九州大学学術情報リポジトリ
Kyushu University Institutional Repository

Self-Indexed Grammar-Based Compression by Edit Sensitive Parsing

Maruyama, Shirou
Kyushu University

Sakamoto, Hiroshi
Kyushu Institute of Technology

Baba, Masahiro
Kyushu University

Ono, Hirotaka
Kyushu University

他

https://hdl.handle.net/2324/17913

出版情報：2010-08-13
バージョン：
権利関係：

Self-Indexed Grammar-Based Compression by Edit Sensitive Parsing

Shirou Maruyama[1], Hiroshi Sakamoto[2,4], Masahiro Baba[1], Hirotaka Ono[1], Kunihiko Sadakane[3], and Masafumi Yamashita[1]

[1] Kyushu University, 680-4 Kawazu, Iizuka-shi, Fukuoka, 820-8502,
[2] Kyushu Institute of Technology, 744 Motooka, Nishi-ku, Fukuoka-shi, Fukuoka 819-0395,
[3] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430,
[4] PRESTO, Japan Science and Technology Agency, 4-1-8 Honcho Kawaguchi, Saitama 332-0012, JAPAN
{shiro.maruyama@i,mbaba@tcslab.csce,mak@csce.}.kyushu-u.ac.jp, hiroshi@ai.kyutech.ac.jp, sada@nii.ac.jp

Abstract. A space-economical self-index on the grammar-based compression is proposed. The algorithm by [20] approximates the optimum compression for a given string within $O(\log^* u \log u)$ ratio for the string length u. Adopting this algorithm with a new succinct data structure for full binary tree, a grammar-based compression represented by a CFG in Chomsky normal form is transformed to a compressed index. The index size is $2n \log n + h \log n + n \log h + 4n + o(n \log n)$ bits, where n is the number of different variables in the grammar-based compression G and h is the height of G bounded by $O(\log u)$. The time to count occurrences for a pattern P of length m is estimated to $O(m \log^2 n + occ_c(\log n \log m + h))$, where occ_c is the occurrence number of *core of* P, which is derived from alphabet reduction in [7]. The parameter occ_c is almost equal to the occurrence of P for sufficiently long P. Experiments for large text data also show the time/space efficiency and the high performance for long pattern search compared with other compact index structures.

1 Introduction

We propose a self-index based on a theory of grammar-based compression. The proposed index is realized by a grammar-based compression described in a CFG and the succinct data structure used in a full binary tree. This compressed index differs from other compressed indexes, such as LZ-index [15], FM-index [8], and Compressed Suffix Array [17], because no sorting for any suffixes is carried out. Since our method avoids sorting issues, it is advanced in terms of long pattern search efficacy, while the search time for frequent patterns does decline. On the other hand, since it is suited for finding the occurrences of long patterns, it is expected to be applicable toward such uses as compressing two documents, comparing their similarities, and extracting common patterns.

First, we cover the theory of grammar-based compression, which is the basis of this research, as well as related research. The smallest grammar-based

compression problem is NP-hard, whereby the size of the CFG, which only generates the given text, is minimized. It is believed impossible to carry out $O(\alpha)$-approximation for any constant α. In this problem, however, any string of length u is $O(\log u)$-approximable by several linear time algorithms [4, 16, 19]. Especially, the algorithm proposed in [20] is deeply relevant to this research. In other algorithms, since random access for all substrings appearing in text and a suffix tree is needed, the space complexity becomes $\Omega(u)$.

The edit sensitive parsing based on the alphabet reduction is the most important concept of this research, and it is the key to working around the sort problem while realizing rapid search from compressed data. Alphabet reduction and related techniques were originally presented to approximate a variant of the edit distance problem [1, 2, 18]. The alphabet reduction in [7] has been applied to the compression algorithm in [20]. Our self-index is based on such the grammar-based compression algorithm. Related to this work, [6] presented an algorithm that combines the alphabet reduction and LZ77 algorithm to nearly optimize the substring compression problem to decide the compression of a substring $S[i, j]$ by the compression of S only.

In order to build the self-indexed grammar-based compression while saving space, this research presents the representation of a new succinct data structure for full binary tree. The succinct data structures used until now within trees involves a method of allocating balanced parentheses while traversing trees [3, 10, 14]. Any tree of size n can be expressed by $2n + o(n)$ bits, where, $o(n)$ is the space of auxiliary data structures for traversing trees. The succinct data structure presented in this research that is used for the parsing tree in the CFG can further reduce this space. The CFG is limited by the Chomsky normal form; thus, the parsing tree becomes a full binary tree. In other words, it becomes a binary tree in which all internal nodes possess exactly two children. In order to encode each node, it is sufficient to distinguish whether that is an internal node or a leaf. For this reason, the size of the data structure can be kept to $n + o(n)$ bits. By expressing the parsing tree of the CFG by this succinct data structure, a self-index can be built in a space less than that of the size of the text.

Now that we let n be the number of variables in grammar compression G used for string S of length u, and also let h be the height of parsing tree G, the size of the index presented in this paper can be $2n \log n + h \log n + n \log h + 4n + o(n \log n)$ bits. By the result in [20], n is bounded by $O(n_* \log u)$ for the optimum solution size n_* and h is bounded by $O(\log u)$. Therefore, if the text can be adequately compressed, the index size and memory consumption will be less than that of the text size. The time to count occurrences for pattern P of length m is represented by $O(m \log^2 n + occ_c(\log n \log m + h))$, Here occ_c represents the number of occurrences of *cores* of P. The core has theoretically been represented as encoding substrings of P, which are sufficiently long, when P is long. For this reason, occ_c is almost equal to the number of occurrences of P.

Finally, we would cover the relation to other compressed indexes. According to the method presented in this research, since the search time is bounded by the grammar size, the text size u is not explicitly included in the search time.

Therefore, if there is an easily compressible text, the theoretical complexity of the method presented herein possesses an advantage over other methods. While theoretical results have been obtained from compression indexes such as [5] that have integrated the advantages of grammar compression, it can be shown that the compressed index presented in this research possesses not only a good approximation of optimal compression but has also been effective in comparative experiments with other compression indexes [8,17,15].

2 Preliminaries

We assume a finite set Σ of alphabet symbols. The set of all strings over Σ is denoted by Σ^*. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. A string $\{a\}^*$ of length at least two is called a *repetition of* a. $S[i]$ and $S[i,j]$ denote the i-th symbol of S and the substring from $S[i]$ to $S[j]$, respectively.

The expression $\log^* n$ denotes the maximum integer j which satisfies $F(j) \leq n$ for $F(0) = 1$, and $F(j) = 2^{F(j-1)}$ $(j \geq 1)$. For instance, $F(3) = 2^4 = 16$, $F(4) = 2^{16} = 65536$, and $F(5) = 2^{65536}$. We thus treat $\log^* n$ as a constant.

Our index is a compact representation obtained by the *edit sensitive parsing* [7] based on a special transformation of string called *alphabet reduction*. We first recall such important notions.

2.1 Alphabet reduction

A string $S \in \Sigma^*$ of length n is partitioned into maximal nonoverlapping substrings of three types:

Type1: a maximal repetition of a symbol;
Type2: a maximal substring longer than $\log^* n$ not containing of a repetition;
Type3: any other short substrings.

Each such substring is called a *metablock*. We focus on only Type2 metablocks since the others are not related to the alphabet reduction. From a Type2 string s, a label string $label(s)$ is computed as follows.

Alphabet reduction: Consider $s[i]$ and $s[i-1]$ represented as binary integers. Denote by ℓ the least bit position in which $s[i]$ differs from $s[i-1]$. For instance, if $s[i] = 101, s[i-1] = 100$ then $\ell = 0$, and if $s[i] = 001, s[i-1] = 101$ then $\ell = 2$. Let $bit(\ell, s[i])$ be the value of $s[i]$ at the ℓth bit location. Then $label(s[i])$ is defined by $label(s[i])$ as $2\ell + bit(\ell, s[i])$.

By this procedure, we generate a string $label(s)$ which is the sequence of such $label(s[i])$. For the resulting $label(s)$, we have the fact that $label(s[i]) \neq label(s[i+1])$ if $s[i] \neq s[i+1]$ for any i (See [7]). Thus the alphabet reduction is recursively applicable to $label(s)$, which is also Type2.

If the alphabet size in s is σ, the new alphabet size in $label(s)$ is $2\log\sigma$. We iterate this process for the resulting string $label(s)$ until the size of the alphabet no longer shrinks. This takes $\log^*\sigma$ iterations.

(1)string in binary	a 000	d 011	e 100	g 110	h 111	e 100	c 010	a 000	d 011	e 100	g 110
(2) labels	_	001	000	011	001	000	011	010	001	000	011
(3) labels as integers	_	1	0	3	1	0	3	2	1	0	3
(4) final labels & landmarks	_	[1]	0	[2]	1	[0]	1	[2]	1	0	[2]

Fig. 1. Alphabet reduction: The line (1) is an original Type2 string s from the alphabet $\{a, b, \cdots, h\}$ with its binary representation. An underline denotes the least different bit position to the left. (2) is the sequence of $label(s[i])$ formed from the alphabet $\{0, 1, 2, 3\}$ whose size is less than 6, and (3) is its integer representation. (4) is the sequence of the final labels reduced to $\{0, 1, 2\}$ and the landmarks indicated by squares.

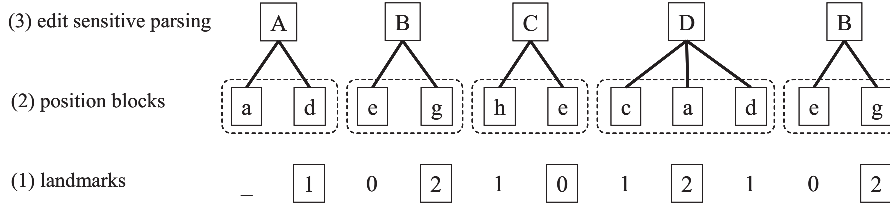

Fig. 2. Edit sensitive parsing (ESP): This figure is one iteration of ESP for the string in Fig. 1. The line (1) is the computed final labels and landmarks from the Type2 string s. (2) shows the groups of all positions in s having two or three around the landmarks. (3) is the corresponding ESP using suitable new symbols A,B,C, and D.

After the final iteration of alphabet reduction, the alphabet size is reduced to at most 6 like $\{0, \cdots, 5\}$ (See [7]). Finally we transform $label(s) \in \{0, \cdots, 5\}^*$ to the same length string in $label(s) \in \{0, 1, 2\}^*$ by replacing each 3 with the least integer in $\{0, 1, 2\}$ that does not neighbor the 3, and doing the same replacement for each 4 and 5. We note that the final string $label(s)$ is also Type2 string.

Landmark: For a final string $label(s)$, we pick out special locations called landmarks that are sufficiently close together. We select any position i as a landmark if $label(s[i])$ is a local maximum. Following this, we select any position j as a landmark if $label(s[j])$ is local minimum and both $j-1, j+1$ are not selected yet. We display the whole process of the alphabet reduction and finding landmarks from a sample Type2 string in Fig. 1.

2.2 Edit sensitive parsing

After computing the sequence $label(s)$ of final labels and landmarks for a Type2 string s, we next partition s into blocks of length two or three around the landmarks in the manner: We make each position part of the block generated by its closest landmark, breaking ties to the right.

As was shown in [7], for any two successive landmark positions i and j, we have $2 \leq |i-j| \leq 3$. Thus, each position block is of length two or three. The string s is transformed to a shorter string s' by replacing any block of two or three symbols to a new suitable symbol. Here "suitable" means that any two blocks partitioning a same substring in s must be replaced by a same symbol. This parsing is called *edit sensitive parsing*. Such renaming can be performed in $O(1)$ time with high provability by a hash function like [11]. These parsing process is shown in Fig. 2.

Finally, we mention the parsing manner for Type1 or Type3 string s. If $|s| \geq 2$, we parse the leftmost two symbols of s as a block and iterate on the remainder and if the length of it is three, then we parse the three symbols as a block. We note that no Type1 s in length one exists. The remained case is Type3 s and $|s| = 1$, which appears in a context a^*bc^*. If $|a^*| = 2$, b is parsed as the block aab. If $|a^*| > 2$, b is parsed as the block ab. If $|a^*| = 0$, b is parsed with c^* analogously.

For an input string S, if it is partitioned into $s_1, \ldots, s_k$ of Type1, Type2, or Type3 substrings. After parsing them, the transformed strings s'_i are concatenated together. This process iterated until $|s'_1 \cdots s'_k| = 1$, i.e. a parsing tree for S is constructed. By the parsing manner for all types of substrings, we can obtain a balanced $2-3$ tree, in which any internal node has two or three children.

3 Self-Indexed Grammar-Based Compression

We introduce a compact representation of ESP tree. Here we define several notations used in this section. Given an ESP tree for an input string, each symbol attached to an internal node is called a *variable*. Since each internal node has two or three children, an ESP tree is identical to a context-free grammar (CFG), whose production rules are denoted by $X \to AB$ or $Y \to ABC$.

A data structure D to access AB from X (or ABC from Y) is called a *dictionary*, and another data structure D^R to access X from AB (or Y from ABC) is called a *reverse dictionary*.

3.1 Pattern matching on ESP tree

We consider a special property of the ESP tree T for a text S: There exist positive integers k, ℓ such that for any pattern P, if $S[i,j] = P$, then T has the internal node labeled by a variable X generating the substring $S[i+k, i+k+\ell]$ of $S[i,j]$. If we can always take such a common variable X for all occurrences of P and ℓ, which is the length of the encoded substring by X, is sufficiently long, this property is available as a necessary condition for searching P, i.e. any occurrence of P in S is restricted around X. In this subsection we show that such a maximal variable, called *a core of P for S*, exists provided T and P are sufficiently long, and we can decide it by parsing P using D^R for a text S.

Lemma 1. There exists a constant $0 < \alpha < 1$ such that for any occurrence of P in S, its core is encoding a substring longer than $\alpha|P|$.

Proof. We first consider the case that P is a Type2 metablock. As shown in [7], determining the closest landmark on $S[i]$ depends on $S[i - \log^* n + 5, i]$ and $S[i, i + 5]$. Thus, if $S[i, j] = P$, then the final labels for the inside part $S[i + \log^* n + 5, j - 5]$ are the same for any occurrence position i of P. By this, each substring equivalent to $S[i + \log^* n + 5, j - 5]$ is transformed to the same S'. Since the ESP tree is balanced $2 - 3$ tree, any variable in S' encodes at least two symbols. If S' is Type2 again, then this process continues. Thus, after k iterations, the length of the string encoded by a variable in S' is at least 2^k. On the other hand, by one iteration, the common substring S' loses its prefix and suffix of length at most $\log^* n + 5$ and each lost variable has at most three children. By this observation, we can take an internal node as a core of P for S, whose height is the maximum k satisfying

$$2(\log^* n + 5)(3 + 3^2 + \cdots 3^k) < (\log^* n + 5)3^{k+2} \leq |P|.$$

By this estimation, since $\log^* n$ is regarded as a constant and a variable in height k encodes a substring of length at least 2^k, we obtain a constant $0 < \alpha < 1$ and a variable as a core of P encoding a substring of length at least $\alpha|P|$. Generally P is divided into m metablocks like $P = P_1 P_2 \cdots P_m$. Type1 and Type3 metablocks in $P_2 \cdots P_{m-1}$ are uniquely parsed in its any occurrence. Thus we can assume $P = P_1 P_2 P_3$ for a long Type2 metablock P_2 and Type1 P_1, P_3 as a worst case. For any occurrence of Type1 metablock, we can obtain a sufficiently large core. Choosing a largest core from the three metablocks, the size is greater than $\alpha|P|$.

Using Lemma 1, the search problem for P is reduced to the search problem for the sequence of adjacent cores.

Lemma 2. For a given ESP tree T of a text S and a pattern P, $S[i, j] = P$ iff there exist $k = O(\log |P|)$ adjacent subtrees in T rooted by variables $x_1, \ldots, x_k$ such that the concatenation of all strings encoded by them is equal to P.

Proof. If the bound $k = O(\log |P|)$ is not necessary, we can always obtain trivial subtrees equal to the leaves $S[i], S[i + 1], \ldots, S[j]$. By using Lemma 1, we can find a core encoding a long substring of $S[i, j]$ longer than $\alpha|j - i|$ for a fixed $0 < \alpha < 1$. The remained substrings are also covered by their own cores. Thus we obtain the bound $k = O(\log |P|)$.

By Lemma 1 and 2, we can obtain the sequence $x_1, \ldots, x_k$ of cores for any pattern P. Let x be one of them. For any node v labeled x, we can check whether the adjacent subtrees rooted by $x_1, \ldots, x_k$ are embedded around v in $O(\log |P|)$ time by traversing at most $O(\log |P|)$ edges in the ESP tree from v. Practically, we choose as large subtree x as possible.

An ESP tree T is represented by a pruned tree to merge common subtrees. Any subtree is merged to the leftmost occurrence of the same subtree by a link structure, and the pruned tree also contains the link to the next occurrence of any subtree. This representation is illustrated in Fig. 3. We next propose a succinct data structure for the representation.

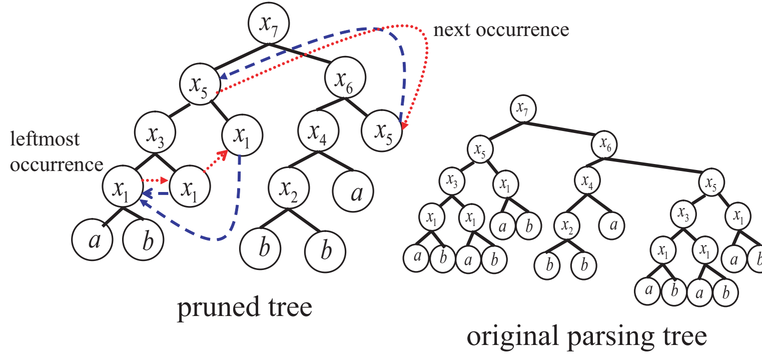

Fig. 3. Pruned parsing tree: This figure shows a parsing tree of a grammar-based compression and its compact representation. For any deleted subtree, the root has two pointers to the leftmost occurrence and the predecessor of the same subtree in the original parsing tree. Here all pointers for leaves are omitted for simplicity.

3.2 Data structure using succinct representation

We define the succinct data structures for the *rank/select/access* operations. Consider a string $S[1,n]$ from an alphabet Σ !%We define three operations for S as follows:

> $rank_c(S,i)$ returns the number of occurrences of $c \in \Sigma$ appearing in $S[1,i]$,
> $select_c(S,i)$ returns the position j such that $rank_c(S,j) = i$, and
> $access(S,i)$ returns $S[i]$.

A basic succinct data structure uses $n + o(n)$ bits for $|\Sigma| = \sigma = 2$ (See [13]), which supports rank/select/access in $O(1)$. For general case, there exists a data structure for such queries in $O(\log \sigma)$ time using $n \log \sigma + o(n \log \sigma)$ bits, called the *wavelet tree* [9]. We use them in this paper.

The proposed succinct data structure is based on the *balanced parentheses encoding* [14], in which a tree is represented by a string B of balanced parentheses in $2n$ bits for the number of nodes, n. A node v is represented by a pair of matching parentheses $'(' \cdots ')'$, and the interval of the $'('$ and $')'$ represents the subtree rooted v.

The *findopen*(B,i)/*findclose*(B,i) is the position of the closing/opening parenthesis that matches a given opening/closing parenthesis $B[i]$. To traverse tree, the findopen/findclose is supported in $O(1)$ time using $2n + o(n)$ bits on the word RAM [14].

A grammar-based compression containing the production rule $X \to ABC$ has an equivalent Chomsky normal form defined by $X \to AX'$ and $X' \to BC$ with the intermediate variable X'. Such normal form is obtained by the compression algorithm in [20]. In this algorithm, the height of each variable produced in i-th loop is exactly i, and in the same loop, the name of variables are assigned from

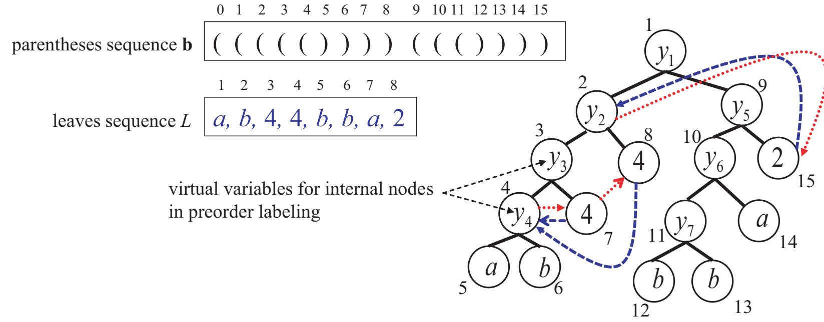

Fig. 4. Succinct representation of ESP: This figure shows a pruned ESP tree T with virtual variables, and the corresponding parentheses sequence $\mathbf{b}$ and the sequences of leaves denoted by L. We can traverse T by $\mathbf{b}$ with a small auxiliary data structure and L is represented by a wavelet tree.

the left occurrence. Using the algorithm, an ESP tree can be simulated by a *full binary tree*, i.e., a binary tree whose internal node has exactly two children. Thus we propose a succinct representation of a full binary tree, which enable us to traverse this tree in $n + o(n)$ bits.

Succinct full binary tree representation: Let T be a full binary tree and $v_1, \ldots v_n$ are the nodes of T in the preorder. Then the succinct representation of T, denoted by $\mathbf{b}$, is defined as $\mathbf{b}[0] =' ('$ as the *virtual root*, $\mathbf{b}[i] =' ('$ if v_i is an internal node, and $\mathbf{b}[i] =')'$ otherwise for $1 \leq i \leq n$. Note that $\mathbf{b}[0, n]$ is also balanced parentheses. Thus, the following operations can be supported in $O(1)$ using the above operations.

1. $\mathit{isleaf}(i) = \mathit{yes}$ iff $\mathbf{b}[i] =')'$, to check if the node i is the leaf,
2. $\mathit{parent}(i) = i - 1$ if $\mathbf{b}[i-1] =' ('$, and $\mathit{parent}(i) = \mathit{findopen}(i-1)$ otherwise, to return the parent of the node i,
3. $\mathit{leftchild}(i) = i + 1$, to return the left child of the node i,
4. $\mathit{rightchild}(i) = \mathit{findclose}(i) + 1$, to return the right child of the node i.

Since the representation have no information about the labeled on the nodes, for traversing the pruned CFG tree T, we must get the variable of any node in T, and access the next occurrence and the leftmost occurrence of any variable in T with other auxiliary data structures.

Let $\mathbf{b}$ be the proposed parentheses sequence for a full binary tree T, and let L be the sequence of labels of leaves in T from the left. L is transformed to its wavelet tree representation of size $n \log n + o(n \log n)$ bits in $O(n \log n)$ time. Each internal node of T has a virtual variable which is defined by the preorder of internal nodes. These data structures are illustrated in Fig. 4. Based on such $\mathbf{b}$ and L, we can support the following operations in $O(1)$ time for $(a), (b)$ and in $O(\log n)$ time for $(c), (d)$.

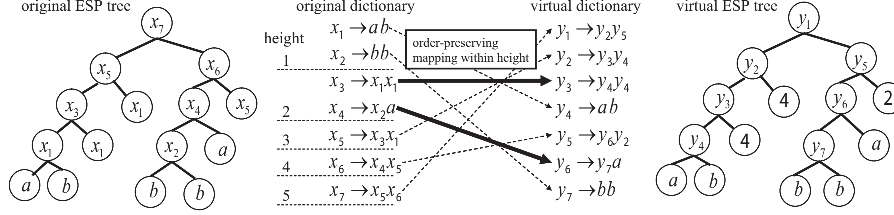

Fig. 5. Virtual variables: This figure shows the correspondence of the original variables and the virtual variables. A given variable x_i, the corresponding virtual node y_i is computed the wavelet tree.

(a) The position in $\mathbf{b}$ corresponding to a virtual variable y_k is $inner(k) = select_{(}(\mathbf{b}, k+1)$,
(b) The virtual variable corresponding to the position i in $\mathbf{b}$ is $variable(i) = rank_{(}(\mathbf{b}, i) - 1$,
(c) The label of a leaf corresponding to the position i in $\mathbf{b}$ is $inleaf(i) = access(L, rank_{)}(\mathbf{b}, i))$, and
(d) The position in $\mathbf{b}$ referred by a leaf i is $refnode(i) = inner(inleaf(i))$.

For any virtual variable y_r, the next i-th occurrence of y_r is obtained by $select_{)}(\mathbf{b}, \ell)$ for $\ell = select_r(L, i)$. For example, in Fig. 4, node 4 is the first occurrence of y_4. From node 4, the next occurrence ($i = 1$) of y_4 is node 7, which is obtained by $select_4(L, 1) = 3$ and $select_{)}(\mathbf{b}, 3) = 7$. Using these operations, we can traverse the pruned ESP tree on its $(\mathbf{b}, L)$-representation. For k leaves, the required space is $2k + o(k)$ bits for $\mathbf{b}$, $k \log k + o(k \log k)$ bits for wavelet tree representation of L, and $o(k)$ bits for each rank/select and other operations. Thus, the total space of this data structure is $k \log k + 2k + o(k \log k)$ bits, and we have the relation $k = n + 1$ for the number n of different variables.

3.3 Time/Space analysis

The remained data structures for the self-indexed grammar-based compression are the function $f(x) = y$ from any original variable x to the corresponding virtual variable y, and the reverse dictionary D^R to get the variable x from a digram yz for the production rule $x \to yz$.

For an ESP tree, the names of original variables are assigned by the order of left-hand preference in the same height, and the names of virtual variables are assigned by the preorder in the depth first search. Thus, the correspondence between two types of variables are preserved in the same height, i.e. for any original variables $x < x'$ in a same height, the corresponding virtual variables y for x and y' for x' satisfy $y < y'$. This relation is illustrated in Fig. 5.

Using this characteristic, we can compute $f(x_i) = y_j$ as follows. For the height h of the ESP tree, let $XH[1, h]$ be the array defined by $XH[k] = i$ iff x_i is the first variable in the height $k = 1, \ldots, h$. The size of XH is $h \log n$ bits. Let $TH[1, n]$

be the array defined by $TH[i] = height(y_i)$ for $i = 1, \ldots, n$. By the wavelet tree representation for TH, we can compute $TH[i] = access(TH, i)$, $rank_c(TH, i)$, $select_c(TH, i)$ in $O(\log h)$ time. The size of TH in wavelet tree is $n \log h + o(n \log h)$ bits. Thus, we can compute $f(x_i) = y_j$ by $f(x_i) = select_c(TH, d)$ for $c = height(x_i)$ and $d = i - TH[d] + 1$. The reverse function is also defined by the rank operation. Therefore, $f(x_i) = y_j$ and $f^{-1}(y_j) = x_i$ are both computed in $O(\log h)$ time and in $n \log h + h \log n + o(n \log h)$ bits.

Finally we analyze the data structure for the reverse dictionary D^R, which is defined by the hash function $H(j, k) = i$ for any production rule $x_i \to x_j x_k$. We can compute $H(j, k)$ in $O(1)$ time with high probability by *Karp-Rabin fingerprints* [11]. However, when we compress a pattern to extract its core, the hash function is already deleted for space-saving. Thus, for the pattern compression, we must restore the information about D^R by the ESP tree itself as follows.

A digram $x'x''$ over the original variables is given, which is appearing in a pattern to be compressed. The aim is to get the head x of $x \to x'x''$. We first transform the digram to $f(x')f(x'') = y_j y_k$ over the virtual variables. If we can find $y_i \to y_j y_k$, then the head x is obtained by $x = f^{-1}(y_i)$ without the reverse dictionary for the original variables.

Using the bucket sort for the two keys j and k, all production rules $y_i \to y_j y_k$ $(1 \leq i \leq n)$ are sorted in $O(n)$ time since the bucket sort is a kind of stable sort. Let $HD[1, n]$ be the array of the heads y_i of the sorted production rules.

If a digram $y_j y_k$ over the virtual variables is given, then we can decide the subarray HD_j of HD corresponding to the production rules $* \to y_j *$ in $O(1)$ time using two bit vectors of length n for each and rank/select operation. For each y_i, its right hand side is computed by the virtual ESP tree in at most $O(\log n)$ time. Thus we can decide the variable y_i which generates $y_j y_k$ for some $HD[\ell] = y_i$ using the binary search on HD_j in $O(\log^2 n)$ time.

Finally we can get the original variable from y_i in in $O(\log h)$ time. The required space for such computation is $n \log n + 2n + o(n)$ bits. Consequently we obtain the following statements for our self-indexed grammar-based compression.

Theorem 1. We can construct the self-indexed grammar-based compression in $O(u + n \log n)$ time and $O(n \log n)$ bits space for the text length u and the number n of different variables. The size of the index is $2n \log n + h \log n + n \log h + 4n + o(n \log n)$ bits, and the time to count occurrences is $O(m \log^2 n + occ_c(\log n \log m + h))$, where $h = O(\log u)$ is the height of the grammar, m is the pattern length, and occ_c is the number of occurrences of the core for the pattern.

4 Experiments

In this section we compare the proposed method to other practical indexes. In our practical implementation, the core is computed for a short prefix of a given pattern. The compressed prefix is restricted at most length 100. This restriction is effective for saving time to find the core.

Table 1. Index Size (MByte)

	ours	LZ-Index	CSArray	FM-Index
ENGLISH (209.7MB)	210.0	323.7	95.3	72.2
DNA (209.7MB)	206.5	236.2	101.0	61.7

Table 2. Construction Time (sec).

	ours	LZ-Index	CSArray	FM-Index
ENGLISH (209.7MB)	134.8	74.3	288.1	623.1
DNA (209.7MB)	104.5	44.3	308.0	420.7

We evaluate the performance of index size, construction time, and time to count occurrences on the benchmarks in Pizza& Chili corpus[1]. The compared methods are LZ-index [15], Compressed Suffix Array [17], and FM-index [8]. The environment is gcc 4.2.1 on 2.53GHz Xeon E5540 CPU and 144GB RAM.

Table 1 shows the index size of two types of texts. For both cases, our index sizes are greater than Compressed Suffix Array and FM-index but smaller than LZ-index. Table 2 shows the construction time for the same texts. Conversely to the result in Table 1, our method is faster than Compressed Suffix Array and FM-index but slower than LZ-index. Thus, by these results, we conclude that the scalability of our method is reasonable.

The search time comparison is shown in Fig. 6. This result tells us that the time to count occurrences for short pattern is slow since a core obtained from a short pattern is small and such a core is frequent, but long pattern search is very fast since sufficiently large cores are obtained from fixed length compression. This result provides us the observation that large cores are found for patterns over length 50 in ENGLISH and length 100 in DNA, and an occurrence of such large core is expected to be very rare.

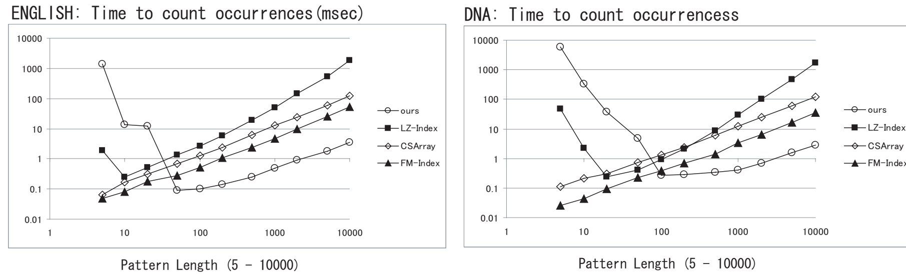

Fig. 6. Time (ms) to count occurrences for ENGLISH and DNA.

[1] http://pizzachili.dcc.uchile.cl/

5 Conclusion

We proposed a self-indexed grammar-based compression using edit sensitive parsing technique [7]. The first theoretical result of self-indexed grammar-based compression is presented in [5]. The algorithm and data structure proposed in this paper are considered as another framework for self-indexing of grammar-based compression, which satisfies the requirements for efficiency theoretically and practically. The proposed succinct data structure for full binary tree can be applied to other grammar-based compressions like Straight-Line Program [5] and Re-Pair [12] for reducing the size of the dictionary.

References

1. Z. Bar-Yossef, T. Jayram, R. Krauthgamer, and R. Kumar. Approximating edit distance efficiently. In *FOCS'04*, pages 550–559, 2004.
2. T. Batu, F. Ergun, J. Kilian, A. Magen, S. Raskhodnikova, R. Rubinfeld, and R. Sami. A sublinear algorithm for weakly approximating edit distance. In *STOC'03*, pages 316–324, 2003.
3. D.A. Benoit, E.D. Demaine, J.I. Munro, R. Raman, V. Raman, and S.S. Rao. Representing trees of higher degree. *Algorithmica*, 43:275–292, 2005.
4. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.
5. F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *MFCS09*, pages 235–246, 2009. to appear in Fundamenta Informaticae.
6. G. Cormode and S. Muthukrishnan. Substring compression problems. In *SODA'05*, pages 321–330, 2005.
7. G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algor.*, 3(1):Article 2, 2007.
8. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS00*, pages 390–398, 2000.
9. R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *SODA04*, pages 636–645, 2004.
10. G. Jacobson. Space-efficient static trees and graphs. In *FOCS'89*, pages 549–554, 1989.
11. R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.
12. N.J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.
13. J.I. Munro. Tables. In *FSTTCS96*, pages 37–42, 1996.
14. J.I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.
15. G. Navarro. Indexing text using the ziv-lempel tire. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
16. W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
17. K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *ISAAC00*, pages 410–421, 2000.

18. S.C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *FOCS'96*, pages 320–328, 1996.
19. H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.
20. H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono. A space-saving approximation algorithm for grammar-based compression. *IEICE Trans. on Information and Systems*, E92-D(2):158–165, 2009.