On Compressing and Indexing Repetitive Sequences ☆

Sebastian Kreft[a,1,2], Gonzalo Navarro[a,2]

[a]*Department of Computer Science, University of Chile*

Abstract

We introduce *LZ-End*, a new member of the Lempel-Ziv family of text compressors, which achieves compression ratios close to those of LZ77 but performs much faster at extracting arbitrary text substrings. We then build the first self-index based on LZ77 (or LZ-End) compression, which in addition to text extraction offers fast indexed searches on the compressed text. This self-index is particularly effective to represent highly repetitive sequence collections, which arise for example when storing versioned documents, software repositories, periodic publications, and biological sequence databases.

Keywords: compression, repetitive texts, self-indexing, Lempel-Ziv, succinct data structures

1. Introduction and Related Work

Regarding compression as the default, instead of the archival, state of the data is becoming popular due to the increasing gap between access times in main and secondary memory. Compressed sequence and text databases, and compact data structures in general, aim at handling the data directly in compressed form, rather than decompressing before using it [3, 4]. This poses new challenges, as now it is required that the compressed texts should, at least, be accessible at random, and more ambitiously, offer indexed searches.

☆Early versions of this article appeared in *Proc. DCC'10* [1] and *Proc. CPM'11* [2].

Email addresses: `skreft@dcc.uchile.cl` (Sebastian Kreft), `gnavarro@dcc.uchile.cl` (Gonzalo Navarro)

[1]Partially funded by Conicyt's Master Scholarship.

[2]Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.

On general text databases, *self-indexes* [4] provide an excellent solution to this challenge. Self-indexes are data structures that represent a text collection in compressed form, in such a way that not only access to arbitrary text substrings is supported, but also indexed pattern matching, in time at most polylogarithmic on the text size (and in some cases totally independent of it). Invented in the past decade [5, 6], they have been enormously successful to drastically reduce the space burden posed by general text indexes such as suffix trees or arrays [7, 8]. Their compression effectiveness is usually analyzed in terms of the *empirical k-th order entropy model* [9]: $H_k(T)$, the k-th order entropy of text T, is a lower bound to the bits-per-symbol compression achievable by any statistical compressor that models symbol probabilities as a function of the k symbols preceding it in the text. The smallest self-indexes [10, 11] are able to represent a text $T[1,n]$ over alphabet $[1,\sigma]$, within $nH_k(T) + o(n \log \sigma)$ bits of space for any $k \leq \alpha \log_\sigma n$ and constant $\alpha < 1$. Note that the k-th entropy model is meaningful only for small $k \leq \log_\sigma n$ [12].

In this article we focus on offering direct access and indexed searches to *highly repetitive* text collections. These are formed by sets of strings that are mostly near-copies of each other. For example, versioned document collections store all the history of modifications of the documents. Most new versions are minor edits of a previous version. Good examples are the Wikipedia database and the Internet archive. Another example are software repositories, which store all the versioning history of software pieces. Again, except for major releases, most versions are relatively minor variants of previous ones. Yet another example comes from Bioinformatics. Given the sharply decreasing sequencing costs, large sequence databases of individuals of the same or closely related species are appearing. The genomes of two humans, for example, share 99.9% to 99.99% of their sequence. In many applications the versioning has a known linear, tree, or acyclic graph structure, but in others, like genome databases, no such clear structure exists.

The k-th order entropy model is adequate for many practical text collections, but not for repetitive ones. If one concatenates two identical texts, the statistical structure of the concatenation is almost the same as that of the pieces, and thus the k-th order entropy basically does not change. Therefore, statistical compression is not a good approach for repetitive sequences. Consequently, those self-indexes that are precisely tailored to the k-th order entropy model [10, 11] are insensitive to repetitiveness and fail to compress repetitive text collections.

Compression models that are adequate for repetitive sequences include grammar compression [13, 14] and Lempel-Ziv compression [15, 16, 17]. These factor the text into substrings that appear repeatedly and take advantage of short factorizations. Grammar compressors enable efficient direct access to the data [18, 19], and they have also been extended to support self-indexing [18], both with good practical results [20, 21, 22]. Similar developments have been pursued on the LZ78 variant [17] of Lempel-Ziv compression [23, 24, 25, 26, 27]. The stronger LZ77 variant [16] has proved much harder to handle. Only some techniques for providing direct access have been recently proposed [28]. These assume that the repetitive collection has a known structure where one can identify which texts are close variants of which, so that representative texts can be chosen and the others can be compressed with respect to the representatives.

Mäkinen et al. [29, 30] faced the challenge of self-indexing repetitive collections, focusing on the case of large DNA databases of the same species. They studied how repetitiveness in the text collection translates into runs of equal letters in its Burrows-Wheeler transform [31] or runs of successive values in the Ψ function [32]. Based on these findings they engineered variants of FM-indexes [33] and Compressed Suffix Arrays (CSAs) [34] that take advantage of repetitiveness. Their best structure, the Run-Length CSA (RLCSA) has standed as the best general-purpose self-index for repetitive collections.

However, Mäkinen et al. showed that their new self-indexes were very far (by a factor of 10) from the space that can be achieved by a compressor based on LZ77. They showed that the runs model is intrinsically inferior to the LZ77 model to capture repetitions. They also confirmed experimentally that LZ78-based self-indexes [24] were too weak to capture high repetitiveness. It is also known [35] that grammar-based compression is intrinsically inferior to LZ77 compression. This has also been confirmed in practice [21, 22]. LZ77 compression is particularly well-suited to capture repetitiveness, as it parses the text into consecutive maximal *phrases* so that each phrase appears earlier in the text. It is then natural to advocate for a self-index based on LZ77.

Decompressing LZ77-compressed data from the beginning is simple and time-optimal. Yet, extracting an arbitrary substring is expensive, with cost bounded only by the collection size in general. Cutting the text into blocks allows decompressing individual blocks, but compression ratio is ruined as long-range repetitions are not captured. As explained, some solutions assuming that one can identify a base sequence and regard the others as close variants of it have been explored [30, 28], but this assumption can be too

strong in many real-life repetitive collections.

In this article we introduce a new Lempel-Ziv parsing, *LZ-End*, whose compression ratio is close to that of LZ77 (we conjecture it is 2-optimal in the worst case, yet in our real-life collections it was never worse by more than 20%). LZ-End forces the source of a phrase to finish at a previous phrase boundary, and as a result it can guarantee that a substring finishing at a phrase boundary can be extracted in optimal time. It is easy to enforce that individual sequences in a collection end at phrase boundaries, so that they can be extracted optimally and fast in practice. In general, a substring of length ℓ can be extracted in time $O(\ell + h)$ on LZ-End, whereas we achieve only $O(\ell h)$ on LZ77. Here h is a measure of the nesting of the parsing, that is, how many times a character is transitively copied.

We then go on to design the first self-index, for repetitive sequences, based on LZ77 or LZ-End compression. There exists a pioneer theoretical proposal for LZ77-based indexing by Kärkkäinen and Ukkonen [36, 37], but it requires to have the text in plain form and has never been implemented. Although it guarantees an index whose size is of the same order of the LZ77 compressed text, the constant factors are too large to be practical. This work can be regarded as the predecessor of all the Lempel-Ziv self-indexes that followed [24, 25, 26, 27]. However, all these are bound to the LZ78 parsing, which as explained is more tractable but too weak to capture high repetitiveness [29].

Our self-index can be seen as a reduced-space variant of Kärkkäinen and Ukkonen's LZ77 index, which solves the problem of not having the text at hand and also makes use of recent compressed data structures. This involves designing new solutions to some subproblems where the original proposal [36] was too space-consuming, for example when handling secondary occurrences.

Some of these solutions have independent interest. For example, consider the problem of preprocessing a sequence of values so that later, given a position and a value, one returns the rightmost position preceding the given one that holds a value smaller than the given one. We show how to solve this query in logarithmic time and sublinear extra space (Theorem 4.10).

Let n' be the number of phrases of the LZ77 or LZ-End parsing of T. Then a basic Lempel-Ziv compressor outputs $n'(2\log n + \log\sigma)$ bits, and a more clever one may achieve $n'(\log n + \log\sigma)$ bits (our logarithms are base 2 by default). The size of our self-index is $3n'\log n + O(n'\log\sigma) + o(n)$ bits, that is, asymptotically 3 times the size of the output of a good Lempel-Ziv compressor. It can determine the existence of pattern $p_{1,m}$ in T in time $O(m^2h + m\log n')$. After this check, each occurrence of p is reported in time

$O(\log n')$. It retains the complexities we have given for substring extraction.

An important feature of our index is that it is *universal*, that is, it compresses repetitive collections without knowing the versioning structure of the data (which in some applications is unknown or even absent).

We implemented our self-index over LZ77 and LZ-End parsings, and compared it with the state of the art on a number of real-life repetitive collections consisting of Wikipedia versions, versions of public software, periodic publications, and DNA sequence collections. We maintain a public repository with those repetitive collections in `http://pizzachili.dcc.uchile.cl/repcorpus.html`, so that standardized comparisons are possible. Our implementations and those of the RLCSA are also available in there.

Our experiments show that in practice the smallest-space variant of our index takes 2.5–4.0 times the space of an LZ77-based compressor, it extracts 0.5–2 million characters per second, and locates each occurrence of a pattern of length 10 in 10–50 microseconds. Compared to the state of the art (RLCSA), our self-index always takes less space, less than a half on our DNA and Wikipedia corpus. Searching for short patterns is faster than on the RLCSA. On longer patterns our index offers competitive space/time tradeoffs.

The article is organized as follows. In Section 2 we survey the main Lempel-Ziv parsings, LZ77 and LZ78, and establish their basic properties. In Section 3 we describe our new parsing, LZ-End, and analyze its performance in terms of extraction of arbitrary substrings and compression ratio. Section 4 describes a self-index built on top of a parsing like LZ77 or LZ-End, and analyzes its space and time complexities. Section 5 gives and analyzes the algorithms for building the parsings and the self-index. Section 6 gives experimental results on space and time performance of our parsing and self-index variants. Finally, Section 7 concludes.

The reader will be referred to an extended version of this work [38] for some secondary results and less relevant details.

2. Lempel-Ziv Parsings

Dictionary-based compression methods build a dictionary of strings, and then *parse* the text into a sequence of *phrases*, each of them belonging to the dictionary. LZ77 [16] can be seen as a dictionary-based compression scheme in which the dictionary used is the set of substrings of the preceding text. This makes it very well suited for repetitive texts.

Definition 2.1 ([16]). *The* LZ77 parsing *of text* $T[1,n]$ *is a sequence* $Z[1,n']$ *of* phrases *such that* $T = Z[1]Z[2]\ldots Z[n']$*, built as follows. Assume we have already processed* $T[1,i-1]$ *producing the sequence* $Z[1,p-1]$*. Then, we find the longest prefix* $T[i,i'-1]$ *of* $T[i,n]$ *which occurs in* $T[1,i-1]$*, set* $Z[p] = T[i,i']$ *and continue with* $i = i'+1$*. The occurrence in* $T[1,i-1]$ *of the prefix* $T[i,i'-1]$ *is called the* source *of the phrase* $Z[p]$*.*

Note that our definition of LZ77 differs from the original one [16] in that we do not allow the source of $T[i,i'-1]$ to extend beyond position $i-1$. While avoiding this is rather typical in the literature on searching Lempel-Ziv compressed text (e.g., [39, 35]), the variant does not have an agreed-upon name. More than that, Ziv and Lempel called their method LZ1, and what is usually called LZ77 is a variant of LZ1 where the referenced text must belong to a window of fixed length preceding $T[i]$. Rather than inventing a new name, we prefer to retain the name LZ77, as it is the most widely known for the general idea.

Note that each phrase is composed of the content of a source, which can be the empty string ε, plus a trailing character. Note also that all phrases of the parsing are different, except possibly the last one. To avoid that case, a special character $ is appended at the end, $T[n] = \$$.

Typically a source is represented as a triple $Z[p] = (start, len, c)$, where *start* is the starting position of the source, *len* is the length of the source and c is the trailing character. Then a simple representation of T based on the parsing requires $n'(2\log n + \log\sigma)$ bits, where σ is the alphabet size.

Example 2.2. *The LZ77 parsing of* T = '`alabar_a_la_alabarda$`' *is as follows:*

a	l	ab	ar	_	a_	la_	alabard	a$

In this example the seventh phrase copies two characters starting at position 2 and has a trailing character '_'.

One of the greatest advantages of this algorithm is the simple and fast decompression scheme, opposed to the construction algorithm which is more complicated. Decompression runs in linear time by copying the source content referenced by each phrase and then the trailing character. However, random text extraction is not as easy.

A weaker Lempel-Ziv variant, which makes the compressed representation more manipulable, is LZ78 [17]. It restricts new phrases to be equal to some previous phrase plus one character.

Definition 2.3 ([17]). *The* LZ78 parsing *of text $T[1,n]$ is a sequence $Z[1,n']$ of* phrases *such that $T = Z[1]Z[2]\dots Z[n']$, built as follows. Assume we have already processed $T[1,i-1]$ producing the sequence $Z[1,p-1]$. Then, we find the longest prefix $T[i,i'-1]$ of $T[i,n]$ which is a phrase $Z[q]$ for some $q < p$, set $Z[p] = T[i,i']$, and continue with $i = i'+1$.*

With respect to compression, both LZ77 and LZ78 converge to the entropy of stationary ergodic sources, and in particular to the empirical entropy.

Definition 2.4 ([40]). *A parsing algorithm is said to be* coarsely optimal *if its compression ratio $\rho(T)$ differs from the k-th order empirical entropy $H_k(T)$ by a quantity depending only on the length of the text and that goes to zero as the length increases. That is, $\forall k\, \exists f_k, \lim_{n\to\infty} f_k(n) = 0$, such that for every text T, $\rho(T) \leq H_k(T) + f_k(|T|)$.*

Theorem 2.5 ([40, 41]). *The LZ77 and LZ78 parsings are coarsely optimal.*

However, converging to $H_k(T)$ is not sufficiently good for repetitive texts. As explained, measure $H_k(T)$ does not capture repetitiveness, in particular $|TT|H_k(TT) \geq 2|T|H_k(T)$, as proved next.

Lemma 2.6. *Let T be a string of length n. For any $k \leq n$ it holds $H_k(TT) \geq H_k(T)$.*

Proof. By definition, $H_k(T) = \frac{1}{|T|}\sum_{S\in C(T,k)} |T^S|H_0(T^S)$, where $C(T,k)$ are the substrings of length k present in T, T^S is the string formed by the characters preceding the occurrences of S in T, and $H_0(T) = \frac{1}{|T|}\sum_{c\in\Sigma} n_c^T \log(|T|/n_c^T)$, where n_c^T is the number of occurrences of c in T [9].

We use the fact that, for any X, Y, it holds $|XY|H_0(XY) \geq |X|H_0(X)$: $|XY|H_0(XY) = \sum_{c\in\Sigma}(n_c^X + n_c^Y)\log\frac{|X|+|Y|}{n_c^X+n_c^Y} \geq |X|\sum_{c\in\Sigma}\frac{n_c^X}{|X|}\log\frac{|X|+|Y|}{n_c^X+n_c^Y} \geq |X|\sum_{c\in\Sigma}\frac{n_c^X}{|X|}\log\frac{|X|}{n_c^X}$ (by Gibbs inequality [42]).

Now, TT has at most k substrings not in T. For each $S \in C(T,k)$, we have $(TT)^S = T^S A^S T^S$, for some A^S such that $|A^S| \leq k$. Then, by definition, $H_k(TT) = \frac{1}{|TT|}\sum_{S\in C(TT,k)} |(TT)^S|H_0((TT)^S)$. Since $C(T,k) \subseteq C(TT,k)$, this is $\geq \frac{1}{2|T|}\sum_{S\in C(TT,k)} |T^S A^S T^S|H_0(T^S A^S T^S)$. Using $X = T^S T^S$ and $Y = A^S$ in the previouds paragraph, this is $\geq \frac{1}{2|T|}\sum_{S\in C(TT,k)} |T^S T^S|H_0(T^S T^S)$, since $|T^S A^S T^S|H_0(T^S A^S T^S) = |T^S T^S A^S|H_0(T^S T^S A^S)$. Since $H_0(TT) = H_0(T)$, finally, this is equal to $\frac{1}{|T|}\sum_{S\in C(TT,k)} |T^S|H_0(T^S) = H_k(T)$. □

That is, a statistical-based compressor applied to TT doubles its output size with respect to T, being blind to the repetitiveness. Instead, the LZ77 parsing captures exact and near-repetitions in the text, as stated in the following easy-to-prove lemma.

Lemma 2.7 ([38, Lem. 4.1])**.** *Given a text T, let $L^{77}(T)$ be the number of phrases of the LZ77 parsing of text T. Then the following statements hold for any characters a, b: $L^{77}(TT) = L^{77}(T) + 1$; $L^{77}(TT\$) \leq L^{77}(T\$) + 1$; $L^{77}(TT') \leq L^{77}(TaT') + 1$; $L^{77}(TaT') \leq L^{77}(TT') + 1$; $L^{77}(TaT') \leq L^{77}(TbT') + 1$.*

The lemma states that a repetition of a text, as well as single-character edits on a text, alter the number of phrases very little. This explains why LZ77 is so strong to compress highly repetitive collections. On the contrary, LZ78 is not that powerful. On $T = a^n$ it produces $n' = \frac{\sqrt{n}}{2} + O(1)$ phrases, and this increases to $n' = \frac{\sqrt{2n}}{2} + O(1)$ on TT. LZ77, instead, produces $n' = \log n + O(1)$ phrases on T and just one more phrase on TT.

3. LZ-End Parsing

In this section we introduce a new LZ77-like parsing. It permits faster extraction of arbitrary text substrings, while achieving compression ratios close to those of LZ77.

Definition 3.1. *The LZ-End parsing of text $T[1,n]$ is a sequence $Z[1,n']$ of* phrases *such that $T = Z[1]Z[2]\ldots Z[n']$, built as follows. Assume we have already processed $T[1,i-1]$ producing the sequence $Z[1,p-1]$. Then, we find the longest prefix $T[i,i'-1]$ of $T[i,n]$ that is a suffix of $Z[1]\ldots Z[q]$ for some $q < p$, set $Z[p] = T[i,i']$ and continue with $i = i'+1$.*

Example 3.2. *The LZ-End parsing of T = '`alabar_a_la_alabarda$`' is as follows:*

a	l	ab	ar	_	a_	la	_a	labard	a$

When generating the seventh phrase we cannot copy two characters as in Example 2.2, because '`la`' does not end at a previous phrase boundary. However, '`l`' does end at a phrase boundary, hence we generate the phrase '`la`'. Notice that the number of phrases increased from 9 to 10.

The LZ-End parsing is similar to that by Fiala and Green [43], *LZFG*, in that theirs restricts where the sources start, while ours restricts where the sources end. This apparently irrelevant difference is the key feature that will allow us to extract arbitrary phrases in optimal time.

3.1. Encoding

The output of an LZ77 compressor is, essentially, the sequence of triplets $z(p) = (j, \ell, c)$, such that the source of $Z[p] = T[i, i']$ is $T[j, j+\ell-1]$, $\ell = i'-i$, and $c = T[i']$. This format allows fast decompression of T, but not extracting an individual phrase $Z[p]$.

Although the LZ-End parsing may generate more phrases than LZ77, it permits a shorter encoding of each phrase. The pth phrase can be encoded as $z(p) = (q, \ell, c)$, such that the source of $Z[p] = T[i, i']$ is a suffix of $Z[1] \ldots Z[q]$, and ℓ and c are as for LZ77. We introduce a more sophisticated encoding that, in addition, will allow us to extract individual phrases in optimal time.

- $source[1, n']$ (using $n' \lceil \log n' \rceil$ bits) encodes the phrase identifier where the source ends (q above).
- $B[1, n]$ (using $n' \log \frac{n}{n'} + O(n' + \frac{n \log \log n}{\log n})$ bits in compressed form [44]) is a bitmap marking the n' ending positions of the phrases in T.
- $L[1, n']$ (using $n' \lceil \log \sigma \rceil$ bits) encodes the trailing characters (c above).

The compressed representation of B [44] supports operations *rank* and *select* in constant time: $rank_b(B, i)$ is the number of bs in $B[1, i]$ ($b = 0$ or 1), and $select_b(B, j)$ is the position of the j-th b in B. Both return 0 if the argument i or j is 0. In our case, phrase p goes from position $select_1(p-1) + 1$ to position $select_1(B, p)$. Thus we have $z(p) = (q, \ell, c) = (source[p], select_1(B, p) - select_1(B, p-1) - 1, L[p])$. We can also compute in constant time that text position i belongs to phrase $Z[rank_1(B, i-1) + 1]$.

3.2. Extraction Algorithm

Figure 1(a) gives the algorithm to extract an arbitrary substring in LZ-End. The extraction works from right to left. First we compute the last phrase p overlapping the substring. If the last character to extract is stored explicitly, that is, it is at the end of phrase p (line 4), we output $L[p]$ after recursively extracting the remaining substring (line 5). Else we split the substring to extract into two parts, a first one intersecting the phrases before

Extract$(start, \ell)$

```
1  if ℓ > 0 then
2    end ← start + ℓ − 1
3    p ← rank_1(B, end − 1) + 1
4    if B[end] = 1 then
5      Extract(start, ℓ − 1)
6      output L[p]
7    else
8      pos ← select_1(B, p − 1) + 1
9      if start < pos then
10       Extract(start, pos − start)
11       ℓ ← end − pos + 1
12       start ← pos
13     Extract(select_1(B, source[p]
       −1) + 1 + start − pos, ℓ)
```

(a) LZ-End extraction algorithm for $T[start, start + \ell - 1]$.

Build

```
1  F ← {⟨0, n + 1⟩}
2  i ← 1, p ← 1
3  while i ≤ n do
4    [sp, ep] ← [1, n]
5    i′ ← i, j ← i, q ← 0
6    while i′ ≤ n do
7      [sp, ep] ← BWS(sp, ep, T[i′])
8      mpos ← RMQ_A(sp, ep)
9      if A[mpos] ≤ n+1−i then break
10     i′ ← i′ + 1
11     ⟨q′, fpos⟩ ← Successor(F, sp)
12     if fpos ≤ ep then j ← i′, q ← q′
13   Insert(F, ⟨p, A^{−1}[n + 1 − j]⟩)
14   output (q, j − i, T[j])
15   i ← j + 1,  p ← p + 1
```

(b) LZ-End construction algorithm. $\mathcal{F}$ stores pairs ⟨phrase identifier, text position⟩ and answers successor queries on the text position. We assume $\text{RMQ}(sp, ep)$ returns 0 if $sp > ep$.

Figure 1: LZ-End extraction and construction algorithms.

p, and the second one intersecting phrase p. If the first is not empty we recursively extract it (line 10). Then we recursively extract the second part from the source of phrase p (line 13).

While the algorithm works for extracting any substring, it is optimal when the substring to extract ends at a phrase boundary.

Theorem 3.3. *Function* Extract *outputs a text substring* $T[start, start+\ell-1]$ *ending at a phrase boundary in time* $O(\ell)$*.*

Proof. Let p be the phrase overlapping $end = start+\ell-1$. Since $T[start, end]$ ends at a phrase boundary, it holds $B[end] = 1$. We proceed by induction on ℓ. The case $\ell \leq 1$ is trivial by inspection. Otherwise, we output $T[end]$ at line 6 after a recursive call on length $\ell - 1$. If in the recursive call we are at phrase $p-1$, then $|Z[p]| = 1$ and we are done by the inductive hypothesis since $T[start, end - 1]$ finishes at the end of phrase $p - 1$. So assume we are still at phrase p. This time we go to line 8. Phrase p starts at pos. If $start < pos$, we carry out a recursive call at line 10 to extract the segment $T[start, pos - 1]$. As this segment finishes at the end of phrase $p - 1$, this

call takes time $O(pos - start)$ by the inductive hypothesis. Now the segment $T[\max(start, pos), end]$ is contained in $Z[p]$ and it finishes one symbol before the phrase ends. Thus a copy of it finishes where $Z[source[p]]$ ends. So induction applies also to the recursive call at line 13, which extracts the remaining string from the source of $Z[p]$, also in optimal time. □

An arbitrary substring is extracted essentially by unrolling the last phrase p (see Figure 1(a)) overlapping the substring, from the end of p until reaching the substring. The following measure is useful to analyze the general extraction cost.

Definition 3.4. *Let $T = Z[1]Z[2] \ldots Z[n']$ be a LZ-parsing of $T[1, n]$. Then the* height *of the parsing is defined as $h = \max_{1 \le i \le n} C[i]$, where C is defined as follows. Let $Z[i] = T[a, b]$ be a phrase whose source is $T[c, d]$, then*

$$\begin{aligned} C[k] &= C[(k - a) + c] + 1, \forall a \le k < b \\ C[b] &= 1 \end{aligned}$$

Array C counts how many times a character was transitively copied from its original source. This is also the extraction cost of that character. Hence, the value h is the worst-case bound for extracting a single character in the LZ parse. The extraction cost can be bound in terms of h.

Theorem 3.5. *Extracting a substring of length ℓ from an LZ-End parsing takes time $O(\ell + h)$.*

Proof. Theorem 3.3 already shows that the cost to extract a substring ending at a phrase boundary is constant per extracted symbol. The only piece of code in Figure 1(a) that does not amortize in this sense is line 13, where we recursively unroll the last phrase, removing the last character each time, until hitting the end of the substring to extract. By definition of h, this line cannot be executed more than h times. So the total time is $O(\ell + h)$. □

Remark 3.6. *Function* Extract *also works on parsing LZ77, by storing absolute starting text positions instead of phrase numbers in $source'[1, n']$ and replacing $select_1(B, source[p] - 1) + 1$ by $source'[p]$ in line 13. However, in this case the best theoretical bound we can prove for extracting a substring of length ℓ is $O(\ell h)$, which follows trivially from the definition of h.*

Next lemma proves that h is upper bounded by the maximum length of a phrase. This gives further intuition on this value.

Lemma 3.7. *In an* LZ-End *parsing it holds that h is smaller than the longest phrase, i.e.,* $h \leq \max_{1 \leq p \leq n'} |Z[p]|$.

Proof. We prove by induction that $C[i] \leq C[i+1] + 1$ for all $1 \leq i < n$. From this inequality the lemma follows: For all positions i_p where a phrase p ends, it holds by definition that $C[i_p] = 1$. Thus, for all positions i in the phrase p, we have $C[i] \leq C[i_p] + i_p - i \leq |Z[p]|$.

The first phrase of any LZ-End parsing is $T[0]$, and the second is either $T[1]$ or $T[1]T[2]$. In the first case, we have $C[1]C[2] = 1, 1$; in the second, $C[1]C[2]C[3] = 1, 2, 1$. In both cases the property holds. Now, suppose the inequality is valid up to position i_p where the phrase $Z[p]$ ends. Let i_{p+1} be the position where the phrase $Z[p+1] = T[a, b]$ ends (so $a = i_p + 1$ and $b = i_{p+1}$) and let $T[c, d]$ be its source. For $i_p + 1 \leq i < i_{p+1}$, it holds $C[i] = C[(i-a)+c] + 1$, and since $d \leq i_p$, the inequality holds by inductive hypothesis for $i_p + 1 \leq i \leq i_{p+1} - 2$. By definition of the *LZ-End* parsing, the source of a phrase ends in a previous end of phrase, hence $C[i_{p+1} - 1] = C[d] + 1 = 2 \leq 1 + 1 = C[i_{p+1}] + 1$. For position i_{p+1} (end of phrase) the inequality trivially holds as it has by definition the least possible value. □

For example, on an ergodic Markov source of entropy H, it holds $h = O(\log(n)/H)$, yet we remind this is not a practical model for repetitive texts. On our repetitive collection corpus (see Section 6) h is between 22 and 259 for LZ-End, and between 22 and 1003 for LZ77. Its average values, on the other hand, are 5–25 on LZ-End and 5–176 on LZ77.

Note that our extraction algorithm is not faster on LZFG parsing [43] than on LZ77, even for substrings starting at phrase boundaries. The reason is that they align to the beginning of phrases but store the trailing character of phrases. To solve the problem while innovating the least with respect to LZFG we could define *LZ-Begin*, which requires that sources start at phrase boundaries (like LZFG) but store the leading, instead of the trailing, symbols of the phrases. LZ-Begin allows for fast extraction, just like LZ-End. However, as we see next, it does not compress well on repetitive sequences.

3.3. Compression Performance

We now study the compression performance of LZ-End, first with respect to the empirical k-th order entropy and then on repetitive texts.

3.3.1. Coarse Optimality

We prove that LZ-End is coarsely optimal. The main tool is the following lemma.

Lemma 3.8. *All the phrases generated by an LZ-End parse are different.*

Proof. Assume for contradiction $Z[p] = Z[p']$ for some $p < p'$. When $Z[p']$ was generated, we could have taken $Z[p]$ as the source, yielding phrase $Z[p']c$, longer than $Z[p']$. This is a valid source as $Z[p]$ is a suffix of $Z[1] \ldots Z[p]$. So this is not an LZ-End parse. □

The uniqueness property does not hold on LZ-Begin: Let $T = AxyAyAz$, where x, y, z are distinct characters and A is a string; if we have parsed up to Ax, then the next phrase will be yA, and the following phrase will also be yA. Moreover, in practice LZ-Begin compresses up to 40 times worse than LZ-End (see Section 6). Thus, despite apparent symmetries, LZ-End is the only variant combining good extraction time and good compression ratio.

The uniqueness lemma enables us to apply a couple of known results on this kind of parsings, which leads us to the coarse optimality proof.

Lemma 3.9 ([15])**.** *Any parsing of $T[1, n]$ into n' distinct phrases satisfies $n' = O\left(\frac{n}{\log_\sigma n}\right)$, where σ is the alphabet size of T.*

Lemma 3.10 ([40])**.** *For any text $T[1, n]$ parsed into n' different phrases, it holds $n' \log n' \leq nH_k(T) + n' \log \frac{n}{n'} + \Theta(n'(1 + k \log \sigma))$, for any k.*

Theorem 3.11. *The LZ-End compression is coarsely optimal.*

Proof. We follow the proof by Kosaraju and Manzini [40] for LZ77. Here we must also consider our particular encoding. The size of the parsing in bits is

$$\begin{aligned} \textit{LZ-End}(T) &= n' \lceil \log n' \rceil + n' \log \frac{n}{n'} + O\left(n' + \frac{n \log \log n}{\log n}\right) + n' \lceil \log \sigma \rceil \\ &= n' \log n + O\left(n' \log \sigma + \frac{n \log \log n}{\log n}\right). \end{aligned}$$

Thus from Lemmas 3.8 and 3.10 we have

$$\textit{LZ-End}(T) \quad \leq \quad nH_k(T) + 2n' \log \frac{n}{n'} + O\left(n'(k+1) \log \sigma + \frac{n \log \log n}{\log n}\right).$$

Now, by means of Lemma 3.9 and since $n' \log \frac{n}{n'}$ is increasing in n', we get

$$\begin{aligned} \textit{LZ-End}(T) &\leq nH_k(T) + O\left(\frac{n \log \sigma \log\log n}{\log n}\right) \\ &\quad + O\left(\frac{n(k+1)\log^2 \sigma}{\log n} + \frac{n \log\log n}{\log n}\right) \\ &= nH_k(T) + O\left(\frac{n \log \sigma (\log\log n + (k+1)\log \sigma)}{\log n}\right). \end{aligned}$$

Thus, diving by n and taking k and σ as constants, we get that the compression ratio is

$$\rho(T) \leq H_k(T) + O\left(\frac{\log\log n}{\log n}\right).$$

□

Corollary 3.12. *The output size in bits of our LZ-End compressor is* $\textit{LZ-End}(T) \leq nH_k(T) + o(n \log \sigma)$ *for any* $k = o(\log_\sigma n)$.

3.3.2. Performance on Repetitive Texts

We have been unable to prove a worst-case bound for the competitiveness of LZ-End compared to LZ77. After many attempts, we have been able to produce families of sequences where LZ-End produces almost twice the number of phrases generated by the LZ77 parsing. Those examples, moreover, require an unbounded alphabet. The following is one of them.

Example 3.13. *Let* $T = 112 \cdot 113 \cdot 214 \cdot 325 \cdot 436 \cdot 547 \cdot \ldots \cdot (\sigma-2)(\sigma-3)\sigma$. *The length of the text is* $n = 3(\sigma-1)$. *The LZ parsings are:*

LZ77	1	12	113	214	325	436	547	…	$(\sigma-2)(\sigma-3)\sigma$					
LZ-End	1	12	11	3	21	4	32	5	43	6	54	7	…	$(\sigma-2)(\sigma-3)$	σ

The size of LZ77 is $n' = \sigma$ *and that of LZ-End is* $n' = 2(\sigma-1)$.

This means that LZ-End is at best 2-competitive with LZ77. We conjecture that this is tight.

Conjecture 3.14. *Parsing LZ-End is 2-competitive with parsing LZ77 in terms of the number of phrases generated.*

On the other hand, we show that LZ-End satisfies one of the key properties of Lemma 2.7, with constant 2. This is in agreement with our conjecture.

Lemma 3.15. *Given a text T, it holds $L^{End}(TT\$) \leq L^{End}(T\$) + 2$, where $L^{End}(\cdot)$ is the number of phrases of the LZ-End parsing.*

Proof. First assume that $T\$$ is parsed as $T\$ = T \mid \$$, where the "$|$" indicates a phrase boundary (i.e., the last phrase of $T\$$ is just \$). Then $TT\$$ will be parsed as $TT\$ = T \mid T\$$ and both will have the same number of phrases. Otherwise, let $T\$ = XyY \mid aA\$$ be parsed into n' phrases, and $y, a \in \Sigma$. In this partition, $aA\$$ is the last phrase of $T\$$ and X is the longest string such that aAX occurs in T aligned at the end of a phrase boundary (this holds at least for $X = \varepsilon$ since aA was chosen as the source of the last phrase). Note that XyY may be formed by several phrases. If Y does not appear anywhere else in T, then $TT\$$ will be parsed as $TT\$ = XyY \mid aAXy \mid Ya \mid A\$$, with $n'+2$ phrases. More in general, let YB the longest prefix of YaA that appears in T aligned at the end of a phrase, with $aA = BcC$, then the parsing will be $TT\$ = XyY \mid aAXy \mid YBc \mid C\$$, also with $n' + 2$ phrases. Note that C is a suffix of A and thus it appears aligned at a phrase boundary in T. □

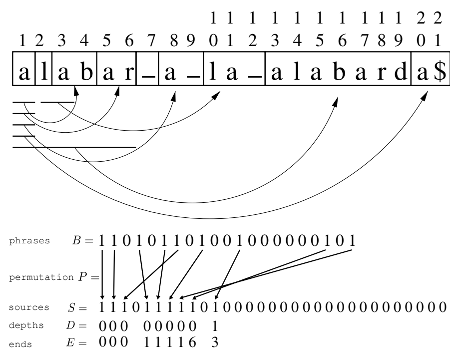
4. A Self-Index

We now start from an LZ77-like parsing, such as LZ77 itself or LZ-End, and design a self-index around the corresponding compressed text representation, which supports direct access to arbitrary text substrings.

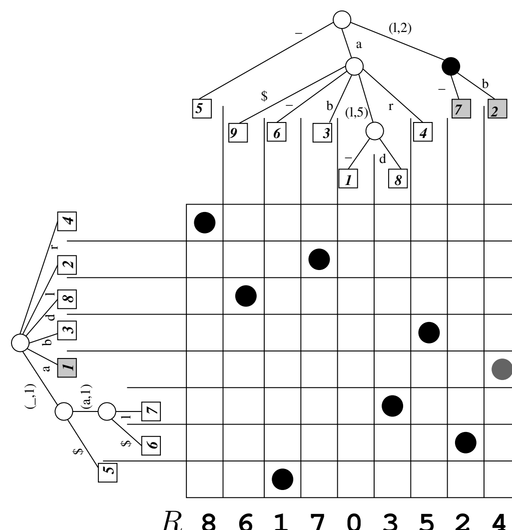
Assume we have a text T of length n, which is partitioned into n' phrases using an LZ77-like compressor. Let $p_{1,m}$ be a search pattern. We call *primary occurrences* of p those overlapping more than one phrase or ending at a phrase boundary; and *secondary occurrences* the others.

Example 4.1. *In Figure 2(a), the occurrence of* `'lab'` *starting at position 2 is primary as it spans two phrases. The second occurrence, starting at position 14, is secondary.*

Our self-index will find first the primary occurrences, and those will be used to recursively find the secondary ones (these, in turn, will be used to find further secondary occurrences). We describe next how we find each kind of occurrence, and which extra data structures we need for that purpose. To begin, we describe how we change the representation of an LZ77 or LZ-End parsing to a format that is more suitable for our self-index.



(a) The LZ77 parsing of the string ‘alabar_a_la_alabarda$’, showing the sources of each phrase on top. On the bottom, bitmap B marks the ends of phrases, the bitmap S marks the starting positions of sources, and the permutation P connects phrases to sources. We also show array D of depths and (virtual) array E of ending source positions (these arrays are exclusive).



(b) Top: The sparse suffix trie. The black node is the one we arrive at when searching for ‘la’, and the gray leaves of its subtree represent the phrases that start with ‘la’. Left: The reverse trie for the string. The gray leaf is the node at which we stop searching for ‘a’. Bottom: The range structure for the string. The gray dot marks the only primary occurrence of the pattern ‘ala’ (it is the only dot in the range defined by the gray nodes).

Figure 2: Our self-index structure over the example text T = ‘alabar_a_la_alabarda$’ and part of the process of searching for p = ‘ala’.

4.1. Parsing Representation

We will store $Z[1, n']$ in a particular way that enables further operations needed by the self-index. We will use $L[1, n']$ and $B[1, n]$ just as in Section 3.1, but $source[1, n']$ will be stored differently: We will store a bitmap $S[1, n+n']$ that describes the structure of the sources in T, as follows. We traverse T from left to right, from $T[1]$ to $T[n]$. At step i, if there are k sources starting at position $T[i]$, we append 1^k0 to S (k may be zero). Empty sources (i.e., $i = i'$ in $Z[p] = T[i, i']$) are assumed to lie just before $T[1]$ and appended at the beginning of S, followed by a 0. So the 0s in S correspond to text positions, and the 1s correspond to the successive sources. Finally, we store

a permutation $P[1, n']$ that maps targets to sources, that is, $P[i] = j$ means that the source of the ith phrase starts at the position corresponding to the jth 1 in S. Figure 2(a) gives an example.

Bitmap $S[1, n + n']$ contains only n' 1s. It is stored using again a compressed representation [44] that takes $n' \log \frac{n}{n'} + O(n') + o(n)$ bits. Permutation P is stored using a representation [45] that computes $P[i]$ in constant time and $P^{-1}[j]$ in time $O(l)$, using $(1 + 1/l)n' \log n' + O(n')$ bits of space. We use parameter $l = \log n'$. Thus our total space for L, B, S, and P, is $n' \log n' + 2n' \log \frac{n}{n'} + n' \log \sigma + O(n') + o(n)$ bits, only slightly more than in Section 3.1. The extraction times of Section 3.2 are retained, as in line 13 of Figure 1(a) we have $select_1(B, source[p]-1)+1 = rank_0(S, select_1(S, P[p]))$, which is computed in constant time.

4.2. Primary Occurrences

Each primary occurrence can be split as $p = p_{1,i}\, p_{i+1,m}$, where the left side $p_{1,i}$ is a nonempty suffix of a phrase and the (possibly empty) right side $p_{i+1,m}$ is the concatenation of zero or more consecutive phrases plus a prefix of the next phrase. To find primary occurrences we partition the pattern into two in every possible way. Then, we search for the left part in the suffixes of the phrases and for the right part in the prefixes of the suffixes of T starting at phrase boundaries. Then, we find which pairs of left and right occurrences are concatenated, thus representing actual primary occurrences of p.

Finding the right part of the pattern. To find $p_{i+1,m}$ we use a suffix trie that indexes all the suffixes of T starting at phrase boundaries. In the leaves of the trie we store the identifiers of the phrases where the corresponding suffixes start. The identifiers form an array id that stores the phrase identifiers in lexicographic order of the suffixes they start, requiring $n' \lceil \log n' \rceil$ bits.

We represent the suffix trie as a Patricia tree [46], encoded using a succinct representation for labeled trees called *dfuds* [47]. As the trie has at most $2n'$ nodes, the succinct representation requires at most $2n' \log \sigma + O(n')$ bits. It supports a large number of operations in constant time, such as going to a child labeled c, going to the leftmost and rightmost descendant leaf, etc. To search for $p_{i+1,m}$ we descend through the tree using the next character of the pattern, skip as many characters as the skip value of the child indicates, and repeat the process until determining that $p_{i+1,m}$ is not in the set, or until reaching a node or an edge whose leftmost and rightmost subtree leaves define the interval in array id of suffixes that start with $p_{i+1,m}$.

Example 4.2. *Figure 2(b) shows, on top, the trie of all suffixes of T starting at a phrase boundary, shading the range [8,9] of leaves found when searching for* $p_{i+1,m} =$ '**`la`**'.

Recall that, in a Patricia tree, after searching for the positions we need to check if they are actually a match, as some characters are not checked because of the skips. Instead of doing the check at this point, we defer it for later, when we connect both searches.

We do not explicitly store the skips, as they may take much space and can be computed from the trie and the text. Given a node in the trie corresponding to a string of length ℓ, we go to the leftmost and rightmost leaves and extract the corresponding suffixes from their $(\ell + 1)$th symbols. The number s of symbols they share from that position is the skip. This takes $O(sh)$ time for LZ77. For LZ-End this is not $O(s + h)$ because extraction goes from left to right and we extract one character at a time until they differ. We can achieve $O(s + h \log s)$ by extracting substrings of doubling lengths. Still, the descent may be formed by $\Theta(m)$ steps with skips of size $s = O(1)$, and thus the total time for extracting the skips as we descend is $O(mh)$ in any case.

Finding the left part of the pattern. Another Patricia trie indexes all the reversed phrases, stored in the same way as the suffix trie. To find the left part of the pattern we search this trie for $(p_{1,i})^{rev}$, that is, $p_{1,i}$ read backwards. The array with the phrase identifiers at the leaves of the trie is called *rev_id*, but it is not stored explicitly. The space is at most $2n' \log \sigma + O(n')$ bits.

Example 4.3. *Figure 2(b) shows, on the left, the trie of reverse phrases of T, with the result of searching for a left part* $p_{1,i} =$ '**`a`**'.

For this trie we do store the skips. As all the strings inserted in the trie add up to length n, and the trie has at most n' internal nodes, we can concatenate the skip values in unary, $0^{skip}1$, forming a bitmap of length n with at most n' bits set. This can be stored in compressed form [44] so that any skip can be extracted in constant time using $select_1$ and the bitmap space is at most $n' \log \frac{n}{n'} + O(n') + o(n)$ bits. The mapping from a node to an internal node rank i is easily computed in constant time using *dfuds* [47]. Thus the searches on this trie take $O(m)$ time.

Connecting both searches. Actual occurrences of p are those formed by a phrase $rev_id[j] = k - 1$ and the following one $id[i] = k$, so that j and i belong to the lexicographical intervals found with the tries. To find those we

use a $n' \times n'$ range structure that connects the consecutive phrases in both trees. If $id[i] = k$ and $rev_id[j] = k - 1$, the structure holds a point in (i, j).

The range structure is represented compactly using a wavelet tree [10, 48], which requires $n' \log n' + O(n' \log \log n')$ bits, or even $n' \log n' + o(n')$ [49]. The wavelet tree stores the permutation $R[1, n']$ so that $R[i] = j$ if (i, j) is a point (note there is only one j per i value). It can compute any $R[i]$ and $R^{-1}[j]$ in $O(\log n')$ time, as well as find all the occ points in a given orthogonal range in time $O((occ + 1) \log n')$. With such an orthogonal range search for the intervals of leaves found in both trie searches, the wavelet tree gives us all the primary occurrences. It also computes any $rev_id[j] = id[R^{-1}[j]] - 1$ in $O(\log n')$ time, thus we do not need to store rev_id.

Example 4.4. *Figure 2(b) shows sequence R at the bottom. It also shows how we find the only primary occurrence of $p =$ '`ala`' by partitioning it into '`a`' and '`la`'.*

At this stage we also verify that the answers returned by the Patricia tree searches are valid. By the Patricia tree properties, it is sufficient to extract the text of one of the occurrences reported and compare it to p, to determine that either all or none of the answers are valid. This costs $O(mh)$ time for LZ77 and $O(m + h)$ for LZ-End.

Note that the structures presented up to now are sufficient to determine whether the pattern exists in the text or not, since p cannot appear if it does not have primary occurrences. If we have to report the occ occurrences, instead, we use bitmap B: An occurrence with partition $p_{1,i}$ and $p_{i+1,m}$ found at $rev_id[j] = k$ is to be reported starting at text position $select_1(B, k) - i + 1$.

Overall, the data structures introduced in this section add up to $n' \log n + n' \log n' + 4n' \log \sigma + O(n') + o(n)$ bits. The occ primary occurrences are found in time $O(m^2 h + m \log n' + occ \log n')$.

Implementation considerations. As most of the skips are usually very small and computing them from the text phrases is slow in practice, we actually store the skips for both tries, using *Directly Addressable Codes* [50]. These allow storing variable-length codes while retaining fast direct access. The time complexity $O(m^2 h)$ drops to $O(m(m + h))$ on LZ-End, yet the space for the skips on the suffix trie can be $n' \log n$ bits in the worst case.

We use a practical *dfuds* implementation [51] that binary searches for the child labeled c, as the theoretical one [47] uses perfect hashing.

Instead of storing the tries we can do a binary search over the *id* or *rev_id* arrays. This alternative modifies the complexity of searching for a prefix/suffix of p to $O(mh \log n')$ for LZ77 or $O((m+h)\log n')$ for LZ-End (actually, since we extract the phrases right-to-left, binary search on the reverse trie costs $O(m \log n')$ for LZ-End). Independently, we could store explicitly array *rev_id*, instead of accessing it through the wavelet tree. Although this increases the space usage of the index and does not improve the complexity, it gives a relevant tradeoff in practice.

4.3. Secondary Occurrences

Secondary occurrences are found from the primary occurrences and, recursively, from other previously discovered secondary occurrences. The idea is to locate all the sources covering the occurrence and then finding their corresponding phrases. Each copy found is reported and recursively analyzed for sources containing it.

For each occurrence found $T[i, i+m-1]$, we find the position *pos* of the 0 corresponding to its starting position in bitmap S, $pos = select_0(S, i+1)$. Then we consider all the 1s to the left of *pos*, looking for sources that start before the occurrence. For each such $S[j] = 1$, $j < pos$, the source starts in T at $t = rank_0(S, j)$ and is the sth source, for $s = rank_1(S, j)$. Its corresponding phrase is $f = P^{-1}[s]$, which starts at text position $c = select(B, f-1) + 1$. Now we compute the length of the source, which is the length of its phrase minus one, $l = select_1(B, f) - select_1(B, f-1) - 1$. Finally, if $T[t, t+l-1]$ covers the occurrence $T[i, i+m-1]$, then this occurrence has been copied to $T[c+i-t, c+i-t+m-1]$, where we report a secondary occurrence and recursively find sources covering it. The time per occurrence reported is dominated by that of computing P^{-1}, $O(\log n')$.

Example 4.5. *Consider in Figure 2(a) the only primary occurrence of pattern* **'la'** *starting at position 2 in our example text. We find the third 0 in the bitmap of sources at position 12. Then we consider all 1s starting from position 11 to the left. The 1 at position 11 maps to a phrase of length 2 that covers the occurrence, hence we report an occurrence at position 10. The second 1 maps to a phrase of length 6 that also covers the occurrence, thus we report another occurrence at position 15. The third 1 maps to a phrase of length 1, hence it does not cover the occurrence and we do not report it. We proceed recursively for the occurrences found at positions 10 and 15.*

Unfortunately, we do not know when to stop looking for 1s to the left of *pos* in S. Stopping at the first source not covering the occurrence works only when no source contains another.

Example 4.6. *Let us start with the primary occurrence of the pattern* `'ba'` *starting at position 4. The first source to the left is* `'la'`*, at position 2 and of length 2, which does not cover the pattern. Assume we stop, reporting no secondary occurrences. Then we miss the source* `'alabar'` *to the left, that does cover the pattern and generates the secondary occurrence starting at position 16.*

We present now a general solution that requires just $2n' + o(n')$ extra bits and reports the *occ* secondary occurrences in time $O(occ \log n')$.

Consider a (virtual) array $E[1, n']$ where $E[s]$ is the text position where the sth source ends. Then an occurrence $T[i, i+m-1]$ is covered by source s if $s \leq e = rank_1(S, pos)$ (i.e., s starts at or before i in T) and $E[s] \geq i + m - 1$ (i.e., s ends at or after $i + m - 1$ in T). Then we must report all values $E[1, e] \geq i + m - 1$. Figure 2(a) shows E on our running example.

A *Range Maximum Query (RMQ)* data structure can be built on $E[1, n']$ so that it (*i*) occupies $2n' + o(n')$ bits of space; (*ii*) answers in constant time queries $\text{RMQ}_E(i, j) = \arg\max_{i \leq k \leq j} E[k]$; (*iii*) it does *not* access E for querying [52]. We build such a data structure on E. The array E itself is not represented; any desired value can be computed as $E[s] = t + l - 1$, using the nomenclature given at the beginning of this subsection, in time $O(\log n')$ as it involves computing $P^{-1}[s]$.

Thus $k = \text{RMQ}_E(1, e)$ gives us the rightmost-ending source among those starting at or before i. If $E[k] < i + m - 1$ then no source in $[1, e]$ covers the occurrence. Else, we report the copied occurrence within phrase $P^{-1}[k]$ (and process it recursively), and now consider the intervals $E[1, k-1]$ and $E[k+1, e]$, which are in turn recursively processed with RMQs until no source covering the occurrence is found. This algorithm was already described by Muthukrishnan [53], who showed that it takes $2 \cdot occ$ computations of RMQ to report *occ* occurrences. Each step takes us $O(\log n')$ time due to the need to compute the $E[k]$ values. Figure 3(a) gives a pseudocode.

4.4. The Previous Smaller Value Problem

The best implemented RMQ-based solution requires in practice around $3n'$ bits and a constant but significant number of complex operations [52,

SecondaryOcc(i,m)

```
1  pos ← select_0(S, i+1)
2  e ← pos − i − 1
3  FindSources(i,m,1,e)
```

FindSources(i,m,s,e)

```
1  k ← RMQ_E(s, e)
2  t ← select_1(S, k) − k
3  f ← P^{-1}[k]
4  c ← select_1(B, f − 1) + 1
5  l ← select_1(B, f) − c
6  if t + l ≥ i + m − 1 then
7     occ_pos ← c + i − t
8     report occ_pos
9     SecondaryOcc(occ_pos,m)
10    FindSources(i,m,s,k − 1)
11    FindSources(i,m,k + 1,e)
```

(a) Reporting secondary occurrences triggered by occurrence $T[i,i+m-1]$.

PSV(τ,d,s,a,b)

```
1  if τ is a leaf then
2     return s
3  c ← ⌊(a + b)/2⌋
4  if d − 1 ≤ c then
5     p ← PSV(τ.l, d, rank_0(τ.V, s), a, c)
6     return select_0(τ.V, p)
7  else
8     v_0 ← select_0(τ.V, rank_0(τ.V, s))
9     p ← PSV(τ.r, d, rank_1(τ.V, s), c+1, b)
10    return max(v_0, select_1(τ.V, p))
```

(b) Solving $PSV(D,s,d)$ on the wavelet tree τ of D, with bitmap V and children l and r. The initial invocation is $\mathbf{PSV}(\tau,d,s-1,0,\delta)$.

Figure 3: Algorithms for reporting secondary occurrences and for the PSV problem.

54]. We present now an alternative development that, although offering worse worst-case complexities, requires in practice reasonable space, 2.88–$4.08n'$ bits, and is faster: On the machine described in Section 6, it takes 1–3 microseconds in total per secondary occurrence, whereas just one RMQ computation takes more than 1.5 microseconds, and we need two of them per occurrence, still ignoring the time to compute $E[k]$ values indirectly. It has, moreover, independent interest.

In early attempts to solve the problem of reporting secondary occurrences, Kärkkäinen [37] introduced the concept of *levels*. We use it in a different way.

Definition 4.7. *Source $s_1 = [l_1, r_1]$ is said to* cover *source $s_2 = [l_2, r_2]$ if $l_1 < l_2$ and $r_1 > r_2$. Let $cover(s)$ be the set of sources covering a source s. Then the* depth *of source s is defined as $depth(s) = 0$ if $cover(s) = \emptyset$, and $depth(s) = 1 + \max_{s' \in cover(s)} depth(s')$ otherwise. We define $depth(\varepsilon) = 0$. Finally, we call δ the maximum depth in the parsing.*

Example 4.8. *The four sources '`a`' and the source '`alabar`' have depth*

zero, as all of them start at the same position. Source **'la'** *has depth 1, as it is contained by source* **'alabar'**.

Assume that sources starting at the same point are sorted by shortest length first. We traverse S leftwards from *pos*. When we find a source not covering the occurrence, we look for its depth d and then consider to the left only sources with depth $d' < d$, as those at depth $\geq d$ are guaranteed not to contain the occurrence: sources to the left with the same depth d will not end after the current source, and deeper sources to the left will be contained in those of depth d. Thus for our traversal we need to solve a subproblem we call *Previous Smaller Value*, $PSV(D, s, d)$, on our array of depths $D[1, n']$.

Definition 4.9. *Let $D[1, n']$ be an array of values over $[0, \delta]$. Then the* Previous Smaller Value *problem is, given a position $s \in [1, n']$ and a value $d \in [0, \delta]$, find $PSV(D, s, d)$, the largest $s' < s$ such that $D[s'] < d$.*

Do not confuse this problem with a simpler variant [55] where $d = D[s]$, nor with another [56] where we give a range $D[s, e]$ and want the largest value smaller than d in the range.

We solve the PSV problem by representing D using a wavelet tree [10]. This time we need to explain its internal structure. The wavelet tree is a balanced tree where each node represents a range of the alphabet $[0, \delta]$. The root represents the whole range and each leaf an individual alphabet member. Each internal node has two children that split its alphabet range $[a, b]$ by half, into $[a, \lfloor (a + b)/2 \rfloor]$ and $[\lfloor (a + b)/2 \rfloor + 1, b]$. Hence the tree has height $\lceil \log(\delta + 1) \rceil$. At the root node, the tree stores a bitmap aligned to D, with a 0 at position i if $D[i]$ is a symbol belonging to the range of the left child, and 1 if it belongs to the right child. Recursively, each internal node stores a bitmap that refers to the subsequence of D formed by the symbols in its range. All the bitmaps are preprocessed for rank/select queries, needed for navigating the tree. The total space is $n' \log(\delta + 1) + o(n')$ bits [49].

We solve $PSV(D, s, d)$ as follows. We descend on the wavelet tree towards the leaf that represents $d-1$. If $d-1$ is to the left of the current node, then no interesting values can be stored in the right child. So we recursively continue in the left subtree, at position $s' = rank_0(V, s)$, where V is the bitmap of the current node. Otherwise we descend to the right child, where the new position is $s' = rank_1(V, s)$. In this case, however, the answer could be at the left child. Any value stored at the left child is $< d$, so we are interested in the rightmost before position s. Hence $v_0 = select_0(V, rank_0(V, s - 1))$

is the last position with a value from the left subtree. We find, recursively, the best answer v_1 from the right subtree, and return $\max(v_0, v_1)$. When the recursion arrives at a leaf we return 0. The running time is $O(\log \delta)$. Figure 3(b) gives a pseudocode.

Theorem 4.10. *The PSV problem can be solved in time $O(\log \delta)$ using a data structure that uses $n' \log(\delta + 1) + o(n')$ bits of space. This structure can replace D as it delivers any value $D[i]$ in time $O(\log \delta)$.*

Using this operation we proceed as follows. We keep track of the smallest depth d that cannot cover our occurrence; initially $d = \delta + 1$. We start considering source s. Whenever s covers the occurrence, we report it, else we set $d = D[s]$. In both cases we proceed to $s' = PSV(D, s, d)$, until $s' = 0$.

In the worst case the first source is at depth δ and then we traverse level by level, finding in each level that the previous source does not contain the occurrence. Therefore the overall time is $O(occ(\log n' + \delta \log \delta))$ to find occ secondary occurrences. This worst case is, however, rather unlikely. Moreover, in practice δ is small: it is also limited by the maximum phrase length, and in our test collections it is at most 46 and on average 1–4.

4.5. The Final Picture

The final result is summarized in the following theorem.

Theorem 4.11. *Let $T[1, n]$ be a text over alphabet $[1, \sigma]$, parsed by LZ77 or LZ-End into n' phrases, and let h be the height of the parsing. Then there exists a data structure using $3n' \log n + 5n \log \sigma + O(n') + o(n)$ bits of space, that can find the occ occurrences of any pattern $p_{1,m}$ in time $O(m^2 h + (m + occ) \log n')$. It can also reproduce any substring of T of length ℓ in time $O(\ell + h)$ (LZ-End parsing) or $O(\ell h)$ (LZ77 parsing).*

Implementation considerations. The term $o(n)$ in the space complexity can be as low as $O(n/\log^c n)$ for any constant c [49]. Still, in practice it can be dominant on highly repetitive collections. It can be removed by using compressed bitmap representations that require $n' \log \frac{n}{n'} + O(n')$ bits [57], but do not operate in constant time. This change multiplies all time complexities by $O(\log \frac{n}{n'})$ (this is better than the results reported by Okanohara and Sadakane [57], and is obtained by using a constant-time rank/select structure for their internal bitmap H).

In practice, the bitmaps are so sparse that we achieve better space and similar time by delta-encoding the differences between consecutive positions of 1s and adding absolute pointers to regularly sampled positions.

5. Construction

In this section we describe the construction algorithms for the proposed parsing and self-index.

5.1. Parsing Algorithm

It is well known how to carry out the LZ77 parsing. In particular we use the algorithm CPS2 of Chen *et al.* [58]. This is based on maintaining a lexicographic range of all the suffixes of the text that start with pattern $p = T[i, i'-1]$ (recall Definition 2.1) and refining it as p grows rightwards. For the parsing of Fiala and Green [43] one must restrict these suffixes to those starting at phrase boundaries. For the LZ-End parsing, however, we maintain the range of all prefixes of T ending with p and at phrase boundaries (recall Definition 3.1). This is significantly more challenging. We describe next an algorithm based on CPS2 to carry out this parsing.

We first build the *suffix array* [8] $A[1, n]$ of the *reverse* text, $T^R = T[n-1] \ldots T[2]T[1]\$$, so that $T^R[A[i], n]$ is the lexicographically i-th smallest suffix of T^R. We also build its inverse permutation: $A^{-1}[j]$ is the lexicographic rank of $T^R[j, n]$. Finally, we build the *Burrows-Wheeler Transform (BWT)* [31] of T^R, $T^{bwt}[i] = T^R[A[i] - 1]$ (or $T^R[n]$ if $A[i] = 1$).

On top of the BWT we will apply *backward search*, by regarding the BWT as an FM-index [25]. This allows determining all the suffixes of T^R that start with a given pattern $p_{1,m}$ by scanning p backwards, as follows. Let $C[c]$ be the number of occurrences of symbols smaller than c in T. After processing $p_{i+1,m}$, indexes sp and ep will be such that $A[sp, ep]$ points to all the suffixes of T^R starting with $p_{i+1,m}$. Initially $i = m$ and $[sp, ep] = [1, n]$. Now, if the invariant holds for $p_{i+1,m}$, we have that the new indexes for $p_{i,m}$ are $sp' = C[p_i] + rank_{p_i}(T^{bwt}, sp - 1) + 1$ and $ep' = C[p_i] + rank_{p_i}(T^{bwt}, ep)$. Operation $rank_c(T^{bwt}, i)$ counts the occurrences of symbols c in $T^{bwt}[1, i]$. Let us call this function a *BWS step*, $[sp', ep'] = \textsc{bws}(sp, ep, p_i)$.

Backward search over T^R adapts very well to our purpose. By considering the patterns $p = (T[i, i'-1])^R$ and carrying out consecutive BWT steps for increasing values of i', we are searching backwards for p in T^R, and thus finding the *ending* positions of $T[i, i'-1]$ in T.

Yet, only those occurrences that finish before position i are useful. As we increase i' in $T[i, i'-1]$, we test whether $A[sp, ep]$ contains some occurrence finishing before i in T, that is, starting after $n + 1 - i$ in T^R. If it does not, then we stop looking for larger i' as there are no matches preceding

$T[i]$. For this, we precompute an RMQ data structure [52] on A. Then, if $A[\text{RMQ}_A(sp, ep)]$ is not large enough, we stop.

Moreover, for LZ-End, occurrences must in addition finish at a previous phrase boundary. Translated to T^R, this means the occurrence must start at some position $n+1-j$, where some previous $Z[p]$ ends at position j in T. We maintain a dynamic set $\mathcal{F}$ where we add the ending positions of the successive phrases we create, mapped to A. That is, once we create phrase $Z[p] = T[i, i']$, we insert $A^{-1}[n+1-i']$ into $\mathcal{F}$. Later, when we process a new phrase $T[i, i'-1]$ and increase i', we verify whether $\mathcal{F}$ contains some value in the corresponding range $[sp, ep]$. A *successor* query on $\mathcal{F}$ finds the smallest value $fpos \geq sp$ in $\mathcal{F}$. If $fpos \leq ep$, then it represents a suitable LZ-End source for $T[i, i']$. Otherwise, as the condition could hold again later, we do not stop but recall the last $j = i'$ so that the condition held for $T[i, j-1]$. Once we stop because no matches ending before $T[i]$ exist, we insert phrase $Z[p] = T[i, j]$ and continue from $i = j+1$. This may retraverse some text since we had processed up to $i' \geq j$. We call $N \geq n$ the total number of text symbols processed. The algorithm is depicted in Figure 1(b).

5.2. Parsing Analysis

The fastest construction time is achieved by using $O(n \log n)$ bits of space. In this case we can build and store explicitly A and A^{-1} in time $O(n)$ [59], as well as build the static BWT in time $O(n(1 + \frac{\log \sigma}{\log \log n}))$ [11], so that it can compute BWS steps in time $O(1 + \frac{\log \sigma}{\log \log n})$. The RMQ structure [52] is built in $O(n)$ time and the same final space of $2n + o(n)$ bits, and answers queries in constant time. We can use predecessor data structures [60] that support queries and updates in time $O(\log \log n)$. The construction time becomes $O(N(\frac{\log \sigma}{\log \log n} + \log \log n))$, dominated by the N BWS steps and queries to $\mathcal{F}$.

On the other hand, we can build the parsing within as little as $2nH_k(T) + O(n) + o(n \log \sigma)$ bits of space, for any $k = o(\log_\sigma n)$. In this case the time complexity is $O(n \frac{\log n \log \sigma}{(\log \log n)^2} + N \log n)$. This comes from building the BWT (i.e., the FM-index) incrementally within space $nH_k(T^{rev}) + o(n \log \sigma) = nH_k(T) + o(n \log \sigma)$ [25, Thm. A.3] in time $O(n \frac{\log n \log \sigma}{(\log \log n)^2})$ [61]. After the BWT construction is completed we make it static using $O(n) + o(n \log \sigma)$ temporary bits,[3] so that it supports BWS steps in time $O(1 + \frac{\log \sigma}{\log \log n})$. It also supports access to A and A^{-1} in time $O(\log n)$ by sampling one position

[3]We make static one node of the multiary wavelet tree at a time, by copying it to a

out of $\log n$ (at the expense of $O(n)$ bits of space) [11]. The successor data structure can be a simple balanced search tree, with leaves holding $\Theta(\log n)$ elements, so that the query and update times are $O(\log n)$ and the space is $n' \log n(1 + o(1))$ [62]. This is proved [63] to be $nH_k(T) + o(n \log \sigma)$, for any $k = o(\log_\sigma n)$, for any parsing where Lemma 3.8 holds. The time complexity is dominated by the incremental BWT construction plus the N accesses to A and A^{-1}, and successor queries on $\mathcal{F}$.

Note that a simplification of our construction algorithm, disregarding $\mathcal{F}$ (and thus $N = n$) builds the LZ77 parsing using just $nH_k(T) + O(n) + o(n \log \sigma)$ bits and time $O(n \log n(1 + \frac{\log \sigma}{(\log \log n)^2}))$, which is better than the best existing solutions [64, 58].

The times for LZ-End depend on N. Next we give an upper bound on it.

Lemma 5.1. *The amount of text retraversed at any step is less than $|Z[p]|$, for some $1 \leq p \leq n'$.*

Proof. Say the last valid match $T[i, j-1]$ was with suffix $Z[1] \ldots Z[p-1]$ for some p, thereafter we worked until $T[i, i'-1]$ without finding any other valid match, and then formed the phrase (with source $p-1$). Then we will retraverse $T[j+1, i'-1]$, which must be shorter than $Z[p]$ since otherwise $Z[1] \ldots Z[p]$ would have been a valid match. □

Therefore, the expected construction time is $O(n \log^2 n)$ on Markovian sources (using compressed space). In practice N is usually slightly larger than n: In our experiments it holds $1.05\,n \leq N \leq 1.37\,n$, but on some pathological texts the ratio can reach 10–14 [38, Sec. 4.5.2].

Implementation considerations. In practice, our construction works within space (measured in bytes) (1) n to maintain T explicitly; plus (2) $2.02n$ for the BWT (following Navarro's "large" FM-index implementation [65] that stores T^{bwt} explicitly; this supports BWS steps efficiently); plus (3) $4n$ for the explicit A; plus (4) $0.7n$ for Fischer's implementation of RMQ [66]; plus (5) n for a sampling-based implementation of inverse permutations [45] for A^{-1}; plus (6) $12n'$ for a balanced binary tree implementing $\mathcal{F}$. This adds up to $\approx 8n$ bytes (and $\approx 6n$ bytes for LZ77). A is built in time $O(n \log n)$; other construction times are $O(n)$. After this, the parsing time is $O(N \log n') = O(N \log n)$.

---

new static memory area. If $\sigma = O(1)$ the extra space is $O(n)$; else the wavelet tree has $\omega(1)$ levels and thus the extra memory for one level is $o(n \log \sigma)$.

5.3. Index Construction

As for the parsing, let us first give a construction using minimum time, and then another achieving minimum space.

For the fastest possible index construction, the space is dominated by the $O(n \log n)$ bits needed to build the parsing. Within this space we can easily complete the index construction in time $O(n' \log n) = O(n \log \sigma)$ (Lemma 3.9) for the Range data structure, plus $O(n)$ for the creation of the Patricia trees (by pruning a suffix tree, for example), plus $O(n)$ for creating compressed bitmaps, plus $O(n')$ for permutations and other minor structures. The total construction time can then be upper bounded by $O(n \log \sigma + N(\frac{\log \sigma}{\log \log n} + \log \log n))$. For LZ77 parsing this becomes just $O(n \log \sigma)$.

Instead, the construction in minimum space is dominated by the Patricia trees. These can be built in time $O(n' \log^{1+\varepsilon} n)$ using $O(n \log \sigma)$ bits, for any constant $\varepsilon > 0$ [67]. Within this space we can build all the other data structures in time $O(n + n' \log n) = O(n \log \sigma)$. Within this space we can also speed up the BWT construction to time $O(n \log \log \sigma)$ [68]. Therefore the parsing plus index construction time becomes $O(n \log^{\epsilon} n \log \sigma + N \log n)$. The following theorem gives a simplified summarization.

Theorem 5.2. *The self-index based on the LZ-End parsing can be built in $O(N(\log \sigma + \log \log n))$ time and $O(n \log n)$ bits of space, or $O(N \log^{1+\epsilon} n)$ time and $O(n \log \sigma)$ bits of space, for any constant $\epsilon > 0$. On the LZ77 parsing the times drop to $O(n \log \sigma)$ and $O(n \log^{1+\epsilon} n)$, respectively.*

Implementation considerations. We refer the reader to the full work [38, Sec. 5.5] for a description of how the index is built in practice.

6. Experimental Evaluation

We ran all our experiments on a 3.0 GHz Core 2 Duo processor with 4GB of main memory, running Linux 2.6.24 and `g++` (gcc version 4.2.4) compiler with `-O3` optimization.

From our testbed `http://pizzachili.dcc.uchile.cl/repcorpus.html` we have chosen four collections representative of distinct applications: `Cere` (37 DNA sequences of S. Cerevisiae), `Einstein` (the versions of the Wikipedia article on Albert Einstein up to Jan 12, 2010), `Kernel` (the 36 versions 1.0.x and 1.1.x of the Linux Kernel), and `Leaders` (pdf files of the CIA World Leaders report, from Jan 2003 to Dec 2009, converted with `pdftotext`).

Collection	Cere	Einstein	Kernel	Leaders
Size	440MB	446MB	247MB	45 MB
p7zip	1.14%	0.07%	0.81%	1.29%
repair	1.86%	0.10%	1.13%	1.78%
bzip2	2.50%	5.38%	21.86%	7.11%
ppmdi	24.09%	1.61%	18.62%	3.56%
lz77	1.48%	0.10%	1.35%	1.73%
lz-end	1.74%	0.10%	1.43%	1.93%
lz-begin	6.15%	4.27%	3.43%	7.97%
lz78	25.33%	9.29%	30.02%	15.89%

Collection	Cere	Einstein	Kernel	Leaders
Size	440MB	446MB	247MB	45 MB
LZIndex	79.37%	33.26%	85.94%	49.74%
ILZI	66.93%	13.28%	60.19%	35.08%
RLCSA	7.60%	0.23%	3.78%	3.32%
$RLCSA_{512}$	8.57%	1.20%	4.71%	4.20%
$LZ77_5$	3.74%	0.18%	3.32%	3.86%
$LZ77_1$	4.52%	0.32%	5.38%	6.50%
$LZ\text{-}End_5$	4.94%	0.25%	4.11%	5.12%
$LZ\text{-}End_1$	7.93%	0.43%	6.63%	8.58%

Table 1: Compression ratios achieved by various compressors (left) and self-indexes (right), as a percentage of a byte-based representation of the plain texts.

We have studied 5 variants of our indexes, from most to least space consuming: (1) with suffix and reverse Patricia trees; (2) binary search on explicit *id* array and reverse Patricia tree; (3) suffix Patricia tree and binary search on *rev_id*; (4) binary search on explicit *id* array and on *rev_id*; (5) binary search on implicit *id* and on *rev_id*. In addition we test parsings LZ77 and LZ-End, so for example $LZ\text{-}End_3$ means variant (3) on parsing LZ-End. In all cases we represent the skips and *rev_id* explicitly. We use δ-encoding with sampling step 16 for all the sparse bitmaps.

Table 1 (left) gives compression ratios achieved with an excellent Lempel-Ziv (`p7zip`, `www.7-zip.org`), grammar (`repair`, `www.cbrc.jp/~rwan/en/restore.html`), Burrows-Wheeler (`bzip2`, `www.bzip.org`), and statistical high-order compressor (`ppmdi`, `pizzachili.dcc.uchile.cl/utils/ppmdi.tar.gz`). Lempel-Ziv and grammar compressors capture repetitiveness, while Burrows-Wheeler captures only some due to the runs with limited contexts, and the statistical one is blind to repetitiveness.

We also include compressors `lz77`, `lz-end`, `lz-begin`, and `lz78`, implemented by ourselves. `lz-end` and `lz-begin` store *source* and L in plain form, and B using delta-encoding. `lz77` is similar but *source* refers to absolute text positions. `lz78` stores only *source* and L. We confirm that `lz78` is unsuitable for repetitive collections, and that `p7zip` uses more clever encodings than our naive `lz77`. Note that `lz-end` is very close to `lz77`, using at most 20% more space, while `lz-begin` can be significantly worse.

Table 1 (right) gives compression ratios achieved by self-indexes. We include the LZIndex [26] (with $\varepsilon = 1/128$) and the ILZI [69] (both based on LZ78), the RLCSA alone (which can count how many times a pattern appears in T but cannot locate the occurrences nor extract text at random), and

	LZ77 Index		LZ-End Index		RLCSA	
Collection	Time	Space	Time	Space	Time	Space
Cere	1.01	5.83	4.22	8.25	2.21	9.48
Einstein	0.47	5.83	3.30	8.10	2.71	9.48
Kernel	0.50	5.79	3.56	8.17	2.19	9.61
Leaders	0.42	5.76	2.83	8.17	1.30	9.20

Table 2: Construction time and space for the indexes. Times are in seconds per MB and spaces are the ratio between construction space and plain text space.

Collection	L	B	Patricia tree	Patricia skips	Reverse tree	Reverse skips	*rev_id*	Range	D	P	S
	LZ77										
Cere	1.87	8.60	12.72	6.85	12.42	6.45	13.06	13.72	3.55	14.12	6.64
Einstein	5.19	8.23	15.06	6.59	14.45	5.95	11.02	11.70	2.88	12.06	6.86
Kernel	5.09	7.47	13.41	6.01	13.25	5.72	12.71	13.37	3.37	13.78	5.82
Leaders	4.47	8.28	13.57	6.83	13.98	6.16	11.49	12.13	4.08	12.52	6.49
	LZ-End										
Cere	1.91	8.65	12.67	6.79	11.88	6.32	13.37	14.04	3.12	14.46	6.79
Einstein	5.16	8.23	14.71	6.57	13.98	5.84	10.97	11.59	3.32	12.00	7.60
Kernel	5.08	7.18	13.40	5.89	13.16	5.59	12.69	13.33	3.79	13.75	6.13
Leaders	4.50	8.18	13.59	6.53	13.99	6.15	11.58	12.19	3.46	12.62	7.20

Table 3: Space breakdown (in %) of LZ77 and LZ-End index structures.

RLCSA using a sampling of 512 (the minimum space that gives reasonable times for locating and extraction). We also show the most and least space consuming of our variants over both parsings.

Our least-space variants take 2.5–4.0 times the space of `p7zip`, and 1.8–2.5 times the space of `lz77`. They are also always smaller than RLCSA_{512} (up to 6.6 times less) and even competitive with the crippled self-index RLCSA-with-no-sampling. As `Einstein` is extremely compressible, it illustrates how the RLCSA achieves much compression in terms of the runs of Ψ, yet it is unable to compress the sampling despite many theoretical efforts [30]. Thus even a sparse sampling has a very large relative weight when the text is so repetitive. The data our index needs for locating and extracting, instead, is proportional to the compressed text size.

Table 2 shows the time and space required to build the indexes. Both the LZ77 and the LZ-End indexes require less construction space than the RLCSA. In addition, the LZ77 index is built much faster than the RLCSA.

Indexes LZ77_1 and LZ-End_1 are the largest, and closest to the theoretical description. The ratio between their size and that of compressors `lz77` and `lz-end` are 3.0–4.0 and 4.3–5.4, respectively. Table 3 gives a space breakdown of the structures that form these self-indexes, to help understand why the

ratio deviates from the 3 predicted by the asymptotic analysis. Structures that require space of the form $n' \log n'$ (rev_id, Range, P) take around 13% of the total space, whereas those requiring $n' \log \frac{n}{n'}$ bits (B, S, skips) require around 7%. Hence the main space term in the asymptotic analysis, $3n' \log n$, accounts for just about 60% of the actual space in practice. From the rest, 30% is in the $O(n' \log \sigma)$ term (L and Patricia trees) and 10% in structures not considered in the theory (D, suffix trie skips). A clever LZ77 compressor requiring $n' \log n + n' \log \sigma$ bits takes around 25% of the space of the self-index. This predicts a ratio of 4, closer to the practical figures.

Figure 4 shows times for extracting random snippets, and Figure 5 shows times for locating random patterns of length 10 and 20. We test RLCSA with various sampling rates (a smaller rate requires more space). It can be seen that our LZ-End-based index extracts text faster than the RLCSA, while for LZ77 the results are mixed. For locating, LZ77 dominates LZ-End, and it operates within much less space than the RLCSA, and is also faster in several cases. Yet, LZ77 and LZ-End performance worsens with m faster than RLCSA, as expected. On the other hand, direct comparisons between our self-indexes and grammar-based self-indexes [21, 22] show that ours use less space and are faster. See the extended version [38] for more results.

7. Conclusions and Future Work

We have presented the first self-index based on the LZ77 parsing, aimed at representing highly repetitive text collections in compressed form while offering efficient access to arbitrary substrings and indexed searches.

Our self-index is the first in achieving a space that is of the same order of the output of an LZ77 compressor, with asymptotic constant 3. Previous self-indexes for repetitive texts require space proportional to the number of runs in the BWT of the text [29], or the size of a grammar that generates the text [18], or the size of the more restricted LZ78 parsing of the text [24, 25, 26, 69]. All those measures of compressibility are weaker than the size of the LZ77 parse of the text.

Our index finds the occ occurrences of a pattern of length m in time $O(m^2 h + (m + occ) \log n)$, where n is the text size and h is a measure of the nesting of the parsing (h is logarithmic on n on Markovian sources). The extraction time for a substring of length ℓ is $O(\ell h)$. This is improved to $O(\ell + h)$ by using another parsing we introduce, LZ-End, which is shown to be coarsely optimal and conjectured to be 2-competitive with LZ77. The

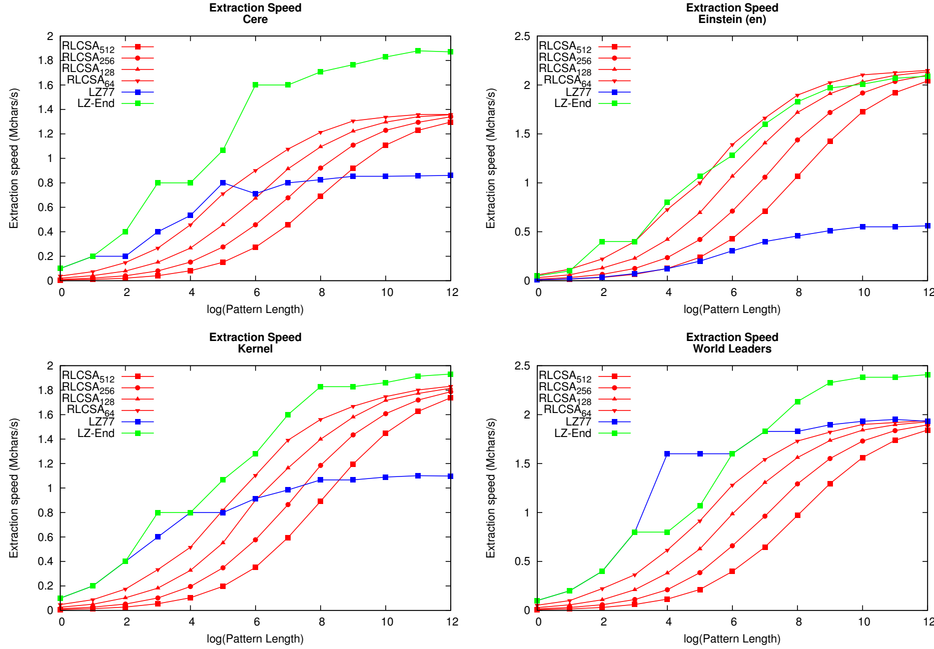


Figure 4: Extraction speed as a function of the snippet size (higher is better).

index can be built in linear space and time $O(n \log \sigma)$ with LZ77 parsing, and $O(N(\log \sigma + \log \log n))$ time with LZ-End parsing, where σ is the alphabet size and N is $O(n \log n)$ on Markovian sources.

In practical terms, our experiments show that our indexes provide a much better space/time trade-off than the previous ones for repetitive collections. The smallest variants of our self-index uses 1/10 of the space of LZ78-based ones, and 1/2 of that of runs- or grammar-based ones [21, 22], while achieving similar time efficiency (each occurrence of a short pattern, for example, is located in 10–50 microseconds). Compared to pure LZ77 compression, the smallest variant of our index takes 2.5-4.0 times the space achieved by `p7zip`. Our LZ-End parsing yields compression ratios close to those of LZ77 (no more than 20% extra), and extracts substrings up to 4 times faster, at around 2 million symbols per second. Our construction needs 6–8 times the original text size and indexes 0.2–2.0 MB/sec.

Our index is universal, in the sense that it compresses repetitive collections without knowing the versioning structure of the data, that is, which documents are near-copies of which. In several applications (particularly Bioinformatics), there is no such thing as a versioning structure.

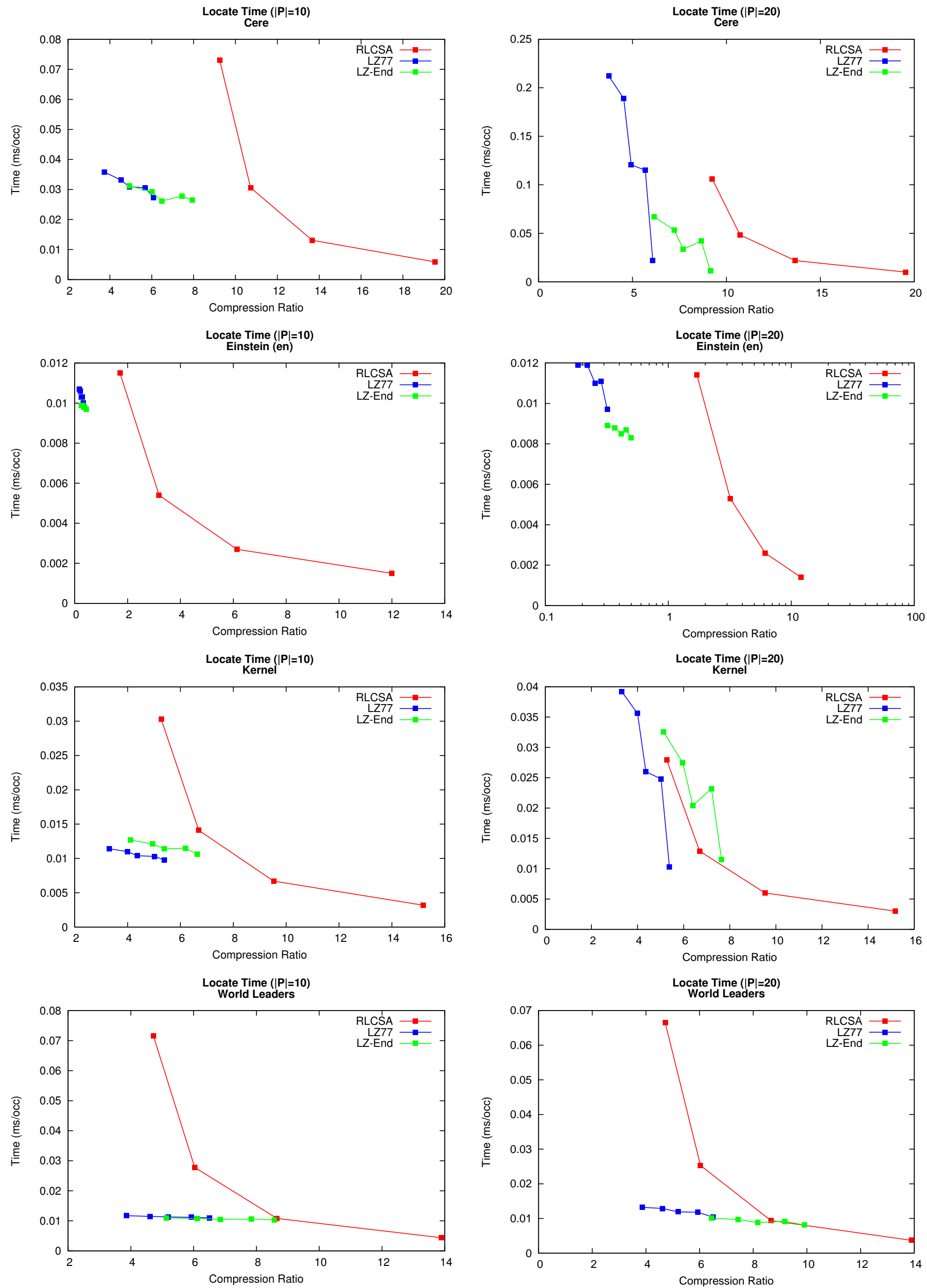


Figure 5: Time per located occurrence for $m = 10$ as a function of the space used by the index, in percentage of text size (lower and leftwards is better). The points for RLCSA refer to different sampling rates; for LZ77 and LZ-End the points refer to the 5 variants (LZ_5 is leftmost, LZ_1 is rightmost).

We have also created a text corpus oriented to repetitive texts, publicly available at `http://pizzachili.dcc.uchile.cl/repcorpus.html`. Our implementation is also public in there, to promote its use in real-world and research applications and to serve as a baseline for future developments.

There are several natural improvements to pursue. An obvious one is to get rid of the dependence on parameter h in the search and extraction time complexity. This is related to the more basic problem of providing direct access to Lempel-Ziv compressed text. This problem is currently active in the community and good solutions have been devised for the simpler cases of LZ78 parsing [23] and grammar compression [19]. Another undesirable factor is the dependence on m^2 in the search time. Such dependence is also present on some self-indexes built on the simpler LZ78 parsing [24, 26], but it has been reduced to linear in different ways [25, 70, 27].

With respect to construction, it is desirable that the time of the LZ-End parsing depends on n instead of N, or better bound N. Another problem is the space required. Our implementation builds the index using uncompressed space. Still switching to compact constructions [68, 61] poses the problem that the space is compressed in terms of, at best, the k-th order empirical entropy of the text, not in terms of the size of its LZ77 parse. For a repetitive sequence the difference can be huge, as shown in our experiments. A real solution would be a dynamic variant of our self-index, that supported at least adding text at the end of the collection. Then the index could be built within (Lempel-Ziv) compressed space by successive insertions.

Finally, an interesting open theoretical problem is to prove or disprove our conjecture about the 2-optimality of LZ-End with respect to LZ77 parsing, also considering the case of bounded alphabets.

References

[1] S. Kreft, G. Navarro, LZ77-like compression with fast random access, in: Proc. 20th Data Compression Conference (DCC'10), pp. 239–248.

[2] S. Kreft, G. Navarro, Self-indexing based on LZ77, in: Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM'11), LNCS 6661, pp. 41–54.

[3] N. Ziviani, E. S. de Moura, G. Navarro, R. Baeza-Yates, Compression: A key for next-generation text retrieval systems, IEEE Computer 33 (2000) 37–44.

[4] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Computing Surveys 39 (2007) article 2.

[5] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS'00), pp. 390–398.

[6] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC'00), pp. 397–406.

[7] A. Apostolico, The myriad virtues of subword trees, in: Combinatorial Algorithms on Words, NATO ISI Series, Springer-Verlag, 1985, pp. 85–96.

[8] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM Journal on Computing 22 (1993) 935–948.

[9] G. Manzini, An analysis of the Burrows-Wheeler transform, Journal of the ACM 48 (2001) 407–430.

[10] R. Grossi, A. Gupta, J. S. Vitter, High-order entropy-compressed text indexes, in: Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'03), pp. 841–850.

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

[12] T. Gagie, Large alphabets and incompressibility, Information Processing Letters 99 (2006) 246–251.

[13] C. Nevill-Manning, I. Witten, D. Maulsby, Compression by induction of hierarchical grammars, in: Proc. 4th Data Compression Conference (DCC'94), pp. 244–253.

[14] N. J. Larsson, A. Moffat, Off-line dictionary-based compression, Proc. IEEE 88 (2000) 1722–1732.

[15] A. Lempel, J. Ziv, On the complexity of finite sequences, IEEE Transactions on Information Theory 22 (1976) 75–81.

[16] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Transactions on Information Theory 23 (1977) 337–343.

[17] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Transactions on Information Theory 24 (1978) 530–536.

[18] F. Claude, G. Navarro, Self-indexed text compression using straight-line programs, in: Proc. 34th International Symposium on Mathematical Foundations of Computer Science (MFCS'09), LNCS 5734, pp. 235–246.

[19] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, O. Weimann, Random access to grammar-compressed strings, in: Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'11), pp. 373–389.

[20] S. Kuruppu, B. Beresford-Smith, T. Conway, J. Zobel, Repetition-based compression of large DNA datasets, in: Proc. 13th Annual International Conference on Computational Molecular Biology (RECOMB'09). Poster.

[21] F. Claude, A. Fariña, M. Martínez-Prieto, G. Navarro, Compressed q-gram indexing for highly repetitive biological sequences, in: Proc. 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE'10), pp. 86–91.

[22] F. Claude, A. Fariña, M. Martínez-Prieto, G. Navarro, Indexes for highly repetitive document collections, in: Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM'11), pp. 463–468.

[23] K. Sadakane, R. Grossi, Squeezing Succinct Data Structures into Entropy Bounds, in: Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'06), pp. 1230–1239.

[24] G. Navarro, Indexing text using the Ziv-Lempel trie, Journal of Discrete Algorithms 2 (2004) 87–114.

[25] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (2005) 552–581.

[26] D. Arroyuelo, G. Navarro, K. Sadakane, Reducing the space requirement of LZ-index, in: Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM), LNCS 4009, pp. 319–330.

[27] L. Russo, A. Oliveira, A compressed self-index using a Ziv-Lempel dictionary, Information Retrieval 5 (2008) 501–513.

[28] S. Kuruppu, S. Puglisi, J. Zobel, Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval, in: Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE'10), pp. 201–206.

[29] J. Sirén, N. Välimäki, V. Mäkinen, G. Navarro, Run-length compressed indexes are superior for highly repetitive sequence collections, in: Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE'08), LNCS 5280, pp. 164–175.

[30] V. Mäkinen, G. Navarro, J. Sirén, N. Välimäki, Storage and retrieval of highly repetitive sequence collections, Journal of Computational Biology 17 (2010) 281–308.

[31] M. Burrows, D. Wheeler, A block sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, 1994.

[32] R. Grossi, J. S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal of Computing 35 (2005) 378–407.

[33] V. Mäkinen, G. Navarro, Succinct suffix arrays based on run-length encoding, Nordic Journal of Computing 12 (2005) 40–66.

[34] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, Journal of Algorithms 48 (2003) 294 – 313.

[35] W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, Theoretical Computer Science 302 (2003) 211–222.

[36] J. Kärkkäinen, E. Ukkonen, Lempel-Ziv parsing and sublinear-size index structures for string matching, in: Proc. 3rd South American Workshop on String Processing (WSP'96), pp. 141–155.

[37] J. Kärkkäinen, Repetition-Based Text Indexes, Ph.D. thesis, Department of Computer Science, University of Helsinki, Finland, 1999.

[38] S. Kreft, Self-Index based on LZ77, MSc thesis, University of Chile, 2010. Available as Tech. Report TR/DCC-2011-13, Dept. of Computer Science, University of Chile, http://www.dcc.uchile.cl/TR/2011/DCC-20111220-013.pdf.

[39] M. Farach, M. Thorup, String matching in Lempel-Ziv compressed strings, in: Proc. 27th ACM Annual Symposium on the Theory of Computing (STOC), pp. 703–712.

[40] S. R. Kosaraju, G. Manzini, Compression of low entropy strings with Lempel-Ziv algorithms, SIAM Journal on Computing 29 (1999) 893–911.

[41] E. Plotnik, M. Weinberger, J. Ziv, Upper bounds on the probability of sequences emitted by finite-state sources and on the redundancy of the Lempel-Ziv algorithm, IEEE Transactions on Information Theory 38 (1992) 66–72.

[42] R. Hamming, Coding and Information Theory, Prentice-Hall, 1986.

[43] E. R. Fiala, D. H. Greene, Data compression with finite windows, Communications of the ACM 32 (1989) 490–505.

[44] R. Raman, V. Raman, S. S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02), pp. 233–242.

[45] J. I. Munro, R. Raman, V. Raman, S. S. Rao, Succinct representations of permutations, in: Proc. 30th International Colloquium on Automata, Languages and Computation (ICALP'03), LNCS 2719, pp. 345–356.

[46] D. Morrison, PATRICIA-Practical algorithm to retrieve information coded in alphanumeric, Journal of the ACM 15 (1968) 514–534.

[47] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, S. Rao, Representing trees of higher degree, Algorithmica 43 (2005) 275–292.

[48] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theoretical Computer Science 387 (2007) 332–347.

[49] M. Pătraşcu, Succincter, in: Proc. 49th IEEE Annual Symposium on Foundations of Computer Science (FOCS'08), pp. 305–313.

[50] N. Brisaboa, S. Ladra, G. Navarro, Directly addressable variable-length codes, in: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE'09), pp. 122–130.

[51] D. Arroyuelo, R. Cánovas, G. Navarro, K. Sadakane, Succinct trees in practice, in: Proc. 11th Workshop on Algorithm Engineering and Experiments (ALENEX'10), pp. 84–97.

[52] J. Fischer, Optimal succinctness for range minimum queries, in: Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN'10), LNCS 6034, pp. 158–169.

[53] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02), pp. 657–666.

[54] S. Gog, J. Fischer, Advantages of shared data structures for sequences of balanced parentheses, in: Proc. 20th Data Compression Conference (DCC'10), pp. 406–415.

[55] J. Fischer, V. Mäkinen, G. Navarro, Faster entropy-bounded compressed suffix trees, Theoretical Computer Science 410 (2009) 5354–5364.

[56] M. Crochemore, C. S. Iliopoulos, M. Kubica, M. S. Rahman, T. Walen, Improved algorithms for the range next value problem and applications, in: Proc. 25th International Symposium on Theoretical Aspects of Computer Science (STACS'08), pp. 205–216.

[57] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07).

[58] G. Chen, S. J. Puglisi, W. F. Smyth, Lempel-Ziv factorization using less time & space, Mathematics in Computer Science 1 (2008) 605–623.

[59] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP'03), LNCS 2719, pp. 943–955.

[60] D. Willard, Log-logarithmic worst-case range queries are possible in space $\Theta(n)$, Information Processing Letters 17 (1983) 81–84.

[61] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, CoRR 0905.0768v5 (2010).

[62] J. I. Munro, An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time, Journal of Computer System Sciences 33 (1986) 66–74.

[63] D. Arroyuelo, G. Navarro, K. Sadakane, Stronger Lempel-Ziv based compressed text indexing, Algorithmica (2011). To appear.

[64] D. Okanohara, K. Sadakane, An online algorithm for finding the longest previous factors, in: Proc. 16th Annual European Symposium on Algorithms (ESA'08), pp. 696–707.

[65] G. Navarro, Implementing the LZ-index: Theory versus practice, ACM Journal of Experimental Algorithmics 13 (2009) article 2.

[66] J. Fischer, V. Heun, A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array, in: Proc. 1st International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07), LNCS 4614, pp. 459–470.

[67] F. Claude, G. Navarro, Self-indexed grammar-based compression, Fundamenta Informaticae 111 (2010) 313–337.

[68] W.-K. Hon, K. Sadakane, W.-K. Sung, Breaking a Time-and-Space Barrier in Constructing Full-Text Indices, SIAM Journal on Computing 38 (2009) 2162–2178.

[69] L. M. S. Russo, G. Navarro, A. L. Oliveira, Fully-compressed suffix trees, in: 8th Latin American Symposium on Theoretical Informatics (LATIN'08), LNCS 4957, pp. 362–373.

[70] D. Arroyuelo, G. Navarro, Smaller and faster Lempel-Ziv indices, in: Proc. 18th International Workshop on Combinatorial Algorithms (IWOCA'07), pp. 11–20.