Journal of Discrete Algorithms 18 (2013) 100–112

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

ESP-index: A compressed index based on edit-sensitive parsing

Shirou Maruyama [a,1], Masaya Nakahara [b,2], Naoya Kishiue [b,3], Hiroshi Sakamoto [b,c,*,1]

[a] Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan
[b] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka-shi, Fukuoka 820-8502, Japan
[c] PRESTO, Japan Science and Technology Agency, 4-1-8 Honcho Kawaguchi, Saitama 332-0012, Japan

ARTICLE INFO

Article history:
Available online 7 August 2012

Keywords:
Grammar-based compression
Self-index
Edit-sensitive parsing
Succinct data structure

ABSTRACT

We propose ESP-index, a self-index based on edit-sensitive parsing. Given a string S, ESP tree is equivalent to a CFG deriving just S, which can be represented as a restricted DAG G. Finding pattern P in S is reduced to embedding the parsing tree of P into G. Adopting several succinct data structures, G is decomposed into two bit strings and a single array, requiring $(1+\varepsilon)n\log n + 4n + o(n)$ bits of space, where n is the number of variables of G and $0 < \varepsilon < 1$. The counting time for the occurrences of P in S is in $O((1/\varepsilon)(m\log n + occ_c \log m \log u)\log^* u)$, where $m = |P|$, $u = |S|$, and occ_c is the number of the occurrences of a maximal common subtree in ESP trees of P and S. With the additional array of $n\log u$ bits of space, our index supports the locating and extracting. Locating time is the same as counting time and extracting time for any substring of length m is $O((1/\varepsilon)(m + \log u))$.

1. Introduction

1.1. The problem

We propose a novel *self-index* that supports the fast *search* and *extract* functionalities for a string without requiring its explicit storage. Given a string S, the *search* answers the counting and locating queries for any pattern P, and the *extract* returns the substring $S[i, j]$ for any interval $[i, j]$. Most self-indexes that have been proposed are based on *sorting* or *parsing*. Grammar-based compression, which includes our method, is categorized as *parsing*, where the string S is represented by the context-free grammar (CFG) that uniquely derives S. Because of the advantage CFG holds for highly redundant text, the ability of self-index based on grammar-based compression has attracted the attention of many researchers in the last decade. Indeed, we can access various types of redundant text in massive repositories, which is one of the most important tasks in managing them with limited resources. We provide a practical self-index, which is especially suited to this problem.

1.2. Our contributions

This study focuses on *Edit-Sensitive Parsing* (ESP) by Cormode and Muthukrishnan [9], which is the derivation tree of a CFG G for S. We develop the self-index on ESP and analyze its performance in theoretical and practical aspects. We use the following notations: u and m are the length of S and P respectively, n is the number of variables in G (i.e., the number of

* Correspondence to: Kyushu Institute of Technology, 680-4 Kawazu, Iizuka-shi, Fukuoka, 820-8502, Japan.
E-mail addresses: maruyama@preferred.jp (S. Maruyama), hiroshi@ai.kyutech.ac.jp (H. Sakamoto).
[1] He moved to Preferred Infrastructure, Inc.
[2] He moved to NaU Data Institute, Inc.
[3] He moved to Hitachi Solutions, Ltd.

1570-8667/$ – see front matter
http://dx.doi.org/10.1016/j.jda.2012.07.009

Table 1
Space and time complexities. σ is the number of alphabet symbols, z is the number of phrases in LZ77 parsing, d is the depth of nesting in LZ77 parsing, and occ is the number of occurrences of P in S. O-notations are omitted in Search and Extract time. n depends on the form of CFG, e.g., only balanced SLPs are allowed in the proposal by Gagie et al. [13].

Source	Space (bits)	Search time	Extract time
Ours	$(1+\varepsilon)n\log n + n\log u + 4n + o(n)$	$(1/\varepsilon)(m\log n + occ_c \log m \log u)\log^* u$	$(1/\varepsilon)(m+\log u)$
[7]	$O(n\log n) + n\log u$	$(m^2 + h(m+occ))\log n$	$(m+h)\log n$
[19]	$2z\log(u/z) + z\log z + 5z\log\sigma + O(z) + o(u)$	$m^2 d + (m+occ)\log z$	md
[13]	$2n\log n + O(z(\log u + \log z \log\log z))$	$m^2 + (m+occ)\log\log u$	$m + \log\log u$

production rules, since for any variable exactly one production rule is defined), ε is any real number in the interval $(0, 1)$, $\log^*$ is the iterated logarithm, and occ_c is the number of occurrences of a *core*.[4] Our contributions are the following.

(1) Fast algorithms for the search and extract on ESP. The time to check if P exists in S as $S[i, i+m] = P$ is $O(\log m \log u \log^* u)$ for a given i.
(2) Compact data structures to simulate the above algorithms on ESP-index. The counting time is $O((1/\varepsilon)(m\log n + occ_c \log m \log u)\log^* u)$ with $(1+\varepsilon)n\log n + 4n + o(n)$ bits of space, the locating time is bounded by the complexity of counting time with $(1+\varepsilon)n\log n + n\log u + 4n + o(n)$ bits of space, and the extracting time is $O((1/\varepsilon)(m+\log u))$ with the same space assuming the standard word random access model.
(3) Implementation of ESP-index. Performance is examined for both natural language texts and biological sequences.

We also compare ESP-index to recent self-indexes [7,13,19] based on SLP and LZ77 in Table 1. We mention the advantages of the related works. The proposal by Claude and Navarro [7] is applicable to all grammar-based compressions. Among them Re-Pair [20] achieves potentially better compression than LZ78 [42] and BWT [11]. The proposal by Kreft and Navarro [19] is based on LZ77 [41]. Rytter showed that the number of phrases in LZ77 parsing is smaller than that of variables of any CFG equivalent to it [31]. The proposal by Gagie et al. [13] achieves the fastest search time under the assumption that a balanced SLP is given. Besides, we can construct a balanced SLP from any string adopting Rytter's technique [31].

Our contributions are further summarized below. We first formalize the pattern matching problem on ESP. For the derivation trees T_S of S and T_P of P, we consider an embedding of T_P into T_S; that is, P appears in S as a substring. T_P is decomposed into a sequence of subtrees, whose roots are labeled by variables $x_1, \ldots, x_k$. This sequence is an *evidence* of any occurrence of P in T: S contains an occurrence of P iff there is a sequence $v_1, \ldots, v_k$ of nodes in T_S such that v_i is labeled by x_i and the subtrees rooted by v_i, v_{i+1} are *adjacent* in this order. Note that P itself, that is, the sequence of the leaves, is invariably one of evidences. The aim is to find as short evidence as possible and to embed it into T_S efficiently.

Next, we develop several compact data structures for the proposed algorithms. An ESP represented by a restricted CFG is equivalent to a DAG G where every internal node has exactly two children. G is then decomposed into two *in-branching* spanning trees. The one called the left tree is constructed by the left edges, whereas the other, called the right tree, is constructed by the right edges. Both left and right trees are encoded by LOUDS [16], one of the succinct data structures supporting many operations for ordered trees. Further, correspondence among the nodes of the trees is stored in a single array. Adding the data structure for the permutation [26] over the array makes it possible to traverse G.

We finally design the mechanism of the binary search on the variables. The compression algorithm must refer to a function, called *reverse dictionary* to get the name of any variable associated with a digram. For example, if a production rule $Z \to XY$ exists, any occurrence of the digram XY, which is determined to be replaced, should be replaced by the same Z. However, the hash function $H(XY) = Z$ when preprocessing P compels the index size to be increased immoderately. Thus our algorithm shall obtain the name of any variable for P from the compressed G itself. Adopting the above succinct data structures, we can develop the index to answer the counting query. By storing the length of each variable in array, we can solve the locating and extracting problems because ESP tree is inherently balanced.

We also implement ESP-index and compare its performance with other practical self-indexes [24,28,33] under several reasonable parameters. Beyond that, we give the experimental result for maximal common substring detection. These common substrings are obtained to find the common variables in T_S and T_P. We conclude that the proposed index is efficient enough for cases where the pattern is long.

1.3. Related work

The framework of grammar-based compression was introduced explicitly by Kieffer and Yang [17]. For the leading grammar-based compression algorithms [1,17,18,20,40,42], their approximation ratios to the optimum grammar size were investigated and the NP-hardness of the minimum CFG problem was proved by Lehman and Shelat [22]. The $O(\log(u/g))$-approximation ratio, the smallest ratio at the present time, was achieved by Charikar et al. [4] and Rytter [31] almost simultaneously, and another algorithm for the same ratio was presented by Sakamoto [37], where g is the size of minimum

[4] This is nearly equal to the number of the occurrences of P, when P is long.

CFG. As mentioned by Lehman and Shelat [22], this approximation ratio is, however, unlikely to be significantly improved due to the relationship between the minimum CFG and the shortest *addition chain*. The relation between ESP and minimum CFG was investigated by Sakamoto et al. [38] and an online algorithm for simplified ESP was recently presented by Maruyama et al. [25].

On the other hand, various self-indexes have been developed in parallel with data compression progress. The first self-indexes were built on BWT [11] and on suffix array [14,23,32]. Afterwards, self-indexes based on grammar-based compression were produced, e.g., LZ78 [2,12,30], LZ77 [13,19], and SLP [7]. As shown by Lehman [21], among these grammar-based compressions, only SLP and LZ77 are potentially powerful.

Succinct representation of trees is one of the most important techniques in our study. LOUDS was introduced by Jacobson [16], and improved by Clark [6] and Delpratt et al. [10] to support a number of operations. Although we cannot refer to all relevant data structures, recent results were presented by Sadakane and Navarro [34] and Bille et al. [3].

ESP is closely related to *deterministic coin tossing* by Cole and Vishkin [8], which is a technique in parallel algorithms applicable to string [35,36]. ESP was originally introduced to approximate a variant of the string edit distance problem where a moving operation for any substring with unit cost is permitted. For instance, $a^n b^n$ is transformed to $b^n a^n$ by a single operation. The edit distance with move is NP-hard, and has proved to be $O(\log u)$-approximable by Shapira and Storer [39]. The harder problem, edit distance *matching* with move, was also proved to be approximable within almost $O(\log u)$ ratio by embedding of string into L_1 vector space using ESP [9]. Our pattern matching problem is a restricted version of the embedding problem.

2. Definitions

2.1. Grammar-based compression

The set of all strings over the alphabet Σ is denoted by Σ^*. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. For a symbol a, a^+ denotes the set $\{a^k \mid k \geqslant 1\}$, and a^k is called a *repetition* if $k \geqslant 2$. $S[i]$ and $S[i, j]$ denote the i-th symbol of S and the substring from $S[i]$ to $S[j]$, respectively. We let $\log^{(1)} u = \log u$, $\log^{(i+1)} u = \log\log^{(i)} u$, and $\log^* u = \min\{i \mid \log^{(i)} u \leqslant 1\}$. In practice, we can consider $\log^* u$ to be constant, since $\log^* u \leqslant 5$ for $u \leqslant 2^{65536}$.

We assume that any CFG G is *admissible*, i.e., G derives just one string and for each variable X, exactly one production rule $X \to \alpha$ exists. The set of variables is denoted by $V(G)$, and the set of production rules, called dictionary, is denoted by $D(G)$. We also assume that, for any $\alpha \in (\Sigma \cup V(G))^*$, at most one $X \to \alpha \in D(G)$ exists. We use V and D instead of $V(G)$ and $D(G)$ when G is omissible. The string derived by D from a string $S \in (\Sigma \cup V)^*$ is denoted by $S(D)$. For example, when $S = aYY$ and $D = \{X \to bc, Y \to Xa\}$, we obtain $S(D) = abcabca$. For $X \in V$, $|X|$ denotes $|X(D)|$, that is, the length of the string derived by D from X. Additionally, $|V|$ denotes the cardinality of the set V.

2.2. Edit-sensitive parsing (ESP)

A string $w \in (\Sigma \cup V)^*$ is uniquely partitioned into $w_1 x_1 w_2 x_2 \cdots w_k x_k w_{k+1}$ by its maximal repetitions, where each x_i is a maximal repetition of a symbol in $\Sigma \cup V$ and w_i is (possibly empty) string containing no repetition. There are nonoverlapping substrings of three types: a repetition x_i is Type1, a substring w_i of length at least $\log^* |w|$, not of the aforementioned Type1, is Type2, and a remaining w_i, not of Type1, is Type3. If $|w_i| = 1$, this is attached to x_{i-1} or x_i, with preference x_{i-1} when both cases are possible. Thus, if $|w| \geqslant 2$, any metablock is longer than or equal to two.

Let S be a metablock and D be a current dictionary starting with $D = \emptyset$. We set $ESP(S, D) = (S', D \cup D')$ for $S'(D') = S$ and S' described as follows:

1. In case S is Type1 or Type3 of length $k \geqslant 2$,
   (a) If k is even, let $S' = t_1 t_2 \cdots t_{k/2}$, and make $t_i \to S[2i-1, 2i] \in D'$.
   (b) If k is odd, let $S' = t_1 t_2 \cdots t_{(k-3)/2} t$, and make $t_i \to S[2i-1, 2i] \in D'$, $t \to S[k-2]t'$, and $t' \to S[k-1, k] \in D'$, where t_0 denotes the empty string for $k = 3$.
2. In case S is Type2,
   (c) for the partitioned $S = s_1 s_2 \cdots s_k$ $(2 \leqslant |s_i| \leqslant 3)$ by *alphabet reduction* [9], let $S' = t_1 t_2 \cdots t_k$, and make $t_i \to XY \in D'$ if $s_i = XY$ and make $t_i \to XY'$, $Y' \to YZ \in D'$ if $s_i = XYZ$.

Cases (a) and (b) denote the typical *left aligned parsing*, e.g., if $S = a^6$, $S' = x^3$ and $x \to a^2 \in D'$, and if $S = a^9$, $S' = x^3 y$ and $x \to a^2$, $y \to ay'$, $y' \to aa \in D'$.

For case (c), we give only the essence of the alphabet reduction by Cormode and Muthukrishnan [9]. For $S[i]$ and $S[i-1]$ in their binary representations, let p be the position of the least significant bit in which $S[i]$ differs from $S[i-1]$, and let $bit(p, S[i]) \in \{0, 1\}$ be the value of $S[i]$ at the p-th position. Then $label(i) = 2p + bit(p, S[i])$ is defined,[5] and the maximal values in $label(1), \ldots, label(|S|)$ indicate the delimiters in the partitions in case (c).

[5] $bit(p, S[i])$ is the trick to obtain the condition of $label(i) \neq label(i-1)$ for any i.

Finally, we define the ESP for $S \in (\Sigma \cup V)^*$ partitioned into $S_1 S_2 \cdots S_k$ by k metablocks; $ESP(S, D) = (S', D \cup D') = (S'_1 \cdots S'_k, D \cup D')$, where D' and each S'_i satisfying $S'_i(D') = S_i$ are defined in the above.

Iteration of the ESP is defined as $ESP^i(S, D) = ESP^{i-1}(ESP(S, D))$. In particular, $ESP^*(S, D)$ denotes the iterations until $|S| = 1$. After computing $ESP^*(S, D)$, the final dictionary represents the rooted ordered binary tree deriving S, which is denoted by $ET(S)$. We recall some useful properties of ESP in the following.

Lemma 1. *(Cormode and Muthukrishnan [9].) The depth of the tree $ET(S)$ is $O(\log |S|)$ and $ET(S)$ can be computed in time $O(|S| \log^* |S|)$ time.*

Lemma 2. *(Cormode and Muthukrishnan [9].) Let S be Type2 and $S = s_1 s_2 \cdots s_k$ be its partition by alphabet reduction. If $S\alpha$ is Type2 and its partition is of $S\alpha = t_1 t_2 \cdots t_n$, we have $t_i = s_i$ for all $1 \leqslant i \leqslant k - 5$. If αS is Type2 and its partition is of $\alpha S = t'_1 t'_2 \cdots t'_m$, we have $t'_{m-j} = s_{k-j}$ for all $0 \leqslant j \leqslant k - \log^* |S| - 5$.*

2.3. Pattern embedding problem

We focus on the problem to find occurrences of P in S by embedding P into a parsing tree. For the parsing tree $T_S = ET(S)$ by D_S and a pattern P, the key idea is to compute $ESP(P, D_S)$ and find an embedding of the resulting tree T_P into T_S. The label of node v is denoted by $L(v) \in \Sigma \cup V$, and $L(v_1 \cdots v_k) = L(v_1) \cdots L(v_k)$. Let $yield(v)$ denote the substring of S derived by $L(v)$, and $yield(v_1 \cdots v_k) = yield(v_1) \cdots yield(v_k)$. Note that T_S and T_P are ordered binary trees.

In an ordered binary tree, the parent and left/right child of node v are denoted by $parent(v)$ and $left(v)/right(v)$, respectively. For an internal node v, the edges $(v, left(v))$ and $(v, right(v))$ are called left edge and right edge. Node v is called the *lowest right ancestor* of x, denoted by $lra(x)$, if v is the lowest ancestor of x such that the path from v to x contains at least one left edge. If x is a rightmost descendant, $lra(x)$ is undefined. Otherwise, $lra(x)$ uniquely exists. The lowest left ancestor of x, denoted by $lla(x)$, is similarly defined. Let v_1, v_2 be different nodes, if $lra(v_1) = lla(v_2)$, v_1, v_2 are *adjacent* in this order, and v_1 is left adjacent to v_2 (or v_2 is right adjacent to v_1).

Fact 1. *v_1 is left adjacent to v_2 iff v_2 is a leftmost descendant of $right(lra(v_1))$, and v_2 is right adjacent to v_1 iff v_1 is a rightmost descendant of $left(lla(v_2))$.*

Let $v_1, \ldots, v_k$ be a sequence of nodes. If any v_i, v_{i+1} are adjacent in this order and z is the lowest common ancestor of v_1, v_k denoted by $z = lca(v_1, v_k)$, the sequence is said to be embedded in z, denoted by $(v_1, \ldots, v_k) \prec z$. If $yield(v_1 \cdots v_k) = P$, z is called the *occurrence node of* P.

Definition 1. A string $Q \in (\Sigma \cup V)^*$ of length k satisfying the following condition is called an *evidence of* P: a node z in T_S is an occurrence node of P iff there is a sequence $v_1, \ldots, v_k$ such that $(v_1, \ldots, v_k) \prec z$, $yield(v_1 \cdots v_k) = P$, and $L(v_1 \cdots v_k) = Q$.

For any T_S and P, at least one evidence of P exists because P itself is an evidence of P. We propose an algorithm to find as short evidence as possible for given T_S and P; we also propose another algorithm to find all occurrences of P in S for the obtained evidence.

3. Algorithms and data structures

We propose two algorithms: one generates an evidence Q of pattern P with respect to $(S', D_S) = ESP^*(S, D)$. The other algorithm finds the occurrence node z of P such that $(v, v_1, \ldots, v_k) \prec z$ for the obtained Q and a node v in T_S satisfying $L(v) = Q[1]$. Finally, we propose the data structures to access the next node v' satisfying $L(v') = L(v)$ for each v in T_S. By this, the number of occurrences of P are found to check if $(v, v_1, \ldots, v_k) \prec z$ for all candidates v satisfying $L(v) = Q[1]$.

3.1. Finding evidence of pattern

The algorithm to generate evidence Q of pattern P is described in Fig. 1. An outline follows. Input is a pair of P and D_S. P is partitioned into $P = P_1 P_2 \cdots P_\ell$ by some metablocks P_is. Let $P = \alpha\beta\gamma$ for $\alpha = P_1$, $\gamma = P_\ell$, and $\beta = P_2 \cdots P_{\ell-1}$. Depending on the types of metablocks, P is further partitioned into $P = \alpha_p \alpha_s \beta \gamma_p \gamma_s$. The algorithm then updates current $Q = Q_p Q_s$ by $Q_p \leftarrow Q_p \alpha_p$ and $Q_s \leftarrow \gamma_s Q_s$, and current P by $P \leftarrow P'$ such that P' is the string produced by $ESP(\alpha_s \beta \gamma_p, D_S)$. This is continued until P is entirely deleted.

Lemma 3. *Let Q be the output string of Find_evidence(P, D_S) and let $Q = Q_1 \cdots Q_k$, $Q_i \in q_i^+$ for some symbol q_i where $q_i \neq q_{i-1}, q_{i+1}$. Then Q is an evidence of P satisfying $k = O(\log m \log^* u)$.*

Find_evidence(P, D_S)
let $D' \leftarrow \emptyset$, $Q_p = Q_s$ be the empty string;
while$(|P| > 1)$**{** /* **appending prefix and suffix of** P **to** Q */
  let $P = \alpha\beta\gamma$ for the first/last metablock α/γ; /* **possibly** $|\beta\gamma| = 0$ */
  $(P', D_S \cup D') \leftarrow ESP(P, D_S)$, where
    $P' = \alpha'\beta'\gamma'$, $\alpha'(D') = \alpha$, $\beta'(D') = \beta$, $\gamma'(D') = \gamma$;

  if(α is Type1 or 3) $\{Q_p \leftarrow Q_p\alpha$, remove the prefix α' of P';$\}$
  else{
    let $\alpha = \alpha_1 \cdots \alpha_\ell$, $\alpha' = p_1 \cdots p_\ell$, where $p_i \to \alpha_i \in D'$;
    $Q_p \leftarrow Q_p\alpha_1 \cdots \alpha_j$, remove the prefix $p_1 \cdots p_j$ of P' for $j = \min(\log^* u + 5, \ell)$;
  }/* **the bound** j **for prefix is guaranteed by Lemma 2** */

  if(γ is Type1 or 3) $\{Q_s \leftarrow \gamma Q_s$, remove the suffix γ' of P';$\}$
  else{
    let $\gamma = \gamma_1 \cdots \gamma_r$, $\gamma' = q_1 \cdots q_r$, where $q_i \to \gamma_i \in D'$;
    $Q_s \leftarrow \gamma_{r-j} \cdots \gamma_r Q_s$, remove the suffix $q_{r-j} \cdots q_r$ of P' for $j = \min(5, r)$;
  }/* **the bound** j **for suffix is also from Lemma 2** */
  $P \leftarrow P'$, $D_S \leftarrow D_S \cup D'$, $D' \leftarrow \emptyset$; /* **update** */
} **if**$(|Q_p Q_s| > 0)$ return $Q \leftarrow Q_p Q_s$; **else** return P;

Fig. 1. Algorithm to find the evidence of P with D_S.

Proof. When P is partitioned into metablocks, let α and γ be the first and last metablocks respectively, and let $P = \alpha\beta\gamma$ for a substring β. If α, γ are Type1 or 3, any occurrence of β inside $S[n, m] = \alpha\beta\gamma$ is transformed into a same β' in the *while* loop. Thus, $\alpha\beta'\gamma$ is an evidence of P and $\alpha\gamma$ contains $O(\log^* u)$ different symbols.

If α, γ are Type2, by alphabet reduction, the prefix α of P is partitioned into $\alpha = \alpha_1 \cdots \alpha_\ell$ $(2 \leqslant |\alpha_i| \leqslant 3)$. Then $j = \min(\log^* u + 5, \ell)$ is determined and $\alpha_1 \cdots \alpha_j$ is appended to current Q, and a short suffix of γ is similarly appended to Q. By Lemma 2, for any $S = x\beta y$ $(|x| \geqslant \log^* u + 5, |y| \geqslant 5)$, any occurrence of β inside $S[n, m] = x\beta y$ is transformed into the same β' in the *while* loop. The selected j for α satisfies either $|\alpha_1 \cdots \alpha_j| \geqslant \log^* u + 5$ or $\alpha_1 \cdots \alpha_j = \alpha$, and similarly, the selected j for β satisfies either $|\gamma_{r-j} \cdots \gamma_r| \geqslant 5$ or $\gamma_{r-j} \cdots \gamma_r = \gamma$. Thus we can obtain the evidence $\alpha_1 \cdots \alpha_j \beta' \gamma_{r-j} \cdots \gamma_r$ of P. The other cases are similarly proved, in which one of α, γ is Type1 or 3 and the other is Type2.

Applying the above analysis to β' until it becomes the empty symbol, we obtain the final Q as the evidence of P. The number of iterations of $ESP(P, D) = (P', D \cup D')$ is $O(\log m)$ because $|P'| \leqslant |P|/2$. In the i-th iteration, when the current $Q = Q_p Q_s$ is updated to $Q_p \alpha_p \gamma_s Q_s$, the string $\alpha_p \gamma_s$ contains $O(\log^* u)$ different symbols. Therefore we conclude $k = O(\log m \log^* u)$. □

3.2. Finding pattern occurrence

The algorithm to find an occurrence node of P is described in Fig. 2. Using *Find_evidence*(P, D_S) as subroutine, *Find_pattern*(Q, D_S, v, T_S) finds the embedding $(v, v_1, \ldots, v_\ell) \prec z$ satisfying $yield(v v_1 \cdots v_\ell) = P$ for the fixed v. By Lemma 3, such a z exists iff z is the occurrence node of P. We show the correctness and the time complexity of this algorithm.

Lemma 4. *Find_pattern*(Q, D_S, v, T_S) *outputs node* z *in* T_S *iff* z *is the occurrence node of* P *satisfying* $(v, v_1, \ldots, v_\ell) \prec z$ *for some* $v_1, \ldots, v_\ell$ *and the fixed* v *in* T_S. *The time complexity is* $O(\log m \log u \log^* u)$.

Proof. For any node v in T_S and $q \in \Sigma \cup V$, we can check if $(v, v') \prec z$ and $L(v') = q$ for some nodes v', z in $O(\log u)$ time since v' must be a leftmost descendant of $right(lra(v))$ and the height of T_S is $O(\log u)$.

Let $Q = Q_1 \cdots Q_k$ and $Q_i \in q_i^+$ for some $q_i \in \Sigma \cup V$. We assume that Q contains no repetition and $(v_1, \ldots, v_j) \prec z$ is already found for the prefix $Q_1 \cdots Q_j = q_1 \cdots q_j$. From $(v_j, v_{j+1}) \prec z'$ and $L(v_{j+1}) = q_{j+1}$, we obtain $(v_1, \ldots, v_{j+1}) \prec lca(z, z')$ in $O(\log u)$ time since z, z' must be in a same path. By Lemma 3, the embedding of such Q of length at most $O(\log m \log^* u)$ from v is computed in $O(\log m \log u \log^* u)$ time.

Generally, let $Q_j = q^\ell$ for some symbol q and $\ell \geqslant 2$. In ESP, any repetition is transformed into a shorter string by the left aligned parsing, and this transformation is continued as long as the resulting string contains a repetition. Thus, by T_S, the occurrence $S[s, t] = q^\ell$ is partitioned into $S[s, t] = S[s_1, t_1]S[s_2, t_2] \cdots S[s_k, t_k]$ such that $|S[s_i, t_i]| = 2^{\ell_i} \geqslant 1$, each $S[s_i, t_i]$ is derived by the maximal complete binary tree rooted by v_i, and $k = O(\log \ell)$. Let $[s_i, t_i]$ be the longest interval. Recall that all symbols in the current string are replaced by the next iteration of ESP. By this characteristic, when $S[s_i, t_i]$ is transformed into $S'[j]$, the digram $S'[j-2, j-1]$ derives the string containing $S[s_1, t_1] \cdots S[s_{i-1}, t_{i-1}]$ as its proper suffix. If not, $S'[j-2, j] = X^3$ for some variable X because the left aligned parsing is determined by the height only. In the next loop, a greater tree is produced for the repetition. This, however, contradicts the assumption that $S[s_i, t_i]$ is the longest segment derived by a complete binary tree. Thus, we can check if $(v_1, \ldots, v_{i-1}) \prec v_p$ for some v_p in $O(\log \ell + \log u) = O(\log m + \log u)$ time. The time to check if $(v_p, v_i) \prec z$ is $O(\log u)$. Hence, the time to embed q^ℓ is $O(\log m + \log u)$. Therefore, the time complexity is $O((\log m + \log u) \log m \log^* u) = O(\log m \log u \log^* u)$. □

```
Find_pattern(Q, D_S, v, T_S) /* L(v) = Q[1] */
let Q = Q_1 ··· Q_k, Q_i is a repetition of q_i ∈ Σ ∪ V, q_i ≠ q_{i−1}, q_{i+1};
initialize j ← 1, z ← v;        /* current block Q_j and embedding in z */
if(|Q| = 1) return z;
while(j ⩽ k){
    if(|Q_j| = 1){ /* block Q_j is just one symbol */
        if((v, v') ≺ z', L(v') = q_{j+1} for some v', z' in T_S)
            v ← v', z ← lca(z, z'), j ← j + 1;
        else return 0;
    }
    else{ /* block Q_j is a repetition */
        ℓ ← |Q_j|;
        while(ℓ > 0){ /* find maximal complete binary tree parsing q_j^ℓ */
            if((v, v') ≺ z', L(v') = q_j for some v', z' in T_S){
                let v_a be the highest ancestor of v' satisfying X_0 = L(v_a), X_d = q_j,
                    X_0 → X_1^2, ..., X_{d−1} → X_d^2 ∈ D_S, 1 ⩽ 2^d ⩽ ℓ;
                v ← v_a, z ← lca(z, z'), ℓ ← ℓ − 2^d;
            }/* next complete binary tree until whole q_j^ℓ is covered */
            else return 0;
        }j ← j + 1;
    }
}return z;
```

Fig. 2. Algorithm to find occurrence node of P from the fixed node v.

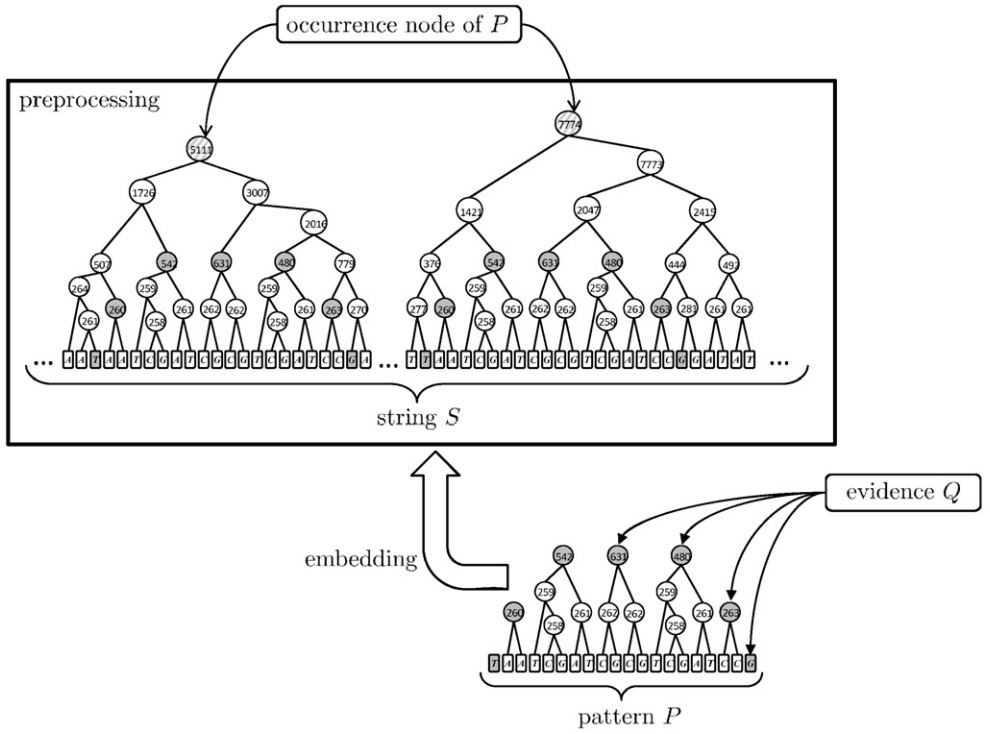

Fig. 3. Outline of the pattern embedding algorithm. S is preprocessed by ESP, the evidence Q formed by the symbols is obtained by *Find_evidence*(P, D_S), and *Find_pattern*(Q, D_S, v, T_S) tries to embed Q from the indicated node v.

The sketch of the *Find_evidence*(P, D_S) and *Find_pattern*(Q, D_S, v, T_S) is shown in Fig. 3.

3.3. Data structures

We develop the compact data structures for *Find_pattern*(Q, D_S, v, T_S) to access the next occurrence of v with $L(v) = Q[1]$. From here on, we attack two issues: one is the compact representation of DAG by its decomposition; the other is the simulation of the *reverse dictionary*. We first treat the decomposition of DAG G, which is the graph representation of D_S. Introducing single super sink v_0 together with the left and right edges from any sink of G to v_0, G is modified to have the unique source and sink. We consider only the such modified G.

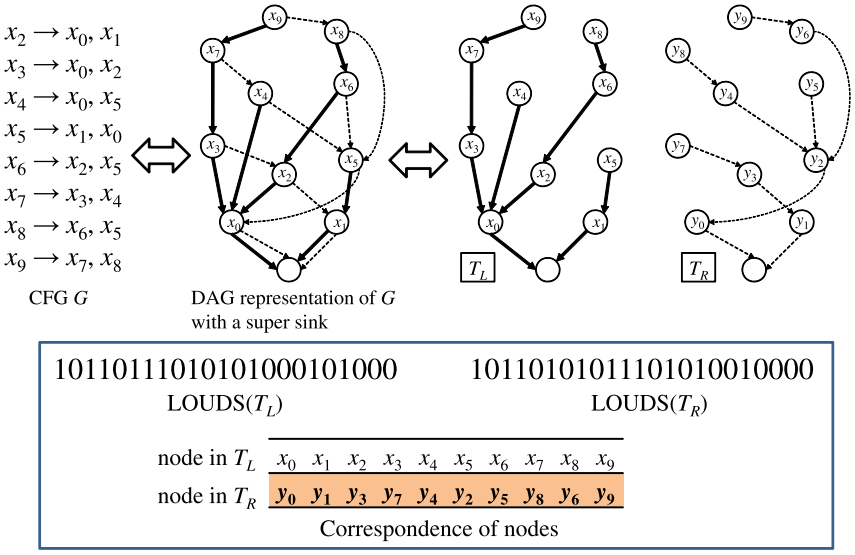

Fig. 4. Compact representation of a CFG by the left tree and right tree in LOUDS bit-strings with the permutation array.

Fact 2. *For any in-branching spanning tree of G, the graph defined by the remaining edges is also an in-branching spanning tree of G.*

The in-branching spanning tree of G constructed by the left edges only is called the *left tree* of G and denoted by T_L. The complementary tree, denoted by T_R, is similarly defined. Examples of G, T_L, and T_R are shown in Fig. 4 with compact representation proposed below.

When a DAG G is decomposed into T_L and T_R, G is represented by the succinct data structures for ordered trees and permutations. The bit-string by LOUDS [10] for an ordered tree is defined below. We visit any node in level-order from the root. As we visit a node v with $d \geqslant 0$ children, we append 1^d0 to the bit-string beginning with the empty string. Finally, we add 10 as the prefix corresponding to the imaginary root, which is the parent of the root of the original tree. For the n-node tree, LOUDS uses $2n + o(n)$ bits of space to support constant time access to the parent, the i-th child, and the number of children of a node, required for our ESP-index.

By the trees T_L, T_R and the correspondence of nodes between them, we can traverse G without explicit link structure. For this purpose, we employ the succinct data structure for permutation [26]. For any permutation π of $N = (1, 2, \ldots, n)$, using $(1 + \varepsilon)n \log n + o(n)$ bits of space, this data structure supports access to $\pi[i]$ in $O(1)$ time and to $\pi^{-1}[i]$ in $O(1/\varepsilon)$ time. For instance, if $\pi = (2, 5, 1, 3, 4)$, then $\pi[3] = 1$ and $\pi^{-1}[5] = 2$; that is, $\pi[i]$ is the i-th member of π and $\pi^{-1}[i]$ is the position of the member i. For each node i in LOUDS(T_L) and the corresponding node j in LOUDS(T_R), we can get the correspondence by $\pi[i] = j$ and $\pi^{-1}[j] = i$.

We introduce another preprocess for G. In each iteration $ESP(S, D) = (S', D \cup D')$, we rename all variables; by sorting all production rules $X \to X_iX_j \in D'$ by (i, j), if the rank of $X \to X_iX_j$ is k, all occurrences of X in D' and S' are renamed to X_k, e.g.,

$$D' = \{X_1 \to ab, X_2 \to bc, X_3 \to ac, X_4 \to aX_2\} \quad \text{and} \quad S' = X_1X_2X_3X_4$$

are renamed to

$$D' = \{X_1 \to ab, X_2 \to ac, X_3 \to aX_4, X_4 \to bc\} \quad \text{and} \quad S' = X_1X_4X_2X_3.$$

By this trick, the variable X_i in G coincides with node i in LOUDS of T_L because they are both named in level-order. The variables in Fig. 4 are already renamed. Adopting this, the size of the array required for node correspondence is reduced to $n \log n$ bits of space.

Finally we simulate the reverse dictionary for pattern compression. When computing $ESP^*(S, D)$, the naming function $H_S(XY) = Z$ for $Z \to XY \in D$ is realized by a hash function. However, when compressing P, we must get the name Z by G only because our index does not explicitly contain H_S. By preprocessing, the variable X_k corresponds to the rank of its left-hand side X_iX_j for $X_k \to X_iX_j$. Conversely, given X_i, the children of X_i in T_L are already sorted by the ranks of their parents in T_R. This idea is summarized in Fig. 5.

Because LOUDS supports constant time access to the number of children and the i-th child, $H_S(X_iX_j) = X_k$ is obtained by the binary search in the following time complexity.

Lemma 5. *For the maximum degree k of T_L, the function $H_S(XY) = Z$ is computable in $O((1/\varepsilon) \log k) = O((1/\varepsilon) \log n)$ time.*

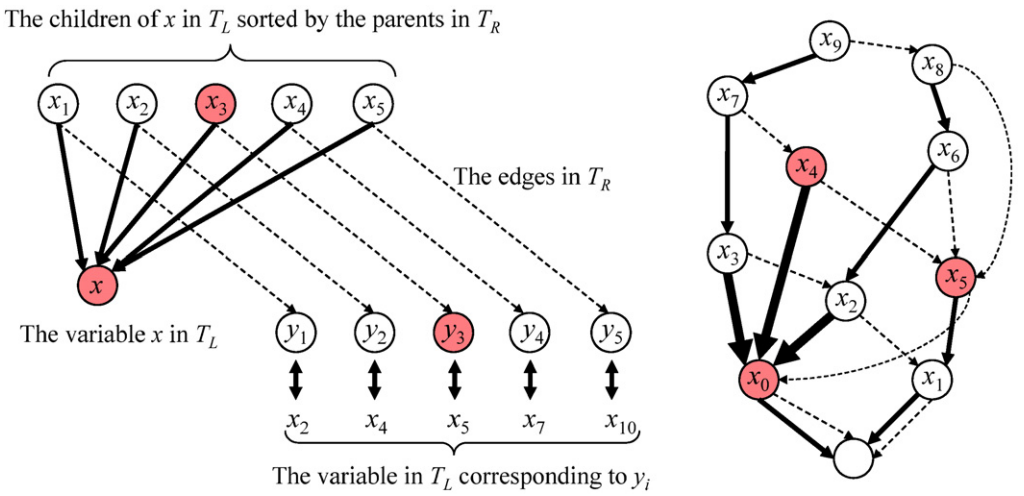

Fig. 5. Binary search for $H(XY) = Z$ on the nodes of T_L. For each node x in T_L, the children x_i of x are already sorted by the variables in T_L corresponding to the parents of x_i in T_R.

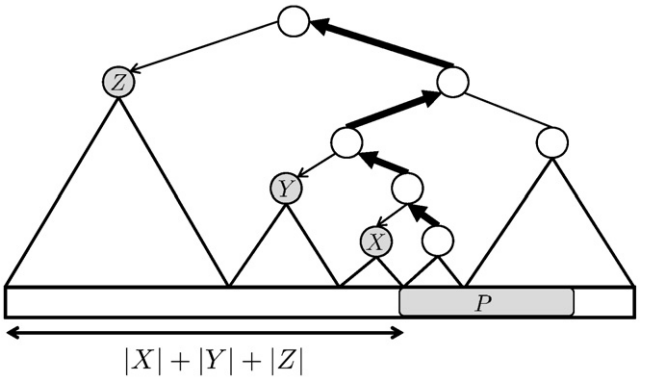

Fig. 6. Locating of pattern P.

Theorem 1. *The size of ESP-index is* $(1+\varepsilon)n\log n + 4n + o(n)$ *bits of space. The number of occurrences of* P *in* S *can be evaluated in* $O((1/\varepsilon)(m\log n + occ_c \log m \log u)\log^* u)$ *time, where* occ_c *is the number of occurrences of a maximal common subtree in* T_S *and* T_P.

Proof. We can modify *Find_pattern* to find $(v_1, \ldots, v_k) \prec z$ from v_k to v_1. Starting with v_ℓ labeled by the symbol q obtained in the final loop in ESP for P, we can check if there is a node z satisfying $(v_1, \ldots, v_\ell) \prec z_1$, $(v_\ell, \ldots, v_k) \prec z_2$, and $(v_1, \ldots, v_k) \prec lca(z_1, z_2) = z$. This derives the time bound. □

To store the length of any variable X, that is, the length of substring encoded by X, we can support the locating and extracting. Locating process is illustrated in Fig. 6.

Theorem 2. *With an auxiliary array of* $n\log u + o(n)$ *bits of space, ESP-index supports locating in the same time complexity as the case of counting, and extracting* $S[i, i+m]$ *in* $O((1/\varepsilon)(m + \log u))$ *time.*

Proof. We first show the locating time. By Theorem 1, we can find the leaf v_1 in T_S satisfying $(v_1, \ldots, v_k) \prec z$ for the occurrence node z of P. If v_1 is the i-th leaf of T_S, $i-1$ is the required position. For the path $(v_1, z_1, \ldots, z_h)$ from v_1 to the root z_h, we can compute $i - 1 = \sum_{j=1}^{r} \|z_j\|$ where

$$\|z_j\| = \begin{cases} |left(z_j)|, & left(z_j) \notin \{z_1, \ldots, z_h\}, \\ 0, & \text{otherwise.} \end{cases}$$

The time to count up all the $\|z_j\|$ is $O(h) = O(\log u)$ because the depth of T_S is $O(\log u)$. Thus, the locating time is the same as the counting time.

We consider the extraction of $S[i, i+m]$ such that v_j is the leaf of T_S whose label is $S[i+j]$ $(1 \leqslant j \leqslant m)$. Let $z = lca(v_1, v_m)$ and z_h be the root again. Suppose T_z is the induced subgraph of T_S consisting of all nodes in the path from

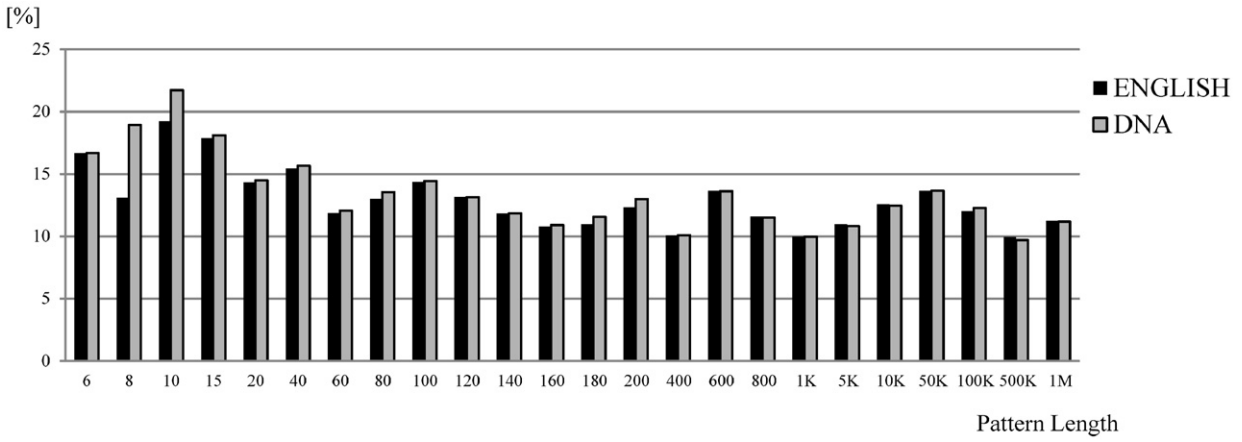

Fig. 7. Ratio of the core length to the pattern length.

z to v_j for $1 \leqslant j \leqslant m$. The algorithm can access the parent, left child, and right child of the current node in $O(1/\varepsilon)$ time. Thus $S[i, i+m]$ is extracted in $O((1/\varepsilon)(t_1+t_2))$ time, where t_1 is the time to traverse the path from z_h to v_1 and t_2 is the time to traverse T_z from v_1. Clearly, $t_1 = O(\log u)$. Let N_k be the number of nodes in T_z produced in the k-th loop in ESP $(0 \leqslant k \leqslant h)$. By the definition of ESP, $|N_{k+1}| \leqslant (2/3)|N_k|$. It derives $t_2 = O(\log u + |N_0| + \cdots + |N_h|) = O(\log u + m)$. □

Finally, for practical use, we propose the technique to save the space of the length of variables, called *sparse* ESP-index. Let V_i be the set of variables produced in the i-th loop for $1 \leqslant i \leqslant h$. We prepare the array A to store the length of any variable in V_i for all even i. Additionally, we make the following bit-string. The string $\mathbf{B} = B_1 \cdots B_h$ is defined by

$$B_i = \begin{cases} 1^{|V_i|}, & i \equiv 0 \pmod 2, \\ 0^{|V_i|}, & i \equiv 1 \pmod 2. \end{cases}$$

The other string $\mathbf{b}$ is defined by $\mathbf{b}[j] = 1$ if V_i exists such that $X, Y \in V_i$ for some $X_j \to XY$ (possibly $X = Y$), and $\mathbf{b}[j] = 0$ otherwise. If $\mathbf{B}[j] = 1$, $|X_j| = A[rank_{\mathbf{B}}(1, j)]$, where $rank_{\mathbf{B}}(1, j)$ is the number of 1 in $\mathbf{B}[1, j]$. Answering this query is $O(1)$ time using $n + o(n)$ bits of space for any binary string of length n [16]. Let $\mathbf{B}[j] = 0$. In case of $\mathbf{b}[j] = 1$, $|X_j| = |X| + |Y|$ for the recorded length $|X|, |Y|$ in A. In case of $\mathbf{b}[j] = 0$, there exist $X_j \to XX'$ and $X' \to YZ$ such that $|X_j| = |X| + |Y| + |Z|$ for the recorded length $|X|, |Y|, |Z|$ in A. With $n + o(n)$ bits of space, we can simulate the locating and extracting in the same complexity.

4. Experiments

We examine the ESP-index in the environment of OS: CentOS 5.5 (64-bit), CPU: Intel Xeon E5504 2.0 GHz (Quad)×2, Memory: 144 GB RAM, and Compiler: gcc 4.1.2.

Datasets of English text and DNA sequences of 200 MB each were obtained from the text collection Pizza&Chili.[6] These are denoted by ENGLISH and DNA.

We first show how a long string is encoded by the evidence Q in Fig. 7. This figure shows the maximum length of the variable, called the *core*, in Q according to the increment of $|P|$. The core is obtained in the final loop of ESP for P. Each value is the average of 1000 time trials. By this, a sufficiently long common substring in S and P is extracted as the core in $ET(S)$ and $ET(P)$; that is, any occurrence where $S[i, j] = P$ is approximately detected by it.

We next compare our ESP-index with other compressed indexes referred to as LZ-index (LZI),[7] Compressed Suffix Array, and FM-index (CSA and FMI).[8] These implementations are based on the results by Navarro [28], Sadakane [33], and Navarro and Mäkinen [29]. Because of the trade-off between construction time and index size, these indexes are examined with respect to several reasonable parameters.

For ESP-index, we set $\varepsilon = 1, 1/4$ for the permutation. In CSA, the option (-P1:L) means that the ψ function is encoded by the gamma function and L specifies the block size for storing ψ. In FMI, (-P4:L) means BWT is represented by Huffman-shaped wavelet tree with the compressed bit-vectors and L specifies the sampling rate for storing rank values; (-P7:L) is the uncompressed version. For detailed information, see the technical report [24] and README of the program.

[6] http://pizzachili.dcc.uchile.cl/texts.html.
[7] http://pizzachili.dcc.uchile.cl/indexes/LZ-index/LZ-index-1.
[8] http://code.google.com/p/csalib/.

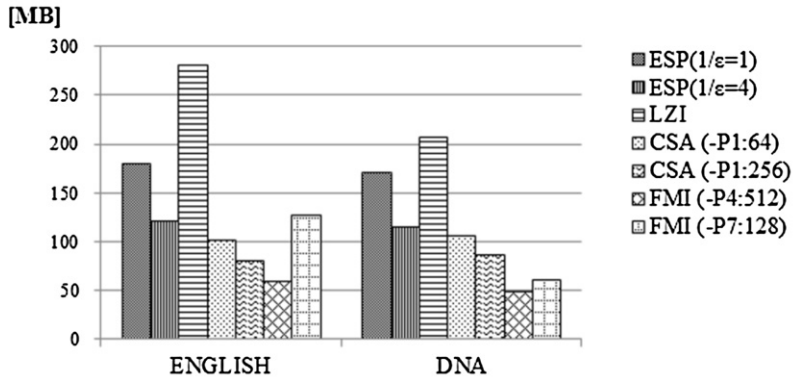

Fig. 8. Index size.

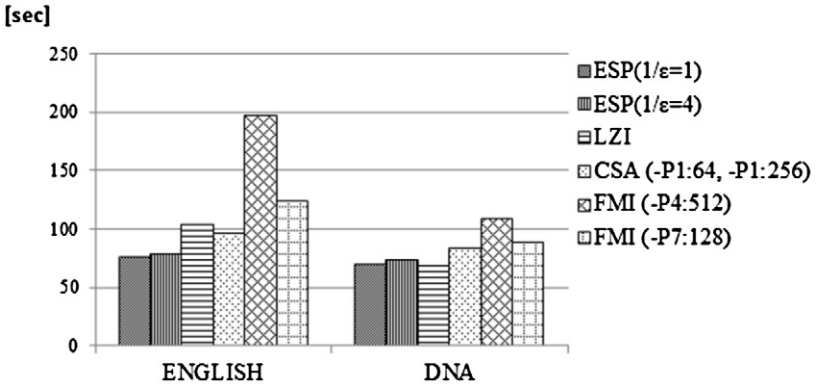

Fig. 9. Construction time.

The result of the index size is shown in Fig. 8, where the space for locating and extracting is not contained in indexes other than LZI; the total sizes of such self-indexes are included in the tables for the results of locating and extracting, shown later. In addition, the scalability of the construction time is shown in Fig. 9. ESP-index with $\varepsilon = 1$ is fastest.

Fig. 10 shows the counting time for all datasets. A substring of S is randomly selected as a pattern P ($5 \leqslant m \leqslant 200\,000$), and the counting time for P is measured. The time is the average of 1000 time trials for each length. In this implementation, we modified our search algorithm so that a single core q is extracted for the short prefix of P. An occurrence of P in S is detected by embedding the evidence $P_1 q P_2$ into the ESP tree of S, where P_1 and P_2 are the remaining prefix and suffix of P, respectively. The length of preprocessed prefix is fixed to be 100, where P is entirely compressed to extract the core if $|P| \leqslant 100$. ESP-index is faster than LZI and comparable to CSA and FMI in case of longer patterns.

Finally, we present the results of the locating and extracting time. Table 2 shows the locating time with several parameters. Each value is the average of the total time to locate randomly selected P in 1000 time trials. In CSA/FMI, option -l:{D}:{D2} indicates the sampling parameter D for suffix array and D2 for the inverse suffix array. As we described in the previous section, ESP sparse denotes the ESP-index with reduced position array in which only the length of the variables in even level is stored. In this table, we awake to the fact that the locating time for $m = 100$ is better than the time for $m = 1000$. The frequency of P is, however, decreasing according to the increment of m. The reason is analyzed in Table 3. Because the core is extracted from the fixed prefix of P, the total time consists almost entirely of the embedding of P and locating of occurrence nodes. The total time is mainly occupied by embedding, and the embedding time grows according to the increment of $|P|$. The results of the extracting time are listed in Table 4. The extracting time of ESP-index is comparable to other practical self-indexes.

5. Discussion

We have a motivation to apply our data structures to practical use. Originally, ESP was proposed to solve a difficult variant of the edit distance by finding the maximal common substrings of two strings. Thus, our method will exhibit its ability if both strings are sufficiently long. Such situations are found in the framework of normalized compression distance [5] to compare two long strings directly. Related to this problem, we obtained a preliminary result [27] based on the online ESP compression [25]. Another important application of this index is found in the problem of ranking the top-k query on a database [15]. There is a possibility to approximate this problem by detecting core and its frequency.

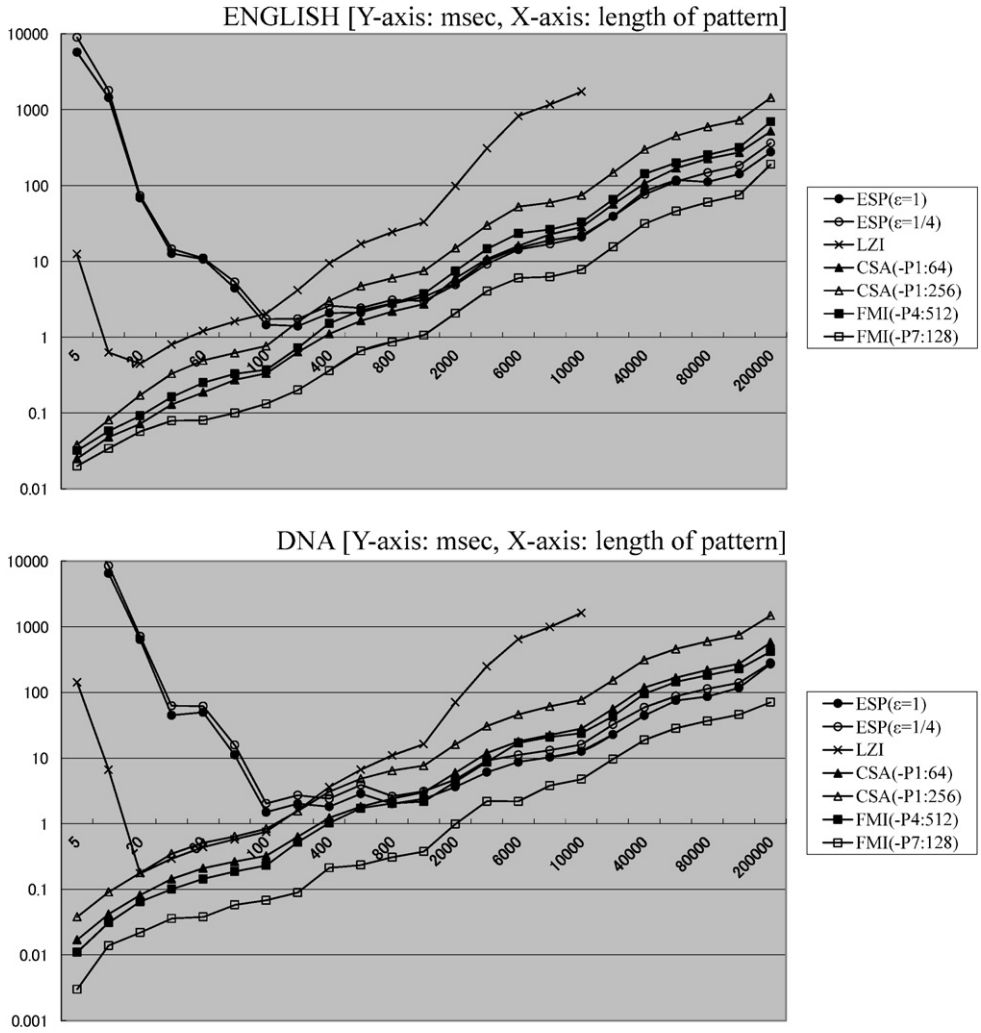

Fig. 10. Counting time.

Table 2
Locating time [ms]. Size denotes the size of the self-index.

	ENGLISH (200 MB)				DNA (200 MB)			
	Size [KB]	$m = 10$	$m = 100$	$m = 1000$	Size [KB]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	224 309	2385.11	1.57	2.94	217 636	8899.43	1.99	3.04
ESP ($1/\varepsilon = 1$)	283 933	1462.89	1.10	2.13	275 486	6879.45	1.42	2.24
ESP sparse ($1/\varepsilon = 4$)	182 530	2067.52	1.60	2.95	176 677	9197.87	2.00	3.04
ESP sparse ($1/\varepsilon = 1$)	242 154	1580.60	1.11	2.13	234 527	6996.57	1.43	2.25
LZI	290 915	0.61	2.00	30.12	214 161	7.23	0.77	16.34
CSA (-P1:64 -I:4:0)	308 927	0.81	0.67	3.36	314 529	1.79	0.66	3.56
CSA (-P1:64 -I:256:0)	107 327	53.65	0.96	3.76	112 929	168.83	1.02	3.17
CSA (-P1:256 -I:4:0)	288 307	1.21	1.32	8.35	293 865	3.02	1.41	8.67
CSA (-P1:256 -I:256:0)	86 707	82.41	1.94	7.81	92 265	314.05	1.43	8.81
FMI (-P4:512 -I:4:0)	265 706	2.24	0.55	3.46	255 483	3.94	0.36	2.54
FMI (-P4:512 -I:256:0)	64106	180.51	1.22	4.26	53 883	412.22	0.64	2.59
FMI (-P7:128 -I:4:0)	336 193	0.80	0.33	1.43	268 264	1.17	0.19	0.67
FMI (-P7:128 -I:256:0)	134 593	55.48	0.78	1.77	66 664	96.22	0.25	0.59

Table 3
Detailed locating time [ms]. Parsing, Embedding, and Locating mean the ratio of the time to extract the core, to find all occurrence nodes in which P is entirely embedded, and to locate all occurrence nodes to the total locating time indicated by Time.

	Length of P	Time	Parsing [%]	Embedding [%]	Locating [%]
ENGLISH	10	**1447.80**	0.001	99.3	0.6
	100	**1.12**	45.8	50.6	3.5
	1000	**2.19**	24.1	74.8	1.0
DNA	10	**6921.90**	0.0002	99.6	0.3
	100	**1.53**	19.4	79.4	1.1
	1000	**2.27**	12.7	87.0	0.2

Table 4
Extracting time [ms]. Index size is the same as locating.

	ENGLISH (200 MB)				DNA (200 MB)			
	Size [KB]	$m = 10$	$m = 100$	$m = 1000$	Size [KB]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	224 309	0.09	0.37	1.63	217 636	0.12	0.21	1.08
ESP ($1/\varepsilon = 1$)	283 933	0.07	0.25	1.18	275 486	0.05	0.16	0.80
ESP sparse ($1/\varepsilon = 4$)	182 530	0.16	0.47	2.58	176 677	0.14	0.34	2.20
ESP sparse ($1/\varepsilon = 1$)	242 154	0.10	0.37	1.99	234 527	0.09	0.30	1.80
LZI	290 915	0.01	0.04	0.27	214 161	0.01	0.03	0.20
CSA (-P1:64 -I:0:4)	308 927	0.04	0.28	1.20	314 529	0.03	0.27	1.16
CSA (-P1:64 -I:0:256)	107 327	0.30	0.47	1.22	112 929	0.31	0.34	1.28
CSA (-P1:256 -I:0:4)	288 307	0.04	0.31	1.83	293 865	0.05	0.21	1.69
CSA (-P1:256 -I:0:256)	86 707	0.25	0.53	2.17	92 265	0.22	0.60	2.01
FMI (-P4:512 -I:0:4)	265 706	0.09	0.39	2.52	255 483	0.05	0.27	1.41
FMI (-P4:512 -I:0:256)	64 106	0.09	0.41	2.27	53 883	0.04	0.27	1.67
FMI (-P7:128 -I:0:4)	336 193	0.05	0.30	0.94	268 264	0.05	0.18	0.58
FMI (-P7:128 -I:0:256)	134 593	0.08	0.37	1.10	66 664	0.05	0.16	0.44

Acknowledgements

This work is partially supported by JST PRESTO program and KAKENHI (23680016). The authors thank the anonymous referees for their careful comments that helped us to improve this paper.

References

[1] A. Apostolico, S. Lonardi, Off-line compression by greedy textual substitution, Proc. IEEE 88 (11) (2000) 1733–1744.
[2] D. Arroyuelo, G. Navarro, K. Sadakane, Reducing the space requirement of LZ-Index, in: CPM, 2006, pp. 318–329.
[3] P. Bille, G. Landau, R. Raman, K. Sadakane, S. Rao, O. Weimann, Random access to grammar-compressed strings, in: SODA, 2011, pp. 373–389.
[4] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, A. Shelat, The smallest grammar problem, IEEE Trans. Inform. Theory 51 (7) (2005) 2554–2576.
[5] R. Cilibrasi, P. Vitanyi, Clustering by compression, IEEE Trans. Inform. Theory 51 (4) (2005) 1523–1545.
[6] D. Clark, Compact pat trees, PhD thesis, University of Waterloo, 1996.
[7] F. Claude, G. Navarro, Self-indexed grammar-based compression, Fund. Inform. 111 (3) (2011) 313–337.
[8] R. Cole, U. Vishkin, Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms, in: STOC, 1986, pp. 206–219.
[9] G. Cormode, S. Muthukrishnan, The string edit distance matching problem with moves, ACM Trans. Algorithms 3 (1) (2007), Article 2.
[10] O. Delpratt, N. Rahman, R. Raman, Engineering the LOUDS succinct tree representation, in: WEA, 2006, pp. 134–145.
[11] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: FOCS, 2000, pp. 390–398.
[12] P. Ferragina, G. Manzini, Indexing compressed text, J. ACM 52 (4) (2005) 552–581.
[13] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, S. Puglisi, A faster grammar-based self-index, in: LATA, 2012, pp. 240–251.
[14] R. Grossi, J. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, in: STOC, 2000, pp. 397–406.
[15] W.-K. Hon, R. Shah, J.S. Vitter, Space-efficient framework for top-k string retrieval problems, in: FOCS, 2009, pp. 713–722.
[16] G. Jacobson, Space-efficient static trees and graphs, in: FOCS, 1989, pp. 549–554.
[17] J. Kieffer, E.-H. Yang, Grammar-based codes: A new class of universal lossless source codes, IEEE Trans. Inform. Theory 46 (3) (2000) 737–754.
[18] J. Kieffer, E.-H. Yang, G. Nelson, P. Cosman, Universal lossless compression via multilevel pattern matching, IEEE Trans. Inform. Theory IT-46 (5) (2000) 1227–1245.
[19] S. Kreft, G. Navarro, Self-indexing based on LZ77, in: CPM, 2011, pp. 41–54.
[20] N. Larsson, A. Moffat, Off-line dictionary-based compression, Proc. IEEE 88 (11) (2000) 1722–1732.
[21] E. Lehman, Approximation algorithms for grammar-based compression, PhD thesis, MIT, 2002.
[22] E. Lehman, A. Shelat, Approximation algorithms for grammar-based compression, in: SODA, 2002, pp. 205–212.
[23] V. Mäkinen, Compact suffix array, in: CPM, 2000, pp. 305–319.
[24] V. Mäkinen, G. Navarro, New search algorithms and time/space tradeoffs for succinct suffix arrays, Tech. rep., University of Helsinki, 2004.
[25] S. Maruyama, M. Takeda, M. Nakahara, H. Sakamoto, An online algorithm for lightweight grammar-based compression, Algorithms 5 (2) (2012) 214–235.
[26] J. Munro, R. Raman, V. Raman, S. Rao, Succinct representations of permutations, in: ICALP, 2003, pp. 345–356.

[27] M. Nakahara, S. Maruyama, T. Kuboyama, H. Sakamoto, Scalable detection of frequent substrings by grammar-based compression, in: DS, 2011, pp. 236–246.
[28] G. Navarro, Indexing text using the Ziv–Lempel tire, J. Discrete Algorithms 2 (1) (2004) 87–114.
[29] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007), Article 2.
[30] L. Russo, A. Oliveira, A compressed self-index using a Ziv–Lempel dictionary, Inf. Retr. 11 (4) (2008) 359–388.
[31] W. Rytter, Application of Lempel–Ziv factorization to the approximation of grammar-based compression, Theoret. Comput. Sci. 302 (1–3) (2003) 211–222.
[32] K. Sadakane, Compressed text databases with efficient query algorithms based on the compressed suffix array, in: ISAAC, 2000, pp. 410–421.
[33] K. Sadakane, New text indexing functionalities of the compressed suffix arrays, J. Algorithms 48 (2) (2003) 294–313.
[34] K. Sadakane, G. Navarro, Fully-functional succinct trees, in: SODA, 2010, pp. 134–149.
[35] S. Sahinalp, U. Vishkin, Symmetry breaking for suffix tree construction, in: STOC, 1994, pp. 300–309.
[36] S. Sahinalp, U. Vishkin, Efficient approximate and dynamic matching of patterns using a labeling paradigm, in: FOCS, 1996, pp. 320–328.
[37] H. Sakamoto, A fully linear-time approximation algorithm for grammar-based compression, J. Discrete Algorithms 3 (2–4) (2005) 416–430.
[38] H. Sakamoto, S. Maruyama, T. Kida, S. Shimozono, A space-saving approximation algorithm for grammar-based compression, IEICE Trans. Inform. and Systems E 92-D (2) (2009) 158–165.
[39] D. Shapira, J. Storer, Edit distance with move operations, J. Discrete Algorithms 5 (2) (2007) 380–392.
[40] E.-H. Yang, J. Kieffer, Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform – Part one: Without context models, IEEE Trans. Inform. Theory 46 (3) (2000) 755–777.
[41] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.
[42] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.