BAT-LZ out of hell

Zsuzsanna Lipták
Dipartimento di Informatica, University of Verona, Italy

Francesco Masillo
Dipartimento di Informatica, University of Verona, Italy

Gonzalo Navarro
Center for Biotechnology and Bioengineering (CeBiB), Department of Computer Science, University of Chile, Chile

Abstract

Despite consistently yielding the best compression on repetitive text collections, the Lempel-Ziv parsing has resisted all attempts at offering relevant guarantees on the cost to access an arbitrary symbol. This makes it less attractive for use on compressed self-indexes and other compressed data structures. In this paper we introduce a variant we call BAT-LZ (for Bounded Access Time Lempel-Ziv) where the access cost is bounded by a parameter given at compression time. We design and implement a linear-space algorithm that, in time $O(n \log^3 n)$, obtains a BAT-LZ parse of a text of length n by greedily maximizing each next phrase length. The algorithm builds on a new linear-space data structure that solves 5-sided orthogonal range queries in rank space, allowing updates to the coordinate where the one-sided queries are supported, in $O(\log^3 n)$ time for both queries and updates. This time can be reduced to $O(\log^2 n)$ if $O(n \log n)$ space is used.

We design a second algorithm that chooses the sources for the phrases in a clever way, using an enhanced suffix tree, albeit no longer guaranteeing longest possible phrases. This algorithm is much slower in theory, but in practice it is comparable to the greedy parser, while achieving significantly superior compression. We then combine the two algorithms, resulting in a parser that always chooses the longest possible phrases, and the best sources for those. Our experimentation shows that, on most repetitive texts, our algorithms reach an access cost close to $\log_2 n$ on texts of length n, while incurring almost no loss in the compression ratio when compared with classical LZ-compression. Several open challenges are discussed at the end of the paper.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Lempel-Ziv parsing, data compression, compressed data structures, repetitive text collections

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.21

Supplementary Material *Software (Source Code)*: `https://github.com/fmasillo/BAT-LZ`

Funding *Zsuzsanna Lipták*: Partially funded by the MUR PRIN project Nr. 2022YRB97K "PINC" (Pangenome INformatiCs. From Theory to Applications) and by the INdAM-GNCS Project CUP_E53C23001670001 (Compressione, indicizzazione, analisi e confronto di dati biologici).
Gonzalo Navarro: Funded by Basal Funds FB0001, Mideplan, Chile, and Fondecyt Grant 1-230755, Chile.

1 Introduction

The sharply growing sizes of text collections, particularly repetitive ones, has raised the interest in compressed data structures that can maintain the texts all the time in compressed form [43, 42, 41]. For archival purposes, the original Lempel-Ziv (LZ) compression format [36] is preferred because it yields the least space among the methods that support compression and decompression in polynomial time – actually, Lempel-Ziv compresses and decompresses a text $T[1..n]$ in $O(n)$ time [48]. For using a compression format as a compressed data structure, however – in particular, to build a compressed text self-index on it [34] –, we need that arbitrary text snippets $T[i\,..\,i+\ell]$ can be extracted efficiently, without the need of decompressing the whole text up to the desired snippet. Grammar compression formats [31] allow extracting such text snippets in time $O(\ell + \log n)$ [5, 22], which is nearly optimal [51]. So, although the compression they achieve is always lower-bounded by the size of the LZ parse [49, 8], grammar compression algorithms are preferred over LZ compression in the design of text indexes [42, 12], and of compressed data structures in general.

The LZ compression algorithm parses the text T into a sequence of so-called phrases, where each phrase points backwards to a previous occurrence of it in T and stores the next symbol in explicit form. While this yields a simple linear-time left-to-right decompression algorithm, consider the problem of accessing a particular symbol $T[i]$. Unless it is the final explicit symbol of a phrase, we must determine the text position $j < i$ where $T[i] = T[j]$ was copied from. We must then determine $T[j]$, which again may be – with low chance – the end of a phrase, or it may – most likely – refer to an earlier symbol $T[j] = T[k]$, with $k < j$. The process continues until we hit an explicit symbol. The cost of extracting $T[i]$ is then proportional to the length of that *referencing chain* $i \to j \to k \to \ldots$ Despite considerable interest in algorithms to access arbitrary text positions from the LZ compression format, and apart from some remarkable results on restricted versions of LZ [30], there has been no progress on the original LZ parse (which yields the strongest compression).

In this paper we introduce and study an LZ variant we call *Bounded Access Time Lempel-Ziv (BAT-LZ)*, which takes a compression parameter c and produces a parse where no symbol has a referencing chain longer than c, thereby guaranteeing $O(c)$ access time.[1] As opposed to classical LZ, BAT-LZ parses allow very fast access to the text, indeed, like a bat out of hell.

We design a *Greedy BAT-LZ parser*, which at each step of the compression chooses the longest possible phrase. Finding such a phrase boils down to solving a 4-sided orthogonal range query in a 3-dimensional grid (in rank space), where one of the coordinates undergoes updates as the parsing proceeds. We design such a data structure, which turns out to handle 5-sided queries and support updates on the coordinate where the query is one-sided. Our data structure handles queries and updates in time $O(\log^3 n)$, yielding a greedy BAT-LZ parsing in time $O(n \log^3 n)$ and space $O(n)$. We then design another BAT-LZ parser, referred to as *Minmax*, which runs on an enhanced suffix tree. It looks for the "best" possible sources of the chosen phrases, that is, with symbols having shorter referencing chains, while not necessarily choosing the longest possible phrase. Finally, we combine the two ideas, resulting in our *Greedier parser*, which runs again on an enhanced suffix tree. These last two algorithms, while their running time is upper bounded by $O(n^3 \log n)$, both run in decent time in practice.

We implemented and tested our three BAT-LZ parsers on various repetitive texts of different sorts, comparing them with the original LZ parse and with two simple baselines that ensure BAT-LZ parses without any optimization. The results show that all three algorithms run in a few seconds per megabyte and produce much better parses than the baselines. For values of $c = O(\log n)$ with a small constant, they produce just a small fraction of extra phrases on top of LZ. In particular, Greedier increases the size of the LZ parse by less than 1% with c values that are about $\log_2 n$ (i.e., 20–30 in our texts).

We note that, unlike the original LZ parse, a greedy parsing does not guarantee obtaining the minimal BAT-LZ parse. Indeed, finding the optimal BAT-LZ parse has recently been shown to be NP-hard for all constant c, and also hard to approximate for any constant approximation ratio [10]. Our results show that, on repetitive texts, a polylog-linear time

[1] A parsing like BAT-LZ was described as a baseline in the experimental results in previous work [33] of one of the authors, but without a parsing algorithm, see Sec. 3 for more details.

greedy algorithm can nonetheless achieve good compression while guaranteeing fast access to text snippets. The other two algorithms are still polynomial time and offer fast access with almost no loss in compression compared to the classical LZ-compression. In our scenarios of interest (i.e., accessing the compressed text at random) the data is compressed only once and accessed many times, so slower compression algorithms can be afforded in exchange for faster access. We discuss at the end this and some other problems our work opens.

2 Basic Data Structures

A string (or text) T is a finite sequence of characters from an alphabet Σ. We write $T = T[1..n]$ for a string T of length n, and assume that the final character is a unique end-of-string marker \$. We index strings from 1 and write $T[i..j]$ for the substring $T[i] \mathinner{.\,.} T[j]$, $T[i \mathinner{.\,.}]$ for the suffix starting in position i, and $T[\mathinner{.\,.} j]$ for the prefix ending in position i.

Bitvectors and Wavelet Matrices. A bitvector $B[1 \mathinner{.\,.} n]$ can be stored using n bits, or actually $\lceil n/w \rceil$ words on a w-bit word machine, while providing access and updates to arbitrary bits in constant time. If the bitvector is static (i.e., does not undergo updates) then it can be preprocessed to answer *rank* queries in $O(1)$ time using $o(n)$ further bits [11, 39]: $rank_b(B, i)$, where $b \in \{0, 1\}$ and $0 \le i \le n$, is the number of times bit b occurs in $B[1 \mathinner{.\,.} i]$.

A wavelet matrix [13] is a data structure that can be used, in particular, to represent a discrete $[1, n] \times [1, n]$ grid, with exactly one point per column, using $n \log_2 n + o(n \log_2 n)$ bits. Let $S[1 \mathinner{.\,.} n]$ be such that $S[i]$ is the row of the point at column i. The first wavelet matrix level contains a bitvector $B_1[1 \mathinner{.\,.} n]$ with the highest (i.e., $\lceil \log_2 n \rceil$th) bit of every value in S. For the second level, the sequence values are stably sorted by their highest bit, and the wavelet matrix stores a bitvector $B_2[1 \mathinner{.\,.} n]$ with the second highest bits in that order. To build the third level, the values are stably sorted by their second highest bit, and so on. Every level i also stores the number $z_i = rank_0(B_i, n)$ of zeros in its bitvector.

The value $S[i]$ can be retrieved from the wavelet matrix in $O(\log n)$ time. Its highest bit is $b_1 = B_1[i_1]$, with $i_1 = i$. The second highest bit is $b_2 = B_2[i_2]$, with $i_2 = rank_0(B_1, i_1)$ if $b_1 = 0$ and $i_2 = z_1 + rank_1(B_1, i_1)$ if $b_1 = 1$. The other bits are obtained analogously.

The wavelet matrix can also obtain the grid points that fall within a rectangle $[x_1, x_2] \times [y_1, y_2]$ (i.e., the values $(i, S[i])$ such that $x_1 \le i \le x_2$ and $y_1 \le S[i] \le y_2$) in time $O(\log n)$, plus $O(\log n)$ per point reported. We start at the first level, in the range $B_1[sp_1, ep_1] = B_1[x_1, x_2]$. We then map the range into two ranges of the second level: the positions i where $B_1[i] = 0$ are all mapped to the range $B_2[sp_2, ep_2] = B_2[rank_0(B_1, sp_1 - 1) + 1, rank_0(B_1, ep_1)]$, and those where $B_1[i] = 1$ are mapped to $B_2[sp_2', ep_2'] = B_2[z_1 + rank_1(B_1, sp_1 - 1) + 1, z_1 + rank_1(B_1, ep_1)]$. The recursive process stops when the range becomes empty; when the sequence of highest bits makes the possible set of values either disjoint with $[y_1, y_2]$ or included in $[y_1, y_2]$; or when we reach the last level. It can be shown that the recursion ends in $O(\log n)$ ranges, at most two per level, so that every value in those ranges is an answer. The corresponding y values can be obtained by tracking them downwards as explained.

These data structures, and our results, hold in the RAM model with computer word size $w = \Theta(\log n)$. The wavelet matrix is then said to use $O(n)$ space – i.e., linear space –, which is counted in w-bit words. The wavelet matrix is easily built in $O(n \log n)$ time, and less [40].

Another relevant functionality that can be offered within $2n + o(n)$ bits is the so-called *range maximum query (RMQ)*: given a static array $A[1 \mathinner{.\,.} n]$, we preprocess it in $O(n)$ time so that we can answer RMQs in $O(1)$ time [19]: $rmq(A, i, j)$ is a position p, $i \le p \le j$, such that $A[p] = \max\{A[k],\ i \le k \le j\}$. The data structure does not need to maintain A. In this paper we will use RMQs where A can undergo updates, see Sec. 5.

Suffix Arrays and Trees. The suffix tree [52] is a classic data structure on texts which is able to answer efficiently many different kinds of string processing queries [24, 1], which uses linear space and can be built in linear time [52, 38, 17, 50]. We give a brief recap; see Gusfield [24] for more details.

The suffix tree $\mathsf{ST}(T)$ of a text T is the compact trie of the suffixes of T; it is a rooted tree whose edges are labeled by substrings of T (stored as two pointers into T), and whose inner nodes are branching. The *label* $L(v)$ of a node v is the concatenation of the labels of the edges on the root-to-v path. There is a one-to-one correspondence between leaves and suffixes of T; leaf_i is then the unique leaf whose label equals the ith suffix $T[i\,..]$. The *stringdepth* $\mathit{sd}(v)$ of a node v is the length of its label, and we assume $\mathit{sd}(v)$ is stored in v.

The suffix array SA of T is a permutation of the index set $\{1,\ldots,n\}$ such that $\mathsf{SA}[i]=j$ if the jth suffix of T is the ith in lexicographic order among all suffixes. The suffix array can be computed from the suffix tree, or directly from the text, in linear time and space [47, 45]. The inverse suffix array, denoted ISA, can be computed in linear time using $\mathsf{ISA}[\mathsf{SA}[i]]=i$.

3 The Lempel-Ziv (LZ) Parsing and its Bounded Version (BAT-LZ)

The Lempel-Ziv (LZ) parsing of a text $T[1\,..\,n]$ [36] produces a sequence of z "phrases", which are substrings of T whose concatenation is T. Each phrase is formed by the longest substring that has an occurrence starting earlier in T, plus the character that follows it.

▶ **Definition 1.** *A* leftward parse *of $T[1\,..\,n]$ is a sequence of substrings $T[i\,..\,i+\ell]$ (called* phrases*) whose concatenation is T and such that there is an occurrence of each $T[i\,..\,i+\ell-1]$ starting before i in T (the occurrence is called the* source *of the phrase). The LZ parse of T is the leftward parse of T that, in a left-to-right process, chooses the longest possible phrases.*

The algorithm moves a pointer i along T, from $i=1$ to $i=n$. At each step, the algorithm has already processed $T[1\,..\,i-1]$, and it must form the next phrase. As said, the phrase is formed by (1) the longest prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ that has an occurrence in T starting before position i, and (2) the next symbol $T[i+\ell]$. If $\ell>0$, then the occurrence of (1), $T[s\,..\,s+\ell-1]=T[i\,..\,i+\ell-1]$ with $s<i$, is called the source of $T[i\,..\,i+\ell-1]$. Once suitable s and ℓ have been determined, the next phrase is $T[i\,..\,i+\ell]$ and the algorithm proceeds from $i\leftarrow i+\ell+1$ onwards. The phrase $T[i\,..\,i+\ell]$ is encoded as the triple $(s,\ell,T[i+\ell])$, and if $\ell=0$ we can encode just the character $(T[i+\ell])$.

This greedy parsing, which maximizes the phrase length at each step, turns out to be optimal [36], that is, it produces the least number z of phrases among all the leftward parses of T. Further, it can be computed in $O(n)$ time [48, 9, 46, 25, 26, 23, 20, 32, 3, 27, 21].

Note that phrases can overlap their sources, as sources must start – but not necessarily end – before i. For example, the LZ parse of $T=\mathtt{a}^{n-1}\$$ is $(\mathtt{a})$ $(0,n-1,\$)$. For illustrative purposes, we describe the parsings by writing bars, "|", between the formed phrases. The parsing of the example is then written as $\mathtt{a}|\mathtt{a}^{n-1}\$$. To illustrate the access problem, consider the LZ parsing of the text `alabaralalabarda$` (disregard for now the numbers below):

$$\begin{array}{c|c|cc|cc|cccc|ccccc|cc|}
\mathtt{a} & \mathtt{l} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{r} & \mathtt{a} & \mathtt{l} & \mathtt{a} & \mathtt{l} & \mathtt{a} & \mathtt{b} & \mathtt{a} & \mathtt{r} & \mathtt{d} & \mathtt{a} & \$ \\
0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 2 & 0 & 2 & 1 & 2 & 1 & 0 & 1 & 0
\end{array}$$

Assume we want to extract $T[11]=\mathtt{a}$. The position is the first of the 6th phrase, `abard`, and it is copied from the third phrase, `ab`. In turn, the first position of that phrase is copied from the first phrase, where `a` is stored in explicit form. We need then to follow a *chain* of length two in order to extract $T[11]$, so the length of that chain is the access cost. The numbers we wrote below the symbols in the parse are the lengths of their chains.

Bounded Access Time Lempel-Ziv (BAT-LZ). We define a leftward parse we call Bounded Access Time Lempel-Ziv (BAT-LZ), which takes as a parameter the maximum length c any chain can have. A BAT-LZ parse is a leftward parse where no chain is longer than c. Note that we do not require a BAT-LZ parse to be of minimal size. For example, a BAT-LZ parse for the above text with $c = 1$ is as follows:

a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	$
0	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0

When the LZ parse produces the phrase $T[i\,.\,.\,i+\ell]$ from the source $T[s\,.\,.\,s+\ell-1]$ and the extra symbol $T[i+\ell]$, the character $T[i+\ell]$ is stored in explicit form, and thus its chain is of length zero. The chain length of every other phrase symbol, $T[i+l]$ for $0 \le l < \ell$, is one more than the chain length of its source symbol, $T[s+l]$.

A special case occurs when sources and targets overlap. If we want to extract $T[n-1]$ from $T = \mathtt{a}^{n-1}\$$, we could note that it is copied from $T[n-2]$, which is in turn copied from $T[n-3]$, and so on, implying a chain of length $n-1$. Instead, we can note that our phrase $T[2\,.\,.\,n]$ overlaps its source $T[1\,.\,.\,n-2]$. In general, when the phrase $T[i\,.\,.\,i+\ell-1]$ overlaps its source $T[s\,.\,.\,s+\ell-1]$ by $0 < b = i-s$ characters, this implies that the word $S = T[s\,.\,.\,s+\ell-1] = T[i\,.\,.\,i+\ell-1]$ has a *border* (a prefix which is also a suffix) of length b. It is well known that if S has a border of length b, then S has a period $p = |S| - b$, see [37, Ch. 8]. Therefore, S can be written in the form $S = U^{\lfloor |S|/p \rfloor}V$, where U is the p-length prefix of S and V a proper prefix of U, and thus, for all $l > p$, $S[l] = S[l \bmod p]$.

▶ **Definition 2** (Chain length). *Let $T[i\,.\,.\,i+\ell]$ be a phrase in a leftward parse of $T[1\,.\,.\,n]$, whose source is $T[s\,.\,.\,s+\ell-1]$. The chain length of the explicit character is $C[i+\ell] = 0$. If $\ell \le i-s$ (i.e., there is no overlap between the source and the phrase), then for all $0 \le l < \ell$, $C[i+l] = C[s+l]+1$. Otherwise, for $0 \le l < i-s$, the chain length is $C[i+l] = C[s+l]+1$, and for $i-s \le l < \ell$, the chain length is $C[i+l] = C[i+(l \bmod (i-s))]$.*

We remark that a parsing like BAT-LZ is described as a baseline in the experimental results of one of the current authors' previous work [33], under the name LZ-Cost, but as no efficient parsing algorithm was devised for it, it could be tested only on the tiny texts of the Canterbury Corpus (`https://corpus.canterbury.ac.nz`). It also did not handle overlaps between sources and targets, so it did not perform well on the text $T = \mathtt{a}^n$. For testing the BAT-LZ parsing on large repetitive text collections we need an efficient parsing algorithm.

4 A Greedy Parsing Algorithm for BAT-LZ

In this section we describe an algorithm that, using $O(n)$ space and $O(n \log^3 n)$ time, produces a BAT-LZ parse of a text $T[1\,.\,.\,n]$ by maximizing the next phrase length at each step. We then show how to reduce the time to $O(n \log^2 n)$ at the price of increasing the space to $O(n \log n)$. Of course, unlike in LZ, this greedy algorithm does not in general produce an optimal BAT-LZ parse, since the problem is NP-hard.

▶ **Definition 3.** *A BAT-LZ parse of $T[1\,.\,.\,n]$ with maximum chain length c is a leftward parse of T where the chain length of no position exceeds c. A* greedy BAT-LZ parse *is a BAT-LZ parse where each phrase, processed left to right, is as long as possible.*

Let $T[1\,.\,.\,i-1]$ be already processed. We call a prefix $T[i\,.\,.\,i+\ell-1]$ of $T[i\,.\,.]$ *valid* if $C[j] \le c$ for all $j = i, \ldots, i+\ell-1$. A leftward parse of T is therefore a BAT-LZ parse if and only if all phrases are valid. Our Greedy BAT-LZ parser proceeds then analogously to the original LZ parser. At each step, it has already processed $T[1\,..\,i-1]$, and it must find the next phrase, which is formed by (1) the longest valid prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ that has an occurrence $T[s\,..\,s+\ell-1]$ with $s<i$, and (2) the next symbol $T[i+\ell]$. In other words, the algorithm enforces that every symbol in $T[s\,..\,s+\ell-1]$ must have a chain length less than c, the maximum chain length allowed. The phrase $T[i\,..\,i+\ell]$ is encoded just as in the standard LZ, as a triple $(s,\ell,T[i+\ell])$.

To efficiently find s and ℓ, our BAT-LZ parsing algorithm stores the following structures:

1. The suffix array $\mathsf{SA}[1\,..\,n]$ of T, represented as a wavelet matrix [13].
2. The inverse suffix array $\mathsf{ISA}[1\,..\,n]$ of T, represented in plain form.
3. An array $C[1\,..\,n]$, where $C[i]$ is the chain length of i. Note that $C[i]$ is defined only for the already parsed positions of T.
4. An array $D[1\,..\,n]$, where $D[s]$ is the minimum $d\geq 0$ such that $C[s+d]=c$. If no such a d exists (in particular, because $C[i]$ is defined only for the parsed prefix), then $D[s]=\infty$ (which holds initially for all s).
5. For each level of the wavelet matrix of SA, a special *dynamic RMQ* data structure to track the text positions that can be used. This structure is related to the values of D and therefore it changes along the parsing.

Note that the definition of BAT-LZ implies that, if the source of $T[i\,..\,i+\ell-1]$ is $T[s\,..\,s+\ell-1]$, then it must be that $\ell\leq D[s]$. This motivates the following observation:

▶ **Observation 4.** *Let $T[1\,..\,i-1]$ be already processed. A prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$ is valid if and only if there exists a source $T[s\,..\,s+\ell-1]$ such that*

(i) *its lexicographic position satisfies $\mathsf{ISA}[s]\in[sp\,..\,ep]$, where $[sp\,..\,ep]$ is the suffix array range of $T[i\,..\,i+\ell-1]$ (i.e., $T[s\,..\,s+\ell-1]=T[i\,..\,i+\ell-1]$);*

(ii) *its starting position in T is $s<i$; and*

(iii) *it does not use forbidden text positions, that is, $\ell\leq D[s]$.*

The parsing then must find the longest valid prefix $T[i\,..\,i+\ell-1]$ of $T[i\,..]$. We do so by testing the consecutive values $\ell=1,2,\ldots$. Note that, once we have determined the next phrase $T[i\,..\,i+\ell]$, we must update C and D as follows: (1) $C[i+l]\leftarrow C[s+l]+1$ for all $0\leq l<\ell$, and $C[i+\ell]\leftarrow 0$.[2], and (2) Every time we obtain $C[t]=c$ in the previous point, we set $D[k]\leftarrow t-k$ for all $k'<k\leq t$, where k' is the last position where $D[k']<\infty$ (so $k'=0$ in the beginning and we reset $k'\leftarrow t$ after this process).

Note that points (i) and (ii) above correspond to the classic LZ parsing problem. In particular, they correspond to determining whether there are points in the range $[sp,ep]\times[1,i-1]$ of the grid represented by our wavelet matrix, which represents the points $(j,\mathsf{SA}[j])$. As the wavelet matrix answers this query in time $O(\log n)$, this yields an $O(n\log n)$ LZ parsing algorithm. Point (iii), however, is exclusive to BAT-LZ. It can be handled by converting the grid into a three-dimensional mesh, where we store the values $(j,\mathsf{SA}[j],D[\mathsf{SA}[j]])$ and look for the existence of points in the range $[sp,ep]\times[1,i-1]\times[\ell,n]$. Note that we need to determine whether the range is empty and, if it is not, retrieve a point from it (whose second coordinate is the desired s). In addition, as the array D is modified along the parsing, we need a dynamic 3-dimensional data structure: every time we modify D in point 2 above, our data structure changes (this occurs up to n times). See Fig. 1.

[2] Recall that a special case occurs if $T[i\,..\,i+\ell-1]$ overlaps $T[s\,..\,s+\ell-1]$: we start copying from $k=s$ and increasing k and, whenever $k=s+l=i$, we restart copying from $k=s$.

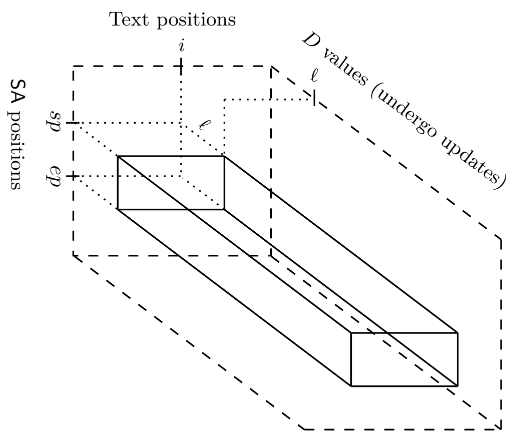

Figure 1 General scheme of our translation of queries onto a 3-dimensional data structure.

Our 3-dimensional problem, then, (a) is essentially a range emptiness query (where we must return one point if there are any), (b) the search is 4-sided (though our solution handles 5-sided queries), and (c) the updates in D occur only to convert some $D[k] = \infty$ into a smaller value, so each value $D[k]$ changes at most once along the parsing process (yet, our solution handles arbitrary updates along the coordinate where the query is one-sided). We have found no linear-space solutions to this problem in the literature; only solutions to less general ones or using super-linear space (indeed, more than $O(n \log n)$): (1) linear space for *two dimensions*, with $O(\log n)$ query time and $O(\log^{3+\epsilon} n)$ update time [44]; (2) linear space for three dimensions with *no updates*, with $O(\log n/\log\log n)$ query time [6]; (3) *super-linear* space (at least $O(n \log^{1.33} n)$ for three dimensions), with $O((\log n/\log\log n)^2)$ query time and $O(\log^{1.33+\epsilon} n)$ update time [7]. In the next section we describe our data structures for this problem: one uses linear space and $O(\log^3 n)$ query and update time; the other uses $O(n \log n)$ space and $O(\log^2 n)$ query time. This yields our first main result.

▶ **Theorem 5.** *A Greedy BAT-LZ parse of a text $T[1..n]$ can be computed using $O(n)$ space and $O(n \log^3 n)$ time, or $O(n \log n)$ space and $O(n \log^2 n)$ time.*

5 A Geometric Data Structure

To solve the 3-dimensional search problem we associate, with each level of the wavelet matrix, a data structure that represents the sequence of values $D[k]$ in the order the text positions k are listed in that level. Because in linear space we cannot store the actual values in every wavelet matrix level, we store only a dynamic RMQ data structure on the internal levels, and store the explicit values only in (the order corresponding to) the last level (in a wavelet matrix, that final level is not the text order, thus we need another array to map it to D).

Let D_l be the array D permuted in the way it corresponds to level l of the wavelet matrix. The dynamic RMQ structure for level l is then a heap-shaped perfectly balanced tree $H_l[1..n]$ whose leaves (implicitly) point to the entries of D_l. The nodes $H_l[p]$ store only one bit, 0 indicating that the maximum in the subtree is to the left and 1 indicating that it is to the right. By navigating H_l from the root p of any subtree, moving to $H_l[2p]$ if $H_l[p] = 0$ and $H_l[2p+1]$ if $H_l[p] = 1$, we arrive in $O(\log n)$ time at the position p where $D_l[p]$ is maximum below that subtree. The actual value $D_l[p]$ is obtained in other $O(\log n)$ time by tracking position p downwards in the wavelet matrix, from level l until the last level, where the values of D are explicitly stored. See Fig. 2 (right); ignore the query for now.

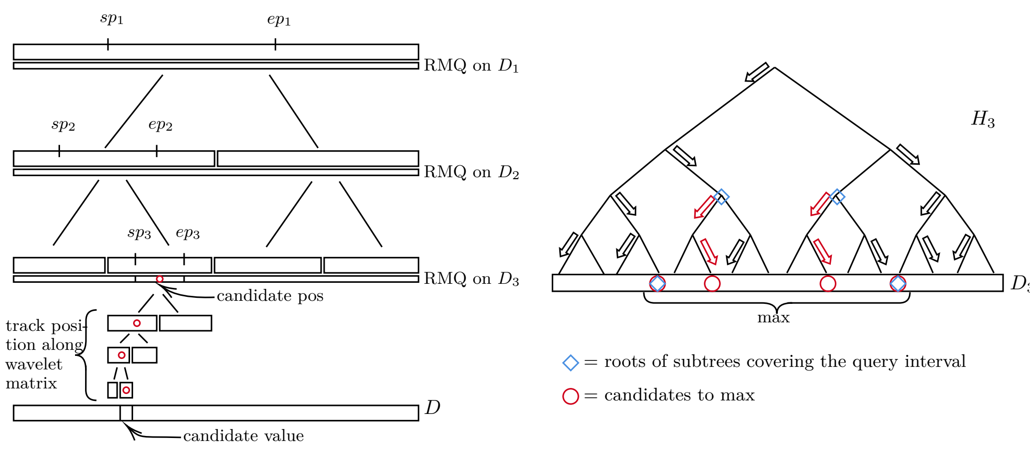

Figure 2 On the left, we reach a candidate area $[sp_3, ep_3]$ of the wavelet matrix and must obtain its maximum D value using the (dynamic) RMQ data structure for D_3. The tree H_3 for this RMQ structure is shown on the right. Arrows point to the child holding the maximum value in D_3. Blue diamonds are the roots $v_3^1, \ldots, v_3^4$ of the subtrees that cover the query area $[sp_3, ep_3]$ and red circles are the candidates in the range. The left plot shows how we find the actual value of one of those circles by tracking it down in the wavelet matrix.

Updates. When a value $D[k]$ decreases from ∞, we obtain its position in the top-level of the wavelet matrix as $p = \mathsf{ISA}[k]$; thus we must reflect in H_1 the decrease in the value of $D_1[p]$. By halving p successively we arrive at its ancestors, $H_1[p_h]$ for $p_h = \lfloor p/2^h \rfloor$, $h = 1, 2, \ldots$ We traverse the path upwards, recomputing the maximum value m below p_h and modifying accordingly the bits of $H_1[p_h]$. Initially, this new maximum is $m = D_1[p] = D[k]$. At any point in the traversal, if the parent $H_1[p_h]$ of the current node indicates that the maximum below p_h descends from the *other* child of p_h, then we can stop updating of H_1, because decreasing $D_1[p]$ does not require further changes. Otherwise, we must obtain the maximum value m' below the other child of $H_1[p_h]$ and compare it with m. The value m' is obtained in $O(\log n)$ time as explained in the previous paragraph. We set $H_1[p_h]$ depending on which is larger between m and m', update $m \leftarrow \max(m, m')$, and continue upwards. This process takes $O(\log^2 n)$ time as we traverse all the levels of H_1. We then track position p downwards to the second level of the wavelet matrix, update H_2 in the same way, and continue updating H_l on all the wavelet matrix levels l, for a total update time of $O(\log^3 n)$.

Searches. The search for a range $[sp, ep] \times [1, i-1] \times [\ell, n]$ first determines, as in the normal wavelet matrix search algorithm, the $O(\log n)$ maximal ranges that cover $[1, i-1]$ along the wavelet matrix levels l (there is at most one range per level because the range $[1, i-1]$ is one-sided; otherwise there could be two), and maps $[sp, ep]$ to $[sp_l, ep_l]$ on each such range (see Sec. 2), all in time $O(\log n)$. We then need to determine if there is some value $D_l[p] \geq \ell$ below some of the ranges $[sp_l, ep_l]$ (see the top-left part of Fig. 2). Each such range is then, again, decomposed into $O(\log n)$ maximal nodes $v_l^1, v_l^2, \ldots$ of H_l (see the right of Fig. 2). We find, in $O(\log n)$ time, the maximum value of D_l below each node v_l^j, stopping as soon as we find some value $\geq \ell$. Note that we use $O(\log n)$ time to find the *position* of the maximum in D_l using H_l, and then $O(\log n)$ time to find the *value* of that maximum by tracking the position down in the wavelet matrix (see the bottom left of Fig. 2). Since we have $O(\log n)$ ranges $[sp_l, ep_l]$, each yielding $O(\log n)$ candidates v_l^i, and the maximum of each candidate is computed in $O(\log n)$ time, the whole search process takes time $O(\log^3 n)$.

Generalizations. Though not necessary for our problem, we remark that our update process can be extended to arbitrary updates on the third coordinate, $D[k]$, not only to reductions in value. Further, our search could support five-sided ranges, not only four-sided, because we would still have $O(\log n)$ ranges $[sp_l, ep_l]$ if the range of the second coordinate was two-sided. Only the range of the third coordinate (the one supporting the updates) must be one-sided.

Faster and larger. By storing the values of D_l in each node of H_l for each wavelet matrix level l, the space increases to $O(n \log n)$ but the time of updates and searches decreases to $O(\log^2 n)$, as we have now the maximum below any $H_l[p_h]$ readily available in $O(1)$ time.

6 The Minmax Parsing Algorithm

We note that our Greedy BAT-LZ algorithm does not necessarily produce the smallest greedy parse, because it may fail in choosing the best *source* for the longest phrase. Consider, say, the text $T =$ `alabaralalabarda$` and $c = 2$. Our implementation parses it into 8 phrases as

$$\begin{array}{c|c|cc|cc|cccc|ccc|cc|cc|}
\texttt{a} & \texttt{l} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{r} & \texttt{a} & \texttt{l} & \texttt{a} & \texttt{l} & \texttt{a} & \texttt{b} & \texttt{a} & \texttt{r} & \texttt{d} & \texttt{a} & \texttt{\$} \\
0 & 0 & 1 & 0 & 2 & 0 & 1 & 1 & 2 & 0 & 2 & 1 & 0 & 1 & 0 & 1 & 0
\end{array}$$

because it chooses $T[3]$ as the source for the 4th phrase, `ar`, and then $T[5]$ has a chain of length two and cannot be used again. If, instead, we choose $T[1]$ as the source of the 4th phrase, the chain of $T[5]$ will be of length 1 and we could parse T into 7 phrases, just as the first parse shown in Sec. 3.

Our second algorithm, the Minmax parser, always chooses a source that minimizes the maximum chain length in the phrase, among all possible sources. It compromises however on the *length* of the phrase, by not always choosing the longest admissible phrase. As we will see, this is well worth it: Minmax always produces a much better compression than Greedy.

High-level description of the Minmax parser. Let $T[1 \mathinner{.\,.} i-1]$ be already processed. We will call a prefix $T[i \mathinner{.\,.} i+\ell-1]$ of $T[i \mathinner{.\,.}]$ *admissible* if it has a source $T[s \mathinner{.\,.} s+\ell-1]$ with $\max C[s \mathinner{.\,.} s+\ell-1] < c$. We would ideally like to find the longest admissible prefix of $T[i \mathinner{.\,.}]$, and then choose its best source if there is more than one. We will use an enhanced suffix tree of the text; this will allow us to store additional information in the nodes. Navigating in the suffix tree, we will then be able to choose the longest admissible prefix *which ends in some node* (i.e., not necessarily the longest), and then choose the best source of this prefix. In order to do this, we will match the current suffix $T[i \mathinner{.\,.}]$ in the usual way in the suffix tree, using the desired information written in the nodes. As this information is dynamic, however, we will have to update it during the algorithm. The algorithm thus proceeds by (1) matching the suffix $T[i \mathinner{.\,.}]$ in the suffix tree and returning the next phrase and its source, and (2) updating the annotation.

Annotation of the suffix tree. On the suffix tree of T, we annotate each node v with three variables $\mathsf{minmax}(v)$, $\mathsf{txtpos}(v)$, and a Boolean $\mathsf{real}(v)$, initializing $\mathsf{minmax}(v)$ to $+\infty$, $\mathsf{txtpos}(v)$ to -1, and $\mathsf{real}(v)$ to 0. Recall that $L(v)$ is the label of v and $sd(v)$ its length. The variables $\mathsf{minmax}(v)$ and $\mathsf{txtpos}(v)$ will point to the current best candidate of an occurrence of $L(v)$, with $\mathsf{txtpos}(v)$ its starting position and $\mathsf{minmax}(v)$ the maximum C-value within this occurrence. The Boolean $\mathsf{real}(v)$ indicates whether this value is *realistic* ($\mathsf{real}(v) = 1$), i.e., a full occurrence with this value has already been seen, or only *optimistic* ($\mathsf{real}(v) = 0$), meaning that no full occurrence has yet been seen. More formally, let i be the current position, and let us first assume that $\mathsf{real}(v) = 1$. Then $\mathsf{minmax}(v) = x$ if $x = \min\{\max C[s\,..\,s + sd(v) - 1] : T[s\,..\,s + sd(v) - 1] = L(v) \text{ and } s + sd(v) - 1 < i\}$, and $\mathsf{txtpos}(v) = s_0$ for one such s_0, i.e., (i) $T[s_0\,..\,s_0 + sd(v) - 1] = L(v)$, (ii) $s_0 + sd(v) - 1 < i$, and (iii) $\max C[s_0\,..\,s_0 + sd(v) - 1] = x$.

Now let us look at the case $\mathsf{real}(v) = 0$, we have yet to see an occurrence of $L(v)$. Initially, $\mathsf{minmax}(v) = +\infty$; when we encounter a non-empty prefix of $L(v)$, of length $0 < d \leq sd(v)$, starting, say, in position s_0, we update $\mathsf{minmax}(v)$ to $\max C[s_0\,..\,s_0 + d - 1]$ and $\mathsf{txtpos}(v)$ to s_0. Thus, we have seen an occurrence of a prefix of $L(v)$ but not yet a full occurrence of $L(v)$, and we are optimistic since we are hoping to find a full occurrence whose max does not exceed the current one. However, as soon as we find the first full occurrence (and set $\mathsf{real}(v) = 1$), from that point on we only update $\mathsf{minmax}(v)$ and $\mathsf{txtpos}(v)$ if we see another full occurrence. Therefore, $\mathsf{real}(v)$ is updated exactly once during the algorithm.

Finding an admissible phrase and choosing its source. Let us now assume that we have processed $T[1\,..\,i-1]$ and want to find the next phrase and source. We match $T[i\,..]$ in the suffix tree, making sure during navigation that we only get admissible prefixes of $T[i\,..]$. In particular, if we are in node v and should go to child u of v next (because $T[i + sd(v)]$ is the first character of the edge label (v, u)), then we first check if $\mathsf{minmax}(u) < c$. If so, then we can descend to u and continue from there, skipping over the next $sd(u) - sd(v)$ positions in T. Otherwise, $\mathsf{minmax}(u) \geq c$ and we return the new phrase $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$. Moreover, the C-array for $j = i, \ldots, i + \ell$ is set according to Def. 2.

Updating the suffix tree annotation. After the new phrase has been computed, we need to update the annotations in the suffix tree. For $j \leq i + \ell$, going backward in the string, we will update the nodes on the leaf-to-root path from leaf j. The idea is the following.

Fix $j \leq i + \ell$. The prefix $T[j\,..\,i + \ell]$ of $T[j\,..]$ now has the C-array filled in, so its max-value $m = \max C[j\,..\,i + \ell]$ is known. This may or may not necessitate updates in the nodes on the path from leaf $leaf_j$ to the root. First, for the leaf j itself, if $i \leq j$, then the minmax is still $+\infty$, so we set $\mathsf{minmax}(leaf_j) \leftarrow m$. Otherwise, we are seeing a longer prefix of $T[j\,..]$ than before, so we update $\mathsf{minmax}(leaf_j) \leftarrow \max(\mathsf{minmax}(leaf_j), m)$. Regarding the nodes v on the path from $leaf_j$ to the root: their labels are increasingly shorter prefixes of suffix $T[j\,..]$, so they need to be updated only as long as $j + sd(v) - 1 \geq i$, since otherwise, the prefix $L(v)$ does not overlap with the newly assigned subinterval $C[i\,..\,i + \ell]$.

So let $j + sd(v) \geq i$, there are two cases. First, if $j + sd(v) - 1 \leq i + \ell$, then m is a realistic value, since the entire corresponding C-array interval has been filled in. Therefore, we can then compute m in a more clever way by using an RMQ on C, i.e., $m = RMQ(C, j, j + sd(v) - 1)$. So if $\mathsf{real}(v) = 0$, then we update $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{real}(v) \leftarrow 1$. Otherwise (if $\mathsf{real}(v) = 1$), an update is needed only if $\mathsf{minmax}(v) > m$, in which case we set $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{txtpos}(v) \leftarrow j$; since $\mathsf{real}(v) = 1$, we have seen the label $L(v)$ before and already had a realistic value for its minmax value. Second, if $j + sd(v) - 1 > i + \ell$, then m is an optimistic value only, and therefore, we update the annotation of v only if $\mathsf{real}(v) = 0$; in that case, we set $\mathsf{minmax}(v) \leftarrow m$ and $\mathsf{txtpos}(v) \leftarrow j$.

Finally, we use the following criterion for how far back in the string we need to go with j. If no node in the path from $leaf_j$ to the root can be effected by the new phrase, then we do not need to consider position j at all in the current iteration. This holds if the label of the parent node does not reach i, i.e., if $j + sd(parent(leaf_j)) - 1 < i$. We compute an auxiliary array $E[1\,..\,n]$ s.t. $E[j] = j + sd(parent(leaf_j)) - 1$. It is easy to see that $E[j] \leq E[j']$ if $j < j'$. This means that, moving back-to-front, we can stop at the first j for which $E[j] < i$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T	a	l	a	b	a	r	a	l	a	l	a	b	a	r	d	a	\$
C	0	0	1	0	1	0											

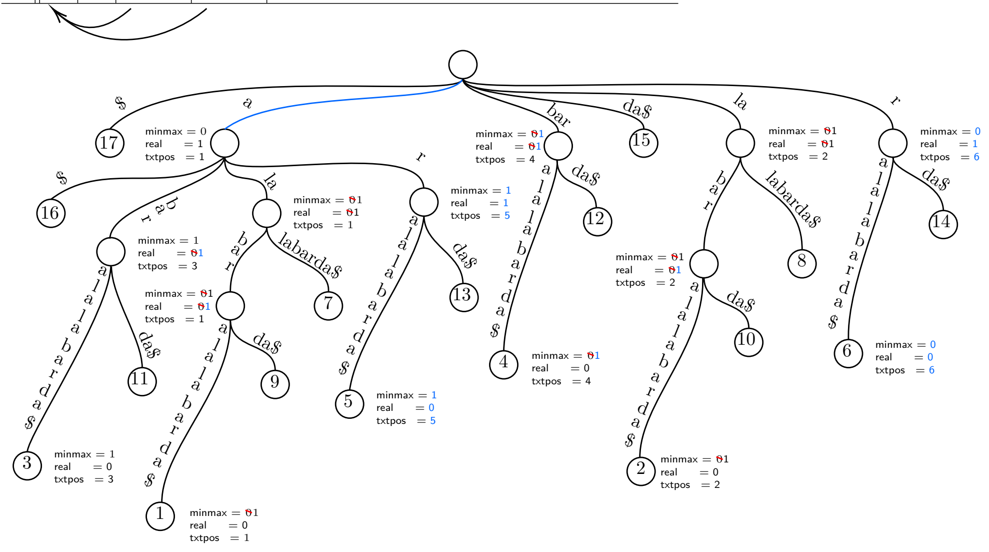

Figure 3 Example of the Minmax algorithm using the suffix tree of T = alabaralalabarda\$. The vertical bars are for delimiting already parsed phrases. See Example 6 for more details.

A worst-case time complexity for a Minmax parse producing z' phrases is $O(z'n^2) \subseteq O(n^3)$, as in principle one can consider every $j \in [1..i-1]$ for every new phrase $T[i..i+\ell]$, traverse the $O(n)$ ancestors of leaf_j, and run an RMQ operation on each. While the RMQ structure we use on C is dynamic, it only undergoes appends to the right, in which case it is possible to support updates in $O(1)$ amortized time and queries in $O(1)$ time [18, p. 5]. We do not know if this cubic complexity is tight, however. In practice we expect z' to be much less than n on highly repetitive texts, and the height of the suffix tree to be logarithmic, yielding a time complexity of $O(z'n \log n)$, which thus becomes practical on repetitive data.

▶ **Example 6.** In Fig. 3, we can see the suffix tree ST for T = alabaralalabarda\$ with some additional annotations in some nodes. In this example, the first three phrases, i.e., a, l, and ab, have been already computed, with the corresponding chain lengths in C and annotations in ST. The annotations exhibit non-trivial updates using the colour red, namely updates that are different than changing the starting value for minmax, i.e., changing minmax from $+\infty$ to a finite value. Nodes whose annotation is not shown have not been updated yet, therefore, they have minmax $= +\infty$, txtpos $= -1$, and real $= 0$. The updates caused by the new phrase ar are highlighted in blue. First, to find the longest previous factor we have to descend to the child with label a, then we check whether the child with label ar has minmax $< c$. In this case, it was not less than c (minmax $= +\infty$), so we stopped the search and output the new phrase ar. Then all suffixes j with $1 \leq j \leq 6$ undergo an update starting from the corresponding leaf; e.g., leaves 5 and 6 and corresponding ancestors get updated to some non-initial value, whereas inner nodes with label abar, alabar, bar and labar change real to 1 because $j + sd(v) - 1 \leq i + \ell$.

7 The Greedier Parser: Combining Greedy with Minmax

We now combine the ideas of the Greedy and the Minmax parsers, using the enhanced suffix tree to consider only longest admissible phrases. Consider when the Minmax algorithm stops in a node v and returns $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$. It did not descend to the next child u because $\mathsf{minmax}(u) = c$, i.e., every occurrence of $L(v)$ seen so far has a value c somewhere in the C-array. However, it is possible that in one of these occurrences, the position of this c is after $L(v)$; in other words, that we could have gone down the edge some way towards u.

To check this, we will use the D-array from Sec. 4, in addition to the C-array and the enhanced suffix tree. Let v and u be as before, i.e., v is parent of u, $\mathsf{minmax}(v) < c$, $\mathsf{minmax}(u) \geq c$, $L(v)$ is a prefix of $T[j\,..]$ and $T[j + sd(v)]$ is the first character of the label of (v, u). Let d be the maximum value of $D[k]$ for some occurrence of $L(u)$ that we have already processed, so d is the largest distance from the start of an occurrence of $L(u)$ to the next c in the C-array. We return $(\mathsf{txtpos}(v), sd(v), T[i + sd(v)])$, as before, if $d \leq sd(v)$, and $(k, D[k], T[i+D[k]])$, where k is a leaf in u's subtree with $D[k] = d$, otherwise. As for updating the annotations, if node v has $\mathsf{minmax}(v) = c$ and some $\mathsf{txtpos}(v) = x$, then, when performing the traversal from $leaf_j$ up to the root, we want to change $\mathsf{txtpos}(v) \leftarrow j$ if $D[j] > D[\mathsf{txtpos}(v)]$. It is easy to see that the Greedier algorithm now returns the longest admissible phrases; otherwise, it works similarly to the Minmax algorithm. The time complexity increases to $O(z'n^2 \log n) \subseteq O(n^3 \log n)$, because we need dynamic RMQs on array D as well, which undergoes updates at arbitrary positions.

8 Experiments

We implemented the BAT-LZ parsing algorithms in C, and ran our experiments on an AMD EPYC 7343, with 32 cores at 1.5 GHz, with a 32 MB cache and 1 TB of RAM. We used the repetitive files from Pizza&Chili (`http://pizzachili.dcc.uchile.cl`) and compared the number of phrases produced by BAT-LZ using different maximum values c for the chains, with the number of phrases produced by LZ (i.e., with no limit c). We used a classic LZ implementation [26] where the source of each phrase is its lexicographically closest suffix.[3]

As a reference point, we also implemented two simple baselines that obtain a BAT-LZ parse. The first, called BAT-LZ1, runs the classic LZ parse and then cuts the phrases at the points where the chain lengths reach $c + 1$. Since the symbol becomes explicit, its chain length becomes zero and the chain lengths of the symbols referencing it decrease by $c + 1$. We should then find the new positions that reach $c + 1$, and so on. It is not hard to see that this laborious postprocessing can be simulated by just adding, to the original z value of LZ, the number of positions i where $C[i] \bmod (c + 1) = 0$.

The second baseline, BAT-LZ2, is slightly stronger: when it detects that it has produced a text position exceeding the maximum c, it cuts the phrase there (making the symbol explicit), and restarts the LZ parse from the next position. This gives the chance of choosing a better phrase starting after the cut, unlike BAT-LZ1, which maintains the original source.

Despite some optimizations, our Greedy BAT-LZ parser consistently reaches the $\Theta(\log^3 n)$ time complexity per text symbol, making it run at about 3 MB per minute. The Greedier and the Minmax parsers, despite their cubic worst-case time complexity, run at a similar pace: 1.9–4.7 MB per minute: our upper bound is utterly pessimistic, and perhaps not tight.

[3] It is likely that using the variant called "rightmost LZ parse", which chooses the rightmost source, gives better results because it tends to distribute the uses of the sources more uniformly. Such a parse seems to be nontrivial to compute [4, 15], however, and we are not aware of practical implementations.

Table 1 Our repetitive text collections and some statistics: alphabet size σ, length n, number z of phrases in the LZ parse, average phrase length n/z, maximum chain length in our LZ parse, size g of a balanced grammar, and height h of that grammar.

File	σ	n	z	n/z	max c	g	h
`coreutils`	236	205,281,779	1,286,070	160	66	2,409,429	28
`kernel`	160	257,961,616	705,791	365	70	1,374,651	32
`einstein`	139	467,626,545	75,779	6,171	1,736	212,902	47
`leaders`	89	46,968,181	155,937	301	60	399,667	27
`para`	5	429,265,758	1,879,635	228	38	5,344,477	26
`influenza`	15	154,808,555	557,349	278	63	1,957,370	26

Table 1 shows the main characteristics of the collections chosen. We included two versioned software repositories (`coreutils` and `kernel`, where the versioning has a tree structure), two versioned documents (`einstein` and `leaders`, where the versioning has a linear structure), and two biological sequence collections (`para` and `influenza`, where all the sequences are pairwise similar). The average phrase length is in the range 160–365 and the maximum chain length of a symbol is in the range 38–70. The exception is `einstein`, which is extremely compressible and also has a very large c value.

As a point of comparison, the table also includes the grammar size and height obtained with a balanced version of RePair [35].[4] We modified the RePair grammar so as to remove the nonterminals that are referenced only once, inserting their right-hand side in that unique referencing place. The maximum grammar height is comparable with c as a measure of access cost in the grammar-compressed text. We can see that the height is considerably smaller than c, for the price of a weaker compression method.

Fig. 4 shows how the quotient between the number of phrases generated by the BAT-LZ parsers and by the optimal number of LZ phrases evolves as we allow longer chains. It can be seen that our Greedy BAT-LZ parser sharply outperforms the baselines in terms of compression performance. Our Greedy parser is, in turn, outperformed by Minmax, and Minmax is outperformed by our Greedier parser. The last one reaches a number of phrases that is only 1% over the optimal for c as low as 20–30, which is 0.7–1.1 times $\log_2 n$.

We also show in the figures the balanced grammar method, using the values of Table 1.[5] We can see that grammars are competitive, in some cases, with the simple baselines, but not with our new algorithms, which yield much better tradeoffs. The only exception to this analysis is `einstein`, which features a huge maximum c value of 1,736 and whose (extremely low) z value is approached only with c values near 700 using our BAT-LZ parsers. On this text, the balanced grammar offers an access time that is not achievable with our techniques.

Fig. 5 (left) zooms in the area where Greedier BAT-LZ reaches less than 10% extra space on top of standard LZ (excluding `einstein`).

[4] From `www.dcc.uchile.cl/gnavarro/software/repair.tgz`, directory `bal/`.

[5] For a fair comparison of space, we consider a tight space needed to support fast extraction: For each of the z phrases we count $\log_2 n$ bits to point to the source, $\log_2(n/z)$ bits for the length (as there are z lengths adding up to n; the cumulative sequence of lengths also allow finding the desired phrase using Elias-Fano codes [14, 16]), and 8 bits for the final symbol. For a grammar of size g and r symbols, we count $g \log_2 r$ bits for the right-hand sides, $g \log_2 n$ bits for the expansion lengths (cumulative on the right-hand sides to binary search them), and $r \log(g/r)$ bits to encode the rule lengths with Elias-Fano.

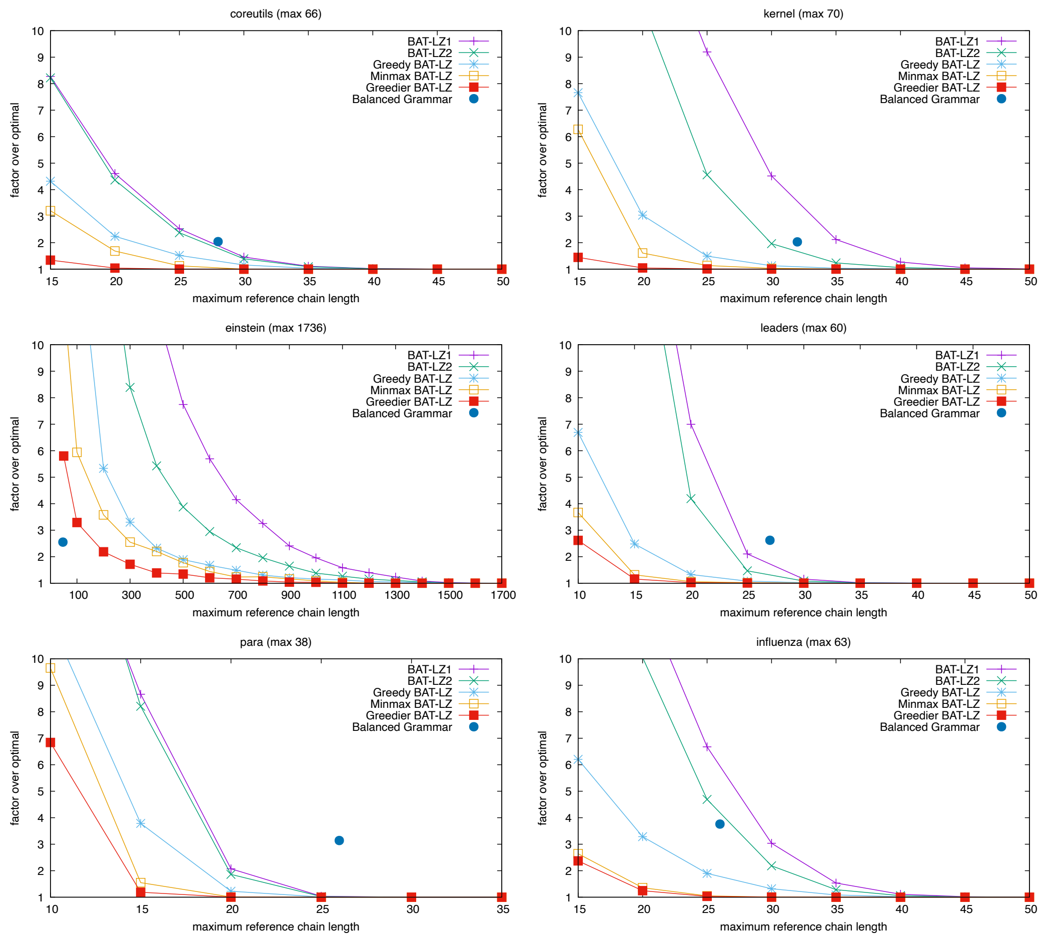

Figure 4 Overhead factor of number of BAT-LZ versus LZ phrases as a function of the maximum length c of a chain, for our different BAT-LZ parsers and a balanced grammar.

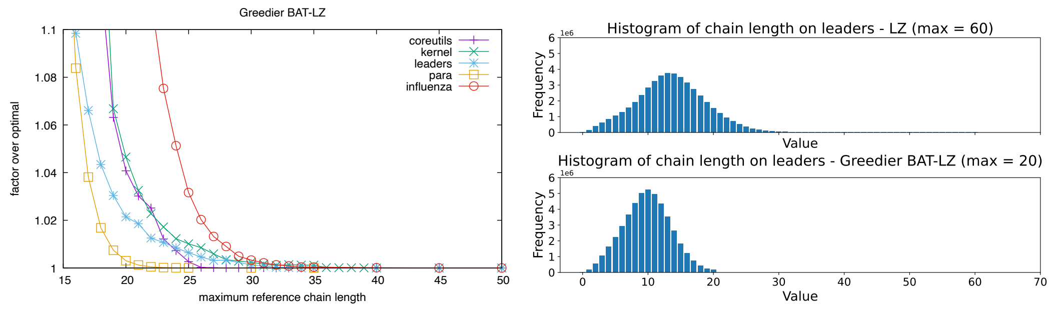

Figure 5 Left: detail of Fig. 4, for the Greedier BAT-LZ parser, focusing on the overheads below 10% over the LZ parse. Right: a comparison of histograms with shared x and y axis representing the chain length values on `leaders`; LZ on top and Greedier BAT-LZ with $c = 20$ on the bottom.

9 Discussion and Future Work

A first question is whether a Greedy BAT-LZ parsing can be produced in $o(n \log^3 n)$ time within linear space, either by solving our geometric problem faster or without recasting the parse into a geometric problem. This question seems to be answered in a very recent work, simultaneous with ours, that gives an $O(n \log \sigma)$-time greedy algorithm [2] based on simulating a suffix tree construction.[6] This algorithm is likely to be faster than ours in practice, but also to use much more space, which is relevant when compressing large repetitive texts. They also propose a parse similar to our BAT-LZ2, along with others that are incomparable to ours (in particular to Greedier, our best performing BAT-LZ parse).

Besides our reduction to a geometric problem being of independent interest, we believe that its flexibility can be exploited to compute more sophisticated parses in $O(n \log^3 n)$ time. For example, it might compute the Greedier parse if we extend the RMQ data structure to incorporate the additional optimization criterion (minmax of sources).

Other heuristics may also be of interest: there may be better ways to rank sources, other than their maximum chain length. Further, we have so far focused on reducing the worst case access time, but we might prefer to reduce the *average* access time. Our parsings do reduce it (Fig. 5 right), but this is just a side effect and has not been our main aim. So we pose as an open problem to efficiently build a leftward parse with bounded average reference chain length whose number of phrases is minimal, or in practice close to that of classical LZ.

Another intriguing line of work is to study the compression performance of BAT-LZ. An important result by Bannai et al. [2] shows that, letting g_{rl} be the size of the smallest run-length context-free grammar that generates a text T, there exists a BAT-LZ parse for T of size $O(g_{rl})$ if we let $c = \Theta(\log n)$ with some convenient multiplying constant. This bound is nearly optimal, because existing bounds [51] forbid the existence of BAT-LZ parses of size $O(g)$ – where $g \geq g_{rl}$ is the size of the smallest context-free grammar – with access time $c = O(\log^{1-\epsilon} n)$ for a constant $\epsilon > 0$. A relevant question is whether there is a BAT-LZ parse of size $O(z)$ – where $z \leq g_{rl}$ is the size of the Lempel-Ziv parse of T – with $c = \Theta(\log n)$.

Finally, from an application viewpoint, it would be interesting to incorporate BAT-LZ in the construction of the LZ-index [34] and measure how much its time performance improves at the price of an insignificant increase in space. Obtaining an efficient bounded version of the LZ-End parsing described in the same article [34] is also an interesting problem since efficient parsings for unrestricted LZ-End have appeared only recently [29, 28].

References

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

2 Hideo Bannai, Mitsuru Funakoshi, Diptarama Hendrian, Myuji Matsuda, and Simon J. Puglisi. Height-bounded Lempel-Ziv encodings. *CoRR*, abs/2403.08209, 2024.

3 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.

4 Djamal Belazzougui and Simon J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2053–2071, 2016.

[6] They use a slightly modified definition of Lempel-Ziv parses, which has no explicit character at the end of the phrases. The precise consequences of this difference are not totally clear to us.

5 Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015.

6 Timothy M. Chan, Yakov Nekrich, Saladi Rahul, and Konstantinos Tsakalidis. Orthogonal point location and rectangle stabbing queries in 3-d. *Journal of Computational Geometry*, 13(1), 2022.

7 Timothy M. Chan and Konstantinos Tsakalidis. Dynamic orthogonal range searching on the RAM, revisited. *Journal of Computational Geometry*, 9(2):45–66, 2018.

8 Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

9 Gang Chen, Simon J. Puglisi, and William F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1:605–623, 2008.

10 Ferdinando Cicalese and Francesca Ugazio. On the complexity and approximability of bounded access Lempel Ziv coding. *CoRR*, abs/2403.15871, 2024. Submitted.

11 David R. Clark. *Compact PAT Trees*. PhD thesis, University of Waterloo, Canada, 1996.

12 Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. *Journal of Computer and System Sciences*, 118:53–74, 2021.

13 Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

14 P. Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21:246–260, 1974.

15 Jonas Ellert, Johannes Fischer, and Max Rishøj Pedersen. New advances in rightmost Lempel-Ziv. In *Proc. 30th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 188–202, 2023.

16 R. Fano. On the number of bits required to implement an associative memory. Memo 61, Computer Structures Group, Project MAC, Massachusetts, 1971.

17 Martin Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143. IEEE Computer Society, 1997.

18 J. Fischer. Combined data structure for previous- and next-smaller-values. *Theoretical Computer Science*, 412(22):2451–2456, 2011.

19 Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

20 Johannes Fischer, Tomohiro I, and Dominik Köppl. Lempel Ziv computation in small space (LZ-CISS). In *Proc. 26th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 9133, pages 172–184, 2015.

21 Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.

22 Moses Ganardi, Artur Jez, and Markus Lohrey. Balancing straight-line programs. *Journal of the ACM*, 68(4):article 27, 2021.

23 Keisuke Goto and Hideo Bannai. Simpler and faster Lempel Ziv factorization. In *Proc. 23rd Data Compression Conference (DCC)*, pages 133–142, 2013.

24 D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

25 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 139–150, 2013.

26 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 189–200, 2013.

27 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lazy Lempel-Ziv factorization algorithms. *ACM Journal of Experimental Algorithmics*, 21(1):2.4:1–2.4:19, 2016.

28 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in compressed space. In *Proc. 27th Data Compression Conference (DCC)*, pages 350–359, 2017.

29 Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in linear time. In *Proc. 25th Annual European Symposium on Algorithms (ESA)*, pages 53:1–53:14, 2017.

30 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-End parsing. In *Proc. 33rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2847–2866, 2022.

31 John C. Kieffer and En-Hui Yang. Grammar-based codes: A new class of universal lossless source codes. *IEEE Transactions on Information Theory*, 46(3):737–754, 2000.

32 Dominik Köppl and Kunihiko Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. 26th Data Compression Conference (DCC)*, pages 3–12, 2016.

33 Sebastian Kreft and Gonzalo Navarro. Lz77-like compression with fast random access. In *Proc. 20th Data Compression Conference (DCC)*, pages 239–248, 2010.

34 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

35 J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

36 Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22(1):75–81, 1976.

37 M. Lothaire. *Algebraic Combinatorics on Words.* Cambridge University Press, 2002.

38 Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

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

40 J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016.

41 Gonzalo Navarro. *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

42 Gonzalo Navarro. Indexing highly repetitive string collections, part I: Repetitiveness measures. *ACM Computing Surveys*, 54(2):article 29, 2021.

43 Gonzalo Navarro. Indexing highly repetitive string collections, part II: Compressed indexes. *ACM Computing Surveys*, 54(2):article 26, 2021.

44 Yakov Nekrich. Orthogonal range searching in linear and almost-linear space. *Computational Geometry*, 42(4):342–351, 2009.

45 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

46 Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 15–26, 2011.

47 Simon J. Puglisi, William F. Smyth, and Andrew Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):article 4, 2007.

48 Michael Rodeh, Vaughan R. Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

49 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

50 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

51 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 7922, pages 247–258, 2013.

52 Peter Weiner. Linear pattern matching algorithms. In *Proc. 14th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE Computer Society, 1973.