LZ77-like Compression with Fast Random Access *

Sebastian Kreft and Gonzalo Navarro
Dept. of Computer Science, University of Chile, Santiago, Chile
{skreft,gnavarro}@dcc.uchile.cl

Abstract

We introduce an alternative Lempel-Ziv text parsing, LZ-End, that converges to the entropy and in practice gets very close to LZ77. LZ-End forces sources to finish at the end of a previous phrase. Most Lempel-Ziv parsings can decompress the text only from the beginning. LZ-End is the only parsing we know of able of decompressing arbitrary phrases in optimal time, while staying closely competitive with LZ77, especially on highly repetitive collections, where LZ77 excells. Thus LZ-End is ideal as a compression format for highly repetitive sequence databases, where access to individual sequences is required, and it also opens the door to compressed indexing schemes for such collections.

1 Introduction

The idea of regarding compression as the default, instead of the archival, state of the data is gaining popularity. 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 [19, 15]. This poses new challenges, as now it is required that the compressed data should, at least, be accessible at random. In particular, the need to manage huge highly repetitive collections, rapidly arising in fields like bioinformatics, temporal databases, versioned document and software repositories, to name a few, is pushing in this direction. A recent work aiming at indexing large DNA databases of the same species (and hence highly repetitive) [17] found that LZ77 compression [18] was much superior than other techniques to capture this repetitiveness, yet it was inadequate as a format for compressed storage, due to its inability to retrieve individual sequences from the collection.

Decompressing LZ77-compressed data from the beginning is simple and fast. 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. Statistical compressors allow relatively simple extraction of arbitrary substrings, but

*Partially funded by Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile and, the first author, by Conicyt's Master Scholarship.

they do not capture long-range repetitions either. Only grammar-based compression can allow extraction of substrings while capturing long-range repetitions [4].

In this paper we introduce an alternative Lempel-Ziv parsing, *LZ-End*, which converges to the entropy and gets in practice very close to LZ77. LZ-End forces the source of a phrase to finish where some previous phrase ends, 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 a phrase boundary, so that they can be extracted optimally and fast in practice.

As a byproduct, we introduce a variant called *LZ-Cost* that can perform similarly while providing such guarantees, yet we have not devised an efficient parsing for it.

2 Basic Concepts

Definition 1 ([18])**.** *The* LZ77 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 occurs in* $T_{1,i-1}$[1]*, set* $Z[p] = T_{i,i'}$ *and continue with* $i = i'+1$*. The occurrence in* $T_{1,i-1}$ *of prefix* $T_{i,i'-1}$ *is called the* source *of the phrase* $Z[p]$*.*

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, $T_n = \$$. We use logarithms in base 2 throughout the paper.

The concept of coarse optimality captures the fact that the algorithm converges to the entropy, as proved for various Lempel-Ziv variants [9].

Definition 2 ([9])**.** *A parsing algorithm is said to be* coarsely optimal *if the 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*,* $\rho(T) \le H_k(T) + f_k(|T|)$*.*

3 LZ-End Compression

Definition 3. *The LZ-End 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* $T_{i,n}$ *that is a suffix of* $Z[1]\dots Z[q]$ *for some* $q < p$*, set* $Z[p] = T_{i,i'}$ *and continue with* $i = i'+1$*.*

The LZ-End parsing is similar to that by Fiala and Green [6], in that theirs restricts where the sources start, while ours restricts where the sources end. This is the key feature that will allow us extract arbitrary phrases in optimal time.

[1]The original definition allows the source of $T_{i,i'-1}$ to extend beyond position $i-1$, but we ignore this feature in this paper.

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 expanding an individual phrase $Z[p]$.

The LZ-End parsing, although potentially generating more phrases, permits a shorter encoding of each, of the form $z(p) = (q, \ell, c)$, such that the source of $Z[p] = T_{i,i'}$ is a suffix of $Z[1] \ldots Z[q]$, and the rest is as above. We introduce a more sophisticated encoding that will, in addition, allow us extract individual phrases in optimal time.

- $char[1, n']$ (using $n'\lceil \log \sigma \rceil$ bits) encodes the trailing characters (c above).
- $source[1, n']$ (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) marks the ending position of the phrases in T. This compressed representation [16] supports operations *rank* and *select* in constant time: $rank_B(i)$ is the number of 1s in $B[1, i]$; $select_B(j)$ is the position of the j-th 1 in B.

Thus we have $z(p) = (source[p], select_B(p + 1) - select_B(p) - 1, char[p])$.

3.2 Coarse Optimality

Lemma 1. *All the phrases generated by an LZ-End parse are different.*

Proof. Say $Z[p] = Z[p']$ for $p < p'$. Then, when $Z[p']$ was generated, a possible source yielding phrase $Z[p']c$, longer than $Z[p']$, was indeed $Z[p]$, a suffix of $Z[1] \ldots Z[p]$. $\square$

Theorem 2. *The LZ-End compression is coarsely optimal.*

The proof is easy following that given for LZ77 by Kosaraju and Manzini [9].

3.3 Extraction Algorithm

The algorithm to extract an *arbitrary* substring in LZ-End is self-explanatory and given in Figure 1 (left). While the algorithm works for extracting any substring, we can prove it is optimal when the substring ends at a phrase.

Theorem 3. *Function* Extract *outputs a text substring $T[start, end]$ ending at a phrase in time $O(end - start + 1)$.*

Proof. If $T[start, end]$ ends at a phrase, then $B[end] = 1$. We proceed by induction on $len = end - start + 1$. The case $len \leq 1$ is trivial by inspection. Otherwise, we output $T[end]$ at line 6 after a recursive call on the same phrase and length $len - 1$. This time we go to line 8. The current phrase (now $p+1$) starts at pos. If $start < pos$, we carry out a recursive call at line 10 to handle the segment $T[start, pos - 1]$. As

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

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

Figure 1: Left: LZ-End extraction algorithm for $T[start, start + len - 1]$. Right: LZ-End construction algorithm. $\mathcal{F}$ stores pairs ⟨phrase identifier, text position⟩ and answers successor queries on the text position.

this segment ends phrase p, induction shows that this takes time $O(pos - start + 1)$. Now the segment $T[\max(start, pos), end]$ is contained in $Z[p+1]$ and it finishes one symbol before the phrase ends. Thus a copy of it finishes where $Z[source[p+1]]$ ends, so induction applies also to the recursive call at line 13, which extracts the remaining string from the source instead of from $Z[p+1]$, also in optimal time. □

4 Construction Algorithm

We present an algorithm to compute the parsing LZ-End, inspired on algorithm CSP2 by Chen *et al.* [3]. We compute the range of all prefixes ending with a pattern P, rather than suffixes starting with P [6]. This is significantly more challenging.

We first build the *suffix array* [11] $SA[1, n]$ of the *reverse* text, $T^R = T_{n-1} \ldots T_2 T_1\$$, so that $T^R_{SA[i],n}$ is the lexicographically i-th smallest suffix of T^R. We also build its inverse permutation: $SA^{-1}[j]$ is the lexicographic rank of $T^R_{j,n}$; and the *Burrows-Wheeler Transform (BWT)* [2] of T^R, $T^{bwt}[i] = T^R[SA[i] - 1]$ (or $T^R[n]$ if $SA[i] = 1$).

On top of the BWT we will apply *backward search* [5]. 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 $SA[sp, ep]$ 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 $rank_c(L, i)$ counts the occurrences of symbols c in $L_{1,i}$. Let us call this a $BWS(sp, ep, P_i)$ *step*.

Backward search over T^R adapts very well to our purpose. By considering the

patterns $P = (T_{i,i'-1})^R$ for consecutive 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, by carrying out one further BWS step for each new i' value.

Yet, only those occurrences that finish before position i are useful. Moreover, for LZ-End, they must in addition finish at a previous phrase end. 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 SA. That is, once we create phrase $Z[p] = T_{i,i'}$, we insert $SA^{-1}[n+1-i']$ into $\mathcal{F}$.

As we advance i' in $T_{i,i'-1}$, we test whether $SA[sp, ep]$ 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 a *Range Maximum Query (RMQ)* data structure [7] on SA, which answers queries $mpos = \arg\max_{sp \le k \le ep} SA[k]$. Then if $SA[mpos]$ is not large enough, we stop.

In addition, we must know if i' finishes at some phrase end, i.e., if $\mathcal{F}$ contains some value in $[sp, ep]$. A *successor* query on $\mathcal{F}$ find the smallest value $fpos \ge sp$ in $\mathcal{F}$. If $fpos \le ep$, 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 $j = i'$ where it was valid. 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 $i' \ge j$. We call $N \ge n$ the total number of text symbols processed.

The algorithm is depicted in Figure 1 (right). In theory it can work within bit space $2n(H_k(T)+1) + o(n \log \sigma)$ and $O(n \log n \lceil \frac{\log \sigma}{\log \log n} \rceil + N \log^{1+\epsilon} n)$ time (details omitted). In practice, it works within byte space (1) n to maintain T explicitly; plus (2) $2.02n$ for the BWT (following Navarro's "large" FM-index implementation [14] that stores T^{bwt} explicitly; this supports BWS steps efficiently); plus (3) $4n$ for the explicit SA; plus (4) $0.7n$ for Fischer's implementation of RMQ [7]; plus (5) n for a sampling-based implementation of inverse permutations [12] for SA^{-1}; plus (6) $12n'$ for a balanced binary tree implementing $\mathcal{F}$. This adds up to $\approx 10n$ bytes. SA is built in time $O(n \log n)$; other construction times are $O(n)$. After this, the parsing time is $O(N \log n') = O(N \log n)$. As we see soon, N is usually (but not always) only slightly larger than n; we now prove it is $O(n \log n)$ under some assumptions.

Lemma 4. *The amount of text retraversed at any step is* $< |Z[p]|$ *for some* p.

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. □

Corollary 5. *In the worst case,* $N \le nL$*, where* L *is the longest phrase of the parsing. On a text coming from an ergodic Markov source, the expected value is* $N = O(n \log n)$*.*

5 Experimental Results

We implemented two different LZ-End encoding schemes. The first is as explained in Section 3.1. In the second (*LZ-End2*) we store the starting position of the source, $select_B(source[p])$, rather than the identifier of the source, $source[p]$. This in theory raises the $nH_k(T)$ term in the space to $2nH_k(T)$ (and noticeably in practice, as seen soon), yet we save one *select* operation at extraction time (line 13 in Figure 1 (left)), which has a significant impact in performance. In both implementations, bitmap B is represented by δ-encoding the consecutive phrase lengths. We store absolute samples $select_B(i \cdot s)$ for a sampling step s, plus pointers to the corresponding position in the δ-encoded sequence. Thus $select_B(i)$ is solved within $O(s)$ time and $rank_B(i)$, using binary search on the samples, in time $O(s + \log(n'/s))$, hence enabling a space-time trade-off related to s. Note $select_B(p)$ and $select_B(p+1)$ cost $O(1)$ after solving $p \leftarrow rank_B(end)$, in Figure 1 (left), thus LZ-End2 does no $select_B$ operations.

We compare our compressors with *LZ77* and *LZ78* implemented by ourselves. Compared to *P7Zip* (`http://www.7-zip.org`), LZ77 differs in the final encoding of the triples, which P7Zip does better. This is orthogonal to the parsing issue we focus on in this paper. We also implemented *LZB* [1], which limits the distance at which the phrases can be from their original (not transitive) sources, so one can decompress any window by starting from that distance behind; and *LZ-Cost*, a novel proposal where we limit the number of times any text character can be copied, thus directly limiting the maximum cost per character extraction. We have found no efficient parsing algorithm for *LZB* and *LZ-Cost*, thus we test them on small texts only. We also implemented *LZ-Begin*, the "symmetric" variant of LZ-End, which also allows optimal random phrase extraction. LZ-Begin forces the source of a phrase to start where some previous phrase starts, just like Fiala and Green [6], yet phrases have a leading rather than a trailing character. Although the parsing is much simpler, the compression ratio is noticeably worse than that of LZ-End, as we see soon (discussion of reasons omitted, but the reader can note that Lemma 1 does not hold). Finally, we tried *Re-Pair* [10], a grammar-based compressor (`http://www.cbrc.jp/~rwan/software/restore.html`).

We used the texts of the Canterbury corpus (`http://corpus.canterbury.ac.nz`), the 50MB texts from the PizzaChili corpus (`http://pizzachili.dcc.uchile.cl`), and highly repetitive texts Para and Cere[2], Coreutils[3], Kernel[4], CIA World Leaders[5], Wikipedia Einstein[6], Fibonacci sequences, and Rich String sequences [8]. We use a

[2] From the Durbin Research Group, `http://www.sanger.ac.uk/Teams/Team118/sgrp`

[3] All 5.x versions without the binary files, `ftp://mirrors.kernel.org/gnu/coreutils`

[4] All 1.0.x, 1.1.x and 1.2.x Linux kernel versions, `ftp://ftp.kernel.org/pub/linux/kernel`

[5] All documents until July 2009, converted to plain text, `https://www.cia.gov/library/publications/world-leaders-1`

[6] All versions of Einstein's article until July 7th 2009 with the XML markup removed, `http://en.wikipedia.org/w/index.php?title=Special:Export&pages=Albert_Einstein&offset=1&action=submit&history`

		LZ77	LZ78	LZ-End	LZ-Cost	LZB	LZ-Begin	Re-Pair
Canterbury	Size(kiB)							
alice29.txt	148.52	47.17	49.91	49.32	48.51	61.75	59.02	72.29
asyoulik.txt	122.25	51.71	52.95	53.51	52.41	66.42	62.34	81.52
cp.html	24.03	43.61	53.60	45.53	46.27	66.26	56.93	78.65
fields.c	10.89	39.21	54.73	41.69	44.44	61.32	60.61	65.19
grammar.lsp	3.63	48.48	57.85	50.41	56.30	67.02	67.14	85.60
lcet10.txt	416.75	42.62	46.83	44.65	43.44	56.72	54.21	57.47
plrabn12.txt	470.57	50.21	49.34	52.06	50.83	63.55	59.15	74.32
xargs.1	4.13	57.87	65.38	59.56	59.45	86.37	73.14	107.33
aaa.txt	97.66	0.055	0.508	0.045	1.56	0.95	0.040	0.045
alphabet.txt	97.66	0.110	4.31	0.105	0.23	1.15	0.100	0.081
random.txt	97.66	107.39	90.10	105.43	107.40	121.11	106.9	219.24
E.coli	4529.97	34.13	27.70	34.72	-	-	35.99	57.63
bible.txt	3952.53	34.18	36.27	36.44	-	-	43.98	41.81
world192.txt	2415.43	29.04	38.52	30.99	-	-	41.52	38.29
pi.txt	976.56	55.73	47.13	55.99	-	-	57.36	108.08
Pizza Chili	Size(MiB)							
Sources	50	28.50	41.14	31.00	-	-	41.95	31.07
Pitches	50	44.50	59.30	45.78	-	-	57.22	59.90
Proteins	50	47.80	53.20	47.84	-	-	54.95	71.29
DNA	50	31.88	28.12	32.76	-	-	34.28	45.90
English	50	31.12	41.80	31.12	-	-	38.54	30.50
XML	50	17.00	21.24	17.64	-	-	25.49	18.50
Repetitive	Size(MiB)							
Wikipedia Einstein	357.40	0.0997	9.29	0.1009	-	-	4.27	0.1038
CIA World Leaders	40.65	1.727	15.89	1.930	-	-	7.97	1.887
Rich String 11	48.80	3.197×10^{-4}	0.815	4.180×10^{-4}	-	-	0.01	3.752×10^{-4}
Fibonacci 42	255.50	7.319×10^{-5}	0.398	5.323×10^{-5}	-	-	6.07×10^{-5}	2.128×10^{-5}
Para	409.38	2.09	25.49	2.48	-	-	7.29	2.74
Cere	439.92	1.48	25.33	1.74	-	-	6.15	1.86
Coreutils	195.77	3.18	27.57	3.35	-	-	7.33	2.54
Kernel	246.01	1.35	30.02	1.43	-	-	3.43	1.10

Table 1: Compression ratio of different parsings, in percentage of compressed over original size. We use parameter $cost = (\log_2 n)/2$ for *LZ-Cost* and $dist = n/5$ for *LZB*.

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.

5.1 Compression Ratio

Table 1 gives compression ratios for the different collections and parsers. For LZ-End we omit the sampling for bitmap B, as it can be reconstructed on the fly at load time. LZ-End is usually 5% worse than LZ77, and at most 10% over it on general texts and 20% on the highly repetitive collections, where the compression ratios are nevertheless excellent. LZ78 is from 20% better to 25% worse than LZ-End on typical texts, but it is orders of magnitude worse on highly repetitive collections. With parameter $\log_2(n)/2$, LZ-Cost is usually close to LZ77, yet some times it is much worse, and it is never better than LZ-End except for negligible margins. LZB is not competitive at all. Finally, LZ-Begin is about 30% worse on typical texts, and up to 40 times worse for repetitive texts. This shows that LZ-End achieves very competitive compression ratios, even in the challenging case of highly repetitive sequences, where LZ77 excells.

Re-Pair shows that grammar-based compression is a relevant alternative. Yet, we note that it is only competitive on highly repetitive sequences, where most of the compressed data is in the dictionary. This implementation applies sophisticated compression to the dictionary, which we do not apply on our compressors, and which

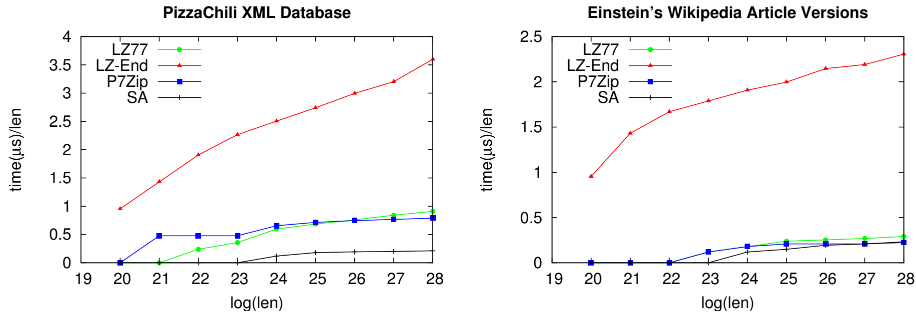

Figure 2: Parsing times for XML and Wikipedia Einstein, in microseconds per character.

prevents direct access to the grammar rules, essential for extracting substrings.

5.2 Parsing Time

Figure 2 shows construction times on two files for LZ77 (implemented following CSP2 [3]), LZ-End with the algorithm of Section 4, and P7Zip. We show separately the time of the suffix array construction algorithm we use, `libdivsufsort` (`http://code.google.com/p/libdivsufsort`), common to LZ77 and LZ-End.

Our LZ77 construction time is competitive with the state of the art (P7Zip), thus the excess of LZ-End is due to the more complex parsing. Least squares fitting for the nanoseconds/char yields $10.4 \log n + O(1/n)$ (LZ77) and $82.3 \log n + O(1/n)$ (LZ-End) for Einstein text, and $32.6 \log n + O(1/n)$ (LZ77) and $127.9 \log n + O(1/n)$ (LZ-End) for XML. The correlation coefficient is always over 0.999, which suggests that $N = O(n)$ and our parsing is $O(n \log n)$ time in practice. Indeed, across all of our collections, the ratio N/n stays between 1.05 and 1.37, except on `aaa.txt` and `alphabet.txt`, where it is 10–14 (which suggests that $N = \omega(n)$ in the worst case).

The LZ-End parsing time breaks down as follows. For XML: BWS 36%, RMQ 19%, tree operations 33%, SA construction 6% and inverse SA lookups 6%. For Einstein: BWS 56%, RMQ 19%, tree operations 17%, and SA construction 8%.

5.3 Text Extraction Speed

Figure 3 shows the extraction speed of *arbitrary* substrings of increasing length. The three parsings are parameterized to use approximately the same space, 550kiB for Wikipedia Einstein and 64MiB for XML. It can be seen that (1) the time per character stabilizes after some extracted length, as expected, (2) LZ-End variants extract faster than LZ77, especially on very repetitive collections, and (3) LZ-End2 is faster than LZ-End, even if the latter invests its better compression in a denser sampling.

We now set the length to 1,000 and measure the extraction speed per character, as a function of the space used by the data and the sampling. Here we use bitmap B and its sampling for the others as well. LZB and LZ-Cost have also their own

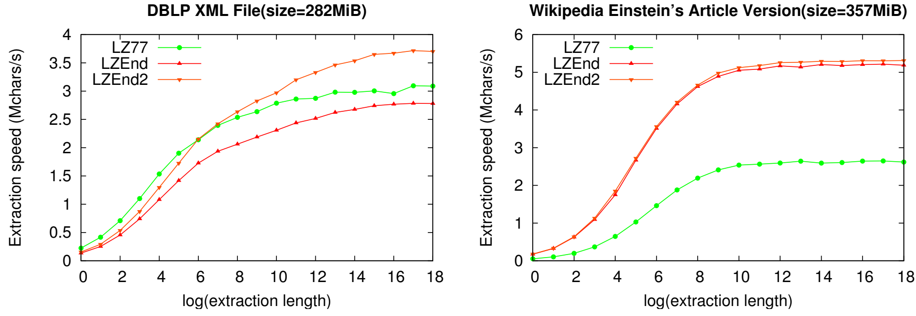

Figure 3: Extraction speed vs extracted length, for XML and Wikipedia Einstein.

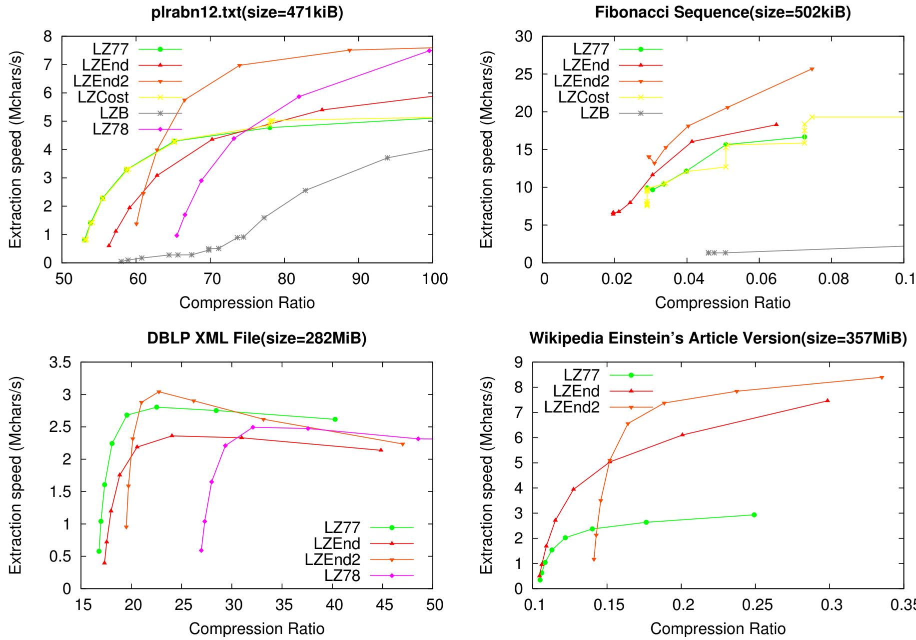

Figure 4: Extraction speed vs parsing and sampling size, on different texts.

space/time trade-off parameter; we tried several combinations and chose the points dominating the others. Figure 4 shows the results for small and large files.

It can be seen that LZB is not competitive, whereas LZ-Cost follows LZ77 closely (while offering a worst-case guarantee). The LZ-End variants dominate all the trade-off except when LZ77/LZ-Cost are able of using less space. On repetitive collections, LZ-End2 is more than 2.5 times faster than LZ77 at extraction.

6 Future Work

LZ-End opens the door to compressed indexed schemes for highly repetitive collections based on Lempel-Ziv, as advocated in previous work [17, 4]. Indeed, there exists already a proposal [13] for a LZ77-based self-index, whose only drawback is that it

needs to extract random text substrings along the search process. This is now feasible with LZ-End parsing, thus our next goal is to implement such a self-index. This would provide not only access to the compressed data, but also efficient indexed search.

References

[1] M. Banikazemi. LZB: Data compression with bounded references. In *Proc. DCC*, page 436, 2009. Poster.

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

[3] G. Chen, S. Puglisi, and W. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, June 2008.

[4] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *Proc. 34th MFCS*, pages 235–246, 2009.

[5] P. Ferragina and G. Manzini. Indexing compressed texts. *JACM*, 52(4):552–581, 2005.

[6] E. Fiala and D. Greene. Data compression with finite windows. *CACM*, 32(4):490–505, 1989.

[7] J. Fischer and V. Heun. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proc. 1st ESCAPE*, pages 459–470, 2007.

[8] F. Franek, R. Simpson, and W. Smyth. The maximum number of runs in a string. In *Proc. AWOCA*, pages 26–35, 2003.

[9] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM J. Comp.*, 29(3):893–911, 1999.

[10] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[11] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comp.*, 22(5):935–948, 1993.

[12] J. Munro, R. Raman, V. Raman, and S. S. Rao. Succinct representations of permutations. In *Proc. 30th ICALP*, pages 345–356, 2003.

[13] G. Navarro. Indexing LZ77: The next step in self-indexing. Keynote talk at the *Third Workshop on Compression, Text, and Algorithms*, Melbourne, Australia, 2008.

[14] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM JEA*, 13(2), 2009.

[15] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM CSUR*, 39(1):article 2, 2007.

[16] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th SODA*, pages 233–242, 2002.

[17] J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proc. 15th SPIRE*, pages 164–175, 2008.

[18] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE TIT*, 23(3):337–343, 1977.

[19] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.