

HOLZ: High-Order Entropy Encoding of Lempel–Ziv Factor Distances

Dominik Köppl^{*}, Gonzalo Navarro[†], and Nicola Prezza[‡]

[*] M&D Data Center	[†] Dept. of Computer Science	[‡] DAIS
TMDU	University of Chile	Ca' Foscari University
Tokyo, Japan	Santiago, Chile	Venice, Italy
koepp1.dsc@tmd.ac.jp	gnavarro@dcc.uchile.cl	nicola.prezza@unive.it

Abstract

We propose a new representation of the offsets of the Lempel–Ziv (LZ) factorization based on the co-lexicographic order of the processed prefixes. The selected offsets tend to approach the k -th order empirical entropy. Our evaluations show that this choice of offsets is superior to the rightmost LZ parsing and the bit-optimal LZ parsing on datasets with small high-order entropy.

1 Introduction

The Lempel–Ziv (LZ) factorization [1] is one of the most popular methods for lossless data compression. It builds on the idea of *factorization*, that is, splitting the text T into *factors*, each being the longest string that appears before in T , and replacing each factor by a reference to its preceding occurrence (called its *source*).

Most popular compression schemes such as `zip` or `gzip` use a variant called LZ77 [2], which finds sources only within a sliding window w . Though this restriction simplifies compression and encoding, it misses repetitions with gaps larger than $|w|$, and thus compressing k copies of a sufficiently long text T results in a compressed file being about k times larger than the compressed file of T . Such long-spaced repetitions are common in highly-repetitive datasets like genomic collections of sequences from the same taxonomy group, or from documents managed in a revision control system. Highly-repetitive datasets are among the fastest-growing ones in recent decades, and LZ compression is one of the most effective tools to compress them [3]. This is one of the main reasons why the original LZ factorization (i.e., without a window) moved into the spotlight of recent research.

Although LZ catches even distant repetitions, the actual encoding of the factorization is an issue. Each factor is usually represented by its length and the distance to its source (called its *offset*). While the lengths usually exhibit a geometric distribution favorable for universal encoders, the offsets tend to approach a uniform distribution and their codes are long. Since the sources are not uniquely defined, different tie breaks have been exploited in order to improve compression, notably the *rightmost* parsing (i.e., choosing the closest source) and the *bit-optimal* parsing [4] (i.e., optimizing the size of the encoded file instead of the number of factors).

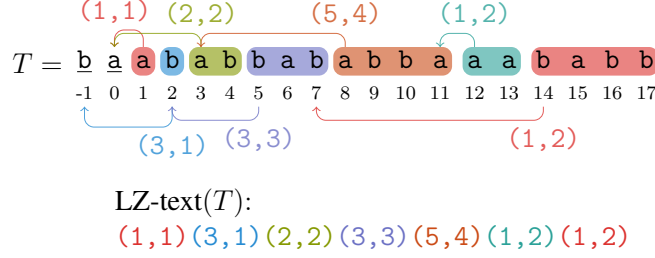


Figure 1: LZ factorization of $T = \text{ababbababbbaaababb}$ and its $\text{LZ-text}(T)$ coding into pairs.

All previous LZ encodings have in common that they encode the *text distance* to the source. In this paper we propose a variant that encodes the distances between the co-lexicographically sorted prefixes of the processed text, and argue that this choice for the offsets approaches the k -th order empirical entropy of the LZ factors. Our experiments show that this encoding is competitive even with the bit-optimal parsing [4], performing particularly well on texts whose high-order entropy is low.

2 LZ Factorization

Let $T[1..n] \in \Sigma^n$ be a text of length n whose characters are drawn from an integer alphabet $\Sigma = [0..\sigma - 1]$ with $\sigma = n^{\mathcal{O}(1)}$. The LZ factorization is a partitioning of the text T into factors $F_1 \cdots F_z$ such that every factor F_x is equal to the longest substring that starts before F_x in the text (its source). Here, we imagine the text being prefixed by the characters of Σ in non-positive positions, i.e. $T[-c] = c$ for each $c \in \Sigma$. Further, we compute the factorization on $T[1..n]$ while taking the prefixes of all suffixes starting in $T[-\sigma..p - 1]$ under consideration for previous occurrences of a factor starting at $T[p]$. In this setting, we always find a factor of length at least one. Hence, each factor F_x can be encoded as a pair $(\text{off}_x^\top, \ell_x)$, where $\ell_x = |F_x|$ is the length of F_x , and off_x^\top is the *textual offset* of F_x , that is, the distance of the starting positions of F_x and its source. Thus, the LZ factorization can be represented as a sequence of these integer pairs of offsets and lengths, which we denote by $\text{LZ-text}(T)$. This representation is not unique with respect to the choice of the offsets in case of a factor having multiple occurrences that start before it. The so-called *rightmost* parsing selects always the smallest possible offset as a tie break.

Example. Let $T = \text{ababbababbbaaababb}$. Then the LZ factorization and its representation as pairs are given in Fig. 1. Remember that we imagine the text as being prefixed by all characters of Σ , being **ba** in this case; we only factorize $T[1..|T|]$, that is, the non-underlined characters of baababbababbbaaababb. \square

With the rightmost parsing it holds that, for a stationary ergodic source, the minimum number of bits required to write the number off_x^\top , which is $\lceil \log_2(\text{off}_x^\top + 1) \rceil$, approaches the zero-order entropy of F_x in T (see [5] or [6, Sec. 13.5.1]). Intuitively, this happens because, being $\Pr(F_x = S)$ the probability that the next LZ factor F_x equals the substring S , on expectation we need to visit $\text{off}_x^\top = 1/\Pr(F_x = S)$ positions back (the log of which is F_x 's entropy) before encountering an occurrence of S . The

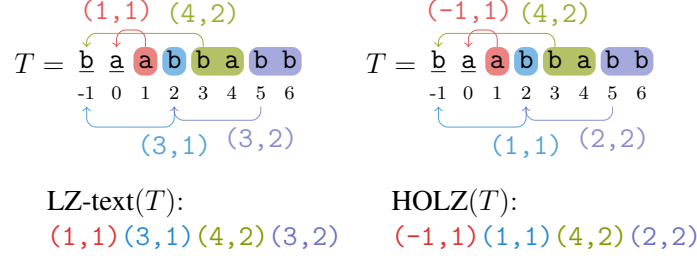


Figure 2: LZ factorization of $T = \text{abbabb}$ with its **LZ-text** (left) and **HOLZ** encoding (right). Since the encodings only differ in the values of their offsets, they are equal in length.

fact that the factors are long ($\Theta(\log_\sigma n)$ characters on average) can be used to show that the size of **LZ-text**(T) approaches the k -th order entropy of T , for small enough $k = o(\log_\sigma n)$. In that light, we aim to reach the high-order entropy of the factors F_x , by means of changing the encoding of the sources.

3 The HOLZ Encoding

We now give a different representation of the sources of the LZ factors, and argue that this representation achieves high-order entropy. Broadly speaking, we compute the offsets in co-lexicographic order¹ of the text's prefixes, rather than in text order. We then compute a new offset off_p instead of the textual offset off_p^T used in **LZ-text**(T).

Definition 1. Let $T_i := T[-\sigma + 1..i]$, for $i \in [-\sigma..n]$, be the prefixes of T , including the empty prefix $T_{-\sigma} = \epsilon$. Assume we have computed the factorization of $T[1..p-1]$ for a text position $p \in [1..n]$ and that a new factor F_x of length ℓ_x starts at $T[p]$.

Let $T_{j_1} \prec T_{j_2} \prec \dots \prec T_{j_{p+\sigma}}$ be the prefixes $T_{-\sigma}, \dots, T_{p-1}$ in co-lexicographic order. Let r_{p-1} be the position of T_{p-1} in that order, that is, $j_{r_{p-1}} = p-1$, and let t_{p-1} be the position closest to r_{p-1} (i.e., minimizing $|r_{p-1} - t_{p-1}|$) satisfying $T[j_{t_{p-1}}+1..j_{t_{p-1}}+\ell_x] = F_x$. Then we define $\text{off}_x := r_{p-1} - t_{p-1}$; note off_x can be negative. We call **HOLZ**(T) the resulting encoding of pairs (off_x, ℓ_x) , where HO stands for high-order entropy.

Example. We present a complete step-by-step factorization of the small sample string $T = \text{abbabb}$. The **HOLZ**(T) factorization is given in Fig. 2, and Table 1 depicts the four factorization steps; remember that we do not factorize the added prefix $T[-\sigma + 1..0]$. A detailed walk-through follows:

1. For the first factor $F_1 = \text{a}$ starting at $p = 1$, the text prefix starting with F_x and being the closest to the r_0 -th co-lexicographically smallest prefix, has rank $t_0 = 3$, thus F_1 's length and offset are $\ell_1 = 1$ and $\text{off}_1 = r_0 - t_0 = 2 - 3 = -1$, respectively. We then represent $F_1 = T[1] = \text{a}$ by the pair $(-1, 1)$.
2. Next, we update the table of the sorted prefixes, obtaining the order shown on the top-right table. The next factor, starting at position $p = 2$, is $F_2 = \text{b}$, so $\ell_2 = 1$ and $\text{off}_2 = r_1 - t_1 = 2 - 1 = 1$. The second pair is thus $(1, 1)$.

¹That is, the lexicographic order of the reversed strings.

Table 1: Step-by-step computation of $\text{HOLZ}(T)$ with $T = \text{abbabb} = F_1 F_2 F_3 F_4$. Underlined characters represent the virtual text prefix containing all alphabet's characters.

	sorted prefixes	text to their right	
$T[-1..-2]$	$= \epsilon$	<u>ba</u> abbabb 1	
$T[-1..0]$	$= \underline{\text{ba}}$	abbabb 2 = r_0	
$T[-1..-1]$	$= \underline{\text{b}}$	<u>a</u> abbabb 3 = t_0	
Computing $F_1 = T[1]$.			

	sorted prefixes	text to their right	
$T[-1..-2]$	$= \epsilon$	<u>ba</u> abbabb 1 = t_1	
$T[-1..1]$	$= \underline{\text{baa}}$	bbabb 2 = r_1	
$T[-1..0]$	$= \underline{\text{ba}}$	abbabb 3	
$T[-1..-1]$	$= \underline{\text{b}}$	<u>a</u> abbabb 4	
Computing $F_2 = T[2]$.			

	sorted prefixes	text to their right	
$T[-1..-2]$	$= \epsilon$	<u>ba</u> abbabb 1 = t_2	
$T[-1..1]$	$= \underline{\text{baa}}$	bbabb 2	
$T[-1..0]$	$= \underline{\text{ba}}$	abbabb 3	
$T[-1..-1]$	$= \underline{\text{b}}$	<u>a</u> abbabb 4	
$T[-1..2]$	$= \underline{\text{baab}}$	babb 5 = r_2	
Computing $F_3 = T[3..4]$.			

	sorted prefixes	text to their right	
$T[-1..-2]$	$= \epsilon$	<u>ba</u> abbabb 1	
$T[-1..1]$	$= \underline{\text{baa}}$	bbabb 2 = t_4	
$T[-1..0]$	$= \underline{\text{ba}}$	abbabb 3	
$T[-1..4]$	$= \underline{\text{baabba}}$	bb 4 = r_4	
$T[-1..-1]$	$= \underline{\text{b}}$	<u>a</u> abbabb 5	
$T[-1..2]$	$= \underline{\text{baab}}$	babb 6	
$T[-1..3]$	$= \underline{\text{baabb}}$	abb 7	
Computing $F_4 = T[5..6]$.			

3. We update the table of the sorted prefixes, obtaining the order shown on the bottom-left table. This time, $p = 3$, $F_3 = \text{ba}$, $\ell_3 = 2$, and $\text{off}_3 = r_2 - t_2 = 5 - 1 = 4$. The third pair is thus $(4, 2)$.

4. We update the table of the sorted prefixes, obtaining the order shown on the bottom-right table; the final pair is $(2, 2)$.

Thus, we obtained $\text{HOLZ}(T) = (-1, 1) (1, 1) (4, 2) (2, 2)$. \square

Towards High-Order Entropy

Only for the purpose of formalizing this idea, let us define a variant of HOLZ , $\text{HOLZ}^k(T)$, which precedes T with a (virtual) de Bruijn sequence of order $k + 1$, so that every string of length $k + 1$ appears in $T[-\sigma^{k+1} - k + 2..0]$ (i.e., classical $\text{HOLZ}(T)$ is $\text{HOLZ}^0(T)$). We modify the LZ factorization so that F_x , starting at $T[p]$ and preceded by the string S_x of length k , will be the longest prefix of $T[p..]$ such that $S_x \cdot F_x$ appears in T starting before position $p - k$. The resulting factorization $T = F_1 F_2 \dots$ has more factors than the LZ factorization, but in exchange, the offsets of the factors F_x are encoded within their k -th order (empirical) entropy. Let $\#S$ be the frequency of substring S in T . Assuming that the occurrences of $S_x F_x$ distribute uniformly among the occurrences of S_x in every prefix of T , the distance $|\text{off}_x|$ between two consecutive sorted prefixes of T suffixed by S_x and followed by F_x is in expectation $\mathbb{E}(|\text{off}_x|) \leq \#S_x / \#S_x F_x$. Then, $\mathbb{E}(\log_2 |\text{off}_x|) \leq \log_2 \mathbb{E}(|\text{off}_x|) \leq \log_2 (\#S_x / \#S_x F_x)$ and the total expected size of the encoded offsets is

$$\mathbb{E} \left(\sum_x \log_2 |\text{off}_x| \right) = \sum_x \mathbb{E}(\log_2 |\text{off}_x|) \leq \sum_x \log_2 \frac{\#S_x}{\#S_x F_x}.$$

In the empirical-entropy sense (i.e., interpreting probabilities as relative frequencies in T), the definition of high-order entropy we can reach is restricted to the factors

we produce. Interpreting conditional probability as following in the text, this is

$$H_k = \sum_x \log_2 \frac{1}{\Pr(F_x|S_x)} = \sum_x \log_2 \frac{\Pr(S_x)}{\Pr(S_x F_x)} = \sum_x \log_2 \frac{\#S_x}{\#S_x F_x}.$$

That is, the expected length of our encoding is bounded by the k -th order empirical entropy of the factors. This is also the k -th order empirical entropy of the text if we assume that the factors start at random text positions.

Recall that, the longer k , the shorter the phrases, so there is an optimum for a likely small value of k . While this optimum may not be reached by HOLZ (which always chooses the longest phrase), it is reached by the bit-optimal variant of HOLZ that we describe in the next section, *simultaneously* for every k .

Our experimental results validate our synthetic analysis, in the sense that **HOLZ**(T) performs better than **LZ-text**(T) on texts with lower k -th order entropy, for small k .

Algorithmic Aspects

For an efficient computation of a factor F_x starting at $T[p]$, we need to maintain the prefixes $T_{-\sigma}, \dots, T_{p-1}$ sorted in co-lexicographic order such that we can quickly find the ranks r_{p-1} of T_{p-1} and the rank t_{p-1} of the starting position of an occurrence of F_x . In what follows, we describe an efficient solution based on dynamic strings.

Our algorithms in this paper work on the word RAM model with a word size of $\Omega(\lg n)$ bits. We use the following data structures: **SA** denotes the suffix array [7] of T , such that $\text{SA}[i]$ stores the starting position of the i -th lexicographically smallest suffix in T . **ISA** is its inverse, $\text{ISA}[\text{SA}[i]] = i$ for all i . The Burrows-Wheeler transform **BWT** of T is defined by $\text{BWT}[i] = T[\text{SA}[i] - 1]$ for $\text{SA}[i] > 1$ and $\text{BWT}[i] = T[n]$ for $\text{SA}[i] = 1$.

We maintain a dynamic wavelet tree on the reverse of $T[-\sigma..p-1]$. This framework was already used [8] to compute **LZ-text**(T) with the dynamic wavelet tree. If we represent this wavelet tree with the dynamic string of [9], their algorithm runs in $\mathcal{O}(n \log n / \log \log n)$ time using $nH_k + o(n \log \sigma)$ bits.

Theorem 3.1 ([9]). *A dynamic string $S[1..n]$ over an alphabet with size σ can be represented in $nH_k + o(n \log \sigma)$ bits while supporting the queries access, rank, and select, as well as insertions or deletions of characters, in $\mathcal{O}(\log / \log \log n)$ time.*

The text is not counted in this space; it can be given online in a streaming fashion. The main idea can be described as follows: Let $R_i := T_i^R\$$. Given that we want to compute a factor F_x starting at text position $T[p]$, we interpret $T[p..]$ as the reverse of a pattern P that we want to search in $\text{BWT}_{R_{p-1}}$ with the backward search steps of the FM-index [10]. The backward search takes the last character of P being $T[p]$ as the initial search range in $\text{BWT}_{R_{p-1}}$, updates $\text{BWT}_{R_{p-1}}$ with the next character $T[p]$ to BWT_{R_p} and recurses on computing the range for $P[1..2] = T[p..p+1]^R$. The recursion ends at the step before the range becomes empty. In that case, all **BWT** positions in the range correspond to occurrences of the factor F_x starting before F_x in T . We choose the one being the closest (in co-lexicographic order) to the co-lexicographic rank of the current text prefix, and compute with $|F_x|$ forward traversals in the **BWT** the original position in $\text{BWT}_{R_{p-1}}$, which gives us t'_{p-1} . The position

r'_{p-1} is where $\$$ is stored in $\text{BWT}_{R_{p-1}}$. We wrote t'_{p-1} and r'_{p-1} instead of t_{p-1} and r_{p-1} since these positions are based on $\text{BWT}_{R_{p+|F_x|-1}}$ instead of $\text{BWT}_{R_{p-1}}$. We can calculate the actual offset $r_{p-1} - t_{p-1}$ by $r'_{p-1} - t'_{p-1}$ if we know the number of newly inserted positions between r'_{p-1} and t'_{p-1} during the steps when we turned $\text{BWT}_{R_{p-1}}$ into $\text{BWT}_{R_{p+|F_x|-1}}$. For that, we additionally maintain a dynamic bit vector that marks the entries we inserted into $\text{BWT}_{R_{p-1}}$ to obtain $\text{BWT}_{R_{p+|F_x|-1}}$ (alternatively, we can store the positions in a list). We conclude that we can compute **HOLZ** in $n(1 + H_k) + o(n \log \sigma)$ bits of space and $\mathcal{O}(n \log n / \log \log n)$ time by using the data structure of Theorem 3.1.

4 The Bit-Optimal HOLZ

Ferragina et al. [4] studied a generalization of the Lempel–Ziv parsing in the sense that they considered for each text position all possible factor candidates (not just the longest ones), optimizing for the representation minimizing a fixed encoding of the integers (e.g. Elias- δ). In other words, in their setting we are free to choose both the offset and factor lengths, thus effectively choosing among all possible unidirectional macro-schemes [11]. Within their framework, LZ can be understood as a greedy factorization that locally always chooses the longest factor among all candidates. This factorization is optimal with respect to the number of computed factors, but not when measuring the *bit-size* of the factors compressed by the chosen encoding for the integers, in general. Given a universal code **enc** for the integers, a *bit-optimal* parsing has the least number of bits among all unidirectional parsings using **enc** to encode their pairs of lengths and offsets. In the setting of textual offsets, [4] proposed an algorithm computing the bit-optimal LZ factorization in $\mathcal{O}(n \lg n)$ time with $\mathcal{O}(n)$ words of space, provided that the code **enc** transforms an integer in the range $[1..n]$ to a bit string of length $\mathcal{O}(\lg n)$. In the following, we take this restriction of **enc** as granted, as it reflects common encoders like Elias- γ .

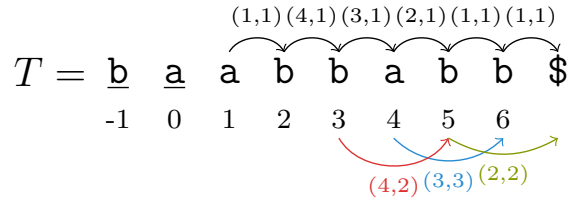


Figure 3: Graph of the factor candidates for $\text{LZ-text}(T)$. Every path from node 1 to node $n+1$ gives us a sequence of pairs that can be used alternatively to $\text{LZ-text}(T)$, which is obtained by always taking the locally longest edge. Although it is guaranteed that the path for $\text{LZ-text}(T)$ has the least number of edges, the compressed representation of the edge labels does not lead to the best compression in general. Using Elias- γ as our encoder **enc** with $|\text{enc}(x)| = 1$ for $x = 1$, $|\text{enc}(x)| = 2$ for $x \in \{2, 3\}$, and $|\text{enc}(x)| = 3$ for $x \in [4..7]$, the compressed size of $\text{LZ-text}(T)$ taking the red and the green arc is $1+1+3+1+3+2+2+2 = 15$ bits. If we exchange the red and green arc with one blue arc and two black arcs, we obtain $1+1+3+1+1+1+1+2+2+1+1 = 14$ bits.

Factor Graph

All possible unidirectional parsings using textual offsets can be represented by the following weighted directed acyclic graph: This graph has $n + 1$ nodes, where the i -th node v_i corresponds to text position $i \in [1..n]$, and the $(n + 1)$ -st node corresponds to the end of the text, which we can symbolize by adding an artificial character $\$$ to T that is omitted in the factorization, see Fig. 3. An arc connecting node v_i with a node $v_{i+\ell-1}$ corresponds to a candidate factor starting at position i with length ℓ . A *candidate factor* is a pair (j, ℓ) of position and length with $j < i$ and $T[j..j+\ell-1] = T[i..i+\ell-1]$. The weight of the arc $(v_i, v_{i+\ell-1})$ corresponding to (j, ℓ) is the cost of encoding its respective factor, $|\text{enc}(j)| + |\text{enc}(\ell)|$ if $|\text{enc}(x)|$ denotes the length of the binary output of $\text{enc}(x)$. By construction, there are no arcs from a node i to a node j with $j < i$, but there is always at least one arc from node i to a node j with $i < j$. That is because we have at least one factor candidate starting at position i since we can always refer to a character in $T[-\sigma..0]$. Hence, node 1 is connected with node $n + 1$. Let the *length* of an arc be the length of its corresponding factor, that is, if an arc connects node i with node j , then its length is $j - i$. We can then obtain the classic LZ factorization by following the maximum-length arcs starting from node 1, and we obtain the bit-optimal parsing by computing the (weighted) shortest path.

While we have n nodes, we can have $\mathcal{O}(n^2)$ arcs since the set of candidate factors for a text position i is $\{(j, \ell) : j < i \text{ and } T[j..j + \ell - 1] = T[i..i + \ell - 1]\}$, and this set can have a cardinality of $\mathcal{O}(n)$. Theorem [4, Thm. 5.3] solves this issue by showing that it suffices to only consider so-called maximal arcs. An arc (v_i, v_j) corresponding to a factor candidate (off, ℓ) is called *maximal* [12, Def. 3.4] if either there is no arc (v_i, v_{j+1}) , or such an arc corresponds to a factor candidate (off', ℓ') with $\text{enc}(\text{off}) + \text{enc}(\ell) < \text{enc}(\text{off}') + \text{enc}(\ell')$. Moreover, there exists a shortest path from v_1 to v_n that only consists of maximal arcs. Since $|\text{enc}(\cdot)| \in \mathcal{O}(\lg n)$, we can divide all maximal arcs spawning from a node v_i into $\mathcal{O}(\lg n)$ cost classes such that each cost class is either empty or has exactly one maximal arc. Hence, the pruned graph has just $\mathcal{O}(n \lg n)$ arcs. Finding the shortest path can be done with Dijkstra's algorithm running in $\mathcal{O}(n \lg n)$ time with Fibonacci heaps. The computed shortest path is a sequence of factors that we encode in our final output. Storing the complete graph would take $\mathcal{O}(n \lg n)$ words. To get the space down to $\mathcal{O}(n)$ words, the idea [4, Cor. 5.5] is to compute the arcs online by partitioning, for each cost class $k \in [1.. \mathcal{O}(\lg n)]$, the text into blocks, and process each of the blocks in batch.

The Bit-Optimal HOLZ

By exchanging the definition of *off* in the factor candidates, we compute a variation of **HOLZ** that is bit-optimal. The major problem is finding the maximal arcs efficiently. Like before, we maintain the dynamic BWT of the reversed processed text. But now we also maintain a dynamic wavelet tree mapping suffix ranks of the processed reversed text to suffix ranks of T . Then we can compute for each cost class of arcs spawning from v_i the maximal arc by searching the suffix ranks closest to the suffix rank of $T[i..]$ within the interval of suffix ranks of the reversed processed text.

In what follows, we explain our algorithm computing the factor candidates corre-

sponding to all maximal arcs. Let again $T[-\sigma + 1..n]$ be the text with all distinct characters prepended, and let T^R denote its reverse. In a preprocessing step, we build SA and ISA on $T[-\sigma + 1..n]$. Like before, we scan the text $T[1..n]$ from left to right. Let p be the text position where are currently processing, such that $T[1..p-1]$ has already been processed. Let $R_{p-1} := T_{p-1}^R \$$ denote the string whose BWT we maintain in $\text{BWT}_{R_{p-1}}$. Further, let **DyWa** be a dynamic wavelet tree mapping $\text{ISA}_{R_{p-1}}[i]$ to $\text{ISA}_T[i]$ for each $i \in [1..|R_{p-1}|]$. Before starting the factorization, we index/process $T[-\sigma - 1..0]^R$ with BWT and DyWa. Since $\text{ISA}_{R_{p-1}}$ is not necessarily a prefix of ISA_{R_p} , for adding a new entry to **DyWa** when processing a position $p \in [-\sigma + 1..n]$, we first need to know $\text{ISA}_{R_p}[1]$, i.e., the rank of the suffix in R_p corresponding to $T[p]$ in $\text{BWT}_{R_{p-1}}$. Luckily, this is given by the position of $\$$ in $\text{BWT}_{R_{p-1}}$. Let $p_\$$ denote this position in $\text{BWT}_{R_{p-1}}$, and suppose that we have processed $T[1..p-1]$. Now, for each of the ranges $I_1 = [1..p_\$ - 1]$ and $I_2 = [p_\$ + 1..p - 1]$, let predRSA_j^p and succRSA_j^p be values in the range I_j that are mapped via **DyWa** to the smallest value larger than $\text{ISA}[p]$ and the largest value smaller than $\text{ISA}[p]$, respectively. Let us call these mapped values predSA_j^p and succSA_j^p . Let maxSA_j^p be the one that has a longer LCE value with $\text{ISA}[p]$, i.e., we compare $\text{lce}(T[\text{SA}[\text{predSA}_j^p]..], T[p..])$ with $\text{lce}(T[\text{SA}[\text{succSA}_j^p]..], T[p..])$. Finally, let maxSA^p be the one among maxSA_1^p and maxSA_2^p having the largest LCE ℓ with p , and let maxRSA^p be its corresponding position in $\text{BWT}_{R_{p-1}}$. This already defines the factor candidate with offset $p_\$ - \text{maxRSA}^p$ and the longest length ℓ among all factor candidates starting at $T[p]$. Next, we compute the factor candidates with smaller lengths but less costly offsets. For that, we partition $[\text{maxRSA}_1^p..p_\$ - 1]$ into cost classes, i.e., intervals of ranks whose differences to $p_\$$ need the same amount of bits when compressed via a universal coder. This gives $\mathcal{O}(\lg n)$ intervals, and for each interval we perform the same query as above to retrieve a maxSA^p value with $\text{lce}(T[\text{SA}[\text{maxSA}^p]..], T[p..]) \leq \text{lce}(T[\text{SA}[\text{maxSA}_1^p]..], T[p..])$. We start with the interval with the shortest costs, and keep track of the maximal LCE value computed up so far. For each candidate interval, if its computed maximal LCE value is not larger than the already computed LCE value, then we discard it since it would produce a factor with the same length but a higher cost for the offset. We cut I_1 at maxRSA_1^p since this gives us the maximum LCE value in the entire range, so going further does not help us in discovering an even longer candidate factor. We process I_2 with maxRSA_2^p symmetrically. In total, we obtain for each text position $\mathcal{O}(\lg n)$ factor candidates, which we collect in a list per text position; the summed size of these lists is $\mathcal{O}((\sigma + n) \lg n)$. Each pair (ℓ, off) in the list of position p consists of its length ℓ and the offset off being the difference between $p_\$$ and its respective position j in $\text{BWT}_{R_{p-1}}$. The cost for encoding this pair is $|\text{enc}(\ell)| + |\text{enc}(\text{off})|$ incremented by one for a bit storing whether $j > p_\$$ or not. The computed pairs determine the edges of the weighted directed acyclic graph explained above.

Complexities The wavelet tree **DyWa** stores $\mathcal{O}(n)$ integers in the range $[1..n]$ dynamically. For each of the $\mathcal{O}(\lg n)$ levels, a rank or select query can be performed in $\mathcal{O}(\lg n)$ time, and thus updates or queries like range queries take $\mathcal{O}(\lg^2 n)$ time. For each text position, we perform one update and $\mathcal{O}(\lg n)$ range queries, accumulating to

$\mathcal{O}(n \lg^3 n)$ time overall for the operations on **DyWa**. This is also the bottleneck of our bit-optimal algorithm using the **HOLZ** encoding. The wavelet tree representing the dynamic BWT of the reversed processed text works exactly as the previous algorithm in Section 3, and the produced graph needs $\mathcal{O}(n \lg n)$ time for processing. Summing up, our algorithm to compute the bit-optimal **HOLZ** encoding runs in $\mathcal{O}(n \lg^3 n)$ time and uses $\mathcal{O}(n \lg n)$ words of space. The space could be improved to $\mathcal{O}(n)$ words using the same techniques discussed in [4].

Decompression The decompression works in both variants (**HOLZ** or its bit-optimal variant) in the same way. We maintain a dynamic BWT on the reversed processed text; When processing a factor F starting at text position p and encoded with pair (off, ℓ) , we know that BWT position $t_{p-1} = p_{\$} - \text{off}$ corresponds to the starting text position of the factor’s source. It is then sufficient to extract ℓ characters from that position, by navigating the BWT using LF queries. At each query, we also extend the BWT with the new extracted character to take into account the possibility that the source overlaps F . Overall, using the dynamic string data structure of Theorem 3.1, the decompression algorithm runs in $\mathcal{O}(n \log n / \log \log n)$ time and uses $nH_k + o(n \log \sigma)$ bits of space (excluding the input, which however can be streamed).

5 Experiments

For our experiments, we focus on the Canterbury and on the Pizza&Chili corpus. For the latter, we took 20 MB prefixes of the data sets. See Tables 2 and 3 for some characteristics of the used datasets. The datasets **KENNEDY.XLS**, **PTT5**, and **SUM** contain ‘0’ bytes, which is a prohibited value for some used tools like suffix array construction algorithms. In a precomputation step for these files, we escaped each ‘0’ byte with the byte pair ‘254’ ‘1’ and each former occurrence of ‘254’ with the pair ‘254’ ‘254’.

For comparison, we used **LZ-text(T)** and the bit-optimal implementation of [12], referred to as **bitopt** in the following. For the former, we remember that the choice of the offsets can be ambiguous. Here, we select two different codings that break ties in a systematic manner: The rightmost parsing introduced in the introduction, and the output of an algorithm [13] computing LZ with next-smaller value (NSV) and previous-smaller value (PSV) arrays built on the suffix array. The PSV and NSV arrays, **PSV** and **NSV**, store at their i -th entry the largest index j smaller than i (resp. smallest index j larger than i) with $\text{SA}[j] < i$. Given a starting position p of a factor, candidates for the starting positions of its previous occurrences are stored at the entries **NSV**[**ISA**[p]] and **PSV**[**ISA**[p]] of **SA**. We take the one that has the larger LCE with p , say r , and output the offset $p - r$. Although the compressed output is at least that of the rightmost parsing, this algorithm runs in linear time, whereas we are unaware of an algorithm computing the rightmost parsing in linear time – the currently best algorithm needs $\mathcal{O}(n + n \log \sigma / \sqrt{\log n})$ time [14]. We call these two specializations **rightmost** and **nsvpsv**. We selected Elias- γ or Elias- δ encoding as the function **enc**, and present the measured compression ratios in Figs. 4 and 5. We write

Table 2: 20 MB prefixes of the Pizza&Chili corpus datasets. H_k denotes the k -th order empirical entropy. z is the length of LZ-text(T), and r is the number of runs of BWT built upon the respective dataset.

dataset	σ	z	r	H_0	H_1	H_2	H_3	H_4
CERE	5	8492391	1060062	2.20	1.79	1.79	1.78	1.78
COREUTILS	235	3010281	910043	5.45	4.09	2.84	1.85	1.31
DBLP.XML	96	3042484	834349	5.22	3.26	1.94	1.26	0.89
DNA	14	12706704	1567099	1.98	1.93	1.92	1.92	1.91
E.COLI	11	8834711	1146785	1.99	1.98	1.96	1.95	1.94
ENGLISH	143	5478169	1277729	4.53	3.58	2.89	2.33	1.94
INFLUENZA	15	876677	210728	1.97	1.93	1.93	1.92	1.91
KERNEL	160	1667038	488869	5.38	4.00	2.87	1.98	1.47
PARA	5	8254129	1028222	2.17	1.83	1.83	1.82	1.82
PITCHES	129	10407645	2816494	5.62	4.85	4.28	3.50	2.18
PROTEINS	25	8499596	1958634	4.20	4.17	4.07	3.71	2.97
SOURCES	111	4878823	1361892	5.52	4.06	2.98	2.13	1.60
WORLDLEADERS	89	408308	129146	4.09	2.46	1.74	1.16	0.73

holz and **holz-opt** for our presented encoding **HOLZ**(T) and its bit-optimal variant, respectively.

We observe that the bit-optimal implementation uses a different variant of the LZ factorization that does not use the imaginary prefix $T[-\sigma..0]$ for references. Instead, it introduces *literal factors* that catch the leftmost occurrences of each character appearing in T . Their idea is to store a literal factor S by its length $|S|$ encoded in 32-bits, followed by the characters byte-encoded. In the worst case, they pay 40σ bits. This can pose an additional overhead for tiny files such as GRAMMAR.LSP, where 40σ bits are roughly 20% of their output size. However, it becomes negligible with files like CP.HTML, where less than 4% of the output size can be accounted for the literal factors.

Discussion The overall trend that we observe is that our encoding scheme **HOLZ** performs better than LZ on datasets characterized by a small high-order entropy. More in detail, when focusing on Elias- δ encoding (results are similar for Elias- γ), **holz-bitopt** compresses better than **bitopt** on all 8 datasets having $H_4 \leq 1$, and on 11 over 14 datasets with $H_4 \leq 1.5$. In general, **holz-bitopt** performed no worse (strictly better in all cases except one) than **bitopt** on 14 over 24 datasets. The trend is similar when comparing the versions of the algorithms that always maximize factor length: **holz** and **rightmost**. These results support the intuition that **HOLZ** is able to exploit high-order entropy, improving when it is small enough the compression ratio of the offsets with respect to LZ.

Table 3: Datasets from the Canterbury corpus; n is the number of text characters. See Table 2 for a description of the other columns.

dataset	n	σ	z	r	H_0	H_1	H_2	H_3	H_4
ALICE29.TXT	152089	74	66903	22897	4.56	3.41	2.48	1.77	1.32
ASYOULIK.TXT	125179	68	62366	21634	4.80	3.41	2.53	1.89	1.37
CP.HTML	24603	86	9199	4577	5.22	3.46	1.73	0.77	0.44
FIELDS.C	11150	90	3411	1868	5.00	2.95	1.47	0.86	0.62
GRAMMAR.LSP	3721	76	1345	853	4.63	2.80	1.28	0.67	0.44
KENNEDY.XLS	1486290	255	219649	145097	3.13	2.04	1.76	1.19	1.12
LCET10.TXT	426754	84	165711	52594	4.66	3.49	2.61	1.83	1.37
PLRABN12.TXT	481861	81	243559	72622	4.53	3.36	2.71	2.13	1.72
PTT5	961861	158	65867	25331	1.60	0.47	0.39	0.31	0.26
SUM	50503	254	13544	7826	4.76	2.52	1.61	1.15	0.90
XARGS.1	4227	74	2010	1172	4.90	3.19	1.55	0.72	0.42

6 Open Problems

We wonder whether there is a connection between our proposed encoding **HOLZ** and the Burrows-Wheeler transform (BWT) combined with move-to-front (MTF) coding, which achieves the k -th order entropy of T with $k = \Theta(\log_\sigma n)$. Applying MTF to BWT basically produces a list of pointers, where each pointer refers to the closest previous occurrence of a character whose context is given by the lexicographic order of its succeeding suffix. The difference is that this technique works on characters rather than on factors. Also, we would like to find string families where the compressed output of **HOLZ**(T) is asymptotically smaller than **LZ-text**(T), or vice-versa.

Our current implementation using dynamic wavelet trees is quite slow. Alternatively, for the encoding process we could use a static BWT and a dynamic prefix sum structure to mark visited prefixes in co-lexicographic order, which should be faster than a dynamic BWT. A more promising alternative would be to not use dynamic structures. We are confident that a good heuristic by hashing prefixes according to some locality-sensitive hash function (sensitive to the co-lexicographic order) will find matches much faster. Note that using the context of length k has the additional benefit to reduce the search space (compared to standard LZ), therefore the algorithm could be much faster and it could be easier to find potential matches.

Acknowledgements

This research was funded by JSPS KAKENHI with grant numbers JP21H05847 and JP21K17701, and by Fondecy Grant 1-200038 (Chile). We are grateful to github user MitL7 for some guidance regarding the dynamic wavelet tree implementation.

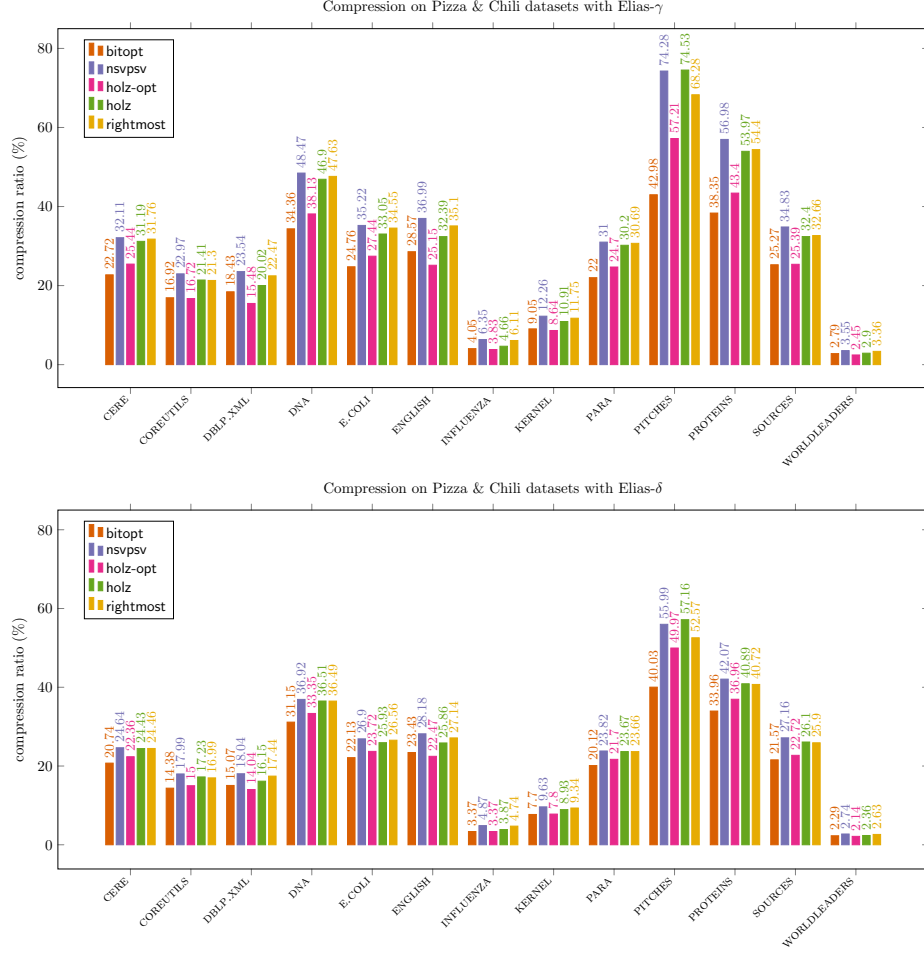


Figure 4: Compression ratios of different encodings and parsings studied in this paper on the Pizza&Chili corpus.

References

- [1] A. Lempel and J. Ziv, “On the complexity of finite sequences,” *IEEE Trans. Inf. Theory*, vol. 22, no. 1, pp. 75–81, 1976.
- [2] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [3] G. Navarro, “Indexing highly repetitive string collections, part I: Repetitiveness measures,” *ACM Comput. Surv.*, vol. 54, no. 2, pp. 29:1–29:31, 2021.
- [4] P. Ferragina, I. Nitto, and R. Venturini, “On the bit-complexity of Lempel–Ziv compression,” *SIAM J. Comput.*, vol. 42, no. 4, pp. 1521–1541, 2013.
- [5] A. D. Wyner and J. Ziv, “The sliding-window Lempel-Ziv algorithm is asymptotically optimal,” *Proc. IEEE*, vol. 82, no. 6, pp. 872–877, 1994.
- [6] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, Wiley, 2012.

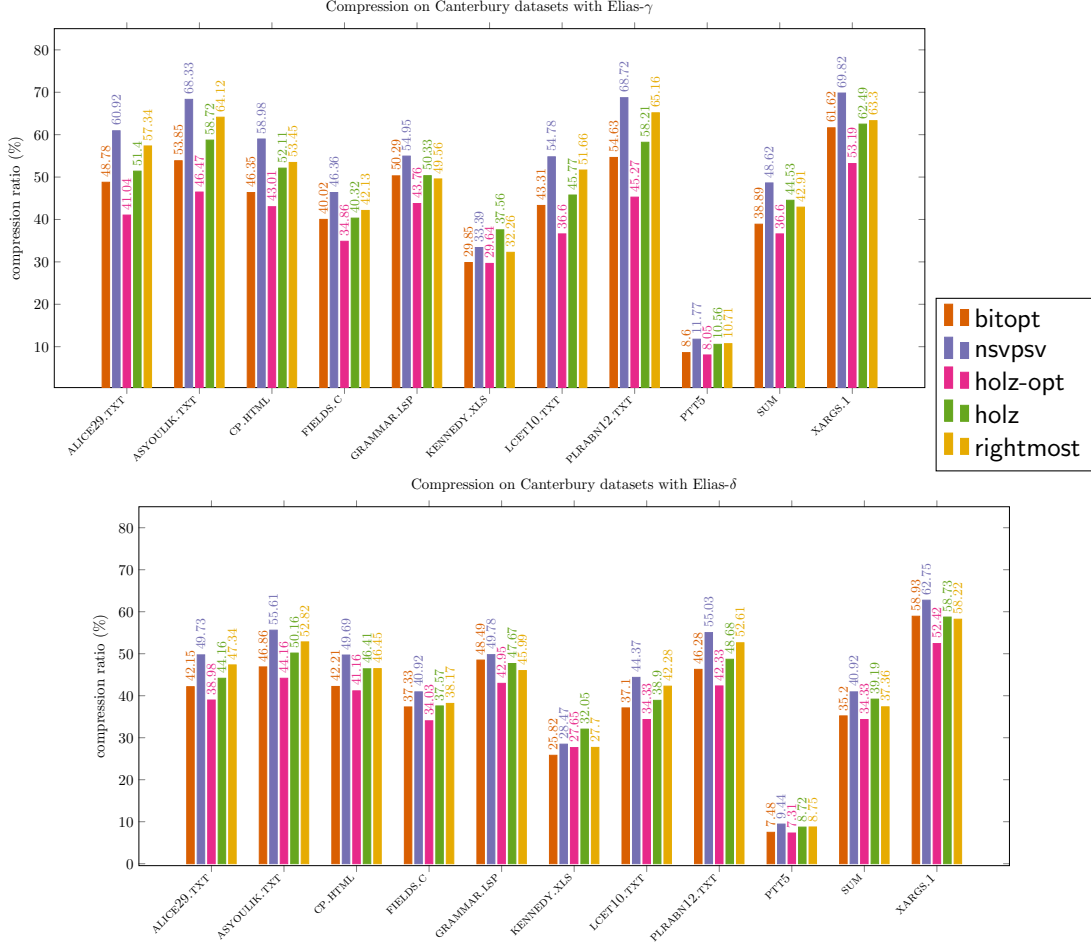


Figure 5: Compression ratios of different encodings and parsings studied in this paper on the Canterbury corpus.

- [7] Udi Manber and Eugene W. Myers, “Suffix arrays: A new method for on-line string searches,” *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.
- [8] A. Policriti and N. Prezza, “LZ77 computation based on the run-length encoded BWT,” *Algorithmica*, vol. 80, no. 7, pp. 1986–2011, 2018.
- [9] J. Ian Munro and Yakov Nekrich, “Compressed data structures for dynamic sequences,” in *Proc. ESA*, 2015, vol. 9294 of *LNCS*, pp. 891–902.
- [10] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Proc. FOCS*, 2000, pp. 390–398.
- [11] J. A. Storer and T. G. Szymanski, “Data compression via textural substitution,” *J. ACM*, vol. 29, no. 4, pp. 928–951, 1982.
- [12] A. Farruggia, P. Ferragina, A. Frangioni, and R. Venturini, “Bicriteria data compression,” *SIAM J. Comput.*, vol. 48, no. 5, pp. 1603–1642, 2019.

- [13] E. Ohlebusch and S. Gog, “Lempel–Ziv factorization revisited,” in *Proc. CPM*, 2011, pp. 15–26.
- [14] D. Belazzougui and S. J. Puglisi, “Range predecessor and Lempel–Ziv parsing,” in *Proc. SODA*, 2016, pp. 2053–2071.