_{lz})..preorder_{lz}(v_{lz})+ subtreesize_{lz}(v_{lz})-1]$ (recall that *preorder* is computed by means of *rank*), and then we check whether the *LZTrie* preorders we get from the nodes in the subtree of v_r lie within the preorder interval of v_{lz}. In this way, we compute just one *rank* per partition to get the interval, and then we check the *LZTrie* preorder of the candidates by using just this interval, rather than computing *selectnode* for every possible candidate in that partition. This introduces very important savings in the practical search time.

There are many other possible schemes that achieve $3nH_k(T)+o(n\log\sigma)$ bits of space. We have focused on the two most promising ones. For example, consider a scheme where we only replace the R mapping of Scheme 2 by the *Rev* mapping of Scheme 1. We have again direct access for occurrences of type 1, but occurrences of type 2 now introduce the computation of *rank* in *LZTrie* for every possible candidate, which is expensive.

4.3 Schemes Requiring $2(1+\epsilon)nH_k+o(n\log\sigma)$ bits

In Section 4.2 we have used the minimal number of arrows to connect the three main components of LZ-index, forming a ring structure. It seems that we cannot reduce the space requirement of the index further by using our navigation-scheme approach. However, many of the data structures of the LZ-index are just permutations, and so the corresponding arrows can be made bidirectional by means of the data structure for permutations [Munro et al. 2003] described in Section 2.3, using just $(1+\epsilon)d\log d+d+o(d)$ bits for both arrows. This opens several new possibilities.

4.3.1 *Scheme 3.* This scheme represents the following data:

(1) *LZTrie*: The Lempel-Ziv trie, defined as for Scheme 1, except that now we use the representation of Munro et al. [2003] for *ids* such that the inverse permutation ids^{-1} can be computed in $O(1/\epsilon)$ time, requiring $(1+\epsilon)d\log d+d+o(d)$ bits for any $0<\epsilon<1$.

(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined as in Scheme 1, but now we add the array *rids* represented using [Munro et al. 2003], so as to be able to compute $rids^{-1}$ efficiently.

The resulting navigation scheme is shown in Fig. 7(a). We can simulate the missing arrays $Node(i) \equiv selectnode_{lz}(ids^{-1}(i))$ and $RNode(i) \equiv selectnode_r(rids^{-1}(i))$, all in $O(1/\epsilon)$ time.

The space requirement is $2(1+\epsilon)d\log d + 3d\log\sigma + 2d\log\log n + 10d + o(n) = 2(1+\epsilon)d\log d + o(n\log\sigma)$ bits, which according to Lemma 2.7 is $2(1+\epsilon)nH_k(T) + o(n\log\sigma)$ bits, for any $k = o(\log_\sigma n)$.

Each occurrence of type 1 can be located in $O(1/\epsilon)$ time, because we must use ids^{-1} to access to *LZTrie*; each candidate check for occurrences of type 2 requires time $O(1/\epsilon)$ because we must use inverse permutations to move between tries; and each candidate check for occurrences of type 3 requires $O(1/\epsilon)$ time, since we need to use *Node* and *RNode*. Thus, the *occ* occurrences of P can be located in $O(\frac{1}{\epsilon}(m\log n + occ\,\sigma^{m/2}))$ average time, for $0 < \epsilon < 1$.

Practical Issues. This scheme stores the phrase identifiers for both tries which, as we have seen for the previous schemes, is very convenient for occurrences of type 2: recall that when traversing the *RevTrie* subtree we have to get the phrase identifier of each node in the subtree; if we do not store the *RevTrie* identifiers, we have to access *LZTrie* to get them (as is the case of Schemes 1 and 2) and then we have to access *LZTrie* again to perform the check. This is not the case for Scheme 3. However, now we have paths including inverse permutations, which introduce an extra time overhead in practice.

Notice also that this scheme is symmetric in the sense that the checks for occurrences of type 2 cost the same in any direction we choose.

4.3.2 *Scheme 4.* This scheme represents the following data:

(1) *LZTrie*: The Lempel-Ziv trie, defined just as in Scheme 3.

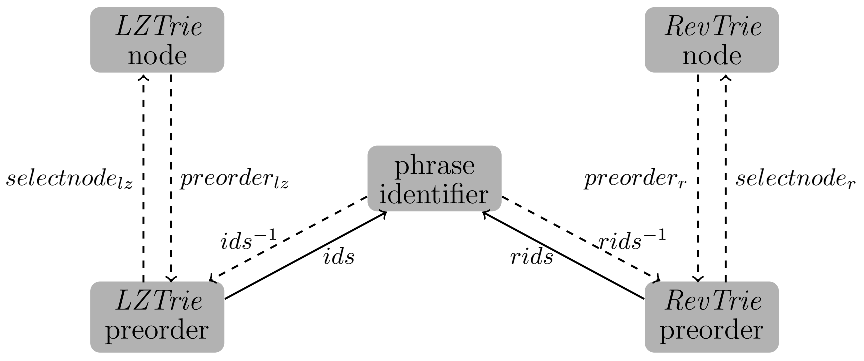
(2) *RevTrie*: The *Patricia* tree of the reversed LZ78 phrases, defined just as in Scheme 1.

(3) $R[1..d]$: The mapping from *RevTrie* preorder positions to *LZTrie* preorder positions, as defined in Scheme 2. This time R is implemented using the succinct data structure for permutations of Munro et al. [2003], requiring $(1+\epsilon)d\log d + d + o(d)$ bits to represent R and compute R^{-1} in $O(1/\epsilon)$ worst-case time.

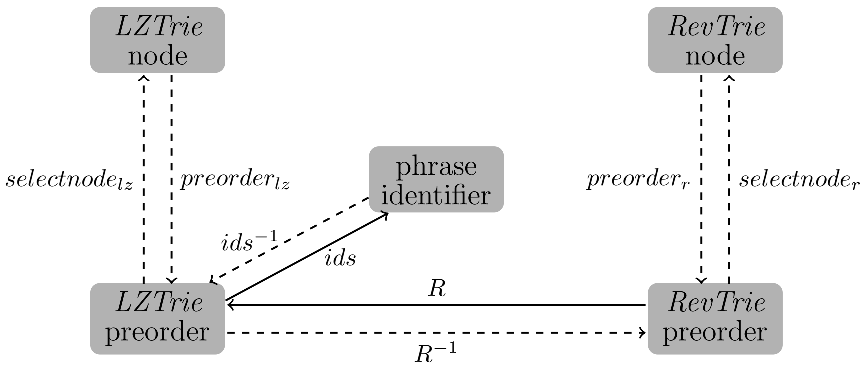
In Fig. 7(b) we draw the navigation scheme. We can simulate the missing arrays $rids(i) \equiv ids[R[i]]$, $RNode(i) \equiv selectnode_r(R^{-1}(ids^{-1}(i)))$, and $Node(i) \equiv selectnode_{lz}(ids^{-1}(i))$, all of which take $O(1/\epsilon)$ time.

The space requirement is $2(1+\epsilon)d\log d + 3d\log\sigma + 2d\log\log n + 10d + o(n) = 2(1+\epsilon)d\log d + o(n\log\sigma)$ bits, which according to Lemma 2.7 is $2(1+\epsilon)nH_k(T) + o(n\log\sigma)$ bits, for any $k = o(\log_\sigma n)$. Just as for the previous scheme, the *occ* occurrences of P can be located in $O(\frac{1}{\epsilon}(m\log n + occ\,\sigma^{m/2}))$ average time, for $0 < \epsilon < 1$.

Practical Issues. This scheme has more efficient access between tries than Scheme 3, as we have to use R in the *RevTrie*-to-*LZTrie* direction, and R^{-1} in the opposite



(a) Scheme 3.



(b) Scheme 4.

Fig. 7. Reduced navigation schemes over LZ-index components, requiring $2(1+\epsilon)uH_k + o(u \log \sigma)$ bits.

way. However, since we store the phrase identifiers only in *LZTrie*, retrieving the identifier of a *RevTrie* node requires to access two arrays. For occurrences of type 2, the checks from *RevTrie* to *LZTrie* require to access R, then ids, and finally ids^{-1}, while in the opposite way we need to use ids, then ids^{-1}, and finally R^{-1}. The latter case can be more expensive since we have to compute two inverse permutations. Note also that ids^{-1} is used in both directions for occurrences of type 2, which means that this inverse permutation is the most used at search time. Hence, given an amount of space we are able to use, we should use a denser sampling for ids^{-1} than for R^{-1}.

Again, we have focused on the most promising schemes requiring $2(1+\epsilon)nH_k(T) + o(n \log \sigma)$ bits, although there are many other choices.

5. SOME IMPLEMENTATION DETAILS

We describe in this section the most important details in the implementation of our indices. We followed the API interface specification provided in the *Pizza&Chili*

Corpus [Ferragina and Navarro 2005], and made the source code available at `http://pizzachili.dcc.uchile.cl/indexes/LZ-index/`.

5.1 Representing the Tries

We defined our indices in Section 4 in such a way that we can use almost any suitable representation for the tries that compose the indices. We just need to define accordingly the *preorder* and *selectnode* operations for the chosen representation. Having this into account, we implement our reduced LZ-indices in two different ways:

(1) Using the BP representation [Munro and Raman 2001] for both *LZTrie* and *RevTrie*;

(2) using the DFUDS representation [Benoit et al. 2005] for *LZTrie*, and the BP representation for *RevTrie*.

Note we do not use DFUDS for *RevTrie*, as it requires more space. Moreover, just as for the original LZ-index, we do not store the symbols labeling the *RevTrie* edges (i.e., the first symbol of each string labeling an edge) nor the Patricia-tree skips. This is in order to save space in practice, since these can be computed by using the connection with the *LZTrie*: previous experiments [Navarro 2004; 2009] showed that this is sufficient for *RevTrie* as most navigations on it are supported by using the *LZTrie* (and those that are not are usually deep in the trie, where the arity is very low and the attractive of DFUDS vanishes). We needed to use these arrays in theory in order to guarantee the average-time complexity of our data structures.

We describe these implementations in which follows.

5.1.1 *Using* BP *Representation.* We implemented the trie operations on top of the data structure for balanced parentheses of Navarro [2004, Section 6.1]. The trie operations are supported as explained in Section 2.3. Recall that in order to support the operations on BP we must support operations $findclose$, *excess*, and *enclose* on the sequence of balanced parentheses. Therefore, we do not store information to support $findopen$, thus saving space. Moreover, we do not need to support operation *parent* on *RevTrie*, and thus we do not store information to support *enclose* on the parentheses representing this trie.

Operation $child(x, \alpha)$ is supported by using $child(x, i)$, for $i = 1, 2, \ldots$, until finding the child labeled α. This is because the symbols labeling the children of x are scattered throughout array *letts* and must be found one by one using the operations on the parentheses. Whenever we need to support *rank* and *select* queries (this is, on top of the parenthesis sequences, to represent bitmap B in *RevTrie*, and for the permutation data structures), we use the data structure of González et al. [2005].

5.1.2 *Using* DFUDS *Representation.* The main idea of using the DFUDS representation for *LZTrie* is to reduce the time overhead for computing operation $child(x, \alpha)$ incurred by BP.

As done for BP, we represent the DFUDS sequence of *LZTrie* on top of the data structure for balanced parentheses of Navarro [2004]. Note that the DFUDS representation of a trie tends to have far matching parentheses, since every node is formed by a number of opening parentheses (indicating the degree of the node),

	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:	(	(	(	(	)	(	(	(	(	(	)	)	(	)	)	)	(	(	(	)	)	)	)	(	)	)	(	)	(	)	)	(	)	(	)	)
letts:		_	l	a		_	r	l	b	$			r				l	d	a					p			a		b			a		p		

Fig. 8. Illustration of our practical representation of array *letts* for the DFUDS representation. Notice that the labels of the children of a given node are stored in reverse order.

and only a few of these parentheses have the corresponding closing parentheses close enough so as not to be stored in the hash tables. Thus the hash tables tend to require more space under DFUDS. We also use the data structure of González et al. [2005] to support *rank* and *select* queries.

We study the way in which the trie operations are used by the LZ-index search algorithm in order to make this representation more efficient. For example, DFUDS introduces a heavier use of *rank* and *select* in its operations (see Section 2.4), according to the original definitions [Benoit et al. 2005]. However, many of these are redundant in the LZ-index (sometimes they are repeated twice in a given sequence of operations, as we will see below), and many others can be replaced by sequential scans over *par*, since the DFUDS position we are looking for should be not so far away from the current position. A list with the most important implementation details follows:

Supporting Operation $findopen$. Unlike the representation of LZ-index based exclusively in BP, with DFUDS we need to provide operation $findopen$ over the parentheses sequence. This is necessary, for example, to compute operation *parent* [Benoit et al. 2005] (recall also Section 2.4). Therefore, we need to add a second data structure, like the one used by Navarro [2004] (i.e., a hash table) to support $findopen$ and *enclose* operations. This adds extra space to DFUDS.

A Practical Representation of letts. Recall from Section 2.4 that the symbol by which a node x descends from its parent can be computed as $letts[rank_{(}(par, parent(x)) - 1 + childrank(x) - 1]$, which involves computing several *rank*, *select*, and parentheses operations. Although these can be computed in constant time in theory, we look for a more practical variant in practice.

This problem comes from the fact that there is an inverse relationship between the symbol labeling the i-th child of node x' of degree s (this symbol is originally stored at position i within the labels of children of x') and the opening parenthesis used to compute the DFUDS position of the i-th child of x' (recall that we use the $(s - i + 1)$-th opening parenthesis within the representation of x').

Therefore, we propose to represent *letts* by traversing the *LZTrie* in preorder and, for every node x reached, writing contiguously the symbols labeling the children of x, this time in *reverse order.* This means that the label of the last child appears in the first place, the label of the first child appears in the last place, and so on. See Fig. 8 for an illustration.

In this way there is a direct relationship between each of the opening parentheses defining a given node x' (which are used to find the children of x') and the labels of the children of x'. Thus, the label by which a given node x descends from its parent can be computed as $letts[rank_{(}(par, findopen(par, x-1))]$, avoiding the use of operation $childrank(x)$ as in Section 2.3, which involves more *rank* and *select* operations [Jansson et al. 2007].

Moreover, in many cases we can reuse the operation *findopen* computed to get the symbol. For example, in the LZ-index, most of the times that we need to know the symbol of a given node, we also need to go to the parent node, as for instance when computing `extract` and `display` operations, and when we use the *LZTrie* to descend in *RevTrie*. Then, after computing the corresponding symbol, we retain the position given by operation *findopen* in order to search for the parent of the current node, avoiding to repeat the same operation *findopen* when computing operation *parent*.

Since the labels of the children of node x are stored contiguously and sorted (yet in reversed order, according to our representation), array *letts* is stored in a plain way, without using any *rank* and *select* data structure, thus saving space. Operation $child(x, \alpha)$ is implemented by binary searching the list of labels of x. When the list is small (say, less than 10 elements) we perform a sequential scan on the symbols. This saves time compared with the BP representation, where we have to repeatedly use operation *findclose* on the parentheses in order to find the child we are looking for. With DFUDS, on the other hand, this work is done on the symbols, and then we map again to the DFUDS sequence to find the corresponding child (this involves only one *findclose* operation).

Avoiding Operation depth on DFUDS. The original DFUDS representation [Benoit et al. 2005] does not provide operation *depth*, which is latter supported by Jansson et al. [2007] in $O(1)$ time and requiring $o(d)$ extra bits of space. However, in the LZ-index we need to use *depth* in a very limited way, which helps us implement this operation simply and efficiently. We choose not to store any extra depth information, an thus we save space. The depth of a node could be computed by brute force, by successively going to the parent in *LZTrie* up to reaching the trie root. Therefore, we seek to avoid the computation of *depth* in the LZ-index search algorithm.

In the LZ-index search algorithm, we only need to use operation *depth* when locating occurrences of type 1 (see Section 3.3.1). Recall that in order to find occurrences of type 1 we must first search for P^r in *RevTrie*, getting node v_r. Then, for each node in the subtree of v_r we map to the corresponding node v_{lz} in *LZTrie*, and then we traverse the subtree of v_{lz} to report occurrences of type 1. The problem here is how to compute the offset of every occurrence within the corresponding phrase (in Section 3.3.1 we use operation *depth* to do that). However, note that the offset for node v_{lz} is m, since the phrase ends with P. Note also that the offset for the children of v_{lz} is $m+1$, and in general the offset of node v'_{lz} within the subtree of v_{lz} is $m+r$, where r is the difference of depths between v_{lz} and v'_{lz}.

So, instead of computing the depth of the nodes within the subtree of v_{lz}, which is expensive in our representation, we compute the offset of the nodes. This problem can be solved in a straightforward way in BP, since we perform a preorder traversal from v_{lz}, with initial offset m. We then increment the offset every time we enter a new subtree (which is indicated by '`(`' in BP), and decrement it when leaving a subtree (which is marked by '`)`' in BP). The preorder traversal is carried out by sequentially traversing the BP sequence and array *ids*, and not by using operation *child* on *LZTrie*. However, this is not so easy in DFUDS, since, for example, there is no clear marker of the end of a subtree (i.e., in a sequential scan on DFUDS there is no direct way to know whether a node is the last one in a given subtree).

Therefore, in the DFUDS representation we start a preorder traversal from node v_{lz}, and store the degree (number of children) of v_{lz} and its offset m in an initially empty stack. We then continue with the next node in preorder. At every step, the offset for the current node is computed as the offset of the parent node (which is the one at the top of the stack) plus one. Every time the degree stored in the top is (or becomes) 0 (leaves are a particular case), we pop it, and then decrement the degree of the node in the top, indicating that a new child of this node has been fully processed (this can eventually produce more pops when all of the children of the top node have been processed). As it can be seen, we are using the stack to know when the subtree of a node has been completely processed. This procedure ends when the stack becomes empty.

As in the case of BP, the preorder traversal is performed by a sequential scan on the DFUDS sequence, and the degree of every node x is computed by counting the number of opening parentheses in the representation of x.

This procedure could be also used to compute the depth of all nodes within the subtree of v_{lz}, by initializing the stack with the depth of v_{lz}, instead of storing the initial offset m. Note also we do not need to explicitly store depths or offsets, as they correspond to the stack height plus a constant.

Implementation of Operation degree. In our implementation we do not use the original definition for operation *degree*, which is based on operation *select* in order to find the next closing parenthesis which finishes the definition of the node. This allows us to count the number of opening parentheses defining the current node. In practice, in most cases this closing parenthesis is not so far away from the current position in DFUDS, except perhaps for the trie root. This is because in practice the tries tend to have high degrees only in the first levels. Thus, we explicitly store the degree of the root node, and for the rest of the nodes we perform a sequential scan on the DFUDS sequence. To avoid looking at every parenthesis in the process, we advance by machine words (in our experiments this shall mean 32 bits), until finding the first word containing a closing parenthesis. We represent opening parentheses with a 0, and closing ones with a 1. Thus, we advance while the corresponding word represents a 0, i.e. it stores only opening parentheses. However, in the case of very large alphabets the original definition of *degree* could be better, or we could replace the sequential search by an exponential search using the rank subdirectory.

5.2 Computing Text Positions

We add to our indices a data structure in order to transform pattern occurrences in the format D$t, offset$T into real text positions. We define array $TPos$ storing the absolute starting position (in the text) for the LZ78 phrases with identifier $i \cdot b$, for $i \geqslant 0$, for a total of $\frac{d}{b}\log n$ bits. We also define array $Offset$, storing in $Offset[i \cdot b + j]$, for $j \geqslant 0$, the offset (in number of text symbols) of phrase $i \cdot b + j$ with respect to phrase $i \cdot b$, requiring $d \log M$ extra bits of space, where M is the maximum number of text positions between two consecutive sampled phrases. If *LZTrie* has height $h = O(\log d)$ (which is true in practice with high probability [Knessl and Szpankowski 2000]), then $M = O(b \log d)$, and thus array

$Offset$ requires $O(d(\log b + \log\log d))$ bits. By choosing $b = \log n$ the total space requirement for both arrays is $d + O(d \log\log n) = o(n \log \sigma)$ bits.

Given an occurrence D$t, offset$T, the real text position for that occurrence can be computed in constant time as $TPos[\lfloor (t+1)/b \rfloor] + Offset[t+1] - offset$. In our experiments, we choose $b = 32$ (in a 32-bit machine) such that the division by b and taking the floor can be carried out efficiently by using shifts on machine words. For `extract` queries, we are given a text position from where to extract and we want to know the corresponding *LZTrie* node from where to start uncompressing the text. Therefore, given a text position *pos* we can obtain the phrase containing *pos* by first binary searching $TPos$, finding the greatest phrase $i \cdot b$ such that its position $TPos[i]$ is smaller or equal to *pos*. Then, we sequentially look in the corresponding segment of $Offset[i \cdot b..(i+1) \cdot b]$ for the greatest phrase t whose starting position does not exceed *pos*. Thus, the text position *pos* belongs to phrase t. The time for this operation is $O(\log n)$, because of the binary search on $TPos$ and the sequential search in the segment of $Offset$.

In Section 6.2 we will show the experimental space requirement of this data structure for a set of real-life texts we have used.

5.3 Supporting Partial `locate` Queries

In many applications it is quite common that we do not need to find all of the pattern occurrences, but just a few (arbitrary) of them. For this kind of applications, we design an algorithm to answer *partial* `locate` queries, where we are interested in locating just K arbitrary pattern occurrences. Our algorithm, which profits from the properties of the LZ-index in order to support fast searches, is as follows:

(1) Given a search pattern P, notice that a particular occurrence of it can be found by searching for P in *LZTrie*. In other words, P equals an LZ78 phrase, which is a particular case of occurrence of type 1, and can be found very fast in practice since this is better than using the slower *RevTrie* [Navarro 2009]. If P exists as a phrase, say corresponding to node v_{lz} in *LZTrie*, then all of the nodes descending from v_{lz} in the trie also correspond to occurrences of P, and can be used to answer the query. Thus, we traverse the subtree of v_{lz} and report every node found, as done for occurrences of type 1 (see Section 3.3.1). We stop the procedure as soon as we find K pattern occurrences.

(2) If the previous step was not enough to answer the query, and in case that P exists in *LZTrie*, then we map to the node corresponding to P^r in *RevTrie* (which exist for sure since P is an LZ78 phrase), and go on to locate the rest of occurrences of type 1 as usual. Otherwise, we delay occurrences of type 1 for a further step and go to the next step.

(3) In case P does not exist as an LZ78 phrase, we proceed with occurrences of type 2, trying to reuse as much as possible the work already done in step (1). We are thus delaying occurrences of type 1 for a further step. Let $P[1..i]$ be the longest proper prefix of the pattern that exists as an LZ78 phrase. Hence, $P^r[1..i]$ exists as a reverse phrase in *RevTrie*. Because of Property 2.3, every prefix of $P[1..i]$ also exists as a phrase in LZ78 (and the corresponding reverses exist in *RevTrie*). Then, to reuse the work already done when searching for P in *LZTrie*, we map to the *RevTrie* node corresponding to $P^r[1..i]$, which gives

us node v_r, and then search for $P[i+1..m]$ in *LZTrie*, to get node v_{lz}. We then search for occurrences of type 2 corresponding to the partition $P[1..i]$ and $P[i+1..m]$, using the nodes v_{lz} and v_r, in the usual way and stopping as soon as we find K occurrences. Note that by choosing the longest prefix $P[1..i]$ that exists in *LZTrie*, we are reducing as much as possible the length of the suffix $P[i+1..m]$ to be searched in *LZTrie*. If this was not enough to answer the query, we repeat the process for the occurrences of $P[1..i-1]$ and $P[i..m]$, in a similar way, and so on.

(4) We search for the remaining occurrences of type 2 (i.e., using those partitions of the pattern that were not tried in the previous step)

(5) Then, we continue with occurrences of type 1, as usual and just if P does not exist in *LZTrie* (i.e., these were not tried before).

(6) Finally we try occurrences of type 3.

We call *level 0* of the search to the step of searching for P in *LZTrie*, *level 1* the search for occurrences of type 1 (either at steps (2) and (5)), *level 2* the search for occurrences of type 2 (either at steps (3) and (4)), and *level 3* the search for occurrences of type 3 (step (6)). As it can be seen, we try to get fast access to the pattern occurrences, avoiding as much as we can the trie navigations, which become more expensive if we want to locate just a few occurrences. We shall use this approach also to support efficient `exists` queries (similar to $K = 1$).

6. EXPERIMENTAL RESULTS

We have now a number of practical reduced schemes for LZ-index, each one requiring a different amount of space. Hence given an amount of available storage, it is interesting to know which alternative is the best for that space. We hope that alternatives with more space are faster in practice, whereas the ones with less space will still be competitive against the best existing indices.

6.1 Experimental Setup

For the experiments of this section we used an Intel(R) Pentium(R) 4 processor at 3 GHz, about 4 gigabytes of main memory and 1024 kilobytes of cache, running version `2.6.13-gentoo` of Linux kernel. We compiled our algorithms with `gcc 3.3.6` using full optimization.

6.1.1 *Text Collections.* We test our indices in different practical scenarios, using the texts provided in the *Pizza&Chili* Corpus [Ferragina and Navarro 2005]:

—English Texts: although in many cases natural-language texts are searched for whole words or phrases, there are many other cases where a more powerful full-text search is needed. For the experiments with English text we use the file `http://pizzachili.dcc.uchile.cl/texts/nlang/english.200MB.gz` of 200 megabytes.

—DNA Sequences: nowadays, one of the main applications of full-text indexing is that of computational biology, in particular indexing DNA sequences. We test with the file `http://pizzachili.dcc.uchile.cl/texts/dna/dna.200MB.gz`, of 200 megabytes.

—MIDI Pitch Sequences: a very interesting application that has appeared in recent years is that of processing MIDI pitch sequences. In this case we test with the file of about 53 megabytes downloadable from `http://pizzachili.dcc.uchile.cl/texts/music/pitches.gz`.

—XML Texts: since XML is becoming the standard to represent semi-structured text databases as well as in many other applications, there exists the need of managing a huge amount of texts of this kind. We test with the XML file of 200 megabytes downloadable from `http://pizzachili.dcc.uchile.cl/texts/xml/dblp.xml.200MB.gz`.

—Proteins: another interesting application of text-indexing tools in biology is that of indexing and searching proteins. We use the file `http://pizzachili.dcc.uchile.cl/texts/protein/proteins.200MB.gz`, of 200 megabytes.

—Source Code: to test our indices in applications like software development, we use the source-code file `http://pizzachili.dcc.uchile.cl/texts/code/sources.200MB.gz`, of 200 megabytes.

6.1.2 *Comparison against Other Indices.* We compare our indices against the most efficient indices we are aware of, most of them available in *Pizza&Chili*:

Sadakane's Compressed Suffix Array (CSA). This index [Sadakane 2003] is a representative of the family of *compressed suffix arrays* [Grossi and Vitter 2005; Grossi et al. 2003; Sadakane 2003]. In practice it requires $nH_0(T) + O(n \log\log\sigma)$ bits of space, a counting time of $O(m \log n)$, a locating time of $O(\log^{1+\epsilon} n)$ per occurrence reported, and an extracting time $O(\ell + \log^{1+\epsilon} n)$ for any text substring of length ℓ, where $0 < \epsilon \leqslant 1$ is a constant parameter. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Compressed_Suffix_Array/sada_csa.tgz`. We have two parameters to set up for this index. The first one is the sample rate of suffix array positions (this information is used to speed up the locating and extracting operations), and the second one is the sample rate of the Ψ function. For `exists` and `count` queries we do not store any suffix array position, but only Ψ values. For `locate` queries, we used values of 4, 8, 16, 32, and 64 for suffix array positions, and the value 128 to sample the Ψ function (as this has shown to be the most efficient alternative [Ferragina et al. 2009]).

Alphabet Friendly FM-index (AF-FMI). This index [Ferragina et al. 2007] is based on the *backward-search* concept [Ferragina and Manzini 2005]. It has a space requirement of $nH_k(T) + o(n \log\sigma)$ bits, a practical counting time $O(m \log\sigma)$, a locating time $O(\log^{1+\epsilon} n)$ per occurrence reported, and an extracting time $O(\ell \log\sigma + \log^{1+\epsilon} n)$, for any constant $\epsilon > 0$, and any $k \leqslant \alpha \log_\sigma n$, where $0 < \alpha < 1$ is any constant. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Alphabet-Friendly_FM-Index/af-index_v2.tgz`. We have only one parameter to set up in the code, which is the sample rate of suffix array positions. We have used sample rates of one suffix array position stored out of 4, 8, 16, 32, and 64 text positions. For `exists` and `count` queries we do not store any suffix array position.

Succinct Suffix Array (SSA). This index [Mäkinen and Navarro 2005] is also based on backward search, but uses only one *wavelet tree* [Grossi et al. 2003], achieving $nH_0(T) + o(n \log \sigma)$ bits of space. The time complexities of this index are just as for the AF-FMI. We use the code provided at `http://pizzachili.dcc.uchile.cl/indexes/Succinct_Suffix_Array/SSA_v2.tgz`. We use the same parameters as for the AF-FMI.

Inverted LZ-index (ILZI). This index [Russo and Oliveira 2008] is based on Lempel-Ziv compression, using a variant of the LZ78 parsing called *maximal parsing*, which has many interesting properties. The ILZI requires at most $(5+\epsilon)nH_k(T) + o(n \log \sigma)$ bits of space, and has a locating complexity of $O((m + occ) \log n)$. We use the implementation by Luis Russo, which does not conform the *Pizza&Chili* API as the other indices. In particular, pattern occurrences are reported in the format D$t, offset$T, just like for the original LZ-index. The index does not include the data structure to transform those occurrences into real text positions, as our implementations do. To be fair, we sum the space of the described data structure for text positions to the space of this index. We also sum the average time to transform occurrences into real text positions for `locate` queries. According to our experiments, this is 1.3 microseconds per occurrence. As an additional consequence, this index does not provide `extract` queries, but only `display` queries. So, we are not able to extract arbitrary text substrings.

It is important to note also that the current implementation of the ILZI does not return to the invoking application an array with the pattern occurrences, as required by the *Pizza&Chili* API. This implementation just prints the number of phrases containing the starting position of the occurrences. We get rid of the print operation in the code, and thus there is no any reporting operation (we just find the occurrences). In particular, we are not accounting for the overhead of managing the occurrence array, which grows dynamically as more occurrences are found.

Russo and Oliveira [2008] define a practical variant of LZ78 parsing, the so-called *LZ78 maximal parsing with quorum l*. The idea is that for every phrase $B_i = B_j \cdot c$, B_j is the longest prefix of the rest of the text that appears at least $l + 1$ times in $B_0 \ldots B_{i-1}$. Note that by using $l = 0$ we get the original LZ78 parsing. In this way, by using larger quorum values we can reduce the number of phrases in the LZ78 maximal parsing, hence reducing the number of nodes in the trie representing those phrases, and thus reducing the space of the index. We use quorum values $l = 0$, 1, 2, 4, 8, and 16 to get different space/time trade-offs, though this is not actually a trade-off parameter, but an optimization parameter, as we shall see in our experiments. Smaller values of l do not yielded a significant reduction in the space requirement.

6.2 Comparison of Space Requirement and Construction Time

In Table III we show the construction time and final space requirement for the indices we have tested. For the permutation data structures used in Schemes 3 and 4, we use $1/\epsilon = 1$, 2, 3, 4, 5, 10, and 15. As it can be seen, we have reduced the space of the original LZ-index, and in the case of Schemes 3 and 4 we offer a space/time trade-off. Note that the maximum space requirement of Scheme 3 (and also Scheme 4) is about the same as that of the original LZ-index, and that the minimum space requirement we achieve is in all cases around 66% the space of the original LZ-index. In many cases, such as for XML documents and DNA data, we are able to provide indexing capabilities with a representation that is smaller than the original text.

In Table IV we show the breakdown of the space requirement of the data structures that form the LZ-index and their associated $o(\cdot)$ terms. Components *ids* and *rids* make up the $2(1+\epsilon)nH_k$ term of the space complexity, for ϵ going from 0 to 1. The other components correspond to the $o(n \log \sigma)$ term, which in practice adds up to 25%–40% of the total space. This shows, once more, that those asymptotically vanishing terms are important in practice.

We also conclude that our indices are much faster to build than competing schemes. As a comparison between LZ-based schemes, the construction time of the ILZI (which is in most cases the slowest index to build) ranges from about 3 times slower than our schemes (in the case of DNA text), to about 9 times slower (in the case of English text). One reason for this is that our indices are constructed by performing only one pass over the text, while the ILZI needs two passes [Russo and Oliveira 2008]. Recall that the ILZI does not construct any text-position data structure, which would further increase its construction time.

It can be argued that construction time is not so important in indexed text searching, where one constructs the index once and queries it several times, so construction time is amortized over a number of queries. However, in the cases in which we deal with large texts (because we would use a classical index otherwise), construction time is not irrelevant.

6.3 Comparison of Search Time

Next we experimentally test whether the trade-offs we provide are competitive for compressed text searching. In our experiments, we call S2 DFUDS the version of Scheme 2 implemented on DFUDS, and S3 DFUDS the DFUDS version of Scheme 3. We do not include Scheme 1 based on DFUDS, since it is outperformed by S2 DFUDS, both of them requiring about the same space. We no dot include Scheme 4 in our plots, since Scheme 3 outperforms it in most cases (though they require almost the same space). However, Scheme 4 is interesting by itself since it can be reduced to $(1+\epsilon)uH_k(T) + o(u \log \sigma)$ bits of space [Arroyuelo et al. 2006; 2010], which cannot be achieved by other schemes.

6.3.1 `Extract` *Queries.* We extract 10,000 random snippets, each of length 100. We measure the time per symbol extracted, so we average over a total of 1 million extracted symbols.

In Fig. 9 we show the experimental results. In most cases we are able to extract about 1 to 2 million symbols per second, being about twice as fast as the most competitive alternatives. In the particular case of DNA text, the SSA excels, since it extracts each symbol in time $O(\log \sigma)$. However, we outperform the SSA as soon as we are able to maintain Scheme 1 in main memory (which in the case of DNA requires about 1.03 times the space of the uncompressed text).

This shows the superiority of our LZ-indices in this important aspect. Also, we can see that our approach to reduce the space of the LZ-index is effective in this case, since we are able to reduce the space while still maintaining a good extracting performance.

Table III. Space requirement and construction time for the different indices we have tested. The space is shown as a fraction of text size. Indexing time is shown in megabytes per second.

Text	Index	Space requirement (fract. of text size)	Indexing speed (MB per second)
English	CSA	0.43 – 1.50	0.45 – 0.44
	SSA	0.87 – 2.87	0.93 – 0.90
	AF-FMI	0.65 – 2.65	0.26
	ILZI	1.23	0.15
	Original LZ-index	1.69	1.47
	Scheme 1	1.39	1.30
	Scheme 2	1.38	1.27
	Scheme 3	1.13 – 1.69	0.91 – 1.33
	Scheme 4	1.13 – 1.69	0.71 – 1.31
DNA	CSA	0.46 – 1.53	0.51
	SSA	0.50 – 2.50	1.33 – 1.28
	AF-FMI	0.48 – 2.48	0.43
	ILZI	0.95	0.66
	Original LZ-index	1.24	2.35
	Scheme 1	1.03	2.02
	Scheme 2	1.01	1.99
	Scheme 3	0.83 – 1.24	1.36 – 2.12
	Scheme 4	0.83 – 1.24	1.03 – 2.06
MIDI Pitches	CSA	0.62 – 1.68	0.94 – 0.92
	SSA	1.04 – 3.04	1.80 – 1.71
	AF-FMI	0.93 – 2.94	0.36
	ILZI	1.86	0.24
	Original LZ-index	2.58	1.76
	Scheme 1	2.16	1.49
	Scheme 2	2.12	1.47
	Scheme 3	1.76 – 2.58	0.88 – 1.58
	Scheme 4	1.76 – 2.58	0.62 – 1.56
XML	CSA	0.29 – 1.35	0.68
	SSA	0.98 – 2.98	1.34 – 1.29
	AF-FMI	0.54 – 2.54	0.44 – 0.43
	ILZI	0.61	0.34
	Original LZ-index	0.93	2.38
	Scheme 1	0.77	2.15
	Scheme 2	0.76	2.15
	Scheme 3	0.63 – 0.93	1.57 – 2.23
	Scheme 4	0.63 – 0.93	1.24 – 2.17
Proteins	CSA	0.67 – 1.73	0.57 – 0.56
	SSA	0.82 – 2.82	0.97 – 0.95
	AF-FMI	0.82 – 2.82	0.38 – 0.37
	ILZI	1.73	0.24
	Original LZ-index	2.40	1.55
	Scheme 1	1.99	1.29
	Scheme 2	1.96	1.25
	Scheme 3	1.62 – 2.40	0.79 – 1.35
	Scheme 4	1.62 – 2.40	0.58 – 1.30
Sources	CSA	0.38 – 1.44	0.76 – 0.75
	SSA	1.01 – 3.01	1.37 – 1.32
	AF-FMI	0.73 – 2.73	0.30
	ILZI	1.15	0.17
	Original LZ-index	1.67	1.73
	Scheme 1	1.39	1.54
	Scheme 2	1.37	1.51
	Scheme 3	1.13 – 1.67	1.04 – 1.59
	Scheme 4	1.13 – 1.67	0.79 – 1.54

Table IV. Space overhead introduced by the different data structures that form our Scheme 4, in bits per symbol.

Text	Data structure	Total space	Space overhead ($o(\cdot)$ term)
English	par	0.107	0.081
	$letts$	0.102	–
	ids	0.358–0.638	0.005
	$rpar$	0.049	0.023
	B	0.018	0.005
	$rids$	0.358–0.638	0.005
	$TPos$	0.140	–
DNA	par	0.067	0.048
	$letts$	0.078	–
	ids	0.263–0.468	0.004
	$rpar$	0.041	0.021
	B	0.014	0.004
	$rids$	0.263–0.468	0.004
	$TPos$	0.110	–
MIDI Pitches	par	0.190	0.158
	$letts$	0.160	–
	ids	0.512–0.909	0.007
	$rpar$	0.069	0.028
	B	0.028	0.007
	$rids$	0.512–0.909	0.007
	$TPos$	0.270	–
XML	par	0.062	0.047
	$letts$	0.056	–
	ids	0.189–0.337	0.003
	$rpar$	0.046	0.031
	B	0.010	0.003
	$rids$	0.189–0.337	0.003
	$TPos$	0.090	–
Proteins	par	0.171	0.135
	$letts$	0.143	–
	ids	0.503–0.896	0.007
	$rpar$	0.063	0.027
	B	0.025	0.007
	$rids$	0.503–0.896	0.007
	$TPos$	0.220	–
Sources	par	0.126	0.101
	$letts$	0.098	–
	ids	0.344–0.613	0.005
	$rpar$	0.048	0.023
	B	0.017	0.005
	$rids$	0.344–0.613	0.005
	$TPos$	0.160	–

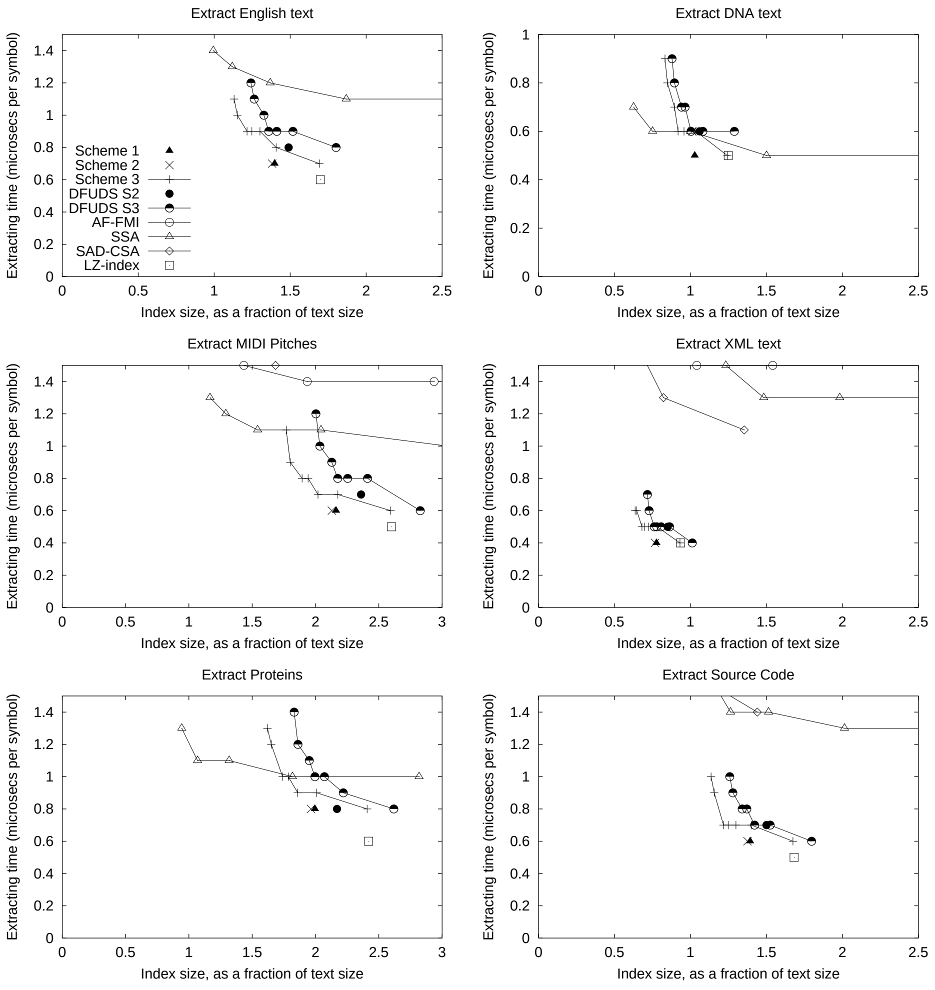


Fig. 9. Experimental extracting time, for random snippets of length $\ell = 100$. Times are measured in microseconds per symbol extracted. Missing values are outside the plot range.

6.3.2 `Display` *Queries.* We search for 5 million pattern occurrences and then show a context of 50 symbols surrounding every occurrence, for patterns of length 10 (in other words, we display 110 symbols per occurrence).

In most indices, $\texttt{display}(P, \ell)$ queries can be thought of as a $\texttt{locate}(P)$ query (in order to find the pattern occurrences) followed by an $\texttt{extract}(i, j)$ query (where i and j are computed by means of the positions obtained with `locate` and the context length ℓ). In our LZ-indices, however, we originally get the occurrences in the format $\mathrm{D}t, offset\mathrm{T}$, to finally transform t and $offset$ into a text position (by means of the data structure described in Section 5.2). This text position must be transformed by `extract` again into an LZ78 phrase (recall that this involves binary searching the text-position data structures), from where we start the extraction of text in *LZTrie*. To avoid repeating this work, we do not transform the occurrences into text positions when performing `display` queries. We rather display text with a simplified version of `extract` that works on LZ78 phrases rather than on text positions.

In Fig. 10 we show the experimental results. The current implementation of the ILZI shows only a context preceding the pattern occurrences, and not surrounding the occurrences as other schemes do. For our LZ-indices, showing a context surrounding the occurrences (which is usually required) introduces the use of extra operations which are not needed when showing a context preceding an occurrence. As it can be seen, just like for `extract` queries, our indices are among the most competitive schemes for displaying text, in many cases outperforming the ILZI.

6.3.3 `Locate` *Queries.* We search for 10,000 patterns extracted at random positions from the text, with length 5, 10, and 15. For short patterns, we limit the total number of occurrences found to 5 million. We measure the time in microseconds per occurrence found.

Partial `locate` *Queries.* This is a challenging problem for our LZ-indices since, for example, the indices based on suffix arrays are very efficient to find the suffix-array interval containing the occurrences, and hence they are rapidly ready to start locating the occurrences. The ILZI has also a very fast $O(m)$ trie navigation before starting the locating procedure.

We test here our algorithm defined in Section 5.3 (recall that we divide the search process into four levels, level 0 up to 3). In Table V we show the percentage of occurrences that are found in each level of search, for $K = 1$ and for the different text collections. Notice that the search of level 0 (i.e., searching for P as a whole phrase in *LZTrie*) is very effective for patterns of length 5 to 10 (in the case of DNA, for example, almost 100% of the queries can be answered at level 0). Notice also that the percentage found at level 1 is relatively small compared to the corresponding percentage of level 0. Recall that with the search of level 0 we look for a particular case of occurrences of type 1. This means that most of the times a pattern exists as an occurrence of type 1, this can be found at level 0 of the search. For longer patterns, $m = 15$, there is a smaller probability of finding the occurrence contained in a single phrase, and thus the percentage found at level 0 is smaller.

In Figs. 11 up to 14 we show the experimental result for values $K = 1$ and 5, and for $m = 5$ and 10. We do not show the results for $m = 15$ since they are poorer, as predicted by the results of Table V.

For $m = 5$ and $K = 1$, the most interesting results are obtained in the cases of English text, DNA, XML, and proteins, though in the latter case our indices do not obtain good compression. For DNA, S3 DFUDS is unbeatable since, as we have seen in Table V, 100% of the occurrences are found at level 0 of the search. Notice that for MIDI pitches our performance is not so competitive (except perhaps for S2 DFUDS), as it was also predicted by Table V: only 73% of the occurrences are found at level 0, and thus we need more trie navigations.

Except for MIDI pitches, in all other cases the best alternatives are the indices based on Lempel-Ziv compression: ours and the ILZI (the best in each case depends on the space usage, as it can be seen). This shows that Lempel-Ziv-based indexing is a very competitive choice in general, despite that suffix-array-based indices basically

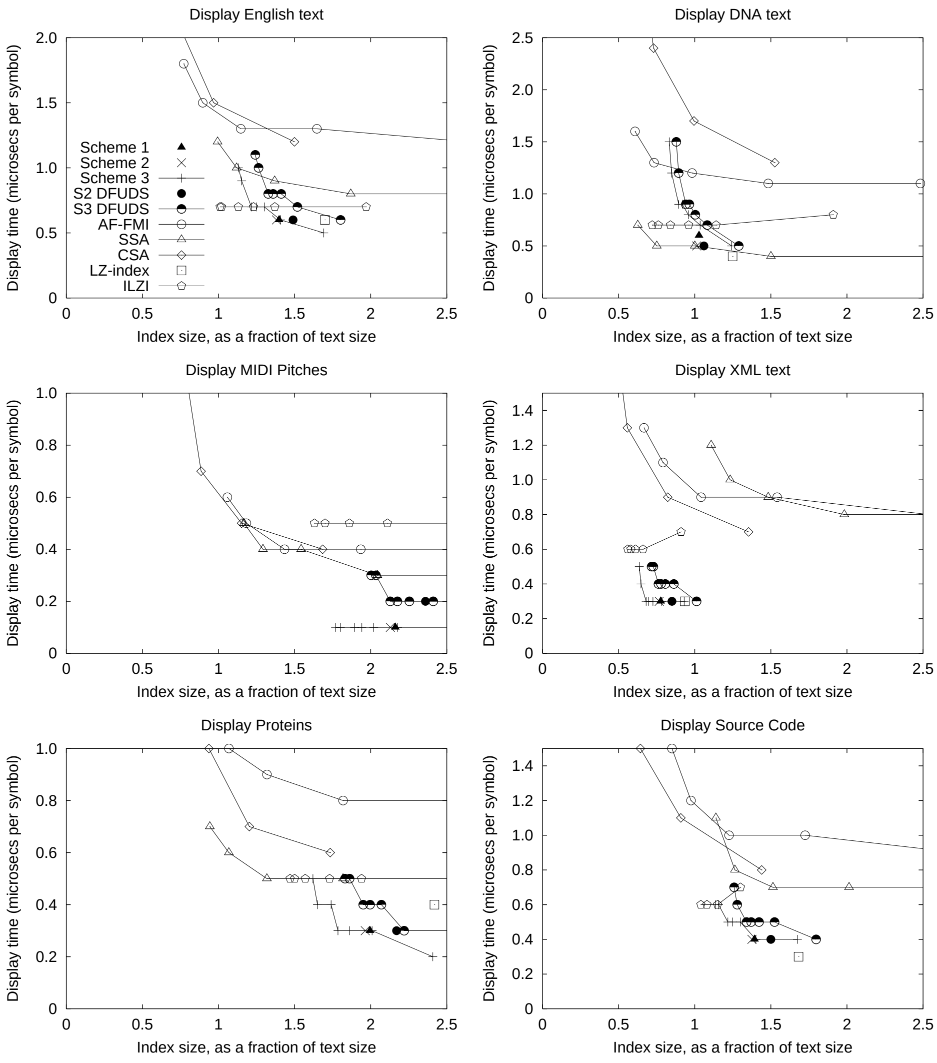


Fig. 10. Experimental display time, for snippets of length $\ell = 110$ around every pattern occurrence. Times are measured in microseconds per symbol extracted.

compute the suffix-array interval containing the occurrences, and then ask the index to obtain just one of these occurrences. As we shall see later with `count` queries, these indices are very efficient to find the suffix-array interval; however, asking them to obtain just one occurrence makes them significantly less competitive.

Notice also that our indices based on DFUDS outperform (in most cases by far) our indices based on BP. This is because the fast navigation on the tries becomes a fundamental aspect, since we are reporting just a few occurrences. For this reason, and in order to make our plots clearer, we do not show the results for Scheme 1, Scheme 2, and the original LZ-index.

As we increase the number of occurrences to locate, in principle the trie navi-

Table V. Percentage of patterns found at each level of search, for partial `locate` queries with $K = 1$ and for different pattern lengths. Level 0: P is found as an LZ78 phrase in *LZTrie*; Level 1: P is found as occurrence of type 1, but not as a whole phrase; Level 2: P is found as occurrence of type 2; Level 3: P is found as occurrence of type 3.

Text	Level	Percentage per level		
		$m = 5$	$m = 10$	$m = 15$
English	0	98.64	55.86	7.80
	1	0.54	6.27	1.83
	2	0.82	34.97	63.14
	3	0.0	2.90	27.23
DNA	0	100.0	99.26	11.63
	1	0.0	0.33	1.06
	2	0.0	0.41	78.74
	3	0.0	0.0	8.57
MIDI Pitches	0	72.28	20.01	9.69
	1	3.69	2.10	1.38
	2	23.82	42.66	19.67
	3	0.21	35.23	69.26
XML	0	95.41	65.61	37.56
	1	2.59	11.53	15.53
	2	1.99	19.55	32.13
	3	0.01	3.31	14.78
Proteins	0	98.96	13.29	8.51
	1	0.71	1.41	1.29
	2	0.33	63.87	11.23
	3	0.0	21.43	78.97
Sources	0	94.31	48.40	21.41
	1	2.73	9.34	6.01
	2	2.94	37.15	44.11
	3	0.02	5.11	28.47

gations are amortized by reporting more occurrences. However, our indices may need to go on more levels of the search, which means more navigations on the tries. Therefore, the total cost depends on the number of occurrences found in each search level. In general, as K grows, it becomes more difficult to compete since occurrences at higher levels are more expensive to obtain. Yet, we still provide some interesting cases for $K = 5$ and $K = 10$ (the latter case is not shown; results are close to those for $K = 5$).

The trie traversals also raise when we increase the pattern length, as it can be seen for $m = 10$.

Thus, we conclude that our technique for solving partial `locate` queries of Section 5.3 is relevant, and more efficient when the probability of finding the occurrences at level 0 of the search is high, as for example when we search for short patterns, the alphabet is small, or we look for very few occurrences (e.g., $K = 1$).

Full `locate` *Queries.* We test here `locate` queries without limiting the number of occurrences. In Fig. 15 we show the experimental results for patterns of length $m = 5$. As we can see, there is no clear winner in all cases, but the performance

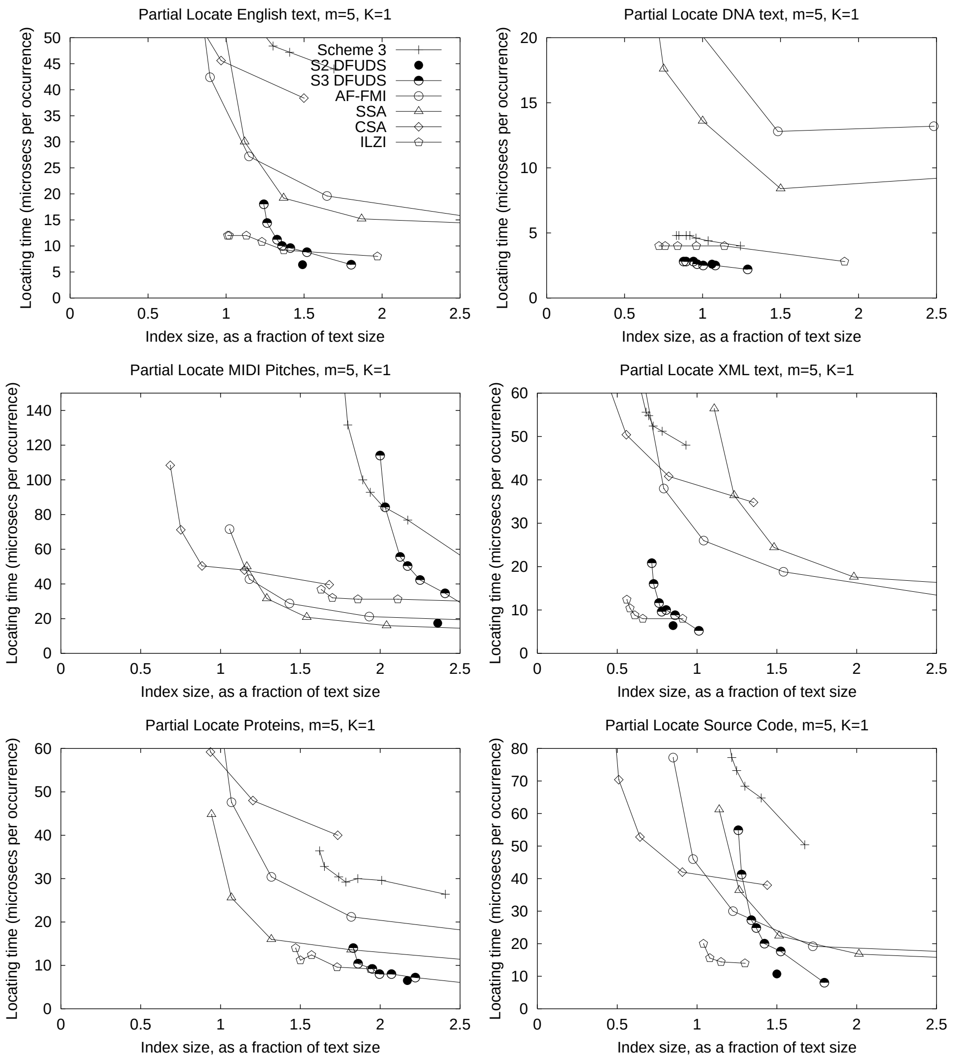


Fig. 11. Experimental time for partial `locate` queries, for patterns of length $m = 5$ and retrieving just $K = 1$ occurrence. We measure the time in microseconds per occurrence reported.

depends on the available space. However, we can see some clear performance patterns: in most cases our schemes outperform all competing schemes (including the very competitive ILZI) as soon as we have space available to store, at least, Scheme 2. When we reduce the space usage of the index, however, the ILZI outperforms Scheme 3, yet the latter is still competitive (in most cases outperforming the competitive CSA). In general, for `locate` queries with short patterns the superiority of Lempel-Ziv-based indices is clear.

Notice also that, in most cases, Scheme 2 outperforms Scheme 1, both requiring about the same space. This means that the shorter average path length of Scheme 2 is better than having direct access from *RevTrie* to *LZTrie* nodes as in Scheme

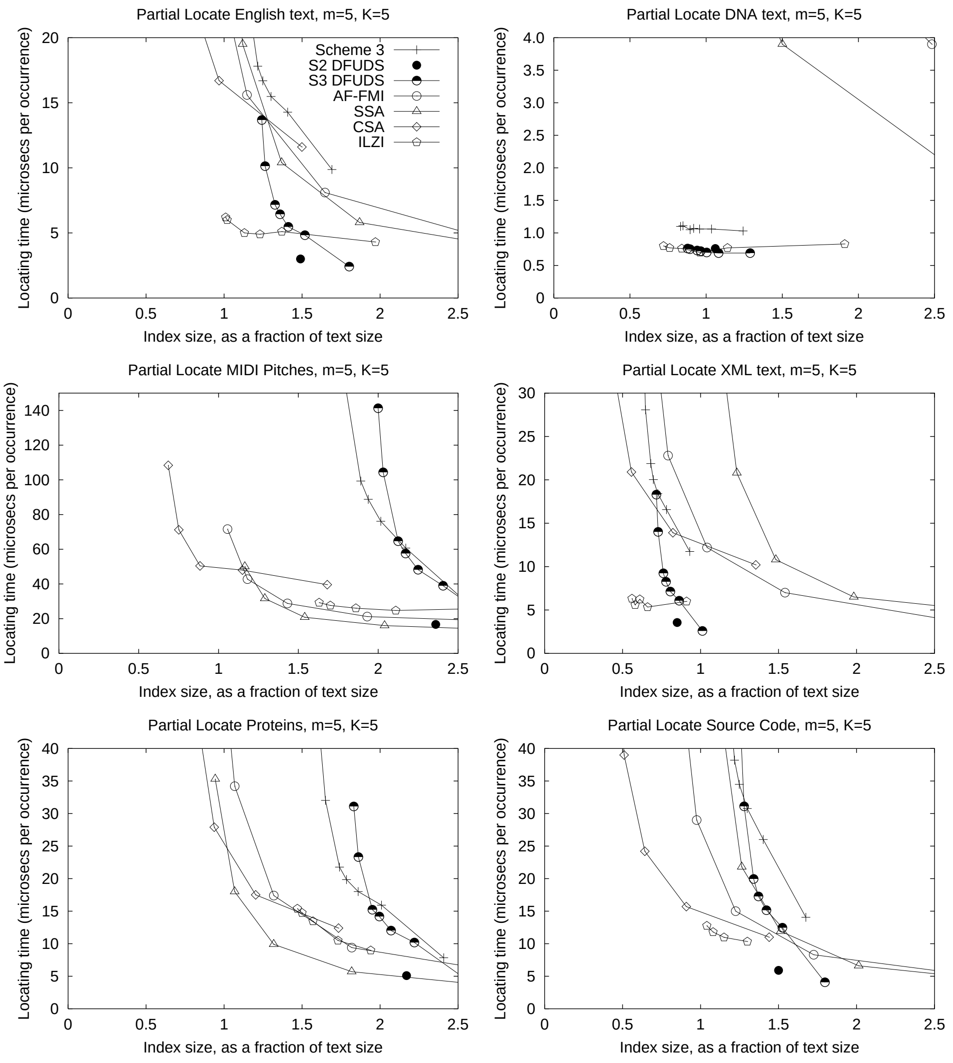


Fig. 12. Experimental time for partial `locate` queries, for patterns of length $m = 5$ and retrieving just $K = 5$ occurrences. We measure the time in microseconds per occurrence reported.

1, which is good only for occurrences of type 1. As we said before, occurrences of type 2 are more costly in Scheme 1. It is also important to note that the DFUDS versions of LZ-index have a performance which is similar to the LZ-index based on BP. Notice, however, that DFUDS requires more space than BP, as it was predicted in Section 5.1.2.

In the particular case of XML text, we can see a very important aspect that distinguishes LZ-indices from indices based on suffix arrays. The latter need to store extra non-compressible information (the sampled suffix array positions) to efficiently carry out `locate` and `extract` queries. The extra data stored by LZ-indices, on the other hand, is largely compressible, for example the size of the

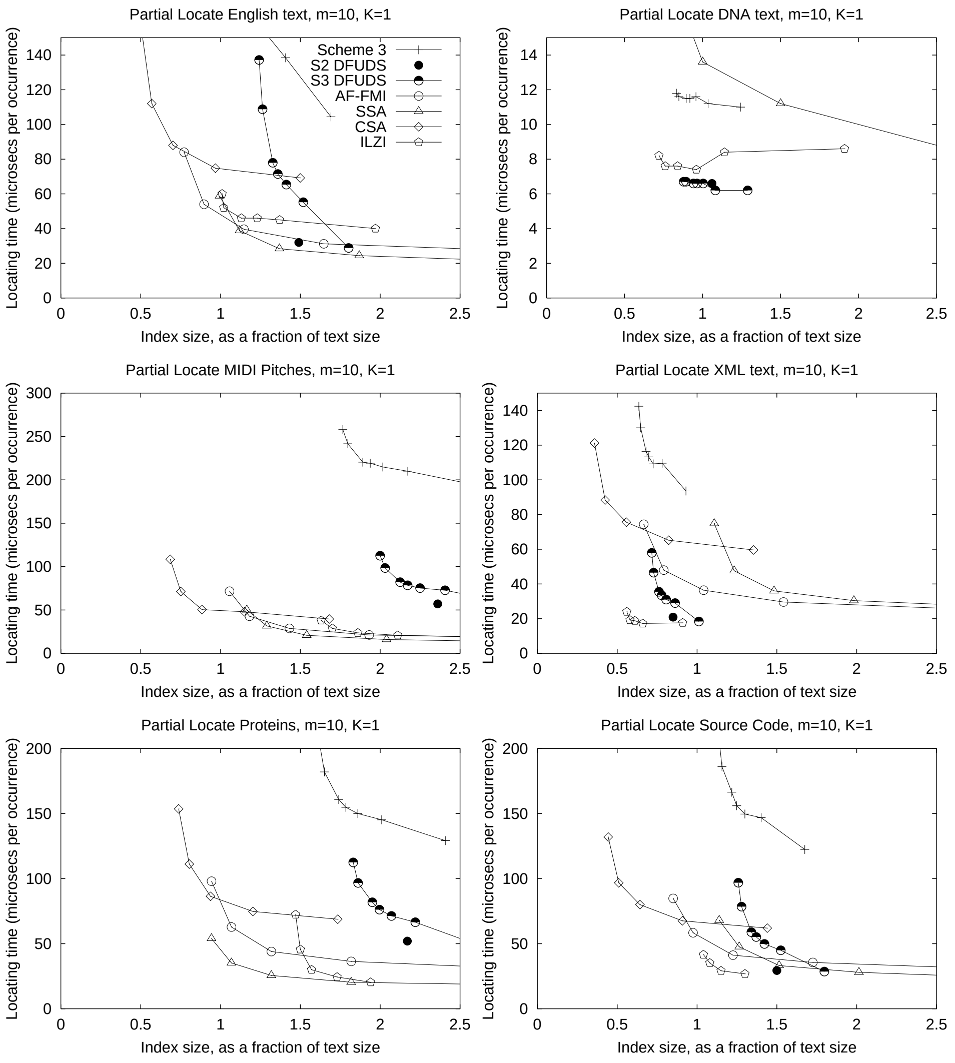


Fig. 13. Experimental time for partial `locate` queries, for patterns of length $m = 10$ and retrieving just $K = 1$ occurrence. We measure the time in microseconds per occurrence reported.

arrays for which we sample the inverse-permutation information depends on n, the number of LZ78 phrases of T. Therefore, when the texts are highly compressible, the LZ-indices can be smaller and faster than alternative indices.

For proteins, as an opposite case, our LZ-indices are larger and slower. They are large since the text is not as compressible as others, which can be also noted in the size of competing schemes. Our LZ-indices are in addition slower in this case, since each search retrieves on average only a few patterns, and therefore the work on the tries is not amortized by reporting many occurrences.

We show the results for patterns of length $m = 10$ in Fig. 16. As we can see, our indices are still competitive, yet not as significantly as in the previous case

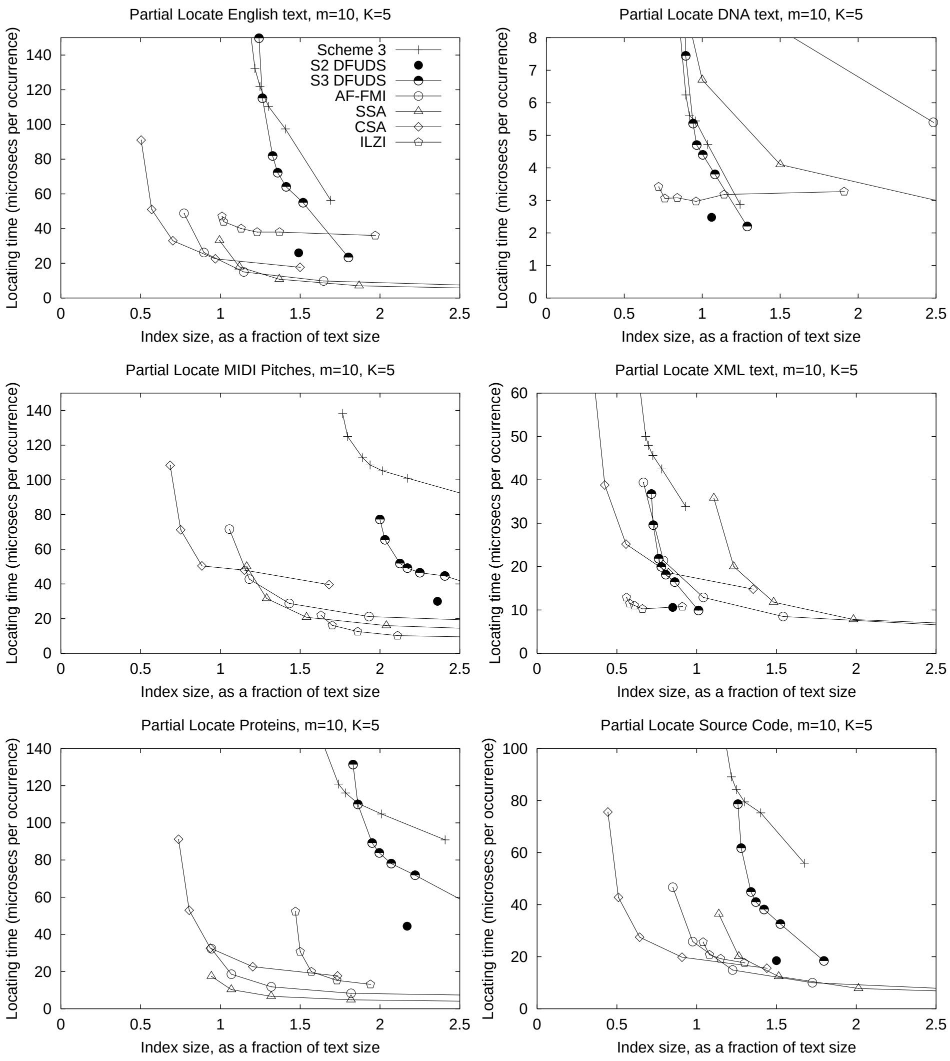


Fig. 14. Experimental time for partial `locate` queries, for patterns of length $m = 10$ and retrieving just $K = 5$ occurrences. We measure the time in microseconds per occurrence reported.

where $m = 5$. The DFUDS implementation of LZ-index outperforms BP in all cases, except for MIDI pitches, XML text, and source code, where these have about the same performance. In particular for proteins and English text, where the number of occurrences per pattern is relatively small, the difference is greater for DFUDS, since the cost of navigating the tries becomes predominant. It is important to note also that Scheme 2 (both for BP and DFUDS implementations) is interesting (in some cases the best) alternative, for the memory space it requires. For patterns of length $m = 15$ (figure omitted), the behavior of the indices is similar to that of length $m = 10$.

Our results indicate that our LZ-indices, usually offer the best space-time trade-

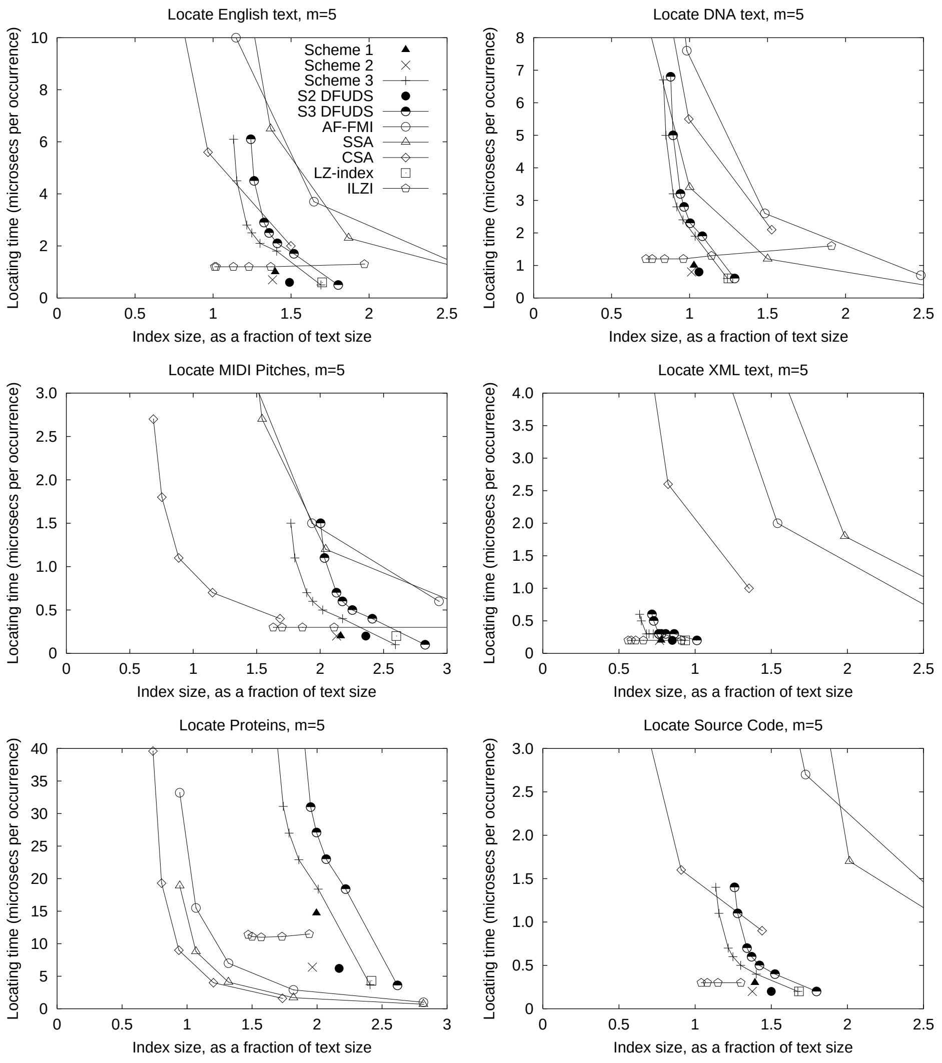


Fig. 15. Experimental locating time, for patterns of length $m = 5$. We measure the time in microseconds per occurrence reported.

off when the space required by Scheme 2 can be accomodated in main memory. In particular, our LZ-indices excel for full `locate` when the total number of occurrences to report is high. For long patterns, the number of occurrences becomes smaller, and therefore the time of searching for the pattern substrings in the tries dominates. The reduced trie navigation time offered by the DFUDS implementation becomes relevant in this case. Yet DFUDS requires more space.

6.3.4 `Count` *Queries.* We search for patterns of length 20 extracted at random positions from the text. We measure the times per symbol of the pattern. As competing schemes based on suffix arrays do not store suffix-array sampling in-

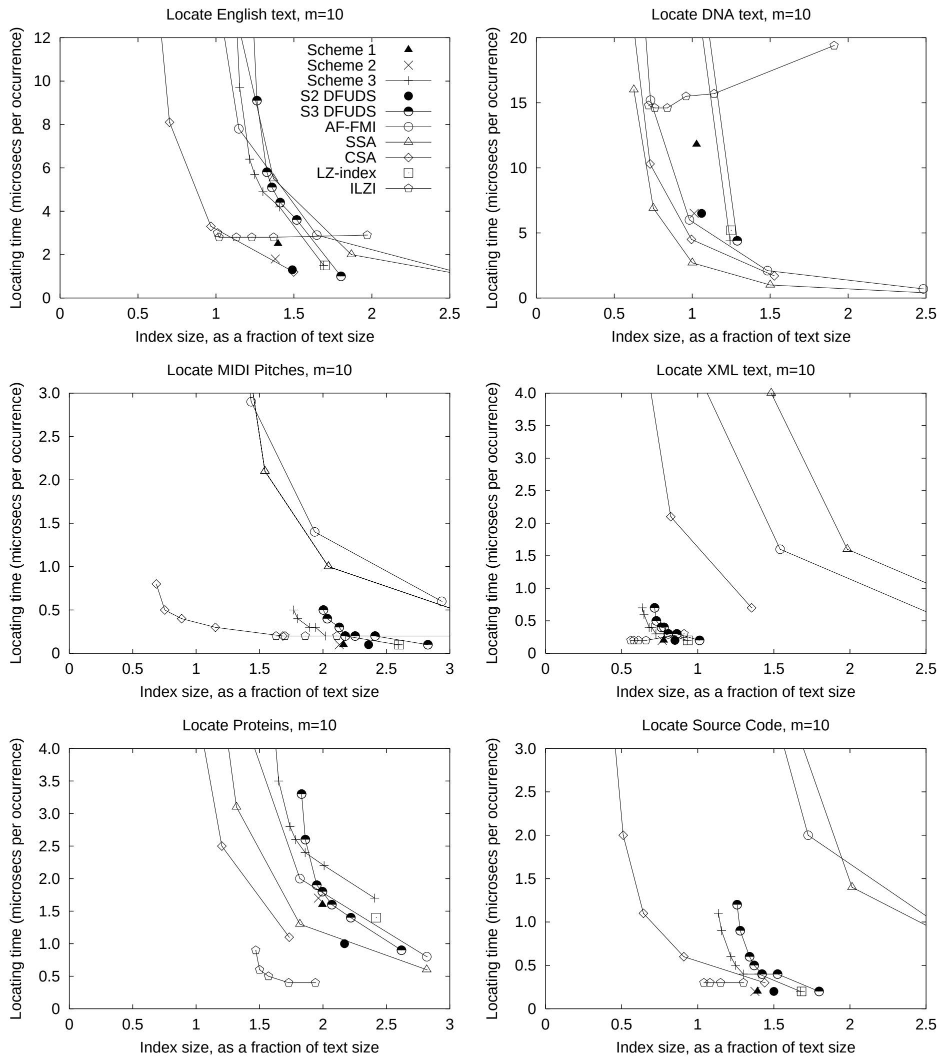


Fig. 16. Experimental locating time, for patterns of length $m = 10$. We measure the time in microseconds per occurrence reported.

formation in order to count (and thus are able to support efficiently only `count` queries), to be fair in this case we do not store the data structure for text positions in our LZ-indices. Yet, note that, within this space, we are still able to support fast `locate` queries (though without reporting text positions), and `display` queries.

The experimental results for `count` queries are shown in Fig. 17. As we can see, our schemes can implement this query, yet they cannot compete against the indices based on suffix arrays, since the counting complexity of these indices is related to the pattern length, and not to the number of pattern occurrences. Our schemes basically need to locate the pattern occurrences in order to count them. We can also see that the DFUDS implementation of LZ-index outperforms in all cases the BP

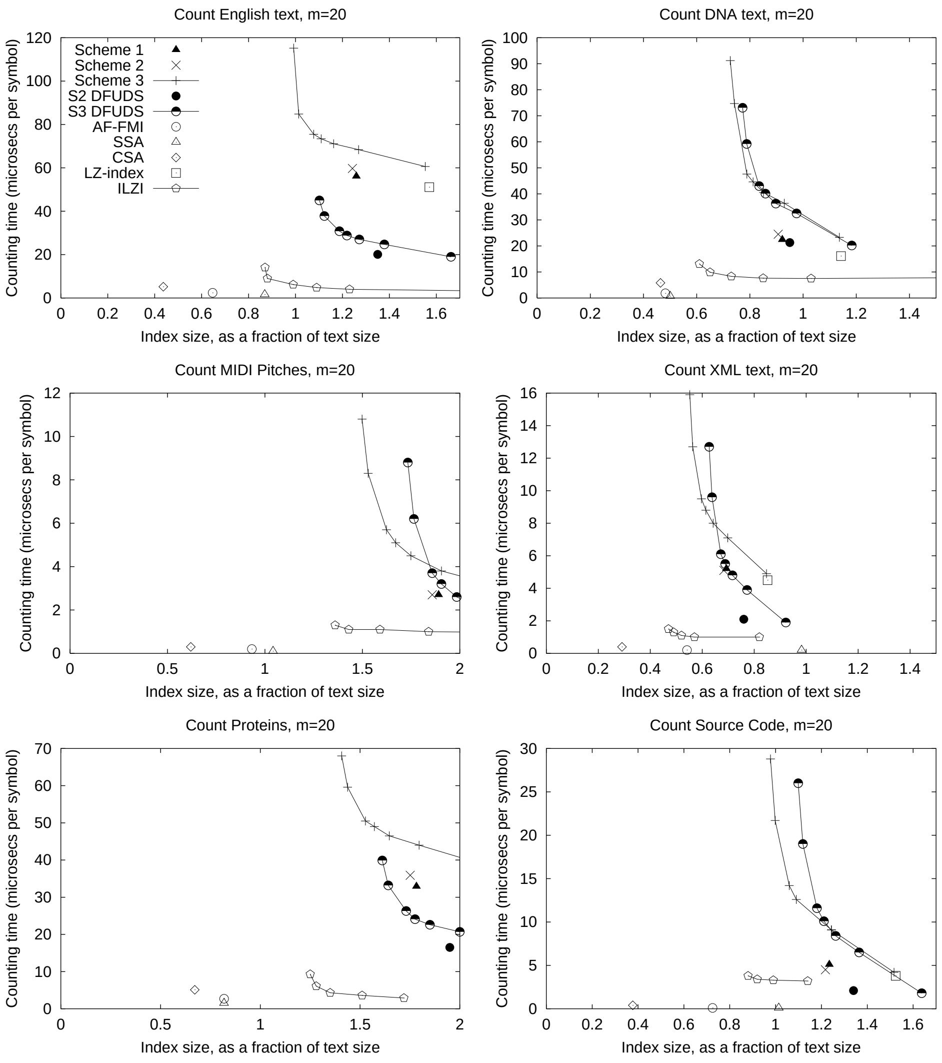


Fig. 17. Experimental counting time, for patterns of length $m = 20$. We measure the time in microseconds per symbol of the pattern.

implementation, yet the former requires slightly more space. This is mainly because we are searching for long patterns, and DFUDS provides more efficient descent in the *LZTrie*.

6.3.5 `Exists` *Queries.* For this kind of queries, indices based on suffix arrays basically need to count the number of occurrences, since they first search for the pattern, and then check whether the suffix-array interval they get is empty or not. Despite that our LZ-indices are not competitive for `count` queries, we show here that they are much more efficient for finding the first occurrence of a pattern, which is useful to support `exists` queries (indeed, this has been already shown in the experiment with partial `locate` queries). The key is that we do not necessarily need to count the occurrences, but just to find the first pattern occurrence as fast as we can. We test with patterns of length 5, 10, and 15, and search for 10,000 patterns that exist in the text. We implement existential queries in our indices using the idea of partial `locate` queries, explained in Section 5.3.

In Fig. 18 we show the experimental results for `exists` queries, for patterns of length 5. Excluded plots for patterns of length $m = 10$ produced similar results, while those for patterns of length $m = 15$ gave worse results (this is because, as predicted in Table V, the heuristic of level 0 is not so effective for longer patterns). As it can be seen, our indices are much more competitive than for `count` queries, showing that our approach of first looking for P in *LZTrie* is effective in practice. Our indices achieve the same times (though not the same space) in many cases. As in the case of `count` queries, our indices are larger than competing schemes, yet ours are able to support more than just `count` and `exists` queries within this space.

7. CONCLUSIONS

We have introduced practical approaches to reduce the space requirement of the LZ-index of Navarro [2004; 2009]. Given a text $T[1..n]$ over an alphabet of size σ, with k-th order empirical entropy $H_k(T)$ [Manzini 2001], the original LZ-index of Navarro [2004] requires $4nH_k(T) + o(n \log \sigma)$ bits of space (in practice, 5 times the size of the compressed text). In this paper we define several new versions of the LZ-index, requiring $3nH_k(T) + o(n \log \sigma)$ and $2(1+\epsilon)nH_k(T) + o(n \log \sigma)$ bits of space, for $0 < \epsilon < 1$. The latter ones allow us to partially overcome one of the drawbacks of the original LZ-index: the lack of space/time tuning parameters. Although our schemes do not provide worst-case guarantees at search time, they ensure $O(\frac{1}{\epsilon}(m \log n + occ\, \sigma^{m/2}))$ average time for locating the occurrences of pattern $P[1..m]$ in T.

We implemented and extensively tested our indices in many practical scenarios, which cover an interesting range of applications, comparing against the best alternative compressed full-text self-indices. From those experiments we conclude that we can effectively reduce the space requirement of the original LZ-index, by a factor of about 2/3. This means in practice about 3 times the size of the compressed text, and about 2.5 times the size of the smallest alternative compressed index. We also noted that our indices are the fastest to build, which is important when we deal with large texts.

When comparing the search performance, we concluded that our indices offer an interesting alternative in practice, for different types of queries: our indices are in most cases the best alternatives for `extract` and `display` queries, which we argue are the most basic queries in the scenario of compressed full-text self-indices, where the text is not available otherwise[3]. For instance, in most cases we are able to extract about 1–2 megabytes of the text per second, being about twice as fast as the most competitive alternatives. Thus, we can reduce the space, still maintaining

[3]In the literature [Navarro and Mäkinen 2007] the `count` operation is taken as the most basic one, but this is probably biased towards suffix-array-based indices, more than to considering typical applications.

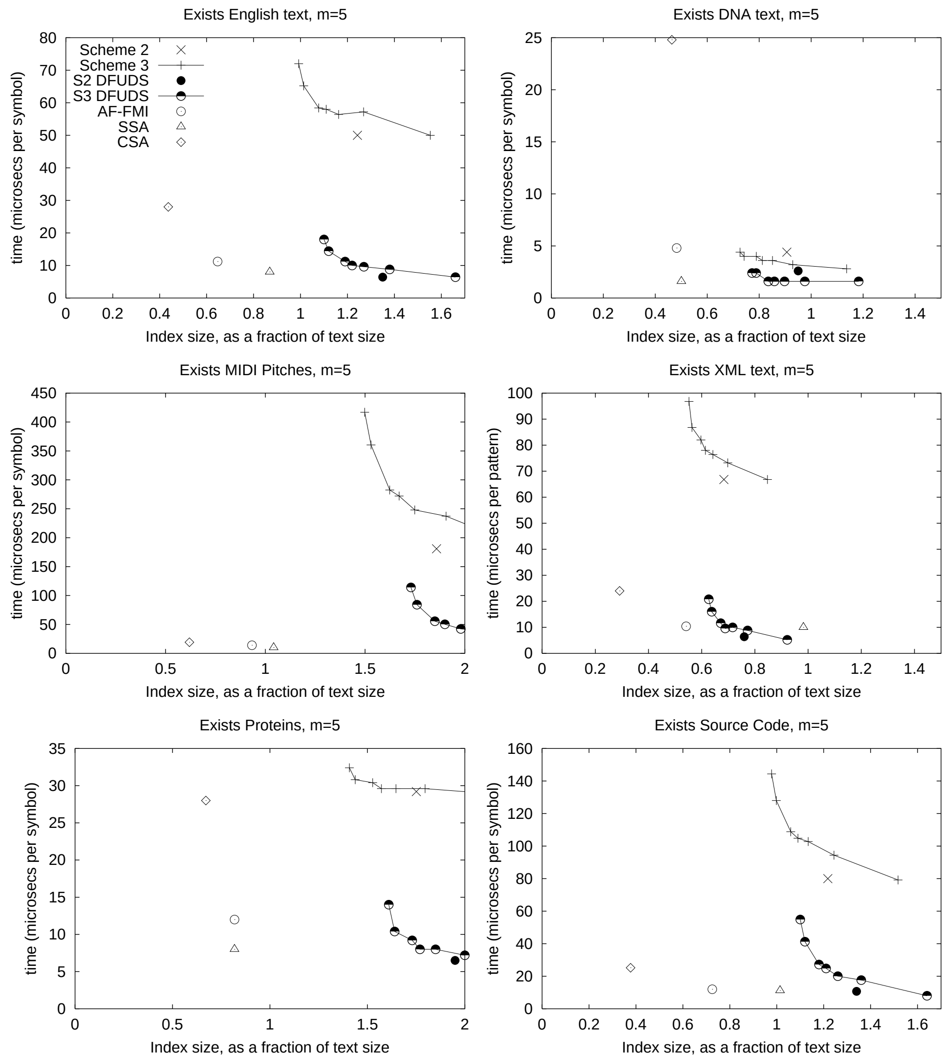


Fig. 18. Experimental time for `exists` queries, for patterns of length $m = 5$. We measure the time in microseconds per pattern.

the competitiveness of the original LZ-index in this respect.

For `locate` queries we tested two alternatives: partial `locate` queries (where a fixed number of occurrences is located) and full `locate` queries. Our experiments show that our technique for solving partial `locate` queries is efficient when searching for short patterns ($m \leqslant 10$), of texts with small alphabets (as on DNA), or when we want to locate a few occurrences (up to 5).

For full `locate` queries, we showed that our indices are more effective when the search pattern is not too long ($m \leqslant 10$), or there are many occurrences to report. In other cases, the navigation time on the tries is predominant, and thus our performance degrades. For example, for short patterns of length 5, in most scenarios our schemes are the best alternative if we can spend at least 4 times the compressed text size. In general, our indices locate up to 1–4 million occurrences per second. When less memory space is available, our indices are outperformed by the very competitive Inverted LZ-index (ILZI) [Russo and Oliveira 2008], which is yet another variant of the Lempel-Ziv-based index family. In this case, however, our indices are still competitive with all suffix-array-based schemes (e.g., the competitive Compressed Suffix Array of Sadakane [2003]).

We also exhibit an important difference between LZ-indices and those based on suffix arrays: the latter need to store extra non-compressible information (the suffix-array samples) in order to carry out `extract`, `display`, and `locate` queries, whereas most of the information stored by our LZ-indices is compressible. Thus, when the text is highly compressible, we get small LZ-indices, which are still fast. Indices based on suffix arrays, on the other hand, cannot use a denser suffix-array sampling (because otherwise they become larger), and henceforth their performance is poor. Thus, our indices are also a good option for highly compressible texts.

We believe that our indices offer an extremely relevant alternative considering their overall performance across the multiple tasks of interest in many real text-search applications.

We made the code of our indices available in the *Pizza&Chili* site, at `http://pizzachili.dcc.uchile.cl/indexes/LZ-index`.

We do not consider in this paper the space-efficient construction of our indices, which is an important issue in practice, since many times a small index requires a large amount of memory space to be build. We plan to adapt the space-efficient algorithm by Arroyuelo and Navarro [2005] for the original LZ-index, so as to construct our LZ-index variants space-efficiently. Currently, our indices are constructed in an uncompressed way, needing about the same space used to build suffix-array-based compressed self-indices. The space-efficient construction of the latter is still at a theoretical stage [Mäkinen and Navarro 2008], or still does not achieve higher-order entropy-bounded space [Hon et al. 2007].

All the compressed self-indexes considered in this paper operate in main memory. Designing data layouts that perform efficiently on secondary memory is becoming an active area of research. In general one must use more space in order to gain locality of reference. For example, a secondary memory variant of the LZ-index requires about twice the space of the original (main-memory) LZ-index [Arroyuelo and Navarro 2007]. We are working on improving the space of this secondary memory variant as well.

REFERENCES

Apostolico, A. 1985. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*. NATO ISI Series. Springer-Verlag, 85–96.

Arroyuelo, D. and Navarro, G. 2005. Space-efficient construction of LZ-index. In *Proc. 16th Annual International Symposium on Algorithms and Computation (ISAAC)*. LNCS 3827. 1143–1152.

Arroyuelo, D. and Navarro, G. 2007. A Lempel-Ziv text index on secondary storage. In *Proc. CPM*. LNCS 4580. 83–94.

Arroyuelo, D., Navarro, G., and Sadakane, K. 2006. Reducing the space requirement of LZ-index. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*. LNCS 4009. 319–330.

Arroyuelo, D., Navarro, G., and Sadakane, K. 2010. Stronger Lempel-Ziv based compressed text indexing. To appear in *Algorithmica*, DOI 10.1007/s00453-010-9443-8. See also `http://www.dcc.uchile.cl/~darroyue/papers/algor2010.pdf`.

Barbay, J., He, M., Munro, J. I., and Rao, S. S. 2007. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 680–689.

Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V., and Rao, S. S. 2005. Representing trees of higher degree. *Algorithmica 43,* 4, 275–292.

Ferragina, P., González, R., Navarro, G., and Venturini, R. 2009. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA) 13*, article 12. 30 pages.

Ferragina, P. and Manzini, G. 2000. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*. 390–398.

Ferragina, P. and Manzini, G. 2005. Indexing compressed texts. *Journal of the ACM 54,* 4, 552–581.

Ferragina, P., Manzini, G., Mäkinen, V., and Navarro, G. 2007. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG) 3,* 2, article 20.

Ferragina, P. and Navarro, G. 2005. Pizza&Chili Corpus — Compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl`.

Geary, R., Rahman, N., Raman, R., and Raman, V. 2006. A simple optimal representation for balanced parentheses. *Theoretical Computer Science 368,* 3, 231–246.

Golynski, A., Munro, J. I., and Rao, S. S. 2006. Rank/select operations on large alphabets: A tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 368–373.

González, R., Grabowski, S., Mäkinen, V., and Navarro, G. 2005. Practical implementation of rank and select queries. In *Poster Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA)*. CTI Press and Ellinika Grammata, 27–38.

Grossi, R., Gupta, A., and Vitter, J. S. 2003. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 841–850.

Grossi, R. and Vitter, J. S. 2000. 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)*. 397–406.

Grossi, R. and Vitter, J. S. 2005. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing 35,* 2, 378–407.

Hon, W. K., Lam, T. W., Sadakane, K., Sung, W.-K., and Yiu, M. 2007. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica 48,* 1, 23–36.

Jansson, J., Sadakane, K., and Sung, W.-K. 2007. Ultra-succinct representation of ordered trees. In *Proc. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 575–584.

Kärkkäinen, J. and Ukkonen, E. 1996. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP)*. Carleton University Press, 141–155.

Kim, D., Na, J., Kim, J., and Park, K. 2005. Efficient implementation of rank and select functions for succinct representation. In *Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA)*. LNCS 3503, 315–327.

Knessl, C. and Szpankowski, W. 2000. Height in a digital search tree and the longest phrase of the Lempel-Ziv scheme. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 187–196.

Kosaraju, R. and Manzini, G. 1999. Compression of low-entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing 29,* 3, 893–911.

Mäkinen, V. and Navarro, G. 2005. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing 12,* 1, 40–66.

Mäkinen, V. and Navarro, G. 2008. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG) 4,* 3, article 32. 38 pages.

Manber, U. and Myers, G. 1993. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing 22,* 5, 935–948.

Manzini, G. 2001. An analysis of the Burrows-Wheeler transform. *Journal of the ACM 48,* 3, 407–430.

Morrison, D. R. 1968. Patricia – practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM 15,* 4, 514–534.

Munro, J. I. 1996. Tables. In *Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS).* LNCS 1180. 37–42.

Munro, J. I., Raman, R., Raman, V., and Rao, S. S. 2003. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP).* LNCS 2719. 345–356.

Munro, J. I. and Raman, V. 2001. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing 31,* 3, 762–776.

Navarro, G. 2004. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms (JDA) 2,* 1, 87–114.

Navarro, G. 2009. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics (JEA) 13,* article 2. 49 pages.

Navarro, G. and Mäkinen, V. 2007. Compressed full-text indexes. *ACM Computing Surveys 39,* 1, article 2.

Okanohara, D. and Sadakane, K. 2007. Practical entropy-compressed rank/select dictionary. In *Proc. Workshop on Algorithm Engineering and Experiments (ALENEX).* 60–70.

Raman, R., Raman, V., and Rao, S. S. 2002. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* 233–242.

Russo, L. and Oliveira, A. 2008. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval 11,* 4, 359–388.

Sadakane, K. 2000. Compressed text databases with efficient query algorithms based on the Compressed Suffix Array. In *Proc. 11th Annual International Symposium on Algorithms and Computation (ISAAC).* LNCS 1969. 410–421.

Sadakane, K. 2003. New Text Indexing Functionalities of the Compressed Suffix Arrays. *Journal of Algorithms 48,* 2, 294–313.

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inform. Theory 23,* 3, 337–343.

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory 24,* 5, 530–536.