Height-Bounded Lempel-Ziv Encodings

Hideo Bannai
M&D Data Science Center, Tokyo Medical and Dental University (TMDU), Japan

Mitsuru Funakoshi
NTT Communication Science Laboratories, Kyoto, Japan

Diptarama Hendrian
M&D Data Science Center, Tokyo Medical and Dental University (TMDU), Japan

Myuji Matsuda
Graduate School of Medical and Dental Sciences, Tokyo Medical and Dental University (TMDU), Japan

Simon J. Puglisi
Department of Computer Science, University of Helsinki, Finland

Abstract

We introduce height-bounded LZ encodings (LZHB), a new family of compressed representations that are variants of Lempel-Ziv parsings with a focus on bounding the worst-case access time to arbitrary positions in the text directly via the compressed representation. An LZ-like encoding is a partitioning of the string into phrases of length 1 which can be encoded literally, or phrases of length at least 2 which have a previous occurrence in the string and can be encoded by its position and length. An LZ-like encoding induces an implicit referencing forest on the set of positions of the string. An LZHB encoding is an LZ-like encoding where the height of the implicit referencing forest is bounded. An LZHB encoding with height constraint h allows access to an arbitrary position of the underlying text using $O(h)$ predecessor queries.

While computing the optimal (i.e., smallest) LZHB encoding efficiently seems to be difficult [Cicalese & Ugazio 2024, arXiv, to appear at DLT 2024], we give the first linear time algorithm for strings over a constant size alphabet that computes the greedy LZHB encoding, i.e., the string is processed from beginning to end, and the longest prefix of the remaining string that can satisfy the height constraint is taken as the next phrase. Our algorithms significantly improve both theoretically and practically, the very recently and independently proposed algorithms by Lipták et al. (CPM 2024).

We also analyze the size of height bounded LZ encodings in the context of repetitiveness measures, and show that there exists a constant c such that the size $\hat{z}_{HB(c\log n)}$ of the optimal LZHB encoding whose height is bounded by $c\log n$ for any string of length n is $O(\hat{g}_{\mathrm{rl}})$, where $\hat{g}_{\mathrm{rl}}$ is the size of the smallest run-length grammar. Furthermore, we show that there exists a family of strings such that $\hat{z}_{HB(c\log n)} = o(\hat{g}_{\mathrm{rl}})$, thus making $\hat{z}_{HB(c\log n)}$ one of the smallest known repetitiveness measures for which $O(\mathrm{polylog}n)$ time access is possible using linear $(O(\hat{z}_{HB(c\log n)}))$ space.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases Lempel-Ziv parsing, data compression

Digital Object Identifier 10.4230/LIPIcs.ESA.2024.18

Related Version *Full Version*: https://arxiv.org/abs/2403.08209

Supplementary Material *Software (Source Code)*: https://github.com/dscalgo/lzhb
archived at swh:1:dir:76aa98e06ac56d0527e30f362413dd53d76e5a1c

Funding *Hideo Bannai*: JSPS KAKENHI Grant Number JP24K02899.
Simon J. Puglisi: Academy of Finland grants 339070 and 351150.

1 Introduction

Dictionary compressors are a family of algorithms that produce compressed representations of input strings essentially as sequences of copy and paste operations. These representations are widely used in general compression tools such as gzip and LZ4, and have also received much recent attention since they are especially effective for *highly repetitive* data sets, such as versioned documents and pangenomes [18].

A desirable operation to support on compressed data is that of *random access* to arbitrary positions in the original data. Access should be supported without decompressing the data in its entirety, and ideally by decompressing little else than the sought positions. Recent years have seen intense research on the access problem in the context of dictionary compression, most notably for grammar compressors (SLPs) and Lempel-Ziv (LZ)-like schemes. The general approach is to impose structure on the output of the compressor and in doing so add small amounts of information to support fast queries. While LZ77 [28] is known to be theoretically and practically one of the smallest compressed representations that can be computed efficiently, a long-standing question is whether $O(\text{polylog}(n))$ time access can be achieved with a data structure using $O(z)$ space [18], where z is the size of the LZ77 parsing.

In an LZ-like compression scheme, the input string T of length n is *parsed* into $z' \leq n$ *phrases* (substrings of T), where each phrase is of length 1, or, is a phrase of length at least 2 which has a previous occurrence. Each phrase can be encoded by a pair (ℓ, s), where $\ell \geq 1$ is the length of the phrase, and s is the symbol representing the phrase if $\ell = 1$, or otherwise, a position of a previous occurrence (source) of the phrase. A greedy left-to-right algorithm that takes the longest prefix of the remaining string that can be a phrase leads to the aforementioned LZ77 parsing. An LZ-like encoding of a string induces a *referencing forest*, where each position of the string is a node, phrases of length 1 are roots of a tree in the forest, and the parent of all other positions are induced by the previous occurrence of the phrase it is contained in. The principle hurdle to support fast access to an arbitrary symbol $T[i]$ from an LZ-like parsing is to trace the symbol to a root in the referencing forest. For LZ77, the height of the referencing forest can be $\Theta(n)$.

Contributions In this paper, we explore LZ-like parsings specifically designed to bound the height of the referencing forest, i.e., LZ-like parsers in which the parsing rules prevent the introduction of phrases that would exceed a specified maximum height of the referencing forest. Our contributions are summarized as follows:

1. We propose the first linear time[1] algorithm LZHB for computing the greedy height-bounded LZ-like encoding, i.e., the string is processed from beginning to end, and each phrase is greedily taken as the longest prefix of the remaining string that can satisfy the height constraint. This problem was first considered by Kreft and Navarro [12], where an algorithm called LZ-COST, with no efficient implementation, is mentioned. Very recently, contemporaneously and independently of our work, this problem was also revisited by Lipták et al. [14], who presented an algorithm called greedy-BATLZ running in $O(n \log^3 n)$ time. Lipták et al. also proposed greedier-BATLZ that runs in $O(z'n^2 \log n) = O(n^3 \log n)$ time, where z' is the size of the parsing, further adding the requirement that the previous occurrence of the phrase is chosen so as to minimize the maximum height. We show that our algorithms can be modified to support this heuristic in $O(n \log \sigma + occ) = O(n \log \sigma + z'n) = O(n^2)$ time, where occ is the total number of previous occurrences of all phrases.

[1] Assuming a constant-size alphabet.

2. We show that our algorithms allow for simple, lightweight implementations based on suffix arrays and segment trees, and that our implementations are an order of magnitude (or two) faster than the recent implementations of related schemes by Lipták et al.
3. We propose a new LZ-like encoding, which may be of independent theoretical interest, which can be considered as a run-length variant of standard LZ-like encodings which can potentially reduce the referencing height further.
4. We analyze the size of height bounded LZ-like encodings in the context of repetitiveness measures [18]. We show that for some constant c, there exits data structures of $O(\hat{z}_{HB(c\log n)})$ size that allow access in $O(\text{polylog}n)$ time, where $\hat{z}_{HB(c\log n)}$ is the size of the smallest LZ-like encoding whose height is bounded by $c\log n$. Furthermore $\hat{z}_{HB(c\log n)}$ is always $O(\hat{g}_{\text{rl}})$, and further can be $o(\hat{g}_{\text{rl}})$ for some family of strings, where $\hat{g}_{\text{rl}}$ denotes the size of the smallest run-length grammar (RLSLP) [21] representing the string. This makes $\hat{z}_{HB(c\log n)}$ one of the smallest known measures that can achieve $O(\text{polylog}n)$ time access using linear space. The other two are the size z_e of the LZ-End parse [12, 13, 10], and the size $\hat{g}_{it(d)}$ of the smallest iterated SLPs (ISLPs) [20]. Note that while there exist string families such that $\hat{z}_{HB(c\log n)} = o(z_e)$, we do not yet know if it can be that $z_e = o(\hat{z}_{HB(c\log n)})$.

2 Preliminaries

2.1 Strings

Let Σ be a set of symbols called the *alphabet*, and Σ^* the set of strings over Σ. For any non-negative integer n, Σ^n is the set of strings of length n. For any string $x \in \Sigma^*$, $|x|$ denotes the length of x. The empty string (the string of length 0) is denoted by ε. For any integer $i \in [1, |x|]$, $x[i]$ is the ith symbol of x, and for any integers $i, j \in [1, |x|]$, $x[i..j] = x[i]\cdots x[j]$ if $i \leq j$, and ε otherwise. We will write $x[i..j)$ to denote $x[i..j-1]$. For string w, $p \in [1, |w|]$ is a *period* of w if $w[i] = w[i+p]$ for all $i \in [1, |w|-p]$. For any string x, $i \in [1, |w|-|x|+1]$ is an *occurrence* of x in w if $w[i..i+|x|) = x$. For any position $i > 1$, the length $\text{lpf}_w(i)$ of the longest previously occurring factor of position i is $\text{lpf}_w(i) = \max\{l \mid w[i'..i'+l) = w[i..i+l), i' < i\}$, and let $\text{lpf}_w(1) = 0$. For any i, ℓ with $1 \leq i \leq i+\ell \leq |w|$, the leftmost occurrence of the length-ℓ substring starting at i is $\text{lmocc}_w(i, \ell) = \min\{j < i \mid w[j..j+\ell) = w[i..i+\ell)\}$, which can be undefined when i is the leftmost occurrence of $w[i..i+\ell)$. We will omit the subscript when the underlying string considered is clear.

A string that is both a prefix and suffix of a string is called a *border* of that string. The *border array* B of a string w is an array $B[1..|w|]$ of integers, where the $B[i]$ is the length of the longest proper border of $w[1..i]$. The border array is computable in $O(|w|)$ time and space [11], in an on-line fashion, i.e., at each step $i = 1, \ldots, |w|$, the border array of $B[1..i]$ is obtained in amortized constant ($O(i)$ total) time. Notice that the minimum period of $w[1..i]$ is $i - B[i]$. Thus, the minimum periods of all prefixes of a (possibly growing) string can be computed in time linear in the length of the string.

▶ **Lemma 1** ([11]). *The minimum period of a semi-dynamic string that allows symbols to be appended is non-decreasing and computable in amortized constant time per symbol.*

The following is another useful lemma, rediscovered many times in the literature.

▶ **Lemma 2** (e.g. Lemma 8 of [22]). *The set of occurrences of a word w inside a word v which is exactly twice longer than w forms a single arithmetic progression.*

Our parsing algorithms make use of the *suffix tree* data structure [26]. We assume some familiarity with suffix trees, and give only the basic properties essential to our methods below. We refer the reader to the many textbook treatments of suffix trees for further details [9, 17].

1. The suffix tree of a string T, denoted $\mathcal{T}_T$ (or just $\mathcal{T}$ when the context is clear) is a compacted trie containing all the suffixes of T. Each leaf of the suffix tree corresponds to a suffix of the string and is labelled with the starting position of that suffix.
2. The suffix tree can be constructed in an online fashion via Ukkonen's algorithm [24], in $O(|T| \log \sigma)$ total time, i.e., at each step $i = 1, \ldots, |T|$, the suffix tree of $T[1..i]$ is obtained in amortized $O(\log \sigma)$ ($O(i \log \sigma)$ total) time. For linearly-sortable alphabets, it is possible to compute the suffix tree offline in $O(|T|)$ time [7].
3. Suffix trees can support *prefix queries* that return a pair (ℓ, s), where ℓ is the length of the longest prefix of the query string that occurs in the indexed string, and s is the leftmost position of an occurrence of that prefix. Prefix queries can be conducted simply by traversing the suffix tree from the root with the query string and take $O(\ell \log \sigma)$ time. The query string can be processed left-to-right, where each symbol is processed in $O(\log \sigma)$ time, which is the cost of finding the correct out-going child edge from the current suffix tree node. Prefix queries can be answered in the same time complexity even if the underlying string T is extended in an online fashion as mentioned in Property 2.
4. The suffix tree of T can be preprocessed in linear time to answer, given i, ℓ, the leftmost occurrence in T of the substring $T[i..i + \ell)$, i.e., $\text{lmocc}_T(i, \ell)$ in constant time [2].

2.2 LZ encodings and random access

An LZ-like parsing of a string is a decomposition of the string into phrases of length 1 (literal phrases), or a phrase of length at least 2 which has a previous occurrence in the string. An LZ-like *encoding* is a representation of the LZ-like parsing, where literal phrases are encoded as the pair $(1, c)$, where $c \in \Sigma$ is the phrase itself, and phrases of length at least 2 are encoded as the pair (ℓ, s), where $\ell \geq 2$ is the length of the phrase and s is a previous occurrence (or the *source*) of the phrase. Although LZ parsing and encoding are sometimes used as synonyms, we differentiate them in that a parsing only specifies the length of each phrase, while an encoding specifies the previous occurrence of each phrase as well. The *size* of an LZ-like encoding is the number of phrases. For example, for the string `ababacbabac`, we can have an encoding $(1, \texttt{a}), (1, \texttt{b}), (3, 1), (1, \texttt{c}), (5, 2)$ of size 5. A common variant of LZ-like encodings adds an extra symbol explicitly to the phrase, and each phrase is encoded by a triplet. For simplicity, our description will not consider this extra symbol; the required modifications for the algorithms to include this are straightforward. We note that our experiments in Section 6 will include the extra character, in order to compare with implementations of previous work.

An LZ-like encoding of a string induces an implicit referencing forest, where: each position of the string is a node, literal phrases are roots of a tree in the forest, and the parent of all other positions are induced by the source of the phrase it is contained in. For example, for the encoding of the string `ababacbabac` as above, the referencing forest is shown in Figure 1. Let $\mathcal{E} = (\ell_1, s_1), \ldots, (\ell_{z'}, s_{z'})$ be an LZ-like encoding of string T. Given an arbitrary position i, suppose we would like to retrieve the symbol $T[i]$. This can be done by traversing the implicit referencing forest using predecessor queries. We first find the phrase (ℓ_j, s_j) with starting position b_j[2] that the position i is contained in, i.e., j s.t. $b_j \leq i < b_j + \ell_j$. Then, we can

[2] We note that it is possible to encode each phrase as (b_j, s_j), since then, $\ell_j = b_{j+1} - b_j$.

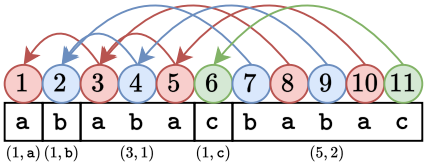

Figure 1 An example of an implicit referencing forest induced from the LZ-like encoding $(1, \mathtt{a}), (1, \mathtt{b}), (3, 1), (1, \mathtt{c}), (5, 2)$ for the string $\mathtt{ababacbabac}$.

deduce that the parent position of i is $i' = s_j + (i - b_j)$. This is repeated until a literal phrase $(1, c)$ is reached, in which case $T[i] = c$. The number of times a parent must be traversed (i.e., the number of predecessor queries) is bounded by the *height* of the referencing forest. This notion of height for LZ-like encodings was introduced by Kreft and Navarro [13].

3 Height bounded LZ-like encodings

We first consider modifying the definition of the height of the referencing forest under some conditions, by implicitly rerouting edges. Consider the case of a unary string $T = \mathtt{a}^n$ and LZ-like encoding $(1, \mathtt{a}), (n-1, 1)$. The straightforward definition above gives a height of $n - 1$ for position n. This is a result of the second phrase being self-referencing, i.e., the phrase overlaps with its referenced occurrence, and each position references its preceding position. An important observation is that self-referencing phrases are periodic; in particular a self-referencing phrase starting at position b_i with source s_i has period $b_i - s_i$. Due to this periodicity, any position in the phrase can refer to an appropriate position in $T[s_i..b_i)$.

More formally, let $\mathcal{E} = (\ell_1, s_1), \ldots, (\ell_{z'}, s_{z'})$ be an LZ-like encoding of T. For any position i, let b_j be the starting position of the phrase (ℓ_j, s_j) such that $b_j \leq i < b_j + \ell_j$. Then, $T[i] = s_j$ if $\ell_j = 0$, and $T[i] = T[s_j + (i - b_j)] = T[s_j + ((i - b_j) \bmod (b_j - s_j))]$ otherwise.

We define the height $height_{\mathcal{E}}(i)$ of position i by an encoding $\mathcal{E}$ as

$$height_{\mathcal{E}}(i) = \begin{cases} 0 & \text{if } \ell_j = 0 \\ height_{\mathcal{E}}(s_j + ((i - b_j) \bmod (b_j - s_j))) + 1 & \text{otherwise.} \end{cases} \tag{1}$$

The subscript will be omitted if the underlying encoding considered is clear. Since j can be computed using a predecessor query on the set of phrase starting positions, $T[i]$ can be computed using $O(height(i)Q(z))$ time using a data structure of size $O(z)$, where $Q(z)$ is the time required for predecessor queries on z elements in $[1, n]$ using $O(z)$ space. $Q(z)$ is $O(\log n)$ using a simple binary search, and faster using more sophisticated methods [19].

For example, for the encoding $(1, \mathtt{a}), (1, \mathtt{a}), (1, \mathtt{b}), (3, 2), (1, \mathtt{c}), (4, 3)$ of string $\mathtt{aababacbaba}$, the heights are: $0, 0, 0, 1, 1, 1, 0, 1, 2, 2, 2$. Notice that the phrase $(3, 2)$ that starts at position 4, representing $\mathtt{aba}$ is self-referencing, and the height of the last position in the phrase (position 6) is 1, since it is defined to reference position $2 = 2 + (6 - 4) \bmod (6 - 4)$. We note that even with this modified definition, the height of the optimal LZ-like encoding (LZ77) can be $\Theta(n)$.

In order to bound the worstcase query time complexity for access operations, we consider LZ-like encodings with bounded height. An *h-bounded LZ-like encoding* is an LZ-like encoding where $\max\{height(i) \mid i \in [1, n]\} \leq h$.

There are many ways one could enforce such a height restriction. Unfortunately, finding the smallest such encoding was very recently shown to be NP-hard by Cicalese and Ugazio [5]. Therefore, we propose several greedy heuristics to compute h-bounded LZ-like encodings.

Given an encoding for $T[1..b_j)$ (which defines $height(i)$ for any $i \in [1, b_j)$), the next phrase (ℓ_j, s_j) starting at position b_j is defined as follows.

LZHB1 ℓ_j is the largest value such that $T[s_j..s_j + \ell_j)$ satisfies the height constraint, where $s_j = \text{lmocc}_T(b_j, \text{lpf}_T(b_j))$, i.e., the leftmost occurrence of the longest previously occurring factor at position b_j.

LZHB2 ℓ_j is the largest value such that for all $1 \leq \ell' \leq \ell_j$, the leftmost occurrence of $T[b_j..b_j + \ell')$ satisfies the height constraint. s_j is the leftmost occurrence of $T[b_j..b_j + \ell_j)$.

LZHB3 ℓ_j is the largest value such that for all $1 \leq \ell' \leq \ell_j$, there exists some previous occurrence of $T[b_j..b_j+\ell')$ that satisfies the height constraint. s_j is the leftmost occurrence of $T[b_j..b_j + \ell_j)$ that satisfies the height constraint.

Note that for all variations, $\ell_j = 1$ if there is no occurrence of $T[b_j]$ that satisfies the corresponding conditions described above, in which case, $s_j = T[b_j]$.

We note that LZHB1 corresponds to the baseline algorithm BATLZ2 in [14]. LZHB3 essentially corresponds to greedy-BATLZ in [14] as well, but greedy-BATLZ does not require that the occurrence is leftmost. The difference between LZHB2 and LZHB3 lies in the priority of the choice of the leftmost occurrence and when to check the height constraint. LZHB2 greedily extends the prefix, checking each time whether its leftmost occurrence satisfies the height constraint. LZHB3 finds the leftmost occurrence out of the longest prefix that can satisfy the height constraint. Note that the condition "for all $1 \leq \ell' \leq \ell_j$" is not actually required for LZHB3 and $\ell' = \ell_j$ suffices, but is included for consistency with LZHB2. Note that when the height is unbounded, the size of all three variants will be equivalent to the regular LZ77 parsing.

We show that LZHB1 and LZHB2 can be computed in $O(n)$ time and space for linearly-sortable alphabets, and LZHB3 can be computed online in $O(n \log \sigma)$ time and $O(n)$ space for general ordered alphabets.

We also propose a new encoding of LZ-like parsings that re-routes edges of the implicit referencing forest in an attempt to further reduce the heights, again using periodicity. In the case of self-referencing phrases, our definition of height in Equation (1) utilized the fact that self-referencing phrases implies a period $b_j - s_j$ in the phrase. Using this period, we could re-route the parents of all positions inside the phrase to the first period in the previous occurrence of the phrase, which is outside the phrase. Here, we make two further observations: 1) since the referenced substring is the first period of the phrase, we could extended the phrase for free while the period continues, possibly making the phrase longer, and 2) the referenced substring could have a previous occurrence further to the left, which should tend to have shorter heights. Therefore, if we were to store the period of the phrase explicitly, this could be applied to and benefit all periodic phrases, self-referencing or not.

Specifically then, under this scheme, a phrase with period 1 can be encoded as the triple $(\ell, c, 1)$, where c is a symbol, and a phrase with period $p \geq 2$ can be encoded as the triple (ℓ, s, p), where $\ell \geq p$ is the length of the phrase with period $p \geq 2$, and s is a previous occurrence of the length-p prefix of the phrase. We will call such an encoding, a *modified* LZ-like encoding. For example, $(2, \mathtt{a}, 1), (1, \mathtt{b}, 1), (3, 2, 2), (1, \mathtt{c}, 1), (4, 3, 2)$ would be a modified LZ-like encoding for the string `aababacbaba`.

The implicit referencing forest and heights of the modified LZ-like encoding can be defined analogously: a position i in the jth phrase (ℓ_j, s_j, p_j) that begins at position b_j, i.e., $b_j \leq i < b_j + \ell_j$, will reference position $s_j + ((i - b_j) \bmod p_j) \bmod (b_j - s_j)$, where the second mod is to deal with the case where the occurrence of the prefix period is self-referencing. For the above example, the heights are: $0, 0, 0, 1, 1, 1, 0, 1, 2, 1, 2$.

A greedy left-to-right algorithm computes the jth phrase (ℓ_j, s_j, p_j) starting at b_j as:

LZHB4 ℓ_j is the largest value such that $T[b_j..b_j+\ell_j)$ has period 1, or, for all $1 \leq \ell' \leq \ell_j$, there exists some previous occurrence of $T[b_j..b_j+p')$ that satisfies the height constraint, where p' is the minimum period of $T[b_j..b_j+\ell')$. s_j is the leftmost occurrence of $T[b_j..b_j+\ell_j)$ that satisfies the height constraint, and p_j is the minimum period of $T[b_j..b_j+\ell_j)$.

4 Efficient parsing algorithms for height-bounded encodings

4.1 A linear time algorithm for LZHB1

▶ **Theorem 3.** *For any integer h, an h-bounded encoding based on LZHB1 for a string over a linearly-sortable alphabet can be computed in linear time and space.*

Proof. The algorithm maintains an array $H[1..n]$ of integers, initially set to 0. Phrases are produced left to right. The algorithm maintains the invariant that when the encoding is computed up to position i, $H[j]$ gives the height in the referencing forest of position $j < i$. Using Property 2 in Section 2.1, we build the suffix tree of T in linear time, and preprocess it, again in linear time, for Property 4. We also precompute and store all the lengths of the longest previously occurring factor at each position, i.e., $\mathrm{lpf}_T(i)$ for all $1 \leq i \leq n$, in linear time [6].

Suppose we have computed the h-bounded encoding for $T[1..b_j)$ and would like to compute the jth phrase (ℓ_j, s_j) starting at position b_j. We can compute $s_j = \mathrm{lmocc}_T(b_j, \mathrm{lpf}_T(b_j))$, i.e., the leftmost occurrence of the longest previously occurring factor starting at position b_j, in constant time (Property 4). The encoding of the phrase starting at b_j is then (s_j, ℓ_j), where $\ell_j \leq \ell$ is the largest value such that every value in $H[s_j.. \min(b_j, s_j+\ell_j))$ is less than h. This can be computed in $O(\ell_j)$ time by simply scanning the above values in H. Notice that we do not need to check the height beyond position b_j, since this would imply that the phrase is self-referencing, and the remaining heights will be copies of the first period of the phrase which were already checked to satisfy the height constraint. After pair (s_j, ℓ_j) is determined, values in $H[b_j..b_j+\ell_j)$ are determined according to Equation (1), in $O(\ell_j)$ time.

The total time is thus proportional to the sum of the phrase lengths, which is $O(n)$. ◀

4.2 A linear time algorithm for LZHB2

The difference between LZHB1 and LZHB2 is that while LZHB1 fixes the source to the leftmost occurrence of the longest previously occurring factor starting at b_j, LZHB2 considers the leftmost position of the candidate substring. Since shorter lengths allow the source to be further to the left where heights tend to be shorter, it may allow longer phrases.

We show below that we can still achieve a linear time parsing algorithm.

▶ **Theorem 4.** *For any integer h, an h-bounded encoding based on LZHB2 for a string over a linearly-sortable alphabet can be computed in linear time and space.*

Proof. The same steps as in the first paragraph in the description of LZHB1 in Theorem 3 are taken (except for the precomputing of lpf_T which is not needed here).

Suppose we have computed the h-bounded encoding for $T[1..b_j)$ and would like to compute the jth phrase (ℓ_j, s_j) starting at position b_j. We start with $\ell = 1$, and increase ℓ incrementally until there is no previous occurrence of $T[b_j..b_j+\ell)$, or the leftmost occurrence of $T[b_j..b_j+\ell)$ violates the height constraint, and will use the last valid value of ℓ for ℓ_j.

For a given ℓ, using Property 4, we can obtain the leftmost occurrence of $T[b_j..b_j+\ell)$, i.e., $s'_j = \mathrm{lmocc}_T(b_j, \ell)$, in constant time. We need to check whether all values in $H[s'_j..\min(b_j, s'_j+\ell))$ are less than h. Since s'_j may change for different values of ℓ, we potentially need to check all values in $H[s'_j..\min(b_j, s'_j+\ell))$.

To do this efficiently, we maintain another array C, where $C[i]$ stores the position j in H, such that j is the smallest position in H that is greater than or equal to i, such that $H[j] = h$, if it exists, or ∞ otherwise, i.e., $C[i] = \min(\{\infty\} \cup \{j \geq i \mid H[j] = h\})$. Then, $\max(H[i..j]) + 1 \leq h$ if and only if $C[i] > j$. Thus, checking the height constraint can be done in constant time for a given ℓ.

After (s_j, ℓ_j) are determined, values in $H[i..i+\ell_j)$ are determined according to Equation (1), in $O(\ell_j)$ time. The array C can be maintained in linear total time: all values in C are initially set to ∞, and whenever we store the value h in $H[j]$ for some j, we set $C[i'] = j$ for all i' such that $\max(\{0\} \cup \{i < j \mid H[i] = h\}) < i' \leq j$, i.e., we update all values in $C[1..i)$ that were ∞. Clearly each element of C is set at most once. Thus, the total time is $O(n)$. ◀

4.3 An $O(n \log \sigma)$ time algorithm for LZHB3

▶ **Theorem 5.** *For any integer h, an h-bounded encoding based on LZHB3 for a string can be computed in $O(n \log \sigma)$ time and $O(n)$ space.*

Proof. Our algorithm is a slight modification of a folklore algorithm to compute LZ77 in $O(n \log \sigma)$ time based on Ukkonen's online algorithm for computing suffix trees [24] described, e.g., by Gusfield [9]. We first describe a simple version which does not allow self-references.

At a high level, when computing the jth phrase that starts at position b_j, we use the suffix tree of $T[1..b_j)$ so that we can find the longest previous occurrence of $T[b_j..]$. We can find this by simply traversing the suffix tree from the root with $T[b_j..]$ as long as possible (Property 3 in Section 2.1). Once we know the length ℓ_j of the phrase, we simply append the phrase $T[b_j..b_j+\ell_j)$ to the suffix tree to obtain the suffix tree for $T[1..b_{j+1})$, and continue. An occurrence (particularly, the leftmost occurrence) can be obtained easily by storing this information on each edge when it is constructed.

For our height-bounded case, the precise height of a position in a phrase can only be determined after ℓ_j is determined, and if $\ell_j \geq 2$, the leftmost occurrence s_j of $T[b_j..b_j+\ell)$ that satisfies the height constraint must be determined. Since a reference adds 1 to the height, we know that the height constraint is satisfied if and only if we reference positions that have height less than h. Therefore, when adding the symbols of the phrase $T[b_j..b_j+\ell_j)$ to the suffix tree after determining their heights, we change symbols corresponding to positions with height h to a special symbol \$ which does not match any other symbols. In other words, for $T[1..b_j)$ and their heights $H[1..b_j)$, we use and maintain a suffix tree for $T'[1..b_j)$, where $T'[i] = T[i]$ if $H[i] < h$ and $T'[i] =$ \$ otherwise. This simple modification allows us to reference only and all substrings with heights less than h, so that we can find the longest prefix of $T[b_j..]$ that occurs in $T[1..b_j)$ and also satisfies the height constraint, by simply traversing the suffix tree of $T'[1..b_j)$ (Property 3).

Next, in order to allow self-references, we utilize a combinatorial observation made previously, that self-references lead to periodicity. Let $x = T[b_j..b_j+|x|)$ be the longest prefix of $T[b_j..n]$ such that there is a non-self-referencing occurrence in $T'[1..b_j)$. We can find a longer self-referencing prefix of $T[b_j..n]$, if it exists, as follows: Any such occurrence must be prefixed by x, and in order for the phrase to be self-referencing, this x must occur as a substring in $T[b_j-|x|..b_j+|x|-1)$. Since the heights after the first period of the phrase are essentially defined as copies of the heights of the first period of the phrase (see Equation (1)), any occurrence of x (and any of its extensions) at or after the leftmost position t_j in $[b_j-|x|, b_j)$ such that $H[t_j..b_j) < h$, would satisfy the height constraint.

If there is no occurrence of x in $T[\max(t_j, b_j - |x|)..b_j + |x| - 1)$, then a longer self-referencing phrase satisfying the height constraint cannot exist, so $\ell_j = |x|$. If there is only one such occurrence of x, we can use this occurrence and extend the phrase by naive symbol comparisons to obtain ℓ_j. If there are more than one such occurrence, it follows that there are at least three occurrences of x in $T[\max(t_j, b_j - |x|)..b_j + |x|)$, a string of length (at most) $2|x|$. From Lemma 2, this implies that the occurrences of x form an arithmetic progression, whose common difference is the minimum period of x. Thus, extending the phrase as above from any occurrence of x in $T[\max(t_j, b_j - |x|)..b_j + |x| - 1)$ will lead to the same maximal length due to the periodicity, i.e., ℓ_j is the maximum value such that $T[b_j..b_j + \ell_j)$ has the same minimum period as x. We can find all such occurrences (which obviously includes the leftmost) in $O(\ell_j)$ time using any linear time pattern matching algorithm, e.g., KMP [11].

The time to compute phrase (ℓ_i, s_i) is $O(\ell_i \log \sigma)$, so we spend $O(n \log \sigma)$ time in total. ◀

4.4 An $O(n \log \sigma)$ time algorithm for LZHB4

▶ **Theorem 6.** *For any integer h, an h-bounded encoding for a string based on LZHB4 can be computed in $O(n \log \sigma)$ time and $O(n)$ space.*

Proof. The algorithm is similar to LZHB3, in that it computes the suffix tree of $T[1..b_j)$ in an online manner, and after determining the new phrase $T[b_j..b_j + \ell_j)$ and their heights $H[b_j..b_j + \ell_j)$, symbols of the new phrase are added to the suffix tree except when its position has height h, in which case \$ is added.

In order to determine the LZHB4 phrase, we first compute the LZHB3 phrase, i.e., the longest prefix of $T[b_j..n]$ that has a previous occurrence satisfying the height constraint. Let its length be ℓ'. This implies that any prefix period of $T[b_j..n]$ that can be referenced (and satisfies the height constraint) is at most ℓ'. Since prefix periods are non-decreasing (Lemma 1), we have that the LZHB4 phrase $T[b_j..b_j + \ell_j)$ is the longest prefix of $T[b_j..n]$ with period $p_j \leq \ell'$. This can be computed in $O(\ell_j)$ time (again Lemma 1). The leftmost occurrence of $T[b_j..b_j + p_j)$ to be referenced can be found simply by traversing the suffix tree if p_j is at most the length of the longest non-self-referencing prefix of the LZHB3 phrase, and otherwise, will coincide with (the starting position of) the self-referencing occurrence of the LZHB3 phrase. Since $\ell' \leq \ell_j$ must hold as well, each LZHB4 phrase is computed in $O(\ell_j \log \sigma)$ time, and thus, LZHB4 can be computed in $O(n \log \sigma)$ total time and $O(n)$ space. ◀

4.5 Algorithms for the greedier heuristic

Lipták et al. [14] also propose a *greedier* heuristic for height bounded LZ-like encodings, in which, for any phrase, the previous occurrence that minimizes the maximum height is chosen. We note that a naïve algorithm runs in $O(n^2)$ time, since computing a phrase of length ℓ_j starting at position b_j can be done by at most ℓ_j symbol comparisons for each of the $b_j - 1 = O(n)$ candidate previous positions, which sums up to $O(n^2)$ since $\sum_j \ell_j = n$. Lipták et al. propose an algorithm for which they show an upper bound of $O(z'n^2 \log n) = O(n^3 \log n)$ time. We show here that LZHB3 and LZHB4 can be modified to support the greedier heuristic in total $O(n \log \sigma + occ)$ time, where $occ = O(z'n)$ is the total number of previous occurrences of all phrases.

Suppose, after determining the length of the phrase as described in Sections 4.3 and 4.4, we are able to obtain all previous occurrences of the phrase. The array H is not static, but only append operations are performed on it, and so a range maximum query data structure for H can be maintained in amortized $O(1)$ time per update and query [8, 23]. Using this, the maximum height of a given occurrence can be checked in amortized $O(1)$ time, and the occurrence giving the smallest maximum height can be found in additional $O(n + occ)$ time.

All previous occurrences of a phrase can be obtained in additional $O(n + occ)$ total time using standard techniques on the suffix tree. Recall that when computing the jth phrase, our algorithm traverses the suffix tree for finding a longest non-self-referencing occurrence that satisfies the height constraint. The leaves below the reached position contain the occurrences of the phrase, and since any non-leaf node of a suffix tree has at least two children, these leaves can be found in time linear in their number, thus in additional $O(occ)$ total time. All self-referencing occurrences can be found in $O(\ell_j)$ time as described in Section 4.3. A minor detail we have skipped is that, since we only have the implicit suffix tree being built via Ukkonen's online construction algorithm, the suffix tree does not have leaves corresponding to suffixes that have a previous occurrence. This can be dealt with as follows. Since Ukkonen's algorithm maintains the longest repeating suffix of the current text as the *active point* in the suffix tree, we can also maintain some previous occurrence $T[u..v]$ of it. Since any previous occurrence of the phrase contained in the longest repeating suffix can be mapped to an occurrence in $T[u..v]$, we can, given all the occurrences of the phrase in $T[u..v]$ for which there is a corresponding leaf in the suffix tree, find all occurrences in the longest repeating suffix in $O(n + occ)$ total time.

4.6 Implementation using suffix arrays

The *suffix array* [15] of a string T of length n, is an array $SA[1..n]$ of integers such that $T[SA[i]..n]$ is the ith lexicographically smallest suffix of T.

Suffix arrays are well known as a lightweight alternative to suffix trees. While it is not difficult to use suffix arrays in place of suffix trees for static strings [1] by mapping nodes of the suffix tree to ranges in the suffix array, it is not straightforward to do this for LZHB3 and LZHB4, since the algorithms work in an online manner: the string T' for which the suffix tree is maintained is determined during the computation, and is not known in advance. Here, we show how to simulate the algorithm on the suffix tree for T' by using the suffix array SA of T, in amortized $O(\log n)$ time per suffix tree operation, thus obtaining an $O(n \log n)$ time algorithm for LZHB3 and LZHB4. The running times of the greedier versions can be similarly bounded by $O(n \log n + occ \log n) = O(n \log n + z'n \log n) = O(n^2 \log n)$.

In a suffix tree of T, the effect of replacing a symbol $T[i]$ at position i with \$ can be viewed as an operation that truncates the paths of suffixes starting at position $j < i$ for which $T[j..i)$ does not contain \$, to length $i - j$, i.e., to $T[j..i)$. We will maintain an array L of integers, where $L[rank[i]]$ holds the valid length of the suffix starting at position i and $rank[i]$ is the lexicographic rank of the suffix $T[i..n]$, i.e., $SA[rank[i]] = i$. All values are initially set to 0, indicating that the suffix has not yet been inserted into the suffix tree. During the online construction of the suffix tree of T', when the suffix $T[i..n]$ is added to the suffix tree, we set $L[rank[i]] = n - i + 1$. When a \$ symbol is appended, we modify the values in L of relevant suffixes. Since the value of each position in L is modified at most twice (from 0 to $n - i + 1$ and then to the length up to the next \$), the total number of updates on L is at most $2n$.

The traversal on the suffix tree can then be simulated by a standard search for finding the lexicographic range in the suffix array of suffixes that are prefixed by the considered substring. This range can be computed in $O(\log n)$ time per symbol by a simple binary search. In order to make sure we do not traverse truncated suffixes, we allow the traversal if and only if there exists at least one suffix in the lexicographic range whose valid length is at least the length of the substring being traversed. This can be checked in $O(\log n)$ time by using a segment tree [3, 4] for representing L, which allows updates and range maximum queries on L in $O(\log n)$ time. All occurrences, i.e., suffixes in the range for which the value in L is at least

the length of the substring, can be enumerated in $O(\log n)$ time per occurrence, by a common technique [16] that calls range maximum queries recursively on sub-ranges excluding the maximum value, until all ranges only contain values less than the desired length.

A minor difference with the suffix tree version is that because we are able to insert the whole suffix starting at a given position rather than just a single symbol at the position, the algorithm will naturally handle self-referencing occurrences, and they no longer require special care. Also, while retrieving an occurrence is easy, we note that we can no longer retrieve the *leftmost* occurrence for the greedy versions in the same time bound - the chosen occurrence will be a prefix of the suffix with longest valid length.

5 Height bounded LZ-like encodings as repetitiveness measures

Here, we consider the strength of height bounded encodings as repetitiveness measures [18]. Denote by $\hat{z}_{HB(h)}$, the optimal (smallest) LZHB encoding whose height is at most h. It easily follows that $\hat{z}_{HB(h)}$ is monotonically non-increasing in h, i.e., for any $h \leq h'$, it holds that $\hat{z}_{HB(h)} \geq \hat{z}_{HB(h')}$. We will also denote by $\hat{\tilde{z}}_{HB(h)}$, the optimal modified LZ-like encoding whose height is at most h.

A first, and obvious, observation is that if the height is allowed to grow to n, then the size of the height-bounded encoding and the LZ parsing are equivalent.

▶ **Observation 7.** $\hat{z}_{HB(n)} = z$.

It is also interesting to note height-bounded and run-length encodings can be related:

▶ **Observation 8.** $\hat{\tilde{z}}_{HB(0)}$ *is equivalent to the run-length encoding.*

The following relation between $\hat{z}_{HB(h)}$ and the smallest size $\hat{g}_{\mathrm{rl}}$ of RLSLPs (SLPs with run-length rules) and $\hat{g}_{it(d)}$ of ISLPs [20] can be shown.

▶ **Theorem 9.** *There exists a constant c such that $\hat{z}_{HB(c\log n)} = O(\hat{g}_{\mathrm{rl}})$ holds.*

▶ **Theorem 10.** *There exist a family of strings such that for some constant c, $\hat{z}_{HB(c\log n)} = o(\hat{g}_{\mathrm{rl}})$ and $\hat{z}_{HB(c\log n)} = o(\hat{g}_{it(d)})$.*

When there is no height constraint, LZHB3 is equivalent to LZ77 and thus gives an optimal LZ-like parsing. For modified LZ-like encodings with no height contraints, LZHB4 can be worse than optimal. For example, for the string `abaxabcdababca`, greedy gives `a|b|a|x|ab|c|d|abab|c|a` while `a|b|a|x|ab|c|d|ab|abca` is a slightly smaller parsing. However, we can show that the greedy algorithm has an approximation ratio of at most 2.

▶ **Theorem 11.** *Let $\tilde{z}$ denote the size of the modified LZ-like encoding generated by LZHB4 with no height constraints. We also denote $\hat{\tilde{z}}_{HB(n)} = \hat{\tilde{z}}$. Then, $\hat{\tilde{z}} \leq \tilde{z} \leq z \leq 2\hat{\tilde{z}}$.*

6 Computational Experiments

We have developed prototype implementations of our algorithms described above, in C++. Since Lipták et al. [14] have already demonstrated the effectiveness and superiority of greedy-BATLZ and greedier-BATLZ (which correspond to LZHB3 and the greedier LZHB3) compared to the baselines and that the height can be reduced without increasing the size of the parse too much, our experiments here focus on the performance of our algorithms

LZHB3 and LZHB4 in comparison with greedy-BATLZ and greedier-BATLZ. For the BAT-LZ variants, we use the code provided by Lipták et al.[3] with slight modifications to fit our test environment. Our implementations for LZHB3 and LZHB4 are available at `https://github.com/dscalgo/lzhb`.

Experiments were conducted on a system with dual AMD EPYC 9654 2.4GHz processors and 768GB RAM running RedHat Linux 8. We use the *Real* subset of the Pizza & Chili repetitive corpus[4]. Due to space constraints, only results for `cere`, `influenza`, and `world_leaders` are shown.

Figure 2 shows the throughput for various height constraints of the BAT-LZ variants as well as LZHB3, LZHB3SA, LZHB4, and LZHB4SA, and their greedier versions, where the suffix SA denotes the suffix array implementation. Although greedy-BATLZ is fast when there is no height constraint, its performance seems to diminish rapidly when a height constraint is set. Interestingly, greedier-BATLZ seemed to run faster than greedy-BATLZ. We observe that all versions of our LZHB3 and LZHB4 implementation outperform greedy-BATLZ and greedier-BATLZ by a large margin. For the greedier variants, the throughput decreases as the height constraint decreases. This is because stricter height constraints generally lead to shorter phrase lengths and therefore increases z' and occ. This becomes more apparent in the suffix array versions, perhaps because the n in the $O(\log n)$ factor for finding the suffix array range is actually the size of the range considered, which becomes larger for shorter pharses.

Figure 3 shows the memory usage of the above mentioned implementations. The memory usage for LZHB3 and LZHB4 (the suffix tree versions) are smallest for small heights, because smaller height constraints imply that many paths in the suffix tree become truncated. On the other hand, the memory usage is the highest for larger heights. Memory usage for the other versions is unaffected by the height constraint. LZHB3SA and LZHB4SA are always superior to greedy-BATLZ, and the same can be said for the greedier versions and their counterparts.

In summary, our suffix array implementations are always faster and use less memory than their BATLZ counterparts, with the exception of greedy-BATLZ with no height constraints.

Figure 4 shows the sizes of the phrases of LZHB3, LZHB4 and their greedier versions Although a phrase of the modified LZ-like encoding is slightly larger than the traditional LZ-like encoding since it includes the period, we can see that a much smaller encoding with the same height, which can more than compensate for this increase, can be obtained in some cases. This seemed to be prominent for cere, para, and Escherichia_Coli, which are all DNA.

7 Discussion

We have introduced new variants of LZ-like encodings that directly focus on supporting random access to the underlying data. We also proposed a modified LZ-like encoding, which naturally connects the run-length encoding ($h = 0$), and LZ77 ($h = n$). We described linear time algorithms for several greedy variants of such encodings, as well as how to adapt them to support the greedier heuristic. We showed that our algorithms are faster both theoretically and practically, compared to contemporaneously proposed algorithms by Lipták et al. [14]. We also showed some relations between height-bounded encodings and existing repetitiveness measures. In particular, we have shown that the optimal LZ-like encoding with a height constraint of $O(\log n)$, is one of the asymptotically smallest repetitiveness measures achieving fast (i.e., $O(\text{polylog}(n))$-time) access.

[3] Available at: `https://github.com/fmasillo/BAT-LZ`.
[4] `https://pizzachili.dcc.uchile.cl/repcorpus.html`

There are numerous avenues future work could take. An obvious next step is to engineer actual random-access data structures from height-bounded encodings, which our experiments presented here indicate should be competitive with current state-of-the-art methods. Also, other heuristics for reducing the height while retaining the small size should be explored.

From a theoretical perspective, a natural open problem is whether we can remove the $\log \sigma$ factor from the running times of LZHB3 and LZHB4. The biggest open problem we leave is whether better bounds on the size of height-bounded encodings can be established. Cicalese and Ugazio show that $\hat{z}_{HB(h)} = O(z)$ [5] cannot be achieved for costant h. We note that since there exist linear sized data structures that can answer predecessor queries for a set of values in the range $[1, n]$ in $O(\log \log n)$ time [27], even better lower bounds on the height for which $\hat{z}_{HB(h)} = O(z)$ can be shown by using access time lower bound results of Verbin and Yu [25]. However, these do not rule out $O(\text{polylog}(n))$ height. Can we *balance* the height (achieve $O(\text{polylog}(n))$ height) of an LZ-like encoding or a modified LZ-like encoding, by only increasing the size by a constant factor?

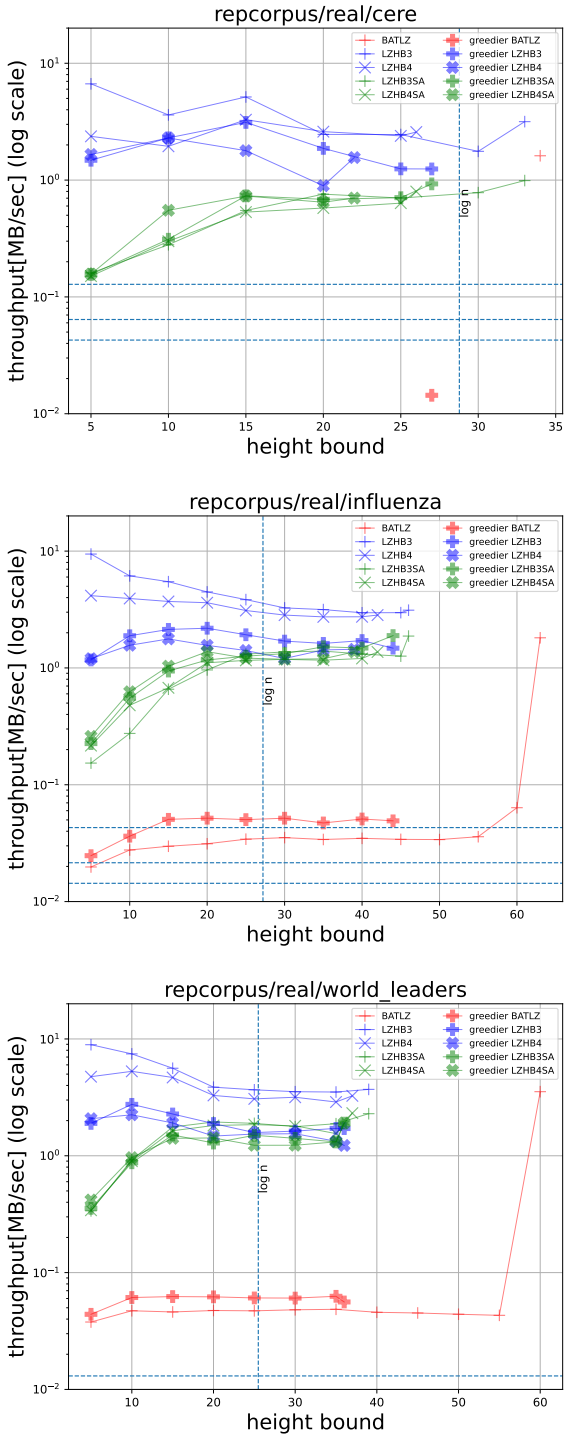

Figure 2 Throughput of BAT-LZ and LZHB variants. For each algorithm, the largest height is the height of the parsing obtained with no height constraint. Results for smaller height constraints are computed for intervals of 5. Missing plots indicate that the computation exceeded 3 hours, which only happened for the BAT-LZ variants, The dotted horizontal lines respectively correspond to the throughput for processing the file in 1,2,3 hours, except for `world_leaders` where only the line for 1 hour is drawn.

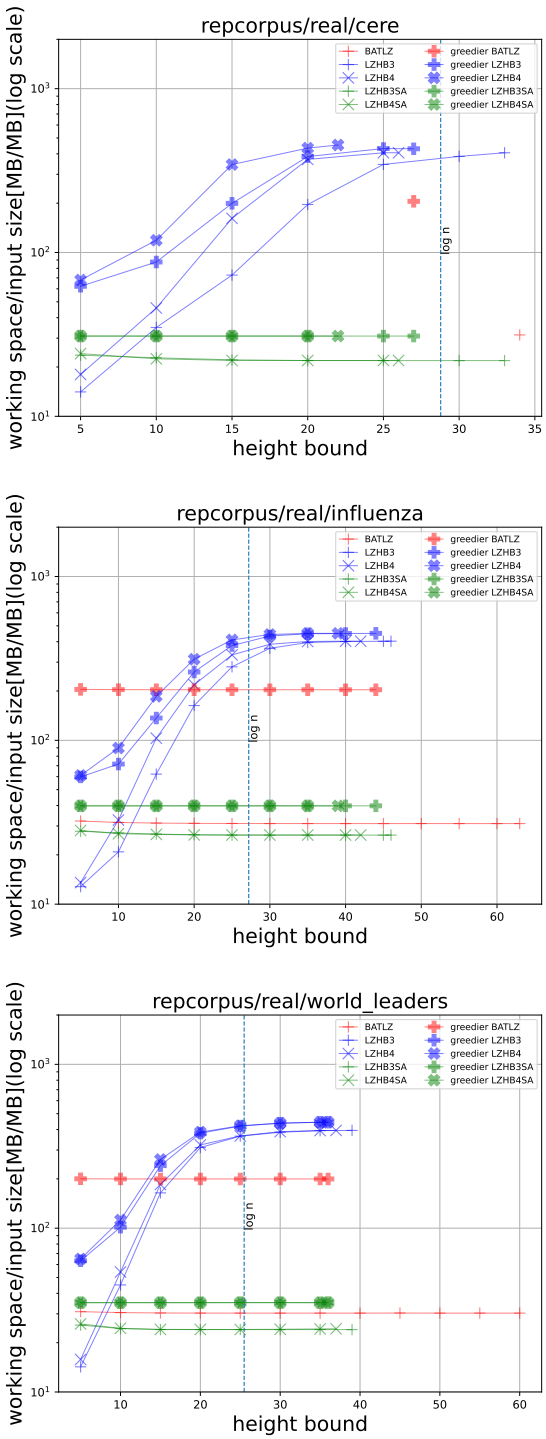

Figure 3 Memory usage per input size of BAT-LZ and LZHB variants measured by `getrusage`. For each algorithm, the largest height is the height of the parsing obtained with no height constraint. The results for smaller height constraints are computed for intervals of 5. Missing plots indicate that the computation exceeded 3 hours.

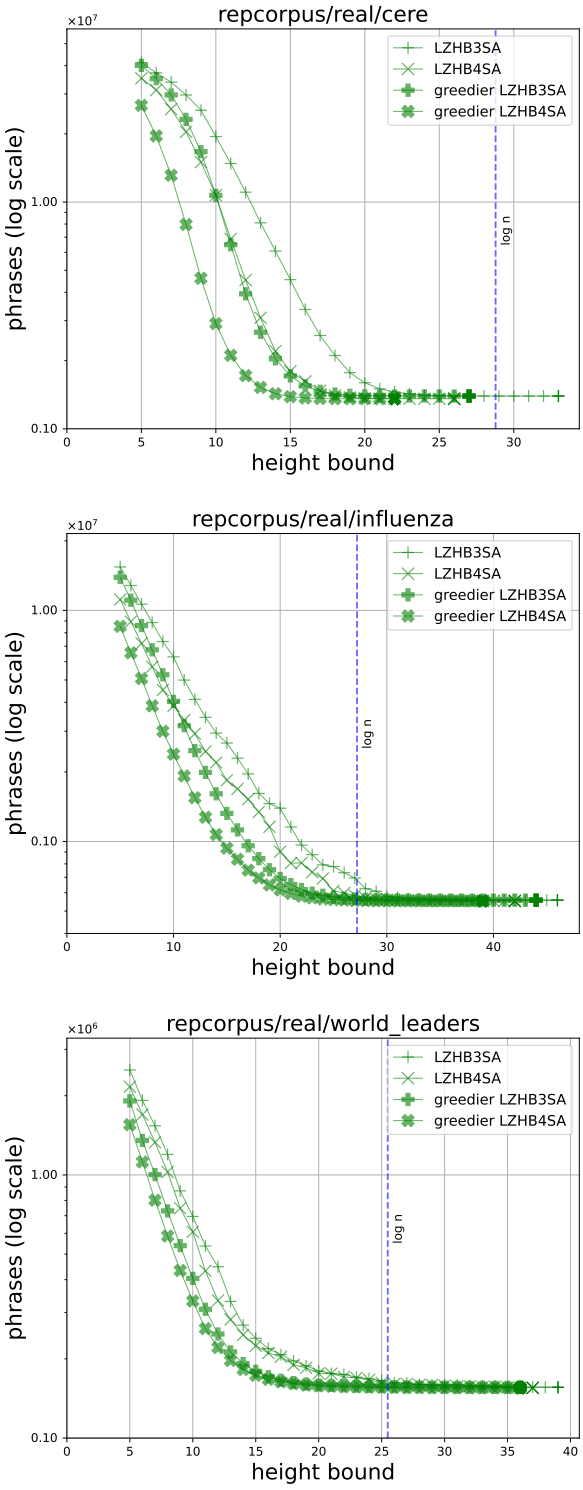

Figure 4 # of phrases of LZHB3, LZHB4 and their greedier variants calculated by the SA algorithms.

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004. doi:10.1016/S1570-8667(03)00065-0.

2 Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman. Weighted ancestors in suffix trees revisited. In Pawel Gawrychowski and Tatiana Starikovskaya, editors, *32nd Annual Symposium on Combinatorial Pattern Matching, CPM 2021, July 5-7, 2021, Wroclaw, Poland*, volume 191 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CPM.2021.8.

3 Jon Louis Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Computers*, 29(7):571–577, 1980. doi:10.1109/TC.1980.1675628.

4 Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988. doi:10.1137/0217026.

5 Ferdinando Cicalese and Francesca Ugazio. On the complexity and approximability of bounded access Lempel Ziv coding. *CoRR*, abs/2403.15871, 2024. doi:10.48550/arXiv.2403.15871.

6 Maxime Crochemore, Lucian Ilie, Costas S. Iliopoulos, Marcin Kubica, Wojciech Rytter, and Tomasz Walen. Computing the longest previous factor. *Eur. J. Comb.*, 34(1):15–26, 2013. doi:10.1016/J.EJC.2012.07.011.

7 Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 137–143. IEEE Computer Society, 1997. doi:10.1109/SFCS.1997.646102.

8 Johannes Fischer. Inducing the lcp-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures - 12th International Symposium, WADS 2011, New York, NY, USA, August 15-17, 2011. Proceedings*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer, 2011. doi:10.1007/978-3-642-22300-6_32.

9 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997. doi:10.1017/CBO9780511574931.

10 Dominik Kempa and Barna Saha. An upper bound and linear-space queries on the LZ-end parsing. In Joseph (Seffi) Naor and Niv Buchbinder, editors, *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, pages 2847–2866. SIAM, 2022. doi:10.1137/1.9781611977073.111.

11 Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977. doi:10.1137/0206024.

12 Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In James A. Storer and Michael W. Marcellin, editors, *2010 Data Compression Conference (DCC 2010), 24-26 March 2010, Snowbird, UT, USA*, pages 239–248. IEEE Computer Society, 2010. doi:10.1109/DCC.2010.29.

13 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/J.TCS.2012.02.006.

14 Zsuzsanna Lipták, Francesco Masillo, and Gonzalo Navarro. BAT-LZ out of hell. In Shunsuke Inenaga and Simon J. Puglisi, editors, *35th Annual Symposium on Combinatorial Pattern Matching, CPM 2024, June 25-27, 2024, Fukuoka, Japan*, volume 296 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.CPM.2024.21.

15 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 319–327. SIAM, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320218.

16 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In David Eppstein, editor, *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms,*

January 6-8, 2002, San Francisco, CA, USA, pages 657–666. ACM/SIAM, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

17 Gonzalo Navarro. *Compact Data Structures - A Practical Approach*. Cambridge University Press, 2016.

18 Gonzalo Navarro. Indexing highly repetitive string collections, part I: repetitiveness measures. *ACM Comput. Surv.*, 54(2):29:1–29:31, 2022. doi:10.1145/3434399.

19 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5):105:1–105:35, 2021. doi:10.1145/3409371.

20 Gonzalo Navarro and Cristian Urbina. Iterated straight-line programs. In José A. Soto and Andreas Wiese, editors, *LATIN 2024: Theoretical Informatics - 16th Latin American Symposium, Puerto Varas, Chile, March 18-22, 2024, Proceedings, Part I*, volume 14578 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2024. doi:10.1007/978-3-031-55598-5_5.

21 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Fully dynamic data structure for LCE queries in compressed space. In *41st International Symposium on Mathematical Foundations of Computer Science, MFCS 2016, August 22-26, 2016 - Kraków, Poland*, volume 58 of *LIPIcs*, pages 72:1–72:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

22 Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*, volume 1443 of *Lecture Notes in Computer Science*, pages 731–742. Springer, 1998.

23 Yohei Ueki, Diptarama, Masatoshi Kurihara, Yoshiaki Matsuoka, Kazuyuki Narisawa, Ryo Yoshinaka, Hideo Bannai, Shunsuke Inenaga, and Ayumi Shinohara. Longest common subsequence in at least k length order-isomorphic substrings. In Bernhard Steffen, Christel Baier, Mark van den Brand, Johann Eder, Mike Hinchey, and Tiziana Margaria, editors, *SOFSEM 2017: Theory and Practice of Computer Science - 43rd International Conference on Current Trends in Theory and Practice of Computer Science, Limerick, Ireland, January 16-20, 2017, Proceedings*, volume 10139 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2017. doi:10.1007/978-3-319-51963-0_28.

24 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

25 Elad Verbin and Wei Yu. Data structure lower bounds on random access to grammar-compressed strings. In Johannes Fischer and Peter Sanders, editors, *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, volume 7922 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2013. doi:10.1007/978-3-642-38905-4_24.

26 Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*, pages 1–11. IEEE Computer Society, 1973.

27 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

28 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.