On Two LZ78-style Grammars: Compression Bounds and Compressed-Space Computation

Golnaz Badkobeh*[1], Travis Gagie[2], Shunsuke Inenaga[3], Tomasz Kociumaka†[4], Dmitry Kosolobov[5], and Simon J. Puglisi‡[5]

[1]Department of Computer Science, University of Warwick, Coventry, England
g.badkobeh@warwick.ac.uk
[2]EIT, Diego Portales University and CeBiB, Santiago, Chile
travis.gagie@mail.udp.cl
[3]Department of Informatics, Kyushu University, Fukuoka, Japan
inenaga@inf.kyushu-u.ac.jp
[4]Institute of Informatics, University of Warsaw, Warsaw, Poland
kociumaka@mimuw.edu.pl
[5]Department of Computer Science, University of Helsinki, Helsinki, Finland
dkosolobov@mail.ru puglisi@cs.helsinki.fi

Abstract

We investigate two closely related LZ78-based compression schemes: LZMW (an old scheme by Miller and Wegman) and LZD (a recent variant by Goto et al.). Both LZD and LZMW naturally produce a grammar for a string of length n; we show that the size of this grammar can be larger than the size of the smallest grammar by a factor $\Omega(n^{\frac{1}{3}})$ but is always within a factor $O((\frac{n}{\log n})^{\frac{2}{3}})$. In addition, we show that the standard algorithms using $\Theta(z)$ working space to construct the LZD and LZMW parsings, where z is the size of the parsing, work in $\Omega(n^{\frac{5}{4}})$ time in the worst case. We then describe a new Las Vegas LZD/LZMW parsing algorithm that uses $O(z \log n)$ space and $O(n + z \log^2 n)$ time with high probability.

Keywords: LZMW, LZD, LZ78, compression, smallest grammar

1 Introduction

The LZ78 parsing [19] is a classic dictionary compression technique, discovered by Lempel and Ziv in 1978, that gained wide use during the 1990s in, for example, the Unix `compress` tool and the GIF image format. Not written about until much later was that LZ78 actually produces a representation of the input string as a context-free grammar. In recent years, grammar compressors have garnered immense interest, particularly in the context of compressed text indexing: it is now possible to efficiently execute many operations directly on grammar-compressed strings, without resorting to full decompression (e.g., see [3, 4, 6, 7, 10, 16]).

A wide variety of grammar compressors are now known, many of them analyzed by Charikar et al. [5] in their study of the smallest grammar problem, which is to compute the smallest context-free grammar that generates the input string (and only this string). Charikar et al. show that this problem is NP-hard, and further provide lower bounds on approximation ratios for many grammar compressors. LZ78 is shown to approximate the smallest grammar particularly poorly, and can be larger than the smallest grammar by a factor $\Omega(n^{\frac{2}{3}}/\log n)$ (in [9] this bound was improved to $\Omega((\frac{n}{\log n})^{\frac{2}{3}})$), where n is the input length.

Our focus in this paper is on the LZD [8] and LZMW [14] grammar compression algorithms, two variants of LZ78 that usually outperform LZ78 in practice. Despite their accepted empirical advantage over LZ78, no formal analysis of the compression performance of LZD and LZMW in terms of the size of the smallest grammar exists. This paper addresses that need. Moreover, we show that the standard

*Supported by the Leverhulme Trust on the Leverhulme Early Career Scheme.
†Supported by Polish budget funds for science in 2013–2017 under the 'Diamond Grant' program.
‡Supported by the Academy of Finland via grant 294143.

algorithms for computing LZD and LZMW have undesirable worst case performance, and provide an alternative algorithm that runs in log-linear randomized time. In particular the contributions of this article are as follows:

1. We show that the size of the grammar produced by LZD and LZMW can be larger than the size of the smallest grammar by a factor $\Omega(n^{\frac{1}{3}})$ but is always within a factor $O((\frac{n}{\log n})^{\frac{2}{3}})$. To our knowledge these are the first non-trivial bounds on compression performance known for these algorithms.

2. Space usage during compression is often a concern. For both LZD and LZMW, parsing algorithms are known that use $O(z)$ space, where z is the size of the final parsing. We describe strings for which these algorithms require $\Omega(n^{\frac{5}{4}})$ time. (The only previous analysis is an $O(n^2/\log n)$ upper bound [8].)

3. We describe a Monte-Carlo parsing algorithm for LZD/LZMW that uses a z-fast trie [2] and an AVL-grammar [15] to achieve $O(z \log n)$ space and $O(n + z \log^2 n)$ time for inputs over the integer alphabet $\{0, 1, \dots, n^{O(1)}\}$. This algorithm works in the streaming model and computes the parsing with high probability. Using the Monte-Carlo solution, we obtain a Las Vegas algorithm that, with high probability, works in the same space and time.

In what follows we provide formal definitions and examples of LZD and LZMW parsings. Section 2 then establishes bounds for the approximation ratios for the sizes of the LZD/LZMW grammars. In Section 3 we consider the time efficiency of current space-efficient parsing schemes for LZD/LZMW. Section 4 provides an algorithm with significantly better (albeit randomized) performance. Conclusions and reflections are offered in Section 5.

1.1 Preliminaries.

We consider strings drawn from an alphabet Σ of size $\sigma = |\Sigma|$. The *empty string* is denoted by ϵ. The ith letter of a string s is denoted by $s[i]$ for i such that $1 \le i \le |s|$, and the substring of s that begins at position i and ends at position j is denoted by $s[i..j]$ for $1 \le i \le j \le |s|$. Let $s[i..j] = \epsilon$ if $j < i$. For any i, j, the set $\{k \in \mathbb{Z} \colon i \le k \le j\}$ (possibly empty) is denoted by $[i..j]$.

For convenience, we assume that the last letter of the input string s is \$, where \$ is a special delimiter letter that does not occur elsewhere in the string.

Definition 1.1. The *LZD (LZ–Double) parsing* [8] of a string s of length n is the parsing $s = p_1 p_2 \cdots p_z$ such that, for $i \in [1..z]$, $p_i = p_{i_1} p_{i_2}$ where p_{i_1} is the longest prefix of $s[k..n]$ and p_{i_2} is the longest prefix of $s[k + |p_{i_1}|..n]$ with $p_{i_1}, p_{i_2} \in \{p_1, \dots, p_{i-1}\} \cup \Sigma$ where $k = |p_1 \cdots p_{i-1}| + 1$. We refer to the set $\Sigma \cup \bigcup_{i \in [1..z]} \{p_i\}$ as the *dictionary of LZD.*

Definition 1.2. The *LZMW (LZ–Miller–Wegman) parsing* [14] of a string s of length n is the parsing $s = p_1 p_2 \cdots p_z$ such that, for $i \in [1..z]$, p_i is the longest prefix of $s[k..n]$ with $p_i \in \{p_j p_{j+1} \colon 1 \le j \le i-2\} \cup \Sigma$ where $k = |p_1 \cdots p_{i-1}| + 1$. We refer to the set $\bigcup_{i \in [2..z]} \{p_{i-1} p_i\}$ as the *dictionary of LZMW.*

Example 1.3. The LZD parsing of the string $s = abbaababaaba\$$ is $p_1 = ab$, $p_2 = ba$, $p_3 = abab$, $p_4 = aab$, and $p_5 = a\$$. This can be represented by $(a, b), (b, a), (1, 1), (a, 1), (a, \$)$. The LZMW parsing of s is the following: $p_1 = a$, $p_2 = b$, $p_3 = b$, $p_4 = a$, $p_5 = ab$, $p_6 = ab$, $p_7 = aab$, $p_8 = a$, and $p_9 = \$$. This can be represented by $(a, b, b, a, 1, 1, 4, a, \$)$.

Notice that the LZD/LZMW parsing of string s can be seen as a grammar that only generates s, with production rules of form $p_i \to p_j p_k$ $(j < i, k < i)$ or $p_i \to a$ $(\in \Sigma)$ for each phrase p_i, and the start rule $S \to p_1 p_2 \cdots p_z$. The *size* of a grammar is the total number of symbols in the right-hand side of the production rules. Thus, the size of the LZD (resp., LZMW) grammar is only by a constant factor larger than the number of phrases in the LZD (resp., LZMW) parsing.

2 Approximating the Smallest Grammar

The following theorem shows that, although LZD and LZMW have good compression performance in practice on high-entropy strings, their performance on low-entropy strings can be very poor.

Theorem 2.1. *For arbitrarily large n, there are strings s of length n for which the size of the grammars produced by the LZD and LZMW parsings is larger than the size of the smallest grammar generating s by a factor $\Omega(n^{\frac{1}{3}})$.*

Proof. Our proof is inspired by [5, Section VI, C]. Let $k \geq 4$ be an integer that is a power of 2. We will construct a string s of length $n = \Theta(k^3)$ that can be encoded by a grammar of size $O(k) = O(n^{\frac{1}{3}})$, but for which the LZMW parsing produces a grammar of size $\Omega(k^2) = \Omega(n^{\frac{2}{3}})$. The input alphabet is $\{a, b, c, d\}$; the letters c and d serve as separators. Denote $\delta_i = a^i bba^{k-i}$ and $\gamma_i = ba^i\, a\, a^i b\, c\, ba\, ba^2\, ba^3 \cdots ba^i$. The string s is as follows:

$$\begin{aligned} x &= \delta_k\delta_{k-1}\,\delta_k\delta_{k-2}\,\delta_k\delta_{k-3}\cdots\delta_k\delta_{k/2+1}\,\delta_k a^{k-1}, \\ s &= \gamma_0\gamma_1\cdots\gamma_{k-1}\delta_0 d\delta_1 d\cdots\delta_k d\, caa\, caa^2a^2\cdots ca^{2^i-1}a^{2^i}a^{2^i}\cdots ca^{\frac{k}{2}-1}a^{\frac{k}{2}}a^{\frac{k}{2}}dc\,x^{\frac{k}{2}}\ . \end{aligned}$$

We have $|s| = \Theta(k^3)$. Consider the prefix $\gamma_0\gamma_1\cdots\gamma_{k-1}\ \delta_0 d\delta_1 d\cdots d\delta_k d$, which will ensure the strings δ_i are in the LZMW dictionary.

We will show by induction on i that each substring γ_i of the prefix $\gamma_0\gamma_1\cdots\gamma_{k-1}$ is composed of the phrases ba^i, a, a^ib, $cbaba^2\cdots ba^i$ in the parsing of the string s. It is trivial for $i = 0$. Suppose that $i > 0$ and the assertion holds for all $\gamma_{i'}$ and $i' < i$. It follows from the inductive hypothesis that ba^i is the longest prefix of γ_i that is equal to a concatenation of two adjacent phrases introduced before the starting position of γ_i. Hence, by the definition of LZMW, the string γ_i starts with the phrase ba^i. In the same way we deduce that the phrase ba^i is followed by the phrases a, a^ib, and $cbaba^2\cdots ba^i$.

By a similar inductive argument, one can show that each substring $\delta_i d$ of the substring $\delta_0 d\delta_1 d\cdots\delta_k dc$ is composed of the phrases a^ib, ba^{k-i}, d. Since the phrases a^ib and ba^{k-i} are adjacent, the LZMW dictionary now contains the strings $\delta_i = a^ibba^{k-i}$ for all $i = 0, 1, \ldots, k$.

Similarly, the substring $caacaa^2a^2\cdots ca^{2^i-1}a^{2^i}a^{2^i}\cdots ca^{\frac{k}{2}-1}a^{\frac{k}{2}}a^{\frac{k}{2}}dc$ is parsed as $c, a, a, ca, a^2, a^2, \ldots, ca^{2^i-1}, a^{2^i}, a^{2^i}, \ldots, ca^{\frac{k}{2}-1}, a^{\frac{k}{2}}, a^{\frac{k}{2}}, dc$. In what follows we need only the string a^k introduced to the dictionary by the pair of phrases $a^{\frac{k}{2}}$.

Finally, consider the substring $x^{\frac{k}{2}}$. Observe that the first occurrence of x is parsed in (almost) the way it is written, i.e., it is parsed as $\delta_k, \delta_{k-1}, \delta_k, \delta_{k-2}, \ldots, \delta_k, \delta_{k/2+1}, \delta_k$. But the last phrase is a^k instead of a^{k-1}. In other words, the parsing of the second occurrence of x starts from the second position of x and, therefore, the first phrases of this parsing are as follows:

$$\delta_{k-1}, \delta_{k-2}, \delta_{k-1}, \delta_{k-3}, \ldots, \delta_{k-1}, \delta_{k/2}, \delta_{k-1}.$$

Again, the last phrase is a^k and, hence, the parsing of the third occurrence of x starts with the third position of x, and so on.

The LZMW parsing of s, therefore, consists of $\Omega(k^2)$ phrases and the size of the LZMW grammar is $\Omega(k^2)$. But there is a grammar of size $O(k)$ producing s:

$$\begin{aligned} &S \to \Gamma_0\Gamma_1\cdots\Gamma_{k-1}\Delta_0 d\Delta_1 d\cdots\Delta_k dcA_2cA_5cA_{11}\cdots cA_{k/2+k-1}dcX^{k/2}, \\ &A_0 \to \epsilon, \quad B_0 \to c, \quad A_i \to A_{i-1}a, \quad B_i \to B_{i-1}bA_i \quad \text{for } i \in [1..2k], \\ &\Gamma_i \to bA_{2i+1}bB_i, \quad \Delta_i \to A_ibbA_{k-i} \quad \text{for } i \in [0..k], \\ &X \to \Delta_k\Delta_{k-1}\,\Delta_k\Delta_{k-2}\cdots\Delta_k\Delta_{k/2+1}\,\Delta_k A_{k-1}\ . \end{aligned}$$

Using similar ideas we can describe a troublesome string for the LZD scheme:

$$s = (a^2\,c^2\,a^3\,c^3\cdots a^kc^k)(bb\,abb\,a^2bb\,a^3\cdots bba^{k-1}bb)(\delta_0 d^2\delta_1 d^3\cdots\delta_k d^{k+2})x^{\frac{k}{2}}\ .$$

As above, the size of the grammar corresponding to the LZD parsing of s is $\Omega(k^2)$ whereas the size of the smallest grammar is $O(k)$; hence, the result follows.

$$\begin{aligned} &S \to A_2C_2A_3C_3\cdots A_kC_kbbA_1bbA_2\cdots bbA_{k-1}bb\Delta_0D_2\Delta_1D_3\cdots\Delta_kD_{k+2}X^{k/2}, \\ &A_0 \to \epsilon, C_0 \to \epsilon, D_0 \to \epsilon, A_i \to A_{i-1}a, C_i \to C_{i-1}c, D_i \to D_{i-1}d \text{ for } i \in [1..k{+}2], \\ &\Delta_i \to A_ibbA_{k-i} \text{ for } i \in [0..k], \quad X \to \Delta_k\Delta_{k-1}\,\Delta_k\Delta_{k-2}\cdots\Delta_k\Delta_{k/2+1}\,\Delta_k A_{k-1}\ . \end{aligned}$$

The analysis is similar to the above but simpler, so, we omit it. To additionally verify the correctness of both constructions, we conducted experiments on small k and, indeed, observed the described behavior; the code can be found in [1]. $\square$

We can also show that the upper bound for the approximation ratio of the LZ78 parsing given in [5] also applies to the LZD and LZMW parsings. For this, we will use the following known results.

Lemma 2.2 ([5]). *If there is a grammar of size m generating a given string, then this string contains at most mk distinct substrings of length k.*

Lemma 2.3 ([8]). *All phrases in the LZD parsing of a given string are distinct.*

Lemma 2.4. *Let $p_1 p_2 \cdots p_z$ be the LZMW parsing of a given string. Then, for any $i \in [2..z]$ and $j \in [i{+}2..z]$, we have $p_{i-1}p_i \neq p_{j-1}p_j$.*

Proof. If $p_{i-1}p_i = p_{j-1}p_j$ for $i < j-1$, then, by the definition of LZMW, the phrase p_{j-1} either is equal to $p_{i-1}p_i$ or contains $p_{i-1}p_i$ as a prefix, which is a contradiction. □

Now we are ready to show an upper bound on the approximation ratio of the LZD and LZMW parsings.

Theorem 2.5. *For all strings s of length n, the size of the grammar produced by the LZD/LZMW parsing is larger than the size of the smallest grammar generating s by at most a factor $O((n/\log n)^{2/3})$.*

Proof. The theorem can be shown by an analogous way as for the upper bound of the LZ78 parsing against the smallest grammar [5] (which is especially straightforward for LZD due to Lemma 2.3), but we provide a full proof for completeness.

Let us consider LZMW. Suppose that s is a string of length n and m^* is the size of the smallest grammar generating s. Let $p_1, p_2, \ldots, p_z$ be the LZMW parsing of s. It suffices to evaluate the number z of phrases since the total size of the grammar produced by LZMW is only by a constant factor larger than z.

Consider the multiset $S = \{p_1p_2, p_2p_3, \ldots, p_{z-1}p_z\}$ (recall that a multiset can contain an element more than one time). Let $p_{i_1}p_{i_1+1}, p_{i_2}p_{i_2+1}, \ldots, p_{i_{z-1}}p_{i_{z-1}+1}$ be a sequence of all strings from S sorted in increasing order of their lengths (again, some strings may occur more than once in the sequence). We partition the sequence by grouping the first $2 \cdot m^*$ strings, then the next $2 \cdot 2m^*$ strings, the next $2 \cdot 3m^*$ strings, and so forth. Let r be the minimal integer satisfying $2(1m^* + 2m^* + \cdots + rm^* + (r+1)m^*) > z$. This implies that $z = O(r^2 m^*)$.

By Lemma 2.4, any string has at most two occurrences in the multiset S. Also, it follows from Lemma 2.2 that s contains at most km^* distinct substrings of length k. Thus, for any $k \geq 1$, there are at most $2km^*$ strings from S that generate substrings of length k. This implies that each string in the kth group generates a substring of length at least k. Hence, we have that

$$2n \geq |p_{i_1}p_{i_1+1}| + |p_{i_2}p_{i_2+1}| + \cdots + |p_{i_{z-1}}p_{i_{z-1}+1}| \geq 2(1^2 m^* + 2^2 m^* + \cdots + r^2 m^*),$$

which implies that $r = O((n/m^*)^{1/3})$. By plugging this into $z = O(r^2 m^*)$, we obtain $z = O((n/m^*)^{2/3} m^*)$ and thus the approximation ratio of the grammar produced by LZMW is $O((n/m^*)^{2/3})$. Since $m^* = \Omega(\log n)$, we finally get the desired bound $O((n/\log n)^{2/3})$.

Let us sketch the analysis of LZD, which is very similar. In this case, we consider the set S' of all phrases $p_1, p_2, \ldots, p_z$ (not pairs as in LZMW) of the LZD parsing. Let $p_{i_1}, \ldots, p_{i_z}$ be the sequence of all strings from S' sorted by the increasing order of lengths. We partition the sequence into groups of size $1m^*, 2m^*, 3m^*, \ldots$ (without the factor 2 as in LZMW). It follows from Lemma 2.3 that any string occurs in S' at most once. Therefore, similar to the case of LZMW, we obtain $n = |p_{i_1}| + |p_{i_2}| + \cdots + |p_{i_z}| \geq 1^2 m^* + 2^2 m^* + \cdots + r^2 m^*$, which implies the result in the same way as above. □

3 Small-Space Computation

In this section we analyze the time required to compute the LZD and LZMW parsings using the $O(z)$-space algorithms described by Goto et al. [8] and Miller and Wegman [14], where z is the number of phrases. We focus on LZD throughout, but a very similar algorithm and analysis applies for LZMW. Goto et al. upperbound the runtime at $O(z(m + \min(z, m)\log\sigma))$, where m is the length of the longest LZD (or LZMW) phrase and σ is the size of the input alphabet. Because $m = O(n)$ and $z = O(n)$, the runtime is upper bounded by $O(n^2)$. Below we provide a lower bound of $\Omega(n^{5/4})$ on the worst-case runtime, but before doing so we provide the reader with a description of Goto et al.'s algorithm [8].[1]

Naïve parsing algorithms. In the compacted trie for a set of strings, each edge label ℓ is represented as a pair of positions delimiting an occurrence of ℓ in the set. In this way we can store the trie for $s_1, \ldots, s_k$ in $O(k)$ space. During parsing Goto et al. [8] maintain the dictionary of LZD phrases in a compacted trie. The trie is of size $O(z)$, but read-only random access to the input string is also required in order to determine the actual values of the strings on the edge labels.

Initially the trie is empty, consisting of only the root. At a generic step during parsing, when we go to compute the phrase $p_i = p_{i_1}p_{i_2}$ starting at position $j = |p_1p_2 \ldots p_{i-1}| + 1$, the trie contains nodes representing the phrases $p_1, p_2, \ldots, p_{i-1}$ and all the distinct symbols occurring in $s[1..j-1]$, and all these nodes (corresponding to phrases and symbols) are marked. Note that there may also be some nodes in the trie that do not correspond to any phrase, i.e., branching nodes. Let $s[j..k]$ be the longest prefix of $s[j..n]$ that can be found by traversing the trie from the root. If $s[j..k]$ cannot be matched even for $k = j$, then $s[j]$ is the leftmost occurrence of symbol $c = s[j]$ in s, and we add a child node of the root labelled with c, mark the node, and set it as the first element of the new phrase, i.e., $p_{i_1} = c$. Otherwise, the first element of p_i, p_{i_1}, is the string written on the path connecting the root and the lowest marked node on the path that spells $s[j..k]$. The second element, p_{i_2}, of the phrase is computed in a similar manner, by searching for $s[j + |p_{i_1}| + 1..n]$ in the trie.

After computing p_i we modify the trie by a standard procedure so that there is a marked node representing p_i: first, we traverse the trie from the root finding the longest prefix of p_i present in the trie, then, possibly, create one or two new nodes, and, finally, mark the node (which, probably, did not exist before) corresponding to p_i (the details can be found in any stringology textbook).

The time taken to compute a new phrase and update the trie afterwards is bounded by $O(m + \min(z, m) \log \sigma)$, where $m = O(n)$ is the length of the longest phrase (and therefore an upper bound on the length of the longest path in the trie), $\min(z, m)$ is an upper bound on the number of branching nodes, and $\log \sigma$ is the time taken to find the appropriate outgoing edge at each branching node during downward traversal. Over all z phrases the runtime is thus $O(z(m + \min(z, m) \log \sigma))$.

The LZMW construction algorithm of Miller and Wegman [14] is analogous but, unlike the LZD algorithm, when we go to compute the phrase p_i, the trie contains the strings $p_1p_2, p_2p_3, \ldots, p_{i-2}p_{i-1}$ and the nodes corresponding to these strings are marked. One can easily show that the running time of this algorithm is $O(z(m + \min(z, m) \log \sigma))$, where z and m are defined analogously as for LZD.

We call both these algorithms *naïve.*

Worst-case time of the naïve algorithms. Now let us investigate the worst-case time complexity of the naïve LZD and LZMW construction algorithms.

Theorem 3.1. *The naïve LZD and LZMW construction algorithms take time $\Omega(n^{\frac{5}{4}})$ in the worst case.*

Proof. Let $k \geq 8$ be an integer that is a power of two. We will describe a string s of length $n = \Theta(k^4)$ for which the basic LZD construction algorithm (see the above discussion) spends $\Theta(n^{\frac{5}{4}})$ time to process. The string s is composed of pairwise distinct letters $a_{i,j}$, for $i, j \in [1..k]$, and "separator" letters, all of which are denoted $\diamond$ and supposed to be distinct. We will first construct a prefix s' of s that forces the algorithm to fill the dictionary with a set of strings that are used as building blocks in further constructions. To this end, denote (with parentheses used only for convenience):

$$\begin{aligned}
&w_i = a_{i,1}a_{i,2} \cdots a_{i,k} \text{ for } i = 1, 2, \ldots, k \text{ and } w = w_1w_2 \cdots w_k,\\
&s_{pre,i} = w_i[1..2]w_i[1..3] \cdots w_i[1..k] \text{ for } i = 1, 2, \ldots, k,\\
&s_{suf,i} = w_i[k{-}1..k]w_i[k{-}2..k] \cdots w_i[2..k] \text{ for } i = 1, 2, \ldots, k,\\
&p = (s_{pre,1}s_{pre,2} \cdots s_{pre,k})(s_{suf,1}s_{suf,2} \cdots s_{suf,k}),\\
&q = (w_{k-2}w_{k-1})(w_{k-3}w_{k-2}w_{k-1}) \cdots (w_1w_2 \cdots w_{k-1})(w),\\
&s' = pq \cdot w^{2^1}w^{2^2} \cdots w^k(w_k[2..k]w^k)(w_k[3..k]w^k) \cdots (w_k[k..k]w^k).
\end{aligned}$$

Analyzing the prefix p of s', it is clear that the LZD construction algorithm adds to the dictionary exactly all prefixes and suffixes of the strings w_i for $i = 1, 2, \ldots, k$; parsing the string q, the algorithm adds the

[1]We concern ourselves here with LZD parsing, but it should be easy for the reader to see that the algorithms are trivially adapted to instead compute LZMW.

strings $w_{k-2}w_{k-1}, w_{k-3}w_{k-2}w_{k-1}, \ldots, w_1w_2\cdots w_{k-1}$, and $w_1w_2\cdots w_k = w$; then, processing the string $w^{2^1}w^{2^2}\cdots w^k$, the algorithm adds $w^{2^1}, w^{2^2}, \ldots, w^k$ (we are interested only in w^k); finally, the strings $w_k[2..k]w^k, w_k[3..k]w^k, \ldots, w_k[k..k]w^k$ are added. So, the algorithm adds to the dictionary exactly the following strings:

- all prefixes and suffixes of w_i (including w_i itself) for $i = 1, 2, \ldots, k$;
- $w_{k-2}w_{k-1}, w_{k-3}w_{k-2}w_{k-1}, \ldots, w_1w_2\cdots w_{k-1}$, and w;
- w^k along with $w^{k/2}, \ldots, w^{2^2}, w^2$ (we use only w^k in what follows);
- $w_k[2..k]w^k, w_k[3..k]w^k, \ldots, w_k[k..k]w^k$.

It is easy to verify that $|w| = k^2$, $|w^k| = k^3$, and $|s'| = \Theta(k^4)$. (The string $w_k[2..k]w^kw_k[3..k]$-$w^k\cdots w_k[k..k]w^k$ contributes the most to the length.)

We first provide an overview of our construction. The main load on the running time of the algorithm is concentrated in the following strings z_i:

$$z_i = w_i[2..k]w_{i+1}\cdots w_kw^{k-2}w_1\cdots w_i \text{ for } i = 1, 2, \ldots, k-2.$$

Put $s = s'x_1z_1\diamond\diamond x_2z_2\diamond\diamond\cdots x_{k-2}z_{k-2}\diamond\diamond$, where $x_1, \ldots, x_k$ are auxiliary strings defined below. Before processing of z_i, the algorithm processes x_i and adds the strings $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ and $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$ for $j \in [2..k]$ to the dictionary (see below). So, analyzing z_i, the algorithm consecutively "jumps", for $j = 2, 3, \ldots, k$, from the string $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ to $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$ and so on. The crucial point is that, while analyzing $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$, the algorithm does not know in advance that the string $w_k[j..k]w^k$ from the dictionary does not occur at this position and, since the length of the longest common prefix of the strings $w_k[j..k]w^k$ and $w_k[j..k]w^{k-j}w_1\cdots w_i\diamond\diamond$ is $\Theta(k - j + 1 + |w^{k-j}|)$, spends $\Theta(|w^{k-j}|) = \Theta((k-j)k^2)$ time verifying this. Therefore, the analysis of the string s takes $\Theta((k-2)\sum_{j=2}^{k}(k-j)k^2) = \Theta(k^5)$ time overall. Since $|z_i| = O(k^3)$ and, as it is shown below, $|x_i| = O(k^3)$, we have $n = |s| = \Theta(k^4)$ and the processing time is $\Theta(n^{\frac{5}{4}})$ as required. We now describe this in more detail.

We prove by induction that the following invariant is maintained: when the algorithm starts the processing of the suffix $x_iz_i\diamond\diamond\cdots x_{k-2}z_{k-2}\diamond\diamond$ of the string s (x_i are defined below), the dictionary contains the following set of strings:

- "building blocks" constructed during the processing of s';
- pairs of separators $\diamond\diamond$ (recall that all separators are distinct);
- for each $i' \in [1..i{-}1]$ and $j \in [2..k]$:

  $w_{i'}[j..k]w_{i'+1}\cdots w_{k-1}w_k[1..j{-}1]$ and $w_k[j..k]w_1\cdots w_{i'-1}w_{i'}[1..j]$,

  $w_{i'}[j..k]w_{i'+1}\cdots w_{k-1}$ and $w_k[j..k]w_1\cdots w_{i'-1}$,

  $w_{i'}[j..k]w_{i'+1}\cdots w_kw_1\cdots w_{i'-1}w_{i'}[1..j]$.

The strings from the last two lines in the above list are not used and appear as byproducts. (But it is still important to have them in mind to verify that the algorithm works as expected.) So, assume that, by inductive hypothesis, the invariant holds for all $i' \in [1..i{-}1]$ (it is trivial for $i = 1$).

Define x_i as follows (the parentheses are only for visual ease):

$$\begin{aligned}
u'_{i,j} &= (w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]),\\
u_{i,j} &= (w_k[j..k]w_1\cdots w_{i-2}w_{i-1}[1..j])(w_{i-1}[j{+}1..k])u'_{i,j},\\
v_{i,j} &= (w_i[j..k]w_{i+1}\cdots w_{k-1})(w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]),\\
x_1 &= (u'_{1,2}\diamond\diamond u'_{1,3}\diamond\diamond\cdots u'_{1,k-1}\diamond\diamond u'_{1,k}\diamond\diamond)(v_{1,2}\diamond\diamond v_{1,3}\diamond\diamond\cdots v_{1,k}\diamond\diamond),\\
x_i &= (u_{i,2}\diamond\diamond u_{i,3}\diamond\diamond\cdots u_{i,k-1}\diamond\diamond u'_{i,k}\diamond\diamond)(v_{i,2}\diamond\diamond v_{i,3}\diamond\diamond\cdots v_{i,k}\diamond\diamond), \text{ for } i \neq 1.
\end{aligned}$$

Observe that $|x_i| = O(k^3)$. Using the inductive hypothesis, one can prove that the algorithm adds the strings $w_k[j..k]w_1\cdots w_{i-1}$ (for $j \neq k$), $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$, $w_i[j..k]w_{i+1}\cdots w_{k-1}$, and $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ for $j \in [2..k]$ to the dictionary after the processing of x_i (plus several pairs

$\diamond\diamond$). It remains to show that the algorithm adds exactly the strings $w_i[j..k]w_{i+1}\cdots w_k w_1 \cdots w_{i-1}w_i[1..j]$, for $j \in [2..k]$, to the dictionary when processing z_i.

Observe that, for $j \in [2..k]$, $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ is the longest string from the dictionary that has prefix $w_i[j..k]$, and $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$ is the longest string from the dictionary that has prefix $w_k[j..k]$ and does not coincide with $w_k[j..k]w^k$. Hence, the algorithm consecutively "jumps" over the substrings w of the string z_i adding after each such "jump" the string $w_i[j..k]w_{i+1}\cdots w_k w_1\cdots w_{i-1}w_i[1..j]$ to the dictionary (for $j = 2, 3, \ldots, k$). No other strings are added.

Each time the algorithm processes a substring $w_k[j..k]w_1\cdots w_{i-1}w_i[1..j]$, it also verifies in $\Theta(ki + |w^{k-j}|)$ time whether the string $w_k[j..k]w^k$ occurs at this position. Therefore, by the above analysis, processing takes $\Theta(|s|^{\frac{5}{4}})$ time.

An analogous troublesome string for the naïve LZMW construction algorithm is as follows (again, all separators $\diamond$ are assumed to be distinct letters):

$$\begin{aligned}
&w_i = a_{i,1}a_{i,2}\cdots a_{i,k} \text{ and } w = w_1w_2\cdots w_k,\\
&s_{pre,i} = w_i[1..2]\diamond w_i[1..3]\diamond\cdots\diamond w_i[1..k]\diamond,\\
&s_{suf,i} = w_i[k{-}1..k]\diamond w_i[k{-}2..k]\diamond\cdots\diamond w_i[2..k]\diamond,\\
&p = s_{pre,1}s_{pre,2}\cdots s_{pre,k}s_{suf,1}s_{suf,2}\cdots s_{suf,k},\\
&q = w_{k-2}w_{k-1}\diamond w_{k-3}w_{k-2}w_{k-1}\diamond\cdots\diamond w_1w_2\cdots w_{k-1}\diamond w\diamond,\\
&s' = pqw^{2^1}\diamond w^{2^2}\diamond\cdots\diamond w^k\diamond w_k[2..k]w^k\diamond w_k[3..k]w^k\diamond\cdots\diamond w_k[k..k]w^k\diamond,\\
&y_j = w_k[j..k]w_1\diamond w_k[j..k]w_1w_2[1..j]\diamond,\\
&t_{i,j} = w_{i-2}[j{+}1..k]w_{i-1}[1..j]\diamond w_{i-1}[j{+}1..k]w_i[1..j],\\
&u_{i,j} = (w_k[j..k]w_1\cdots w_{i-3}w_{i-2}[1..j])(w_{i-2}[j{+}1..k]w_{i-1}[1..j]),\\
&v_{i,j} = w_i[j..k]w_{i+1}\cdots w_{k-1}\diamond w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1],\\
&x_i = t_{i,2}\diamond t_{i,3}\diamond\cdots\diamond t_{i,k-1}\diamond u_{i,2}\diamond u_{i,3}\diamond\cdots\diamond u_{i,k}\diamond v_{i,2}\diamond v_{i,3}\diamond\cdots\diamond v_{i,k}\diamond,\\
&z_i = w_i[2..k]w_{i+1}\cdots w_k w^{k-2}w_1\cdots w_i\diamond,\\
&s = s'y_2y_3\cdots y_k x_4z_4x_6z_6\cdots x_{2j}z_{2j}\cdots x_{k-2}z_{k-2}.
\end{aligned}$$

Let us explain on a high level why the LZMW algorithm works slowly on s. While analyzing the prefix $s'y_2y_3\cdots y_k$, the algorithm adds a number of "building block" strings into the LZMW dictionary, including the strings $w[j..k]w^k$ for $j = 2, 3, \ldots, k$ (recall that, unlike the LZD dictionary containing phrases, the LZMW dictionary contains pairs of adjacent phrases). Before the processing of z_i, the algorithm processes x_i and adds the strings $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ (from $v_{i,j}$), $w_k[j..k]w_1\cdots w_{i-2}w_{i-1}[1..j]$ (from $u_{i,j}$), and $w_{i-1}[j{+}1..k]w_i[1..j]$ (from $t_{i,j}$) to the dictionary. The concatenation of these three strings is $w_i[j..k]w_{i+1}\cdots w_k w_1\cdots w_{i-1}w_i[1..j]$, so, analyzing z_i, the algorithm consecutively "jumps", for $j = 2, 3, \ldots, k$, from the string $w_i[j..k]w_{i+1}\cdots w_{k-1}w_k[1..j{-}1]$ to $w_k[j..k]w_1\cdots w_{i-2}w_{i-1}[1..j]$ and then to $w_{i-1}[j{+}1..k]w_i[1..j]$, thus producing three new phrases (and then moves on to $j{+}1$). The point is that, while analyzing the string $w_k[j..k]w_1\cdots w_{i-2}w_{i-1}[1..j]$, the algorithm does not know in advance that the string $w_k[j..k]w^k$ from the dictionary does not occur at this position and, since the length of the longest common prefix of the strings $w_k[j..k]w^k$ and $w_k[j..k]w^{k-j}w_1\cdots w_i\diamond\diamond$ is $\Theta(k - j + 1 + |w^{k-j}|)$, spends $\Theta(|w^{k-j}|) = \Theta((k-j)k^2)$ time verifying this. Therefore, the analysis of the string s takes $\Theta((k/2)\sum_{j=2}^{k}(k-j)k^2) = \Theta(k^5)$ time overall. Since $n = |s| = \Theta(k^4)$, the processing time is $\Theta(n^{\frac{5}{4}})$ as required. We omit the detailed proof since it is very similar to the LZD case.

To additionally verify the correctness of both constructed examples, we performed the naïve LZD and LZMW algorithms (with some diagnostics to track their execution) on the examples for small k and, indeed, observed the expected "bad" behavior in the special positions described above. Our verifying code (it can be found in [1]) thoroughly checks the correspondence of the behavior of the parsers in the special positions to the behavior discussed in the above text. Thus, we hope that the correctness of both our constructions is well supported. □

We now explain how to decrease the alphabet size in the examples of Theorem 3.1. The construction for both parsing schemes relies on the following reduction.

Lemma 3.2. *Consider the parsing scheme LZD or LZMW and a string $s \in \Sigma^*$. There exists a string $t \in \{0,1\}^*$ of length $\Theta(|\Sigma|\log|\Sigma|)$ and a morphism ϕ with $\phi(\Sigma) \subseteq \{0,1\}^\ell$ for $\ell = \Theta(\log|\Sigma|)$ such that the parsing of $t \cdot \phi(s)$ consists of the parsing of t followed by the image with respect to ϕ of the parsing of s.*

Proof. We analyze the two parsing schemes separately. For LZD, we recursively define $A_L \subseteq \{0,1\}^{2^L}$, setting $A_0 = \{0,1\}$ and $A_L = \{xy : x, y \in A_{L-1} \wedge x \le y\}$ for $L > 0$. Let $(\alpha_i)_{i=1}^{\infty}$ be the infinite sequence of all elements of A_L, for all $L \geq 1$, with members of each set A_L listed in the lexicographic order; e.g., $\alpha_1, \dots, \alpha_{12} = 00, 01, 11, 0000, 0001, 0011, 0101, 0111, 1111, 00000000, 00000001, 00000011$. We will define $t = \alpha_1 \cdots \alpha_m$ for some m. Let us characterize parsings of such strings.

Claim. For any non-negative integer m and any string $w \in \{0,1\}^*$, the first m phrases of the LZD parsing of the binary string $\alpha_1 \cdots \alpha_m \cdot w$ are $\alpha_1, \dots, \alpha_m$.

Proof. We proceed by induction on m; the base case of $m = 0$ is trivial.

For $m > 0$, the inductive assumption implies that the first $m-1$ phrases are $\alpha_1, \dots, \alpha_{m-1}$. Our goal is to prove that the mth phrase is α_m. Before processing α_m, the LZD dictionary is $D = \{0, 1, \alpha_1, \dots, \alpha_{m-1}\}$. Suppose that $\alpha_m = xy \in A_L$ with $x, y \in A_{L-1}$. Recall that $x \leq y$; consequently, $D \cap (y \cdot \{0,1\}^*) = \{y\}$ and

$$D \cap (x \cdot \{0,1\}^*) = \{x\} \cup \{xy' : y' \in A_{L-1} \land x \leq y' < y\}.$$

Thus, the longest prefix of $\alpha_m \cdot w$ contained in D is x, and the longest prefix of $y \cdot w$ contained in D is y. This means that the mth phrase is indeed $\alpha_m = xy$. □

Consider a string $s \in \Sigma^n$. We choose the smallest L with $|A_L| \geq |\Sigma|$ and define $t = \alpha_1 \cdots \alpha_m$ so that t is shortest possible and the LZD dictionary after processing t contains at least $|\Sigma|$ elements of A_L. The morphism ϕ is then defined by injectively mapping Σ to these dictionary strings from A_L.

Note that $|A_{L-1}| \leq |\Sigma|$ and $m \leq |\Sigma| + \sum_{\ell=1}^{L-1} |A_\ell|$, so we have $m = \Theta(|\Sigma|)$, $\ell = 2^L = \Theta(\log |\Sigma|)$, and $|t| = \Theta(|\Sigma| \log |\Sigma|)$, as desired.

We are to prove that the LZD parsing of $t \cdot \phi(s)$ is $\alpha_1, \dots, \alpha_m, \phi(p_1), \dots, \phi(p_z)$, where $p_1, \dots, p_z$ is the LZD parsing of s. For this, we inductively prove that the LZD dictionary D after parsing $p_1 \cdots p_i$ is related to the LZD dictionary $\hat{D}$ after parsing $t \cdot \phi(p_1 \cdots p_i)$ by the following invariant: $\hat{D} \cap (\phi(\Sigma) \cdot \{0,1\}^*) = \phi(D)$. The base case follows from the claim ($\hat{D} \cap (\phi(\Sigma) \cdot \{0,1\}^*) = \phi(\Sigma) = \phi(D)$), and the inductive step is straightforward. This completes the proof for the LZD scheme.

The construction for LZMW is more involved, but the idea is the same. We recursively define $B_L \subseteq \{0,1\}^{2^L}$, setting $B_0 = \{0,1\}$ and $B_L = \{xy : x, y \in B_{L-1} \land xy \neq 1^{2^{L-1}} 0^{2^{L-1}}\}$ for $L > 0$. Let $(\beta_i)_{i=1}^{\infty}$ be the infinite sequence that lists all elements of B_L consecutively for all $L \geq 0$, with members of each B_L listed in the lexicographic order (i.e., $(\beta_i)_{i=1}^{\infty}$ is defined by analogy with $(\alpha_i)_{i=1}^{\infty}$ for LZD but starting with $L = 0$). For $\beta_m \in B_L$, define $b(\beta_m) = \beta_M \beta_m \cdot \beta_{M+1} \beta_m \cdots \beta_{m-1} \beta_m \cdot \beta_m$, where $\beta_M = 0^{2^L}$ is the first element of B_L in $(\beta_i)_{i=1}^{\infty}$. For example, $b(\beta_1) \cdots b(\beta_6) = 0 \cdot 0\ 1\ 1 \cdot 00 \cdot 00\ 01\ 01 \cdot 00\ 11\ 01\ 11\ 11 \cdot 0000$.

Claim. For $m \geq 1$, consider a binary string $b(\beta_1) \cdots b(\beta_m) \cdot 0^{|\beta_m|} \cdot w$ for $w \in \{0,1\}^*$. The LZMW parsing decomposes its fragments $b(\beta_i)$ into phrases of length $|\beta_i|$.

Proof. We proceed by induction on m. The base case $m = 1$ is straightforward: it suffices to note that the first phrase of $0 \cdot 0 \cdot w$ is 0. Below, we consider $m > 1$.

First, suppose that $\beta_m = 0^{2^L}$, i.e., $\beta_{m-1} = 1^{2^{L-1}} \in B_{L-1}$. Note that $b(\beta_m)$ starts with $0^{2^{L-1}}$, so the inductive hypothesis yields that the prefix $b(\beta_1) \cdots b(\beta_{m-1})$ is parsed as desired. Observe that after parsing this prefix, the LZMW dictionary is $D = \{1^{2^{\ell-1}} 0^{2^\ell} : 0 < \ell < L\} \cup \bigcup_{\ell=0}^{L} B_\ell$. Consequently, we obtain $D \cap (B_L \cdot \{0,1\}^*) = B_L$ and, therefore, $b(\beta_m) = \beta_m$ is parsed as claimed.

Finally, suppose that $\beta_m \in B_L \setminus \{0^{2^L}\}$. In this case, $\beta_{m-1} \in B_L$ and $\beta_M = 0^{2^L}$ for some $M < m$. Since $b(\beta_m)$ starts with $\beta_M = 0^{2^L}$, the inductive hypothesis lets us assume that the prefix $b(\beta_1) \cdots b(\beta_{m-1})$ is parsed as desired. Due to $1^{2^{L-1}} 0^{2^L - 1} \notin B_L$, after parsing this prefix, the LZMW dictionary D satisfies:

$$D \cap (B_L \cdot \{0,1\}^*) = B_L \cup \{\beta_k \beta_{k'} : M \leq k, k' < m \land (k, k') \neq (m-1, M)\}.$$

Let us consider the parsing of $b(\beta_m) 0^{2^L} w = \beta_M \beta_m \cdot \beta_{M+1} \beta_m \cdots \beta_{m-1} \beta_m \cdot \beta_m \cdot 0^{2^L} w$. One can inductively prove that before parsing $\beta_k \beta_m \cdot \beta_{k+1} \cdots$, for $M \leq k < m$, we have $D \cap (\beta_k \cdot \{0,1\}^*) = \{\beta_k\} \cup \{\beta_k \beta_{k'} : M \leq k' < m\}$, so the subsequent phrase is β_k. Next, before parsing $\beta_m \cdot \beta_{k+1} \cdots$, for $M \leq k < m$, we have $D \cap (\beta_m \cdot \{0,1\}^*) = \{\beta_m\} \cup \{\beta_m \beta_{k'} : M < k' \leq k\}$, so the subsequent phrase is β_m. Finally, before parsing $\beta_m \cdot 0^{2^L} w$, we have $D \cap (\beta_m \cdot \{0,1\}^*) = \{\beta_m\} \cup \{\beta_m \beta_{k'} : M < k' < m\}$, so the last phrase is also β_m. Thus, $b(\beta_m)$ is parsed as claimed. □

Consider a string $s \in \Sigma^n$. We choose the smallest L with $|B_L| \geq |\Sigma|$ and define $t = b(\beta_1) \cdots b(\beta_m)$ so that t is shortest possible and the LZMW dictionary after processing t contains at least $|\Sigma|$ members of B_L (note that $\beta_m \in B_{L-1}$ in this case). The morphism ϕ is then defined by injectively mapping Σ to these dictionary strings from B_L. Moreover, we put $\phi(s[1]) = 0^{2^L}$ so that the claim is applicable for $t \cdot \phi(s)$. The remaining proof is analogous to the LZD counterpart. We only need to observe that the LZMW dictionary additionally contains $\beta_m 0^{2^L}$, but $\beta_m 0^{2^{L-1}} \notin \phi(\Sigma)$ and, hence, this does not affect the parsing of $t \cdot \phi(s)$. $\square$

The hard binary examples are now straightforward to derive.

Theorem 3.3. *The naïve LZD and LZMW parsing algorithms take time $\Omega(n^{5/4}/\log^{1/4} n)$ in the worst case even on a binary alphabet.*

Proof. We apply Lemma 3.2 for a string $s \in \Sigma^*$ of length n constructed in the proof of Theorem 3.1 for the appropriate parsing algorithm, which results in a binary string $\hat{s} := t \cdot \phi(s)$. Without loss of generality, we may assume $|\Sigma| \leq n$, so $\hat{n} := |\hat{s}| = \Theta(|\Sigma| \log |\Sigma| + n \log |\Sigma|) = \Theta(n \log |\Sigma|)$. Recall that the naïve parsing algorithm traverses at least $\Omega(n^{5/4})$ trie edges while parsing s. Since the parsing of the suffix $\phi(s)$ of $\hat{s}$ is the ϕ-image of the parsing of s, this algorithm traverses at least $\Omega(n^{5/4} \log |\Sigma|)$ trie edges while parsing $\hat{s}$. In terms of $\hat{n}$, the running time is at least $\Omega(\hat{n}^{5/4}/\log^{1/4} |\Sigma|)$, which is $\Omega(\hat{n}^{5/4}/\log^{1/4} \hat{n})$ due to $|\Sigma| \leq n < \hat{n}$. $\square$

4 Faster Small-Space Computation

In this section we describe a new parsing algorithm that works in $O(n + z \log^2 n)$ time (randomized, in expectation) and uses $O(z \log n)$ working space to parse the input string over the integer alphabet $\{0, 1, \ldots, n^{O(1)}\}$. Before getting to the algorithm itself, we review four tools that are essential for it: Karp–Rabin hashing [11], AVL-grammars of Rytter [15], the dynamic z-fast trie of Belazzougui et al. [2], and the dynamic marked ancestor data structure of Westbrook [17].

Karp–Rabin hashing. A Karp–Rabin [11] hash function ϕ has the form $\phi(s[1..n]) = \sum_{i=1}^{n} s[i]\delta^{i-1} \bmod p$, where p is a fixed prime and δ is a randomly chosen integer from the range $[0..p-1]$ (this is a more popular version of the original hash proposed in [11]). The value $\phi(s)$ is called s's Karp–Rabin hash. It is well-known that, for any $c > 3$, if $p > n^c$, then the probability that two distinct substrings of the given input string of length n have the same hash is less than $\frac{1}{n^{c-3}}$.

We extensively use the property that the hash of the concatenation $s_1 s_2$ of two strings s_1 and s_2 can be computed as $(\phi(s_1) + \delta^{|s_1|}\phi(s_2)) \bmod p$. Therefore, if the values $\phi(s_1)$ and $\phi(s_2)$ are known and $p \leq n^{O(1)}$, then $\phi(s_1 s_2)$ can be calculated in $O(1)$ time provided the number $(\delta^{|s_1|} \bmod p)$ is known.

AVL-grammars. Consider a context-free grammar G that generates a string s (and only s). Denote by $Tree(G)$ the derivation tree of s. We say that G is an *AVL-grammar* (see [15]) if G is in the Chomsky normal form and, for every internal node v of $Tree(G)$, the heights of the trees rooted at the left and right children of v differ by at most 1. The following result straightforwardly follows from the algorithm of Rytter described in [15].

Lemma 4.1 (see [15, Th. 2])**.** *Let G be an AVL-grammar generating a prefix $s[1..i{-}1]$ of a string s. Suppose that the string $s[i..k]$ occurs in $s[1..i{-}1]$; then one can construct an AVL-grammar generating the string $s[1..k]$ in $O(\log i)$ time modifying at most $O(\log i)$ rules in G.*

Let G be an AVL-grammar generating a string s. It is well-known that, for any substring $s[i..j]$, one can find in $O(\log n)$ time $O(\log n)$ non-terminals $A_1, \ldots, A_k$ such that $s[i..j]$ is equal to the string generated by $A_1 \cdots A_k$. Hence, if each non-terminal A of G is augmented with the Karp–Rabin hash $\phi(t)$ of the string t generated by A and with the number $\delta^{|t|} \bmod p$, then we can compute $\phi(s[i..j])$ in $O(\log n)$ time. One can show that, during the reconstruction of the AVL-grammar in Lemma 4.1, it is easy to maintain the described integers augmenting the non-terminals (see [15]).

Z-fast tries. Let x be a string such that one can compute the Karp–Rabin hash of any prefix of x in $O(t_x)$ time. The z-fast trie [2] is a compacted trie containing a dynamic set of variable-length strings that supports the following operations:

- we can find (w.h.p.) in $O(t_x \log |x|)$ time the highest explicit node v such that the longest prefix of x present in the trie is written on the root-v path;
- we can insert x into the trie in $O(|x| + t_x \log |x|)$ randomized time.

The space occupied by the z-fast trie is $\Theta(k)$, where k is the number of strings inserted in the trie.

Dynamic marked ancestor. Let T be a dynamic compacted trie (or just tree) with k nodes. The dynamic marked ancestor data structure of [17] supports the following two operations on T (both in $O(\log k)$ time): for a given node v, (1) mark v, (2) find the nearest marked ancestor of v (if any).

Algorithm. Our faster parsing algorithm computes the LZD phrases from left to right one by one, spending $O(\log^{O(1)} n)$ time on each phrase. We maintain an AVL-grammar G for the prefix $s[1..i{-}1]$ of s we have already parsed, and a z-fast trie T containing the first phrases $p_1, p_2, \ldots, p_r$ of the LZD parsing of s such that $s[1..i{-}1] = p_1 p_2 \cdots p_r$. We augment T with the dynamic marked ancestor data structure and mark all nodes corresponding to phrases (i.e., all nodes v such that the string written on the path from the root to v is equal to $t \in \{p_1, \ldots, p_r\}$). We augment each non-terminal of G with the Karp–Rabin hash $\phi(t)$ of this non-terminal's expansion t and with the number $\delta^{|t|} \bmod p$, so that the hash of any substring of $s[1..i{-}1]$ can by calculated in $O(\log n)$ time.

Suppose we are looking for the first part of the next phrase and that, in addition to having parsed $s[1..i{-}1]$, we have already read $s[i..j{-}1]$ without parsing it — but we have found the endpoints of an occurrence of $s[i..j{-}1]$ in $s[1..i{-}1]$. (Notice $s[i..j{-}1]$ can be empty, i.e., $i = j$.) Denote by x the longest prefix of $s[i..j{-}1]$ that is also a prefix of some of the phrases $p_1, \ldots, p_r$. Since we can compute quickly with G the hash of any prefix of $s[i..j{-}1]$, we can use the z-fast search to find in $O(\log^2 n)$ time a node v of T such that x is written on the path connecting the root and v. Let $s[\ell_v..r_v]$ be a substring of $s[1..i{-}1]$ corresponding to v (the numbers ℓ_v and r_v are stored with the node v). Using hashes and the binary search, we find the longest common prefix of the strings $s[i..j{-}1]$ and $s[\ell_v..r_v]$ (with high probability) in $O(\log^2 n)$ time; this prefix must be x.

If $s[i..j{-}1] \neq x$, then we perform a marked-ancestor query on the vertex corresponding to x (which can be found in $O(\log^2 n)$ time in the same way as v) and thus find the longest phrase that is a prefix of $s[i..j{-}1]$. We take that phrase as the first part of the next phrase and start over, looking for the second part, with the remainder of $s[i..j{-}1]$ now being what we have read but not parsed (of which we know an occurrence in $s[1..i{-}1]$). On the other hand, if $s[i..j{-}1] = x$, then we read $s[j..n]$ in blocks of length $\log^2 n$, stopping when we encounter an index k such that $s[i..k]$ is not a prefix of a phrase $p_1, \ldots, p_r$; the details follow.

Suppose that we have read q blocks and the concatenation $s[i..j + q\log^2 n - 1]$ of $s[i..j{-}1]$ and the q previous blocks is a prefix of a phrase $t \in \{p_1, \ldots, p_r\}$. We compute in $O(\log^2 n)$ time the hashes of all the prefixes of the block $s[j + q\log^2 n..j + (q+1)\log^2 n - 1]$, which allows us to compute the hash of any prefix of $s[i..j + (q+1)\log^2 n - 1]$ in $O(\log n)$ time. Therefore, again using z-fast search and binary search, we can check in $O(\log^2 n)$ time if the block $s[j + q\log^2 n..j + (q+1)\log^2 n - 1]$ contains such a k — and, if so, find it. If k is not found, then using information from the search, we can find a phrase $t' \in \{p_1, \ldots, p_r\}$ — which may or may not be equal to t — such that $s[i..j + (q+1)\log^2 n - 1]$ is a prefix of t'; we then proceed to the (q+2)nd block.

Once we have found such a k, we conceptually undo reading the characters from $s[k]$ onwards (which causes us to re-read later those $O(\log^2 n)$ characters), then perform a search and marked-ancestor query in T, which returns the longest phrase that is a prefix of $s[i..k{-}1]$. We take that longest phrase as the first part of the next phrase and start over, looking for the second part, with the remainder of $s[i..k{-}1]$ now being what we have read but not parsed (of which we know an occurrence in $s[1..i{-}1]$).

Once we have found both the first and second parts of the next phrase — say, p'_1 and p'_2 — we add the next phrase $p_{r+1} = p'_1 p'_2$ to G (by Lemma 4.1) and to T, which takes $O(|p_{r+1}| + \log^2 n)$ time. In total, since processing each block takes $O(\log^2 n)$ time and the algorithm processes at most $z + \frac{n}{\log^2 n}$ blocks, we parse s in $O(n + z\log^2 n)$ time. Our space usage is dominated by G, which takes $O(z \log n)$ space.

Finally, we verify in a straightforward manner in $O(n)$ time whether the constructed parsing indeed encodes the input string. If not (which can happen with probability $\frac{1}{n^{c-3}}$, where $p > n^c$), we choose a different random $\delta \in [0..p{-}1]$ for the Karp–Rabin hash and execute the whole algorithm again.

The computation of the LZMW parsing in $O(n + z\log^2 n)$ expected time and $O(z\log n)$ space is similar: the z-fast trie stores pairs $p_1p_2, p_2p_3, \ldots, p_{z-1}p_z$ of adjacent phrases in this case and the nodes corresponding to these pairs are marked. We omit the details as they are straightforward.

5 Concluding Remarks

We believe that our new parsing algorithms can be implemented efficiently, and we leave this as future work. Perhaps a more interesting question is whether there exists an LZD/LZMW parsing algorithm with better working space and the same (or better) runtime. We note that the algorithmic techniques we have developed here can also be applied to, e.g., develop more space-efficient parsing algorithms for LZ-End [13], a variant of LZ77 [18] with which each phrase $s[i..j]$ is the longest prefix of $s[i..n]$ such that an occurrence of $s[i..j{-}1]$ in $s[1..i{-}1]$ ends at a phrase boundary. Kempa and Kosolobov [12] very recently gave an LZ-End parsing algorithm that runs in $O(n\log\ell)$ expected time and $O(z+\ell)$ space, where ℓ is the length of the longest phrase and z is the number of phrases.

To reduce Kempa and Kosolobov's space bound, we keep an AVL-grammar (again augmented with the non-terminals' Karp–Rabin hashes, meaning our algorithm Monte-Carlo) of the prefix of s we have processed so far; a list of the endpoints of the phrases so far, in the right-to-left lexicographic order of the prefixes ending at the phrases' endpoints; and an undo stack of the phrases so far. For each character $s[k]$ in turn, for $1 \le k \le n$, in $O(\log^{O(1)} n)$ time we use the grammar and the list to find the longest suffix $s[j..k]$ of $s[1..k]$ such that an occurrence of $s[j..k{-}1]$ in $s[1..j{-}1]$ ends at a phrase boundary. We use the undo stack to remove from the grammar, the list, and the stack itself, all the complete phrases lying in the substring $s[j..k{-}1]$, and then add the phrase consisting of the concatenation of those removed phrases and $s[k]$. By [12, Lemma 3], we remove at most two phrases while processing $s[k]$, so we still use a total of $O(\log^{O(1)} n)$ worst-case time for each character of s. Again, the space bound is dominated by the grammar, which takes $O(z\log n)$ words. We leave the details for the full version of this paper.

Regarding compression performance, we have shown that like their ancestor, LZ78, both LZD and LZMW sometimes approximate the smallest grammar poorly. This, of course, does not necessarily detract from their usefulness in real compression tools; now however, practitioners have a much clearer picture of these algorithms' possible behavior. The future work includes closing the gap between the lower bound $\Omega(n^{\frac{1}{3}})$ and the upper bound $O((n/\log n)^{\frac{2}{3}})$ for the approximation ratio and designing parsing algorithms with better guarantees.

Acknowledgements

We thank H. Bannai, P. Cording, K. Dabrowski, D. Hücke, D. Kempa, L. Salmela for interesting discussions on LZD at the 2016 StringMasters and Dagstuhl meetings. Thanks also go to D. Belazzougui for advice about the z-fast trie and to the anonymous referees.

References

[1] Supplementary materials for the present paper: C++ code for described experiments. `https://bitbucket.org/dkosolobov/lzd-lzmw`.

[2] Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 6393 of *LNCS*, pages 159–172. Springer, 2010. `doi:10.1007/978-3-642-16321-0_15`.

[3] Djamal Belazzougui, Patrick Hagge Cording, Simon J. Puglisi, and Yasuo Tabei. Access, rank, and select in grammar-compressed strings. In *Proc. 23rd Annual European Symposium on Algorithms (ESA)*, volume 9294 of *LNCS*, pages 142–154. Springer, 2015. `doi:10.1007/978-3-662-48350-3_13`.

[4] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM Journal on Computing*, 44(3):513–539, 2015. doi:10.1137/130936889.

[5] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and abhi shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005. doi:10.1109/TIT.2005.850116.

[6] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011. doi:10.3233/FI-2011-565.

[7] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *Proc. 6th International Conference on Language and Automata Theory and Applications (LATA)*, volume 7183 of *LNCS*, pages 240–251. Springer, 2012. doi:10.1007/978-3-642-28332-1_21.

[8] Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD factorization: simple and practical online grammar compression with variable-to-fixed encoding. In *Proc. 25th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 9133 of *LNCS*, pages 219–230. Springer, 2015. doi:10.1007/978-3-319-19929-0_19.

[9] Danny Hücke, Markus Lohrey, and Carl Philipp Reh. The smallest grammar problem revisited. In *Proc. 23rd International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 9954 of *LNCS*, pages 35–49. Springer, 2016. doi:10.1007/978-3-319-46049-9_4.

[10] Tomohiro I, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Efficient Lyndon factorization of grammar compressed text. In *Proc. 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 7922 of *LNCS*, pages 153–164. Springer, 2013. doi:10.1007/978-3-642-38905-4_16.

[11] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Reseach and Development*, 31(2):249–260, 1987. doi:10.1147/rd.312.0249.

[12] Dominik Kempa and Dmitry Kosolobov. LZ-End parsing in compressed space. In *Proc. Data Compression Conference (DCC)*, pages 350–359. IEEE, 2017. doi:10.1109/DCC.2017.73.

[13] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

[14] Victor S. Miller and Mark N. Wegman. Variations on a theme by Ziv and Lempel. In *Proc. NATO Advanced Research Workshop on Combinatorial Algorithms on Words*, volume 12 of *NATO ASI*, pages 131–140. Springer, 1985. doi:10.1007/978-3-642-82456-2_9.

[15] Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

[16] Toshiya Tanaka, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Computing convolution on grammar-compressed text. In *Proc. Data Compression Conference (DCC)*, pages 451–460. IEEE, 2013. doi:10.1109/DCC.2013.53.

[17] Jeffery Westbrook. Fast incremental planarity testing. In *Proc. 19th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 623 of *LNCS*, pages 342–353. Springer, 1992. doi:10.1007/3-540-55719-9_86.

[18] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi:10.1109/TIT.1977.1055714.

[19] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978. doi:10.1109/TIT.1978.1055934.