Engineering Practical Lempel-Ziv Tries

DIEGO ARROYUELO, Millennium Institute for Foundational Research on Data (IMFD) and Universidad Técnica Federico Santa María, Chile
RODRIGO CÁNOVAS, Baker Heart and Diabetes Institute, Australia
JOHANNES FISCHER, TU Dortmund, Germany
DOMINIK KÖPPL, Tokyo Medical and Dental University, Japan
MARVIN LÖBEL, TU Dortmund, Germany
GONZALO NAVARRO, Millennium Institute for Foundational Research on Data (IMFD) and University of Chile, Chile
RAJEEV RAMAN, University of Leicester, UK

The Lempel-Ziv 78 (LZ78) and Lempel-Ziv-Welch (LZW) text factorizations are popular, not only for bare compression but also for building compressed data structures on top of them. Their regular factor structure makes them computable within space bounded by the compressed output size. In this article, we carry out the first thorough study of low-memory LZ78 and LZW text factorization algorithms, introducing more efficient alternatives to the classical methods, as well as new techniques that can run within less memory space than the necessary to hold the compressed file. Our results build on hash-based representations of tries that may have independent interest.

CCS Concepts: • **Theory of computation → Data compression;**

Additional Key Words and Phrases: Lempel–Ziv 78 factorization, compact hashing, space-efficient computation, Monte Carlo algorithm

Parts of this work have already been presented at the 24th International Symposium on String Processing and Information Retrieval [6, 29].
We acknowledge the funding by ANID—Millennium Science Initiative Program—Code ICN17_002 (D.A. and G.N.), and from JSPS KAKENHI with grant numbers JP18F18120 and JP21K17701.
Authors' addresses: D. Arroyuelo, Millennium Institute for Foundational Research on Data (IMFD) and Universidad Técnica Federico Santa María, Department of Informatics, Vicuña Mackenna 3939, Santiago, Chile; email: darroyue@inf.utfsm.cl; R. Cánovas, Cambridge Baker Systems Genomics Initiative, Baker Heart and Diabetes Institute, Melbourne, 75 Commercial Rd, Melbourne VIC 3004, Australia; email: rodrigo.canovas@baker.edu.au; J. Fischer and M. Löbel, TU Dortmund, Department of Computer Science, Chair of Algorithm Engineering (LS11), 44221 Dortmund, Germany; emails: johannes.fischer@cs.uni-dortmund.de, loebel.marvin@gmail.com; D. Köppl, Tokyo Medical and Dental University, 2-chōme-3-10 Kanda Surugadai, Chiyoda City, Tōkyō-to, 101-0062 Japan, M&D Data Science Center; email: koeppl.dsc@tmd.ac.jp; G. Navarro, Millennium Institute for Foundational Research on Data (IMFD) and University of Chile, Department of Computer Science, Beauchef 851, Santiago, Chile; email: gnavarro@dcc.uchile.cl; R. Raman, University of Leicester, Department of Informatics University Road, Leicester, LE1 7RH; email: rr29@leicester.ac.uk.

1084-6654/2021/10-ART1.14 $15.00
https://doi.org/10.1145/3481638

ACM Reference format:
Diego Arroyuelo, Rodrigo Cánovas, Johannes Fischer, Dominik Köppl, Marvin Löbel, Gonzalo Navarro, and Rajeev Raman. 2021. Engineering Practical Lempel-Ziv Tries. *J. Exp. Algor.* 26, 1, Article 1.14 (October 2021), 47 pages.
https://doi.org/10.1145/3481638

1 INTRODUCTION

A *text factorization*, in the following just called *factorization*, is a partitioning of a given text into substrings. Factorizations are the essential preprocessing step of many compression and text indexing data structures. One of the most famous factorizations with focus on compression is the Ziv-Lempel factorization of 1978 [73], known as LZ78. Its variants, especially *Lempel-Ziv-Welch* (LZW) [70], are used in compression software like unix `compress`, compression standards like `V.42bis`, and image and document formats such as GIF, TIFF, PDF, and PostScript.

Compared to the stronger LZ77 format [72], LZ78 has a more regular structure, which has made it the preferred choice for direct searching in compressed texts [2, 32, 33, 42, 44, 45, 57, 59], implementing string dictionaries [7], compressed sequence representations supporting optimal-time access [67], compressed text indexes for pattern matching [4, 27, 66], and document retrieval [25, 26].

Another important advantage of LZ78 over LZ77 is that LZ78 allows for an easy construction *within compressed space* and in *near-linear time*, which is (to date) impossible for LZ77. Still, although LZ77 factorizations often lead to marginally better compression rates, the output of LZ78 is usually small enough to be useful in practice as the basis of a compressor (especially the LZW variant) and to be the preferred choice in the scenarios mentioned above (where the LZ78 variant is more popular to build compressed data structures).

Computing the LZW and LZ78 factorizations both fast and space-economically allows us to compress larger files without splitting them into chunks. This not only yields better compression in general, but it is the only attractive choice when building data structures based on a Lempel-Ziv factorization. A simple and classical LZ78 or LZW implementation factorizes a text of length n over an alphabet $[1..\sigma]$ into z factors, where $\sqrt{n} \leq z = O(n/\log_\sigma n)$, in $O(n \lg \sigma)$ worst-case time (and $O(n)$ expected time) using $O(z \lg n)$ bits of main memory space, while reading the text from disk by small buffers. This working space is $O(n \lg \sigma)$ in the worst case, but more importantly, it is proportional to the size of the input text. Comparable results for LZ77, using $O(n \lg \sigma)$ bits of working space, were obtained only recently [31, 48, 55] and require much more sophisticated algorithms. Another line of research (e.g., References [10, 60, 64]) computes the LZ77 factorization in $O(r \lg n)$ bits of space, where r is the number of runs in the Burrows-Wheeler transform [15]. The only other LZ77 construction we are aware of with working space bounded by a function of the compressed file size poses a superlogarithmic penalty factor [61] on the time or only approximates the factorization within a constant factor [30].

1.1 Our Contribution

We introduce and study various LZ78 and LZW factorization algorithms that use lower memory space than the classical one. Some of them may run within memory sizes unable to fit even the compressed file. Most of our algorithms are *streaming*: On reading the text, they compute the factorization online and output a compressed stream, which can be written to external memory or to a network stream.

Our ideas are heavily based on hash-based trie representations, which represent nodes by identifiers of entries in a hash table. We present two such representations, where a node uses either (a) $O(\lg(z\sigma))$ bits or just (b) $O(\lg \sigma)$ bits. The former (a) can be used directly in conjunction with the

classic LZ78 and LZW factorization algorithms running in $O(n)$ expected time with $O(z \lg(z\sigma))$ bits of working space (cf. Theorem 4.1). The latter (b) uses an implicit representation of the identifiers in the hash table that are immutable until rehashing. For (b), the main challenge is to design a factorization algorithm that supports the rehashing of the hash table in such a way that

- the identifiers can be efficiently migrated during a rehashing, while
- the hash table is limited to have $O(z)$ cells without knowing z in advance.

With the representation (b), we present a specialized family of algorithms that carry out the LZ78 factorization within $O(z \lg \sigma)$ bits of main memory, which is less than any other previous LZ78 algorithm and asymptotically less than the size of the encoded output, $z(\lg z + \lg \sigma)$ bits.[1] One of our algorithms requires $O(n)$ expected compression time (cf. Theorem 5.3), but may rewrite and reread the output multiple times, whereas the other takes $O(n \lg \sigma)$ expected time but writes the output only once in a streaming fashion (cf. Theorem 5.4). Both algorithms produce a compressed output that can be decompressed in $O(n)$ time while using $O(z \lg \sigma)$ bits of main memory, where previous decompression algorithms need to fit the whole compressed text in main memory. This compressed output consists of a serialized compact hash table of $O(z \lg \sigma)$ bits and an array of z integers where the xth integer is a pointer to the cell in the hash table representing the xth factor. Consequently, our solution of (b) computes a special output format based on hash values.

Our results hold under some simplifying assumptions on randomness (cf. Section 3.5.3). Nevertheless, our experimental results demonstrate that these assumptions do not affect the practical competitiveness of the new algorithms, which outperform current alternatives in space by a factor from 2 to 5. We elaborate on these results by presenting, together with other trie representations, the first thorough study of LZ78 and LZW factorization algorithms. We highlight different algorithm engineering techniques supporting fast computation of the LZ78 and LZW factorization in practice. Our compact trie implementations may have independent interest beyond Lempel-Ziv factorization.

1.2 Article Outline

We first set our contribution in relation to previous achievements for computing the LZ78 factorization in Section 1.3. Subsequently, we review the LZ78 and LZW factorizations as well as two coding variants—the classic coding and the coding with the LZ trie—in Section 2. In Section 3, we review compact hash tables and propose a sparse layout for open-addressing compact hash tables. Next, we propose in Section 4 and 5 factorization algorithms that are partially based on these hash tables. While we use the classic coding in Section 4, a variant of the LZ trie coding allows us in Section 5 to slim down the working space, at the expense of using external memory, additional running time, or worse compression ratio.

1.3 Related Work

Although our focus is set on practically space-efficient algorithms, in what follows, we give a theoretical overview of other algorithms that compute the LZ78 factorization and draw a comparison with ours.

Unlike the preceding LZ77 factorization, LZ78 was designed [73] so it could be parsed easily using a trie called the *LZ trie*, within space proportional to the output of the factorization (i.e., to the size of the compressed text). A classic pointer-based implementation of the LZ trie with balanced binary trees to handle the children of each node carries out the LZ78 factorization in

[1] Our output is a rather naive representation of the factorization, i.e., we do not consider to further compress the representation of our factors; see Section 2.1.

$O(n \lg \sigma)$ time and $O(z \lg n)$ bits of space, where z is the size of the factorization. The same can be obtained for Welch's variant, LZW [70]. Using hashing to store the children of each node, the expected factorization time becomes $O(n)$. Despite some spare results we describe next, we are not aware of any practical systematic study of LZ78/LZW factorization algorithms.

With the trie representation of Fischer and Gawrychowski [28], we can accelerate the time of the classic implementation to $O(n + z\frac{\lg^2 \lg \sigma}{\lg \lg \lg \sigma})$. Spending $O(n(\lg \sigma + \lg \log_\sigma n)/\log_\sigma n)$ bits of space, Jansson et al. [40] can compute the factorization in $O(n \lg \sigma (\lg \lg n)^2/(\lg n \lg \lg \lg n))$ time. Their algorithm needs two sequential passes over the text, which involves I/Os if the text is stored in external memory.

Nakashima et al. [56] obtained the first $O(n)$ worst-case time factorization algorithm for LZ78, though it uses $O(n \lg n)$ bits of space. This space was recently improved by Fischer et al. [31], who need only $\min(O(n \lg \sigma), (1+\epsilon)n \lg n + n \lg \sigma + O(n))$ bits of working space, where ϵ with $0 < \epsilon \leq 1$ is a selectable constant tradeoff parameter, and $n \lg \sigma$ bits are used for the input text stored in a read-only memory with constant-time random access.

Considering that the computed LZ78 factorization can be stored in two arrays with $z \lg \sigma$ and $z \lg z$ bits to represent the last character and the referred index, respectively, of each factor, the $z \lg(\sigma z)$ bits of the factorization can be smaller than a working space of $O(z \lg n)$ bits, even if asymptotically similar for $\sigma = n^{O(1)}$. Closest to the exact output space is the approach of Arroyuelo and Navarro [3, Lem. 8], taking $z(2 \lg n + \lg \sigma + O(1))$ bits and $O(n(\lg \sigma + \lg \lg n))$ time for the LZ78 factorization. Having external memory, they manage to perform a single pass over the text, in exchange for $2z \lg z$ additional bits of I/O (on external memory), and a total time of $O(n(\lg \sigma + \lg \lg n))$, where the peak memory usage is $z(\lg n + \lg \sigma + O(1))$ bits. Measuring bits of I/Os instead of blocks differs from common approaches working in the external memory model. Here, each bit of I/O is read or written by a random access on disk.

They use the compact LZ trie representation described in Section 2.1.4, where $z \lg z$ bits are used for an array L storing preorder numbers of the LZ trie. An obstacle to further reduce the space is that they need to build the whole LZ trie before they can build the array L, because preorder numbers vary as new leaves are inserted. Later improvements on dynamic tries introduced by Arroyuelo et al. [5, Theorem 2] reduce the time to $O(n \log \sigma \cdot \frac{1}{\lg \lg \sigma})$, while consuming $z(\lg n + \lg \sigma + O(1))$ bits of space (in this theorem, we set $b := \lg n$ for the number of bits used by a satellite data entry). Their time result can be improved to $O(n \lg \sigma \cdot \frac{1}{\lg \lg n})$ by changing the arity of the used trie representation from $\Theta(\lg \sigma)$ to $(\lg n)/2$. This introduces an extra term of $o(\sqrt{n})$ bits of space, which is bounded by $o(z)$ according to Lemma 2.1. The time is now $O(n)$ for small alphabet sizes $\sigma = O(\text{polylog}(n))$. However, the peak space usage remains the same.

For comparison, we list the aforementioned results in Table 1 and 2 and put our contribution in this context.

2 PRELIMINARIES

With $\log_x$, we denote the logarithm to base x, and with lg the logarithm $\log_2$ to base two. Our computational model is the word RAM model with machine word size $w := \Omega(\lg n)$ for the input size n. Accessing a word costs $O(1)$ time.

Let T be a text of length n over an integer alphabet $\Sigma = [1..\sigma]$ with $\sigma = n^{O(1)}$. Given $X, Y, Z \in \Sigma^*$ with $T = XYZ$, then X, Y, and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. We call $T[i..]$ the ith suffix of T and denote a substring $T[i] \dots T[j]$ with $T[i..j]$.

Given a binary string $T \in \{0,1\}^*$, a bit $\mathsf{c} \in \{0,1\}$, and an integer j, the *rank* query $T.\text{rank}_\mathsf{c}(j)$ counts the occurrences of c in $T[1..j]$, and the *select* query $T.\text{select}_\mathsf{c}(j)$ gives the position of the jth c in T. We stipulate that $\text{rank}_\mathsf{c}(0) = \text{select}_\mathsf{c}(0) = 0$. There are data structures [19, 39] that use $o(|T|)$

Table 1. Previous and New LZ78 Compression Algorithms

Reference	Time	Working Space in Bits
Ziv and Lempel [73]	$O(n \lg \sigma)$	$O(z \lg n)$
	$O(n)^*$	$O(z \lg n)$
Nakashima et al. [56]	$O(n)$	$O(n \lg n)$
Fischer et al. [31]	$O(n/\epsilon^2)$	$(1+\epsilon)n \lg n + n \lg \sigma + O(n)$
Fischer et al. [31]	$O(n)$	$O(n \lg \sigma)$
Köppl and Sadakane [48]	$O(n \lg \lg \sigma)$	$O(n \lg \sigma)$
Fischer and Gawrychowski [28]	$O\left(n + z \frac{(\lg \lg \sigma)^2}{\lg \lg \lg \sigma}\right)$	$O(z \lg n)$
Jansson et al. [40]	$O\left(n \lg \sigma \cdot \frac{(\lg \lg n)^2}{\lg n \lg \lg \lg n}\right)$	$O(z \lg n)$
Arroyuelo and Navarro [3, Lem. 8]	$O(n \lg \sigma + n \lg \lg n)$	$z(2 \lg n + \lg \sigma + O(1))$
Arroyuelo et al. [5, Thm. 2]	$O\left(n \lg \sigma \cdot \frac{1}{\lg \lg n}\right)$	$z(\lg n + \lg \sigma + 2) + o(z \lg \sigma)$
Theorem 4.1	$O(n/\alpha^2)^*$	$\frac{z}{1-\alpha}(3 \lg(z\sigma) + 11)$
Theorem 5.4	$O(n \lg \sigma)^*$	$O(z \lg \sigma)$

Expected times are annotated with a star (*). $\epsilon \in (0, 1]$ and $\alpha \in (0, 1)$ are user-defined constants. We first list the classic approaches, then all deterministic ones, from most to least space-consuming (and, generally, from fastest to slowest). At the end, we present our randomized approaches. The methods that are asymptotically dominated by another in space and time are grayed.

Table 2. Semi-external Approaches Computing the LZ78 Factorization

Reference	Time	Working Space in Bits	I/Os in Bits
Arroyuelo and Navarro [3]	$O(n(\lg \sigma + \lg \lg n))$	$z(\lg n + \lg \sigma + O(1))$	$2z \lg z$
Theorem 5.3	$O(n)^*$	$O(z \lg \sigma)$	$z \lg^2 z$

Expected times are annotated with a star (*).

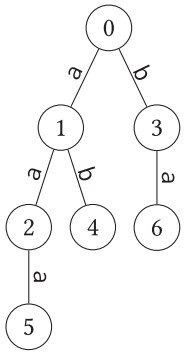
extra bits of space, and can compute rank and select in constant time, respectively. Each of those data structures can be constructed in time linear in $|T|$. We say that a bit vector has a *rank-support* and a *select-support* if it is endowed by data structures providing constant time access to rank and select, respectively.

Finally, a *factorization* of T of size z partitions T into z substrings $F_1 \dots F_z = T$. Each such substring F_x is called a *factor*. In this article, we are interested in the LZ78 and LZW factorizations.

2.1 Factorization and Coding

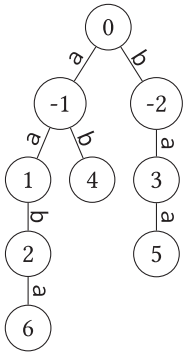
Stipulating that F_0 is the empty string, a factorization $F_1 \dots F_z = T$ is called the *LZ78 factorization* [73] of T iff, for all $x \in [1..z]$, the factor F_x is the longest prefix of $T[|F_1 \dots F_{x-1}| + 1..]$ such that $F_x = F_y c$ for some $y \in [0..x-1]$ and $c \in \Sigma$, that is, F_x is the longest possible previous factor F_y appending by the following character in the text. Formally, $y = \text{argmax}\{|F_{y'}| : F_{y'} = T[|F_1..F_{x-1}| + 1..|F_1..F_{x-1}| + |F_{y'}|]\}$. We say that y is the *referred index* of the factor F_x. Figure 1 gives an example. All factors F_x are distinct except maybe the last factor F_z, which needs to be treated as a border case. In what follows, we omit this border case analysis for the sake of simplicity (in LZ78 as well as in LZW). If T terminates with a character appearing nowhere else in T, then the last factor is also distinct from the others.

A factorization $F_1 \dots F_z = T$ is called the *LZW factorization* [70] of T iff, for all $x \in [1..z]$, it holds that the factor F_x is the longest prefix of $T[|F_1 \dots F_{x-1}| + 1..]$ such that (a) $F_x = F_y F_{y+1}[1]$ for some $y \in [0..x-1]$, or (b) $F_x = c \in \Sigma$ if no such y exists. Formally, if $F_x = F_y F_{y+1}[1]$, then



i	1	2	3
Factor	a	aa	b
Coding	(0,a)	(1,a)	(0,b)
i	4	5	6
Factor	ab	aaa	ba
Coding	(1,b)	(2,a)	(3,a)

Fig. 1. The LZ78 factorization and its LZ trie for the text T = aaababaaaba. The xth factor is the concatenation of the edge labels of the path from the root to the node labeled with x.



i	1	2	3	4
Factor	a	aa	b	a
Coding	−1	1	−2	−1
i	5	6	7	
Factor	ba	aab	a	
Coding	3	2	−1	

Fig. 2. The LZW factorization and its LZ trie for the text T = aaababaaaba. The xth factor is the concatenation of the edge labels of the path from the root to *the parent* of the node labeled with x.

$y = \text{argmax}\{|F_{y'}| : F_{y'}F_{y'+1}[1] = T[|F_1..F_{x-1}| + 1..|F_1..F_{x-1}| + |F_{y'}| + 1]\}$, and we call y the *referred index* of the factor F_x. Otherwise ($F_x = c \in \Sigma$), we have that $F_x = F_{-c}$ for a $c \in \Sigma$, and say that the referred index of F_x is $-c < 0$. The difference with LZ78 is that we do not include the symbol c that follows F_y when outputting the code of F_x, yet we take it as part of F_x when we reference it. The advantage of LZW is that, although it may produce more factors, encoding them requires just to output the referred indices, whereas LZ78 requires also to output the final characters. Figure 2 gives an example.

2.1.1 LZ Trie. As shown in Figure 1 and 2, the factors can be represented in a trie, the so-called *LZ trie*. The root node corresponds to F_0. In LZW, the root has additionally σ children, where the cth child has the label $-c$ (representing F_{-c}) and is connected to the root with an edge with label $c \in \Sigma$.[2] In both LZ78 and LZW, the node of $F_x = F_y \cdot T[j]$ is the child of the node of the factor F_y with the edge labeled by $T[j]$. Consequently, the LZ trie stores $z + 1$ indices that can be referred for LZ78, and $z + \sigma + 1$ for LZW.[3] The node corresponding to the xth factor is labeled with the factor index x. The set of factors is prefix-closed for both LZ78 and LZW.

[2]A different way is to represent the trie for LZW by a forest of σ trees, where the ith root has label $-i \in [-\sigma..-1]$. This saves one node, since an LZW factor never refers to the factor F_0.
[3]The number of factors z differs for LZ78 and LZW in general.

The following bounds on the number of LZ trie nodes will be useful:

LEMMA 2.1. *The number m of nodes in the LZ trie of LZ78 and LZW satisfies $\sqrt{2n+1/4}+1/2 \le m \le cn/\log_\sigma n$, for a fixed positive constant c. The upper bound can be refined to $(m-1)(\log_\sigma m - 3) < n$ if we do not count the root of the LZW trie.*

PROOF. For LZ78, the number of nodes is exactly the size of the factorization plus 1 (for the root, F_0). A lower bound of $\sqrt{2n+1/4} - 1/2$ for the number of LZ78 factors was proved by Bannai et al. [9, Lem. 1], using the only property that the length of each new factor is at most one plus the length of some previous factor. This fact holds for LZW, too, if we count the σ values $-c \in [-\sigma..-1]$ as factors. Since those nodes are included in the LZ trie of LZW, the same bound holds.

The upper bound holds for LZ78 because, for any factorization into z distinct factors (which is the case for all LZ78 factors except possibly the last one), we have that $z < \frac{n}{\log_\sigma n - 2\log_\sigma(1+\log_\sigma n)-2}$ [51, Thm. 2], which is $z \le cn/\log_\sigma n$ for some suitable constant c. In the case of LZW, we have that the strings $F_y F_{y+1}[1]$ are all distinct, therefore, we can form a text of length $n+z$ by concatenating those unique substrings. A crude upper bound $n+z \le 2n$ yields $z \le 2cn/\log_\sigma n$.

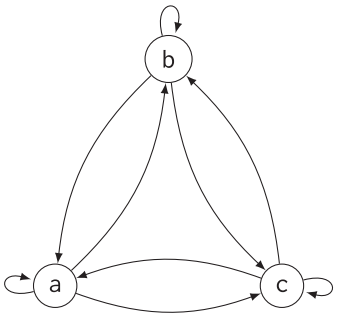
To obtain the precise upper bound, we build a worst-case text with all the shortest possible factors, as follows[4]: For LZ78, consider a text concatenating all distinct strings of length 1, then all distinct strings of length 2, and so on up to length k. The text is of length $n_k = \sum_{d=1}^{k} d\sigma^d = \frac{\sigma^{k+1}}{\sigma-1}\left(k - \frac{1}{\sigma-1}\right) + \frac{\sigma}{(\sigma-1)^2}$. Each distinct string produces a new factor, so the text has $z_k = \sum_{d=1}^{k}\sigma^d = \frac{\sigma^{k+1}-\sigma}{\sigma-1} < \frac{\sigma^{k+1}}{\sigma-1} - 1$ factors. Therefore, $n_k > \frac{\sigma^{k+1}}{\sigma-1}(k-1) > (z_k+1)(k-1)$.

To reach any arbitrary length $n_k \le n < n_{k+1}$, we complete the text with $\Delta < \sigma^{k+1}$ distinct factors of length $k+1$, plus a final possibly shorter one, each of which becomes a new LZ78 factor. Therefore, the total number of factors is $z = z_k + \Delta + 1 < \frac{n_k}{k-1} - 1 + \frac{n-n_k}{k+1} + 1 < \frac{n}{k-1}$. Since $z \le z_{k+1} < \frac{\sigma^{k+2}}{\sigma-1} - 1$, it follows that $k+2 > \log_\sigma(\sigma-1) + \log_\sigma(z+1) \ge \log_\sigma(z+1)$, thus $k-1 > \log_\sigma(z+1) - 3$. We then conclude that $z < \frac{n}{\log_\sigma(z+1)-3}$ and thus $z(\log_\sigma(z+1)-3) < n$. Since the trie of LZ78 has $m = z+1$ nodes, we obtain $(m-1)(\log_\sigma(m)-3) < n$.

Forming a worst-case text for LZW is slightly more complicated. Let $\Sigma = \{a_1, \ldots, a_\sigma\}$. Assume we have already formed all the trie nodes of depth d. Let $S_1, \ldots, S_{\sigma^{d-1}}$ be all the distinct strings of length $d-1$. Consider the complete directed graph with σ nodes and σ^2 edges (a_i, a_j) for all i, j. This graph is Eulerian because all indegrees and outdegrees are σ; let $e_1, \ldots, e_{\sigma^2} = (b_1, b_2)(b_2, b_3)\ldots(b_{\sigma^2}, b_1)$ be an Eulerian circuit, where each b_k for $k \in [1..\sigma^2]$ is some $a \in \Sigma$. For each S_i, if we append $b_1 S_i b_2 S_i b_3 S_i \ldots b_{\sigma^2} S_i b_1$, the LZW factorization will find each factor $b_j S_i$ (for $j \in [1..\sigma^2]$) in the trie and output it, inserting the new node corresponding to $b_j S_i b_{j+1 \bmod \sigma^2}$. See Figure 3 for an example. Because of the Eulerian circuit, all the strings aS_ia' for all $a, a' \in \Sigma$ are then formed. The last symbol we append, b_1, is then reused as the first character of a new string $b_1 S_{i'} b_2 S_{i'} b_3 S_{i'} \ldots\ldots b_{\sigma^2} S_{i'} b_1$ for another string $S_{i'} \ne S_i$. After repeating this for all the strings S of length $d-1$, we have created all the trie nodes of depth $d+1$. We have appended $1 + \sigma^{d-1}\sigma^2 d = 1 + d\sigma^{d+1}$ symbols and have created $\sigma^{d-1}\sigma^2 = \sigma^{d+1}$ factors. The last b_1 emitted can always be used as the first symbol of the next level.

Since the trie of LZW starts with the first level completed, we do this process for depths $d = 1, \ldots, k-1$, forming a text of length $n_k = 1 + \sum_{d=1}^{k-1} d\sigma^{d+1} = \frac{\sigma^{k+1}}{\sigma-1}\left(k - \frac{\sigma}{\sigma-1}\right) + \frac{\sigma^2}{(\sigma-1)^2} + 1$ that is parsed into $z_k = \sum_{d=1}^{k-1}\sigma^{d+1} = \frac{\sigma^{k+1}-\sigma^2}{\sigma-1} < \frac{\sigma^{k+1}}{\sigma-1} - \sigma$ factors. Therefore, $n_k > \frac{\sigma^{k+1}}{\sigma-1}(k-1) > (z_k+\sigma)(k-1)$.

[4]Generalized from https://ocw.mit.edu/courses/mathematics/18-310-principles-of-discrete-applied-mathematics-fall-2013/lecture-notes/MIT18_310F13_Ch20.pdf .



A possible Eulerian circuit of the left graph is (a,a), (a,b), (b,b), (b,c), (c,c), (c,a), (a,c), (c,b), (b,a). For the depth $d = 1$, the number of all distinct strings of length $d - 1 = 0$ is one, namely the empty string S. Hence, we only append $b_1Sb_2Sb_3S \cdots\cdots b_{\sigma^2}Sb_1$ = aabbccacba to our (yet empty) text. The LZW parsing of this text inserts the strings aa, ab, bb, bc, cc, ca, ac, cb, and ba into the LZ trie, thus creating all possible nodes at depth 2.

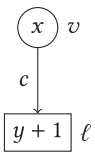
Fig. 3. Example for the graph described in the proof of Lemma 2.1 in the LZW part with $\Sigma = \{a, b, c\}$.

Analogously as for LZ78, we complete the text of length $n_k \le n < n_{k+1}$ with Δ distinct strings of length k, plus a final possibly shorter one, each of which becomes a new LZW factor (and creates a new trie node). The total number of factors is then $z = z_k + \Delta + 1 < \frac{n_k}{k-1} - \sigma + \frac{n-n_k}{k} + 1 < \frac{n}{k-1} - \sigma + 1$. Since $z \le z_{k+1} < \frac{\sigma^{k+2}}{\sigma-1} - \sigma$, it follows that $k+2 > \log_\sigma(\sigma-1) + \log_\sigma(z+\sigma) \ge \log_\sigma(z+\sigma)$, thus $k-1 > \log_\sigma(z+\sigma) - 3$. We then conclude that $z < \frac{n}{\log_\sigma(z+\sigma)-3} - \sigma + 1$ and thus $(z+\sigma-1)(\log_\sigma(z+\sigma)-3) < n$. Since the trie of LZW without the root has $m = z + \sigma$ nodes, we obtain $(m-1)(\log_\sigma(m)-3) < n$. □

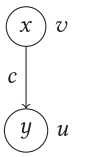
The lower and the upper bounds of Lemma 2.1 yield $\lg z = \Theta(\lg n)$.

2.1.2 Factorization Algorithm. To describe the classic factorization algorithm, we implement the LZ trie as a dynamic trie supporting two operations:

- insert(x, c) inserts a leaf ℓ connected to a node v with an edge labeled with $c \in \Sigma$, where v represents the xth factor. If the LZ trie contains y nodes (not counting those corresponding to F_d for $d \le 0$) before inserting ℓ, then the label of ℓ is the factor index $y + 1$.



- lookup(x, c) returns the index of the factor $F_y = F_x\, c$ whose corresponding node is connected to its parent v with an edge labeled with c, where the node v represents the xth factor. If v does not have such a child, then it returns an invalid index $\perp$.



The classic LZ78 factorization algorithm scans the text $T[1..n]$ from left to right. Suppose we have already factorized $T[1..i-1]$ into $x - 1$ factors $F_1F_2 \ldots F_{x-1}$. To compute F_x, we find the longest prefix $T[i..i+\ell-2]$ (with $i+\ell-1 \le n$) that is equal to some F_y, with $y \in [0..x-1]$ with F_0 being the empty string. Then, we define $F_x := F_y \cdot T[i+\ell-1]$, and we continue the parsing from $T[i+\ell]$.

We can use the LZ trie for the LZ78 factorization in the following way: To process $F_x = T[i..i+\ell-1]$, we traverse the LZ trie from the root downwards following the characters $T[i], T[i+1], \ldots$, traversing the nodes $y_0 := 0$ and $y_{k+1} := \text{lookup}(y_k, T[i+k])$, until $i+\ell-1 = n$ or we fall out of the tree at lookup$(y_{\ell-1}, T[i+\ell-1]) = \perp$. By doing so, we know that the referred index of F_x is $y_{\ell-1}$. Finally, we create a new node for F_x with insert$(y_{\ell-1}, T[i+\ell-1]) = x$.

Given that z is the number of LZ78 factors, the algorithm performs z searches for a text substring. It inserts z times a new leaf in the LZ trie. Since the total length of all factors is n, it traverses n times an edge from a node to one of its children. In total, it calls insert z times and lookup n times.

The case of LZW is very similar. We start with the LZ trie having the σ children y_{-c} of the root representing the single characters $c \in [1..\sigma]$. We traverse the trie with $T[i], T[i+1], \ldots$ via lookup

until finding the node $y_{\ell-1}$, where we perform insert exactly as before (now the limit is $i + \ell \leq n$). The difference is that we continue the factorization from $T[i + \ell - 1]$, not from $T[i + \ell]$. Therefore, for LZW, we call insert z times and lookup $n + z$ times.

A straightforward representation of the LZ trie storing z nodes consists of

- an array storing the labels within $z \lg \sigma$ bits,
- an array of the referred indices with $z \lg z$ bits, and
- a data structure of $O(z \lg n)$ bits for navigating from a node v to a child connected to v by an edge with a given label.

In total, this representation requires $O(z(\lg n + \lg \sigma)) \subseteq O(n \lg \sigma)$ bits for the factorization, which can be performed in $O(n \lg \sigma)$ deterministic time by implementing the child navigation data structure with balanced search trees, or in $O(n)$ randomized time by implementing this data structure with hash tables whose sizes double when needed.

2.1.3 Classic Coding. Having computed the factorization, we can achieve compression by encoding the factors. For that, we transform the list of factors into a list of integer values. In detail, we linearly process each factor F_x for $x \in [1..z]$, as follows:

LZ78: Given $F_x = F_y\, c$ for a $c \in \Sigma$ and $y \in [0..x-1]$, we output the pair (y, c).
LZW: If $F_x = F_y F_{y+1}[1]$, then we output y. Otherwise, $F_x = c \in \Sigma$ and we output $-c$.

Both factorizations also differ in how their output is encoded and decoded:

LZ78 Coding. The usual way to represent the LZ78 tuples in the compressed output consists of two (separate or interlaced) arrays $S[1..z]$ and $R[1..z]$ such that $R[x] = y \in [0..x-1]$ and $S[x] = c \in \Sigma$ for $F_x = F_y\, c$. We can naively store $S[1..z]$ in $z\lceil \lg \sigma \rceil$ bits, and $R[1..z]$ in $z\lceil \lg z \rceil$ bits. However, the referred index y (with $y > 0$) of a factor F_x can actually be stored in $\lceil \lg x \rceil$ bits, because a factor F_x can have a referred index y only with $y < x$. We can restore the referred index encoded in $\lceil \lg x \rceil$ bits on decompression, since we know the index of the xth factor and hence the number of bits $\lceil \lg x \rceil$ used to store its referred index.[5] This yields

$$\sum_{i=1}^{z} \lceil \lg i \rceil = z\lceil \lg z \rceil - (\lg e)z + O(\lg z) \text{ bits} \tag{1}$$

for storing the referred indices, where we used Stirling's approximation [35, Problem 3.34]. In total, we can represent S and R in $z(\lceil \lg z \rceil + \lceil \lg \sigma \rceil - \lg e) + O(\lg z)$ bits.

LZ78 Decoding. Having S and R, we can decode each factor F_x in turn: F_x is equal to $S[x]$ in case $R[x] = 0$, or otherwise it is the concatenation of the $R[x]$th factor (which we decode recursively) with $S[x]$, $F_x = F_{R[x]} \cdot S[x]$. If we have random access to $R[y]$ and $S[y]$ for $y \in [1..x]$, then we can decode the xth factor in $O(|F_x|)$ time and then decode the complete text in $O(n)$ time.

LZW Coding. For LZW, we cope with the negative integer values by adding σ to all output values, so its output consists of non-negative integers. With the same coding for the referred indices as in LZ78, the xth factor requires $\lceil \lg(x + \sigma) \rceil$ bits. By splitting up the sum $\sum_{i=1}^{z} \lceil \lg(i + \sigma) \rceil = \sum_{i=1}^{z+\sigma} \lceil \lg i \rceil - \sum_{i=1}^{\sigma} \lceil \lg i \rceil$, we get the total number of bits of the LZW output using Equation 1, $z(\lceil \lg(z + \sigma) \rceil - \lg e) + \sigma \lceil \lg \frac{z+\sigma}{\sigma} \rceil + O(\lg(z + \sigma))$. Assuming that both factorizations have the same number of factors, the coding of LZW uses less bits than LZ78 if $\sigma \ll z$. However, the number of factors of the LZ78 and the LZW factorization for the same text differ in general, so a comparison is not immediate.

[5] This is a folklore idea, see for example http://www.cplusplus.com/articles/iL18T05o.

LZW Decoding. A factor F_x with negative referred index, $R[x] = -c$, is decoded directly (we output $c \in \Sigma$). Any other factor is of the form $F_x = F_y F_{y+1}[1]$. We extract F_y as for LZ78 and repeat the process for F_{y+1}, just to obtain its first symbol. To recover the linear-time complexity, we can make a first pass over the z factors to obtain the first character of each: F_{-c} is associated with c and $F_x = F_y F_{y+1}[1]$ inherits the first character of F_y. Once this is done, requiring $z \lg \sigma$ extra bits of memory, we can obtain any $F_{y+1}[1]$ in constant time and decompress in total time $O(n)$.

2.1.4 LZ Trie Coding. An alternative way [4] to represent the LZ78 (or LZW) factorization uses a succinct encoding of the LZ trie, with:

- $2z + o(z)$ bits to represent the trie topology in a way that constant-time node navigation operations are supported, such as balanced parentheses [58] or depth-first unary degree sequence [12],
- $z \lg \sigma$ bits for the edge labels in preorder, and
- $z \lg z$ bits for an array $L[1..z]$ whose entry $L[x]$ stores the preorder number of the LZ trie node corresponding to the factor F_x.

All three data structures combined allow us to answer lookup in constant time. To extract an LZ78 factor F_x, we start from the node with preorder $L[x]$ in the LZ trie and use the trie topology to climb the trie upwards to the root. While climbing up, we read the edge labels of the visited path, which constitute F_x. In total, we need $O(|F_x|)$ time for decompressing F_x. To extract an LZW factor F_x, we proceed similarly, but start reading the characters at the parent of the node with preorder $L[x]$.

The LZ78 coding based on the LZ trie is more complex than that with the two arrays R and S. It also uses slightly more space than the former, namely, the $2z + o(z)$ bits for the tree topology. Yet, it is sometimes preferred because it enables operations other than just decompressing T. For instance, Sadakane and Grossi [67] showed how to obtain any substring of length ℓ of T in optimal time $O((\ell \lg \sigma)/w)$. In Section 5, a different representation of the LZ trie with a similar coding allows us to carry out the factorization within just $O(z \lg \sigma)$ bits of main memory.

2.2 Experimental Setup

We describe the common setting of our experiments throughout this article. We performed the experiments on an Intel Xeon CPU X5670 at 2.93 GHz with 49 GB of RAM running a 64-bit version of Arch Linux 2020 with Linux kernel 5.4.23-1-lts. We used a single execution thread for the experiments. We wrote our code in C++17 and compiled it with gcc version 9.2.1 via the compile flags `-O3 -march=native -DNDEBUG`. We measured the number of allocated bytes with tudostats,[6] which overrides the standard memory allocation (`new` and `malloc`) to additionally monitor the maximum allocated memory during execution.

The texts considered in this article are:

XML : a highly compressible XML text;
ENGLISH : an English text;
PROTEINS : a not very compressible proteins file;
DNA : a DNA file consisting of a prefix of a human genome[7];
COMMONCRAWL : ASCII webpages from commoncrawl;
FIBONACCI : the 46th Fibonacci word;
GUTENBERG : an excerpt of the Gutenberg project; and
WIKIPEDIA : Wikipedia's most vital articles.

[6] https://github.com/tudocomp/tudostats.
[7] http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/est.fa.gz.

Table 3. Text Files Used in the Experiments

text	n [M]	σ	z_{LZ78} [M]	c_{LZ78} [MB]	z_{LZW} [M]	c_{LZW} [MB]
COMMONCRAWL	10,739.46	127	679.55	3093.66	740.40	2642.29
DNA	3336.57	51	227.42	989.84	247.50	832.71
ENGLISH	1073.74	237	96.99	407.54	105.30	338.62
FIBONACCI	1836.31	2	1.52	5.26	1.52	3.74
GUTENBERG	1000.00	95	63.43	261.17	68.37	213.96
PROTEINS	1184.05	27	147.48	630.11	169.40	559.33
WIKIPEDIA	244.73	204	24.21	95.68	26.47	78.53
XML	296.14	97	16.21	62.72	18.05	52.21

Columns marked with "[M]" are divided by 10^{-6}; z_{LZ78} and z_{LZW} are the number of factors of the LZ78 and LZW factorization, respectively; c_{LZ78} and c_{LZW} are the size of the encoding of the LZ78 and the LZW factors, as described in Section 2.1.3 (with $\sigma = 2^8$).

The texts ENGLISH, PROTEIN, and XML are from the Pizza&Chili Corpus.[8] The other texts (except DNA) are from the tudocomp project.[9] Table 3 lists the texts with their main characteristics. Throughout all evaluations, we assume that the input sequence is drawn from a byte alphabet (i.e., $\sigma = 2^8$). It can be seen that, in all cases, $z_{\text{LZ78}} < z_{\text{LZW}}$ but $c_{\text{LZW}} < c_{\text{LZ78}}$, that is, LZ78 produces fewer factors but LZW outperforms it by encoding them better.

3 COMPACT HASH TABLES

We start with an analysis of space-efficient dictionaries, as we will use dictionaries for representing the LZ trie. In particular, we focus on compact hash tables and propose a new variant of them using a so-called *sparse layout*. To understand what compact hash tables are used for, we start with some abstract dictionary data types and then draw a connection between them and compact hash table instances.

A *set data structure* is a dynamic data structure storing keys from a universe $\mathcal{K}$. It provides the following methods:

- insert(K): inserts the key $K \in \mathcal{K}$.
- lookup(K): returns whether the key $K \in \mathcal{K}$ has been inserted.

The information-theoretic lower bound for storing n keys in a dynamic set is $\mathcal{B}(|\mathcal{K}|, n)$ bits of space, where $\mathcal{B}(|\mathcal{K}|, n) := \lg \binom{|\mathcal{K}|}{n}$. A set data structure can be augmented with satellite data, making it a *dictionary*. The satellite data can then be retrieved with lookup.

For our purposes, it is desirable to assign each key stored in a set data structure a unique identifier. For instance, when using a dynamic array A as the underlying data structure for the dictionary, A can give each key K the index ι of its stored entry $A[\iota] = K$ as its identifier. An *ID dictionary* is a set data structure that assigns each inserted key an identifier from a range $[1..\rho]$ for a variable ρ dependent on the number of stored elements n. These identifiers are unique and immutable up to at least $\Omega(n)$ update operations. The methods of the ID dictionary are:

- insert(K): inserts the key $K \in \mathcal{K}$ and returns its identifier.
- lookup(K): returns the identifier of the key $K \in \mathcal{K}$ if K has been inserted.
- key(ι): returns the key $K \in \mathcal{K}$ of the identifier $\iota \in [1..\rho]$ if such a key has been inserted.

[8] http://pizzachili.dcc.uchile.cl/texts.
[9] https://github.com/tudocomp/datasets.

Table 4. Overview of Known Bonsai Tables

Ref.	Space [bits]	ρ	Rebuilding	t_{Bonsai}
cleary [21]	$(1+\epsilon)\mathcal{B}$	$\Theta\left(\frac{n \lg n}{\lg \lg n}\right)$ whp	$\Theta(\epsilon n)$	$O(\frac{1}{\epsilon^2})$
elias [65]	$(1+\epsilon)\mathcal{B}$	$\Theta(n)$	$\Theta(\epsilon n)$	$O(\frac{1}{\epsilon})$
layered$_3$ [65]	$(1+\epsilon)\mathcal{B} + O(n \lg^{(5)} \lvert\mathcal{K}\rvert)$	$\Theta(n)$	$\Theta(\epsilon n)$	$O(\frac{1}{\epsilon})$
layered$_2$ (Section 3.2.3)	$(1+\epsilon)\mathcal{B} + O(n \lg^{(3)} \lvert\mathcal{K}\rvert)$	$\Theta(n)$	$\Theta(\epsilon n)$	$O(\frac{1}{\epsilon})$
bucket [49]	$\mathcal{B} + O(n \lg^{(2)} \lvert\mathcal{K}\rvert)$	$\Theta(n)$	$\Theta(n)$	$O(\lg \lvert\mathcal{K}\rvert)$

$\epsilon > 0$ is a user-specified constant. t_{Bonsai} is the *expected* time for an insert or lookup operation. The column *Rebuilding* shows the asymptotic number of insertion operations after which the hash table must be rebuilt. We abbreviate $\mathcal{B}(\lvert\mathcal{K}\rvert, n)$ to $\mathcal{B}$ and write $\lg^{(k)} = \lg^{(k-1)} \lg$ with $\lg^{(1)} = \lg$ for $k \geq 2$.

A dictionary can be converted into an ID dictionary by using as identifier the address where each key is stored internally, provided this is sufficiently immutable.

A *compact hash table* H [20] is a set data structure geared towards representing the keys memory-efficiently. It is associated with a hash function h with $h(K) \in [1..|H|]$ for every key $K \in \mathcal{K}$, where $|H|$ is the number of cells of H.[10] There is a restriction on the choice of the hash function h, as we request h to be accompanied by a *quotient* function q such that (h, q) is a bijective transformation, that is, we can restore K from the values $h(K)$ and $q(K)$ for all keys K. The idea is that we infer $h(K)$ from the position in H where K is stored, and thus only need to store $q(K)$ for retrieving K. This saves space if the number of bits needed to store $q(K)$, $|q| := \max_{K \in \mathcal{K}} \lceil \lg q(K) \rceil$, is less than that to store K, $\max_{K \in \mathcal{K}} \lceil \lg K \rceil$ [24, 47].

Interestingly, the works of Darragh et al. [21] and Poyias et al. [65] implicitly introduce compact hash tables that are implementations of ID dictionaries, which we call *Bonsai tables* in the following. We give an overview of these, together with a practically engineered hash table [49], in Table 4. The major design difference is that Darragh et al. [21] and Poyias et al. [65] apply open addressing, while Köppl et al. [49] apply closed addressing. In what follows, we briefly review the open-addressing approaches and present a variation of their table layout that uses less space in practice. The following scheme summarizes the various combinations to implement an open-addressing Bonsai table:

open-addressing Bonsai table = *layout* × *displacement*

- *layout*: plain, sparse
- *displacement*: cleary [20], array [65, Sect. 3.1], elias [65, Sect. 3.2], layered [65, Sect. 3.3]

Collisions in the open-addressing Bonsai hash tables are resolved by open addressing with linear probing,[11] that is, a call to insert(K) looks for the first vacant space in $H[h(K) + i \mod |H|]$ iteratively, for increasing integers $i \geq 0$, and inserts the quotient $q(K)$ at that place, that is,

$$H[h(K) + \min\{i \geq 0 : H[h(K) + i \mod |H|] \text{ is empty}\}] \leftarrow q(K).$$

[10] The number of cells is not smaller than the number of stored entries n, and it is at least n/α for open-addressing hash tables with a maximum load factor of $\alpha \in (0, 1]$.

[11] Actually, Darragh et al. [21] used bidirectional probing, but we stick to linear probing to simplify the explanation (all results explained here also work with bidirectional probing).

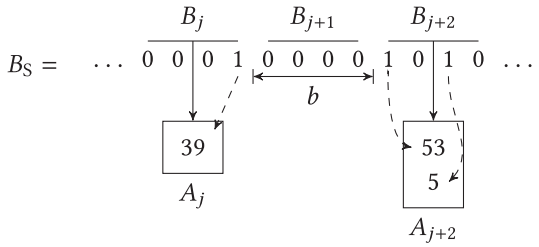


Fig. 4. Sparse hash table layout. The bit vector B_S of length $|H|$ is the concatenation of the bit vectors $B_1 \dots B_{|H|/b}$ with $|B_j| = b$ for all $j \in [1..|H|/b]$, where $b = 4$ in this example. B_S stores at position i whether $H[i]$ stores an entry. The actual values of H are stored in dynamic arrays A_j. The value of $H[i]$ can be retrieved by following the respective dashed arrow from $B_S[i]$.

In this article, we stipulate that $x \bmod n := x$ if $x \leq n$ and $x - n \bmod n$ otherwise, for an integer $x \geq 1$. Hence, $n \bmod n = n$, but $n + 1 \bmod n = 1$.

3.1 Sparse Table Layout

In its plain layout, a hash table H with open addressing uses $|H| \lg |q|$ bits for storing its entries, regardless of the number of entries. On the one hand, this is wasteful for low load factors. On the other hand, open-addressing hash tables become exponentially slower at high load factors (cf. Section 3.5.2). As a remedy, we propose the *sparse table layout*, which does not allocate memory pages for all entries at once, but uses an array of pointers to small tables that are allocated dynamically when needed. It is inspired by Google's sparse hash table[12] and consists of a bit vector B_S of length $|H|$ and dynamic arrays. The entries of the hash table are mapped to entries of the arrays with B_S. Figure 4 sketches this layout.

In detail, we partition the hash table into $|H|/b$ sections, where b is a (small) constant that is a power of two. We assume that $|H|$ is divisible by b, so all sections have the same length b. For instance, this is the case when $|H|$ and b are powers of two (with $b < |H|$). Given that we want to access the kth cell of the hash table for $k \in [1..|H|]$, there are integers $i := k \bmod b \in [1..b]$ and $j := \lceil k/b \rceil \in [1..|H|/b]$ such that $k = i + (j-1)b$. Then $H[k]$ is mapped to the ith entry of the jth section. By interpreting B_S as the concatenation $B_1 \dots B_{|H|/b}$ of bit vectors of length b, the jth section is represented by the bit vector B_j and a dynamic array A_j. We maintain B_j and A_j such that the ith entry of the jth section is stored at position $B_j.\mathrm{rank}_1(i)$ in A_j. For a sufficiently small b, the rank query can be answered with a constant number of CPU instructions (on a modern CPU architecture) on the bit vector B_j without the need of a (dynamic) rank support. Finally, we store pointers to the dynamic arrays A_j of the sections in an array of length $|H|/b$.

We can insert an entry in the jth section by setting the appropriate bit in B_j and rearranging the elements in A_j. We can rearrange A_j efficiently if the elements of A_j (which are at most b) fit into the CPU cache. Whenever we want to insert an element into a full array A_j with $|A_j| < b$, we double A_j's size. Initially, all arrays A_j for $j \in [1..|H|/b]$ are empty. In total, we need $|H|$ bits and $(|H|/b)(w + \lg b)$ bits for the bit vector B_S and the arrays A_j (each consisting of a pointer with w bits and a counter with $\lg b$ bits maintaining its size), respectively, in addition to the actual entries in A_j, which use $n \lg |q|$ bits.

Compared to the plain layout using $|H| \lg |q|$ bits, the sparse layout is more space-economical for low load factors and large values of q. Specifically, for $b = w$, the sparse layout is more

[12] https://github.com/sparsehash/sparsehash.

space-economical if

$$2|H| + (|H| \lg w)/w + n \lg |q| < |H| \lg |q| \Leftrightarrow 2 + (\lg w)/w \leq \frac{|H| - n}{|H|} \lg |q|.$$

If we use a maximum load factor $\alpha \in (0, 1]$, double the size of H when reaching $\alpha|H|$ elements, and neglect deletions (as we do throughout this article), then $|H| \in [n/\alpha..2n/\alpha]$. Hence, $\frac{|H|-n}{|H|} \in [1 - \alpha, 1 - \alpha/2]$. A consequence is that the sparse layout pays off at least when the bit widths of the quotients is larger than $(2 + (\lg w)/w)/(1 - \alpha)$.

On the downside, the sparse layout is slower in practice due to the indirect access to B_S needed to determine the corresponding position in a dynamic array A_j. Additionally, an insert operation may cause the reallocation of a dynamic array.

The sparse table layout resembles a hash table of size $|H|/b$ with closed addressing using bucketing (cf. Section 3.4). The difference arises when one of the dynamic arrays A_j becomes full. In such a case ($|A_j| = b$), when trying to add a new element in A_j, the sparse table layout probes the next array A_{j+1} (due to the linear probing) instead of enlarging A_j or rehashing the entire hash table (cf. Reference [49, Section 2.3]).

3.2 Displacement

Given that the quotient $q(K)$ of a key K is stored in the $h(K)$th cell of the hash table, it is easy to retrieve K, since we have $q(K)$ and $h(K)$ at hand. In case a key K cannot be stored in the cell $h(K)$ due to a collision, we need to resolve this collision and store $q(K)$ in a different cell. For restoring the value $h(K)$ of a stored entry $H[i] = q(K)$, we need additional information about the difference $i - h(K)$. This difference is called a *displacement*. It can be represented as an array or with bit vectors, as we describe in the following:

3.2.1 Array Displacement. The array displacement maintains an integer array D of size $|H|$, and hence needs $|H| \lg |H|$ additional bits of space. Given a key K whose quotient is stored in $H[i]$, its entry $D[i]$ is $i - h(K) \mod |H| \in [1..|H|]$, except that $D[i] = 0$ means that K is stored directly in $H[h(K)]$. Having D at hand, $h(K) = i - D[i] \mod |H|$ is the hash value of a key K with $H[i] = q(K)$. Further, we stipulate the invariant that $D[i] = -1$ signals that the ith cell is empty.

An insertion works as follows: Suppose that we want to insert a key K into the cell $H[p]$ with $p = h(K)$. If $H[p]$ is free, then we are done and set $D[p] = 0$. Otherwise, we probe consecutive positions $H[p]$ with $p = h(K) + j \mod |H|$, for $j = 1, 2, \ldots$, where one of the following cases is met:

(1) $H[p]$ is free. In this case, we terminate with $H[p] \leftarrow q(K)$ and $D[p] \leftarrow j$, so $D[p]$ indicates the number of probes between $h(K)$ and the final position p where K is finally written.
(2) $H[p]$ is not free, $H[p] = q(K)$, and $(p - D[p]) \mod |H| = h(K)$. This means that the key K is already stored in H, thus there is no need to insert it.
(3) $H[p]$ is not free, but $H[p] \neq q(K)$ or $(p - D[p]) \mod |H| \neq h(K)$. Thus, the cell is occupied by another key and we continue probing.

Operation lookup(K) works in the same way, except that Case 1 implies that K is not stored in the set and Case 2 implies that we have found K.

3.2.2 Displacement Elias. The displacement elias divides the array D of the array displacement into blocks of a fixed size b. Each block stores its associated displacement values in a bit string encoded with Elias-γ [23]. For $b = (\lg|H|)^{3/2}$, the displacement information uses $O(|H|)$ bits of additional space in expectation [65, Lemma 6].

3.2.3 Displacement Layered. The displacement layered$_3$ (called recursive in Reference [65, Section 3.3]) is parameterized with two integer constants b_0, b_1 with $0 \le b_0 \le b_1 \le |H|$. It uses a three-layer approach (hence the 3 in the subscript) to store the values of D, consisting of

(a) an array D' with $D'[i] = D[i]$ if $D[i]$ can be represented within b_0 bits or $D'[i] = \perp$, for an escape symbol $\perp$, otherwise,
(b) a compact hash table storing all remaining displacement values that can be represented within b_1 bits, and finally
(c) a dictionary (e.g., a plain hash table) for all displacement values that cannot be represented within b_1 bits.

In addition to layered$_3$, we introduce the simplification layered$_2$, which uses two layers by omitting b_1 and the compact hash table. We prove next that our simplification has an overhead of $O(\lg^{(3)} n)$ bits per entry, while layered$_3$ has an overhead of only $O(\lg^{(5)} n)$ bits per entry.

Lemma 3.1. *Assuming full randomness, the Bonsai table layered$_2$ takes constant expected time for* lookup *and* insert *with an overhead of $O(\lg^{(3)} n)$ bits per entry.*

Proof. Our idea is to choose a sufficiently large b_0 for the bit width of the array D' in the first layer such that the probability of a larger displacement becomes $O(1/\lg n)$. Thus, the dictionary of the second layer only needs to store $O(n/\lg n)$ elements in expectation. We can implement this dictionary with a classical dynamic search structure like an AVL tree, which takes $O(\lg n)$ time per operation and needs $O(n)$ bits in expectation. Assuming randomness for queries, we use this dictionary expectedly once per $O(\lg n)$ operations, and hence an operation on the hash table costs $O(1)$ expected time. We show that these space and time bounds are guaranteed with $b_0 \ge \lg^{(3)} n$: Poyias et al. [65, Theorem 7] show that the probability of storing a number larger than k in D' is $O(c^k)$ for some $0 < c < 1$; therefore, the probability of an overflow in D' is $O(c^{2^{b_0}})$. Choosing $b_0 \ge \lg^{(3)} n - \lg\lg(1/c)$, this probability is $O(1/\lg n)$. □

3.2.4 Cleary Displacement. A different approach [20] that does not store the displacement values explicitly is cleary, which uses two bit vectors B_V and B_C to represent the displacement.[13] The bit vectors B_V and B_C are of length $|H|$ and are defined as follows:

- $B_V[i] = 1$ if and only if there is a key K stored in the hash table with $h(K) = i$.
- $B_C[i] = 1$ if and only if
  - the cells $H[i] = q(K)$ and $H[i-1] = q(K')$ have different hash values $h(K) \ne h(K')$, or
  - the cell $H[i]$ is empty.

Suppose that $B_C[i] = 1$ and $H[i]$ is non-empty, then all cells $H[k]$ for $k \in [i, B_C.\text{select}_1(B_C.\text{rank}_1(i) + 1))$ have the same hash value. We say that these cells belong to the same *group*. To determine their hash value v, we search the largest position s smaller than i that is empty. This position s has the property that $B_C[s] = B_C[s+1] = B_V[s+1] = 1$. With s, we can compute the hash value v with $v = B_V.\text{select}_1(B_V.\text{rank}_1(s+1) + B_C.\text{rank}_1(i) - B_C.\text{rank}_1(s+1))$. This formula is correct, because we maintain the key order across contiguous groups, as we show soon in the insertion process.

The rank/select operations are computed by linearly scanning the two bit vectors. This is not a time bottleneck, since $i - s$ is small in expectation for low load factors. Figure 5 gives an example.

[13] "V" for virgin and "C" for change [20].

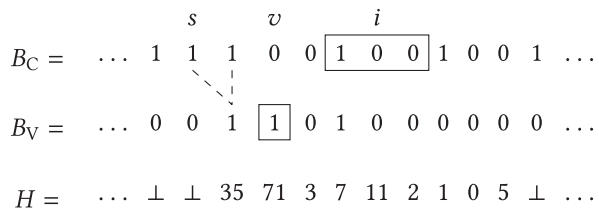


Fig. 5. Hash table H of quotients with cleary displacement storing the bit vectors B_C and B_V. The hash value v of the entry stored in cell $H[i] = 11$ is equal to that of the entries $H[i-1] = 7$ and $H[i+1] = 2$, because they are in the same group (marked by the rectangle in B_C). The number of ones from $s+1$ up to this group is 2, so the hash value of the group is the second one in B_V starting with $s+1$, which is $s+2$ in this example.

With B_V and B_C, cleary needs only $2|H|$ additional bits of space. The rearranging strategy is similar to Hopscotch hashing [38], where entries are rearranged such that the distance of an entry $q(K)$ to $h(K)$ is within a cache line.

Insertions are done in the following way: Suppose that we want to insert K into the cell $H[p]$ with $p = h(K)$. If $H[p]$ is empty, then we are done by setting $H[p] \leftarrow q(K)$, $B_C[p] \leftarrow 1$, and $B_V[p] \leftarrow 1$. If $H[p]$ is not empty, then we check in $B_V[p]$ whether there is a group with hash value p.

- If this is the case, then we find this group as above by locating the largest position $s < p$ with $H[s]$ being empty and then linearly scanning the cells until we find this group. We scan to the end of this group, checking if one of the stored quotients matches the quotient $q(K)$ we want to insert.
  - If we find such a quotient, then we are done, since the element we want to insert has already been inserted.
  - Otherwise, we reach the end of this group at position i. We shift all succeeding consecutive cells (i.e., the contents of H, B_C and B_V) that are not empty by one position to the right, and then insert $q(K)$ at position $i+1$, setting $B_C[i+1] \leftarrow 0$.
- Last, if there is no such group, then we find the group with the preceding hash value, ending at position i, and do the same steps (we shift the cells and insert $q(K)$). However, this time, we set $B_V[p] \leftarrow 1$ and $B_C[i+1] \leftarrow 1$.

In our sparse layout, we partition not only H but also B_V and B_C into sections. We can handle the bit vectors with the same logic as for handling satellite data. This works well, because we do not rely on rank or select support data structures, that is, we naively scan both bit vectors.

3.3 Identifier

The identifier is an integer in the range $[1..\rho]$, where $\rho = \Omega(n)$ is defined independently for each hash table (cf. Table 4).

Displacement array D. All approaches maintaining the displacement array D (such as elias and layered) can guarantee that a hashed element will not be moved until rehashing. Therefore, the identifier is the position in the hash table at which an element is stored, thus $\rho = |H|$.

cleary. Because cleary might move elements upon insertions, we cannot simply use the positions in H as identifiers. Instead, we use the pair $(h(K), g)$ for a key K, where g is the number of keys that are in the same group as K and have been inserted in H prior to K. This identifier stays unchanged upon insertions [21, p. 282] and can be stored in $\lg(|H|\lambda)$ bits, where λ is the

maximum group size. For $\lambda = \Omega(\lg n / \lg \lg n)$, it is guaranteed that a rehashing can be delayed for $\Theta(n)$ insertions [65, Section 2.3], which yields the bounds for ρ in Table 4.

3.4 Other Collision Resolutions

A displacement strategy is usually based on the collision resolution, which is linear probing in our case. In fact, we can also use bidirectional linear probing [1], as originally proposed by Cleary [20], by considering negative values in the displacement array. Other strategies like Cuckoo-Hashing, instead, do not make much sense as ID dictionaries, since we would need to store the identifiers explicitly to support element swapping. If the identifiers are not necessary (e.g., our LZ78 and LZW trie representation cht in Section 4.2.2 only needs a compact hash table), then other strategies like Hopscotch hashing [38] or Robin-Hood hashing [16] can also be combined with our displacement strategies above, yielding other interesting compact hash tables.

Here, we focus on a compact hash table resorting to closed addressing by hashing entries to a bucket with a limited maximum size. Besides the one given in Table 4, which we name bucket (it is called cht in Reference [49]), Köppl et al. [49] proposed another compact hash table called grp, using $\mathcal{B}(|\mathcal{K}|, n) + O(n)$ bits and with $O(\lg |\mathcal{K}|)$ worst-case time for insert and $O(1)$ expected time for lookup. Unfortunately, this table groups the buckets similarly to cleary but in such a way that it is not clear how to represent the identifiers in a space-friendly way to guarantee $\Omega(n)$ insertions before a rehashing.

3.5 Open-addressing Implementations

For the practical implementation of our open-addressing hash tables, we describe how to perform resizes and which hash function we choose.

First, we double the number of cells of H on rehashing, that is, when reaching the maximum number of entries $\alpha|H|$ for a user-defined constant $\alpha \in (0, 1]$, which we call the *maximum load factor*.

3.5.1 Table Size. We choose the hash table size $|H|$ to be a power of two. Having $|H| = 2^k$ for $k \in \mathbb{N}$, we can compute the remainder of the division of a hash value by the hash table size with a bitwise-and operation: $h(K) \bmod 2^k = 1 + ((h(K) - 1) \,\&\, (2^k - 1))$, which is faster in practice (see, e.g., Reference [54]).

3.5.2 Reasons for Linear Probing. Linear probing inserts a tuple with key K at the first free entry, starting at the hash value $h(K)$. It is cache-efficient if the keys have a small bit width (i.e., fitting in a computer word). Using large hash tables and small keys, the cache-efficiency can compensate the chance of more collisions [8, 37]. Linear probing excels if the load ratio is below 50%, and it is still competitive up to a load ratio of 80% [14, 53]. Nevertheless, its main drawback is *clustering*: Linear probing creates runs, that is, entries whose hash values are equal. With a sufficiently high load, it is likely that runs merge and long sequences of entries with different hash values emerge. When looking up a key K, we have to search the sequence of successive cells starting at the $h(K)$th cell until finding a tuple whose key is K or an empty entry. Fortunately, this search is fast if the maximum load factor $\alpha \in (0, 1]$ is not too close to 1, since it takes $O(1/(1-\alpha)^2)$ expected time [47] under the assumption that the used hash function h distributes the keys independently and uniformly. In practice, even weak hash functions (like those we use in this article) tend to behave as truly independent hash functions [18].

3.5.3 Bijective Transform. We follow the ideas of Poyias et al. [65] for the bijective transform: We use $f(K) = aK \bmod p$ with $h(K) := f(K) \bmod |H|$ and $q(K) := \lfloor f(K)/|H| \rfloor$, where $a \in [1..p-1]$ is a randomly chosen constant, and p is the first prime such that $p > |\mathcal{K}|$.

Since $p \leq 2|\mathcal{K}|$ [36, p. 343] (see also Reference [17]), it holds that $q(K) = O(|\mathcal{K}|/|H|)$, and thus the quotients stored in H require $\lg|\mathcal{K}| - \lg|H| + O(1)$ bits. With this information, we can reconstruct $aK \bmod p = q(K)|H| + h(K)$, and then $K = a^{-1} \cdot (a \cdot K) \mod p$, where a^{-1} is the modular multiplicative inverse of a with modulus p, which can be computed with the extended Euclidean algorithm in $O(\lg|\mathcal{K}|)$ time in a precomputation step [46, Section 4.5.2].

Note that the obtained hash function $h(K) = f(K) \mod |H|$ is only 1-independent for randomly chosen a and p. Just 1-independence is not sufficient to ensure randomness in the case of linear probing; this has been proven only for 5-independence [62, 63]. To make the hash function 5-independent, the component $a \cdot K$ of our bijective transform f should become a polynomial of degree four, $f(K) = (a_0 + a_1K + a_2K^2 + a_3K^3 + a_4K^4) \mod p$. However, we do not know how to invert this function f for arbitrarily chosen constants $a_1, a_2, a_3, a_4 \in [1..p-1]$.

Luckily, the following experiments reveal that those theoretical reservations do not have a significant impact on the practical performance of the scheme.

3.6 Experiments

We evaluate our proposed sparse layout with experiments measuring their usefulness compared to the plain layout. Our focus lies on inserting and querying elements, which are the main tasks for the LZ78 construction.

For the open-addressing Bonsai tables, we append subscripts "P" or "S" if the respective variant uses the plain or sparse form, respectively. We set the load factor of all open-addressing hash tables to $\alpha \leftarrow 0.95$. For layered = layered$_2$, we set $b_0 := 4$, and use `std::unordered_map` as the dictionary for all values with bit width $> b_0$. For elias, we set the block lengths to 1,024.

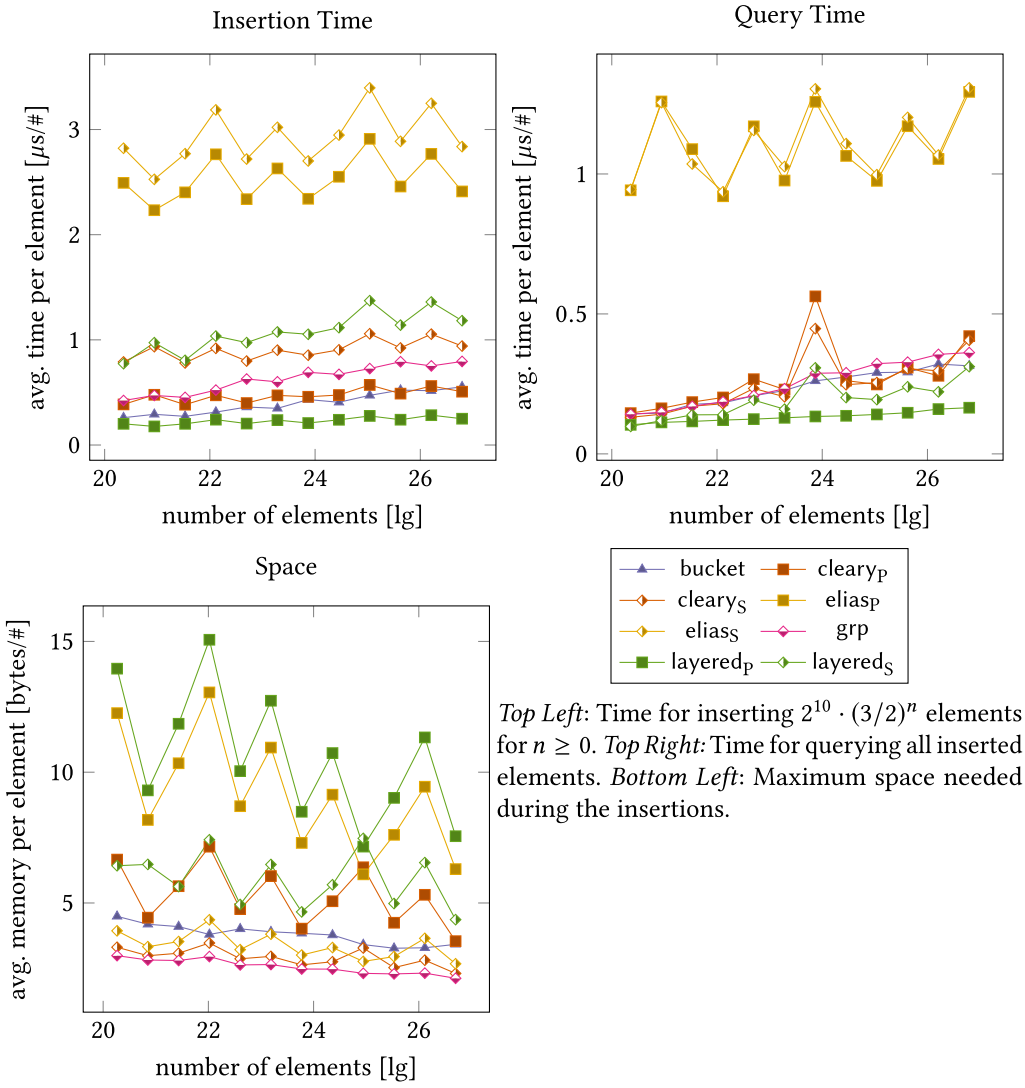
We evaluated all the compact hash tables mentioned in Figure 6 on randomly generated inputs consisting of 32-bit keys and 8-bit values. The first observation is that the sparse variants indeed save space while being somewhat slower than their plain counterparts. The variant layered$_\text{P}$ is one of the extreme solutions, being the fastest option both for insertion and querying but also the worst solution regarding space requirements. While its sparse variant layered$_\text{S}$ has nearly as good query times, it is outpaced by several other approaches for insertions. cleary$_\text{S}$ and cleary$_\text{P}$ are always superior to elias$_\text{S}$ and elias$_\text{P}$ with respect to time and mostly to memory, and are better balanced than the layered variants. The most space-economical approaches are grp, followed by cleary$_\text{S}$, elias$_\text{S}$, and bucket. For the insertions, bucket is faster than grp, which is again faster than cleary$_\text{S}$. At querying, all these variants behave roughly the same.

Comparison and Outlook. We conclude that grp and cleary are well-suited as compact hash tables. If we focus on small identifiers, then both approaches are inferior to layered, which is a well-suited Bonsai table. In the rest of this article, we stick to

- grp and cleary for compact hash table representations, and to
- layered$_2$ and layered$_3$ for Bonsai table representations, where layered$_3$ uses bucket as the compact hash table in the middle layer.

4 LZ TRIE REPRESENTATIONS

In this section, we focus on the classic LZ78 and LZW algorithms described in Section 2.1.2. These algorithms compute the respective factorization by maintaining the LZ trie in memory. For the LZ trie, we study five representations providing different tradeoffs between computation speed and memory consumption. All representations have in common that they work with dynamic arrays.



Top Left: Time for inserting $2^{10} \cdot (3/2)^n$ elements for $n \geq 0$. *Top Right:* Time for querying all inserted elements. *Bottom Left*: Maximum space needed during the insertions.

Fig. 6. Managing randomly generated 32-bit keys and 8-bit values with the compact hash tables described in Section 3.

Resize Hints. The usual strategy for dynamic arrays is to double the size of an array when it becomes full. To reduce the memory consumption, a hint on how large the number of factors z might be is convenient for a dynamic LZ trie data structure. We provide such a hint based on Lemma 2.1. At the beginning of the factorization, we let a dynamic trie reserve enough space to store at least $\sqrt{2n}$ elements without resizing, as this is the lower bound on the number of factors. Upon enlarging a dynamic trie, we usually double its size. However, if the number r of remaining characters to parse is below a certain threshold, then we scale the data structure up to a value with which we expect that all factors can be stored without resizing the data structure again. Let z' be the computed number of factors up to now. If $r > n/2$, then we use $z' + 3r/\lg r$ as an estimate

index	1	2	3	4	5	6
first child	2	5	6			
next sibling	3	4				
character	a	a	b	b	a	a

Fig. 7. Array data structures of binary built on the LZ78 example given in Figure 1.

(the number 3 is chosen empirically[14]), derived from $z - z' = O(r/\log_\sigma r)$ based on Lemma 2.1. Otherwise, we interpolate $z' + z'r/(n-r)$ with the assumption that the ratio between z' and $n-r$ will be roughly the same as between z and n.

4.1 Deterministic LZ Tries

We first cover two trie implementations that use arrays to store at position x the node labeled with the factor index $x \in [1..z]$.

4.1.1 Binary Search Trie. The first-child next-sibling representation binary maintains its nodes in three arrays. A node stores a pointer to one of its children, and a pointer to one of its siblings. It additionally stores the label (i.e., a character) of the edge to its parent. The trie binary takes $2z \lg z + z \lg \sigma$ bits when storing z nodes. Figure 7 gives an example. To navigate from a node v to its child with label $c \in \Sigma$, we take the first child of v and then sequentially scan all its next siblings until finding a node storing the character c. We propose three variants regarding the order of the siblings: In the first variant, called binary, we store the siblings in the order in which they are inserted. In the second variant, called $\text{binary}_{\text{mtf}}$, we apply a move-to-front heuristic: We store a new child as the leftmost child, shifting all other children to the right. Similarly, we make each child we visit the leftmost child. This heuristic is 2-competitive with the number of accesses needed by the optimal child ordering [68]. In the last variant, called binary_{s}, we sort the nodes according to the character on their incoming edge. This helps us to speed up unsuccessful searches, as we can stop when accessing a node whose incoming edge has a label larger than the label of the query.

4.1.2 Ternary Search Trie. The Ternary Search Trie [13], ternary, differs from binary in that a ternary node stores one more pointer to a sibling: A node of ternary stores a character, a pointer to one of its children, a pointer to one of its smaller siblings, and a pointer to one of its larger siblings. The trie ternary then takes $3z \lg z + z \lg \sigma$ bits when storing z nodes. Similarly to binary, we do not rearrange the nodes. To navigate from a node v to its child with label $c \in \Sigma$, we take the pointer to one of its children and then binary search for the sibling storing the character c: Given that we are at a node storing a character d, we

- take its smaller sibling if $c < d$,
- take its larger sibling if $c > d$, or otherwise
- descend to the current child, since $c = d$.

4.1.3 Space Analysis. Since we double the arrays when they become full, our peak memory usage happens during the last resizing, where we keep the old trie with m cells and the new trie with $2m$ cells in memory, where $z \in (m, 2m]$. The best and worst cases are $z = 2m$ and $z = m + 1$, respectively. Let m be the last size of a trie before doubling its size. For binary, we need $m \lg(m^2\sigma) + 2m \lg(4m^2\sigma) = 3m(\lg(m^2\sigma) + 4/3)$ bits of space, which is $(3/2)z(\lg(z^2\sigma) - 2/3)$ bits

[14]There are artificial texts like $\mathtt{a}^n$ for which we overestimate the number of factors.

Table 5. Upper and Lower Bound of the Maximum Memory Used during an LZ78/LZW Factorization with z Factors

	Space [bits]			Space [bytes]
Trie	Entry	Best Case	Upper Bound	Practice
binary	$\lg(z^2\sigma)$	$\frac{3}{2}z(\lg(z^2\sigma) - \frac{2}{3})$	$3z(\lg(z^2\sigma) + \frac{4}{3})$	$27m$
ternary	$\lg(z^3\sigma)$	$\frac{3}{2}z(\lg(z^3\sigma) - 1)$	$3z(\lg(z^3\sigma) + 2)$	$39m$
hash	$\lg(z^2\sigma)$	$\frac{3}{2\alpha}z(\lg(z^2\sigma) - \frac{2}{3})$	$\frac{6}{\alpha}z(\lg(z^2\sigma) + \frac{4}{3})$	$\frac{27}{\alpha}m$
cht	$\lg(2\alpha z\sigma)$	$\frac{3}{2\alpha}z(\lg(\alpha z\sigma) + \frac{8}{3})$	$\frac{3}{\alpha}z(\lg(\alpha z\sigma) + \frac{11}{3})$	$\frac{129}{8\alpha}m$
rolling	w	$\frac{3}{\alpha}z(w + \lg(z) - \frac{1}{3})$	$\frac{3}{\alpha}z(w + \lg(z) + \frac{2}{3})$	$\frac{36}{\alpha}m$

The size of a fingerprint is w bits. The best case and the upper bound are the values for $z = 2m$ and $z = m$, respectively, where m is the capacity of the respective trie (i.e., the maximum number of storable nodes without a reallocation of memory). The last column gives the maximum memory peak when setting $w = 64$, $\lg m = 32$, and $\lg \sigma = 8$ to constant, which is the setting in our practical evaluation in Section 4.4.

for the best case $z = 2m$. For ternary, we need $m\lg(m^3\sigma) + 2m\lg(8m^3\sigma) = 3m(\lg(m^3\sigma) + 2)$, which is $(3/2)z(\lg(z^3\sigma) - 1)$ bits for the best case. Table 5 puts these space bounds in relation to the following trie data structures.

4.2 LZ Tries with Hashing

A dictionary (such as a hash table) can simulate a trie by representing the trie nodes as elements in the dictionary: Given a trie edge (u, v) with label c, we use the unique key (ℓ, c) to store v, where ℓ is the label (factor index) of u; the root is assigned the label 0. This allows us to find and create nodes in the trie by simulating top-down-traversals. This trie implementation is called hash in the following: If an operation on the dictionary of hash can be carried out in $O(1)$ expected time, then we can carry out the whole factorization in $O(n)$ expected time.

We use a hash table as underlying implementation of the dictionary of hash. If the resize hint described at the beginning of Section 4 suggests that the next power of two is sufficient for storing all factors, then we set the maximum load factor α to 0.95 before enlarging the size (if necessary). We also implemented a hash table variant that changes its size to fit the provided hint. This variant then cannot use the fast bit mask (cf. Section 3.5.1) to simulate the operation mod $|H|$. Instead, it uses a practical alternative that scales the hash value by $|H|$ and divides this value by the largest possible hash value,[15] that is, $|H|h(K)/(\max_{K'} h(K'))$. We mark those hash table variants with a plus sign, for example, hash$_+$ is the respective variant of hash.

4.2.1 Space Analysis. Let m be the capacity of H, that is, the maximum number of elements H can store before a rehashing is needed (this is $\alpha|H|$ for open-addressing hash tables having $|H|$ cells). The hash table always stores trie nodes with labels that are at most m; this is an invariant, since we enlarge the hash table and consequently let m grow before inserting a node with label $m+1$. Therefore, the key of a node can be represented by a $\lceil\lg(m\sigma)\rceil$-bit integer by interpreting a key as a single integer with

$$[1..m] \times \Sigma \to [0..m\sigma - 1], (y, c) \mapsto (\sigma y + c). \tag{2}$$

Consequently, the hash table needs $(m/\alpha)\lceil\lg(m^2\sigma)\rceil$ bits of space. Since an entry of binary has the same space cost, the total space cost of hash is the same as that of binary divided by the maximum load factor α.

[15] http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints/.

4.2.2 Compact Hashing. We can further reduce the space requirements by switching to one of the compact hash tables described in Section 3. We call this approach cht, which works as follows: When enlarging the compact hash table, we choose a new bijective transform and rebuild the entire table with the new size and a newly chosen transform. We first choose a bijective transform f according to Section 3.5.3, adjusting the prime number $p \in [m\sigma..2m\sigma]$ of f such that f maps from $[1..p]$ to $[1..p]$ bijectively. Since f is a bijection, the function

$$\begin{aligned}[0..m\sigma-1] &\to [1..|H|] \times [0..\lfloor (2m\sigma-1)/|H| \rfloor] \\ i &\mapsto (h(K), q(K)) := (f(K) \bmod |H|, \lfloor f(K)/|H| \rfloor)\end{aligned}$$

is injective. Consequently, a quotient costs $\lg(2\alpha\sigma)$ bits, and therefore an entry uses $\lg(2\alpha\sigma m)$ bits in total. With cleary using two bit vectors of total size $2|H|$ for the displacement, we obtain that cht uses $|H|(2 + \lg(2\alpha\sigma m))$ bits. Given that m is the capacity of H before the last rehashing, our peak memory usage during this rehashing needs $|H| \lg(2\alpha\sigma m) + 2|H| \lg(4\alpha\sigma m) = (3m/\alpha)(\lg(\alpha\sigma m) + 11/3)$ bits. In the best case, this is $3z(\lg(\alpha z\sigma) + 8/3)/2\alpha$ bits for $z = 2m$. This yields the following result:

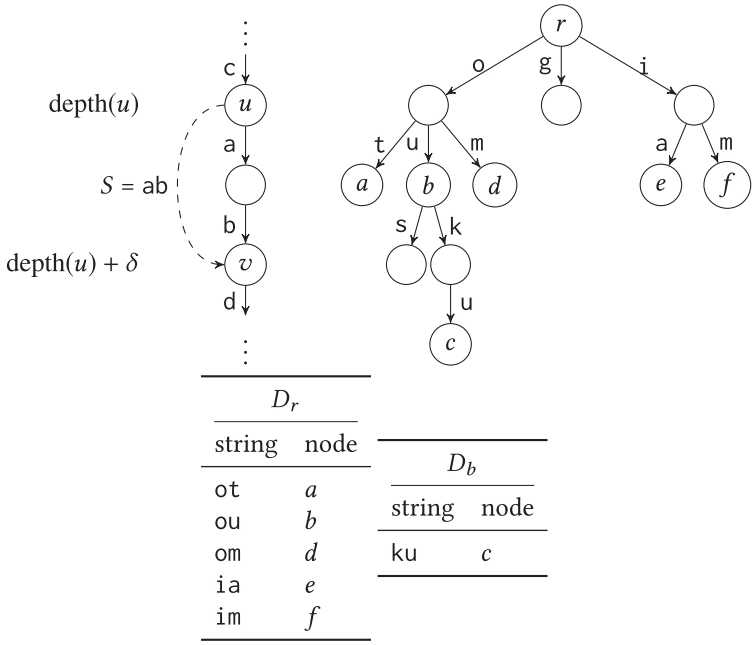
Theorem 4.1. *We can compute the LZ78 or LZW factorization online in $O(n/(1-\alpha^2))$ expected time using at most $z(3\lg(z\sigma\alpha) + 11)/\alpha$ bits of working space, for a user-defined constant $\alpha \in (0, 1)$.*

4.2.3 Rolling Hashing. Last, we present an alternative trie representation with hashing called rolling. The idea is to maintain the Karp-Rabin fingerprints [43] of all computed factors in a hash table such that the navigation in the trie is simulated by matching the fingerprint of a substring of the text with the fingerprints in the hash table. Given that the fingerprint of the substring $T[i..i+\ell-1]$ matches the fingerprint of a node u, we can compute the fingerprint of $T[i..i+\ell]$ to find the child of u that is connected to u by an edge with label $T[i+\ell]$. We compute a fingerprint with the randomized Karp-Rabin fingerprint family ID37 [50][16] with $ID37(T) = \sum_{i=1}^{|T|} h(T[i])37^{|T|-i} \bmod 2^w$, where w is the machine word size and h is a hash function that maps the alphabet uniformly to the range $[0..2^{32}-1]$. This rolling hash function discards the classic modulo operation with a prime number in favor of integer overflows due to performance reasons; this trick was already suggested by Gonnet and Baeza-Yates [34]. The LZ78/LZW factorization algorithm using rolling is a Monte Carlo algorithm, since the computation can produce a wrong factorization if the computed fingerprints of two different strings are the same (because the fingerprints *are* the hash table keys). With *wrong*, we mean that some computed referred indexes might differ from the correct ones, and thus the decompression might produce a different string. Given that rolling has a capacity to store m entries, an entry takes $\lg w + \lg m$ bits. On rehashing, we need $m(w + \lg m) + 2m(w + \lg(2m)) = 3m(w + \lg m + 2/3)$ bits. In the optimal case ($z = 2m$), this is $(3z/2)(w + \lg m - 1/3)$ bits.

4.3 Algorithm Engineering Aspects

We consider the following options for tweaking the presented trie data structures to improve time and/or space requirements: First, we propose jump pointers (Section 4.3.1), which help us traverse the LZ trie more quickly when parsing a long factor. Next, we propose two adaptations that use multiple LZ tries for the factorization. While the first variant (Section 4.3.2) is independent from the actual LZ trie implementation, the second variant (Section 4.3.3) only makes sense with the hash tries hash and cht. Finally, we propose a combination of these two variants (Section 4.3.4).

[16] https://github.com/lemire/rollinghashcpp.



D_r

string	node
ot	a
ou	b
om	d
ia	e
im	f

D_b

string	node
ku	c

Fig. 8. Jump pointers with $\delta = 2$. *Top Left*: We augment the node v whose depth is a multiple of δ with a dictionary that helps us to directly traverse the δ characters of the string S downwards to the descendant u. *Top Right:* An example trie. *Bottom:* The two dictionaries D_r and D_b of the example trie.

4.3.1 Jump Pointers. Similarly to the word-packed lookup techniques of compacted tries [11, 69], we can speed up the LZ trie traversal by augmenting certain nodes with dictionaries to jump over multiple heights: Given an integer parameter δ, we augment each node u having a depth of the form δd with a dictionary D_u mapping strings of length δ to its descendants at depth $\delta(d+1)$ whose heights are at least δ. Suppose that, during the factorization, we look for the factor having the longest common prefix with the remaining text by traversing the LZ trie. Whenever we reach a node u having a depth of the form δd, we query D_u for the string consisting of the next δ text characters. If this string exists in D_u, then it directly leads us to a descendent of u at depth $\delta(d+1)$, so we have processed δ characters in one step. Otherwise, we proceed character-wise from u. See Figure 8 for an illustration.

We can maintain the jump pointer dictionaries D_u as we build the trie: Suppose that we did an LZ trie traversal to add a new leaf ℓ, where we visited the last 2δ nodes including the leaf ℓ without using a pointer of one of the dictionaries (i.e., we traversed the last 2δ nodes character-wise). This means that the ancestor u of ℓ at distance 2δ from ℓ does not have a jump pointer to the ancestor v of ℓ at distance δ, which can only happen because v was of height less than δ. However, the new leaf ℓ has made v of height δ and thus now we can add a jump pointer from u to v. By keeping the last 2δ characters read from the text and the two last visited nodes whose depths are multiples of δ in memory, we can easily augment the dictionary D_u with v.

A hash table equipped with a hash function treating an input string as an integer array (by means of word-packing) can hash a string of length δ in $O(\delta(\lg\sigma)/w)$ expected time and check two strings for equality within the same time. Implementing D_u as such a hash table results in $O(\delta(\lg\sigma)/w)$ expected time for retrieving a pointer.

The number of pointers is $O(z/\delta)$, since we only add a node into a jump pointer dictionary when (a) its depth is a multiple of δ and (b) its height is at least δ. This ensures that every node v inserted in some D_u has at least $\delta - 1$ descendants that are not inserted in any dictionary. This gives us an overhead of $O((z/\delta)w)$ bits. Matching a string of length ℓ with the LZ trie then costs $O(\ell/\delta + \delta)$ time, since we need to compare less than 2δ edges without using a jump pointer. This gives us $O(n/\delta + \delta z)$ time for the entire factorization. If we implement the dictionaries of the jump pointers with a hash table and set $\delta := O(w/\lg\sigma)$, then we pay $O(((z\lg\sigma)/w)w) = O(z\lg\sigma)$ bits of additional space for storing the pointers, but can query for a jump pointer in constant expected time and therefore conduct the whole factorization in $O(n\lg\sigma/w + wz/\lg\sigma)$ time, which is $o(n)$ for small alphabets and compressible texts with $z = o(n\lg\sigma/w)$.

Especially for trie representations with (practically) slow lookup times like the compact hash tables, this technique can help boost the factorization. On our datasets, we could observe a benefit with $\delta = 8$ (so the keys fit into a 64-bit machine word) when storing only jump pointers from the root. We could not observe a benefit for deeper nodes, since it is less likely to query those dictionaries, and it is even less likely to find the desired substrings.

4.3.2 Key-split Variant. The idea is to represent the LZ trie with multiple trie instances with different index ranges (for binary and ternary of Section 4.1) or different key bit widths (for hash and cht of Section 4.2). For the former group, we store an array A of pointers to trie instances such that $A[k]$ stores the trie nodes whose labels are in the range $[2^{k-1}..2^k - 1]$ for $k \geq 1$. For the latter group, this array A stores hash tables for each key bit width such that the kth hash table $A[k]$ manages keys whose binary representation uses exactly k bits. All solutions start with the array A where only $A[1]$ points to an allocated trie while the rest of pointers are null. The number of pointers is $\lg z$ for the deterministic tries, and $\lg(z\sigma)$ for the hash tries. Starting with $k = 1$, whenever the trie $A[k]$ becomes full, we allocate $A[k+1]$ with twice the number of cells of $A[k]$, that is, $|A[k]| = 2|A[k-1]| = \sum_{j=1}^{k-1} |A[j]| + 1$. On a global perspective, such an allocation increases the total number of cells from m to $2m + 1$.

On the upside, this technique can yield a speedup, since we omit the resize operations. cht can profit from this technique with respect to memory consumption, since each compact hash table can tailor its quotient bit width $\lceil \lg|q| \rceil$ individually according to the key bit width, whereas trie data structures like binary represent a factor index implicitly as an array index or explicitly (as a content in the arrays for the first child or next sibling) needing $\lg z$ bits in general.

On the downside, this method needs an additional pointer indirection from A to delegate a call of insert or lookup with the key K (respectively, factor index K for the non-hash-based tries) to the trie $A[\lceil \lg K \rceil]$, which can slow down the computation. Here, $\lceil \lg K \rceil$ is computable in constant time on most architectures (and in theory by using small precomputed tables).

4.3.3 Key/Value-split Variant. When working with hash or cht, another possibility is to maintain multiple hash tables for each possible key and value bit widths, resulting in a matrix A of hash tables such that the hash table $A[k][v]$ stores keys and values with bit width k and v, respectively. A has $\lceil \lg z\sigma \rceil$ rows and $\lceil \lg z \rceil$ columns, where the vth column stores all hash tables with value bit width v.

A disadvantage is that, to find the value of a key K, we may need to query all (k, v)th hash tables with $k = \lceil \lg K \rceil$. Hence, the expected time for lookup becomes $O(\lg z)$. Nevertheless, we can bound a root-to-leaf traversal on a path of length ℓ in the LZ trie to $O(\ell + \lg z)$ expected time with the following amortization argument: While descending from the root to a leaf, the labels of the visited nodes on the path are in increasing order, meaning that the returned values of lookup are in increasing order. Therefore, whenever we query the (k, v)th hash table during a traversal, we will never query a hash table with key and value bit widths smaller than k and v, respectively.

In total, we selected at most $\ell + \lceil \lg z \rceil$ hash tables. This gives us a total expected running time of $O(n + z \lg z)$ for the whole factorization.

On the upside, we can improve the memory footprint, since each hash table can store the values bit-optimally. On the downside, we maintain more hash tables that might be far from being full. This could be mitigated with the sparse hash table layout or by storing only entries with high key and value bits in the matrix while keeping small entries in a single hash table (in particular, the (k, v)th hash table with $v < k - \lg \sigma$ is always empty). We present next an engineered version aiming to combine the best from this representation with previous ones.

4.3.4 Combined Four-tier Approach. We implement the compact hash table grp of Section 3 as a four-tier trie data structure, where

(1) the children of the root are stored in a plain array,
(2) the keys whose bit widths are at most $8 + \lceil \lg \sigma \rceil$ are stored in a single grp table,
(3) the remaining keys, subtracting $2^{8+\lceil \lg \sigma \rceil}$ from each, whose bit widths are below an integer parameter $\beta \geq 0$ are put in an array of grp tables for different key bit widths, as described in Section 4.3.2, and
(4) all other keys are put in a matrix of grp tables with different key and value bit widths, as described in Section 4.3.3.

We call this variant grp_β in the following experiments.

4.4 Experiments

We implemented our LZ trie representations in the C++ framework tudocomp [22].[17] The framework provides the implementation of an LZ78 and an LZW compressor. Both compressors are parameterized with an LZ trie and an encoder. The encoder is a function that takes the output of the factorization and generates the final binary output. We selected the encoder `bit`, which produces the encoding as described in Section 2.1.3.

The LZ78 and LZW compressors are independent of the LZ trie implementation, that is, all the trie data structures described in the previous sections can be easily plugged into the LZW or LZ78 compressors. Moreover, hash and cht work with any hash table or compact hash table, respectively. Here, we used a simple linear-probing hash table for hash, and the compact hash table cleary from Section 3 for cht. We additionally add unordered (the C++17 standard implementation `std::unordered_map` of a hash table) and grp as alternative implementations of hash and cht, respectively. Finally, we incorporated the Judy array judy into tudocomp, which is advertised to be optimized for avoiding CPU cache misses (cf. References [41, 52] for evaluations). We added a lightweight wrapper around it to provide the same interface for all tries.[18]

Our implementation of cht uses cleary_S as the default. Its variants using an array for each different key bit width (cf. Section 4.3.2) or a matrix for each different key and value bit width (cf. Section 4.3.3) have subscripts "k" and "kv," respectively. We also add grp and grp_β parameterized with β (cf. Section 4.3.4) as alternative implementations of cht.

We represent the indices of the factors with 32-bit integers. Hence, the values stored by hash, rolling, and cht are 32-bit integers. Since we use a byte alphabet representing a character in 8 bits, the keys of hash and cht are 40-bit integers. The fingerprints of rolling are 64-bit integers. For all variants working with open-addressing hash tables, we initially set α to 0.3.

[17] The source code of our implementations is freely available at https://github.com/tudocomp.
[18] Unfortunately, the cedar trie [71] evaluated by Fischer and Köppl [29] fails to handle our datasets, which are many times larger than the 200 MiB datasets studied there.

4.4.1 Structure of the Benchmarks. We study the time and space tradeoffs of all the aforementioned trie implementations during the factorization of all the datasets described in Table 3. To ease the visualization, we put each trie in one of three groups. The first group comprises those tries that do not show interesting characteristics and are therefore evaluated only in Section 4.4.2. The LZ78 and LZW factorization benchmark results of all the other tries are visualized in Figures 10 and 11 and in Figures 12 and 13, respectively. We provide a joint evaluation in Figures 10 and 12, and then separate time-efficient (Section 4.4.3) from memory-efficient (Section 4.4.4) implementations in Figures 11 and 13. In each plot, a vertical dashed line at eight bits marks the size of a single character such that every approach to the left of this line uses less memory than the space of the input data.

4.4.2 Preliminary Evaluation. A preliminary benchmark in Figure 9 allows us to discard some alternatives upfront.

First, we observe that binary_s is always less performant than binary, while binary_mtf is sometimes a little bit faster and sometimes a little bit slower than binary. An empirical conclusion is that applying sorting seems not to pay off. Indeed, we see a similar behavior when using a hash table with a sorting technique like ordered hash tables [1] or Robin-Hood hashing [16] for hash. We also apply the trie split techniques introduced in Section 4.3.2 and 4.3.3 only to the compact hash tables, as we do not see any benefits for the other trie variants. For instance, we observe that the variant binary_k using multiple binary tries for different key bit widths (Section 4.3.2) is inferior to the plain binary representation.

Second, unordered always uses much more space than hash, to the extent that we could not benchmark unordered on COMMONCRAWL without running out of memory (therefore, there is no data point available for that instance). Except for FIBONACCI, hash is also always faster that unordered.

Last, we study the jump pointer technique applied to binary_k and grp_0, tagged with the names $\text{binary}_\text{k}^\text{J}$ and grp_0^J, respectively. We see only a slight time improvement, which is always tied with an increase in the memory requirement for storing these pointers.

Therefore, to simplify the following benchmarks, we omit the variants of binary (such as binary_s) and unordered, as well as the jump pointer technique of Section 4.3.1.

4.4.3 Time-efficient Tries. From Figures 10 and 12, we observe that rolling, followed by its variant rolling_+, is the most memory-hungry option, but also in multiple cases the fastest (for ENGLISH, PROTEINS, and WIKIPEDIA). Remember that hash_+ and rolling_+ are variants of hash and rolling, respectively, following our resize hint as explained in Section 4.2. The size of its fingerprints is a tradeoff between space and the probability of a correct output. When space is an issue, rolling with 64-bit fingerprints is no match for more space-saving trie data structures. hash follows rolling and its variants in terms of memory consumption. With 40-bit keys, it uses less memory than rolling, but it is slightly slower on most datasets (an exception is COMMONCRAWL). Depending on the quality of the resize hint, the variants hash_+ and rolling_+ take 50% to 100% of the size of hash and rolling, respectively. hash_+ and rolling_+ are mostly always slower than their respective standard variants, sometimes slower than the deterministic data structures ternary and binary (cf. FIBONACCI, XML, or GUTENBERG). binary's speed excels in texts with very small alphabets (cf. DNA, FIBONACCI), while ternary usually outperforms binary on larger ones (cf. COMMONCRAWL, WIKIPEDIA, and ENGLISH). However, binary is always smaller than ternary. The third-party data structure judy is always outperformed by ternary or/and binary. Only cht and its variants can compete with binary in terms of space, but it is significantly slower than all aforementioned implementations.

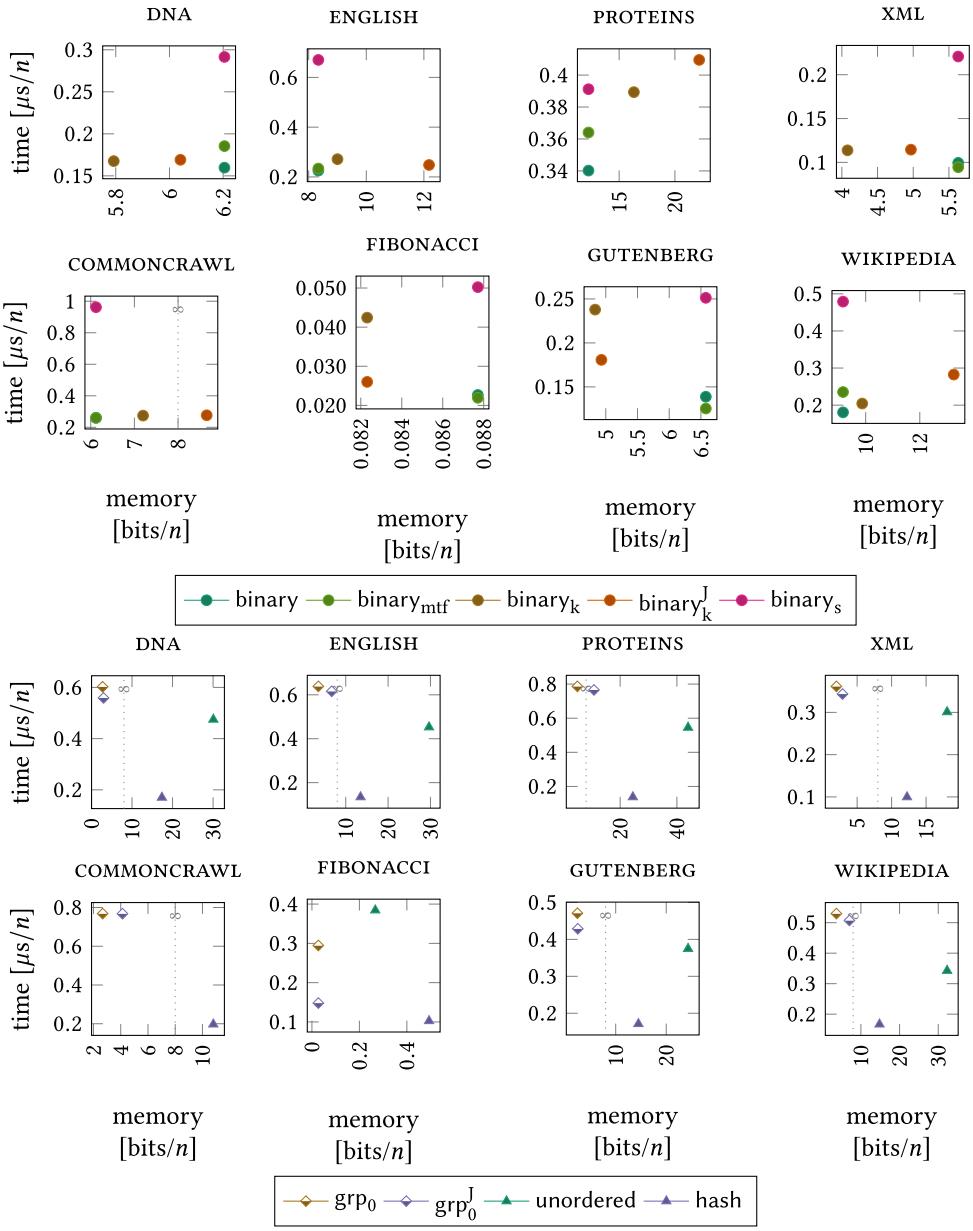


Fig. 9. Evaluation of the LZ78 factorization with the trie implementations studied in Section 4.4.2, namely, $\text{binary}_{\text{mtf}}$ (binary using MTF-encoding), binary_{k} (binary using the technique of Section 4.3.2), $\text{binary}_{\text{k}}^{\text{J}}$ (binary_{k} combined with the jump pointer technique of Section 4.3.1), binary_{s} (binary with sorting), grp_0^{J} (jump pointers applied to grp_0 defined in Section 4.3.4 for $\beta = 0$), and unordered (hash with the C++ STL hash table), which we omit in the following evaluations.

4.4.4 Space-efficient Tries. We evaluated cht with cleary as the default hash table and with grp (cf. Section 3.4) and grp_β (cf. Section 4.3.4). The variants of cht described in Section 4.3.2 and Section 4.3.3 are denoted cht_k and cht_{kv}, respectively. From the results shown in Figures 11 and 13, we can conclude that cht_{kv} is the most memory-efficient variant, but most of the times the slowest

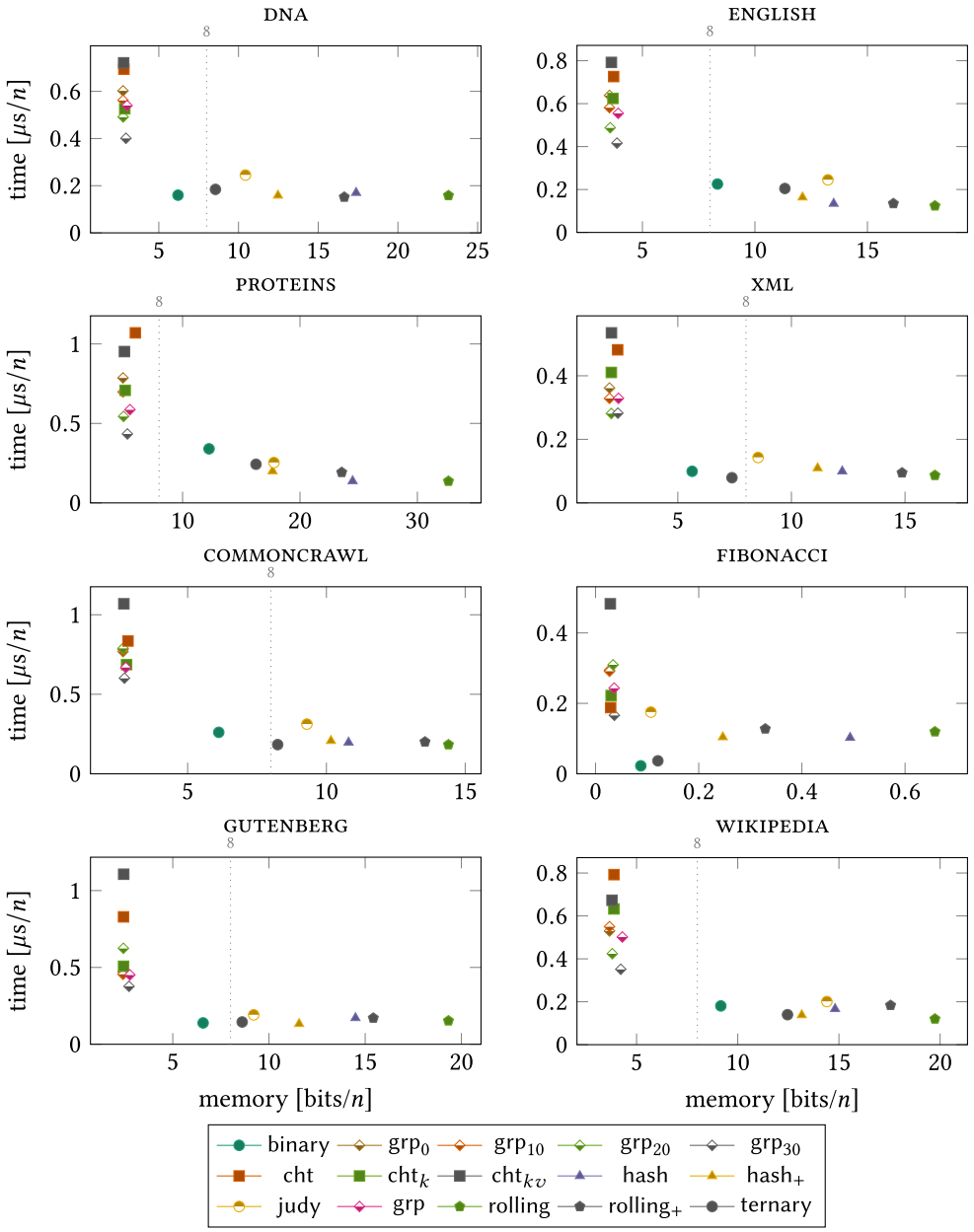


Fig. 10. Juxtaposition of all LZ trie implementations for the LZ78 factorization.

(an exception is WIKIPEDIA). cht_k is often faster and more memory-efficient than cht (using just a single cleary hash table). When comparing the alternative hash tables cleary and grp for cht, we see that grp is most of the times faster, but not a winner with respect to the space. Here, we see an improvement when using grp_β, which is grp with the technique of Section 4.3.4.

For grp_β, we conducted our experiments with $\beta = \{0, 10, 20, 30\}$, where $\beta = 0$ disables the key-split array of Section 4.3.2, while $\beta = 30$ disables the key-value-split matrix of Section 4.3.3 for our

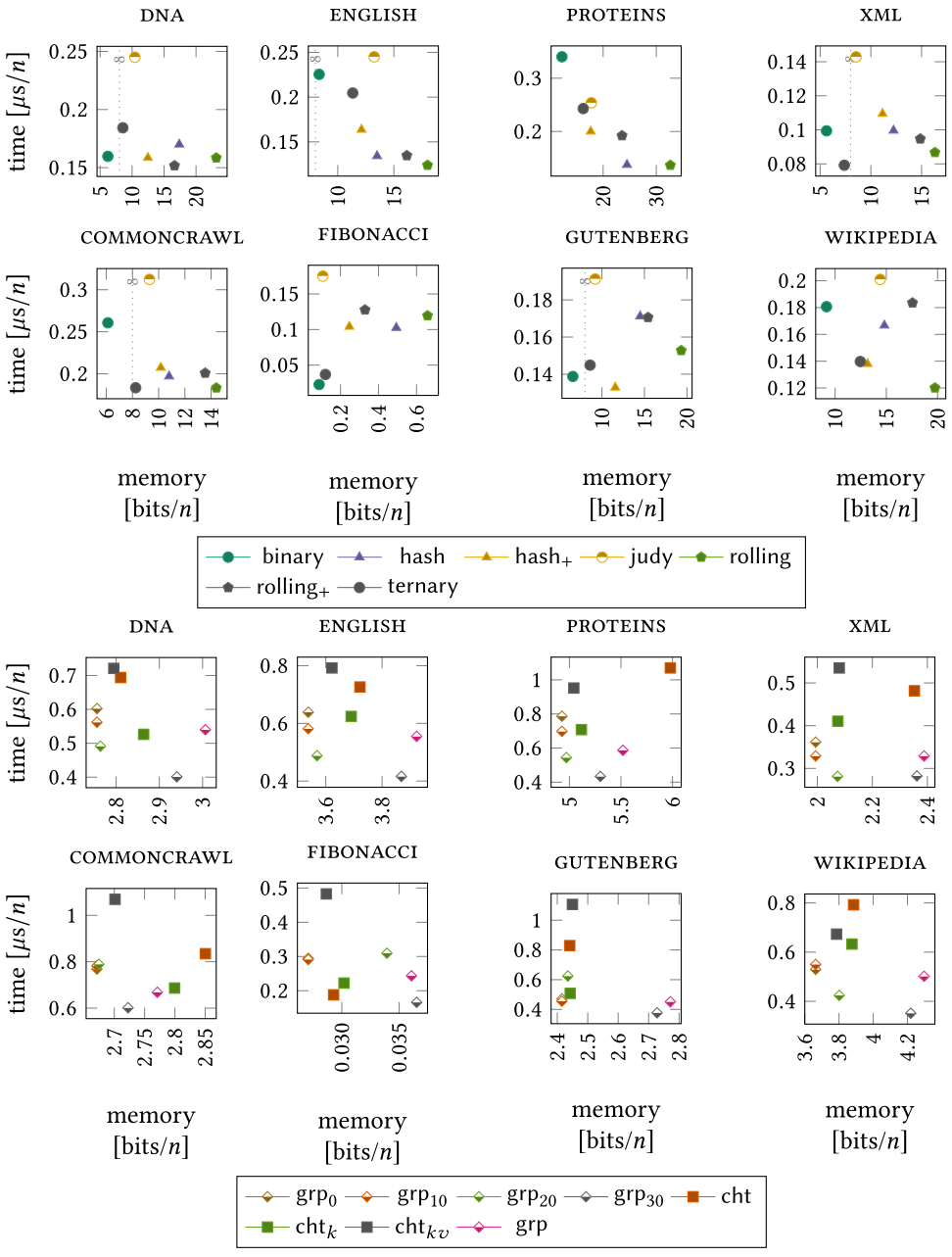


Fig. 11. Evaluation of our LZ trie implementations for the LZ78 factorization.

datasets with $\lceil \lg z \rceil \leq 32$ and $\lceil \lg \sigma \rceil = 8$. The evaluation shows that the larger the parameter β, the faster and less memory-efficient the data structure becomes. Overall, grp_β parameterized by β forms most of the time a Pareto-front dominating the cleary variants (cht, cht_k, and cht_{kv}) and grp (an exception is FIBONACCI and COMMONCRAWL, respectively).

4.4.5 Evaluation of rolling. Selecting a strong rolling hash function for rolling is crucial to avoid the possibility of a hash collision. Hash collisions happened during the experiments when using

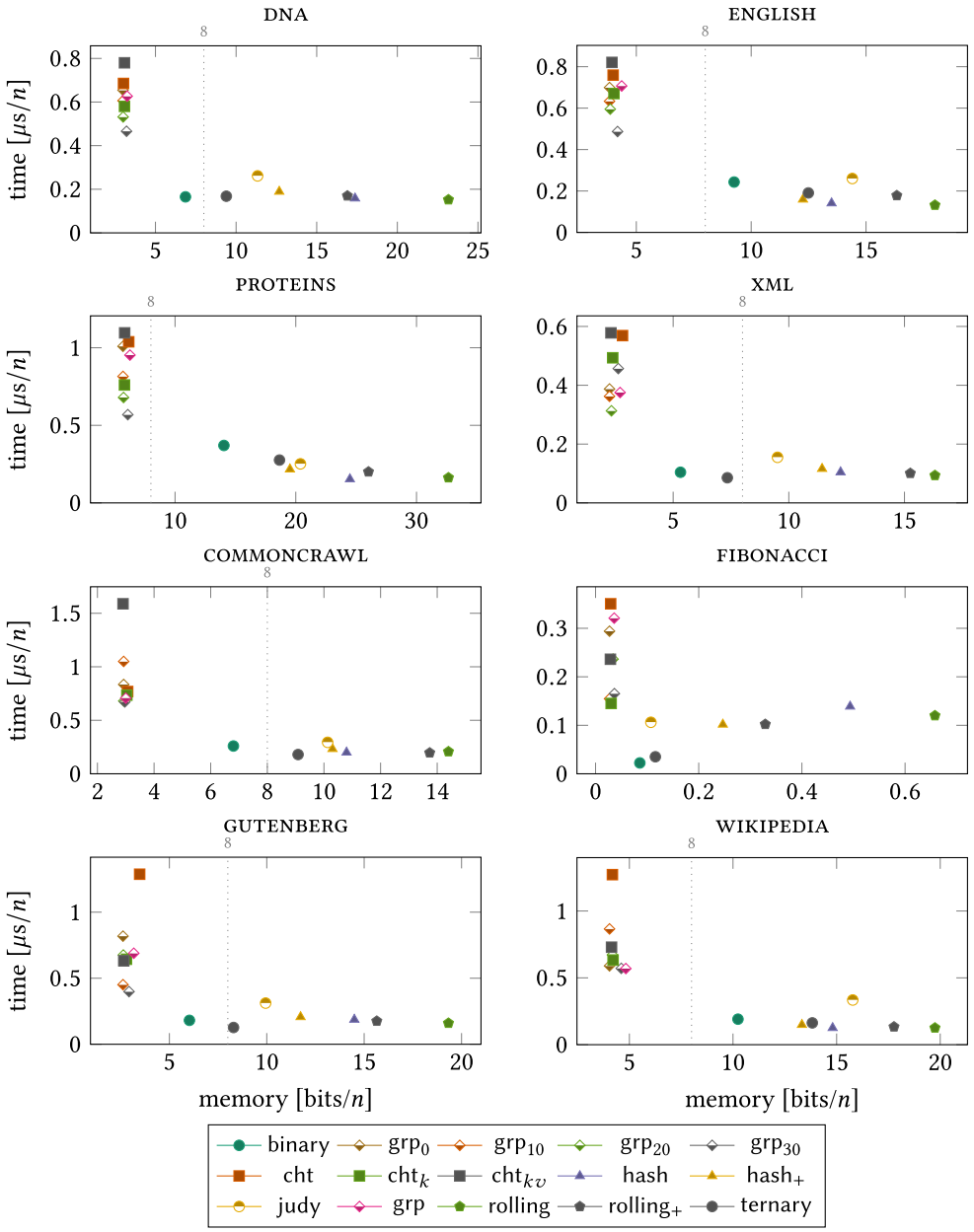


Fig. 12. Juxtaposition of all LZ trie implementations for the LZW factorization.

a simple rolling hash function such as $h(T) = \sum_{i=1}^{|T|}(T[i]-1)(\sigma+1)^{|T|-i} \mod 2^w$, where w is the word size and the modulo by the maximum value 2^w surrogates the integer overflow.

The likelihood that the fingerprints of two different substrings match is anti-correlated to the number of bits used for storing the fingerprint if we assume that the used rolling hash function distributes uniformly. This means that the domain of the Karp-Rabin fingerprints can be made large enough to be robust against collisions when hashing large texts. In our case, we used 64-bit

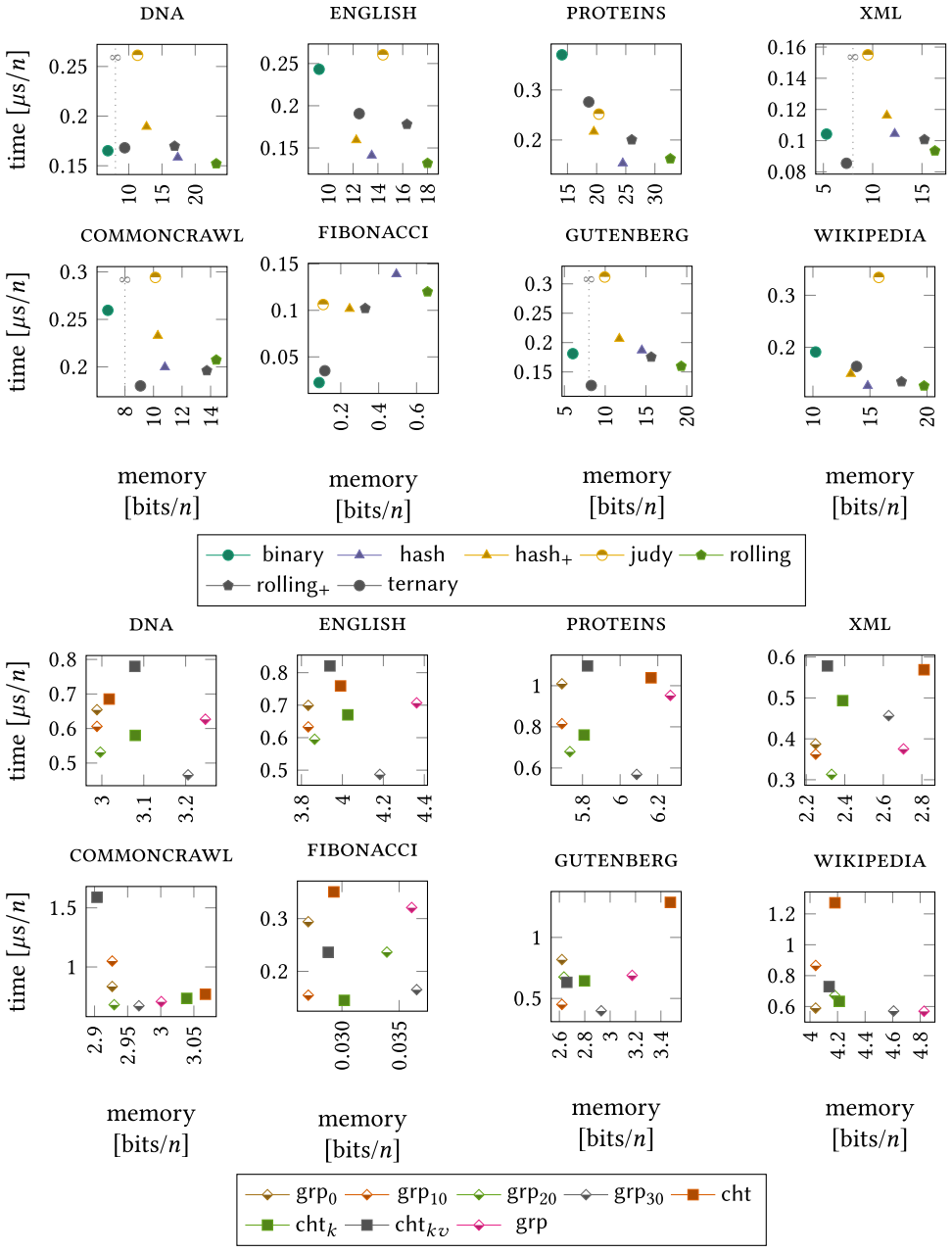


Fig. 13. Evaluation of our LZ trie implementations for the LZW factorization.

fingerprints because, unlike 32-bit and 40-bit fingerprints, the factorization produced by rolling is correct for all test instances with the rolling hash function *ID37*. Nevertheless, this bit width can be considered as too weak for processing massive datasets: Even if the rolling hash function distributes uniformly, the probability of a collision is $1/2^{64}$. Although this number is very small, processing 10^9 datasets, each 200 MiB large, would give a collision probability of roughly 1%. This probability can be reduced by enlarging the bit width, and hence improving the correctness

Table 6. Performance Comparison of 64-bit and 128-bit Fingerprints of rolling with Its Variant rolling$_+$ when Computing the LZ78 and LZW Factorization

	LZ78		LZW			LZ78		LZW	
dataset / trie	time $[\mu s/n]$	space $\frac{\text{bits}}{n}$	time $[\mu s/n]$	space $\frac{\text{bits}}{n}$	dataset / trie	time $[\mu s/n]$	space $\frac{\text{bits}}{n}$	time $[\mu s/n]$	space $\frac{\text{bits}}{n}$
DNA					COMMONCRAWL				
rolling	0.16	23.17	0.15	23.17	rolling	0.18	14.40	0.21	14.40
rolling$_+$	0.15	16.63	0.17	16.90	rolling$_+$	0.20	13.55	0.20	13.74
rolling$_{128}$	0.16	38.62	0.17	38.62	rolling$_{128}$	0.21	24.00	0.20	24.00
rolling$_{128+}$	0.17	27.72	0.19	28.17	rolling$_{128+}$	0.21	22.59	0.22	22.89
ENGLISH					FIBONACCI				
rolling	0.12	18.00	0.13	18.00	rolling	0.12	0.66	0.12	0.66
rolling$_+$	0.13	16.15	0.18	16.34	rolling$_+$	0.13	0.33	0.10	0.33
rolling$_{128}$	0.17	30.00	0.15	30.00	rolling$_{128}$	0.11	1.10	0.13	1.10
rolling$_{128+}$	0.16	26.92	0.17	27.23	rolling$_{128+}$	0.11	0.55	0.12	0.55
PROTEINS					GUTENBERG				
rolling	0.14	32.65	0.16	32.65	rolling	0.15	19.33	0.16	19.33
rolling$_+$	0.19	23.56	0.20	26.02	rolling$_+$	0.17	15.42	0.18	15.65
rolling$_{128}$	0.15	54.41	0.18	54.41	rolling$_{128}$	0.13	32.21	0.18	32.21
rolling$_{128+}$	0.23	39.26	0.34	43.37	rolling$_{128+}$	0.14	25.70	0.20	26.08
XML					WIKIPEDIA				
rolling	0.09	16.32	0.09	16.32	rolling	0.12	19.74	0.13	19.74
rolling$_+$	0.09	14.87	0.10	15.25	rolling$_+$	0.18	17.55	0.13	17.77
rolling$_{128}$	0.09	27.19	0.13	27.19	rolling$_{128}$	0.12	32.91	0.13	32.91
rolling$_{128+}$	0.10	24.78	0.12	25.42	rolling$_{128+}$	0.18	29.25	0.21	29.61

probability by sacrificing working space. We reran our experiments with 64-bit and 128-bit fingerprints, and we measured time and space usage in Table 6. There, we observe that switching to a greater bit width slightly degrades the running time, but severely degrades the space usage. On all instances, the + variants (cf. Section 4.2) use less memory than the standard variants; on FIBONACCI, rolling$_{128+}$ uses even less memory than the default rolling.

Another option to sustain a correct computation is to check the output of the factorization. This check can be done by reconstructing the text with the output and the LZ trie built. However, a compression with rolling combined with a decompression step takes more time than other approaches such as hash or binary. Hence, a Las Vegas algorithm based on rolling is practically not interesting.

4.4.6 Compression Ratio. Finally, we compare the compression ratios obtained with the classic encoding described in Section 2.1.3 with the Unix tool `compress`.[19] This tool uses a modified `LZW` coding and has the upper bound of 2^{16} on the LZ trie nodes. Whenever this upper bound is reached, it no longer inserts new nodes into the LZ trie; instead, it clears the trie based on a heuristic. Despite being fast and memory-friendly, its compression ratios are inferior to the classic encoding working without such kind of restriction, as can be seen in Figure 14.

In what follows, we study an alternative coding of the LZ78 factors based on the Bonsai tables (which are a sub-class of compact hash tables).

[19]We used the implementation `ncompress` version 4.2.4.6.

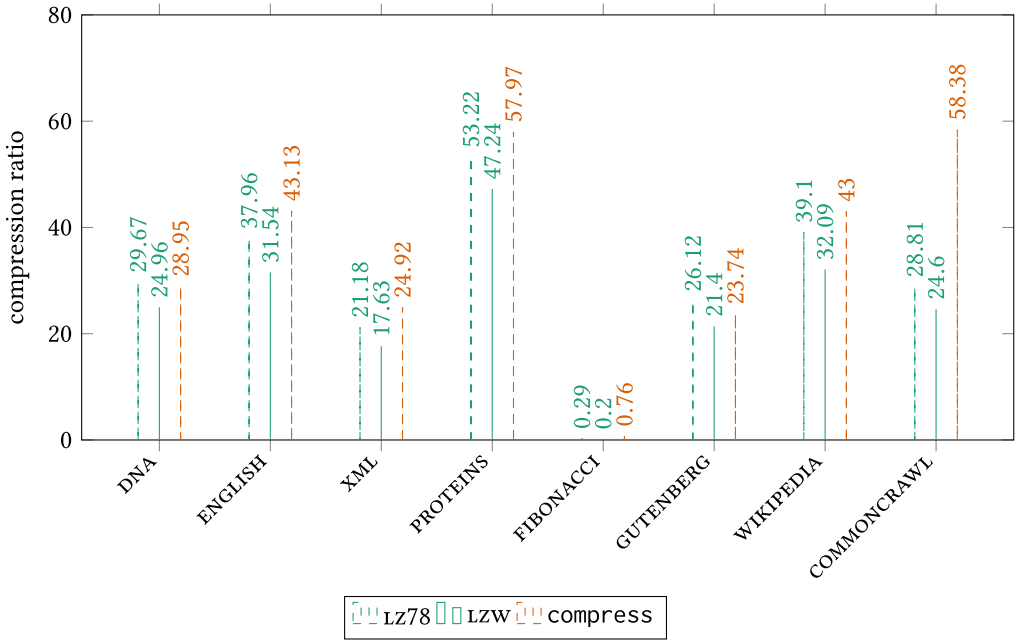
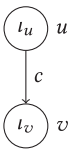


Fig. 14. Compression ratios of the classic coding of LZ78 and LZW described in Section 2.1.3, compared with the Unix tool `compress` with its best compression.

5 LZ78 COMPUTATION WITH BONSAI TRIES

An application for the Bonsai tables introduced in Section 3 is the Bonsai trie introduced by Darragh et al. [21]. A *Bonsai trie* is an ID dictionary (defined in Section 3) with $\mathcal{K} := [1..\rho]\times\Sigma$,[20] where $[1..\rho]$ is the domain of all identifiers of the ID dictionary. A node of the trie is represented by an identifier. Given that the root is present as an entry in the hash table, every other node v with identifier ι_v is represented by the key (ι_u, c) such that $\text{lookup}(\iota_u, c) = \iota_v$, where ι_u is the identifier of the parent u of v, and c is the character labeling the edge connecting u with v. The methods insert, lookup, and key can be directly translated to trie methods:

- $\text{insert}(\iota_u, c)$ creates a leaf v as a child of the node with identifier ι_u, connecting them by an edge labeled with $c \in \Sigma$, and returns the identifier ι_v of v.
- $\text{lookup}(\iota_u, c)$ returns the identifier of the child of the node u with identifier ι_u that is connected to u with an edge labeled with $c \in \Sigma$, or $\perp$ if such a child does not exist.
- $\text{key}(\iota_v)$ returns the key (ι_u, c) of a node v with identifier ι_v, where ι_u is the identifier of the parent u of v connected with v by an edge with label c.



For instance, we can insert a new child of a node with identifier ι with connecting edge label c by inserting (ι, c) into H. We additionally need the function root(), returning the identifier of the root in the trie.

With the Bonsai trie, we can create the LZ trie in a top-down manner. However, performing DFS or BFS traversals on this trie implementation is not efficient: Although we can move with lookup

[20] We can represent elements of $\mathcal{K}$ by integers, transforming a pair $(i, c) \in \mathcal{K}$ into an integer $i\sigma + (c-1) \in [1..(\rho+1)\sigma)$, similarly to Equation 2 of Section 4.2.1.

Table 7. Overview of the Results of Section 5

Theorem	Time	Internal Space	External Space
Theorem 5.1	$O(n)$	$O(n \lg \sigma / \log_\sigma n)$	streaming
Theorem 5.2	$O(n \lg z)$	$O(z \lg \sigma)$	streaming, overwritable
Theorem 5.3	$O(n)$	$O(z \lg \sigma)$	sequential reads/writes
Theorem 5.4	$O(n \lg \sigma)$	$O(z \lg \sigma)$	streaming

Times are the *expected* time for the LZ78 factorization. Internal space is measured in bits. Each solution has an external memory working space of $z \lg \lceil \rho \rceil$ bits, which can be used only for streaming the final output, rewriting the streamed output, or used for reading and writing.

to a specific child, enumerating all children can only be done by trying out all characters, which is costly for non-constant alphabets.

Our following solutions for the LZ78 factorization use a Bonsai table as an ID dictionary taking $O(t_{\text{Bonsai}})$ time for any of the above operations (see Table 4 for concrete implementations) and are based on a technique similar to hash (Section 4.2) along with its simplifying assumptions on the hash function (Section 3.5.3): A usual analysis assumes that the hash function is chosen independently of the set of keys to hash. For the Bonsai trie, however, a key (ι, c) to hash depends on the identifier ι, which in turn depends on the hash function for all known Bonsai tables. So, at least in principle, the typical assumptions to prove 2-independence do not hold, even if we change our bijective transform to the standard $h(K) = (a_0 + a_1 K) \mod p$ for randomly chosen a_0 and a_1.

Assuming that we can read/write from/to disk sequentially with constant time per machine word, we give an overview of our following results, which we additionally have collected in Table 7. In Section 5.1, we start with a solution running in $O(n)$ expected time while using $O(n \lg \sigma / \log_\sigma n)$ bits of working space. The space can be lowered to $O(z \lg \sigma)$ bits by running the algorithm $O(\lg z)$ times, giving $O(n \lg z)$ expected time. We improve this time bound to $O(n)$ expected time in Section 5.2 with the help of external memory as working space. Finally, we can avoid working with the external memory with the solution of Section 5.3, running in $O(n \lg \sigma)$ time. The output of all solutions consists of the Bonsai trie representing the LZ trie, and a list of the identifiers of the LZ nodes sorted by creation time.

5.1 A Bonsai Table of Fixed Size

We start with a solution, called fix, which manages a Bonsai trie whose underlying Bonsai table is sufficiently large such that it can guarantee no rehashing while populating the trie with *all* LZ trie nodes. To this end, we set the upper bound on the number of nodes m to the smallest number with $(m-1)(\log_\sigma(m) - 3) \geq n$; recall Lemma 2.1. Thus, $m = \Theta(n/\log_\sigma n)$. Further, we use an array $L[1..z]$ to store in $L[x]$ the position in H where the LZ trie node of the xth factor is stored. Each entry of L takes $\lceil \lg \rho_m \rceil$ bits, where $[1..\rho_m]$ is the domain of identifiers for a Bonsai table that can store m entries. This array is generated on external memory in a streaming fashion.

To compute the next factor $F_x = T[i..i + \ell - 1]$, we start from the trie root with identifier $\iota_0 :=$ root() and recursively lookup the identifiers $\iota_k = (\iota_{k-1}, T[i + k - 1])$ with $k \geq 1$ until lookup$(\iota_{\ell-1}, T[i + \ell - 1]) = \perp$ does not exist in the trie. At this point, we

- insert the node representing the factor $F_x = F_y T[i + \ell - 1]$ with $\iota_\ell :=$ insert$(\iota_{\ell-1}, T[i + \ell - 1])$, where F_y is represented by the node with identifier $\iota_{\ell-1}$, subsequently
- write the next value $L[x] \leftarrow \iota_\ell$ to disk, and
- continue with $T[i + \ell..n]$.

The working space is dominated by the large allocated Bonsai table. After we have computed the factorization, we serialize the Bonsai trie such that it can be later reloaded into main memory for decompression.

To improve the space needed for the serialization, we first write a bit vector B of length ρ_m storing at $B[\iota]$ whether there is a key in H having identifier ι. Hence, B has exactly z ones, and requires ρ_m bits. Having B, we only need to write the (non-empty) z entries of H to the compressed file, yielding a final compressed file size of $z(\lg \rho_m + \lg \sigma) + \rho_m$ bits. We call this representation of the factorization the *Bonsai coding*.

If we select elias as the underlying Bonsai table, then we have $\rho_m = O(m)$ and $|\mathcal{K}| = \rho_m\sigma = O(m\sigma)$. Hence, our Bonsai trie storing $m = \Theta(n/\log_\sigma n)$ nodes needs $O(\mathcal{B}(|\mathcal{K}|, z)) = O(n \lg \sigma / \log_\sigma n)$ bits of space according to Table 4, where we used that $\mathcal{B}(x, y) = \Theta(y \lg((x+y)/y))$ for $y \le x$. With this space bound, we obtain the following theorem:

Theorem 5.1. *We can compute the LZ78 factorization in $O(nt_{\text{Bonsai}})$ time within $O(n \lg \sigma / \log_\sigma n)$ bits of main memory working space, streaming the Bonsai coding to disk.*

To further reduce the output size, we can postprocess L to use only $z \lg z$ bits instead of $z\lceil \lg \rho_m \rceil$ bits of disk space. For that, we scan L and replace $L[x]$ by $B.\operatorname{rank}_1(L[x])$. An original value can be recovered with $B.\operatorname{select}_1(L[x])$. Both operations work in constant time if B is equipped with a rank- and select-support.

5.1.1 Decompression. We can decompress in streaming mode while using memory space only for the Bonsai trie. For that, we only need to deserialize the Bonsai trie, keeping L still on disk. Having the Bonsai trie representing the LZ trie into memory, we read the consecutive entries of $L[1..z]$ in streaming mode, starting with $L[1]$. Suppose we read $L[x] = \iota_\ell$ for an identifier ι_ℓ of a node with an arbitrary depth ℓ. Then, we know that there is a path $(\iota_0, \ldots, \iota_\ell)$ from the root with identifier ι_0 to a node with identifier ι_ℓ, With $(\iota_{\ell-1}, c_{\ell-1}) \leftarrow \text{key}(\iota_\ell)$ and $(\iota_{\ell-k-1}, c_{\ell-k-1}) \leftarrow \text{key}(\iota_{\ell-k})$ for $k \in [1..\ell-1]$, we climb up from the node with identifier ι_ℓ until reaching the root with $(\iota_0, c_0) \leftarrow \text{key}(\iota_1)$. Then, we append the string $c_0c_1 \ldots c_{\ell-2}c_{\ell-1}$ to the decompressed text in streaming mode. For that, we use a stack that we fill with up to $\max_{x\in[1..z]} |F_x|$ characters. The stack may require up to $z \lg \sigma$ bits, but this is still within our main memory budget.[21] (Alternatively, we can read L backwards and generate T from the end to the beginning at the expense of increased I/Os.) Depending on the size of the serialized Bonsai trie, the decompression may require less main memory working space than the classical method (cf. Section 2.1), where each factor is represented by a character and a reference to an earlier factor index. There, the decompression is done in linear time by keeping S and R in memory, which requires $z \lg \sigma + z \lg z$ bits.

As a side note, the Bonsai coding permits retrieving the contents of any individual factor F_x by traversing the LZ trie upwards from $L[x]$, just as done for decompression. This observation can make our output useful as a practical compressed data structure for substring extraction as well.

5.1.2 Space Improvement. The obvious disadvantage of our simple factorization algorithm fix is that it uses more than $O(z \lg \sigma)$ bits of space when $z = o(n/\log_\sigma n)$, which is when it is most interesting to compress T! A simple workaround is to start with $m = \sqrt{2n}$ based on the lower bound on the number of LZ78 factors (cf. Lemma 2.1). If, during the factorization, this limit is exceeded, then we double the value of m and repeat the whole process. Since we may rerun the process $O(\lg z)$ times, the total expected time is $O(n \lg z) = O(n \lg n)$ (recall that $\lg z = \Theta(\lg n)$ according to Lemma 2.1).[22] In exchange, the main memory space is now always $O(z \lg \sigma)$ bits. Further, the

[21] This stack size is a rough upper bound; a generally stricter bound is $\sqrt{2n} \lg \sigma$ bits [9, Lemma 1].

[22] In fact, the time is linear in most texts, because we process roughly a doubling amount of text in each run. In some texts, however, this is not the case; consider n equal characters followed by other n random characters.

extra space added to the compressed file due to the Bonsai trie is just $O(z \lg \sigma) + \rho_m$ with $m \leq 2z$. We call this variant brute, inspired by the fact that it uses a brute force strategy to find z.

THEOREM 5.2. *We can compute the LZ78 factorization in $O(n \lg z)$ expected time within $O(z \lg \sigma)$ bits of main memory working space, streaming the Bonsai coding to a rewritable external memory.*

PROOF. Using elias as our Bonsai table representation, we have $t_{\text{Bonsai}} = O(1)$ expected time and $\rho_m = O(m)$. Therefore, $O(n \lg \sigma / \log_\sigma n) + \rho_{2z} = O(z \lg \sigma)$ bits. □

Apart from the increased time, a flaw of this algorithm is that it may read T several times from disk, and thus it is not a streaming algorithm. In the next sections, we explore two faster solutions that in addition scan T only once.

5.2 A Growing Bonsai Table

We can obtain $O(z \lg \sigma)$ bits of working space for any input text by letting the Bonsai table rehash when reaching the maximum supported number of entries. We start with a Bonsai table that can store $m \leftarrow \sqrt{2n}$ elements without rehashing, since $\sqrt{2n}$ is a lower bound on z (cf. Lemma 2.1). Whenever we want to insert a new element into H storing already m elements, we allocate a new Bonsai table H' capable of storing $2m$ elements and populate it with the trie nodes stored in H. By doing so, we read T only once, but we now need to read and rewrite L on each rehash.

The main challenge is how to relocate each LZ trie node u from H to H', since its identifier ι_u in H is mentioned not only in L but also in the identifiers of its children $H.\text{lookup}(\iota_u, c)$ for $c \in \Sigma$. However, as explained, to map u from ι_u to its new identifier ι'_u in H', we need to know the new identifier ι'_w of its parent w. To resolve this dependency problem, we map the LZ trie nodes top-down. However, a simple DFS traversal on the LZ trie is slow, because enumerating the children of a node w costs $O(\sigma)$ expected time: We can only try $H.\text{lookup}(\iota_w, c)$ for all possible characters $c \in \Sigma$.

Instead, we relocate the nodes as follows: We scan L sequentially (on disk), starting with $L[1]$. Given we access $L[x]$, we traverse the LZ trie stored in H upwards to the root, starting at the node v with identifier $L[x]$ (i.e., v corresponds to the xth factor F_x). During this traversal, we put the edge labels $c_1, \ldots, c_{|F_x|}$ (with $F_x = c_1 \ldots c_{|F_x|}$) of the traversed path on a stack. Since all the nodes visited during this traversal, starting from the parent of v, must already exist in H', we can use the computed stack of characters to traverse downwards in H'. In detail, we traverse the trie of H' downwards from the root with $\iota'_0 := H'.\text{root}()$ and $\iota'_k := H'.\text{lookup}(\iota'_{k-1}, c_k)$ for $k \in [1..|F_x|)$. Then, we insert v into H' with $\iota'_{|F_x|} := H'.\text{insert}(\iota'_{|F_x|-1}, c_{|F_x|})$. Finally, we rewrite $L[x] \leftarrow \iota'_{|F_x|}$, because we use $\iota'_{|F_x|}$ as the identifier of v from now on. In total, our retraversal costs $O(|F_x| t_{\text{Bonsai}})$ time. Given that H stores y factors, the total time for relocating all nodes is $O(|F_1 \ldots F_y| t_{\text{Bonsai}}) = O(n t_{\text{Bonsai}})$. Since we perform $O(\lg z)$ passes, the total time reaches $O(t_{\text{Bonsai}} n \lg z) = O(t_{\text{Bonsai}} n \lg n)$.

With a technique similar to the jump pointers of Section 4.3.1, we can reduce the time to $O(z t_{\text{Bonsai}} \log_\sigma n) = O(n t_{\text{Bonsai}})$ by storing, while relocating nodes from H to H', $O(z/\log_\sigma n)$ sampled nodes of H in a (classic) hash table W, which maps the identifiers of H to the identifiers of H'. The table W uses $O(z \lg \sigma)$ bits, which is within our budget. During the relocation of the nodes, we fill it with every LZ trie node whose depth is a multiple of $\log_\sigma n$ and whose height is at least $\log_\sigma n$. By doing so, we ensure that $O(z/\log_\sigma n)$ nodes are sampled and that we traverse less than $2 \log_\sigma n$ nodes in H from any identifier $L[x]$ before reaching a sampled node, from which we can descend in H' and update $L[x]$ in time $O(t_{\text{Bonsai}} \log_\sigma n)$. Thus, we do the complete relocation in $O(t_{\text{Bonsai}} |L| \log_\sigma n)$ time. Since the size of L doubles each time, we increase the table, the total work amounts to $O(z \log_\sigma n)$ Bonsai operations.

Once we have built H', we continue with H' and discard H. The peak space usage is when we have both H and H' in memory, taking $(3/\alpha) m \lg \sigma + O(m) = O(z \lg \sigma)$ bits with $z \in (m..2m]$ and

an open-addressing Bonsai table with the maximum load factor $\alpha = m/|H|$. We can always keep the entries of L within $\lceil \lg \sigma \rceil + O(1)$ bits, slightly expanding them when we retraverse L to rewrite the new positions in H'. At the end, L stores identifiers of the keys stored in a Bonsai table that can store up to $2z$ entries, thus using $z \lg \lceil \rho_{2z} \rceil + O(z)$ bits, which is $O(z)$ bits for elias.

THEOREM 5.3. *We can compute the LZ78 factorization in $O(n)$ expected time within $O(z \lg \sigma)$ bits of main memory working space and $O(z)$ bits of external memory, streaming the Bonsai encoding to disk at the end. We need $z \lg^2 z$ bits of I/O, or $O((z \lg^2 z)/b + \lg z)$ I/O transfers in the external memory model with a block size of b bits.*

5.3 Multiple Bonsai Tables

Similar to Section 4.3.3, a way to avoid rebuilding the Bonsai table is to maintain multiple Bonsai tables H_h for $h \geq 0$, starting with $h = 0$ and a single Bonsai table H_0. When H_h becomes full (i.e., when its load factor reaches its maximum limit), we allocate a new table H_{h+1}, with $|H_{h+1}| = 2|H_h|$, where all the subsequent insertions take place from then on.

To properly address the nodes, we need to build a global identifier matching the entry of any Bonsai table. Our idea is to regard the tables as their concatenation, that is, $G := H_0 H_1 H_2 \ldots$. In this perspective, we compose a global identifier $G.\text{lookup}(x) := |H_0 H_1 \ldots H_{h-1}| + H_h.\text{lookup}(x)$ of a key x stored in table H_h by adding $|H_0 H_1 \ldots H_{h-1}|$ to the (local) identifier of x with respect to H_h.

Suppose we are at a node u with the global identifier ι_u in a table H_g and append a leaf v to u with edge label c by inserting v into the current hash table H_h with $H_h.\text{insert}(\iota_u, c)$. By doing so, we leave no indication in H_g of the existence of v. A consequence is that, if we want to descend from u by the character c, then we must probe the tables $H_g, H_{g+1}, \ldots, H_h$ to see if a child with edge label c was inserted in one of these later tables. Therefore, the cost of traversing towards a child worsens to $O(t_{\text{Bonsai}} \lg z)$ time, because we can build at most $\lg z$ tables during the factorization. However, since the children are always inserted later than their parents, the current table index does not decrease as we descend from the root towards the node where we will insert the new leaf, and thus, we do these $O(\lg z)$ probes once per inserted leaf, for a total time of $O(t_{\text{Bonsai}} z \lg z) = O(t_{\text{Bonsai}} n \lg \sigma)$. The Bonsai coding now stores $G = H_1 \ldots H_h$ and L.

This technique has the advantage that it treats T and L in streaming mode. The entries written in L are final (note that their bit widths grow each time we start using a new table). These can be compacted as outlined in Section 5.1 if we are willing to perform a second pass on L.

THEOREM 5.4. *We can compute the LZ78 factorization in $O(n \lg \sigma)$ expected time within $O(z \lg \sigma)$ bits of main memory working space, streaming the Bonsai coding of the factorization to disk.*

For decompression, we load all Bonsai tables $H_1, \ldots, H_h$ back into memory and treat them as one global table G. Remembering that L stores the list of global identifiers of the LZ trie nodes, we can compute the local identifier ι_v and the corresponding Bonsai table index $g \in [1..h]$ from the global identifier $L[x]$ such that $H_g.\text{key}(\iota_v) = (\iota_u, c)$, where ι_u is the *global* identifier of the parent of the node v having the global identifier $L[x]$. Since $|H_g| = 2|H_{g-1}|$ for every $g \geq 1$, finding the table H_g from the global identifier $L[x]$ is a matter of dividing $L[x]$ by $|H_0|$ and then taking the logarithm to base 2, similarly to Section 4.3.2.

Implementation Details. Each table H_g has its own identifier domain $[1..\rho_{|H_g|}]$, hash function with prime number p_g, and so on. The prime p_g of its hash function must be larger than $(\rho_{|H_0|} + \rho_{|H_1|} + \ldots + \rho_{|H_g|}) \cdot \sigma$, so any element (ι, c) can be stored in H, where ι is a global identifier of $H_0 \ldots H_g$. For the Bonsai tables using the displacement array such as layered and elias, we have $p_g/\rho_{|H_g|} = \Theta(p_g/|H_g|) \leq 2\sigma + O(1)$.

Table 8. Alias Names of the Algorithms Introduced in Section 5 Computing the Bonsai Scheme

Name	Location	Description
fix	Section 5.1 and Theorem 5.1	fixed Bonsai table of maximum size, no rebuilding,
fix_+	Section 5.4	fix knowing z in advance,
brute	Section 5.1 and Theorem 5.2	try-and-error on different fixed table sizes,
grow	Section 5.2	growing Bonsai table without node sampling,
grow_s	Section 5.2 and Theorem 5.3	growing Bonsai table with hash table W storing the node sampling,
multi	Section 5.3 and Theorem 5.4	multiple Bonsai tables.

5.4 Recompression

After having computed the LZ78 factorization, we know the number of factors z, and therefore can represent the LZ trie with a Bonsai table that can just store z entries. Storing this dense Bonsai table makes the bit vector B marking the free cells useless and can improve the compression ratio. We can directly convert a Bonsai coding with the algorithm of Section 5.2 to this format. To see how well this recompression may work, we add the variant fix_+ of fix, which allocates a Bonsai table that can store exactly z elements (the number of cells is then z/α, α being the maximum load factor). fix_+ requires the number of factors z of the input as an additional parameter.

5.5 Implementation

For the following experiments, we use layered_2 and layered_3 of Section 3.2.3 as the Bonsai table, indicating the use of the latter with a subscript "3" in the alias names listed in Table 8. We use bucket (Section 3.4) as the compact hash table in the second layer of layered_3. The last layer in both variants always uses the C++ STL class `std::map`, which is a balanced binary search tree implementation. The coding consists of the bit vector B described in the paragraph before Theorem 5.1, and the non-empty cells of the Bonsai table. Our compressors are publicly available at `https://github.com/koeppl/Low-LZ78`.

We evaluated the experiments with the maximum load factors $1/\alpha = 1.05, 1.10, 1.20, 1.40, 1.60$ for layered, where the displacement values are stored in two or three layers with the first layer being an array storing integers with a bit width of $b_0 = \lceil \lg 1/(1/\alpha - 1) \rceil$ for layered_2 and $b_0 = 3, b_1 = 7$ for the first and second layer, respectively, for layered_3. The idea behind b_0 of layered_2 is that large displacement values become more likely with larger load factors.

We use $\Theta(\lg \sigma)$ as our sampling rate in grow_s; more precisely, $\lceil \lg (p + |H|)/|H| \rceil = \Theta(\lg \sigma)$ for a prime number p with $|\mathcal{K}| = |H|\sigma < p \leq 2|H|\sigma$, where $\mathcal{K} = [1..|H|] \times \Sigma$ is the universe of keys that can be stored in a Bonsai table H.

5.6 Experimental Results

We experimentally evaluate our introduced algorithms computing the Bonsai coding and compare them with some space-efficient solutions empirically observed in Section 4 for computing the classic LZ78 code, namely, grp_β, defined in Section 4.3.4, and binary and ternary, defined in Section 4.1. We name the methods of this section as described in Figure 8. Like in Section 4.4, we measure the time and memory needed for the factorization. We also measure these features for the decompression and study the overhead of the different Bonsai codings compared with the classic LZ78 format of Section 2.1.3. We do not show data points for fix and fix_3 applied on FIBONACCI, because in this instance the gap between the number of LZ78 factors and its upper bound is too large, causing us to allocate much more memory than actually needed.

Figure 15 shows the maximum RAM used by each structure during compression and the resulting compression time. Here, it can be seen that multi and multi_3 largely outperform grow and grow_3, respectively, in both space and time, using 1.0–2.2 bits and 0.2–0.3 μsec per symbol with

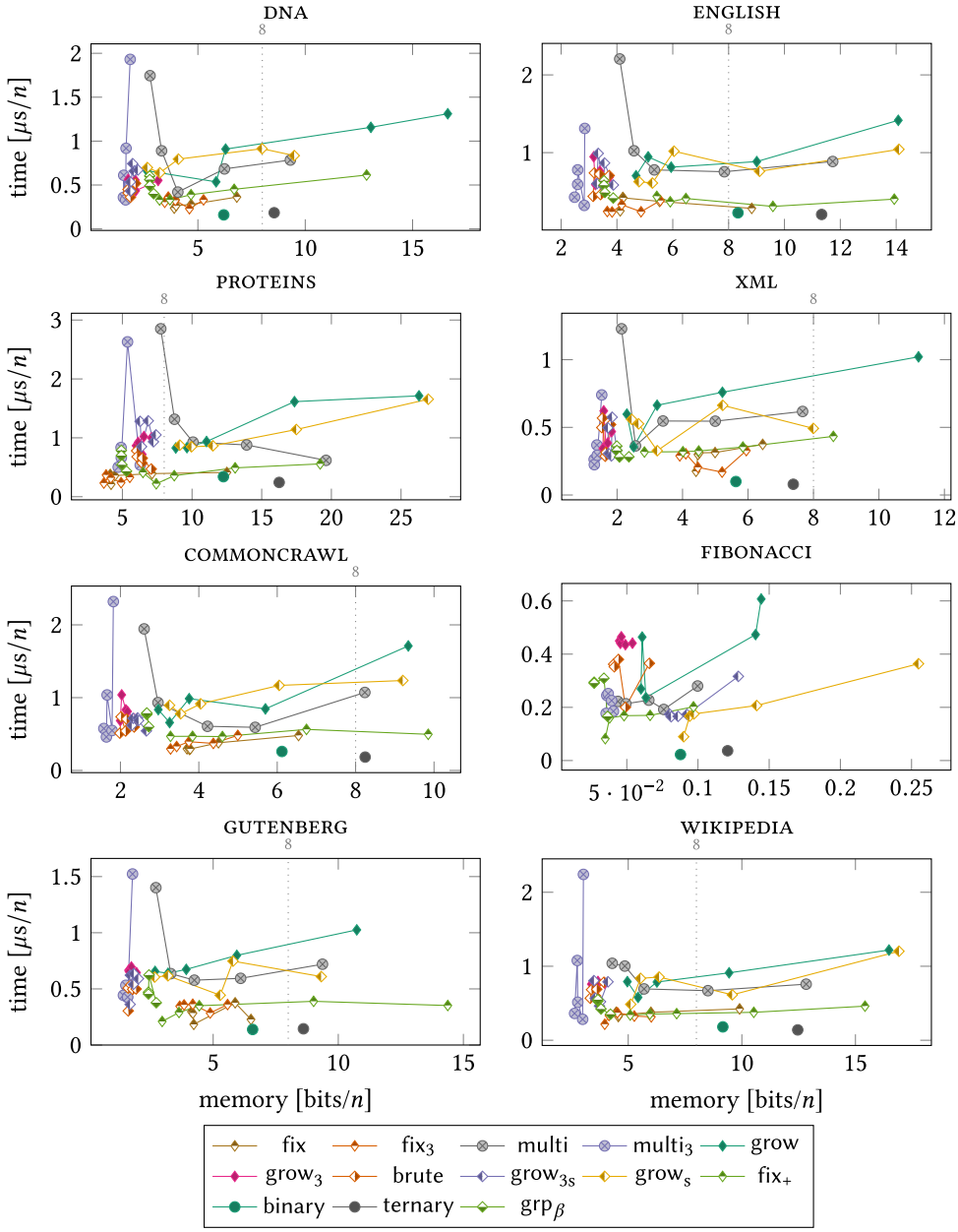


Fig. 15. Maximum RAM and time used during the LZ78 factorizations with the Bonsai coding variants of Section 5 and the space-efficient variants of Section 4.

$1/\alpha = 1.40$ (omitting FIBONACCI). For the same space, the overhead of using multiple tables is lower than that of rebuilding the table, which implies rereading the L array from disk. In general, the time of multi and in particular multi$_3$ is very sensitive to high load factors, without significantly improving the space. Especially multi$_3$ is one of the most memory-efficient variants. It gets close to and even beats fix on most instances (with the exception of PROTEINS, where the final-size guess of fix is nearly optimal). The maximum space usage of grow occurs when it has to expand the table,

at which moment it has the old and new tables in RAM. This requires more space than multi even when the multi tables are emptier on average. Compared with the tries for the classic LZ coding, we see that binary and ternary are always the fastest, but also among the least memory-efficient approaches. The variant grp_β of cht is close to brute and fix, but it is almost always outperformed by an instance of multi_3.

Figure 16 shows the RAM used by each structure during decompression. This time grow always obtains better space than multi, though it still requires more time. grow uses 0.9–1.8 bits and 0.1–0.2 μsec per symbol, even outperforming fix, which uses much more space (except on PROTEINS). grow does not need to make the hash tables grow at decompression, thus it is much faster than at compression and uses less space than multi, which has emptier tables. multi, in turn, is faster than at compression because it traverses the paths upwards, although it still uses multiple tables.

Our classic LZ78 decoder moves the arrays S and R into RAM, storing them in arrays with fixed bit widths (i.e., 8 bits per entry in S and 32 bits per entry for R). We observe that decoding the classic LZ78 coding is faster than any of the Bonsai code decompressors. Its memory footprint, however, is much larger than the codes using the lightweight layered_3 Bonsai tables, though it is competitive with the heavyweight layered_2 Bonsai tables.

Our Bonsai encoders do not store the same information of a classical LZ78 compressor. We expect their compression ratio to be suboptimal, because they store hash tables with empty cells (or mark those cells in a bitmap B), and they must store the displacements of the hash tables. An alternative approach to store the Bonsai coding is to use the Elias-γ encoding for the displacement values (like the hash table elias does). In Figure 17, we evaluate the use of Elias-γ encoding [23] for the first layer of displacements stored by layered. We can observe that the maximum load factor and the compression ratio are anti-correlated in most of the instances, that is, the higher the load factor, the smaller the compressed size. For high load factors, however, the first displacement layer in layered_2 uses more bits per entry according to our definition of b_0. Since the first layer is usually mostly empty, the Elias-γ encoding produces a smaller output than dumping this array in its plain form. This is, however, not true for smaller load factors, where the Elias-γ encoding can be even more expensive (see the top tables in Figure 17 for examples). Since we gain with the Elias-γ when encoding the best compression ratios, we stick to it in the following evaluations.

Figure 18 gives the best compression ratios of our algorithms along with the LZ78 coding (see Table 3). We observe that the coding using layered_3 can be stored more compactly than the coding using layered_2 except for grow, where we observe the contrary on some instances. In general, the best compression ratios are obtained with variants of grow and brute, which (excluding FIBONACCI) pose an overhead of 24%–37% over the plain LZ78 encoding. The fully streaming algorithm, multi, poses an overhead of 31%–41%. These ratios are obtained when using, roughly, the minimum memory and maximum time for the factorization. Figure 19 relates the ratios with time needed for computing the factorization. It can be seen that the given overheads are obtained with a compression speed of 0.3–0.5 μsec per symbol for grow and brute, and 0.2–0.5 for multi.

6 CONCLUSIONS

We have presented the first practical evaluation of LZ78 and LZW algorithms, mostly focusing in low-memory footprints so large files can be processed. We introduced new compression algorithms based on compact hashing, which can efficiently compute the LZ78 and LZW factorizations in space considerably less than the input size. For example, our most memory-efficient approaches, grp_0 and grp_{10}, typically use 25%–60% of the space required by the input text and compress the text at a speed of about 1–2.5 MB/sec. If speed is of concern, then we can perform the factorization 1.5–5 times faster than a standard approach like judy, by using rolling or ternary, reaching a speed of 5–10 MB/sec.

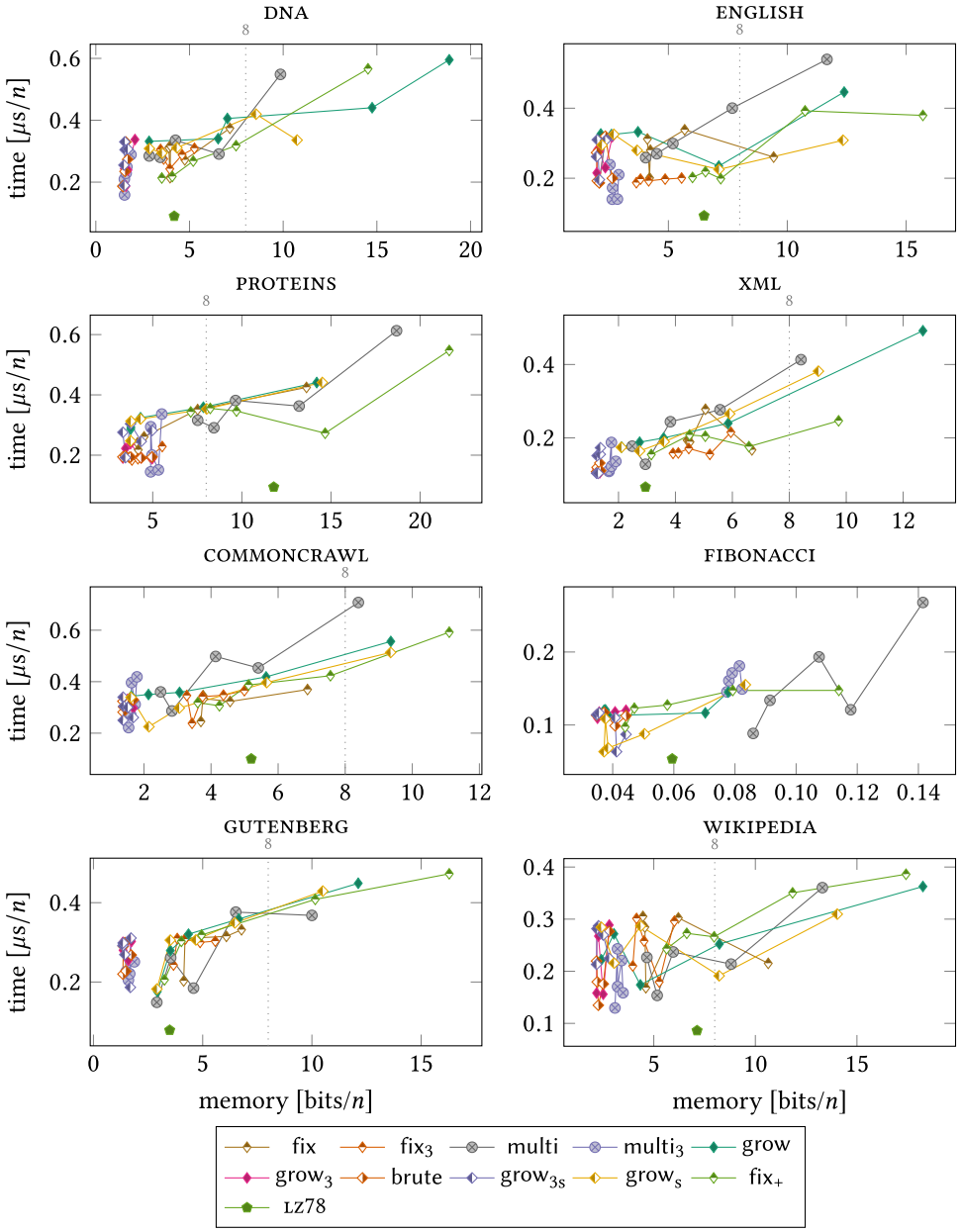


Fig. 16. Maximum RAM and time used during decoding of a file stored with one of the Bonsai coding variants of Section 5. This is compared with the decompression of the same file compressed with LZ78 using the classic coding described in Section 2.1.3.

We then pushed even more on the space usage, developing a family of LZ78 compression algorithms that, under some simplifying assumptions, use $O(z \lg \sigma)$ bits of main memory in expectation (where z is the number of LZ78 factors and σ is the alphabet size), that is, less memory space than the $O(z \lg n)$ bits needed to store the compressed file (where n is the text length). These algorithms

ENGLISH, grow			PROTEINS, fix_3			FIBONACCI, fix		
	Ratio			Ratio			Ratio	
$\frac{1}{\alpha}$	plain	γ	$\frac{1}{\alpha}$	plain	γ	$\frac{1}{\alpha}$	plain	γ
1.05	53.31	47.89	1.05	73.58	67.80	1.05	20.87	7.22
1.10	53.60	49.49	1.10	74.12	67.57	1.10	18.27	7.54
1.20	55.91	52.90	1.20	75.33	68.01	1.20	16.00	8.20
1.40	65.13	63.71	1.40	77.84	69.11	1.40	14.05	9.50
1.60	79.73	89.68	1.60	80.45	81.63	1.60	10.80	10.80

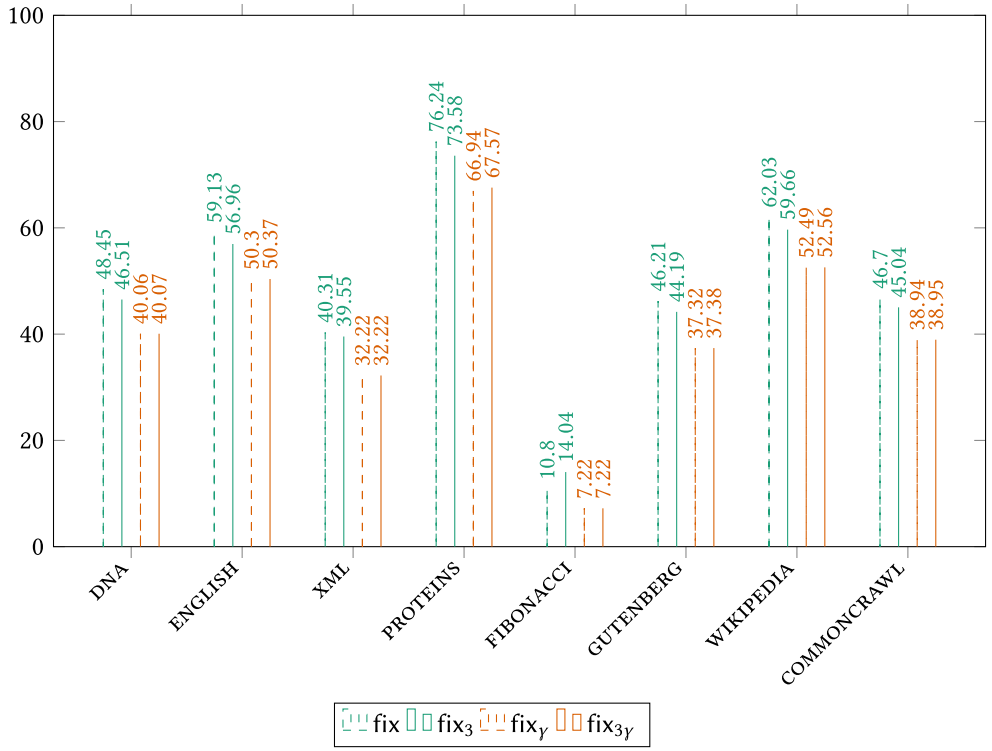


Fig. 17. Top: Compression ratios of the plain Bonsai coding and of the Bonsai coding with the stored displacement values of the first layer in layered$_2$ or layered$_3$ encoded with Elias-γ (labeled with 'γ'). Bottom: Compression ratios of fix (respectively, fix_3) with and without Elias-γ encoding. We add γ in the subscript if we apply Elias-γ encoding. For each dataset, we selected the best compression ratios while varying the maximum load factor of the Bonsai table.

run in $O(n \lg \sigma)$ expected time for compression and in $O(n)$ expected time for decompression. Most of them read the text and write the output in streaming mode, producing the output in a special format.

None of the previous algorithms achieves such a low memory footprint. One of our most memory-efficient variants, multi_3, uses about half the space of the most memory-efficient implementation computing the classic LZ78 factorization of most typical texts, at 2–5 MB/sec. Their compressed format can be decompressed using typically 30%–60% of the memory requirement of the classic LZ78 decoder, while taking about 50%–100% additional time, that is, 5–10 MB/sec. A disadvantage is that, although they compute the correct LZ78 factorization and trie, their special

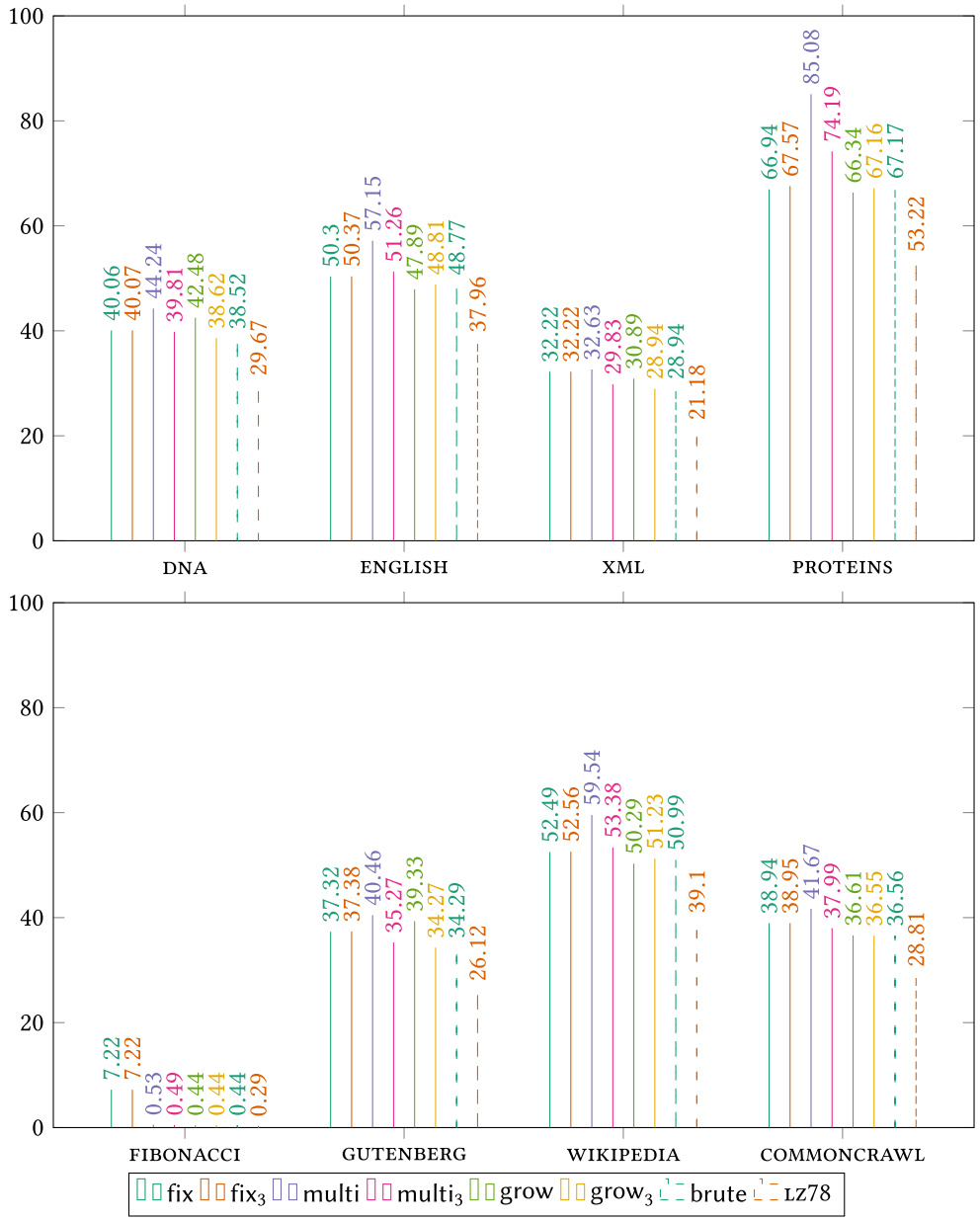


Fig. 18. Compression ratios of our algorithms of Section 5 computing the Bonsai coding. We omit the "s" variants of grow and grow_3, since they only make a difference in the compression speed, not in the final output. We compare these ratios with those of the classic coding of LZ78 described in Section 2.1.3. For each dataset, we selected the best compression ratios while varying the maximum load factor of the Bonsai table.

encoding is 25%–40% larger than the classic LZ78 encoding, and thus they are weaker as plain compressors.

This encoding overhead makes these latter compressors more interesting for other purposes, such as building compressed text representations for substring extraction [67] or for the

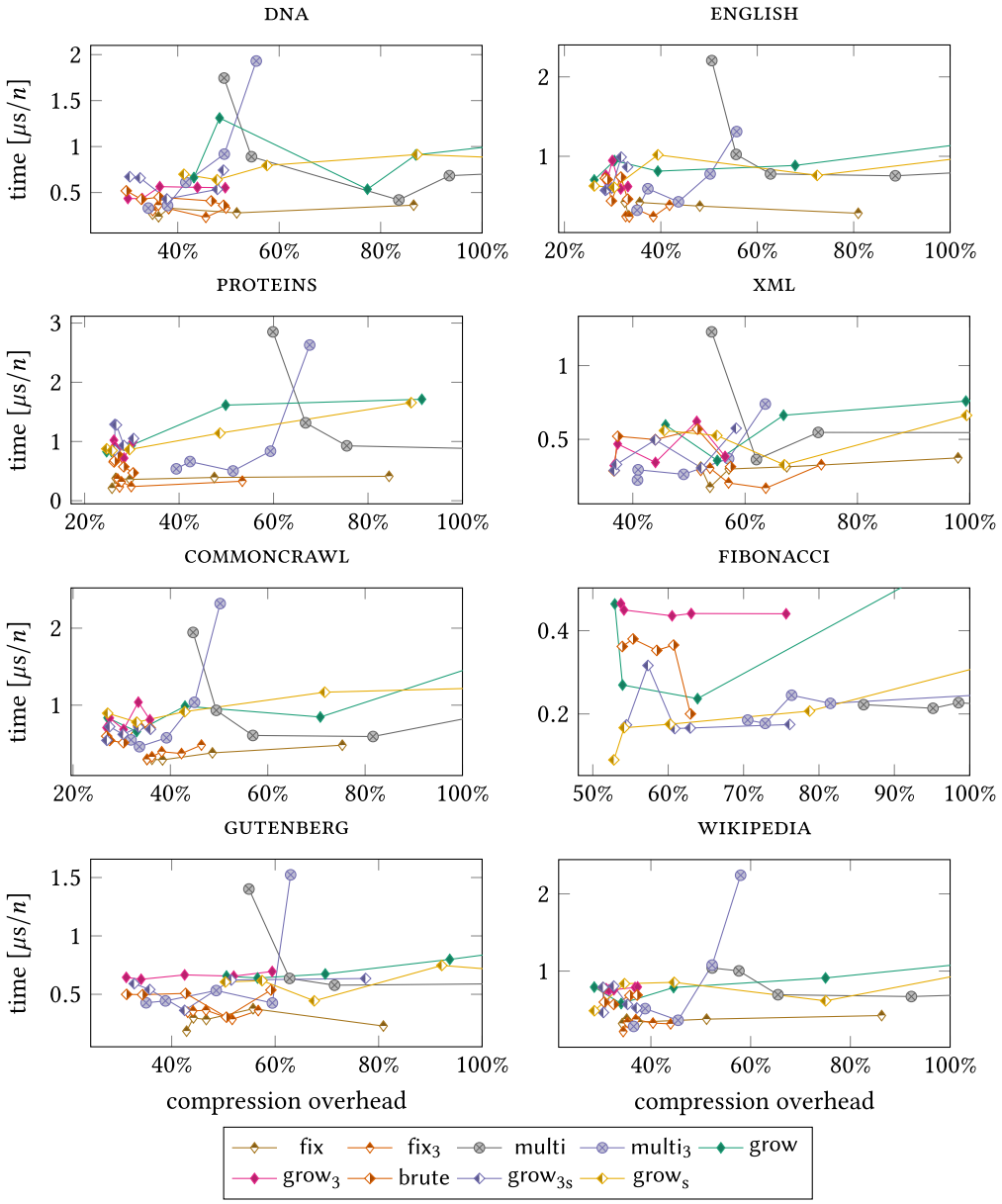


Fig. 19. Compression time versus the compression overhead, i.e., the ratio of the final compressed size over the classical LZ78 output size (cf. Table 3) minus 1. We clipped the plots at 100% compression overhead, where the output size is twice the classical LZ78 output size.

compressed-space construction of LZ78-based text indexes [3]. In general, our insert-only compact tries can be useful for many other applications unrelated to Lempel-Ziv compression.

As a final note for practical applications, we point out that writing a dedicated memory allocator is crucial to actually achieving the space bounds observed by our experiments. This is because we monitored the sizes requested by calls to `malloc` or `new`, neglecting the costs for actually

maintaining these requested parts of memory (i.e., both the memory allocator and our implementations maintain the lengths of the allocated arrays). In the usual setting, `malloc` stores additional information about the size of the requested memory, which is again rounded up to be a multiple of the machine word size. Having a lot of tiny memory fragments requested by a standard `malloc` call considerably wastes much more memory than when using a memory manager tailored for the compact hash tables studied here.

ACKNOWLEDGMENTS

Part of this collaboration started during the Dagstuhl Seminar 16431, "Computation over Compressed Structured Data."

We thank Juha Kärkkäinen for sharing some thoughts about adding jump pointers to the LZ trie.

REFERENCES

[1] Ole Amble and Donald E. Knuth. 1974. Ordered hash tables. *Comput. J.* 17, 2 (1974), 135–142.
[2] Amihood Amir, Gary Benson, and Martin Farach. 1996. Let sleeping files lie: Pattern matching in Z-compressed files. *J. Comput. Syst. Sci.* 52, 2 (1996), 299–307.
[3] Diego Arroyuelo and Gonzalo Navarro. 2011. Space-efficient construction of Lempel-Ziv compressed text indexes. *Inf. Comput.* 209, 7 (2011), 1070–1102.
[4] Diego Arroyuelo, Gonzalo Navarro, and Kunihiko Sadakane. 2012. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica* 62, 1–2 (2012), 54–101.
[5] Diego Arroyuelo, Pooya Davoodi, and Srinivasa Rao Satti. 2016. Succinct dynamic cardinal trees. *Algorithmica* 74, 2 (2016), 742–777.
[6] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Rajeev Raman. 2017. LZ78 compression in low main memory space. In *Proceedings of the 24th SPIRE (LNCS)*, Vol. 10508. 38–50.
[7] Julian Arz and Johannes Fischer. 2018. Lempel-Ziv-78 compressed string dictionaries. *Algorithmica* 80, 7 (2018), 2012–2047.
[8] Nikolas Askitis. 2009. Fast and compact hash tables for integer keys. In *Proceedings of the 30th ACSC (CRPIT)*, Vol. 91. 101–110.
[9] Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. 2012. Efficient LZ78 factorization of grammar compressed text. In *Proceedings of the 19th SPIRE (LNCS)*, Vol. 7608. 86–98.
[10] Hideo Bannai, Travis Gagie, and Tomohiro I. 2020. Refining the r-index. *Theor. Comput. Sci.* 812 (2020), 96–108.
[11] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. 2010. Dynamic Z-fast tries. In *Proceedings of the 17th SPIRE (LNCS)*, Vol. 6393. 159–172.
[12] David Benoit, Erik D. Demaine, J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. 2005. Representing trees of higher degree. *Algorithmica* 43, 4 (2005), 275–292.
[13] Jon Louis Bentley and Robert Sedgewick. 1997. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th SODA*. 360–369.
[14] John R. Black, Charles U. Martel, and Hongbin Qi. 1998. Graph and hashing algorithms for modern architectures: Design and performance. In *Proceedings of the 2nd WAE*. 37–48.
[15] Michael Burrows and David J. Wheeler. 1994. *A Block Sorting Lossless Data Compression Algorithm*. Technical Report 124. Digital Equipment Corporation, Palo Alto, CA.
[16] Pedro Celis, Per-Åke Larson, and J. Ian Munro. 1985. Robin Hood hashing. In *Proceedings of the 26th FOCS*. 281–288.
[17] Pafnuty Lvovich Chebyshev. 1852. Mémoire sur les nombres premiers. *J. Mathém. Pures Appliq.* 1 (1852), 366–390.
[18] Kai-Min Chung, Michael Mitzenmacher, and Salil P. Vadhan. 2013. Why simple hash functions work: Exploiting the entropy in a data stream. *Theor. Comput.* 9 (2013), 897–945.
[19] David R. Clark. 1996. *Compact Pat Trees*. Ph.D. Dissertation. University of Waterloo, Canada.
[20] John G. Cleary. 1984. Compact hash tables using bidirectional linear probing. *IEEE Trans. Comput.* 33, 9 (1984), 828–834.
[21] John J. Darragh, John G. Cleary, and Ian H. Witten. 1993. Bonsai: A compact representation of trees. *Softw. Prac. Exper.* 23, 3 (1993), 277–291.
[22] Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane. 2017. Compression with the tudocomp Framework. In *Proceedings of the 16th SEA (LIPIcs)*, Vol. 75. 13:1–13:22. arXiv:1702.07577.
[23] Peter Elias. 1974. Efficient storage and retrieval by content and address of static files. *J. ACM* 21, 2 (1974), 246–260.
[24] Jerome A. Feldman and James R. Low. 1973. Comment on Brent's scatter storage algorithm. *Commun. ACM* 16, 11 (1973), 703.

[25] Hector Ferrada and Gonzalo Navarro. 2013. A Lempel-Ziv compressed structure for document listing. In *Proceedings of the 20th SPIRE (LNCS)*, Vol. 8214. 116–128.
[26] Héctor Ferrada and Gonzalo Navarro. 2014. Efficient compressed indexing for approximate top-k string retrieval. In *Proceedings of the 21st SPIRE (LNCS)*, Vol. 8799. 18–30.
[27] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.
[28] Johannes Fischer and Pawel Gawrychowski. 2015. Alphabet-dependent string searching with wexponential search trees. In *Proceedings of the 26th CPM (LNCS)*, Vol. 9133. 160–171.
[29] Johannes Fischer and Dominik Köppl. 2017. Practical evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch tries. In *Proceedings of the 24th SPIRE (LNCS)*, Vol. 10508. 191–207. arXiv:1706.03035.
[30] Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. 2015. Approximating LZ77 via small-space multiple-pattern matching. In *Proceedings of the 23rd ESA (LNCS)*, Vol. 9294. 533–544.
[31] Johannes Fischer, Tomohiro I., Dominik Köppl, and Kunihiko Sadakane. 2018. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica* 80, 7 (2018), 2048–2081.
[32] Leszek Gasieniec and Wojciech Rytter. 1999. Almost optimal fully LZW-compressed pattern matching. In *Proceedings of the 9th DCC*. 316–325.
[33] Leszek Gasieniec, Marek Karpinski, Wojciech Plandowski, and Wojciech Rytter. 1996. Efficient algorithms for Lempel-Ziv encoding (extended abstract). In *Proceedings of the 5th SWAT*. 392–403.
[34] Gaston H. Gonnet and Ricardo A. Baeza-Yates. 1990. An analysis of the Karp-Rabin string matching algorithm. *Inform. Process. Lett.* 34, 5 (1990), 271–274.
[35] Ronald L. Graham, Donald Ervin Knuth, and Oren Patashnik. 1994. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley.
[36] Godfrey Harold Hardy and Edward Maitland Wright. 2008. *An Introduction to the Theory of Numbers* (6th ed.). Oxford University Press.
[37] Gregory L. Heileman and Wenbin Luo. 2005. How caching affects hashing. In *Proceedings of the 7th ALENEX*. 141–154.
[38] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. 2008. Hopscotch hashing. In *Proceedings of the 22nd DISC (LNCS)*, Vol. 5218. 350–364.
[39] Guy Jacobson. 1989. Space-efficient static trees and graphs. In *Proceedings of the 30th FOCS*. 549–554.
[40] Jesper Jansson, Kunihiko Sadakane, and Wing-Kin Sung. 2015. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica* 71, 4 (2015), 969–988.
[41] Shunsuke Kanda, Dominik Köppl, Yasuo Tabei, Kazuhiro Morita, and Masao Fuketa. 2020. Dynamic path-decomposed tries. *ACM J. Exper. Algor.* 25, 1 (2020), 1.13:2–1.13:28.
[42] J. Kärkkäinen, G. Navarro, and E. Ukkonen. 2003. Approximate string matching on Ziv-Lempel compressed text. *J. Disc. Algor.* 1, 3/4 (2003), 313–338.
[43] Richard M. Karp and Michael O. Rabin. 1987. Efficient randomized pattern-matching algorithms. *IBM J. Res. Devel.* 31, 2 (1987), 249–260.
[44] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, Masamichi Miyazaki, and Setsuo Arikawa. 1998. Multiple pattern matching in LZW compressed text. In *Proceedings of the 8th DCC*. 103–112.
[45] Takuya Kida, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. 1999. Shift-and approach to pattern matching in LZW compressed text. In *Proceedings of the 10th CPM*. 1–13.
[46] Donald E. Knuth. 1981. *Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison Wesley.
[47] Donald E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley.
[48] D. Köppl and K. Sadakane. 2016. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proceedings of the 26th DCC*. 3–12.
[49] Dominik Köppl, Simon J. Puglisi, and Rajeev Raman. 2020. Fast and simple compact hashing via bucketing. In *Proceedings of the 19th SEA (LIPIcs)*.
[50] Daniel Lemire and Owen Kaser. 2010. Recursive n-gram hashing is pairwise independent, at best. *Comput. Speech Lang.* 24, 4 (2010), 698–710.
[51] Abraham Lempel and Jacob Ziv. 1976. On the complexity of finite sequences. *IEEE Trans. Inf. Theor.* 22, 1 (1976), 75–81.
[52] Hua Luan, Xiaoyong Du, Shan Wang, Yongzhi Ni, and Qiming Chen. 2007. J^+-Tree: A new index structure in main memory. In *Proceedings of the 12th DASFAA (LNCS)*, Vol. 4443. 386–397.
[53] Tobias Maier and Peter Sanders. 2017. Dynamic space efficient hashing. In *Proceedings of the 25th ESA (LIPIcs)*, Vol. 87. 58:1–58:14.
[54] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. 2019. Masking Dilithium—efficient implementation and side-channel evaluation. In *Proceedings of the 17th ACNS (LNCS)*, Vol. 11464. 344–362.
[55] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. 2017. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the 28th SODA*. 408–424.

[56] Yuto Nakashima, Tomohiro I., Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2015. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inform. Process. Lett.* 115, 9 (2015), 655–659.
[57] G. Navarro. 2003. Regular expression searching on compressed text. *J. Disc. Algor.* 1, 5/6 (2003), 423–443.
[58] Gonzalo Navarro and Kunihiko Sadakane. 2014. Fully functional static and dynamic succinct trees. *ACM Trans. Algor.* 10, 3 (2014), 16:1–16:39.
[59] Gonzalo Navarro and Jorma Tarhio. 2005. LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Softw. Prac. Exper.* 35, 12 (2005), 1107–1130.
[60] Takaaki Nishimoto and Yasuo Tabei. 2019. Conversion from RLBWT to LZ77. In *Proceedings of the CPM (LIPIcs)*, Vol. 128. 9:1–9:12.
[61] Takaaki Nishimoto, Tomohiro I., Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. 2020. Dynamic index and LZ factorization in compressed space. *Disc. Appl. Math.* 274 (2020), 116–129.
[62] Anna Pagh, Rasmus Pagh, and Milan Ruzic. 2011. Linear probing with 5-wise independence. *SIAM Rev.* 53, 3 (2011), 547–558.
[63] Mihai Patrascu and Mikkel Thorup. 2016. On the k-independence required by linear probing and minwise independence. *ACM Trans. Algor.* 12, 1 (2016), 8:1–8:27.
[64] Alberto Policriti and Nicola Prezza. 2016. Computing LZ77 in run-compressed space. In *Proceedings of the DCC*. 23–32.
[65] Andreas Poyias, Simon J. Puglisi, and Rajeev Raman. 2018. m-Bonsai: A practical compact dynamic trie. *Int. J. Found. Comput. Sci.* 29, 8 (2018), 1257–1278.
[66] Luís M. S. Russo and Arlindo L. Oliveira. 2008. A compressed self-index using a Ziv-Lempel dictionary. *Inf. Retr.* 11, 4 (2008), 359–388.
[67] Kunihiko Sadakane and Roberto Grossi. 2006. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th SODA*. 1230–1239.
[68] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Amortized efficiency of list update and paging rules. *Commun. ACM* 28, 2 (1985), 202–208.
[69] Takuya Takagi, Shunsuke Inenaga, Kunihiko Sadakane, and Hiroki Arimura. 2017. Packed compact tries: A fast and efficient data structure for online string processing. *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.* 100-A, 9 (2017), 1785–1793.
[70] Terry A. Welch. 1984. A technique for high-performance data compression. *IEEE Comput.* 17, 6 (1984), 8–19.
[71] Naoki Yoshinaga and Masaru Kitsuregawa. 2014. A self-adaptive classifier for efficient text-stream processing. In *Proceedings of the 25th COLING*. 1091–1102.
[72] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theor.* 23, 3 (1977), 337–343.
[73] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.* 24, 5 (1978), 530–536.

Received September 2020; revised March 2021; accepted August 2021