LZD Factorization: Simple and Practical Online Grammar Compression with Variable-to-Fixed Encoding

Keisuke Goto(✉), Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University, Fukuoka, Japan
keisuke.gotou@gmail.com, {bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract. We propose a new variant of the LZ78 factorization which we call the LZ Double-factor factorization (LZD factorization). Each factor of the LZD factorization of a string is the concatenation of the two longest previous factors, while each factor of the LZ78 factorization is that of the longest previous factor and the following character. Interestingly, this simple modification drastically improves the compression ratio in practice. We propose two online algorithms to compute the LZD factorization in $O(m(M + \min(m, M) \log \sigma))$ time and $O(m)$ space, or in $O(N \log \sigma)$ time and $O(N)$ space, where m is the number of factors to output, M is the length of the longest factor(s), N is the length of the input string, and σ is the alphabet size. We also show two versions of our LZD factorization with variable-to-fixed encoding, and present online algorithms to compute these versions in $O(N + \min(m, 2^L)(M + \min(m, M, 2^L) \log \sigma))$ time and $O(\min(2^L, m))$ space, where L is the bit-length of each fixed-length code word. The LZD factorization and its versions with variable-to-fixed encoding are actually grammar-based compression, and our experiments show that our algorithms outperform the state-of-the-art online grammar-based compression algorithms on several data sets.

1 Introduction

Large-scale, highly repetitive texts such as collections of genomes of the same or similar species or the edit history of version controlled documents, have been increasing. Grammar compression algorithms, which are compression algorithms that output a compressed representation of the input text in the form of a context free grammar (CFG), have recently been gaining renewed interest since they are effective for such text collections [3], and also since CFGs are a convenient compressed representation that allows for various efficient processing on the strings without explicit decompression, e.g. pattern matching [13], q-gram frequencies [4], and edit-distance [5] computation.

While many previous grammar compression algorithms such as RE-PAIR [6] or SEQUITUR [10] give good compression ratios and run in linear time and working space, smaller working space is essential in order to compress large-scale data that does not fit in main memory. Maruyama et al. [7] proposed a fast and space efficient algorithm OLCA, which uses a simple strategy to

F. Cicalese et al. (Eds.): CPM 2015, LNCS 9133, pp. 219–230, 2015.
DOI: 10.1007/978-3-319-19929-0_19

determine the priority in selecting pairs of consecutive characters to form a production rule. Their algorithm runs online, and the working space depends only on the output, i.e., the compressed size of the input string. The working space was further reduced to the information theoretic lower bound of the output size in [9]. Maruyama and Tabei [8] proposed a variant that uses only constant working space, at the cost of some degradation in the compression ratio. Sekine et al. [12] proposed a modified version of RE-PAIR, called ADS, that splits the input string into blocks and compresses each block. In order to maintain a good compression ratio, they devised a technique to reuse non-terminal variables that are created and used frequently in each block, to the next block. Each non-terminal variable is encoded as a fixed-length code word, and since the length of the decompressed string that a code represents may vary, it is a variable-to-fixed code. The algorithm runs in $O(N)$ time and $O(B + 2^L)$ working space, where N is the length of the input string, B is the block size, and L is the bit-length of each fixed-length code word.

In this paper, we propose a new grammar-based compression based on the LZ78 factorization, which we call the *LZ Double-factor factorization* (*LZD factorization*). While each factor of the LZ78 factorization of a string is the longest previous factor and the following character, each factor of the LZD factorization is the concatenation of the two longest previous factors. We propose two online algorithms to compute the LZD factorization in $O(m(M + \min(m, M) \log \sigma))$ time and $O(m)$ space, or in $O(N \log \sigma)$ time and $O(N)$ space, where m is the number of factors to output, M is the length of the longest factor(s), N is the length of the input string, and σ is the alphabet size. We also show two versions of our LZD factorization with variable-to-fixed encoding, and present online algorithms to compute these versions in $O(N+\min(m, 2^L)(M+\min(m, M, 2^L) \log \sigma))$ time and $O(\min(2^L, m))$ space, where L is the bit-length of each fixed-length code word. When L can bee seen as a constant, these algorithms run in $O(N)$ time and $O(1)$ space. Computational experiments show that, in practice, our algorithms run fast and compress well while requiring small working space, outperforming the state-of-the-art online grammar-based compression algorithms on several data sets.

2 Preliminaries

Let Σ be a finite *alphabet*, and let $\sigma = |\Sigma|$. An element of Σ^* is called a *string*. The length of a string T is denoted by $|T|$. The empty string ε is the string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. If a prefix X (resp. substring Y, suffix Z) is of a string T is shorter than T, then it is called a *proper prefix* (resp. *proper substring*, *proper suffix*) of T. The set of suffixes of T is denoted by $\mathbf{Suffix}(T)$.

The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of T that begins at position i and ends at position j is denoted by $T[i..j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i..j] = \varepsilon$ if $j < i$. For convenience, we assume that $T[|T|] =$\$, where \$ is a special delimiter character that does not occur elsewhere in the string.

The Patricia tree of a set S of k strings, denoted $\mathbf{PT}_S$, is a rooted tree satisfying the following: (1) each edge is labeled with a non-empty substring of a string in S, (2) the labels of any two distinct out-going edges of the same node must begin with distinct characters; (3) for each string $s \in S$ there exists a node v such that $\mathbf{str}(v) = s$, where $\mathbf{str}(v)$ represents the concatenation of the edge labels from the root to v; (4) a string p is a non-empty prefix of a string $s \in S$ iff there are nodes u, v such that u is the parent of v, $\mathbf{str}(u)$ is a proper prefix of p, and p is a prefix of $\mathbf{str}(v)$. Because of conditions (2)-(4), there are at most k non-branching nodes (including leaves) and at most $k-1$ branching nodes in $\mathbf{PT}_S$. Also, if we represent each edge label ℓ by a pair of the beginning and ending positions of an occurrence of ℓ in one of the strings in S, then $\mathbf{PT}_S$ can be stored in $O(k)$ space (excluding the string S). For a node u of $\mathbf{PT}_S$, let $\mathbf{depth}(u) = |\mathbf{str}(u)|$. If the string p of Condition (4) is $\mathbf{str}(v)$ itself, then we say that p is represented by an *explicit* node of $\mathbf{PT}_S$. Otherwise (if p is a proper prefix of $\mathbf{str}(v)$), then we say that it is represented by an *implicit* node of $\mathbf{PT}_S$.

The suffix tree of a string T, denoted $\mathbf{ST}_T$, is the Patricia tree of $\mathbf{Suffix}(T)$, namely $\mathbf{ST}_T = \mathbf{PT}_{\mathbf{Suffix}(T)}$. Since we have assumed T terminates with a special character \$, there is a one-to-one correspondence between the suffixes of T and the leaves of $\mathbf{ST}_T$. $\mathbf{ST}_T$ has at most $2N-1$ nodes, and can be stored in $O(N)$ space, where $N = |T|$. For a string T of length N over an alphabet of size σ, $\mathbf{ST}_T$ can be constructed in $O(N \log \sigma)$ time and $O(N)$ space in an online manner [14].

3 LZD Factorization

We propose a new greedy factorization of a string inspired by the LZ78 factorization [16], which is able to achieve better compression ratios. We simply change the definition of a factor f_i, from the pair of the longest previously occurring factor and the immediately following character, to the pair of the longest previously occurring factor f_{j_1} and the longest previously occurring factor f_{j_2} which also appears at position $|f_1 \cdots f_{i-1}| + |f_{j_1}| + 1$. We call this new factorization the *LZ Double-factor factorization* (*LZD*), and its formal definition is the following:

Definition 1 (LZD Factorization). The LZD factorization of a string T of length N, denoted $\mathbf{LZD}_T$, is the factorization $f_1, \ldots, f_m$ of T such that $f_0 = \varepsilon$, and for $1 \le i \le m$, $f_i = f_{i_1} f_{i_2}$ where f_{i_1} is the longest prefix of $T[k..N]$ with $f_{i_1} \in \{f_j \mid 1 \le j < i\} \cup \Sigma$, f_{i_2} is the longest prefix of $T[k + |f_{i_1}|..N]$ with $f_{i_2} \in \{f_j \mid 0 \le j < i\} \cup \Sigma$, and $k = |f_1 \cdots f_{i-1}| + 1$.

Note that for any $1 \le i < m$ the length of f_i is at least 2, while f_m can be of length 1. This happens only when $|f_1 \cdots f_{m-1}| = N - 1$.

$\mathbf{LZD}_T = f_1, \ldots, f_m$ can be represented by a sequence of m integer pairs, where each pair (i_1, i_2) represents the ith factor $f_i = f_{i_1} f_{i_2}$. For example, the LZD factorization of string `abaaababababaabbbbabab`\$ is $f_1 =$ `ab`, $f_2 =$ `aa`, $f_3 =$ `abab`, $f_4 =$ `abaa`, $f_5 =$ `bb`, $f_6 =$ `bbabab`, $f_7 =$ \$, and can be represented by (`a`, `b`), (`a`, `a`), (1, 1), (1, 2), (`b`, `b`), (5, 3), and (\$, 0).

One can regard $\mathbf{LZD}_T$ as a context-free grammar which only generates string T, with $m+1$ production rules $S \to f_1 \cdots f_m$, $f_i \to f_{i_1} f_{i_2}$ for $1 \le i \le m$, where the set of rules $f_i \to f_{i_1} f_{i_2}$ $(1 \le i \le m)$ is called the dictionary.

Lemma 1. *For any string T, all factors of $\mathbf{LZD}_T$ are different.*

Proof. Let $\mathbf{LZD}_T = f_1, \ldots, f_m$. Since $f_m[|f_m|] = \$$, f_m is different from any other factors. Assume on the contrary that $f_h = f_i$ for some $1 \le h < i < m$. Since both f_{i_1} and f_{i_2} are of length at least 1, $|f_{i_1}| < |f_i|$. However, we have assumed $f_h = f_i$, and this contradicts that f_{i_1} is the longest prefix of $T[|f_1 \cdots f_{i-1}|+1..N]$ which belongs to $\{f_j \mid 1 \le j < i\} \cup \Sigma$. Hence each factor f_i is distinct. □

Using the idea of [16] and Lemma 1, we get the following lemma:

Lemma 2. *For any string T of length N, the number of factors in $\mathbf{LZD}_T$ is $O(N/\log_\sigma N)$.*

Let $F = \{f_0, \ldots, f_m\}$ be the set of factors of $\mathbf{LZD}_T$. In a similar way to the case of LZ78 factorization, computing $\mathbf{LZD}_T$ reduces to computing $\mathbf{PT}_F$, the Patricia tree of F. We call $\mathbf{PT}_F$ the *LZD* tree of T. Figure 1 illustrates the LZD tree of the example string `abaaababababaabbbbabab`$.

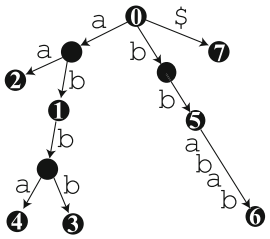

Fig. 1. The LZD tree for string `abaaababababaabbbbabab`$. Each node numbered i represents the ith factor f_i of the LZD factorization of the string

In what follows, we will propose two algorithms to compute $\mathbf{LZD}_T$ for a given string T of length N in an online manner. The first one is space-efficient, namely, its extra space usage is linear in the number of factors in $\mathbf{LZD}_T$. The second one is fast, namely, it runs in $O(N \log \sigma)$ time.

3.1 Space-Efficient Online Algorithm for LZD Factorization

We present an online algorithm to compute $\mathbf{LZD}_T$ for a string T of length N in $O(m(M + \min\{M, m\} \log \sigma))$ time using $O(m)$ working space, where m is the number of factors in $\mathbf{LZD}_T$ and M is the length of the longest factor in $\mathbf{LZD}_T$.

The LZD tree of a given string T can be computed incrementally, in quite a similar way to the LZ78 trie [16], as follows: We first construct a tree only with the root. To compute a factor $f_i = f_{i_1} f_{i_2}$ starting at a position $k = |f_1 \cdots f_{i-1}| + 1$, we assume that the LZD tree contains nodes which represent all previous factors f_1 to f_{i-1}, and these nodes are marked. We also assume that the LZD tree contains nodes which represent all characters occurring in $T[1..|f_1 \cdots f_{i-1}|]$, and these nodes are marked. Let $T[k..q]$ be the longest prefix of $T[k..N]$ that is represented by the LZD tree, where $k \le q \le N$. This string $T[k..q]$ can be computed by traversing the tree from the root. If k is the first occurrence of character $c = T[k]$ (namely $q = 0$), then we create a new child of

the root representing c, and mark this node. The first element f_{i_1} is c in this case. Otherwise, since there are at most $\min(m-1, M-1)$ branching nodes in any path of the LZD tree, and since $\mathbf{depth}(v) \leq M$ for any leaf v, the number of character comparisons to compute $T[k..q]$ is $O(M + \min(m, M) \log \sigma)$. Then, the lowest marked node in the path which spells out $T[k..q]$ is exactly the first element f_{i_1} of f_i. The second element f_{i_2} of f_i can be computed analogously, traversing the LZD tree with $T[k + |f_{i_1}|..N]$ in $O(M + \min(m, M) \log \sigma)$ time. After computing f_i, we update the LZD tree so that f_i is represented by an explicit marked node in the tree. Recall that in the LZD tree there always exists a path spelling out f_{i_1} from the root. We traverse f_{i_2} from the end of this path, to compute the longest prefix y of f_i that is represented by the current LZD tree. There are four cases to consider:

1. If $y = f_i$ and f_i is represented by an explicit node u (i.e., $\mathbf{str}(u) = f_i$), then we simply mark u. Note that, by Lemma 1, u was always unmarked before computing f_i.
2. If $y = f_i$ and f_i is represented by an implicit node, then we create a new internal non-branching node v such that $\mathbf{str}(v) = f_i$ by splitting the edge on which the path spelling out f_i ends. We then mark v.
3. If $|y| < |f_i|$ and f_i is represented by an explicit node u, then we create a new leaf node v such that $\mathbf{str}(v) = f_i$, with a new edge from u to v. We then mark v.
4. If $|y| < |f_i|$ and f_i is represented by an implicit node, then we first create a new internal node u such that $\mathbf{str}(u) = y$, by splitting the edge on which the path spelling out y ends. Next, we create a new leaf node v such that $\mathbf{str}(v) = f_i$, with a new edge from u to v. We finally mark v.

Since we repeat the above procedure m times, it takes a total of $O(m(M + \min(m, M) \log \sigma))$ time to compute the LZD tree for all the factors. Notice that $N \leq mM$, and hence N is hidden in the above time complexity. Since the LZD tree is the Patricia tree for the set of m factors of $\mathbf{LZD}_T$, the size of the tree (and hence the extra space requirement of this algorithm) is $O(m)$.

Since $\mathbf{LZD}_T$ is a kind of context-free grammar which only generates string T, we can obtain the original string T in $O(N)$ time from $\mathbf{LZD}_T$.

The following theorem summarizes this subsection.

Theorem 1 (Space-Efficient Online LZD Factorization). *Given a string T of length N, we can compute $\mathbf{LZD}_T = f_1 \cdots f_m$ in $O(m(M+\min(m, M) \log \sigma))$ time and $O(m)$ space in an online manner, where M is the length of the longest factor in $\mathbf{LZD}_T$.*

Since $m = O(N/\log_\sigma N)$ and $M = O(N)$[1], the space-efficient algorithm takes $O(N^2/\log_\sigma N)$ time. However, we have not found an instance which gives the above bound. As we will see in Sect. 5, this algorithm runs fast in practice.

[1] The bound $M = O(N)$ can be achieved with string $a^{N-1}\$$ with $N - 1 = 2^k$ for some k. Observe that $f_1 = aa$, $f_2 = f_1 f_1 = aaaa$, ..., $f_{m-1} = a^{\frac{N-1}{2}}$, and $f_m = \$$.

3.2 Fast Online Algorithm for LZD Factorization

Here, we present a fast online algorithm to compute $\mathbf{LZD}_T$ for a given string T of length N. Our algorithm uses the suffix tree $\mathbf{ST}_T$ of a given string T. Since every factor f_i of $\mathbf{LZD}_T = f_1, \ldots, f_m$ is a substring of T, it is also represented by either an implicit or explicit node of $\mathbf{ST}_T$. Hence we have the following observation: For any string T, the LZD tree for $\mathbf{LZD}_T$ can be superimposed on $\mathbf{ST}_T$, by possibly introducing some non-branching internal nodes. Due to this observation, we can compute $\mathbf{LZD}_T$ in $O(N)$ time and space in an *offline* manner for integer alphabets, using the offline algorithm of [2] which computes the LZ78 factorization of T from the suffix tree of T. In what follows, we show how to compute $\mathbf{LZD}_T$ in $O(N \log \sigma)$ time using $O(N)$ space in an *online* manner.

The basic strategy of our online algorithm is as follows. We first build the suffix tree of T incrementally, using Ukkonen's online suffix tree construction algorithm [14]. Then, for each $1 \leq i \leq m$, we find f_{i_1} and f_{i_2} on the suffix tree, and then mark the node which represents f_i (if there is no such node in the tree, then we create a new node and mark it).

We modify Ukkonen's algorithm as follows. As soon as we find the first occurrence of each character c at some position r in the string, we create a marked non-branching node v representing c, i.e., $\mathbf{str}(v) = c$. A new leaf for the suffix starting at position r is then created as a child of v. This permits us to superimpose the children of the root of the LZD tree onto the suffix tree.

We construct the suffix tree of $T[1..j]$ online, for increasing $j = 1, \ldots, N$. For each position $1 \leq j \leq N$, Ukkonen's algorithm maintains the following invariant: the longest suffix $T[s_j..j]$ of $T[1..j]$ that has an occurrence in $T[1..j-1]$. For convenience, when the longest suffix is the empty string ε, then let $s_j = j + 1$. Also, let $s_0 = 0$. We will use this suffix (and its location in the suffix tree) to determine the first and second elements of each LZD factor.

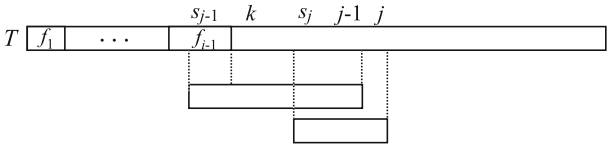

Fig. 2. $T[s_{j-1}..j-1]$ (resp. $T[s_j..j]$) is the longest suffix of $T[1..j-1]$ (resp. $T[1..j]$) that has an occurrence in $T[1..j-2]$ (resp. $T[1..j-1]$). We have computed $f_1, \ldots, f_{i-1}$ for the minimum i satisfying $s_{j-1} \leq |f_1 \cdots f_{i-1}| + 1 < s_j$.

Assume that we have constructed the suffix tree for $T[1..j]$ for some $1 \leq j < N$ such that $s_{j-1} < s_j$. Also, assume that we have computed $f_1, \ldots, f_{i-1}$ for the minimum integer i satisfying $s_{j-1} \leq |f_1 \cdots f_{i-1}| + 1 < s_j$ (see also Fig. 2). For any $s_{j-1} \leq k < s_j$, let P_k be the path spelling out $T[k..j-1]$ from the root. While we update the suffix tree of $T[1..j-1]$ to that of $T[1..j]$ by Ukkonen's algorithm, the ending position of path P_k in the tree can be found in amortized constant time for each k, in increasing order. Let $f_i, \ldots, f_{i'}$ be the consecutive

LZD factors such that i' is the minimum integer with $|f_1 \cdots f_{i'}| + 1 \geq s_j$. Since a node of the suffix tree is marked iff it represents one of the previous LZD factors or a single character, for any k ($s_{j-1} \leq k < s_j$) the lowest marked node v_k in the path P_k represents the longest prefix $T[k..k + \mathbf{depth}(v_k) - 1]$ of $T[k..N]$ which is also a previous LZD factor or a single character. This allows us to efficiently compute f_ℓ for each $\ell = i, \ldots, i' - 1$ in increasing order. As soon as we finish computing each f_ℓ, we maintain the suffix tree so that it contains a marked node which represents f_ℓ. Since we already know the location of the node which represents f_{ℓ_1}, we can find the ending position of the path spelling out $f_\ell = f_{\ell_1} f_{\ell_2}$ simply by traversing f_{ℓ_2} from the node representing f_{ℓ_1}. If f_ℓ is represented by an explicit node in the current tree, we mark the node. Otherwise, we insert a new marked node representing f_ℓ into the tree. Since $\sum_{\ell=i}^{i'-1} |f_{\ell_2}| < |f_i \cdots f_{i'-1}|$, this takes a total of $O(|f_i \cdots f_{i'-1}| \log \sigma)$ time for all $i \leq \ell < i'$.

In the sequel, we show how to compute the first element $f_{i'_1}$ of $f_{i'}$. If $s_j = j+1$ (i.e., j is the first occurrence of character $T[j]$ in $T[1..j]$), then $f_{i'_1} = T[j]$. After computing this, we mark the node representing $T[j]$. Otherwise, let z be the lowest marked node in the path from the root which spells out $T[|f_1 \ldots f_{i'-1}| + 1..j]$. By definition, it holds that $|f_1 \ldots f_{i'-1}| + \mathbf{depth}(z) \leq j$. If $|f_1 \ldots f_{i'-1}| + \mathbf{depth}(z) < j$, then $f_{i'_1}$ is computed in the same way as above, namely $f_{i'_1} = \mathbf{str}(z)$. If $|f_1 \ldots f_{i'-1}| + \mathbf{depth}(z) = j$, then we update the suffix tree of $T[1..j]$ to that of $T[1..j']$, where $j' > j$ is the minimum integer such that $s_j = s_{j'-1} \leq |f_1 \ldots f_{i'-1}| + 1 < s_{j'}$. Then, we can compute $f_{i'_1}$ in the same way as above, on the suffix tree for $T[1..j']$. The second element $f_{i'_2}$ can be computed analogously, and the node representing $f_{i'}$ can be found and marked in $O(|f_{i'}| \log \sigma)$ time. We repeat this procedure till we obtain all LZD factors for T (Fig. 3).

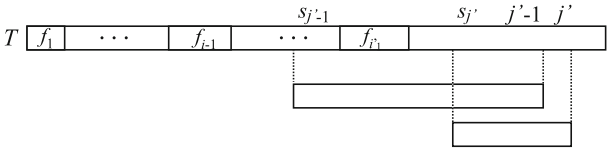

Fig. 3. When computing $f_{i'_1}$, if $|f_1 \ldots f_{i'-1}| + \mathbf{depth}(z) > j$, then we update the suffix tree of $T[1..j]$ to that of $T[1..j']$ with the minimum $j' > j$ such that $s_j = s_{j'-1} \leq |f_1 \ldots f_{i'-1}| + 1 < s_{j'}$. Then, $f_{i'_1}$ is represented by the lowest marked node in the path $P_{|f_1 \ldots f_{i'-1}|+1}$.

What remains is how to efficiently compute the lowest marked node in each path P_k. We use the following result:

Lemma 3 ([1,15]). *A semi-dynamic rooted tree can be maintained in linear space in its size so that the following operations are supported in amortized $O(1)$ time: (1) find the nearest marked ancestor of any node; (2) insert an unmarked node; (3) mark an unmarked node.*

By semi-dynamic we mean that insertions of new nodes to the tree are allowed, while deletions of existing nodes from the tree are not allowed. Since Ukkonen's algorithm does not delete any existing nodes, we can use the above lemma in our algorithm. If path P_k ends on an edge (i.e., if $T[k..j-1]$ is represented by an implicit node), then we can use the lowest explicit node in the path P_k to find the desired nearest marked ancestor.

After computing all LZD factors, we can discard the suffix tree. Ukkonen's algorithm constructs the suffix tree $\mathbf{ST}_T$ of string T in $O(N \log \sigma)$ time and $O(N)$ space. Since we can find all LZD factors in $O(\sum_{i=1}^{m} |f_i| \log \sigma) = O(N \log \sigma)$ time and $O(N)$ space, we obtain the following theorem:

Theorem 2 (Fast Online LZD Factorization). *Given a string T of length N, we can compute $\mathbf{LZD}_T = f_1, \ldots, f_m$ in $O(N \log \sigma)$ time and $O(N)$ space in an online manner, where σ is the alphabet size.*

4 LZD Factorization with Variable-to-Fixed Encoding

This section proposes an extension of LZD factorization of Sect. 3 to a variable-to-fixed encoding that runs in $O(N + \min(m, 2^L)(M + \min(m, M, 2^L) \log \sigma))$ time and $O(\min(2^L, m))$ space, where L is the fixed bit-length of code words representing factors, m is the number of factors, and M is the length of the longest factor. We call this variant the *LZDVF* factorization.

Since we are allowed to use only 2^L codes to represent the factors, we can store at most 2^L previous factors to compute new factors. A naïve solution would be to compute and store the first 2^L factors for the prefix $T[1..|f_1 \ldots f_{2^L}|]$, and then factorize the remaining suffix $T[|f_1 \ldots f_{2^L}| + 1..N]$ using the existing dictionary, without introducing new factors to it. We store these factors in a Patricia tree, and hence this algorithm uses $O(\min(2^L, m))$ space. Since there are at most $\min(m, M, 2^L) - 1$ branching nodes in the trie, this algorithm runs in $O(N + \min(m, 2^L)(M + \min(m, M, 2^L) \log \sigma))$ time. However, when the content of the remainder $T[|f_1 \ldots f_{2^L}| + 1..N]$ is significantly different from that of the prefix $T[1..|f_1 \ldots f_{2^L}|]$, then the naïve algorithm would decompose the remainder into many short factors, resulting in a poor compression ratio.

To overcome the above difficulties, our algorithms reuse limited encoding space by deleting some factors, and store new factors there. We propose two kinds of replacement strategies which we call LZDVF Count and LZDVF Pre respectively. The first one counts the number of factors appearing in the derivation trees of the factors that are currently stored in the dictionary, and deletes factors with low frequencies. This method is similar to the ones used in [8,12]. The second one deletes the least recently used factor in the dictionary in a similar way to [11] which uses an LRU strategy for LZ78 factorization.

In both strategies, there are at most 2^L entries in the dictionary and thus each factor is encoded by an L-bit integer. Since code words are reused as new factors are inserted and old factors are deleted from the dictionary, one may think that this introduces difficulties in decompression. However, since the procedure is deterministic, the change in assignment can be recreated during decompression, and thus will not cause problems.

4.1 Counter-Based Strategy

We define the derivation tree of each factor $f_i = f_{i_1} f_{i_2}$ recursively, as follows. The root of the tree is labeled with f_i, with two children such that the subtree rooted at the left child is the derivation tree of f_{i_1}, and the subtree rooted at the right child is the derivation tree of f_{i_2}. If f_{i_1} is a single character a, then its derivation tree consists only of the root labeled with a. The same applies to f_{i_2}. Let $\mathbf{vOcc}_i(f_j)$ denote the number of nodes in the derivation tree of f_i which are labeled with f_j. For all factors f_j that appear at least once in the derivation tree of f_i, we can compute $\mathbf{vOcc}_i(f_j)$ in a total of $O(|f_i|)$ time by simply traversing the derivation tree. Let $\mathbf{count}(f_j)$ be the sum of $\mathbf{vOcc}_q(f_j)$ for all factors f_q that are currently stored in the dictionary.

Assume that we have just computed a new factor $f_i = f_{i_1} f_{i_2}$. For each factor f_j with $\mathbf{vOcc}_i(f_j) > 0$, we first add $\mathbf{vOcc}_i(f_j)$ to $\mathbf{count}(f_j)$. If 2^L factors are already stored, then we do the following to delete factors from the dictionary. Depending on whether f_{i_1} and f_{i_2} are single characters or not, at least one (just f_i), and at most 3 (f_i and both f_{i_1}, f_{i_2}) new factors are introduced. For all factors f_h that are currently stored in the dictionary, we decrease $\mathbf{count}(f_h)$ one by one, until for some factor f_k, $\mathbf{count}(f_k) = 0$. We delete all such factors and repeat the procedure until enough factors have been deleted.

As the number of nodes in the derivation tree of each factor f_j is $O(|f_j|)$, the sum of counter values for all factors is $O(N)$. Hence, the total time required to increase and decrease the counter values is $O(N)$. Thus, the counter-based algorithm takes $O(N+\min(m, 2^L)(M+\min(m, M, 2^L)\log\sigma))$ time and $O(\min(2^L, m))$ space. When L can be seen as a constant, the algorithm runs in $O(N+M+\log\sigma) = O(N)$ time and uses $O(1)$ space.

4.2 Prefix-Based Strategy

Assume that we have computed the first i factors $f_1, \ldots, f_i$. In the prefix-based strategy, we consider a factor to be *used* at step i if it is a prefix of f_i. If $f_{h_1} (= f_i), f_{h_2}, \ldots, f_{h_k}$ are the sequence of all k factors in the dictionary which are prefixes of f_i in decreasing order of their lengths, then we consider that these factors are used in this chronological order. Hence, f_{h_k} will be the most recently used factor for step i. We use a doubly-linked list to maintain the factors, with the most recently used factor at the front and the least recently used factor at the back of the list. At each step i, we update the information for the factors $f_{h_1}, \ldots, f_{h_k}$. For any $1 \leq j \leq k$, if f_{h_j} is already in the list, we simply move it to the front of the list. Since the list is doubly linked, this can be done in $O(1)$ time. Otherwise, we simply insert a new element for f_{h_j} to the front of the list, and delete the LRU factor at the back of the list if the size of the list exceeded 2^L. This can also be done in $O(1)$ time.

The factors $f_{h_1}, \ldots, f_{h_k}$ can easily be found by maintaining the existing factors in a trie. Note that in each step of the algorithm, the LRU factor to be deleted is always a leaf of the trie since we have inserted the most recently used factors in decreasing order of their lengths. Hence, it takes $O(1)$ time to remove

the LRU factor from the trie. Overall, the prefix-based algorithm also takes $O(N + \min(m, 2^L)(M + \min(m, M, 2^L) \log \sigma))$ time and $O(\min(2^L, m))$ space, which are respectively $O(N)$ and $O(1)$ when L is a constant.

5 Computational Experiments

All computations were conducted on a Mac Xserve (Early 2009) (Mac OS X 10.6.8) with 2 x 2.93 GHz Quad Core Xeon processors and 24 GB Memory, but only running a single process/thread at once. Each core has L2 cache of 256 KB and L3 cache of 8 MB. The programs were compiled using LLVM C++ compiler (`clang++`) 3.4.2 with the `-Ofast` option for optimization.

We implemented the space efficient on-line LZD algorithm described in Sect. 3.1, and the algorithms LZDVF Count and Pre with variable-to-fixed encoding described in Sect. 4[2], and compared them with the state-of-the art of grammar compression algorithms OLCA [7] and FOLCA [9]. For LZD, the resulting grammar is first transformed to a Straight line program (SLP) by transforming the first rule $S \to f_1 \cdots f_m$; replacing consecutive factors with non-terminal variables iteratively until the number of non-terminal variables equals to 1, and then the SLP is encoded in the same way as [7]. The output of LZDVF is a sequence of pairs of fixed-length code words that describes each LZD factor.

We evaluated the compression ratio, compression and decompression speed[3] of each algorithm for data (non highly-repetitive[4] and highly-repetitive[5]) taken from the Pizza & Chili Corpus. The running times are measured in seconds, and includes the time reading from and writing to the disk. The disk and memory caches are purged before every run using the `purge` command. The average of three runs is reported. The results are shown in Fig. 4 (a)-(d). We can see that compared to LZ78, LZD improves the compression ratio for all cases, as well as compression/decompression times in almost all most cases. The compression ratio of LZD is roughly comparable to OLCA, but the compression time of LZD slightly outperforms that of OLCA for highly repetitive texts, though not for the non-highly repetitive texts.

We also evaluated the performance of our algorithms for large-scale highly repetitive data, using 10 GB of English Wikipedia edit history data[6] (See Fig. 4 (e) and (f)). In this experiment, we modified LZDVF Pre and Count so that they do not read the whole input text into memory, and to explicitly store the edge labels of the Patricia tree that represents the factors. This modification increases the required working space from $O(\min(2^L, m))$ to $O(\min(2^L, m)M)$, but allows us to process large-scale data which does not fit in main memory. We compared the modified version of LZDVF Pre and Count with Freq and Lossy

[2] Source codes are available at https://github.com/kg86/lzd.
[3] The number of characters the algorithm can process a second.
[4] http://pizzachili.dcc.uchile.cl/texts.html.
[5] http://pizzachili.dcc.uchile.cl/repcorpus.html.
[6] The first 10 GB of enwiki-20150112-pages-meta-history1.xml-p000000010p00000 2983.7z, downloaded from http://dumps.wikimedia.org/backup-index.html.

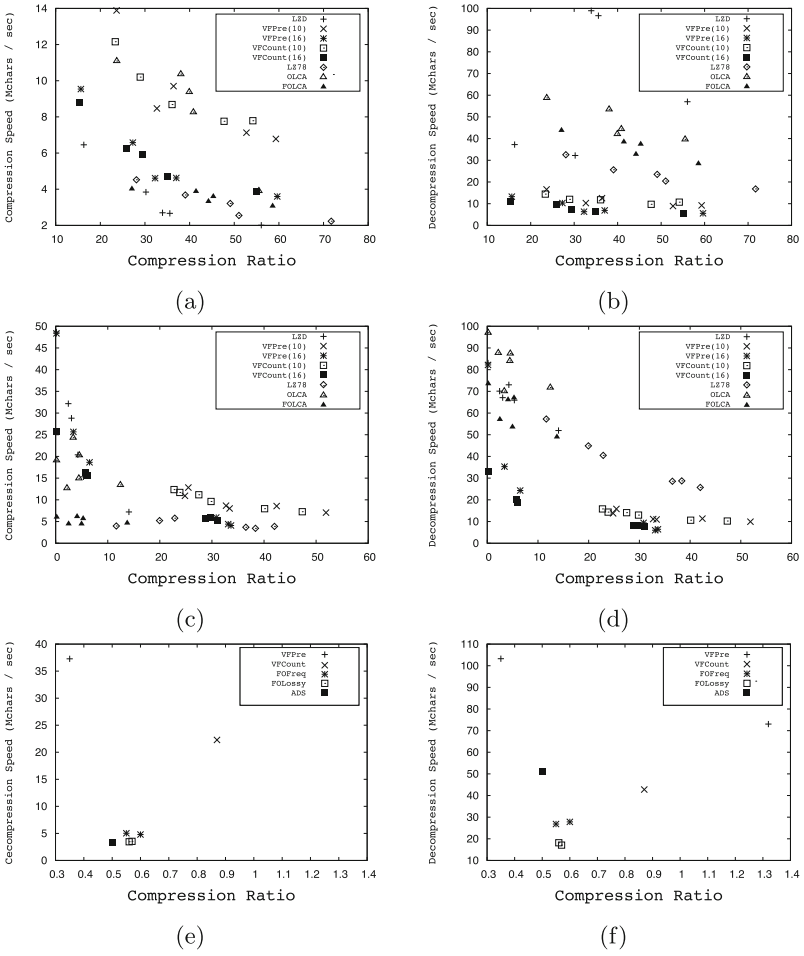

Fig. 4. Compression and decompression speed wrt. compression ratios. Results for LZD, LZDVF Pre and Count (VFPre and VFCount), OLCA [7] and FOLCA [9] on: (a), (b) non highly repetitive texts (DNA, English, Proteins, Sources, XML) of size 200 MB and (c), (d) highly repetitive texts (einstein.en, Escherichia_Coli, influenza, kernel, para, world_leaders). (e), (f): Results for LZDVF Pre and Count (VFPre and VFCount), Freq and Lossy FOLCA [8] (FOFreq and FOLossy), and ADS [12], which are grammar compression algorithms that do not store the whole input text in RAM, on 10 GB of English Wikipedia edit history. The parameters that determine the maximum number of non-terminal variables that VFPre, VFCount, FOFreq, ADS can store are varied between 2^{12}, 2^{14}, 2^{16} respectively. The block size parameter is varied 100 MB and 500 MB for ADS, and 100 MB, 500 MB, 1000 MB for FOLossy. Note that the points out of the frame are not plotted for visibility.

FOLCA [8], and ADS [12] which use constant space. In this experiment, LZDVF Pre with bit-size of 16 shows the best performance. Surprisingly, it reduces the compression time to about a seventh of that of FOLCA Freq, which is the fastest of the previous grammar compression algorithms applicable to such large-scale data, while achieving a better compression ratio.

Acknowledgements. We would like to thank Shirou Maruyama and Takuya Kida for providing source codes of their compression programs FOLCA and ADS.

References

1. Amir, A., Farach, M., Idury, R.M., Poutré, J.A.L., Schäffer, A.A.: Improved dynamic dictionary matching. Inf. Comput. **119**(2), 258–282 (1995)
2. Bannai, H., Inenaga, S., Takeda, M.: Efficient LZ78 factorization of grammar compressed text. In: Calderón-Benavides, L., González-Caro, C., Chávez, E., Ziviani, N. (eds.) SPIRE 2012. LNCS, vol. 7608, pp. 86–98. Springer, Heidelberg (2012)
3. Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundamenta Informaticae **111**(3), 313–337 (2011)
4. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Speeding up q-gram mining on grammar-based compressed texts. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 220–231. Springer, Heidelberg (2012)
5. Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: Unified compression-based acceleration of edit-distance computation. Algorithmica **65**(2), 339–353 (2013)
6. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: DCC 1999, 296–305 (1999)
7. Maruyama, S., Sakamoto, H., Takeda, M.: An online algorithm for lightweight grammar-based compression. Algorithms **5**(2), 214–235 (2012)
8. Maruyama, S., Tabei, Y.: Fully online grammar compression in constant space. In: DCC 2014, pp. 173–182 (2014)
9. Maruyama, S., Tabei, Y., Sakamoto, H., Sadakane, K.: Fully-online grammar compression. In: Kurland, O., Lewenstein, M., Porat, E. (eds.) SPIRE 2013. LNCS, vol. 8214, pp. 218–229. Springer, Heidelberg (2013)
10. Nevill-Manning, C.G., Witten, I.H., Maulsby, D.L.: Compression by induction of hierarchical grammars. In: DCC 1994. pp. 244–253 (1994)
11. Peter, T.: A modified LZW data compression scheme. In: Australian Computer Science Communications, pp. 262–272 (1987)
12. Sekine, K., Sasakawa, H., Yoshida, S., Kida, T.: Adaptive dictionary sharing method for re-pair algorithm. In: DCC 2014, p. 425 (2014)
13. Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding up pattern matching by text compression. In: Bongiovanni, G., Petreschi, R., Gambosi, G. (eds.) CIAC 2000. LNCS, vol. 1767, pp. 306–315. Springer, Heidelberg (2000)
14. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
15. Westbrook, J.: Fast incremental planarity testing. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 342–353. Springer, Heidelberg (1992)
16. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)