Top-k Frequent Patterns in Streams and Parameterized-Space LZ Compression

Patrick Dinklage
TU Dortmund University, Germany

Johannes Fischer
TU Dortmund University, Germany

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

We present novel online approximations of the Lempel-Ziv 77 (LZ77) and Lempel-Ziv 78 (LZ78) compression schemes [Lempel & Ziv, 1977/1978] with parameterizable space usage based on estimating which k patterns occur the most frequently in the streamed input for parameter k. This new approach overcomes the issue of finding only local repetitions, which is a natural limitation of algorithms that compress using a sliding window or by partitioning the input into blocks. For this, we introduce the *top-k trie*, a summary for maintaining online the top-k frequent consecutive patterns in a stream of characters based on a combination of the Lempel-Ziv 78 compression scheme and the Misra-Gries algorithm for frequent item estimation in streams. Using straightforward encoding, our implementations yield compression ratios (output over input size) competitive with established general-purpose LZ-based compression utilities such as gzip or xz.

2012 ACM Subject Classification Theory of computation → Data compression; Theory of computation → Pattern matching; Theory of computation → Sketching and sampling

Keywords and phrases compression, streaming, heavy hitters, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.9

Supplementary Material *Software (Source Code)*: `https://github.com/pdinklag/top-k-compress` archived at `swh:1:dir:c586220ee1ae265be3a77be75b09f2fdbc55843c`

Funding *Patrick Dinklage*: Funded by the Deutsche Forschungsgemeinschaft (DFG) under the Research Grants programme (project No. 501086801).
Johannes Fischer: Funded by the Deutsche Forschungsgemeinschaft (DFG) under the Research Grants programme (project No. 501086801).
Nicola Prezza: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements The authors gratefully acknowledge the computing time provided on the Linux HPC cluster at Technical University Dortmund (LiDO3), partially funded in the course of the Large-Scale Equipment Initiative by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as project 271512359.

1 Introduction

Lempel-Ziv schemes are arguably among the most popular both in theory and practice of compressing data. The main idea is to identify repeating patterns and replace them by references to other occurrences. Compression is achieved if these references can be encoded with less bits than the sequence they replace.

The memory requirement for computing Lempel-Ziv parsings, however, is a major practical issue. If inputs become large, linear dependency on the input or output size becomes prohibitive. One of the most common approaches to tackle this issue is by processing the input using a *sliding window*: the memory requirement then depends only on the (paramaterizable) window size. As a trade-off, the detection of repetitions is limited to within the window. Despite this, sliding windows back popular everyday compression utilities such as *gzip* (very small window) or *xz*/*7-zip* (arbitrarily large window).

In this work, we attempt to alleviate the issue of being able to find only local repetitions and instead get a global sense of repeating patterns, all while keeping the space requirement parameterizable. The key to our approach is the notion of *heavy hitters*: if we can estimate which k patterns are the most frequent at any time while streaming the input, that helps us find repetitions globally. Because the patterns are *frequent*, the hope is that they also contribute the most to compression. We can encode them as references into a universe of size k, i.e., an index into the k currently frequent patterns. The produced output can then be decoded by following the same protocol. Because we never account for more than k patterns, the parameter k directly controls our space consumption.

Related Work

Mining frequent items in a stream has a long history of research. Aggarwal and Yu [2] give a survey on established techniques to estimate the frequencies of items. In this work, we use and adapt the algorithm due to Misra and Gries [28] and the Space-Saving data structure due to Metwally et al. [27] to estimate online the k most frequent items (outlined in section 2.1).

Maruyama and Tabei [25] use an idea similar to our work for grammar compression. Their algorithm computes online and in $\mathcal{O}(k)$ space a straight-line program that produces the input. They explore methods for estimating online the k most frequent production rules, in the hope that these contribute the most for compression, using techniques similar to what we describe in section 3. On large genomic sequences (hundreds of gigabytes), their approach achieves good compression rates with impressively short running times. However, they make heavy use of the fact that genomic sequences have a very small alphabet of nucleotide bases, and it is not clear how their algorithm performs for larger alphabets. Another related result is that of De Agostino [3], who considered a constrained-dictionary version of the Lempel-Ziv 78 scheme (LZ78). The dictionary is limited to size k and if it is full, a heuristic is used to determine which entry to replace next, with the focus being on the *last recently used* (LRU) heuristic and variants. Their experiments targeted inputs of few megabytes and very small dictionaries, and they did not focus on performance other than the constrained memory consumption. To that end, their results are difficult to project to larger inputs. The general idea of constraining the size of the LZ78 dictionary, however, matches our work. Apart from this, there has been work on finding trade-offs between the running time and space consumption of LZ78 parsers, Arroyuelo et al. [6] give a recent overview.

Apart from these, there has been work on identifying frequent *itemset subsequences*, Gan et al. give a survey in [17]. The problem is much more general than what we consider, as (1) the subsequences need not be consecutive in the stream and (2) each item in a subsequence may be any one from a set. Apostolico et al. [5] use mining techniques to find so-called *motifs* later used for Lempel-Ziv-Welch compression. Motifs can roughly be considered strings with wildcards, and thus their scenario is more related to mining frequent subsequences than it is to mining frequent consecutive patterns. The bioinformatics community has done work on estimating the *number* of distinct k-mers, or a histogram thereof, in streamed genomic sequences [26, 29, 31, 8] (here, k refers to the fixed length of relevant substrings and is not

to be confused with top-k). However, their algorithms are not concerned about what *are* the most frequent k-mers. Furthermore, they aim at a fixed pattern length k and on a known constant-size alphabet, properties that cannot be exploited without loss of generality. Fischer et al. [14] consider the problem of string mining under frequency constraints, which is more similar to our scenario. Their solution is optimal in terms of accuracy, but it requires a full-text index and therefore disqualifies in an online scenario. It furthermore finds patterns that have a given minimal support in the input, for which we know no direct translation to maintaining the k most frequent patterns.

Our Contributions

We introduce the *top-k trie* (section 3), a framework to estimate and maintain online the k most frequent patterns in a stream. It uses space at most $\mathcal{O}(k)$ and requires constant amortized time per input character. We apply this to online approximations of the LZ78 (section 4) and LZ77 (section 5) parsings, which we implement and compare against established general-purpose compressors (section 6). Our algorithms are competitive in terms of speed and compression ratio albeit using a straightforward encoding. We finally analyze the impact of the parameter k on resource usage and the number of emitted phrases.

2 Preliminaries

Let $s \in \Sigma^*$ be a string over an alphabet Σ. For $i \in [1, |s|]$, we denote by $s[i]$ the i-th character of s. Let $j \in [1, |s|]$ and $j \geq i$, then $s[i..j]$ denotes the (consecutive) *substring* of characters from position i and j, both included, of s. For analysis, we use the *word RAM* model [18], where the memory consists of contiguous words of length $\Theta(\lg n)$ bits each (by default, we state logarithms as base-2). We can access and perform arithmetic operations on a constant number of words in constant time. We further consider a *streaming* model, particularly the *cash register* model [30], where the input is a stream $S = x_1, x_2, \ldots, x_n$ of n items drawn from some domain Γ. The items are received one by one and in non-rewindable fashion. There is no previous knowledge about the length $n \to \infty$ of the stream, and it may far exceed the size M of available memory. To that end, it is not possible to store S in memory for random access. Furthermore, nothing is known about Γ, so even storing information about all $\sigma \leq n$ distinct items that occur in S is infeasible (e.g., a histogram – see also [4]). However, we are allowed to store pointers into the stream or count the number of occurrences of a selected set of items. In that regard, we assume $M = \Omega(\text{polylog}\, n)$.

2.1 The Misra-Gries algorithm for top-k frequent item estimation

For an item $x \in \Gamma$, let $f_x \in [0, n]$ be its *frequency*, i.e., the number of occurrences of x in S. We consider the problem of finding the top-k most frequent items from Γ that occur in S. Formally, we want to find a set F with $|F| = k$ and $\forall x \in F : f_x \geq \max\{f_y \mid y \in \Gamma \setminus F\}$. As noted earlier, we cannot hope to maintain a histogram of frequencies for all the items of Γ.

The Misra-Gries algorithm [28] computes a *summary* of F in space $\mathcal{O}(k)$. For some $x \in \Gamma$, we call f'_x the *estimated frequency* of x in S. Before reading from S, we initialize $F := \emptyset$. After reading the next item x from S, we act as follows: if $x \in F$, we increment the estimated frequency $f'_x := f'_x + 1$. If $x \notin F$ and $|F| < k$, we insert x into F with $f'_x := 1$. Otherwise ($x \notin F \ \wedge \ |F| = k$), we decrement the estimated frequencies $f'_y := f'_y - 1$ for all $y \in F$ and then delete any y from F for which $f'_y = 0$.

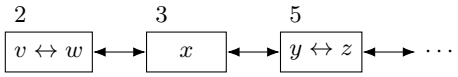

Figure 1 Space-Saving data structure for $F \supseteq \{v, w, x, y, z\}$, $f'_v = f'_w = 2$, $f'_x = 3$ and $f'_y = f'_z = 5$.

It is easy to see that $f'_x \leq f_x$ holds for any $x \in \Gamma$, that is, the Misra-Gries algorithm underestimates frequencies. As shown in [9], it holds that $f'_x \geq f_x - n/k$. The algorithm is fast in practice, even though it has a relatively high false positive rate [10, referred to as FREQUENT]. For this work, however, false positives are not a serious disadvantage.

2.2 The Space-Saving Data Structure

The Space-Saving data structure of [27], even though proposed independently, can be used to implement the Misra-Gries algorithm. For each distinct estimated frequency f' of the items in F, it maintains a *bucket*. Each bucket holds a doubly-linked list of the items $x \in F$ such that $f'_x = f'$ in arbitrary order. The buckets are also maintained in a doubly-linked list ordered by their represented frequencies f'. Figure 1 shows an example.

The data structure requires space at most $\mathcal{O}(k)$ as $|F| = k$. Creating, deleting, inserting items into or deleting items from a bucket each takes constant time thanks to the use of doubly-linked lists. Thus, a new item x with $f'_x = 1$ can be inserted in constant time by inserting it into the first bucket, and the frequency of an item y can be incremented in constant time by removing it from bucket f'_y and inserting it into bucket $f'_y + 1$.

For the Misra-Gries algorithm, we require an operation to decrement all frequencies. This can be simulated by maintaining a threshold θ that is initially zero [1]. Let $x \notin F$ and $|F| = k$. Instead of decrementing all frequencies, we first test whether there is any item y with $f'_y \leq \theta$. This can be done in constant time by testing whether the minimum bucket represents a frequency $f' \leq \theta$. If that is the case, we delete an arbitrary item y contained in the bucket and *recycle* its entry for x, inserting it with $f'_x := \theta$. Otherwise, we increment $\theta := \theta + 1$. This lazy approach simulates the decrement-all operation in constant time.

2.3 Tries

A *trie* [16] represents a set $\mathcal{S}$ of m distinct strings in a tree as follows. The root represents the empty string ε. For every string $s \in \mathcal{S}$, there is a path $\pi_s = v_0, v_1, \ldots, v_{|s|}$ of nodes such that v_0 is the root and each node v_i represents the prefix $s[1..i]$. To that end, the edge from node v_{i-1} to v_i is labeled by $s[i]$, and thus s is the concatenation of the edge labels along π_s. The total number of nodes in the trie is at most $1 + \sum_{s \in \mathcal{S}} |s|$.

2.4 Lempel-Ziv 78

The Lempel-Ziv 78 (*LZ78*) compression scheme [35] factorizes the input $S \in \Sigma^*$ into z phrases $f_1, \ldots, f_z$ such that $f_1 \cdots f_z = S$. The i-th phrase ($1 \leq i \leq z$) is $f_i = f_j \alpha$ for $j < i$ and $\alpha \in \Sigma$ such that f_j is the longest possible previous phrase such that $f_j \alpha$ does not occur in $f_1 \cdots f_{i-1}$. In case no such j exists, we say $f_j = f_0 := \varepsilon$ and $f_i = \alpha$ denotes a new character.

Standard algorithms to compute the LZ78 parsing use a trie T where each node represents a phrase and is labeled by the phrase number. Figure 2 shows an example. Suppose that we have already computed the first $i - 1$ phrases. We memorize the *current* node number j, which is initially the root (representing $j = 0$ and the empty phrase ε). When we read the next character α from S, we attempt to follow the edge at node j labeled α. If that edge

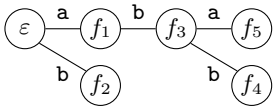

Figure 2 LZ78 trie after parsing the string `abababbaba` with $f_1 = \texttt{a}$, $f_2 = \texttt{b}$, $f_3 = \texttt{ab}$, $f_4 = \texttt{abb}$ and $f_5 = \texttt{aba}$. The root represents the empty string ε, other nodes represent one phrase each.

exists, we set j to the number of the connected node. Otherwise, node j spells out f_j, the longest phrase such that $f_j\alpha$ has no previous occurrence in S, and thus the next phrase is $f_j\alpha$, which we encode as the tuple (j, α). We create a new child node i and connect it to j with an edge labeled α. Then, we reset $j := 0$ back to the root to begin with the next phrase. This process is repeated until S has been fully parsed.

2.5 Lempel-Ziv 77

The Lempel-Ziv 77 (*LZ77*) compression scheme [34] factorizes the input $S \in \Sigma^*$ into z phrases $f_1, \ldots, f_z$ such that $f_1 \cdots f_z = S$. The i-th phrase f_i is either the first occurrence of some character $\alpha = f_i$ in S, or it is the longest possible prefix of $S[|f_1 \cdots f_{i-1}| + 1 \,..\, n]$ that already occurs previously in $S[1 \,..\, |f_1 \cdots f_i|]$.

One key difference to LZ78 is that LZ77 phrases may *overlap*. Consider, as an example, the input $S = \alpha^6$ for some character α: while the LZ78 parsing of S consists of the three phrases α, α^2 and α^3, the LZ77 parsing consists only of the two phrases $f_1 = \alpha$ and $f_2 = \alpha^5$. As a result of overlaps, LZ77 parsings typically consist of fewer phrases than LZ78 parsings for the same input, which in turn allows for smaller encodings of the compressed input.

However, compared to LZ78, computing the LZ77 parsing is more complicated. Finding the next LZ77 phrase means finding the longest previous occurrence of a prefix of the remaining input. Algorithms to compute LZ77 therefore typically make use of some kind of (possibly compressed) full-text index. In appendix C, we describe the well-known algorithm that computes the LZ77 parsing in time and space $\mathcal{O}(n)$ via the suffix array of the input. More recent examples that improve on the time and/or space requirement include [12, 19, 32, 15].

Approximations

The requirement of space linear in the input or output size becomes prohibitive in practice when the size of the input or the index data structure exceeds the available memory. It is therefore straightforward to think about a trade-off where only an approximation of LZ77 is computed in smaller – preferably parameterized – space at the cost of worse compression. Myriad approximation algorithms have been proposed that compute so-called *LZ-like* parsings via hashing [13], references [24] or samplings with synchronizing properties [12], to name but a few examples. Considering everyday compression utilities such as *gzip* or *xz*, however, the arguably most popular strategy involves using a sliding window of fixed size w. Here, the requirement of finding a longest previous occurrence is relaxed to a search only within the window. While the space is then governed only by parameter w, the number of produced phrases may increase dramatically. A common way to counter this is to encode phrases using sophisticated encoding schemes (e.g., DEFLATE [11] or LZMA[1]).

[1] To the best of our knowledge, there is no official specification of LZMA, but a de-facto specification of the XZ format is available at `https://tukaani.org/xz/xz-file-format.txt`.

Algorithm 1 Framework for maintaining online the top-k trie T_F for an input stream. In this listing, the trie is modeled as a set of node numbers where 0 denotes the root node.

```
   Input  : Stream S ∈ Σ*, parameter k ∈ ℕ
   Output : Top-k trie T_F (maintained online)
 1 T_F ← {0}, θ ← 0, v ← 0
 2 while there is another character α on S do
 3     if v has a child u with edge labeled α then
 4         f'_u ← f'_u + 1
 5         v ← u
 6     else
 7         if |T_F| < k then
 8             T_F ← T_F ∪ {new node u}                          // grow trie
 9             f'_u ← θ + 1
10             make u a child of v with edge labeled α
11         else if there is a leaf u in T_F with f'_u ≤ θ then
12             delete the edge to u                        // recycle non-frequent
13             f'_u ← θ + 1
14             make u a child of v with edge labeled α
15         else
16             θ ← θ + 1                                       // decrement all
17         v ← 0
```

3 The Top-k Trie for Frequent Pattern Estimation

Consider a stream $S \in \Sigma^*$ of characters from the alphabet Σ and the problem of maintaining online the k most frequent *substrings* (consecutive patterns of characters) occurring in S. Even if the alphabet is small (e.g., a byte alphabet, an ASCII alphabet or even nucleotide bases from a DNA sequence), the number of substrings is quadratic in the number of characters read from S and thus maintaining a histogram of frequencies for all patterns is impractical. Instead, this motivates the use of a summary.

We propose a variation of the lazy Misra-Gries algorithm that maintains the currently frequent patterns in a trie T_F of size at most k, the Space-Saving data structure containing the nodes of T_F and a threshold θ. At each node $v \in [0, k]$ in T_F, we store the estimated frequency f'_v of the pattern spelled out by the edge labels on the path from the root to v, as well as a back link into the Space-Saving data structure for constant-time access. In the remainder of this article, we refer to this data structure as the *top-k trie*. Because T_F has at most k nodes and the Space-Saving data structure can only have k distinct buckets holding a total of k entries, the space of the top-k trie is $\mathcal{O}(k)$.

We process the input stream S as shown in algorithm 1. Initially, T_F consists of only the root numbered 0. We call v the *current node*, initially the root. Upon reading the next character α from S, we find the edge labeled α from v to a child u. If the edge exists, we increment f'_u and continue with $v := u$. Otherwise, if T_F is not yet full ($|T_F| < k$, line 7), we create a new leaf u and make it a child of v with edge labeled α. If T_F is full, let u be a leaf (not an inner node, as we will discuss later) with $f'_u \leq \theta$ that we find using the Space-Saving data structure. If u exists, we *recycle* it by severing its edges (from its parent and to its children) and making it a child of v with edge label α (line 11). Otherwise, if there is no node to recycle, we increment the threshold θ to simulate the decrementation for all frequent patterns (line 15). Unless we can follow an existing edge, we reset the current node to the root (line 17). Whenever a frequency f'_u is modified for some node u, we update the Space-Saving data structure accordingly.

There is a very apparent similarity between algorithm 1 and the LZ78 compression scheme (section 2.4). Our algorithm makes implicit use of the intuitive correlation between frequent patterns and Lempel-Ziv phrases, which capture repeating patterns. In section 4, we show how the top-k trie can be used to approximate LZ78.

Preventing Orphans by Recycling Leaves

The recycling of a node in algorithm 1 comes with a complication: u may be an inner node with up to $\mathcal{O}(k)$ children that become orphans as a result of the operation. If the represented pattern was to be inserted back due to becoming frequent again, to ensure integrity, the orphaned former children that still remain would have to be connected back. However, this is non-trivial, because the node number may now be other than u.

A pragmatic solution would be to delete the entire subtrie of u from T_F if u is to be recycled. This would be safe in the sense that we cannot lose any frequent patterns, because the represented string is a prefix of all strings represented in the subtrie, and thus their frequencies are at most $f'_u \leq \theta$. However, this would take time $\mathcal{O}(k)$.

Instead, we ensure that only the leaves of T_F can ever be recycled. To achieve this, only the leaves are maintained in the Space-Saving data structure to begin with. (Besides preventing orphans, this yields practical speedups because for realistic inputs, a vast majority of nodes in T_F are inner nodes. Avoiding the operations for incrementing the frequencies of inner nodes, albeit constant time, saves a considerable amount of work.) This requires us to consider two new operations in the Space-Saving data structure. First, whenever a leaf becomes an inner node because a child is added to it, it needs to be deleted from the data structure. This can trivially be done in constant time thanks to the use of doubly-linked lists. Second, when an inner node v becomes a leaf because its last child has been recycled, we need to insert it into the Space-Saving data structure. The insert frequency f'_v can be arbitrarily large, and finding the bucket into which to insert v would require up to $\mathcal{O}(k)$ steps in a doubly-linked list of buckets. We address this in the following.

Frequency Limitation and Renormalization for Fast Bucket Access

Instead of maintaining the buckets of the Space-Saving data structure in a doubly-linked list, we propose to maintain them in an array of fixed size $f'_{\max} = \Theta(k)$ where the head of the bucket for frequency f' is stored in the f'-th entry. Trivially, this requires space $\mathcal{O}(f'_{\max}) = \mathcal{O}(k)$. Along with this change, we introduce two constraints that ensure that estimated frequencies always fall in the range $[0, f'_{\max}]$ independent of the input stream:

1. We do not any increment a frequency $f'_v = f'_{\max}$ (for some node v) any further.
2. Whenever θ is incremented to $f'_{\max}/2$, we *renormalize* the data structure by setting $f'_v := f'_v - f'_{\max}/2$ for all nodes v of T_F and reset $\theta := 0$. (Note that θ is only incremented if there is no node u with $f'_u \leq \theta$. Thus, $f'_v > \theta$ holds before and after renormalization.)

The renormalization introduced by constraint 2 takes time $\mathcal{O}(k)$, but amortizes to constant time since at least $\Theta(f'_{\max}) = \Theta(k)$ incrementations are needed for another renormalization to be required. In actuality, let us recall how T_F behaves a lot like the LZ78 trie (see section 4). The only occasions at which θ may be incremented is when a leaf of T_F has been reached and algorithm 1 attempts to follow an edge that does not exist. Let z be the number of LZ78 phrases of S, then there are only ever $\mathcal{O}(z)$ occasions at which θ may be incremented. Thus, renormalizations are expected to occur rarely if S is repetitive and/or $f'_{\max}$ is sufficiently large. In our experiments on inputs of size 100 GiB (section 6), we never observed even one renormalization. This practical observation also justifies neglecting the bias introduced

by constraint 1, which causes very frequent patterns to be underestimated more than less frequent patterns. For this underestimation to cause some pattern u of frequency $f_u > f'_{\max}$ to fade out of T_F, at least one renormalization would have to occur and then, additionally, a sufficiently long substring of S with no occurrence of u would have to follow.

Error in Frequency Estimation

We note[2] that the estimated frequency of some frequent pattern u is actually the sum of the frequencies stored in the subtrie of the node corresponding to u in T_F. For example, let $S = \alpha^{10}$ for some $\alpha \in \Sigma$ (and $k \geq 4$): after processing S with algorithm 1, the frequency of stored at the node corresponding to α will be only $f'_\alpha = 4$. The sum of the frequencies stored in its subtrie, however, is indeed 10.

It has been shown [9] that the underestimation of item frequencies by the Misra-Gries algorithm is bounded by n/k. However, it is crucial to see that this is based on the assumption that every occurrence of an item on the stream is treated equally. That is not the case in our scenario: we do not account for all substrings of S, but only a subset that is the result of a parsing similar to LZ78. It seems straightforward that this contributes to additional error in the Misra-Gries algorithm if considering the frequency of every substring of S individually, but leave open a detailed analysis that we conjecture to be non-trivial. However, we can still look at our running time improvements and how they contribute to the error of the underlying Misra-Gries structure under the assumption that all substrings are counted.

First, we established that we only recycle leaves from T_F and never inner nodes. This does not introduce any additional error, because the Misra-Gries algorithm cares not *what* item is removed as long as is has zero frequency. (In fact, in Misra-Gries, *all* items of zero frequency would be removed immediately.)

The limitation of estimated frequencies to $f'_{\max}$, on the other hand, introduces arbitrary error. Consider $S = \alpha^n$ for some $\alpha \in \Sigma$: there are n occurrences of α, but the highest estimated frequency will be $\mathcal{O}(f'_{\max}) = \mathcal{O}(k)$, and clearly it is $\lim_{n\to\infty} n/k = \infty$. This corresponds to the underestimation bias against very frequent patterns that we discussed earlier. Even though the error is unbounded in the mentioned case, we found it to be negligible in practice for sufficiently large $f'_{\max}$.

The renormalization of frequencies does not contribute to any error directly. Recall how the threshold θ simulates a marker for items that have zero estimated frequency. If $\theta > 0$, it means that in the original Misra-Gries algorithm, the frequencies of all items would have been decremented θ times. To this end, the described renormalization can be considered a lazy application of these decrementations.

Summary

The top-k trie requires space $\mathcal{O}(k)$ and allows processing each character from S in constant amortized time with a very unlikely worst case (renormalization). Even though it is not of interest in the following where we compress S, the k patterns maintained by the algorithm could be enumerated via a traversal of T_F.

[2] We thank the anonymous reviewer to point out this fact.

4 Online Approximation of LZ78 in Parameterized Space

The similarity between the LZ78 and our top-k trie of section 3 should be very apparent at this point. We make it more explicit in a parameterized-space algorithm to approximate the LZ78 parsing of a stream S using the top-k trie. The algorithm can be seen as constrained-dictionary LZ78 similar to that of De Agostino [3], with the heuristic that only the k most frequently used phrases are maintained in the trie.

It works largely similar to the original LZ78 algorithm: let v be the current node in T_F (initially the root), and consider the next character α from S. If there is an outgoing edge labeled α from v, we follow that edge and set v to the number of the connected node. Otherwise, we output $f_v\alpha$ as the next phrase (e.g., by encoding the tuple (v, α)). Whenever we visit a node or create a new node, we update the Space-Saving data structure as described in section 3. This algorithm is an immediate application of the top-k trie and therefore, it is easy to see that it requires $\mathcal{O}(k)$ space and amortized constant time per input character.

To decode the produced parsing, the decoder can simulate T_F like the encoder. For this, the parameter k must be stored at the beginning of the compressed output.

Let z be the number of LZ78 phrases and z' be the number of phrases produced by our parsing. It holds that $z' \geq z$: for $k \geq z$, the trie T_F is able to hold a node for every LZ78 phrase and it is $z' = z$. For $k < z$, our approximation produces more phrases. Consider a unary string α^n for some $\alpha \in \Sigma$. In this case, $z = \Theta(\sqrt{n})$. For our approximation, because the size of T_F is limited to k, it is $z' = \Theta(n/k)$, and so the approximation ratio is $z'/z = \Theta(\sqrt{n}/k)$. We conjecture that this also holds for arbitrary strings.

5 Online Approximation of LZ77 in Parameterized Space

We propose a new approximation of LZ77 that combines the top-k trie of section 3 with a blockwise computation of LZ77. We partition S into blocks of size B such that the i-th block is $S_i := S[iB \mathinner{..} iB + B - 1]$. Let the top-$k$ trie T_F be initially empty. After reading block S_i from S, we first compute its LZ77 parsing (using the algorithm of appendix C) consisting of the z' phrases $f_1, \ldots, f_{z'}$. Then, we compute a parsing of S_i consisting of $\leq z'$ phrases as follows. Let m be the current position within S_i (initially $m := 1$). Let $c := \min\{j \mid |f_1 \cdots f_j| \geq m\}$ be the index of the LZ77 phrase that contains position m. We call $\ell := |f_1 \cdots f_c| - m$ the length of the *remainder* of f_c. Furthermore, let s_m be the longest prefix of $S_i[m \mathinner{..} B-1]$ that can be spelled using the top-k trie T_F starting at the root. Now, if $|s_m| > m - |f_1 \cdots f_c|$, then the next phrase in our parsing is $g := s_m$ and we advance to position $m := m + |s_m|$. Otherwise, we choose the phrase $g := S_i[m \mathinner{..} m + \ell - 1]$ (the remainder of f_c) and advance to position $m := m + \ell$. We then enter g into T_F in similar fashion as we parse the input in section 4 (LZ78): starting from the root, we navigate (and possibly insert) the edges of T_F using the characters from g as deep as possible and increment the frequencies of the corresponding prefixes.

To create an intuition, this algorithm attempts to get the best out of both worlds: it looks at the (remainder of the) next LZ77 phrase and the longest prefix that can be spelled out by T_F, and then greedily picks whatever allows us to advance further in S_i. This is visualized in figure 3. It is easy to see that the number of phrases produced this way is at most z', and may be less than z' if T_F contains useful strings. Doing this for all $\lceil n/B \rceil$ blocks results in an approximation of LZ77 with the total number of phrases being lower bounded by z (for $B \geq n$ or suitable S, we produce exactly z phrases).

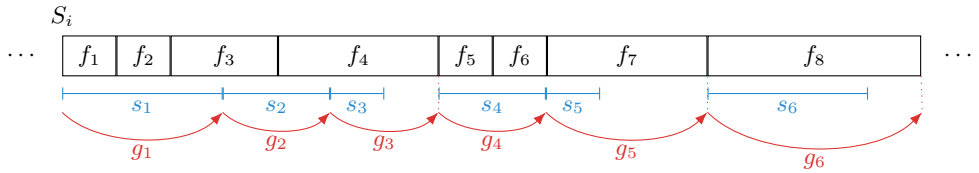

Figure 3 Visualization of the top-k LZ77 algorithm processing some block S_i. The LZ77 parsing consists of the phrases $f_1, \ldots, f_8$, the patterns $s_1, \ldots, s_6$ are contained in the top-k trie T_F. The arrows at the bottom represent the produced greedy parsing $g_1, \ldots, g_6$, taking the remainder of a LZ77 phrase or a string from T_F depending on which advances further towards the end of S_i.

The LZ77 parsing of S_i can be computed in time and space $\mathcal{O}(B)$. Processing S_i then takes $\mathcal{O}(B)$ amortized time: we navigate an edge in T_F at most twice for every character, first for finding s_m, and a second time for entering the new phrase g. Over all $\lceil n/B \rceil$ blocks, the total amortized time to process S becomes $\mathcal{O}(n)$, i.e., constant amortized time per input character. By choosing $B = \mathcal{O}(k)$, the space requirement becomes $\mathcal{O}(k)$.

Phrase Encoding

We encode an LZ77 phrase g as a tuple (ℓ, δ). The value $\ell = 0$ indicates that g is a *literal phrase*. Then, $\delta = g \in \Sigma$ is a character. Otherwise, g has a previous occurrence and it is a *referencing phrase*. Then, it is $\ell = |g| > 0$ and $\delta \in \mathbb{N}$ is the distance from said occurrence and a decoder can interpret the tuple as *copy ℓ characters starting δ characters ago*.

In our case, g may also be a string from the top-k trie T_F. To support this, we apply two changes to the semantics of the tuple (ℓ, δ). First, we forbid referencing phrases of length $\ell = 1$ and instead replace them by the corresponding literal phrases with $\ell = 0$ and $\delta = g$ their only character. Second, we reserve length $\ell = 1$ to indicate a *top-k phrase*, where δ is the number of the node in T_F that represents g. If k is stored at the beginning of the output, the parsing can be decoded by simulating T_F like the encoder.

6 Experiments

6.1 Implementation

We implemented the online approximations of LZ78 (`topk-lz78`, section 4) and LZ77 (`topk-lz77`, section 5) in C++20. The source code is publicly available as referred to on the front matter. In the experiment, we also consider `blockwise-lz77`, which computes the LZ77 parsing for every input block of size B and then encodes it, i.e., without using the top-k trie. The variant `blockwise-lz77+` simply uses a larger block size $B' > B$.

To be able to compare compression ratios, we implemented a blockwise encoding of the tuples representing LZ78 or LZ77 phrases, respectively, emitted by the compressors. For a parameter b, we buffer b tuples in memory and write them to file once the buffer overflows. In a preliminary experiment, we found that $b := 32 \cdot 2^{10}$ gives the best overall results and thus use this value throughout our experiments. We encode tuples as follows.

- In `topk-lz78`, tuples are of the form (j, α) where $j < k$ is the number of a node in the top-k trie and $\alpha \in \Sigma$ is the appended character. We encode j in binary using $\lceil \lg(k-1) \rceil$ bits and α using Huffman codes according to the distribution among the b buffered tuples.
- In `topk-lz77`, tuples are of the form (ℓ, δ) as described in section 5. We encode ℓ using Huffman codes according the distribution of all values ℓ among the b buffered tuples. We exploit the practical observation that most referencing phrases are short: if $\ell \geq 255$,

Table 1 Listing of compressors that we compare against. For each compressor, we list the URL of its main website as well as the command line flags we use to execute it for best compression.

Compressor	URL	Execution Flags
`gzip`	`http://gzip.org/`	`-9`
`xz`	`https://tukaani.org/xz/`	`-9`
`zstd`	`http://facebook.github.io/zstd/`	`-19`
`bzip2`	`https://sourceware.org/bzip2/`	`-9`
`bsc`	`http://libbsc.com/`	`-b2047`

Table 2 Selected statistics on the inputs used in the experiments: n is the file size, $\sigma = |\Sigma|$ the number of distinct characters (bytes), $\mathcal{H}_0 := \sum_{i=1}^{\sigma}(n_i/n)\log_2(n_i/n)$ the zeroth-order entropy, z_{78} the number of LZ78 phrases and z_{77} the number of LZ77 phrases.

Input	n	σ	$\mathcal{H}_0$	z_{78}	z_{77}
CommonCrawl	$100 \cdot 2^{30}$	243	6.20	7,149,629,111	2,596,068,363
DNA	$100 \cdot 2^{30}$	4	2.00	5,895,129,082	4,414,716,848
Wikipedia	$100 \cdot 2^{30}$	213	5.37	5,340,147,122	3,513,405,017

we encode the length 255 followed by the binary representation of $\ell - 255$ using $\lceil \lg B \rceil$ bits. If the phrase is a literal phrase ($\ell = 0$), $\delta \in \Sigma$ is a character that we encode using a Huffman code as for `topk-lz78`. If the phrase is a top-k phrase ($\ell = 1$), we encode the top-k trie node number $\delta < k$ in binary using $\lceil \lg(k-1) \rceil$ bits. Otherwise, if the phrase is referencing ($\ell > 1$), then we encode δ in binary using at most $\lceil \lg \delta_{\max} \rceil$ bits (where $\delta_{\max} < B$ is the largest δ encountered among the b tuples).

- In `blockwise-lz77`, tuples are also of the form (ℓ, δ), but there are only literal or referencing phrases and no top-k phrases.

6.2 Experimental Setup

The experiment is conducted on a machine with an Intel Xeon E5-2640v4 processor running at 2.4 GHz with a 25 MB cache and 64 GB of RAM. We measure the average compression *time* over three iterations as well as the compression *ratio* (output size divided by input size) for our compressors from section 6.1 and the compressors listed in table 1, which we execute with flags for lowest compression ratio. For a fair comparison, we disabled the parallel computation features of `bsc` by compiling it without OpenMP. Our code is compiled with GCC 13.2 and all optimizations enabled (`-O3`). We set k as high as possibly in our compressors to fill the available RAM. Preliminary experiments revealed that the threshold θ increases only very rarely for large k, confirming our assessments of section 3. This allows us to set $f'_{\max}$ very low with no need for any renormalization. Specifically,

- for `topk-lz78`, we set $k := 896 \cdot 2^{20}$ and $f'_{\max} := 2^{20}$,
- for `topk-lz77`, we set $B := 2^{31} - 1$, which is the largest block size that allows us to use 32-bit indices for computing the LZ77 parsing. We set $k := 640 \cdot 2^{20}$ and $f'_{\max} := 2^{20}$, and
- for `blockwise-lz77`, we set $B := 2^{31} - 1$ for a direct comparison with `topk-lz77`.
- for `blockwise-lz77+`, we set $B' := 3 \cdot 2^{30} \approx 1.5B$ to fill the available RAM (here, 64-bit indices are required to randomly access the block).

The input files for the compressors are CommonCrawl (web crawls from the Common Crawl Project, text only), DNA (raw human genomic sequences consisting of only `A`, `C`, `G` and `T`) and Wikipedia (XML dumps of the German and English versions of Wikipedia). Each file is of size 100 GiB. Table 2 shows additional statistics on the inputs. In appendix B, we give more detailed information as to how they have been generated.

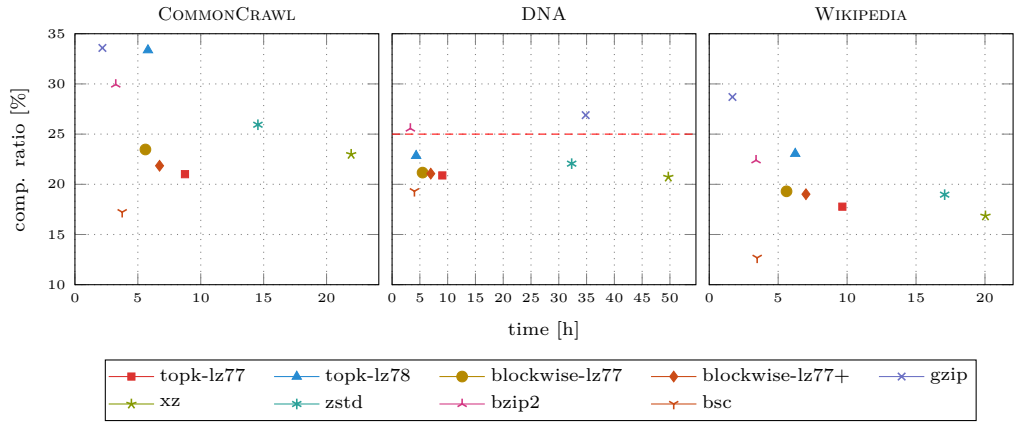

Figure 4 Running time versus compression ratio of the evaluated algorithms on inputs of size 100 GiB. For DNA, the dashed line marks the compression ratio of 25 % that can be trivially achieved by encoding the file using two bits per character as opposed to one byte (since the alphabet size is four).

6.3 Results

Our main results are shown in figure 4, where we plot the running times and achieved compression ratios of the stated compressors for the individual inputs.

Among the LZ-based compressors (`gzip`, `xz`, `zstd` and ours), our top-k LZ77 approximation `topk-lz77` achieves the best combination of compression time and ratio. Even though `xz` can achieve slightly lower compression ratios (under 1 % lower on DNA and Wikipedia), it takes a multiple of the time (about twice as long on CommonCrawl and Wikipedia, and nearly five times as long on DNA). The running times are also roughly independent of the input with a standard deviation of only approximately 37 minutes. This is in contrast to the other LZ-based compressors, which take substantially longer to compress DNA (for computing the next phrase, `gzip` generates candidate lists based on the first three characters and then processes these rather naïvely – if the alphabet is small and near-uniformly distributed, the lists become very long; `xz` and `zstd` presumably use similar heuristics).

As expected – due to the fact that LZ78 typically produces a larger number phrases than LZ77 – the compression ratios of our LZ78 approximation `topk-lz78`, albeit comparatively fast, are among the worst. However, we note that computing the exact LZ78 factorization of the inputs could not be computed within the available memory.

The block-sorting compressor `bsc` achieves the best compression ratios by far, which indicates that compression based on the Burrows-Wheeler Transform (BWT) works very well for natural languages. Since it involves mainly suffix sorting – a problem very well studied in practice – it achieves this also at relatively short running times. Even though `bzip2` is based on the same fundamentals, it uses much smaller block sizes and thus achieves only much higher compression ratios, albeit also very fast.

Impact of k

To analyze the impact of the parameter k on compression and resource usage, we run `topk-lz77` and `topk-lz78` for different k and present the results in figures 5 (appendix A).

The memory usage is approximately 60 B per trie node. The block text index used by `topk-lz77` requires 8 B per character (suffix array, its inverse, and the LCP array). Therefore, with the chosen $B := 2^{31} - 1$, `topk-lz77` requires almost 16 GiB of memory additional to

the top-k trie and independent of k. Despite the fact that our algorithms require constant (amortized) time per input character, the running times become longer for larger k. This can be explained by the higher number of cache misses that occur navigating the trie and updating the Space-Saving data structure: for the largest k, these are spread across the majority of the available RAM and it becomes far more likely that every navigation step (either in the top-k trie or the Space-Saving data structure) incurs a cache miss.

The parameter k affects the number of phrases emitted by `topk-lz78` significantly. As an example, CommonCrawl has 7,149,629,111 LZ78 phrases. For $k = 2^{20}$, `topk-lz78` emits 15,661,924,249 (more than twice as many), but only 7,910,221,374 phrases (10.6 % more than LZ78) for the largest $k = 896 \cdot 2^{20}$. The compression ratio improves accordingly. The difference between the number of emitted phrases and the number z_{78} of actual LZ78 phrases increases only sublinearly with decreasing k for all inputs. Empirically, this respects our conjecture of section 4 that `topk-lz78` is a $\Theta(\sqrt{n}/k)$ approximation of LZ78.

For `topk-lz77`, there is a much less visible effect. We refer to figure 6 for a closer look on the number of phrases. For $k = 2^{20}$, on average, `topk-lz77` emits only less than 0.5 % fewer phrases than `blockwise-lz77` and can hardly be considered an improvement. For the largest $k = 2^{29}$, it emits approximately 2.5 % phrases less than `blockwise-lz77` on average. Increasing k helps only marginally to approach the number z_{77} of actual LZ77 phrases.

Top-k LZ77 versus (blockwise) LZ77

To give a more detailed comparison, table 3 (in appendix A lists the number of phrases emitted by `blockwise-lz77` and `topk-lz77` (for $k = 640 \cdot 2^{20}$) in comparison to the number z_{77} of LZ77 phrases. The top-k trie contributes significantly to the output: on average, 55.6 % of the phrases are top-k phrases (70.9 % on CommonCrawl, 36.6 % on DNA and 59.2 % on Wikipedia). This indicates that the trie contains frequent patterns typically longer than referencing (block-local) LZ phrases starting at the same position.

As stated previously, each trie node occupies approximately 60 B of memory, whereas each additional character in the block only adds 8 B to the text index size. Even though these are not directly comparable, a fair question is why not simply increase the block size to fill the RAM instead of using the top-k trie? To address this, consider `blockwise-lz77+`, where we set the block size accordingly (1 GiB larger than `blockwise-lz77+`). An important difference is that now, with blocks $\geq 2^{31}$, each character contributes 16 B to the text index as we require 64 bits for block access. (We did not consider implementing, e.g., 40-bit integers). In table 3, we can see that indeed, `blockwise-lz77+` emits slightly less phrases than `topk-lz77` (4.0 % less on CommonCrawl, 1.9 % less on DNA and 0.9 % less on Wikipedia). However, from figure 4, we can take that `topk-lz77`, albeit somewhat slower, still yields better compression ratios, which implies that top-k phrases can be encoded more efficiently.

7 Conclusions and Outlook

Our top-k LZ77 approximation achieves competitive compression ratios among other Lempel-Ziv-based compressors. This is remarkable considering the fact that our chosen encoding (described in section 6.1) is far less sophisticated than that of the competitors and likely allows for future improvements. Furthermore, this result is achieved at competitive speeds. Our top-k LZ78 approximation is a viable alternative to LZ78 if memory is limited, yielding good compression (albeit not competitive) at relatively fast running times.

To improve running times, parallel variants of our compressors could be considered. The key issue would be how to use the top-k trie, which currently only supports strictly sequential operations. If p is the number of processors, working with p independent tries that are

synchronized regularly can be an option. Then, however, each trie can only be of size at most $\mathcal{O}(k/p)$ to stay within $\mathcal{O}(k)$ total space. Synchronizing (e.g., merging) the p tries is also non-trivial, as the node numbering may be different in each of the tries.

To improve compression, an idea that we did not pursue in this work would be to encode more frequent patterns with shorter codewords than less frequent patterns. For example, instead of encoding simply the index $v < k$ of a node in the trie using $\lceil \lg k \rceil$ bits, one could encode the tuple (f'_v, v'), where f'_v is the frequency of the pattern represented by v and v' is v's index within the bucket corresponding to frequency f'_v. Because larger f'_v will naturally be encoded more frequently, they could be encoded using Huffman codes, much like the phrase lengths. The issue that arises with this particular proposal would be how to locate and access v' efficiently in dynamic buckets (currently, they are doubly-linked lists).

The top-k trie as presented tends to contain redundant information. This comes from the fact that every proper substring of a frequent pattern is also frequent. Particularly, consider a proper suffix v of a frequent pattern u that may also be frequent independently of u (not all occurrences of u imply an occurrence of v). Every pattern of which v is a proper prefix is contained in an additional branch off the trie's root, potentially rephrasing many substrings also contained in the path to u. To make most of the working space, an important open goal is therefore to compress the trie or use a different representation altogether. Ideally, based on the idea of string attractors [23], it could be of interest to maintain dynamically the smallest string that is a *k-attractor* for the input, where k is a parameter to control the space usage.

A more general issue with using heavy hitters for compression is that they tend to consist of shorter patterns, as these are naturally more frequent. Going away from a strict streaming model, future work may consider a precompression pass, where a different algorithm, aimed specifically at finding long repetitions, preprocesses the input.

It is straightforward to think about random access to the compressed string. In the presented encoding, a node index v always refers to the trie T_F in the instant in which it is was encoded. Accessing a particular character $S[i]$ would therefore require to decode the entire prefix $S[1..i]$. It may be interesting to consider a two-pass variant of top-k LZ78 similar to the work of Arz and Fischer [7]: in the first pass, we compute the top-k trie T_F for all of S. In the second pass, we then compute a greedy parsing of S using the fixed trie T_F. By keeping T_F in RAM and precomputing a suitable rank/select data structure for the phrases, it is then possible to access any $S[i]$ in time at most $\mathcal{O}(h)$, where h is the height of T_F.

Looking past compression, the top-k trie may have applications as a framework for online frequent pattern estimation where frequent patterns are of interest, e.g., for mining key phrases in a document. In this work, we did not consider output of frequent patterns, but this can easily be done by traversing the trie after processing the input. Depending on the application, the error introduced by our approach as discussed in section 3 may be relevant.

References

1 Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. *ACM Trans. Database Syst.*, 38(4):26, 2013. doi:10.1145/2500128.

2 Charu C. Aggarwal and Philip S. Yu. A survey of synopsis construction in data streams. In *Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*, pages 169–207. Springer, 2007. doi:10.1007/978-0-387-47534-9_9.

3 Sergio De Agostino. Bounded size dictionary compression: Relaxing the lru deletion heuristic. *Int. J. Found. Comput. Sci.*, 17(6):1273–1280, 2006. doi:10.1142/S0129054106004406.

4 Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999. doi:10.1006/jcss.1997.1545.

5 Alberto Apostolico, Matteo Comin, and Laxmi Parida. Bridging lossy and lossless compression by motif pattern discovery. In *General Theory of Information Transfer and Combinatorics*, volume 4123 of *Lecture Notes in Computer Science*, pages 793–813. Springer, 2006. doi:10.1007/11889342_51.

6 Diego Arroyuelo, Rodrigo Cánovas, Johannes Fischer, Dominik Köppl, Marvin Löbel, Gonzalo Navarro, and Rajeev Raman. Engineering practical Lempel-Ziv tries. *ACM J. Exp. Algorithmics*, 26:14:1–14:47, 2021. doi:10.1145/3481638.

7 Julian Arz and Johannes Fischer. Lempel-Ziv-78 compressed string dictionaries. *Algorithmica*, 80(7):2012–2047, 2018. doi:10.1007/S00453-017-0348-7.

8 Sairam Behera, Sutanu Gayen, Jitender S. Deogun, and N. V. Vinodchandran. KmerEstimate: A streaming algorithm for estimating k-mer counts with optimal space usage. In *ACM International Conference on Bioinformatics, Computational Biology and Health Informatics (BCB)*, pages 438–447. ACM, 2018. doi:10.1145/3233547.3233587.

9 Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Bounds for frequency estimation of packet streams. In *10th Internaltional Colloquium on Structural Information Complexity (SIROCCO)*, volume 17 of *Proceedings in Informatics*, pages 33–42. Carleton Scientific, 2003.

10 Graham Cormode and Marios Hadjieleftheriou. Finding the frequent items in streams of data. *Commun. ACM*, 52(10):97–105, 2009. doi:10.1145/1562764.1562789.

11 Peter Deutsch. DEFLATE compressed data format specification version 1.3. *RFC*, 1951:1–17, 1996. doi:10.17487/RFC1951.

12 Jonas Ellert. Sublinear time Lempel-Ziv (LZ77) factorization. In *30th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 14240 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2023. doi:10.1007/978-3-031-43980-3_14.

13 Johannes Fischer, Travis Gagie, Pawel Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *23rd European Symposium on Algorithms (ESA)*, volume 9294, pages 533–544. Springer, 2015. doi:10.1007/978-3-662-48350-3_45.

14 Johannes Fischer, Volker Heun, and Stefan Kramer. Optimal string mining under frequency constraints. In *10th European Conference on Principles and Practice of Knowledge Discovery in Databases PKDD*, volume 4213 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2006. doi:10.1007/11871637_17.

15 Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiko Sadakane. Lempel-ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018. doi:10.1007/S00453-017-0333-1.

16 E. Fredkin. Trie memory. *Commun. ACM*, 3:490–499, 1960. doi:10.1145/367390.367400.

17 Wensheng Gan, Jerry Chun-Wei Lin, Philippe Fournier-Viger, Han-Chieh Chao, and Philip S. Yu. A survey of parallel sequential pattern mining. *ACM Trans. Knowl. Discov. Data*, 13(3):25:1–25:34, 2019. doi:10.1145/3314107.

18 Torben Hagerup. Sorting and searching on the word RAM. In *15th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer, 1998. doi:10.1007/BFb0028575.

19 Aaron Hong, Massimiliano Rossi, and Christina Boucher. LZ77 via prefix-free parsing. In *25th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 123–134. SIAM, 2023. doi:10.1137/1.9781611977561.CH11.

20 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Linear time lempel-ziv factorization: Simple, fast, small. In *24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 7922 of *Lecture Notes in Computer Science*, pages 189–200. Springer, 2013. doi:10.1007/978-3-642-38905-4_19.

21 Juha Kärkkäinen, Dominik Kempa, and Simon J. Puglisi. Lempel-Ziv parsing in external memory. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 153–162. IEEE, 2014. doi:10.1109/DCC.2014.78.

22 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

23 Dominik Kempa and Nicola Prezza. At the roots of dictionary compression: string attractors. In *50th Annual ACM Symposium on Theory of Computing (STOC)*, pages 827–840. ACM, 2018. doi:10.1145/3188745.3188814.

24 Dmitry Kosolobov, Daniel Valenzuela, Gonzalo Navarro, and Simon J. Puglisi. Lempel-Ziv-like parsing in small space. *Algorithmica*, 82(11):3195–3215, 2020. doi:10.1007/S00453-020-00722-6.

25 Shirou Maruyama and Yasuo Tabei. Fully online grammar compression in constant space. In *Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014*, pages 173–182. IEEE, 2014. doi:10.1109/DCC.2014.69.

26 Páll Melsted and Bjarni V. Halldórsson. Kmerstream: streaming algorithms for k-mer abundance estimation. *Bioinform.*, 30(24):3541–3547, 2014. doi:10.1093/bioinformatics/btu713.

27 Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *10th International Conference on Database Theory (ICDT)*, volume 3363, pages 398–412. Springer, 2005. doi:10.1007/978-3-540-30570-5_27.

28 Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program*, 2(2):143–152, 1982. doi:10.1016/0167-6423(82)90012-0.

29 Hamid Mohamadi, Hamza Khan, and Inanç Birol. ntcard: a streaming algorithm for cardinality estimation in genomics data. *Bioinform.*, 33(9):1324–1330, 2017. doi:10.1093/bioinformatics/btw832.

30 S. Muthukrishnan. Data streams: algorithms and applications. In *14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 413–413. SIAM, 2003. URL: http://dl.acm.org/citation.cfm?id=644108.644174.

31 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinform.*, 34(4):568–575, 2018. doi:10.1093/bioinformatics/btx636.

32 Alberto Policriti and Nicola Prezza. LZ77 computation based on the run-length encoded BWT. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.

33 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst.*, 41(4):589–607, 2007. doi:10.1007/S00224-006-1198-X.

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

35 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.

A Additional Data

This appendix contains figures showing additional data from the experiments of section 6, namely:

- Figures 5 and 6 (impact of k on the number of phrases, compression ratio, running time and required memory of `topk-lz78` and `topk-lz77`
- Table 3 (comparison of the number of phrases between exact, blockwise and top-k LZ77)

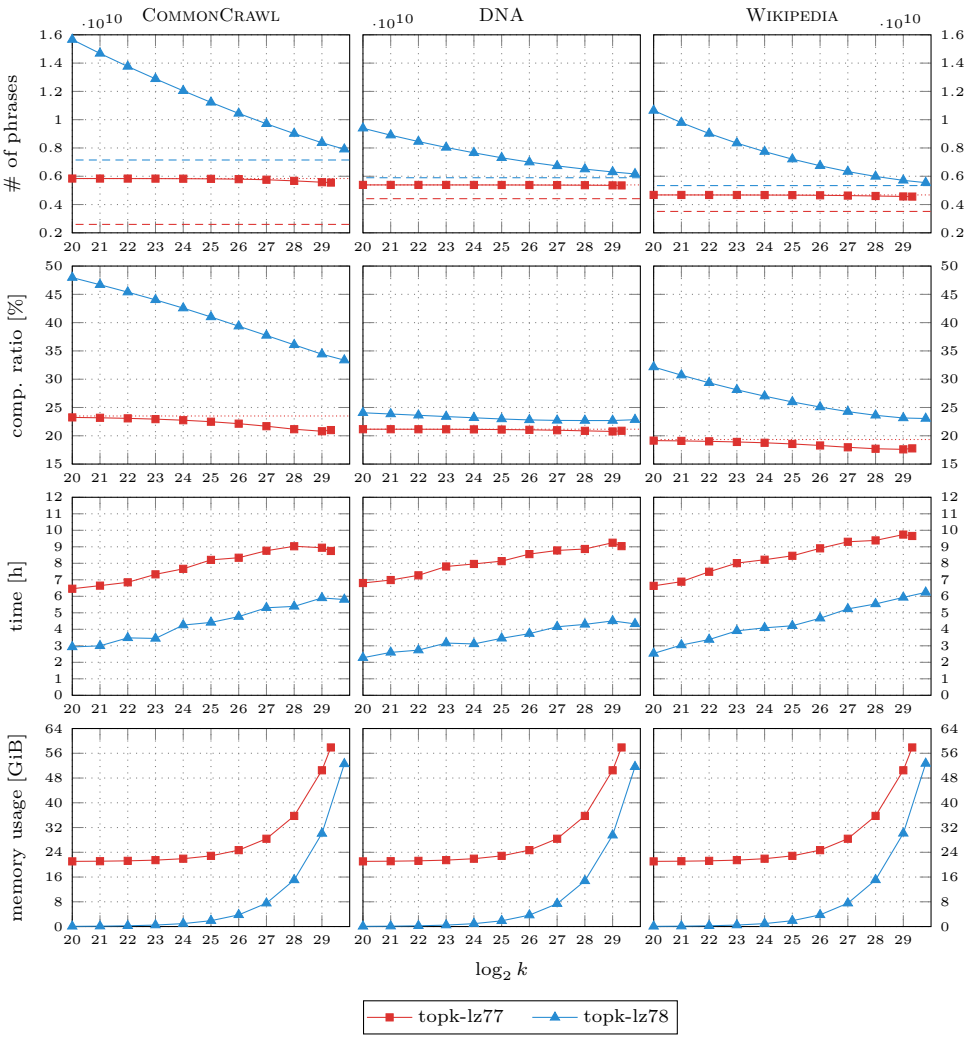

Figure 5 Impact of k on the number of emitted phrases, compression ratio, running time (average of three iterations) and memory usage of our algorithms. Regarding the number of phrases, the dotted line marks the number of phrases emitted by `blockwise-lz77` (upper bound) and the dashed lines mark the exact number of LZ77 or LZ78 phrases, respectively (lower bounds). Please refer to figure 6 for a closer look at the number of phrases emitted by `topk-lz77`.

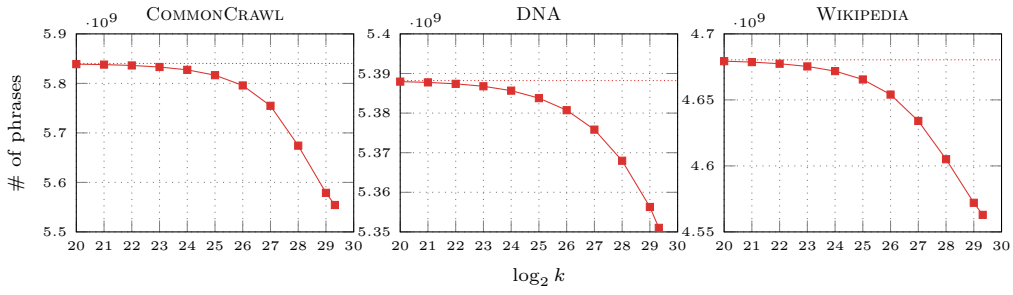

Figure 6 Impact of k on the number of phrases emitted by `topk-lz77`. This shows the same data as figure 5, but on a different scale for each input. The dotted lines mark the number of phrases emitted by `blockwise-lz77` (upper bound).

Table 3 Phrases emitted by `blockwise-lz77`, `topk-lz77` ($k = 640 \cdot 2^{20}$) and `blockwise-lz77+` compared to the number z_{77} of LZ77 phrases. Values are given as billions. For `topk-lz77`, we distinguish between the number of (block-local) LZ phrases as well as phrases from the top-k trie.

Input	`blockwise-lz77`	`topk-lz77` Total	LZ phrases	top-k phrases	`blockwise-lz77+`	z_{77}
CommonCrawl	5.840	5.554	1.502	4.052	5.333	2.596
DNA	5.388	5.351	3.178	2.173	5.251	4.415
Wikipedia	4.680	4.563	1.797	2.765	4.520	3.513

B Input Corpus

For the experiments in section 6, we use the 100 GiB prefixes of the following corpus.

- CommonCrawl: Web crawls from the Common Crawl Project (`https://commoncrawl.org`). We use the WET files 00000 to 00600 from the `CC-MAIN-2019-09` crawl and concatenate them in numerical order, removing all WARC meta information (one line consisting of `WARC/1.0` and the following eight lines).
- DNA: Human genomic sequences from the European Nucleotide Archive (`https://www.ebi.ac.uk`). We extract the raw sequences from the FASTQ datasets `DRR00000` to `DRR000426` (non-continuous) available at `ftp://ftp.sra.ebi.ac.uk/vol1/fastq/DRR000`, removing all characters other than `A`, `C`, `G` and `T`. These are concatenated in numerical order (`DRR<N>`, where `<N>` is the file number).
- Wikipedia: XML dumps of the German and English versions of Wikipedia (`dumps.wikimedia.org`). We concatenate the dumps as of March 20, 2019 in the order `de`, `en`. The contained articles are of that date with no version history.

C Computing LZ77 in Linear Time and Space

To compute a LZ77 parsing of a string $S \in \Sigma^n$ of length n – as required by our blockwise LZ77 implementations (with and without the top-k trie) – we implement a well-known approach that simulates the *longest previous factor* (LPF) array as shown in algorithm 2.

We first compute the suffix array A of S and its inverse A^{-1}, which can be done in time and space $\mathcal{O}(n)$ (see, e.g., [22]). With this, we parse S from left to right computing the next LZ77 phrase in each step.

Let $i < n$ be the current position in S. The next LZ77 phrase is the longest prefix of $S[i..n]$ that has an occurrence in S beginning at a position $j < i$. We can find j in i's *neighbourhood* in the suffix array. Let $i' = A^{-1}[i]$ be the position of i in the suffix array. Because of the lexicographic order of suffixes in A, the values nearest to i' in A that are $< i$ are candidates for j. We call them the *previous smaller value* (PSV, lexicographically smaller than $S[i..n]$) or the *next smaller value* j_2 (NSV, lexicographically larger than $S[i..n]$) and denote them by j_1 or j_2, respectively. By computing the *longest common extension* (LCE) between $S[i..n]$ and $S[j_1..n]$ or $S[j_2..n]$, respectively, we find the lengths ℓ_1 and ℓ_2 of the candidate LZ77 phrases. Because LZ77 phrases are defined greedily, whichever candidate is longer is the next LZ77 phrase. Ties are broken arbitrarily. The special case where $\ell_1 = \ell_2 = 0$ occurs if $S[i]$ is the first occurrence of the corresponding character in S, then the next LZ77 phrase is a literal phrase. The algorithm requires time and space $\mathcal{O}(n)$ (the LCE computations can be done using the LCP array, which can be computed in $\mathcal{O}(n)$ extra time and space, or naïvely comparing character by character, contributing to at most two additional scans of S in total). For a more detailed elaboration of this algorithm, we refer the reader to [20].

Algorithm 2 Computing a LZ77 parsing for an input S by simulating the LPF array. For clarity, we omitted the handling of some border cases in the LCE computations. The operator $\circ$ concatenates two strings.

Input : String $S \in \Sigma^n$ of length n
Output : LZ77 parsing S' of S

1 $S' \leftarrow \varepsilon$
2 $A \leftarrow$ suffix array of S, $A^{-1} \leftarrow$ inverse suffix array of S
3 $i \leftarrow 1$
4 **while** $i < n$ **do**
5 $\quad i' \leftarrow A^{-1}[i]$ `// suffix array neighbourhood of` i `in` A
6 $\quad j_1 \leftarrow A[\max\{j' \mid j' < i' \ \wedge \ A[j'] < i\}]$ `// PSV in` A `w.r.t.` i'
7 $\quad \ell_1 \leftarrow \max\{\ell \mid S[i \mathinner{..} i+\ell] = S[j_1 \mathinner{..} j_1+\ell] \ \wedge \ S[i+\ell+1] \neq S[j_1+\ell+1]\}$ `// LCE`
8 $\quad j_2 \leftarrow A[\min\{j' \mid j' > i' \ \wedge \ A[j'] < i\}]$ `// NSV in` A `w.r.t.` i'
9 $\quad \ell_2 \leftarrow \max\{\ell \mid S[i \mathinner{..} i+\ell] = S[j_2 \mathinner{..} j_2+\ell] \ \wedge \ S[i+\ell+1] \neq S[j_2+\ell+1]\}$ `// LCE`
10 $\quad \ell \leftarrow \max\{\ell_1, \ell_2, 1\}$
11 $\quad$ **if** $\ell > 1$ **then**
12 $\quad\quad \delta \leftarrow \begin{cases} i - j_1 & \text{if } \ell_1 > \ell_2 \\ i - j_2 & \text{otherwise} \end{cases}$
13 $\quad\quad S' \leftarrow S' \circ (\ell, \delta)$ `// referencing phrase`
14 $\quad$ **else**
15 $\quad\quad S' \leftarrow S' \circ (0, S[i])$ `// literal phrase`
16 $\quad i \leftarrow i + \ell$

In our implementation (algorithm 2), we implicitly convert referencing phrases of length one to literal phrases (line 10), which is more beneficial towards the encoding. We do not use the LCP array but compute LCEs naïvely by scanning. This requires significantly less working memory and is, in practice, less time-consuming than computing the LCP array and accessing it randomly. We use `libsais` for computing the suffix array and inverse it trivially in n steps ($A^{-1}[A[i]] := i$ for all i). Thus, if $n < 2^{31}$, we can use 32-bit integers and the practical space requirement, including the input S, becomes $9n$ bytes (`libsais` requires the 32nd bit to temporarily store auxiliary information). For larger n, we use 64-bit integers, such that the requirement becomes $17n$ bytes. (It would be possible to implement custom integer types such as 40-bit integers, but we did not consider that for this work.)

C.1 Fast Semi-External Memory Implementation

For computing the number $z = |S'|$ of LZ77 phrases of the 100 GiB inputs of section 6, we faced a practical problem: the machine with the largest RAM available to us has 1 TB, which is slightly less than $10n$ bytes. The algorithm described above, however, requires $17n$ bytes (since $n \geq 2^{31}$, we require 64-bit integers).

We considered using succinct representations of the suffix array (e.g., [33] as provided by the `SDSL`), however, the working space required for their construction exceed our limitations. We furthermore considered using the external-memory algorithms by Kärkkäinen et al. [21], however, these would have required several weeks of running time per input file.

To tackle this, we implemented a semi-external memory algorithm that requires $9n$ bytes of RAM for 64-bit integers and $8n$ bytes of external memory ($16n$ during construction) and is included in our public source code (`lpfs`). It is an adaptation of algorithm 2 that works as follows:

1. Load S and compute A as usual using $9n$ bytes of RAM.
2. Write A to file `sa`.
3. Compute A^{-1} in the space allocated for A by streaming the file `sa`. Let x be the i-th value read from the file, then we set $A^{-1}[x] := i$.
4. Write A^{-1} to file `isa`.
5. Load file `sa` back into A and delete file `sa`.
6. To parse S, stream the values $A^{-1}[i]$ from file `isa` and skip entries as needed.
7. When done parsing S, delete file `isa`.

The external memory portions of the algorithm are optimal in the external memory model in the sense that they require only $\mathcal{O}(n/B)$ I/O operations.

Since $n < 2^{40}$ in our case, we encoded the entries of files `sa` and `isa` using 40 bits each. Thus, the external memory requirement dropped to $5n$ bytes ($10n$ bytes during construction). We used this algorithm to successfully compute the LZ77 parsings for the stated inputs within approximately 24 hours each.