Fully Online Grammar Compression in Constant Space

Shirou Maruyama∗ and Yasuo Tabei†

∗Preferred Infrastructure, Inc. maruyama@preferred.jp
†PRESTO, Japan Science and Technology Agency tabei.y.a@m.titech.ac.jp

Abstract
We present novel variants of fully online LCA (FOLCA), a fully online grammar compression that builds a straight line program (SLP) and directly encodes it into a succinct representation in an online manner. FOLCA enables a direct encoding of an SLP into a succinct representation that is asymptotically equivalent to an information theoretic lower bound for representing an SLP (Maruyama et al., SPIRE’13). The compression of FOLCA takes linear time proportional to the length of an input text and its working space depends only on the size of the SLP, which enables us to apply FOLCA to large-scale repetitive texts. Recent repetitive texts, however, include some noise. For example, current sequencing technology has significant error rates, which embeds noise into genome sequences. For such noisy repetitive texts, FOLCA working in the SLP size consumes a large amount of memory. We present two variants of FOLCA working in constant space by leveraging the idea behind stream mining techniques. Experiments using 100 human genomes corresponding to about 300GB from the 1000 human genomes project revealed the applicability of our method to large-scale, noisy repetitive texts.

1 Introduction

Large-scale, highly repetitive text collections are becoming more and more ubiquitous. Examples are sequences of individual human genomes, source code in repositories and version controlled documents. In particular, the 1000 human genomes project stores more than 2,000 human genomes, which makes it possible for us to analyze human genetic variations [1]. Thus, we face a problem storing and processing a vast number of individual genome sequences of the same species, perhaps millions of genomes in the near future. There is therefore a strong demand for developing powerful methods to compress and process such text data on a large scale.

Grammar compression is effective for compressing and processing repetitive texts. There are two types of problems in the field: (i) building a small context-free grammar (CFG) that generates a single text and (ii) representing the obtained CFG as compactly as possible. The method also reveals a processing ability for repetitive texts, e.g., pattern matching [2], pattern mining [3] and edit distance computation [4]. As far as we know, existing methods use two-step approaches: they first build a CFG from an input text and then encode it into a compact representation. Although these methods achieve a high compression ratio, they require a large working space consumed by building the CFG and its encoding. Even worse, they are not applicable to streaming texts because of the indirectness of these steps.

This work was supported by JSPS KAKENHI(24700140).
Table 1: Comparison with existing algorithms. Here, $N$ is the length of the input string, $\sigma$ is the alphabet size, $n$ is the number of generated rules, and $\alpha$ is a parameter between 0 and 1 (the load factor of hash tables). The expected time complexities are due to the use of a hash function.

<table>
<thead>
<tr>
<th>Compression time</th>
<th>Working space (bits)</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(N/\alpha)$ expected</td>
<td>$(3 + \alpha)n \lg(n + \sigma)$</td>
<td>[7]</td>
</tr>
<tr>
<td>$O(N/\alpha)$ expected</td>
<td>$(\frac{11}{4} + \alpha)n \lg(n + \sigma)$</td>
<td>[9]</td>
</tr>
<tr>
<td>$O(N \lg n)$</td>
<td>$2n \lg n(1 + o(1)) + 2n \lg \rho$ ($\rho \leq 2\sqrt{n}$)</td>
<td>[6]</td>
</tr>
<tr>
<td>$O(\frac{N \lg n}{\alpha \lg \lg n})$ expected</td>
<td>$(1 + \alpha)n \lg(n + \sigma) + n(3 + \lg(\alpha n))$</td>
<td>[5]</td>
</tr>
<tr>
<td>$O(N/\alpha)$ expected</td>
<td>constant</td>
<td>this study</td>
</tr>
</tbody>
</table>

Maruyama et al. [5] overcame the efficiency problem by introducing a fully online grammar compression called FOLCA that builds a straight line program (SLP), a normal form of a CFG, and directly encodes it into a succinct representation in an online manner. To achieve a small working space, they also presented a space-efficient implementation of reverse dictionary, a crucial data structure for checking whether or not a production rule in an SLP already exists in execution. FOLCA consumes a small working space that depends only on the size of the SLP and not on the size of the input text because of the directness of building the SLP and its encoding in an online manner. In addition, the size of the succinct representation of an SLP of $n$ variables by FOLCA is $\lg n + 2n + o(n)$ bits, which is asymptotically equivalent to the information theoretic lower bound of $\lg n + n + o(n)$ bits presented in [6]. The compression time and approximation ratio are $O(N)$ and $O(\lg^2 N)$, respectively, for the length $N$ of an input text, which are the same as those of the previous version of FOLCA [7]. Such nice properties of FOLCA enable the compression of a large-scale repetitive text in a small working space depending on the SLP size.

Recent repetitive texts, however, include some noise. For example, current sequencing technology has significant error rates, which embeds noise into genome sequences. Actually, there is a 9% difference on average among sequences stored in the 1000 human genomes project, although it is said that the genetic difference between individual human genomes is less than 0.01%. Since FOLCA works in a space depending on the SLP size, applying it to such noisy repetitive texts requires a large working space. One can solve the problem by dividing the input text into several blocks and applying compressors, e.g., LZ77 and RePair, into each block, but the compression ratio is ruined as long-range repetitions are not captured [8]. Since the amount of large-scale repetitive text data including noise is ever increasing, developing a scalable grammar compression using a constant working space remains a challenge.

In this paper, we present novel variants of FOLCA that are fully online grammar compressions working in constant space for large-scale, noisy repetitive texts. Our variants output a succinct representation of an SLP to a secondary storage device and keep a hash table for a reverse dictionary in a main memory; this accounts for most of the memory consumed by the hash table for compression. To conserve working space, our variants keep only frequent production rules in the hash table by leveraging the idea behind stream mining techniques [10–12]. Since these frequent production rules can represent as many as possible of the same digrams, they are expected to
help achieve a high compression ratio. In addition, we also present a decompression algorithm working in constant space from our succinct representation of an SLP to efficiently recover a large text. Our results and those of existing algorithms are summarized in Table 1. For the space limitation, we omit several details of our method in this proceedings version. See the full version of the paper.

Experiments were performed on compressing and decompressing 100 human genomes from the 1000 human genomes project. The results show the applicability of our method to large-scale, noisy repetitive texts.

2 Preliminaries

2.1 Basic notations

Let $\Sigma$ be a finite alphabet for the symbols forming input texts throughout this paper. All elements in $\Sigma$ are totally ordered. $\Sigma^*$ denotes the set of all strings over $\Sigma$, and $\Sigma^i$ denotes the set of all strings of length $i$. The length of $w \in \Sigma^*$ is denoted by $|w|$, and the cardinality of a set $C$ is similarly denoted by $|C|$. $\mathcal{X}$ is a recursively enumerable set of variables with $\Sigma \cap \mathcal{X} = \emptyset$. A sequence of symbols from $\Sigma \cup \mathcal{X}$ is also called a string, and an ordered pair of symbols from $\Sigma \cup \mathcal{X}$ is called a digram. Strings $x$ and $z$ are said to be the prefix and suffix of the string $w = xyz$, respectively, and $x, y, z$ are called substrings of $w$. The $i$-th symbol of $w$ is denoted by $w[i]$ ($1 \leq i \leq |w|$).

For integers $i, j$ with $1 \leq i \leq j \leq |w|$, the substring of $w$ from $w[i]$ to $w[j]$ is denoted by $w[i, j]$. $\lg n$ stands for $\log_2 n$. Let $N$ be the length of an input text, which can be variable in an online setting.

2.2 Grammar compression

A CFG is a quadruple $G = (\Sigma, V, D, X_s)$ where $V$ is a finite subset of $\mathcal{X}$, $D$ is a finite subset of $V \times (V \cup \Sigma)^*$ of production rules, and $X_s \in V$ represents the start symbol. $D$ is also called a phrase dictionary. Variables in $V$ are called nonterminals. We assume a total order over $\Sigma \cup V$. The set of strings in $\Sigma^*$ derived from $X_s$ by $G$ is denoted by $L(G)$. A CFG $G$ is called admissible if for any $X \in \mathcal{X}$ there is exactly one production rule $X \to \gamma \in D$ and $|L(G)| = 1$. An admissible $G$ deriving a text $S$ is called a grammar compression of $S$. The size of $G$ is the total of the lengths of strings on the right hand sides of all production rules; it is denoted by $|G|$. The problem of grammar compression is formalized as follows: Given a string $w \in \Sigma^*$, compute a small, admissible $G$ that derives only $w$. In the following, we assume the case $|\gamma| = 2$ for any production rule $X \to \gamma$. This assumption is reasonable because any grammar compression with $n$ variables can be transformed into such a restricted CFG with at most $2n$ variables.

The parse tree of $G$ is represented as a rooted ordered binary tree such that internal nodes are labeled by variables in $V$ and the yields, i.e., the sequence of labels of leaves that are equal to $S$. In a parse tree, any internal node $Z \in V$ corresponds to the production rule $Z \to XY$, and it has a left child labeled $X$ and a right child.

\footnote{http://arxiv.org/abs/1401.5143}
Let $height(X_i)$ be the height of the subtree having the root $X_i$ in the parse tree. We assume an SLP for the CFG as follows: An SLP is a grammar compression over $\Sigma \cup V$ whose production rules are formed by $X_k \rightarrow X_i X_j$, where $X_i, X_j \in \Sigma \cup V$ and $1 \leq i, j < k \leq |V| + |\Sigma|$.

2.3 Reverse dictionary

A reverse dictionary $D^{-1} : (\Sigma \cup X')^2 \rightarrow X'$ is a mapping from a given digram to a nonterminal symbol. $D^{-1}$ returns a nonterminal $Z$ associated with a digram $XY$ if $Z \rightarrow XY \in D$; otherwise, it creates a new nonterminal symbol $Z' \notin V$ and returns $Z'$. We can implement a reverse dictionary using a chaining hash table that has a load factor $\alpha$. The hash table has $\alpha n$ entries and each entry stores a list of a triple $(X_k, X_i, X_j)$ for a production rule $X_k \rightarrow X_i X_j$. For the rule $X_k \rightarrow X_i X_j$, the hash value is computed from $X_i$ and $X_j$. Then, the list corresponding to the hash value is scanned to search for $X_k$. Thus, the expected access time is $O(1/\alpha)$. The space is $\alpha n \log (n + \sigma)$ bits for the hash table and $3n \log (n + \sigma)$ bits for the lists. Therefore, the total size is $n(3 + \alpha) \log (n + \sigma)$ bits.

3 FOLCA in Compressed Space

FOLCA builds a post-order SLP (POSLP) as a special form of an SLP that can be transformed into a post-order partial parse tree (POPPT): a partial parse tree whose internal nodes are post-ordered. FOLCA directly encodes a POSLP into a succinct representation of a POPPT. In the original FOLCA, the succinct representation of a POPPT is kept in a main memory and used as the reverse dictionary in combination with a hash table. Instead, we modified FOLCA as it outputs the succinct representation of a POPPT into a secondary storage device, and we implemented a reverse dictionary using a chaining hash table kept in a main memory. Thus, the hash table is a crucial data structure for large-scale applications of FOLCA, because it accounts for most of the working space. In the next section, we present two space-efficient versions of FOLCA created by reducing the space of the hash table.
3.1 Post-order SLP

A partial parse tree defined by Rytter [13] is a binary tree formed by traversing a parse tree in a depth-first manner and pruning out all descendants under every node of nonterminal symbols appearing no less than twice. A POPPT and POSLP are defined as follows.

**Definition 1 (POSLP and POPPT [5])** A post-order partial parse tree (POPPT) is a partial parse tree whose internal nodes have post-order variables. A post-order SLP (POSLP) is an SLP whose partial parse tree is a POPPT.

Note that the number of nodes in the POPPT is $2^n + 1$ for a POSLP of $n$ variables, because the numbers of internal nodes and leaves are $n$ and $n + 1$, respectively. Examples of a POSLP and POPPT are shown in Figure 1-(i) and -(iii). In this example, all the descendants under every node having the second $X_1$ and $X_2$ in the parse tree (ii) are pruned out. The resulting POPPT (iii) has internal nodes consisting of post-order variables.

A major advantage of the POSLP is that we can directly encode the corresponding POPPT into a succinct representation of a bit string $B$ and a label sequence $L$. A bit string $B$ representing a POPPT is built by traversing a POPPT in a depth-first manner and placing '0' if a node is a leaf and '1' otherwise in post-order. The last bit '1' in $B$ represents a virtual node and is placed for fast tree operations without computing boundary conditions. Thus, a POPPT of a POSLP of $n$ variables consists of $n$ internal nodes, $n + 1$ leaves and a virtual node. Label sequence $L$ keeps symbols in the leaves of a POPPT. The length of $L$ is $n + 1$ and its space is $(n + 1)[\log(n + \sigma)]$ bits. Thus, the POPPT size of a POSLP of $n$ variables is $(n + 1)[\log(n + \sigma)] + 2n + 2$ bits.

3.2 FOLCA

The basic idea of FOLCA is to (i) start from symbols of an input text, (ii) replace as many as possible of the same pairs of symbols in common substrings by the same nonterminal symbols, and (iii) iterate this process in a bottom-up manner until it generates a complete POSLP. The POSLP is built in an online fashion.

FOLCA consists of three functions, FOLCA, ProcessSymbol and Update. The main function is FOLCA, which reads new characters from an input text and gives them to the function ProcessSymbol one by one. The function ProcessSymbol builds local subtrees in a POPPT in an online manner. In ProcessSymbol, the function Update returns a nonterminal symbol replacing a given digram. If the digram is a novel digram in execution, it returns a new nonterminal symbol, and otherwise, it returns the existing nonterminal symbol replacing the digram.

The number of variables of a POSLP is $O(n_\ast \log^2 N)$ for the optimal grammar size $n_\ast$ and the length $N$ of an input text. The partial parse tree built by FOLCA is a POPPT. See [5] for the proof.

FOLCA builds nodes of the POPPT and directly encodes them into a succinct representation in an online manner. These nodes are output to a secondary storage.
device immediately after they have been built. Most of the working space is consumed by a hash table implementing a reverse dictionary (Figure 1-(v)). Therefore, the working space of FOLCA is at most \( n(3 + \alpha) \log(n + \sigma) \) bits, while the compression time is \( O(N/\alpha) \). In the next section, we introduce techniques for reducing the space of a hash table.

4 FOLCA in Constant Space

We present novel variants of FOLCA working in constant space. We reduce the size of the hash table implementing a reverse dictionary in FOLCA by leveraging the idea behind data mining techniques proposed in [10–12] for finding frequent items in data streams. Our methods are counter-based algorithms that compute the frequencies of production rules of a dictionary in a streaming text and remove infrequent ones from the dictionary at a point in the algorithm. The frequent production rules in the resulting dictionary are expected to greatly contribute to string compression. We present two algorithms called frequency counting and lossy counting that can be used instead of the function \( \text{Update} \) in FOLCA to update a phrase dictionary. We shall refer to FOLCAs using frequency counting and lossy counting as \( \text{FREQ}_{-}\text{FOLCA} \) and \( \text{LOSSY}_{-}\text{FOLCA} \), respectively.

4.1 FREQ_{-}FOLCA

The basic idea of frequency counting is to use a phrase dictionary of fixed-size \( k \) and keep only frequent production rules in the phrase dictionary. Such frequently appearing production rules are expected to replace as many as possible of the same
Algorithm 2 Lossy counting. \(\ell\): parameter, \(N\): length of an input string at a point in time, \(X, Y \in (V \cup \Sigma)\).

1: Initialize \(\Delta := 0\)
2: function LossyCounting(X,Y)
3: \(Z := D^{-1}(X,Y)\)
4: if \(Z \rightarrow XY \in D\) then
5: \(c(Z) := c(Z) + 1\)
6: else
7: \(D := D \cup \{Z \rightarrow XY\}\)
8: \(c(Z) := \Delta + 1\)
9: end if
10: if \(\left\lfloor \frac{N}{\ell} \right\rfloor \neq \Delta\) then
11: \(\Delta := \frac{N}{\ell}\)
12: for all \(Z' \rightarrow X'Y' \in D\) do
13: if \(c(Z') < \Delta\) then
14: \(D := D \setminus \{Z' \rightarrow X'Y'\}\)
15: end if
16: end for
17: end if
18: return \(Z\)
19: end function

digrams by the same nonterminal symbols. Algorithm 1 shows an algorithm for frequency counting. We use a frequency counter \(c\) for each variable in a dictionary to compute its frequency. \textsc{FREQ\_FOLCA} builds a production rule \(Z \rightarrow XY\) for a novel digram \(XY\) and pushes it into a dictionary \(D\) where the frequency counter \(c(Z)\) for the left variable \(Z\) is initialized to \(1\). \(c(Z)\) is incremented by one each time a digram \(XY\) for \(Z \rightarrow XY\) appears. If the dictionary size reaches \(k\), \textsc{FREQ\_FOLCA} removes the bottom \(\epsilon\) percent of infrequent production rules for a vacancy rate \(\epsilon\).

\textsc{FREQ\_FOLCA} generates one succinct POPPT each time \textsc{FrequencyCounting} is called for an efficient decompression. Thus, each succinct POPPT represents a subtext of an input text. \textsc{FREQ\_FOLCA} recovers the original text for every succinct POPPT, which enables decompression using the same working space for compression.

When a reverse dictionary is implemented using a chaining hash table that has a load factor \(\alpha\), the working space of \textsc{FREQ\_FOLCA} is \(k(\alpha + 3)\log(k + \sigma)\) bits, while the compression time is expected to be \(O(N/\alpha)\).

4.2 LOSSY\_FOLCA

The basic idea of lossy counting is to divide an input string into intervals of fixed length, and keep production rules in the next successive intervals according to the number of appearances of these production rules. Thus, if a production rule appears no less than the total number of intervals, it is kept until the end of the execution.

Algorithm 2 shows a lossy counting algorithm. An input string is divided into \(\ell\) intervals. Thus, the length of each interval is \(N/\ell\). We use a counter \(c\) for each variable for counting the number of appearances of each variable \(Z\). For a variable \(Z\) appearing for the first time, \(c(Z)\) is initialized to \(N/\ell + 1\), which means that a production rule \(Z\) is kept in a dictionary for at least the next interval. If an existing
variable $Z$ in a dictionary appears $r$ times, $c(Z)$ is incremented by $r$, resulting in $Z$ being kept in a dictionary for the next $r$ successive intervals. LOSSY_FOLCA generates one succinct POPPT each time LOSSY_COUNTING is called for an efficient decompression just like FREQ_FOLCA.

The working space of LOSSY_FOLCA is at most $2\ell(\alpha+3)\log(2\ell+\sigma)$ bits as a result of implementing a reverse dictionary as a chaining hash table, while the compression time is expected to be $O(N/\alpha)$.

We also implemented another variant of FOLCA as a baseline by fixing $\Delta = \infty$ in Algorithm 2, which corresponds to dividing an input text into several blocks and applying FOLCA to each block. We shall call this variant of FOLCA BLOCK_FOLCA. Comparing FREQ_FOLCA and LOSSY_FOLCA with BLOCK_FOLCA reveals that BLOCK_FOLCA using such a standard strategy for compressing large-scale texts cannot capture long-range repetitions, which is presented in Section 6.

5 Decompression in Constant Space

We recover the original text from multiple succinct POPPTs that FREQ_FOLCA and LOSSY_FOLCA output into a secondary storage device. Our succinct representation of POPPTs consists of a bit string $B$ and a label sequence $L$ presented in Figure 1-(iv). We build a phrase dictionary $D$ by simulating a depth-first traversal which we can perform by gradually reading a bit from string $B$ and a label corresponding to a leaf label from $L$ from the beginning. Since a substring corresponding to one POPPT in $B$ has the same numbers of $'0'$ and $'1'$, respectively, we can detect each substring corresponding to a POPPT in $B$ by counting the number of $'0'$ and $'1'$. When we reach a position of $B$ corresponding to one POPPT, we recover the original string corresponding to a substring in $B$ from an obtained phrase dictionary $D$.

The working spaces for decompression depend on the compression method. They are the same for FREQ_FOLCA; likewise, they are the same for LOSSY_FOLCA. The decompression time is $O(N/\alpha)$. 

Figure 2: Working space for the length of text in compression (left) and decompression (right).
Table 2: Compression ratio (CR), compression time (CT) in seconds (s), decompression time (DT) in seconds and maximum working space (WS) in megabytes (MB) for each method on 100 genome sequences.

<table>
<thead>
<tr>
<th>Method</th>
<th>CR</th>
<th>CT (s)</th>
<th>DT (s)</th>
<th>WS (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQ_FOLCA (k=1000MB)</td>
<td>31.39</td>
<td>86,098</td>
<td>18,658</td>
<td>38,048</td>
</tr>
<tr>
<td>FREQ_FOLCA (k=2000MB)</td>
<td>19.71</td>
<td>93,823</td>
<td>19,442</td>
<td>76,096</td>
</tr>
<tr>
<td>LOSSY_FOLCA (ℓ=5000MB)</td>
<td>20.07</td>
<td>87,548</td>
<td>22,103</td>
<td>36,246</td>
</tr>
<tr>
<td>LOSSY_FOLCA (ℓ=10000MB)</td>
<td>17.45</td>
<td>87,446</td>
<td>20,378</td>
<td>56,878</td>
</tr>
<tr>
<td>BLOCK_FOLCA (ℓ=5000MB)</td>
<td>31.85</td>
<td>88,501</td>
<td>–</td>
<td>23,276</td>
</tr>
<tr>
<td>BLOCK_FOLCA (ℓ=10000MB)</td>
<td>25.91</td>
<td>92,007</td>
<td>–</td>
<td>34,665</td>
</tr>
</tbody>
</table>

6 Experiments

6.1 Setup

We evaluated the performances of FREQ_FOLCA and LOSSY_FOLCA by comparing them with FOLCA, BLOCK_FOLCA, and LZMA on one core of a quad-core Intel Xeon CPU E5-2680 (2.8GHz). We implemented FREQ_FOLCA, LOSSY_FOLCA, BLOCK_FOLCA, and FOLCA in C++. LZMA is a general string compressor effective for repetitive texts, and we used its software downloadable from http://sourceforge.jp/projects/sfnet_p7zip/. We also used 100 human genomes corresponding to 306GB downloadable from the 1000 human genomes project. We tried \( k = \{1000\text{MB}, 2000\text{MB}\} \) and \( \epsilon = 0.3 \) for FREQ_FOLCA and \( \ell = \{5000\text{MB}, 10000\text{MB}\} \) for LOSSY_FOLCA.

6.2 Results

Figure 2 shows the working space of each method for various lengths of a genome sequence in compression and decompression. FOLCA took more than 100GB of working space for a genome sequence of 52GB, which shows that FOLCA, whose working space depends on the POSLP size, was not applicable to large-scale, noisy repetitive texts. LZMA did not finish within 5 days, which shows that LZMA is also not applicable to large-scale genome sequences. For both FREQ_FOLCA and LOSSY_FOLCA, the working spaces for compression and decompression were almost the same, despite parameter values being different. The working space of FREQ_FOLCA remained constant for compression and decompression: 36GB and 76GB for \( k = 1000\text{MB} \) and \( k = 20000\text{MB} \), respectively. The working space of LOSSY_FOLCA fluctuated, and the maximum values for \( \ell = 5000\text{MB} \) and \( \ell = 10000\text{MB} \) were 36GB and 56GB, which did not depend on the text length and SLP size.

Table 2 shows the compression ratio, maximum working space for compression and decompression, and compression/decompression time for 100 human genomes. A tradeoff between compression time and working space was observed for each method. The compression ratio for a larger working space was better for each method. The compression ratio of LOSSY_FOLCA was better than that of BLOCK_FOLCA for
the same parameter value $\ell$, which showed that the strategy of LOSSY_FOLCA for removing infrequent production rules was more effective than that of BLOCK_FOLCA. While LOSSY_FOLCA using a small working space achieved a higher compression ratio than FREQ_FOLCA, one can use FREQ_FOLCA when one wants to conserve working space. Compression using LOSSY_FOLCA and FREQ_FOLCA finished within about 24 hours. These results demonstrate the applicability of LOSSY_FOLCA and FREQ_FOLCA to large-scale, noisy repetitive texts.

7 Conclusion

We have presented fully-online grammar compressions working in constant space. Experimental results using 100 human genomes from the 1,000 human genome project demonstrate the applicability of our method to large-scale, noisy repetitive texts. Extensions are to develop various text processing methods applicable to large-scale repetitive texts. This would be beneficial to users for compressing and processing repetitive texts.

8 References