A Space-Optimal Grammar Compression*

Yoshimasa Takabatake[1], Tomohiro I[2], and Hiroshi Sakamoto[3]

1 **Kyushu Institute of Technology, Fukuoka, Japan**
`takabatake@ai.kyutech.ac.jp`
2 **Kyushu Institute of Technology, Fukuoka, Japan**
`tomohiro@ai.kyutech.ac.jp`
3 **Kyushu Institute of Technology, Fukuoka, Japan**
`hiroshi@ai.kyutech.ac.jp`

Abstract

A grammar compression is a context-free grammar (CFG) deriving a single string deterministically. For an input string of length N over an alphabet of size σ, the smallest CFG is $O(\lg N)$-approximable in the offline setting and $O(\lg N \lg^* N)$-approximable in the online setting. In addition, an information-theoretic lower bound for representing a CFG in Chomsky normal form of n variables is $\lg(n!/n^\sigma) + n + o(n)$ bits. Although there is an online grammar compression algorithm that directly computes the succinct encoding of its output CFG with $O(\lg N \lg^* N)$ approximation guarantee, the problem of optimizing its working space has remained open. We propose a fully-online algorithm that requires the fewest bits of working space asymptotically equal to the lower bound in $O(N \lg \lg n)$ compression time. In addition we propose several techniques to boost grammar compression and show their efficiency by computational experiments.

1998 ACM Subject Classification E.4 Coding and Information Theory

Keywords and phrases Grammar compression, fully-online algorithm, succinct data structure

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.67

1 Introduction

1.1 Motivation

Data never ceases to grow. Especially, we have witnessed so-called *highly-repetitive* text collections are rapidly increasing. Typical examples are genome sequences collected from similar species, version controlled documents and source codes in repositories. As such datasets are highly-compressible in nature, employing the power of data compression is the right way to process and analyze them. In order to catch up the speed of data increase, there is a strong demand for *fully online* and *really scalable* compression methods.

In this paper, we focus on the framework of grammar compression, in which a string is compressed into a context-free grammar (CFG) that derives the string deterministically [23]. In the last decade, grammar compression has been extensively studied from both theoretical and practical points of view: While it is mathematically clean, it can model many practical compressors such as LZ78 [48], LZW [47], LZD [13], repair [22], sequitor [33], and so on. Furthermore, there are wide varieties of algorithms working on grammar compressed strings, e.g., self-indexes [3, 9, 21, 25, 34, 38, 45, 46], pattern matching [10, 17], pattern mining [12, 8], machine learning [41], edit-distance computation [14, 43], and regularities detection [29, 15].

* This work was supported by JST CREST (Grant Number JPMJCR1402), and KAKENHI (Grant Numbers 17H01791 and 16K16009).

Table 1 Improvement of FOLCA: the fully-online grammar compression. Here N is the length of the input string received so far, σ and n are the numbers of alphabet symbols and generated variables, respectively, and $\frac{1}{\alpha} \geq 1$ is the load factor of the hash table.[2] For any input string, these algorithms construct the same SLP, which has $O(\lg N \lg^* N)$ approximation guarantee.

algorithm	compression time	working space (bits)
FOLCA ([28])	$O(\frac{N \lg n}{\alpha \lg \lg n})$ expected	$(1+\alpha)n \lg(n+\sigma) + n(3 + \lg(\alpha n)) + o(n)$
SOLCA (ours)	$O(N \lg \lg n)$ expected	$n \lg(n+\sigma) + o(n \lg(n+\sigma))$

Note that in order to take full advantage of these applications, a text should be compressed globally, that is, typical workarounds to memory limitation such as setting window-size or reusing variables (by forgetting previous ones) are prohibitive. This further motivates us to design really scalable grammar compression methods that can compress huge texts.

The primary goal of grammar compression is to build a small CFG that derives an input string only. The problem to build the smallest grammar is known to be NP-hard, but approximable within a reasonable ratio, e.g., $O(\lg N)$-approximable in the offline setting [23] and $O(\lg N \lg^* N)$-approximable[1] in the online setting [28], where N is input size and $\lg^*$ is the iterative logarithms.

On the other hand, to get a scalable grammar compression we have to seriously consider reducing the working space to fit into RAM. First of all, the algorithm should work in space comparable to the output CFG size. This has a great impact especially when we deal with highly-repetitive texts because output CFG size grows much slower than input size. We are aware of several work (including other compression scheme than grammar compression) addressing this [28, 13, 7, 20, 36, 35], but very few care about a constant factor hidden in big-O notation. More extremely and ideally, the output CFG should be encoded succinctly in an online fashion, and the algorithm should work in "succinct space", i.e., the encoded size plus lower order terms. To the best of our knowledge, fully-online LCA (FOLCA) [28] (and its variants) is the only existing algorithm addressing this problem. Whereas FOLCA achieved a significant improvement in memory consumption, there is still a gap between the memory consumption and its theoretical lower bound because FOLCA requires extra space for a hash table other than the succinct encoding of the CFG. Therefore the problem of optimizing the working space of FOLCA has been a challenging open problem.

In this paper, we tackle the above mentioned problem, resulting in the first space-optimal fully-online grammar compression. In doing so, we propose a novel succinct encoding that allows us to simulate the hash table of FOLCA in a dynamic environment. We further introduce two techniques to speed up compression. We call this improved algorithm *Space-Optimal FOLCA (SOLCA)*. See Table 1 for the improved time and space complexities. Experimental results show that both working space and running time are significantly improved from original FOLCA. We also compare our algorithm with other state-of-the-arts, and see that ours outperforms others in memory consumption, while the compression time is four to seven times slower than the fastest opponent.

[1] The authors in [28] only claimed $O(\lg^2 N)$ approximation, but it can be improved to $O(\lg N \lg^* N)$ adopting edit sensitive parsing (ESP) technique [4], which was pointed out in [40]. Naively the use of ESP adds $\lg^* N$ factor to computation time, but it can be eliminated by a neat trick of table lookup (e.g., see Theorem 6 of [8]). In practice, we have observed that the use of ESP does not improve the compression ratio much (or often even worsens), so our implementation still uses the algorithm with $O(\lg^2 N)$ approximation guarantee.

[2] In the previous papers, the inverse of the load factor is mistakenly referred to as the load factor. Here we fix the misuse.

1.2 Our Contribution in More Details

In the framework of grammar compression, an algorithm must refer to two data structures, the *dictionary* D (a set of production rules) and the *reverse dictionary* D^{-1}. Considering any symbol Z_i to be identical to integer i, D is regarded as an array such that $D[i]$ stores the phrase β if the production rule $Z_i \to \beta$ exists. Without loss of generality, we can assume that G is a *straight-line program (SLP)* [19] such that any β is a *bigram*, i.e., a pair of symbols (each symbol is a variable or an alphabet symbol). It follows that a naive representation of D occupies $2n \lg(n+\sigma)$ bits for n variables and σ alphabet symbols. Because an information-theoretic lower bound of SLP is $\lg((n+\sigma)!/n^{\sigma}) + 2(n+\sigma) + o(n)$ bits [42], the naive representation is highly redundant. Fully-online LCA (FOLCA) [28] is the first fully-online algorithm that directly outputs an encoded $e(D)$ whose size is asymptotically equal to the size of the optimal one.

On the other hand, given a phrase β, D^{-1} is required to return Z if $Z \to \beta$ exists. Using D^{-1}, a grammar compression algorithm can remember the existing name Z associated with β, i.e., we can avoid generating a useless $Z' \to \beta$ for the same β. In previous compression algorithms [26, 44, 42, 28], the reverse dictionary was simulated by a hash table whose size is comparable to the size of $e(D)$. This is the reason that the space optimization problem has remained open.

To solve this problem, we introduce a novel mechanism that allows FOLCA to directly compute D^{-1} by $e(D)$ with an auxiliary data structure in a dynamic environment. We develop a very simple data structure satisfying those requirements, and then we improve the working space of FOLCA. Note that the new data structure itself is independent from FOLCA/SOLCA, and applicable to any SLP for which fast access to both D and D^{-1} is required. Thus, it can be a new standard of succinct SLP encoding for such purposes.

FOLCA and SOLCA share the same idea to encode the topology of the derivation tree of the SLP by a succinct indexable dictionary, and heavily use it for simulating several navigational operations on the tree. As its operation time is the theoretical bottleneck of FOLCA, appearing as $O(\lg n / \lg \lg n)$ factor, we show that we can improve it to constant time. We then propose a practical implementation. Experimental results show that the improved version runs about 30% faster than original FOLCA.

Finally, we introduce a customized cache structure to grammar compression. The idea is inspired by the work [27] that proposed a variant of FOLCA working in constant space, in which only a constant number of frequently used variables are maintained to build SLP. Although the algorithm of [27] cannot make use of infrequent variables, it runs very fast as it is quite cache friendly. On the basis of this idea, we introduce a hash table (of size fitting into L3 cache) to lookup reverse dictionary for self-maintained frequent variables. Unlike [27], infrequent variables are looked up by the SOLCA's reverse dictionary. Experimental results show that this simple cache structure significantly improves the running time of plain SOLCA with a small overhead in space.

1.3 Related Work

There are compression algorithms with smaller space. For example, Maruyama and Tabei [27] proposed a variant of FOLCA working in constant space where the reverse dictionary with a fixed size is reset when the vacancy for a new entry runs out. We can find similar algorithms in constant space, e.g., repair, gzip, bzip, and etc. On the other hand, restricting the memory size not only saturates the compression ratio but also interferes with an important application like self-indexes [3, 9, 21, 25, 34, 38, 45, 46] because the memory is reset according

to the increase of input for the constant memory model. In fact, the SLP produced by FOLCA/SOLCA can be used for self-indexes [46], and for this application it is important that the whole text is globally compressed.

2 Framework of Grammar Compression

2.1 Notation

We assume finite sets Σ and V of symbols where $\Sigma \cap V = \emptyset$. Symbols in Σ and V are called *alphabet symbols* and *variables*, respectively. Σ^* is the set of all strings over Σ, and Σ^q the set of strings of length just q over Σ. The length of a string S is denoted by $|S|$. The i-th character of a string S is denoted by $S[i]$ for $i \in [1, |S|]$. For a string S and interval $[i, j]$ $(1 \leq i \leq j \leq |S|)$, let $S[i, j]$ denote the substring of S that begins at position i and ends at position j. Throughout this paper, we set $\sigma = |\Sigma|$, $n = |V|$ and $N = |S|$.

2.2 SLPs

We consider a special type of CFG $G = (\Sigma, V, D, X_s)$ where V is a finite subset of $\mathcal{X}$, D is a finite subset of $V \times (V \cup \Sigma)^*$, and $X_s \in V$ is the start symbol. A grammar compression of a string S is a CFG that derives only S deterministically, i.e., for any $X \in V$ there exists exactly one production rule in D and there is no loop.

We assume that G is an SLP [19]: any production rule is of the form $X_k \to X_i X_j$, where $X_i, X_j \in \Sigma \cup V$, and $1 \leq i, j < k \leq n + \sigma$. The size of an SLP is the number of variables, i.e., $|V|$, and we let $n = |V|$. For variable $X_i \in V$, $val(X_i)$ denotes the string derived from X_i. Also for $c \in \Sigma$, let $val(c) = c$. For $w \in (V \cup \Sigma)^*$, let $val(w) = val(w[1]) \cdots val(w[|w|])$.

The parse tree of G is a rooted ordered binary tree such that (i) the internal nodes are labeled by variables and (ii) the leaves are labeled by alphabet symbols. In a parse tree, any internal node Z corresponds to a production rule $Z \to XY$, where X (resp. Y) is the label of the left (resp. right) child of Z.

The set D of production rules is regarded as the data structure, called the dictionary, for accessing the phrase $X_i X_j$ for any X_k, if $X_k \to X_i X_j$ exists. On the other hand, the reverse dictionary D^{-1} is the data structure for accessing X_k for $X_i X_j$, if $X_k \to X_i X_j$ exists.

2.3 Succinct Data Structures

Here we introduce some succinct data structures, which we will use for encoding an SLP.

A rank/select dictionary for a bit string B [16] is a data structure supporting the following queries: $rank_c(B, i)$ returns the number of occurrences of $c \in \{0, 1\}$ in $B[1, i]$; $select_c(B, i)$ returns the position of the i-th occurrence of $c \in \{0, 1\}$ in B; $access(B, i)$ returns the i-th bit in B. There is a rank/select dictionary for B that uses $|B| + o(|B|)$ bits of space and supports the queries in $O(1)$ time [37]. In addition, the rank/select dictionary can be constructed from B in $O(|B|)$ time and $|B| + o(|B|) + O(1)$ bits of space.

It is natural to generalize the queries for a string T over an alphabet of size > 2. In particular, we consider the case where the alphabet size is $\Theta(|T|)$. Using a data structure called GMR [11], we obtain rank/select dictionary that occupies $|T| \lg |T| + o(|T| \lg |T|)$ bits of space and supports both rank and access queries in $O(\lg \lg (|T|))$ time and select queries in $O(1)$ time. Here we introduce the ingredients of the GMR for T (we remark that we use a simplified GMR as we consider only $\Theta(|T|)$-size alphabets), each of which we refer to as GMRDS1–4. Note that each query uses a distinct subset of them: $select_c(T, i)$ uses GMRDS1–2; $rank_c(T, i)$ uses GMRDS1–3; and $access(T, i)$ uses GMRDS1–2 and GMRDS4.

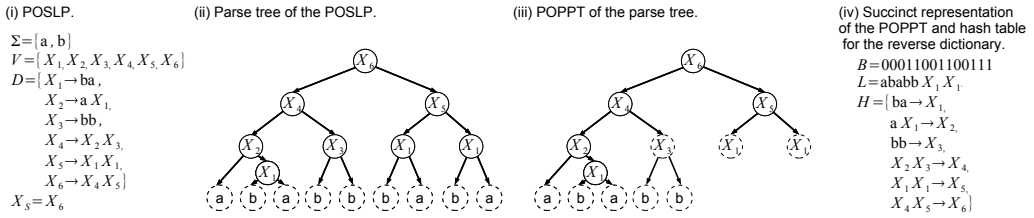


Figure 1 Example of post-order SLP (POSLP), parse tree, post-order partial parse tree (POPPT), and succinct representation of POPPT.

GMRDS1: A permutation π_T of $[1, |T|]$ obtained by stably sorting $[1, |T|]$ according to the values of $T[1, |T|]$. It is stored naively, and thus, occupies $|T| \lg |T|$ bits of space.

GMRDS2: A unary encoding of $T[\pi_T[1]]T[\pi_T[2]] \cdots T[\pi_T[|T|]]$ to support rank/select operations on $GB_T = 0^{T[\pi_T[1]]}10^{T[\pi_T[2]]-T[\pi_T[1]]}1 \ldots 0^{T[\pi_T[|T|]]-T[\pi_T[|T|-1]]}1$. The space usage is $O(|T|)$ bits.

GMRDS3: A data structure to support predecessor queries on sub-ranges of $\pi_T[1, |T|]$. Note that for any character c appearing in T there is a unique range $[i_c, j_c]$ s.t. $T[\pi_T[k]] = c$ iff $k \in [i_c, j_c]$. Also, the sequence $\pi_T[i_c], \pi_T[i_c + 1], \ldots, \pi_T[j_c]$ is non-decreasing. The task is, given such a range and an integer x, to compute the largest position $k \in [i_c, j_c]$ with $\pi_T[k] < x$ if such exists. We can employ y-fast trie to support the queries in $O(\lg \lg |T|)$ time by adding extra $O(|T|)$ bits on top of π_T (note that the search on bottom trees of y-fast trie can be implemented by simple binary search on a sub-range of π_T as we only consider static GMR).

GMRDS4: A data structure to support fast access to $\pi_T^{-1}[i]$ for any $1 \leq i \leq |T|$. We can use the data structure of [31] to compute $\pi_T^{-1}[i]$ in $O(\lg \lg |T|)$ time. It adds extra $O(|T| + \lg |T| / \lg \lg |T|)$ bits on top of π_T.

2.4 Online Construction of Succinct SLP

▶ **Definition 1** (POSLP and post-order partial parse tree (POPPT) [39, 26])**.** A partial parse tree is a binary tree built by traversing a parse tree in a depth-first manner and pruning all of the descendants under every node of a previously appearing nonterminal symbol. A POPPT is a partial parse tree whose internal nodes have post-order variables. A POSLP is an SLP whose partial parse tree is a POPPT.

Figures 1(i) and (iii) show an example of a POSLP and POPPT, respectively. The resulting POPPT (iii) has internal nodes consisting of post-order variables. FOLCA [28] is a fully-online grammar compression for directly computing the succinct POSLP (B, L) of a given string, where B is the bit string obtained by traversing POPPT in post-order, and putting '0', if a node is a leaf, and '1', otherwise, and L is the sequence of leaves of the POPPT. B encodes the topology of POPPT in $2n$ bits by taking advantage of the fact that POPPT is a full binary tree (note that for general trees we need $4n$ bits instead). By enhancing B with a data structure supporting some primitive operations considered in [32] (*fwdsearch* and *bwdsearch* on the so-called *excess array* of B), we can support some basic navigational operations (like move to parent/child) on the tree as well as rank/select queries on B. Using the dynamic data structure proposed in [32], we can support these operations as well as dynamic updates on B in $O(\lg n / \lg \lg n)$ time. In theory, FOLCA uses this result to get Theorem 2 (though its actual implementation uses a simplified version, which only has $O(\lg n)$-time guarantee).

▶ **Theorem 2** ([28]). *Given a string of length N over an alphabet of size σ, FOLCA computes a succinct POSLP of the string in $O(\frac{N \lg n}{\alpha \lg \lg n})$ expected time using $(1+\alpha)n \lg(n+\sigma)+n(3+\lg(\alpha n))$ bits of working space, where $\frac{1}{\alpha} \geq 1$ is the load factor of the hash table.*

In Section 3 we improve FOLCA in two ways: First, we improve the running time for operations on B from both theoretical and practical points of view in Subsection 3.1. Second, we slash $O(\alpha n \lg(n+\sigma))$ bits of working space of FOLCA needed for implementing D^{-1} by hash table. In Subsection 3.2, we propose a novel dynamic succinct POSLP to remove the redundant working space.

3 Improved Algorithm

3.1 Improving and Engineering Operations on B

Recall that FOLCA uses the dynamic tree data structure of [32], for which improving $O(\lg n / \lg \lg n)$ operation time is unlikely due to known lower bound. However, in our problem fully dynamic update operations are not needed as new tree topologies (bits) are always "appended". Therefore, in theory it is not difficult to get constant time operations: While appending bits, we mainly manage to update *range min-max trees* (*RmM-trees* in short) and a weighted level-ancestor data structure. For the former, it is fairly easy to fill up the min/max values for nodes of RmM-trees incrementally in worst case constant time per addition. For the latter, we can use the data structure of [1] supporting weighted level-ancestor queries and updates under adding leaf/root in worst case constant time. As a result, the running time of FOLCA can be improved to $O(N/\alpha)$ expected time.

Next we present a more practical implementation utilizing the fact that our B is well-balanced: Because FOLCA produces a well-balanced grammar, the resulting POPPT has height of at most $2 \lg N$. In our actual implementation, we allow the following overhead in space: We use some precomputed tables that occupy 2^8 bytes each so that some operations (like rank/select) on a single byte can be performed by a table lookup in constant time. Such tables are commonly used in modern implementations of succinct data structures (e.g., sdsl-lite `https://github.com/simongog/sdsl-lite`).

Now we briefly review the static data structure of [32]. Let E denote the excess array of B, i.e., for any $1 \leq i \leq n$, $E[i]$ is the difference of $\mathit{rank}_0(B,i)$ and $\mathit{rank}_1(B,i)$. Note that E is conceptual and we do not have a direct access to E. We consider a primitive query $\mathit{fwdsearch}(E,i,d)$ that returns the minimum $j > i$ such that $E[j] = E[i] + d$, where we assume $d \leq 0$ (it is simplified from the original *fwdsearch*, but enough for our problem). The data structure consists of three layers. The lowest layer partitions B into equal length mini-blocks of $\beta = \Theta(\lg N)$ bits. If query can be answered in a mini-block, it is processed by $O(\beta/8)$ table lookups, otherwise the query is passed to the middle layer. The middle layer partitions B into equal length block of $\beta' = \Theta(\lg^3 N)$ bits. Each block contains $O(\lg^2 N)$ mini-blocks and is managed by an RmM-tree. If the answer exists in a block, the RmM-tree identifies the right mini-block where the answer exists, otherwise the query is passed to the top layer. The task of the top layer is, given a block and target excess value e $(= E[i] + d)$, to find the nearest block (to the right for *fwdsearch*) whose minimum excess value is no greater than e, which is exactly the block where the answer exists.

Our ideas for a practical implementation are listed below:

- Since all excess values are in $[0, 2 \lg N]$, each node of RmM-trees can hold absolute excess value using $1 + \lg \lg N$ bits. (Note that in general case we only afford to store relative values, and thus, we have to retrieve absolute values by traversing from the root of the

tree when needed.) In particular, we can directly access absolute excess values at every ending position of mini-block by storing them in an array $E'[1, \lceil n/\beta \rceil]$, which only uses $O(n \lg\lg N/\beta) = O(n \lg\lg N/\lg N)$ bits.

- Since $rank_0(B, i) = (i - E[i])/2$ and $rank_1(B, i) = (i + E[i])/2$, rank queries are answered by computing $E[i]$, which can be now computed by accessing $E'[\lceil i/\beta \rceil]$ and $O(\lg N/8)$ table lookups.
- For select query $select_0(B, j)$ whose answer is i, we remark that $rank_0(B, i) = j = (i - E[i])/2$ holds. Since $i = 2j + E[i]$ and $E[i] \in [0, 2 \lg N]$, the answer i exists in $[2j, 2j + 2 \lg N]$. Thus, $select_0(B, j)$ can be computed by accessing $E'[\lceil 2j/\beta \rceil]$ and $O(\lg N/8)$ table lookups. Similarly, $select_1(B, j)$ can be answered by screening the range $[2j - 2 \lg N, 2j]$.
- For the top layer, we can simply remember, for every combination of block and target excess value, the answer for *fwdsearch* query. Since the number of possible combinations is $O(n \lg N/\beta')$, it takes $O(n \lg^2 N/\beta') = O(n/\lg N)$ bits.

3.2 Improved Dynamic Succinct POSLP

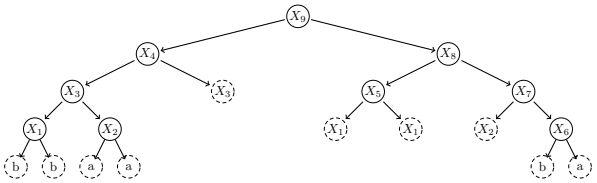
We propose a novel space-efficient representation of POSLP that occupies $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of space *including* the reverse dictionary. The concept of a succinct representation of POSLP is unchanged, but now we consider integrating the reverse dictionary into it.

We start with categorizing every production rule into two groups. A production rule $Z \to XY \in (V \cup \Sigma)^2$ (or variable Z) is said to be *outer*, if both children of the node corresponding to Z in the POPPT are leaves, and *inner*, otherwise. The reverse dictionaries for inner and outer variables are implemented differently. Particularly, the reverse dictionary for inner variables can be implemented *without* having any other data structures than (B, L) (see Section 3.2.1). Although we do not know which dictionary is to be used when looking up a phrase, it is sufficient to try them both.

The proposed dynamic succinct POSLP consists of the same (B, L) as the previous POSLP. The difference is the encoding of L: We partition L into L_1, L_2, and L_3 such that L_2 (resp. L_3) consists of every element of L that is a left (resp. right) child of an outer variable (preserving their original order), and L_1 consists of the remaining elements. In addition, we add functions $rank_{001}(B, i)$ and $select_{001}(B, i)$ to B, which return the number of occurrences of 001 in $B[1, i + 2]$ and the position of the i-th occurrence of 001 in B, respectively. Note that each occurrence of 001 corresponds to an occurrence of outer variable, and $rank_{001}/select_{001}$ enables us to map any leaf to the corresponding entry distributed to one of L_1, L_2 and L_3. More precisely, given any position i in B representing a leaf (i.e., $B[i] = 0$), the corresponding label is retrieved as follows: return $L_2[rank_{001}(B, i)]$, if $B[i, i + 2] = 001$; return $L_3[rank_{001}(B, i)]$, if $B[i - 1, i + 1] = 001$; and return $L_1[rank_0(B, i) - 2rank_{001}(B, i)]$, otherwise. While storing L_1 in a standard variable length array that supports pushback of elements, we store L_2 and L_3 implicitly in a data structure that provides the functionality of the reverse dictionary for outer variables.

Let n_{in} and n_{out} be the numbers of inner and outer variables, respectively, i.e., $n_{\mathsf{in}} = |L_1|$ and $n_{\mathsf{out}} = |L_2| = |L_3|$. Each of L_2 and L_3 is further partitioned into the prefix of length n'_{out} and the suffix of length $n_{\mathsf{out}} - n'_{\mathsf{out}}$ for some n'_{out} satisfying $n_{\mathsf{out}} - n'_{\mathsf{out}} < \frac{n_{\mathsf{out}}}{\lg\lg n_{\mathsf{out}}}$, that is, the suffixes are relatively short. Let π_2 be the permutation of $[1, n'_{\mathsf{out}}]$ obtained by sorting $[1, n'_{\mathsf{out}}]$ stably according to the values of $L_2[1, n'_{\mathsf{out}}]$, and let $\hat{L}_2 = L_2[\pi_2[1]]L_2[\pi_2[2]] \cdots L_2[\pi_2[n'_{\mathsf{out}}]]$ and $\hat{L}_3 = L_3[\pi_2[1]]L_3[\pi_2[2]] \cdots L_3[\pi_2[n'_{\mathsf{out}}]]$. Roughly we consider a two-stage GMR, the first for $L_2[1, n'_{\mathsf{out}}]$ and the second for $\hat{L}_3$ (although we only use select/access queries for

(i) Example of the POPPT.



(ii) Succinct representation of the POPPT.

$$B = 00100110100100011111$$
$$L = b, b, a, a, X_3, X_1, X_1, X_2, b, a$$

(iii) Decomposition of L: if the parent of $L[i]$ is inner, $L[i] \in L_1$, else if $L[i]$ is the left child, $L[i] \in L_2$, and otherwise, $L[i] \in L_3$.

$$L_1 = X_3, X_2$$
$$L_2 = b, a, X_1, b$$
$$L_3 = b, a, X_1, a$$

(iv) Encode of L: L_1 is represented by the integer array. The prefix $L_2[1, n'_{\mathsf{out}}]$ is represented by the bit array GB_2 and the permutation π_2 in GMR. The remaining short suffix of L_2 is represented by the integer array GA_2 (iv-1). In the GMR encoding of $L_2[1, n'_{\mathsf{out}}]$, $L_2[1, n'_{\mathsf{out}}]$ is sorted in lexicographical order and each $L_3[i]$ is sorted by the rank of $L_2[i]$ (iv-2). Then, L_3 is similarly encoded with n'_{out} dividing them into the suffix and prefix (iv-3). Additionally, the hash table h returns i ($i > n'_{\mathsf{out}}$) if $L_2[i] = X_j$ and $L_3[i] = X_k$ for the query X_jX_k (iv-4).

(iv-1) Data structure for L_2 ($n'_{\mathsf{out}} = 2$).
$GB_2 = 101$, $\pi_2 = 2, 1$
$GA_2 = X_1, b$

(iv-3) Data structure for L_3.
$GB_3 = 101$, $\pi_3 = 1, 2$
$GA_3 = X_1, a$

(iv-2) Sort $L_2[1, n'_{\mathsf{out}}]$ and $L_3[1, n'_{\mathsf{out}}]$ to $\hat{L}_2[1, n'_{\mathsf{out}}]$ and $\hat{L}_3[1, n'_{\mathsf{out}}]$, respectively.
$\hat{L}_2[1, n'_{\mathsf{out}}] = a, b$
$\hat{L}_3[1, n'_{\mathsf{out}}] = a, b$

(iv-4) Hash table for $L_2[i]$ and $L_3[i]$ ($i > n'_{\mathsf{out}}$).
$h = \{X_1X_1 \to 3, ba \to 4\}$

(v) The proposed dynamic succinct POPPT is formed by L_1 of (iii), (iv-1), (iv-3), and (iv-4).

Figure 2 Example of the proposed data structure for dynamic succinct POSLP.

$L_2[1, n'_{\mathsf{out}}]$). By the data structures, fitting in $2n'_{\mathsf{out}} \lg(n + \sigma) + o(n'_{\mathsf{out}} \lg(n + \sigma))$ bits of space in total, we can lookup a phrase of outer variables in $[1, n'_{\mathsf{out}}]$ in $O(\lg \lg n)$ time (see Section 3.2.2).

The reverse dictionary for the remaining outer variables (that are in short suffix) is implemented by dynamic perfect hashing [5] that occupies $O(\frac{n_{\mathsf{out}} \lg n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}) = o(n_{\mathsf{out}} \lg n_{\mathsf{out}})$ bits of space and supports lookup and addition in $O(1)$ expected time.

Note that we use "static" GMRs for $L_2[1, n'_{\mathsf{out}}]$ and $\hat{L}_3$. Since most dynamic updates of POSLP are supported by the hash (adding variables in the short suffix one by one), we do nothing to GMRs. When the short suffix becomes too long, i.e., $n_{\mathsf{out}} - n'_{\mathsf{out}}$ reach $\frac{n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}$, we increase n'_{out} (i.e., the number of variables managed by GMRs) by $\frac{n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}$ and just "reconstruct" the static GMRs from scratch (and clear all variables in the hash). Since the GMR for a string can be constructed in linear time to the length of the string, the total cost of reconstruction is $O(\frac{n}{\lg \lg n} \sum_{i=1}^{\lg \lg n} i) = O(n \lg \lg n)$.

Figure 2 shows an example of our POSLP.

In what follows we show how to implement the reverse dictionaries as well as access to the production rules of outer variables.

3.2.1 Reverse dictionary for inner variables

If there is an inner variable deriving XY, at least one of the following conditions holds, where v_X (resp. v_Y) is the corresponding node of X (resp. Y) in the POPPT:

- **(i)** v_X is a left child of its parent, and the parent has a right child (regardless of whether an internal node or leaf) representing Y, and
- **(ii)** v_Y is a right child of its parent, and the parent has a left child (regardless of whether an internal node or leaf) representing X.

Therefore, $D^{-1}(XY)$ can be looked up by a constant number of parent/child queries on B and access to L_1. Moreover, the next lemma suggests that we do not need to check both conditions (i) and (ii); check (ii), if $X < Y$, and check (i), otherwise.

▶ **Lemma 3.** *Let Z be an inner variable deriving $XY \in (V \cup \Sigma)^2$, and v_Z be the corresponding node of Z in the POPPT. If $X < Y$, the right child of v_Z is an internal node. Otherwise the left child of v_Z is an internal node.*

Proof. $X < Y$: Assume for the sake of contradiction that the right child of v_Z is a leaf (which represents Y). As Z is inner, the left child of v_Z must be the internal node corresponding to X. Since Y is larger than X and smaller than Z, the internal node corresponding to Y must be in the subtree rooted at the right child of v_Z, which contradicts the assumption.

$X \geq Y$: Assume for the sake of contradiction that the left child of v_Z is a leaf (which represents X). As Z is inner, the right child of v_Z must be the internal node corresponding to Y. Since the internal node corresponding to X appears before the left child of v_Z, $X < Y$ holds, a contradiction. ◀

Due to Lemma 3 and the above discussions, we get the following lemma.

▶ **Lemma 4.** *We can implement the reverse dictionary for inner variables that supports lookup in $O(1)$ time.*

3.2.2 Reverse dictionary for outer variables

▶ **Lemma 5.** *We can implement the reverse dictionary for outer variables to support lookup in $O(\lg \lg n)$ expected time.*

Proof. Recall that for any $1 \leq i \leq n'_{\mathsf{out}}$ the pair $L_2[i]L_3[i]$ is the right-hand side of the i-th outer production rule (in post-order). Given i, we can compute the post-order number of the variable deriving $L_2[i]L_3[i]$ by $rank_1(B, select_{001}(B, i)) + 1$. Hence, the task of our reverse dictionary is, given $XY \in (V \cup \Sigma)^2$, to return integer i such that $L_2[i] = X$ and $L_3[i] = Y$, if such exists. If a phrase is found in the short suffix, the query is answered in $O(1)$ expected time by using hash table. Thus, in what follows, we focus on the case where the answer is not found in the short suffix.

By the GMRDS2 GB_2 for $L_2[1, m']$, we can compute in constant time, given an integer X, the range $[i_X, j_X]$ in π_2 such that the occurrences of X in L_2 is represented by $\pi_2[i_X, j_X]$ in increasing order, namely, $i_X = rank_1(GB_2, select_0(GB_2, X)) + 1$ and $j_X = rank_1(GB_2, select_0(GB_2, X + 1))$. Note that Y occurs in $\hat{L}_3[i_X, j_X]$ (the occurrence is unique) iff there is an outer variable deriving XY. In addition, if $k \in [i_X, j_X]$ is the occurrence of Y, then $\pi_2[k]$ is the post-order number of the variable we seek. Hence, the

Table 2 Detail of memory consumption (MB).

	Wikipedia				genome			
method	B	L	H	CRD	B	L	H	CRD
FOLCA	17.63	180.06	1342.43	–	141.00	1247.67	9442.64	–
FOLCA+	17.26	180.06	1342.43	–	138.09	1247.67	9442.64	–
SOLCA	17.26	523.85	–	–	138.09	3856.84	–	–
SOLCA+CRD	17.26	523.85	–	22.00	138.09	3856.84	–	22.00

problem reduces to computing $select_Y(\hat{L}_3, rank_Y(\hat{L}_3, i_X - 1) + 1)$, which can be performed in $O(\lg \lg n)$ time by using the GMR for $\hat{L}_3$. ◀

3.2.3 Access to the production rules of outer variables

Since L_2 and L_3 are stored implicitly, here we show how to access the production rules of outer variables.

▶ **Lemma 6.** *Given $1 \leq i \leq n_{\mathsf{out}}$, we can access $L_2[i]L_3[i]$ in $O(\lg \lg n)$ time.*

Proof. If $i > n'_{\mathsf{out}}$, $L_2[i]L_3[i]$ is in the short suffixes. As we can afford to store $L_2[i]L_3[i]$ in a plain array of $O(\frac{n_{\mathsf{out}} \lg n_{\mathsf{out}}}{\lg \lg n_{\mathsf{out}}}) = o(n_{\mathsf{out}} \lg n_{\mathsf{out}})$ bits of space, we can access it in $O(1)$ time.

If $i \leq n'_{\mathsf{out}}$, $L_2[i]L_3[i]$ is represented by GMRs for $L_2[1, n_{\mathsf{out}}]$ and $\hat{L}_3$. Using GMRDS4 for $L_2[1, n_{\mathsf{out}}]$, we can compute $j = \pi_2^{-1}[i]$ in $O(\lg \lg n)$ time. Then, we can obtain $L_2[i]$ by $rank_0(GB_2, select_1(GB_2, j))$ in $O(1)$ time. In addition, $L_3[i]$ can be retrieved by accessing $\hat{L}_3[j]$, which is supported in $O(\lg \lg n)$ time by GMR for $\hat{L}_3$. ◀

To tell the truth, SOLCA does not access the production rules of outer variables during compression, and hence, the implementation of SOLCA is further simplified by deleting GMRDS4 for both $L_2[1, n_{\mathsf{out}}]$ and $\hat{L}_3$, needed to support access queries on the GMRs.

3.3 SOLCA

Plugging our new succinct representation of POSLP into FOLCA, we get a space-optimal grammar compression algorithm, SOLCA.

▶ **Theorem 7.** *Given a string of length N over an alphabet of size σ, SOLCA computes a succinct POSLP of the string in $O(N \lg \lg n)$ expected time using $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of working space.*

Proof. SOLCA processes the input string online exactly the same as FOLCA does. During compression, it is required to lookup a phrase by the reverse dictionary and append new variables to POSLP if the phrase does not exist so far. By Lemmas 4 and 5, this is done in $O(\lg \lg n)$ expected time. Our dynamic succinct POSLP including the reverse dictionary takes only $n \lg(n + \sigma) + o(n \lg(n + \sigma))$ bits of space as described in Section 3.2. ◀

4 Experiments

We implement FOLCA applying the dynamic succinct tree representation introduced in Section. 3.1 called FOLCA+ and the SOLCA proposed in Section 3.3.[3] Furthermore, as

[3] Currently we do not implement the last idea of Section 3.1 for *fwdsearch* queries. Instead we answer queries by traversing up a tree (so called 2D-Min-Heap [6]) built on the minimum excess values of

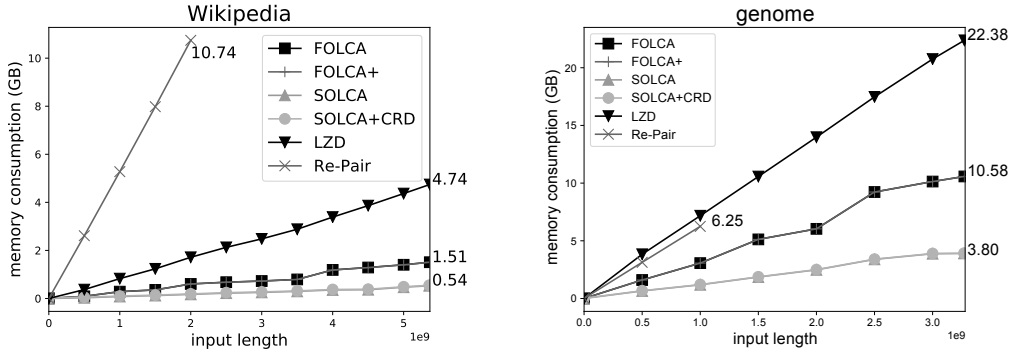


Figure 3 Working space for Wikipedia (left) and genome (right).

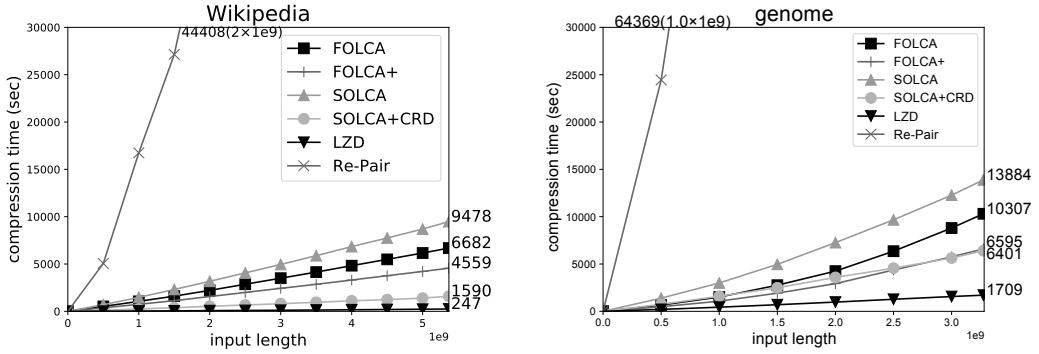


Figure 4 Compression time for Wikipedia (left) and genome (right).

a practical method for the fast computation of SOLCA, we implement the SOLCA with the constant space reverse dictionary (CRD) storing frequent production rules. We call it SOLCA+CRD[4]. The CRD is proposed in [27] and it supports the reverse dictionary query in constant expected time while keeping a constant space by constant space algorithms for finding frequent items [18, 24, 30]. The reverse dictionary query of SOLCA+CRD is performed by two phases: (1) we check if a given X_iX_j exists in the CRD and (2) if the X_iX_j is not found in phase (1), we check the reverse dictionary of SOLCA. Although the worst case time of the reverse dictionary query of SOLCA+CRD is the same as SOLCA's $O(\lg\lg(n+\sigma))$ time, if the query rule exists in the CRD, we can support the query in constant expected time. Our implementation of CRD is based on [18] and restricts the space to 22MB that is almost the same cache size of experimental machine. We compare the time/space consumption of these variants of FOLCA with that of existing three grammar compression algorithms: FOLCA, LZD [5] [13] and Re-Pair [6] [2]. The Re-Pair is a space-efficient version of the original algorithm [22]. The experiments perform on Intel Xeon Processor E7-8837

blocks. In the worst case it requires an $O(\lg N)$-long traversal, but it works well enough in practice as performing such a long traversal is rare.

[4] This implementation is downloadable from `https://github.com/tkbtkysms/solca`. We will show additional experiments in this web site.

[5] The patricia trie space computation (the compress function of the class STree::Tree) in `https://github.com/kg86/lzd`

[6] `https://github.com/nicolaprezza/Re-Pair`

Table 3 Statistical information of input strings.

dataset	length of string (N)	alphabets (σ)	compression ratio (%)		
			SOLCA	LZD	Re-Pair
Wikipedia	5,368,709,120	210	3.65	3.46	0.62[9]
genome	3,273,481,150	20	41.38	36.34	9.05[10]

(2.67GHz, 24MB cache, 8 cores) and 1TB RAM. Here, the load factor of the hash table used in FOLCA is fixed to $\frac{1}{\alpha} = 1$.

We use two large-scale datasets: Wikipedia[7] (5GB) and genome[8] (3GB). The detail is shown in Table 3 where we note that POSLP by SOLCA is exactly the same as FOLCA's. The difference is only their succinct representations.

Figure 3 shows a comparison of the memory consumption of each method for Wikipedia and genome. The points are displayed for every length of 5×10^8. FOLCA and FOLCA+ maintain data structure (B, L, H); B is the skeleton of POSLP T, L is the sequence of the leaves of T, and H is the reverse dictionary. When $\alpha = 1$, H occupies almost $2n \lg(n + \sigma)$ bits. Since the size of B and L is $n \lg(n + \sigma)$ bits and $2n$ bits, respectively, the total space of FOLCA's variants is about $3n \lg(n + \sigma)$ bits. On the other hand, SOLCA and SOLCA+CRD maintains (B, L') supporting the reverse dictionary; L' is the representation of L in Section 3.2. The size is almost the same as L. Thus, it is expected that the memory consumption of SOLCA and SOLCA+CRD is about $\frac{1}{3}$ of FOLCA's. The experimental result confirms this prediction on both datasets. Furthermore, the memory consumption of each data structure is shown in Table 2. Comparing with other methods, the space of SOLCA and SOLCA+CRD is significantly small for each string.

Figure 4 shows a comparison of the construction time for the input. Our succinct tree representation used in FOLCA+ improves the time consumption of FOLCA. The difference of SOLCA from FOLCA+ comes from the use of L' (queries to L' and reconstruction of L'). SOLCA+CRD is fastest in FOLCA's and SOLCA's variants for Wikipedia and competitive with FOLCA+ for genome. By this result, we can confirm the efficiency of the fast computation of CRD. SOLCA's and FOLCA's variants are faster than Re-pair and slower than LZD.

5 Conclusion

We have presented SOLCA: a space-optimal version of fully-online LCA (FOLCA) [28]. Since FOLCA is extended to its self-index in [46], our future work is developing a self-index based on our SOLCA while preserving the optimal working space.

References

1 Stephen Alstrup and Jacob Holm. Improved algorithms for finding level ancestors in dynamic trees. In *27th International Colloquium on Automata, Languages and Programming*, pages 73–84, 2000. doi:10.1007/3-540-45022-X_8.

[7] `https://dumps.wikimedia.org/enwikinews/20170101/enwikinews-20170101-pages-meta-history.xml` (the first 5GB)
[8] `http://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/chr*`
[9] Up to the length of 2.0×10^9.
[10] Up to the length of 1.0×10^9.

2 Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-efficient re-pair compression. In *Data Compression Conference*, pages 171–180, 2017. doi:10.1109/DCC.2017.24.

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

4 Graham Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algorithms*, 3(1):2:1–2:19, 2007. doi:10.1145/1219944.1219947.

5 Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994. doi:10.1137/S0097539791194094.

6 Johannes Fischer. Optimal succinctness for range minimum queries. In *Theoretical Informatics, 9th Latin American Symposium, LATIN 2010, Oaxaca, Mexico, April 19-23, 2010. Proceedings*, pages 158–169, 2010. doi:10.1007/978-3-642-12200-2_16.

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

8 Shouhei Fukunaga, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. Online grammar compression for frequent pattern discovery. In *13th International Conference on Grammatical Inference*, pages 93–104, 2016.

9 Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. A faster grammar-based self-index. In *6th International Conference Language and Automata Theory and Applications*, pages 240–251, 2012. doi:10.1007/978-3-642-28332-1_21.

10 Pawel Gawrychowski. Optimal pattern matching in LZW compressed strings. *ACM Trans. Algorithms*, 9(3):25:1–25:17, 2013. doi:10.1145/2483699.2483705.

11 Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 368–373, 2006.

12 Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18:89–99, 2013. doi:10.1016/j.jda.2012.07.006.

13 Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. LZD factorization: Simple and practical online grammar compression with variable-to-fixed encoding. In *26th Annual Symposium on Combinatorial Pattern Matching*, pages 219–230, 2015. doi:10.1007/978-3-319-19929-0_19.

14 Danny Hermelin, Gad M. Landau, Shir Landau, and Oren Weimann. A unified algorithm for accelerating edit-distance computation via text-compression. In *26th International Symposium on Theoretical Aspects of Computer Science*, pages 529–540, 2009. doi:10.4230/LIPIcs.STACS.2009.1804.

15 Tomohiro I, Wataru Matsubara, Kouji Shimohira, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, Kazuyuki Narisawa, and Ayumi Shinohara. Detecting regularities on grammar-compressed strings. *Inf. Comput.*, 240:74–89, 2015. doi:10.1016/j.ic.2014.09.009.

16 Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

17 Artur Jez. Faster fully compressed pattern matching by recompression. *ACM Trans. Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

18 Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003. doi:10.1145/762471.762473.

19 Marek Karpinski, Wojciech Rytter, and Ayumi Shinohara. An efficient pattern-matching algorithm for strings with short descriptions. *Nord. J. Comput.*, 4(2):172–186, 1997.

20 Dominik Kempa and Dmitry Kosolobov. Lz-end parsing in compressed space. In *Data Compression Conference*, pages 350–359, 2017. doi:10.1109/DCC.2017.73.

21 Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theor. Comput. Sci.*, 483:115–133, 2013. doi:10.1016/j.tcs.2012.02.006.

22 N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000. doi:10.1109/5.892708.

23 Eric Lehman. *Approximation algorithms for grammar-based data compression*. PhD thesis, MIT, Cambridge, MA, USA, 2002.

24 Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *28th International Conference on Very Large Data Bases*, pages 346–357, 2002.

25 Shirou Maruyama, Masaya Nakahara, Naoya Kishiue, and Hiroshi Sakamoto. Esp-index: A compressed index based on edit-sensitive parsing. *J. Discrete Algorithms*, 18:100–112, 2013. doi:10.1016/j.jda.2012.07.009.

26 Shirou Maruyama, Hiroshi Sakamoto, and Masayuki Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):214–235, 2012. doi:10.3390/a5020214.

27 Shirou Maruyama and Yasuo Tabei. Fully online grammar compression in constant space. In *Data Compression Conference*, pages 173–182, 2014. doi:10.1109/DCC.2014.69.

28 Shirou Maruyama, Yasuo Tabei, Hiroshi Sakamoto, and Kunihiko Sadakane. Fully-online grammar compression. In *20th International Symposium on String Processing and Information Retrieval*, pages 218–229, 2013. doi:10.1007/978-3-319-02432-5_25.

29 Wataru Matsubara, Shunsuke Inenaga, Akira Ishino, Ayumi Shinohara, Tomoyuki Nakamura, and Kazuo Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theor. Comput. Sci.*, 410(8-10):900–913, 2009. doi:10.1016/j.tcs.2008.12.016.

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

31 J. Ian Munro, Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct representations of permutations and functions. *Theor. Comput. Sci.*, 438:74–88, 2012. doi:10.1016/j.tcs.2012.03.005.

32 Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Trans. Algorithms*, 10(3):16:1–16:39, 2014. doi:10.1145/2601073.

33 Craig G. Nevill-Manning and Ian H. Witten. Compression and explanation using hierarchical grammars. *Comput. J.*, 40(2/3):103–116, 1997.

34 Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Dynamic index and LZ factorization in compressed space. In *Prague Stringology Conference*, pages 158–170, 2016.

35 Tatsuya Ohno, Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A faster implementation of online run-length Burrows-Wheeler Transform. In *28th International Workshop on Combinatorial Algorithms (to appear)*, 2017.

36 Alberto Policriti and Nicola Prezza. Computing LZ77 in run-compressed space. In *2016 Data Compression Conference*, pages 23–32, 2016. doi:10.1109/DCC.2016.30.

37 Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4):43, 2007. doi:10.1145/1290672.1290680.

38 Luís M. S. Russo and Arlindo L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Inf. Retr.*, 11(4):359–388, 2008. doi:10.1007/s10791-008-9050-3.

39 Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003. doi:10.1016/S0304-3975(02)00777-6.

40 Hiroshi Sakamoto, Shirou Maruyama, Takuya Kida, and Shinichi Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Transactions*, 92-D(2):158–165, 2009. doi:10.1587/transinf.E92.D.158.

41 Yasuo Tabei, Hiroto Saigo, Yoshihiro Yamanishi, and Simon J. Puglisi. Scalable partial least squares regression on grammar-compressed data matrices. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1875–1884, 2016. doi:10.1145/2939672.2939864.

42 Yasuo Tabei, Yoshimasa Takabatake, and Hiroshi Sakamoto. A succinct grammar compression. In *Combinatorial Pattern Matching, 24th Annual Symposium, CPM 2013, Bad Herrenalb, Germany, June 17-19, 2013. Proceedings*, pages 235–246, 2013. doi:10.1007/978-3-642-38905-4_23.

43 Yoshimasa Takabatake, Kenta Nakashima, Tetsuji Kuboyama, Yasuo Tabei, and Hiroshi Sakamoto. siedm: An efficient string index and search algorithm for edit distance with moves. *Algorithms*, 9(2):26, 2016. doi:10.3390/a9020026.

44 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Variable-length codes for space-efficient grammar-based compression. In *19th International Symposium on String Processing and Information Retrieval*, pages 398–410, 2012. doi:10.1007/978-3-642-34109-0_42.

45 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved esp-index: A practical self-index for highly repetitive texts. In *13th International Symposium on Experimental Algorithms*, pages 338–350, 2014. doi:10.1007/978-3-319-07959-2_29.

46 Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online self-indexed grammar compression. In *22nd International Symposium on String Processing and Information Retrieval*, pages 258–269, 2015. doi:10.1007/978-3-319-23826-5_25.

47 Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984. doi:10.1109/MC.1984.1659158.

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