Practical Evaluation of Lempel-Ziv-78 and Lempel-Ziv-Welch Tries

Johannes Fischer   Dominik Köppl

Department of Computer Science, TU Dortmund, Germany

Abstract

We present the first thorough practical study of the Lempel-Ziv-78 and the Lempel-Ziv-Welch computation based on trie data structures. With a careful selection of trie representations we can beat well-tuned popular trie data structures like Judy, m-Bonsai or Cedar.

1   Introduction

The LZ78-compression scheme [34] is an old compression scheme that is still in use today, e.g., in the Unix `compress` utility, in the GIF-standard, in string dictionaries [2], or in text indexes [1]. Its biggest advantage over LZ77 [33] is that LZ78 allows for an easy construction *within compressed space* and in *near-linear time*, which is (to date) not possible for LZ77. Still, although LZ77 often achieves marginally better compression rates, the output of LZ78 is usually small enough to be used in practice, e.g. in the scenarios mentioned above [4, 1].

While the construction of LZ77 is well studied both in theory [4, 11, e.g.] and in practice [14, 13, e.g.], only recent interest in LZ78 can be observed: just in 2015 Nakashima et al. [26] gave the first (theoretical) linear time algorithm for LZ78. On the practical side, we are not aware of any systematic study.

We present the first thorough study of LZ78-construction algorithms. Although we do not present any new theoretical results, this paper shows that if one is careful with the choices of tries, hash functions, and the handling of dynamic arrays, one can beat well-tuned out-of-the-box trie data structures like Judy[1], m-Bonsai [27], or the Cedar-trie [32].

Related Work.   An LZ78 factorization of size z can be stored in two arrays with $z \lg \sigma$ and $z \lg z$ bits to represent the character and the referred index, respectively, of each factor. This space bound has not yet been achieved by any efficient trie data structure. Closest to this bound is the approach of Arroyuelo and Navarro [1, Lemma 8], taking $2z \lg z + z \lg \sigma + \mathcal{O}(z)$ bits and $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ time for the LZ78 factorization. Allowing $\mathcal{O}(z \lg z)$ bits, $\mathcal{O}\left(n + z \frac{\lg^2 \lg \sigma}{\lg \lg \lg \sigma}\right)$ time is possible [10]. Another option is the dynamic trie of Jansson et al. [12] using $\mathcal{O}(n(\lg \sigma + \lg \lg_\sigma n)/ \lg_\sigma n)$ bits of working space and $\mathcal{O}\left(n \lg^2 \lg n/ (\lg_\sigma n \lg \lg \lg n)\right)$ time. All these tries are favorable for small alphabet sizes (achieving linear or sub-linear time when $\lg \sigma = o(\lg n \lg \lg \lg n/\lg^2 \lg n)$). If the alphabet size σ becomes large, the upper bounds on the time get unattractive. Up to $\lg \sigma = o(\lg n)$, we can use a linear time solution taking $\mathcal{O}(n \lg \sigma)$ bits of space [17, 25]. Finally, for large σ, there is a linear time approach taking $(1 + \epsilon)n \lg n + \mathcal{O}(n)$ bits of space [11]. Further *practical* trie implementations are mentioned in Section 4.

2   Preliminaries

Let T be a text of length n over an alphabet $\Sigma = \{1, \ldots, \sigma\}$ with $|\Sigma| \le n^{\mathcal{O}(1)}$. 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 i-th suffix of T, and denote a substring $T[i] \cdots T[j]$ with $T[i..j]$. A **factorization** of T of size z

---

[1]http://judy.sourceforge.net

Figure 3: LZ78 trie and LZW trie. Given the text $T =$ `aaababaaaba`, LZ78 factorizes T into $\overset{1}{\text{a}}|\overset{2}{\text{aa}}|\overset{3}{\text{b}}|\overset{4}{\text{ab}}|\overset{5}{\text{aaa}}|\overset{6}{\text{ba}}$, where the vertical bars separate the factors. The LZ78 factorization is output as: `a|(1,a)|b|(1,b)|(2,a)|(3,a)`. This output is represented by the left trie (a). The LZW factorization of the same text is $\overset{1}{\text{a}}|\overset{2}{\text{aa}}|\overset{3}{\text{b}}|\overset{4}{\text{a}}|\overset{5}{\text{ba}}|\overset{6}{\text{aab}}|\overset{7}{\text{a}}$. We output it as `-1|1|-2|-1|3|2|-1`. This output induces the right trie (b).

(a) LZ78-Trie   (b) LZW-Trie

partitions T into z substrings (**factors**) $F_1 \cdots F_z = T$. In this article, we are interested in the LZ78 and LZW factorization. If we stipulate that F_0 and $F_{z+1}[1]$ are the empty string, we get:

A factorization $F_1 \cdots F_z = T$ is called the **LZ78 factorization** [34] of T iff $F_x = F_y c$ with $F_y = \mathrm{argmax}_{S \in \{F_{y'}:0 \leq y' < x\}} |S|$ and $c \in \Sigma$ for all $1 \leq x \leq z$; we say that y is the **referred index** of the factor F_x.

A factorization $F_1 \cdots F_z = T$ is called the **LZW factorization** [31] of T iff $F_x = F_y F_{y+1}[1]$ with $F_y = \mathrm{argmax}_{S \in \{F_{y'}:1 \leq y' < x\}} |S|$, or $F_x = c \in \Sigma$ if no such F_y exists, for all $1 \leq x < z$. If $F_x = F_y F_{y+1}[1]$ for a y with $1 \leq y < x$, we call y the **referred index** of the factor F_x. Otherwise, $F_x = c$ for a $c \in \Sigma$; we set its referred index to $-c < 0$.

The factors can be represented in a trie, the so-called **LZ trie**. Each factor F_x (except the last factor in LZW) is represented by a trie node v labeled with x ($1 \leq x \leq z$) such that the parent u of v is labeled with y if y is the referred index of F_x. The edge (u, v) is then labeled with the last character of the factor F_x (or the first character of F_{x+1} for LZW).

Output.   We transform the list of factors to a list of integer values as follows: We linearly process each factor F_x for $1 \leq x \leq z$. If F_x's referred index is not positive, F_x is equal to a character c that is output (we output $-c$ in case of LZW). A factor F_x with a referred index $y > 0$ is processed as follows:

LZ78: If $F_x = F_y c$ for a $c \in \sigma$, we output the tuple (y, c).

LZW: If $F_x = F_y F_{y+1}[1]$ (or $F_x = F_y$ for $x = z$), we output $y + \sigma$.

Algorithm.   The folklore algorithm computing LZ78 and LZW uses a dynamic LZ trie that grows linearly in the number of processed factors. The dynamic LZ trie supports the creation of a node, the navigation from a node to one of its children, and the access to the labels.

Given that z is the number of LZ78 or LZW factors, the algorithm performs z searches of a prefix of a given suffix of the text. It inserts z times a new leaf in the LZ trie. It takes n times an edge from a node to one of its children.

3   LZ-Trie representations

In this section, we show five representations, each providing different trade-offs for computation speed and memory consumption. All representations have in common that they work with dynamic arrays.

Resize Hints.   The usual strategy for dynamic arrays is to double the size of an array when it gets full. To reduce the memory consumption, a hint on how large the number of factors z might get is advantageous to know for a dynamic LZ trie data structure. We provide such a hint based on the following lemma:

Lemma 3.1 ([3, 34]). *The number of LZ78 factors z is in the range $\sqrt{2n + 1/4} - 1/2 \leq z \leq cn/\lg_\sigma n$, for a fixed constant $c > 0$.*

At the beginning of the factorization, we let a dynamic trie reserve so much space such that it can store at least $\sqrt{2n}$ elements without resizing. On enlarging a dynamic trie, we usually double its size. But if the

number of remaining characters r to parse is below a certain threshold, we try to scale the data structure up to a value for which we expect that all factors can be stored without resizing the data structure again. Let z' be the currently computed number of factors. If $r < n/2$ we use $z' + 3r/\lg r$ as an estimate (the number 3 is chosen empirically), derived from $z - z' = \mathcal{O}(r/\lg_\sigma r)$ based on Lemma 3.1, otherwise we use $z' + z'r/(n - r)$ derived from the expectation that the ratio between z' and $n - r$ will be roughly the same as between z and n (interpolation).

3.1 Deterministic LZ Tries

We first recall two trie implementations using arrays to store the node labeled with x at position x, for each x with $1 \leq x \leq z$.

Binary Search Trie. The first-child next-sibling representation binary maintains its nodes in three arrays. A node stores a character, a pointer to one of its children, and a pointer to one of its siblings. We do not sort the nodes in the trie according to the character on their incoming edge,

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

Figure 4: Array data structures of binary built on the example given in Figure 3

but store them in the order in which they are inserted. (We found this faster in our experiments.) binary takes $2z \lg z + z \lg \sigma$ bits when storing z nodes. See Figure 4 for an example.

Ternary Search Trie. A node of the Ternary Search Trie [5] 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. Similar to binary, we do not rearrange the nodes. ternary takes $3z \lg z + z \lg \sigma$ bits when storing z nodes.

3.2 LZ Tries with Hashing

We use a hash table $H[0..M - 1]$ for a natural number M, and a hash function h to store key-value pairs. We determine the position of a pair (k, v) in H by the ***initial address*** $h(k) \bmod M$; we handle collisions with linear probing. We enlarge H when the maximum number of entries $m := \alpha M$ is reached, where α is a real number with $0 < \alpha < 1$.

A hash table can simulate a trie as follows: Given a trie edge (u, v) with label c, we use the unique key $c + \sigma \ell$ 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.

Table Size. We choose the hash table size M to be a power of two. Having $M = 2^k$ 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, i.e., $h(x) \bmod 2^k = h(x) \& (2^k - 1)$, which is practically faster[2].

If the aforementioned resize hint suggests that the next power of two is sufficient for storing all factors, we set $\alpha = 0.95$ before enlarging the size (if necessary). We also implemented a hash table variant that will change its size to fit the provided hint. This variant then cannot use the fast bit mask to simulate the operation $\bmod M$. Instead, it uses a practical alternative that scales the hash value by M and divides this value by the largest possible hash value[3], i.e., $Mh(k)/(\max_{k'} h(k'))$. We mark those hash table variants with a plus sign, e.g., hash+ is the respective variant of hash.

3.2.1 Compact Hashing

In terms of memory, hash is at a disadvantage compared to binary, because the key-value pairs consist of two factor indices and a character; for an $\alpha < 1$, hash will always take more space than binary. To reduce the size of the stored keys, we introduce the representation cht using compact hashing.

The idea of compact hashing [16, 9] is to use a bijective hash function such that when storing a tuple with key k in H, we only store the value and the quotient $\lfloor h(k)/M \rfloor$ in the hash table. The original key of an

---

[2]http://blog.teamleadnet.com/2012/07/faster-division-and-modulo-operation.html
[3]http://www.idryman.org/blog/2017/05/03/writing-a-damn-fast-hash-table-with-tiny-memory-footprints/

entry of H can be restored by knowing the initial address $h(k) \mod M$ and the stored quotient $\lfloor h(k)/M \rfloor$. To address collisions and therefore the displacement of a stored entry due to linear probing, Cleary [7] adds two bit vectors with which the initial address can be restored.

For the bijective hash function h, we consider two classes:

The class of linear congruential generators (LCGs). The class of LCGs [6] contains all functions $\mathsf{lcg}_{a,b,p} : [0..p-1] \to [0..p-1], x \mapsto (ax+b) \mod p$ with $p \in \mathbb{N}, 0 < a < p, 0 \le b < p$. If p and a are relative prime, then there exists a unique inverse $a^{-1} \in [1..p-1]$ of a such that $aa^{-1} \mod p = 1$. Then $\mathsf{lcg}_{a,b,p}^{-1} : y \mapsto (y-b)a^{-1}$ $\mod p$ is the inverse of $\mathsf{lcg}_{a,b,p}$. If p is prime, then $a^{-1} = a^{p-2} \mod p$ due to Fermat's little theorem.

The class of xorshift functions. The xorshift hash function class [23] contains functions that use shift- and exclusive or (xor) operations. Let $\oplus$ denote the binary xor-operator and w the number of bits of the input integer. For an integer $j < -\lfloor w/2 \rfloor$ or $j > \lfloor w/2 \rfloor$, the xorshift operation $\mathsf{sxor}_{w,j} : [0..2^w - 1] \to [0..2^w - 1], x \mapsto \left(x \oplus \left(\lfloor 2^j x \rfloor \mod 2^w \right) \right) \mod 2^w$ is inverse to itself: $\mathsf{sxor}_{w,j} \circ \mathsf{sxor}_{w,j} = \mathsf{id}$.

It is possible to create a bijective function that is a concatenation of functions of both families[4].

A compact hash table can use less space than a traditional hash table if the size of the keys is large: If the largest integer key is u, then all keys can be stored in $\lceil \lg u \rceil$ bits, whereas all quotients can be stored in $\lceil \lg(\max_u h(u)/M) \rceil$ bits. By choosing the hash function carefully, it is possible to store the quotients in a number of bits independent of the number of the keys.

Enlarging the hash table. On enlarging the hash table, we choose a new hash function, and rebuild the entire table with the new size and a newly chosen hash function. We first choose a hash function h out of the aforementioned bijective hash classes and adjust h's parameters such that h maps from $[0..m\sigma - 1]$ to $[0..2m\sigma - 1]$ (m has already its new size). This means that

- we select a function $\mathsf{lcg}_{a,b,p}$ with a prime $m\sigma < p < 2m\sigma$ (such a prime exists [30] and can be precomputed for all $M = 2^k$, $1 \le k \le \lg n$) and $0 < a, b \le p$ randomly chosen, or that

- we select a function $\mathsf{sxor}_{w,j}$ with $\lg(m\sigma) \le w \le \lg(2m\sigma)$ and j arbitrary.

The hash table always stores trie nodes with labels that are at most m; this is an invariant due to the following fact: before inserting a node with label $m + 1$ we enlarge the hash table and hence update m. Therefore, the key of a node can be represented by a $\lceil \lg(m\sigma) \rceil$-bit integer (we map the key to a single integer with $[0..m - 1] \times [0..\sigma - 1] \to [0..m\sigma - 1], (y, c) \mapsto \sigma y + c$. Since h is a bijection, the function $[0..m\sigma - 1] \to [0..M - 1] \times [0.. \lfloor (2m\sigma - 1)/M \rfloor], i \mapsto (h_1(i), h_2(i)) := (h(i) \mod M, \lfloor h(i)/M \rfloor)$ is bijective, too. We use h_1 to find the locations of the entries in of our hash table H. When we want to store a node with label x and key $y\sigma + c$ in the hash table, we put x and $h_2(\sigma y + c)$ in an entry of the hash table (the entry is determined by h_1, the linear probing strategy, and a re-arrangement with the bit vectors). In total, we need $M(\lg(2a\sigma) + \lg m) + 2M$ bits. Since $m \le 2z - 1$ there is a power of two such that $M = 2^{\lfloor \lg(z/\alpha) \rfloor + 1} \le (2z-1)/\alpha$. On termination, the compact hash table takes at most $M(2 + \lg(2a\sigma m)) \le (2z-1)(3 + \lg(\alpha\sigma z))/\alpha$ bits. The memory peak is reached when we have to copy the data from the penultimate table to the final hash table with the above size. The memory peak is at most $M(3 + \lg(m\alpha\sigma)) + M/2(2 + \lg(m\alpha\sigma)) \le (2z - 1)(11 + 3\lg(z\alpha\sigma))/2\alpha$.

If we compare this peak with the approach using a classic hash table (where we need to store the full key), we get a size of $M(\lg m + \lg m + \lg \sigma) + M/2(\lg(m/2) + \lg(m/2) + \lg \sigma) \le 3(2z - 1)(4/3 + \lg(\sigma z^2))/\alpha$ bits.

This gives the following theorem:

Theorem 3.2. *We can compute the LZ78 and LZW factorization online using linear time with high probability and at most $z(3\lg(z\sigma\alpha) + 11)/\alpha$ bits of working space, for a fixed α with $0 < \alpha < 1$.*

For the evaluation, we use a preliminary version of the implementation of Poyias et al. [28] that is based on [7] with the difference that Cleary uses bidirectional probing ([28] uses linear probing).

---

[4]Popular hash functions like MurmurHash 3 (https://github.com/aappleby/smhasher) use a post-processing step that applies multiple LCGs $\mathsf{lcg}_{a,0,2^{64}}$ with a as a predefined odd constant, and some xorshift-operations.

Trie	Space Best Case (bits)	Space Worst Case (bits)
binary	$3z(\lg(z^2\sigma) - 2/3)/2$	$3z(\lg(z^2\sigma) + 4/3)$
ternary	$3z(\lg(z^3\sigma) - 1)/2$	$3z(\lg(z^3\sigma) + 2)$
hash	$3z(\lg(z^2\sigma) - 2/3)/2\alpha$	$6z(\lg(z^2\sigma) + 4/3)/\alpha$
cht	$3z(\lg(\alpha z\sigma) + 8/3)/2\alpha$	$3z(\lg(\alpha z\sigma) + 11/3)/\alpha$
rolling	$3z(w + \lg(z\sigma) - 1/3)/2\alpha$	$6z(w + \lg(z\sigma) + 2/3)/\alpha$

Figure 5: Upper and lower bound of the maximum memory used during an LZ78/LZW factorization with z factors. The size of a fingerprint is w bits.

3.2.2 Rolling Hashing

Here, we present an alternative trie representation with hashing, called rolling. The idea is to maintain the Karp-Rabin fingerprints [15] 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 $T[i+\ell]$. To compute the fingerprints, we choose one of the two rolling hash function families:

- a function of the randomized Karp-Rabin *ID37* family [19][5], and

- the function $\textbf{\textit{fermat}}(T) = \sum_{i=1}^{|T|}(T[i] - 1)(\sigma + 1)^{|T|-i} \mod 2^w$, where the modulo by the word size w surrogates the integer overflow, and $T[i] - 1$ is in the range $[0..\sigma-1]$. In the case of a byte alphabet, $\sigma + 1 = 2^8 + 1 = 257$ is a Fermat prime. We compute $\textbf{\textit{fermat}}(T)$ with Horner's rule.

The LZ78/LZW computation 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).

3.2.3 Summary

We summarize the description of the trie data structures in this and the previous section by Figure 5 showing the maximum space consumption of each described trie. The maximum memory consumption is due to the peak at the last enlargement of the dynamic trie data structure, i.e., when the trie enlarges its space such that $z \le m \le 2z - 1$ (where m is the number of elements it can maintain).

4 Experiments and Conclusion

We implemented the LZ tries in the tudocomp framework [8][6]. The framework provides the implementation of an LZ78 and an LZW compressor. Both compressors are parameterized by an LZ trie and a coder. The coder is a function that takes the output of the factorization and generates the final binary output. We selected the coder `bit` that stores the referred index y (with $y > 0$) of a factor F_x in $\lceil \lg x \rceil$ bits. That is because the factor F_x can have a referred index y only with $y < x$. We can restore the coded referred index on decompression since we know the index of the factor that we currently process and hence the number of bits used to store its referred index (if we coded it)[7]. This yields $\sum_i^z \lceil \lg i \rceil = z \lceil \lg z \rceil - (\lg e)z + \mathcal{O}(\lg z)$ bits for storing the (positive) referred indices. The additional characters in LZ78 and the negative referred indices in LZW are output naively as 8-bit integers.

The LZ78 and LZW compressor are independent of the LZ trie implementation, i.e., all trie data structures described in the previous sections can be plugged into the LZW or LZ78 compressor easily. We additionally incorporated the following trie data structures into tudocomp:

---

[5]`https://github.com/lemire/rollinghashcpp`

[6]The source code of our implementations is freely available at `https://github.com/tudocomp`, except for `cht` and `bonsai` due to copyright restrictions.

[7]this approach is similar to `http://www.cplusplus.com/articles/iL18T05o`

cedar: the Cedar trie [32], representing a trie using two arrays.

judy: the Judy array, advertised to be optimized for avoiding CPU cache misses (cf. [21] for an evaluation).

bonsai: the m-Bonsai (γ) trie [27] representing a trie whose nodes are not labeled. It uses a compact hash table, but unlike our approach, the key consists of the position of the parent in the hash table (instead of the label of the parent) and the character. Due to this fact, we need to traverse the complete trie for enlarging the trie. We store the labels of the trie nodes in an extra array.

The data structures are realized as C++ classes. We added a lightweight wrapper around each class providing the same interface for all tries.

Online Feature. Given an input stream with known length, we evaluate the online computation of the LZ78 and LZW compression for different LZ trie representations. We assume that Σ is a byte alphabet, i.e., $\sigma = 2^8$. On computing a factor, we encode it and output it instantaneously. This makes our compression program a filter [24], i.e., it processes the input stream and generates an output stream, buffering neither the input nor the output.

Implementation Details. The keys stored by hash are stored in integers with a width of 40 bit. The values stored by hash, rolling and bonsai are 32-bit integers. For all variants working with hash tables, we initially set α to 0.3.

According to the birthday paradox, 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. We used 64-bit fingerprints because, unlike 32-bit fingerprints, the factorization produced by rolling are correct for all test instances and all tested rolling hash functions.

Hash Function. We use cht with a hash function of the LCG family. Our hash table for hash uses a xorshift hash function[8] derived from [29]. It is slower than simple multiplicative functions, but more resilient against clustering. Alternatives are sophisticated hash functions like CLHash [20] or Zobrist hashing [35, 18]. These are even more resilient against clustering, but have practical higher computation times in our experiments.

Setup. The experiments were conducted on a machine with 32 GB of RAM, an Intel Xeon CPU E3-1271 v3 and a Samsung SSD 850 EVO 250GB. The operating system was a 64-bit version of Ubuntu Linux 14.04 with the kernel version 3.13. We used a single execution thread for the experiments. The source code was compiled using the GNU compiler g++ 6.2.0 with the compile flags -O3 -march=native -DNDEBUG.

Datasets. We evaluated the combinations of the aforementioned tries with the LZW and LZ78 algorithms on the 200MiB text collections provided by tudocomp. We assume that the input alphabet is the byte alphabet ($\sigma = 2^8$). The indices of the factors are represented with 32-bit integers. Due to space restrictions, we choose PC-ENGLISH as a representative for a standard English text and PCR-CERE as a representative for a highly-repetitive text with small alphabet size (the evaluation on the other datasets were quite similar). We plotted the memory consumption against the time (in logarithmic scale) for both datasets in Figure 6 and Figure 7. To avoid clutter, we selected one hash function per rolling hash table: We chose *fermat* with rolling and *ID37* with rolling+ for the plots. We additionally added the number of LZ77 factors [33] as a reference.

Overall Evaluation. The evaluation shows that the fastest option is rolling. The size of its fingerprints is a trade-off 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 is the second fastest LZ trie in the experiments. With 40-bit keys it uses less memory than rolling, but is slightly slower. Depending on the quality of the resize hint, the variants hash+ and rolling+ take 50% up to 100% of the size of hash and rolling, respectively. hash+ and rolling+ are always slower than their respective standard variants, sometimes slower than the deterministic data structures ternary and binary. binary's speed excels at texts with very small alphabets, while ternary usually outperforms binary. Only cht can compete with binary in terms of space, but is magnitudes slower than most alternatives. The third party data structures cedar, bonsai and judy could not make it to the Pareto front.

---

[8] http://xorshift.di.unimi.it/splitmix64.c

Figure 6: Evaluation of LZ78/LZW on the English text PC-ENGLISH with $\sigma = 226$. Left: LZ78 factorization with $z = 21.4\mathrm{M}$, $\lceil \lg z \rceil = 25$ and $z \lg z + z \lg \sigma = 83.8\mathrm{MiB}$. The compressed file size is 80.2MiB. Right: LZW factorization with $z = 23.5\mathrm{M}$, $\lceil \lg(255 + z) \rceil = 25$ and $z \lg(255 + z) = 70.13\mathrm{MiB}$. The compressed file size is 66.1MiB. The LZ77 factorization consists of $z = 14\mathrm{M}$ factors, and can be stored in $2z \lg n = 93.3\mathrm{MiB}$.

Evaluation of rolling. The hash table with the rolling hash function *fermat* is slightly faster than with a function of the *ID37* family, but the hash table with *fermat* tends to have more collisions (cf. Table 1). It is magnitudes slower at less compressible texts like PC-PROTEINS due to the high occurrence of collisions. The number of collisions can drop if we post-process the output of *fermat* with a hash function that is more collision resistant (like the hash function used for hash). Applying the hash function on *fermat* speeds up the computation only if the number of collisions is sufficiently high (e.g., rolling+ with *fermat* in Table 1).

The domain of the Karp-Rabin fingerprints can be made large enough to be robust against collisions when hashing large texts. In our case, 64-bit fingerprints fitting in one computer word were sufficient. Checking whether the factorization is correct can be done by reconstructing the text with the output and the built LZ trie. However, a compression with rolling combined with a decompression step takes more time than other approaches like hash or binary. Hence, a Las Vegas algorithm based on rolling is practically not interesting.
Outlook. An interesting option is to switch from the linear probing scheme to a more sophisticated scheme whose running time is stable for high loads, too [22]. This could be especially beneficent if the resize hint provides a more accurate lower bound on the number of factors.

Speaking of novel hash tables, we could combine the compact hash table [7] with the memory management of Google's sparse hash table[9] leading to an even more memory friendly trie representation.
Acknowledgements. We are grateful to Patrick Dinklage for spell-checking the paper, and to Marvin Löbel for providing the basement of the LZ78/LZW framework in tudocomp. Further, we thank Andreas Poyias for sharing the source code of the m-Bonsai trie [27] and the compact hash table [28].

References

[1] D. Arroyuelo and G. Navarro. Space-efficient construction of Lempel-Ziv compressed text indexes. *Inf. Comput.*, 209(7):1070–1102, 2011.

[2] J. Arz and J. Fischer. LZ-compressed string dictionaries. In *Proc. DCC*, pages 322–331. IEEE Press, 2014.

---

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

Figure 7: Evaluation of LZ78/LZW on the repetitive DNA sequence PCR-CERE with $\sigma = 6$. Left: LZ78 factorization with $z = 15.8$M, $\lceil \lg z \rceil = 24$ and $z \lg z + z \lg \sigma = 50.0$MiB. The compressed file size is 80.2MiB. Right: LZW factorization with $z = 17.1$M, $\lceil \lg(255 + z) \rceil = 25$ and $z \lg(255 + z) = 50.9$MiB. The compressed file size is 50.9MiB. The LZ77 factorization consists of $z = 1.38$M factors, and can be stored in $2z \lg n = 9.66$MiB.

[3] H. Bannai, S. Inenaga, and M. Takeda. Efficient LZ78 factorization of grammar compressed text. In *Proc. SPIRE*, volume 7608 of *LNCS*, pages 86–98. Springer.

[4] D. Belazzougui and S. J. Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proc. SODA*, pages 2053–2071. SIAM, 2016.

[5] J. L. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proc. SODA*, pages 360–369. ACM/SIAM, 1997.

[6] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[7] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE Trans. Computers*, 33(9): 828–834, 1984.

[8] P. Dinklage, J. Fischer, D. Köppl, M. Löbel, and K. Sadakane. Compression with the tudocomp framework. In *Proc. SEA*, 2017, to appear; ArXiv CoRR, 1702.07577.

[9] J. A. Feldman and J. R. Low. Comment on Brent's scatter storage algorithm. *Commun. ACM*, 16(11): 703, 1973.

[10] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Proc. CPM*, volume 9133 of *LNCS*, pages 160–171. Springer, 2015.

[11] J. Fischer, T. I, and D. Köppl. Lempel-Ziv computation in small space (LZ-CISS). In *Proc. CPM*, volume 9133 of *LNCS*, pages 172–184. Springer, 2015.

[12] J. Jansson, K. Sadakane, and W.-K. Sung. Linked dynamic tries with applications to LZ-compression in sublinear time and space. *Algorithmica*, 71(4):969–988, 2015.

[13] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proc. SEA*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013.

Trie	#Collisions	M	memory	time
rolling with				
- *ID37*	36M	33.6M	576.0MiB	11.6s
- *fermat*	137M	33.6M	576.0MiB	11.4s
- *fermat⊕*	36M	33.6M	576.0MiB	11.8s
rolling+ with				
- *ID37*	140M	24.0M	466.9MiB	14.7s
- *fermat*	938M	24.0M	466.9MiB	21.0s
- *fermat⊕*	142M	24.0M	466.9MiB	15.8s
hash	36M	33.6M	432.0MiB	15.3s
hash+	137M	24.0M	350.2MiB	19.1s

Table 1: Detailed evaluation of the tries using hashing. We evaluated the number of collisions and the final table size M for the LZ78 factorization of 200 MiB PC-ENGLISH. $\oplus$ denotes that the output of the rolling hash function is plugged into the xorshift hash function used by hash. An entry in rolling costs $64 + 32$ bits, an entry in hash $40 + 32$ bits.

[14] J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lazy Lempel-Ziv factorization algorithms. *ACM Journal of Experimental Algorithmics*, 21(1):2.4:1–2.4:19, 2016.

[15] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[16] D. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley, 1973.

[17] D. Köppl and K. Sadakane. Lempel-Ziv computation in compressed space (LZ-CICS). In *Proc. DCC*, pages 3–12. IEEE Press, 2016.

[18] D. Lemire. The universality of iterated hashing over variable-length strings. *Discrete Applied Mathematics*, 160(4-5):604–617, 2012.

[19] D. Lemire and O. Kaser. Recursive n-gram hashing is pairwise independent, at best. *Computer Speech & Language*, 24(4):698–710, 2010.

[20] D. Lemire and O. Kaser. Faster 64-bit universal hashing using carry-less multiplications. *J. Cryptographic Engineering*, 6(3):171–185, 2016.

[21] H. Luan, X. Du, S. Wang, Y. Ni, and Q. Chen. J$^+$-tree: A new index structure in main memory. In *Proc. DASFAA*, volume 4443 of *LNCS*, pages 386–397. Springer, 2007.

[22] T. Maier and P. Sanders. Dynamic Space Efficient Hashing. *ArXiv CoRR, 1705.00997*.

[23] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14), 2003.

[24] M. D. McIlroy. A research UNIX reader: Annotated excerpts from the programmer's manual, 1971–1986. Technical Report CSTR 139, AT&T Bell Laboratories, 1987.

[25] J. I. Munro, G. Navarro, and Y. Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proc. SODA*, pages 408–424. SIAM, 2017.

[26] Y. Nakashima, T. I, S. Inenaga, H. Bannai, and M. Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *Inform. Process. Lett.*, 115(9):655 – 659, 2015.

[27] A. Poyias and R. Raman. Improved practical compact dynamic tries. In *Proc. SPIRE*, volume 9309 of *LNCS*, pages 324–336. Springer, 2015.

[28] A. Poyias, S. J. Puglisi, and R. Raman. Compact dynamic rewritable (CDRW) arrays. In *Proc. ALENEX*, pages 109–119. SIAM, 2017.

[29] G. L. Steele Jr., D. Lea, and C. H. Flood. Fast splittable pseudorandom number generators. In *Proc. OOPSLA*, pages 453–472. ACM, 2014.

[30] P. Tchebychev. Mémoire sur les nombres premiers. *Journal de mathématiques pures et appliquées*, 1: 366–390, 1852.

[31] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[32] N. Yoshinaga and M. Kitsuregawa. A self-adaptive classifier for efficient text-stream processing. In *Proc. COLING*, pages 1091–1102. ACL, 2014.

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

[34] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

[35] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Sciences Department, University of Wisconsin, 1970.

| Trie | PC-ENGLISH | | | | PCR-CERE | | | |
| | LZ78 | | LZW | | LZ78 | | LZW | |
	Time	Space	Time	Space	Time	Space	Time	Space
`hash` with hash table								
`std::unordered_map`	51.0s	856.6MiB	54.0s	937.9MiB	42.3s	703.2MiB	44.1s	760.8MiB
`std::map`	161.2s	980.2MiB	167.2s	1.1GiB	98.8s	722.5MiB	104.6s	781.6MiB
`rigtorp`[a]	14.9s	960.0MiB	15.2s	960.0MiB	12.0s	960.0MiB	12.3s	960.0MiB
`flathash`[b]	33.5s	24GiB	24.5s	24GiB	18.5s	6GiB	19.2s	6GiB
`flathash`[c]	15.1s	1.3GiB	15.7s	1.3GiB	12.4s	1.3GiB	13.0s	1.3GiB
`densehash`[d]	23.0s	576.0MiB	24.4s	576.0MiB	29.4s	576.0MiB	30.8s	576.0MiB
`sparsehash`[d]	49.1s	255.7MiB	52.2s	280.0MiB	68.6s	191.3MiB	72.4s	206.1MiB
LZ-index [7]	24.6s	1047MiB			14.5s	817.3MiB		

Table 2: `hash` with different hash tables, and the LZ-index.

[a] `https://github.com/rigtorp/HashMap`, $\alpha = 0.5$ hard coded
[b] `https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/`, it uses the identity as a hash function and doubles its size when experiencing too much collisions
[c] See Footnote b, but with our default hash function
[d] `https://github.com/sparsehash/sparsehash`

A   Variations of Hash Tables

The trie representation `hash` can be generalized to be used with any associative container. The easiest implementation is to use the balanced binary tree `std::map` or the hash table `std::unordered_map` provided by the standard library of C++11. `std::unordered_map` is conform to the interface of the C++ standard library, but therefore sacrifices performance. It uses separate chaining that tends to use a lot of small memory allocations affecting the overall running time (see Table 2). Another pitfall is to use the standard C++11 hash function for integers that is just the identity function. Although this is the fastest available hash function, it performs poorly in the experiments. There are two reasons. The first is that $k \mapsto k \mod M$ badly distributes the tuples if M is not a prime. The second is that the input data is not independent: In the case of LZ78 and LZW, the composed key $c + \ell\sigma$ of a node v connected to its parent with label ℓ by an edge with label c holds information about the trie topology: all nodes whose keys are $\ell\sigma + d$ for a $d \in \Sigma$ are the siblings of v. Since ℓ is smaller than the label of v (ℓ is the referred index of the factor corresponding to v), larger keys depend on the existence of some keys with smaller values. Both problems can be tackled by using a hash function with an avalanche effect property, i.e., flipping a single bit of the input changes roughly half of the bits of the output. In Table 2 we evaluated the identity and our standard hash function (see Footnote 8) as hash functions for the hash table `flathash`, which seems to be very sensitive for hash collisions.

We selected the LZ trie of the LZ-index [7] as an external competitor in Table 2. We terminated the execution after producing the LZ trie of the LZ78 factorization. We did not integrate this data structure into tudocomp.

The only interesting configuration is `hash` with the hash table `sparsehash`, since it takes 4.1MB less space than `binary` while still being faster than `cht`, at the LZ78-factorization of PCR-CERE.

B   Reasons for Linear Probing

Collisions in our hash table are resolved by linear probing. Linear probing inserts a tuple with key k at the first free entry, starting with the entry at index $h(k) \mod M$. Linear probing is cache-efficient if the keys have a small bit width (like fitting in a computer word). Using large hash tables and small keys, the

Figure 8: Evaluation of LZ78/LZW on the tab-spaced-version file HASHTAG with $\sigma = 179$. Left: LZ78 factorization with $z = 18.9$M, $\lceil \lg z \rceil = 25$ and $z \lg z + z \lg \sigma = 73.4$MiB. The compressed file size is 70.6MiB. Right: LZW factorization with $z = 21.1$M, $\lceil \lg(255 + z) \rceil = 25$ and $z \lg(255 + z) = 62.9$MiB. The compressed file size is 58.9MiB. The LZ77 factorization consists of $z = 13.7$M factors, and can be stored in $2z \lg n = 90.4$MiB.

cache-efficiency can compensate the chance of higher collisions [1, 4]. Linear probing excels if the load ratio is below 50%, and it is still competitive up to a load ratio of 80% [6, 2]. Nevertheless, its main drawback is *clustering*: Linear probing creates runs, i.e., entries whose hash values are equal. With a sufficient high load, it is likely that runs can merge such that long sequence of entries with different hash values emerge. When trying to look up a key k, we have to search the sequence of succeeding elements starting at the initial address until finding a tuple whose key is k, or ending at an empty entry. Fortunately, the expected time of a search is rather promising for an α not too close to one: Given that the used hash function h distributes the keys independently and uniformly, we get $\mathcal{O}\big(1/(1 - \alpha)^2\big)$ expected time for a search [5]. In practice, even weak hash functions (like we use in this article) tend to behave as truly independent hash functions [3]. These properties convinced us that linear probing is a good candidate for our representations of the LZ trie using a hash table.

C More Evaluation

We additionally evaluated the presented trie data structures on two other datasets in Figures 8 and 9, showing similar characteristics as the plots in Figures 6 and 7.

References

[1] N. Askitis. Fast and compact hash tables for integer keys. In *Proc. ACSC*, volume 91 of *CRPIT*, pages 101–110. Australian Computer Society, 2009.

[2] J. R. Black, C. U. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Proc. WAE*, pages 37–48. Max-Planck-Institut für Informatik, 1998.

[3] K. Chung, M. Mitzenmacher, and S. P. Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. *Theory of Computing*, 9:897–945, 2013.

Figure 9: Evaluation of LZ78/LZW on the DNA sequence PC-DNA with $\sigma = 17$. Left: LZ78 factorization with $z = 16.4M$, $\lceil \lg z \rceil = 24$ and $z \lg z + z \lg \sigma = 54.8\text{MiB}$. The compressed file size is 60.5MiB. Right: LZW factorization with $z = 17.8M$, $\lceil \lg(255 + z) \rceil = 25$ and $z \lg(255 + z) = 52.9\text{MiB}$. The compressed file size is 48.9MiB. The LZ77 factorization consists of $z = 13.9M$ factors, and can be stored in $2z \lg n = 92.1\text{MiB}$.

[4] G. L. Heileman and W. Luo. How caching affects hashing. In *Proc. ALENEX*, pages 141–154. SIAM, 2005.

[5] D. Knuth. *Sorting and Searching*, volume III of *The Art of Computer Programming*. Addison-Wesley, 1973.

[6] T. Maier and P. Sanders. Dynamic Space Efficient Hashing. *ArXiv CoRR, 1705.00997*.

[7] G. Navarro. Implementing the LZ-index: Theory versus practice. *ACM Journal of Experimental Algorithmics*, 13(2):2:1.1–2:1.49, 2008.