ESP-index : A Compressed Index Based on Edit-Sensitive Parsing

Maruyama, Shirou
Kyushu University

Masaya, Nakahara
Kyushu Institute of Technology

Naoya, Kishiue
Kyushu Institute of Technology

Hiroshi, Sakamoto
Kyushu Institute of Technology

ESP-Index: A Compressed Index Based on Edit-Sensitive Parsing

Shirou Maruyama[1], Masaya Nakahara[2], Naoya Kishiue[2⋆], Hiroshi Sakamoto[2,3⋆⋆]

[1] Kyushu University, 744 Motooka, Nishi-ku, Fukuoka-shi, Fukuoka 819-0395
[2] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka-shi, Fukuoka, 820-8502
[3] PRESTO JST, 4-1-8 Honcho Kawaguchi, Saitama 332-0012, JAPAN
shiro.maruyama@i.kyushu-u.ac.jp, m_nakahara@donald.ai.kyutech.ac.jp, kishiue.n@gmail.com, hiroshi@ai.kyutech.ac.jp

Abstract. We propose a compressed self-index based the edit-sensitive parsing (ESP). Given a string S, its ESP tree is equivalent to a context-free grammar deriving just S, which can be represented as a DAG G. Finding pattern P in S is reduced to embedding P into G. Succinct data structures are adopted and G is then decomposed into two LOUDS bit strings and a single array for permutation, requiring $(1+\varepsilon)n\log n + 4n + o(n)$ bits for any $0 < \varepsilon < 1$ where n corresponds to the number of different symbols in the grammar. The time to count the occurrences of P in S is in $O(\frac{\log^* u}{\varepsilon}(m\log n + occ_c(\log m \log u)))$, where $m = |P|$, $u = |S|$, and occ_c is the number of occurrences of a maximal common subtree in ESP trees of P and S. Using an additional array in $n\log u$ bits of space, our index supports locating P and displaying substring of S. Locating time is the same as counting time and displaying time for a substring of length m is $O(m + \log u)$.

1 Introduction

We propose a compressed index based on the *edit-sensitive parsing* (ESP), which was introduced to approximate a variant of string edit distance 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. This problem called edit distance *with move* is NP-hard, and the distance was proved to be $O(\log u)$-approximable [9] for strings of length u. Moreover, 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 [3].

When we consider *tighter* embedding, i.e. a string is embedded into another one as a substring, this problem becomes the pattern matching. In this work we use ESP to represent a grammar which is transformed to a compressed index based on our theoretical results for efficient pattern matching and data structures

⋆ He moved to Hitachi Solutions, Ltd.

⋆⋆ This work was partially supported by JST PRESTO program.

on ESP. The following outlines the proposed data structures and its operations. We first preprocess the text S and build ESP tree T_S. Given a pattern P, we construct T_P and decomposes it into a sequence of subtrees, whose roots are labeled by $x_1, \ldots, x_k$. This sequence is an *evidence* of 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 the label of v_i is x_i and the subtrees rooted by v_i, v_{i+1} are *adjacent* in this order. We should note that P itself is always evidence of P. We design algorithms to extract as short evidence as possible by analysis of ESP, and to embed the evidence in T_S.

Another contribution is to develop compact data structures for the proposed algorithms. An ESP is represented by a restricted CFG, and is equivalent to a DAG G where every internal node has its left and right 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 the left and right trees are encoded by LOUDS [4], one of the succinct data structures for ordered trees. Further, correspondence among the nodes of the trees is stored in a single array. Adding the data structure for the permutation [5] over the array makes it possible to traverse G. The size of such data structures is at most $(1+\varepsilon)n \log n + 4n + o(n)$ bits of space for arbitrary $0 < \varepsilon < 1$, where n is the number of variables in G.

On the other hand, the compression algorithm should refer to a function, called *reverse dictionary* to get a name of the variable associated with a digram. For example, if a production rule $Z \to XY$ exists, any occurrence of the digram XY in P, which is determined to be replaced, should be replaced by the same Z. Taking up the hash function $H(XY) = Z$ for preprocessing P compels that index size be increased. Thus our algorithm obtains the names of variables for P directly from the compressed G using binary search.

The time to count all occurrences of a pattern P in S is $O(\frac{\log^* u}{\varepsilon}(m \log n + occ_c(\log m \log u)))$ time, where $m = |P|$, $u = |S|$, and occ_c is the number of occurrences of a *core* in T_S, which is the maximal common subtrees of T_S and T_P. The other two operations to locate/display are supported by an additional array requiring $n \log u$ bits of space to store the length of the substring encoded by each variable. This array can be reduced by level-wise sampling because the ESP tree is *balanced*. The time to display $S[i, j]$ is $O(j - i + \log u)$.

We compare the performance of ESP-index with other practical self-indexes [6–8] under several reasonable parameters. Beyond that, we give an experimental result for maximal common substring detection. These common substrings are obtained to find common variables in compressed T_S and T_P. In these experiments, we conclude that the proposed index is efficient enough for cases where the pattern is long.

2 Pattern Matching on ESP

The set of all strings over an alphabet Σ is denoted by Σ^*. The length of a string $w \in \Sigma^*$ is denoted by $|w|$. A string a^k ($k \geq 1$) is also denoted by a^+, and is called

a repetition, denoted by a^{++}, if $k \geq 2$. $S[i]$ 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 $\log^* u = \min\{i \mid \log^{(i)} u \leq 1\}$, i.e. $\log^* u \leq 5$ for $u \leq 2^{65536}$. Thus We can treat any $\log^* u$ as constant in a practical sense.

We assume that any context-free grammar 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 $S(D) = abcabca$.

2.1 Edit-sensitive parsing (ESP)

We start with the outline of ESP. For any string, it is uniquely partitioned to $w_1 a_1^{++} w_2 a_2^{++} \cdots w_k a_k^{++} w_{k+1}$ by maximal repetitions, where each a_i is a symbol and w_i is a string containing no repetition. Each a_i^{++} is called Type1 metablock, w_i is called Type2 metablock if $|w_i| \geq \log^* n$, and other short w_i is called Type3 metablock, where if $|w_i| = 1$, this is attached to a_{i-1}^{++} or a_i^{++}, with preference a_{i-1}^{++} when both are possible. 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 $k \geq 2$,
   (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 $S = s_1 s_2 \cdots s_k$ $(2 \leq |s_i| \leq 3)$ by *alphabet reduction*, 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 $s_i = XYZ$.

Case (a) and (b) denote a typical *left aligned parsing*. For example, in case $S = a^6$, $S' = x^3$ and $x \to a^2 \in D'$, and in case $S = a^9$, $S' = x^3 y$ and $x \to a^2, y \to ay', y' \to aa \in D'$. In Case (c), we omit the description of alphabet reduction [3] because the details are unnecessary in this paper.

Finally, we define ESP for any string $S \in (\Sigma \cup V)^*$ that is partitioned to $S_1 S_2 \cdots S_k$ 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 ESP is defined by $ESP^i(S, D) = ESP^{i-1}(ESP(S, D))$. In particular, $ESP^*(S, D)$ denotes the iterations of ESP until $|S| = 1$. After computing $ESP^*(S, D)$, the final dictionary represents a rooted ordered binary tree deriving S, which is denoted by $ET(S)$. We refer to several characteristics of ESP, which are bases of our study.

Lemma 1. (Cormode and Muthukrishnan [3]) The height of $ET(S)$ is $O(\log |S|)$ and $ET(S)$ can be computed in time $O(|S| \log^* |S|)$ time.

Lemma 2. (Cormode and Muthukrishnan [3]) Let $S = s_1 s_2 \cdots s_k$ be the partition of a Type2 metablock S by alphabet reduction. For any $1 \leq j \leq |S|$, the block s_i containing $S[j]$ is determined by at most $S[j - \log^* |S| - 5, j + 5]$.

2.2 Pattern embedding problem

We focus on the problem to find occurrences of P by embedding of P into a parsing tree. Given a 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 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 a substring of S derived by $L(v)$, and $yield(v_1 \cdots v_k) = yield(v_1) \cdots yield(v_k)$. We note that T_S and T_P are ordered binary trees.

Let us define some notations for 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, 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 satisfying 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 $lra(v_1) = lla(v_2)$, we call that v_1, v_2 are *adjacent* in this order, and we also call that v_1 is left adjacent to v_2 (or v_2 is right adjacent to v_1). We can derive the following characterization immediately.

***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 $left(lla(v_2))$.*

Using this adjacency, we define an embedding of sequence of nodes, $v_1, \ldots, v_k$ ($k \geq 2$), as follows: if v_i, v_{i+1} are adjacent in this order ($1 \leq i \leq k-1$) and z is the lowest common ancestor of v_1 and v_k denoted by $z = lca(v_1, v_k)$, we state that the sequence is embedded in z and it is denoted by $(v_1, \ldots, v_k) \prec z$. For such z, if $yield(v_1 \cdots v_k) = P$, z is called an *occurrence node of* P. We introduce a special type of strings to guarantee occurrences of P in S.

Definition 1. A string $Q \in (\Sigma \cup V)^*$ of length k satisfying the following condition is called an *evidence of* P: 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 using obtained evidence.

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 $p_i \rightarrow \alpha_i \in D'$;
    $Q_p \leftarrow Q_p\alpha_1 \cdots \alpha_j,$ 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 $q_i \rightarrow \gamma_i \in D'$;
    $Q_s \leftarrow \gamma_{r-j} \cdots \gamma_r Q_s,$ 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**$(|Q_pQ_s| > 0)$ return $Q \leftarrow Q_pQ_s$; **else** return P;

Fig. 1. Algorithm to find evidence of pattern P using dictionary D_S from $ESP^*(S, D)$.

3 Algorithms and Data Structures

We propose two algorithms: one generates an evidence Q of pattern P from preprocessed $ESP^*(S, D) = (S', D_S)$, and the other algorithm finds the occurrence node z of P such that $(v, v_1 \ldots, v_k) \prec z$ for given Q and v in T_S satisfying $L(v) = Q[1]$. Finally, we propose data structures to access the next node v' satisfying $L(v') = L(v)$ for each v in T_S. By this, all 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 pattern P and final dictionary D_S from $ESP^*(S, D)$. P is partitioned to $P = \alpha\beta\gamma$ for the first metablock α and the last metablock γ. Depending on the type of metablocks, P is further partitioned to $P = \alpha_p\alpha_s\beta\gamma_p\gamma_s$. The algorithm then updates current $Q = Q_pQ_s$ by $Q_p \leftarrow Q_p\alpha_p$ and $Q_s \leftarrow \gamma_sQ_s$, and $P \leftarrow P'$ 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 an 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 and $q_i \neq q_{i-1}, q_{i+1}$. Then Q is an evidence of P satisfying $k = O(\log m \log^* u)$ for $m = |P|$.

Proof. For $P = \alpha\beta\gamma$ by the first/last metablock α/γ, if α, γ are Type1 or 3, $\alpha\beta\gamma$ is clearly an evidence of P. In this case, any occurrence of β inside $S[n, m] = \alpha\beta\gamma$ is transformed to a same β' in this while loop. Thus, $\alpha\beta'\gamma$ is an evidence of P.

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

Applying the above analysis to β' until its length becomes one, we can finally obtain Q as an evidence of P. The number of iterations of $ESP(P, D) = (P', D \cup D')$ is $O(\log m)$ because $|P'| \leq |P|/2$. In i-th iteration, if current $Q = Q_p Q_s$ is updated to $Q_p \alpha_p \gamma_s Q_s$, $\alpha_p \gamma_s$ 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, the algorithm *Find_pattern*(Q, D_S, v, T_S) finds an embedding $(v, v_1, \ldots, v_\ell) \prec z$ satisfying $yield(vv_1 \cdots v_\ell) = P$ for fixed v having the label $Q[1]$. By Lemma 3, such z exists iff z is an occurrence node of P. We show the correctness of this algorithm and the time complexity.

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

Proof. We outline the 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 because such 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 $q_i \in \Sigma \cup V$. If Q contains no repetition and we assume that $(v_1, \ldots, v_j) \prec z$ is found for $Q_1 \cdots Q_j = q_1 \cdots q_j$. 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 because z, z' must be in a same path. Thus an embedding of length at most $O(\log m \log^* u)$ from v is computed in $O(\log m \log u \log^* u)$ time.

Let $Q_j = q^\ell$ for some symbol q and $\ell \geq 2$. In ESP, any repetition is transformed to 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, an occurrence $S[s, t] = q^\ell$ is partitioned to $S[s, t] = S[s_1, t_1]S[s_2, t_2] \cdots S[s_k, t_k]$ such that $|S[s_i, t_i]| = 2^{\ell_i} \geq 1$, v_i in T_S is the root of the complete binary tree deriving $S[s_i, t_i]$, and $k = O(\log \ell)$. Let $S[s_i, t_i]$ be the longest segment. We note that all symbols in current string are replaced by the next iteration of ESP. By this characteristic, when $S[s_i, t_i]$ is transformed to $S'[j]$, the adjacent digram

```
Find_pattern(Q, D_S, v, T_S) /* L(v) = Q[1] */
let Q = Q_1 ··· Q_k, Q_i ∈ {q_i^+}, 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^{++} */
            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^{++} is covered */
            else return 0;
        }j ← j + 1;
    }
}return z;
```

Fig. 2. Algorithm to find occurrence node of P starting with a given node v.

$S'[j-2, j-1]$ derives a string containing $S[s_1, t_1] \cdots S[s_{i-1}, t_{i-1}]$ as its suffix. 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, we find the occurrence of P in $O(\log m \log^* u(\log m + \log u)) = O(\log m \log u \log^* u)$ time. □

3.3 Data structures

We develop compact data structures for *Find_pattern*(Q, D_S, v, T_S) to access a next occurrence of v satisfying $L(v) = q$ for any $q \in \Sigma \cup V$. These improvements are achieved by two techniques: one is decomposition of DAG into *left tree and right tree*; the other is simulation of the *reverse dictionary* for pattern compression. First we treat decomposition of DAG G, which is a graph representation of D_S. Introducing a super sink v_0 together with left and right edges from any sink of G to v_0, G can be modified to have the unique source/sink.

***fact* 2** *Let G be a DAG with single source/sink such that any node except the sink has exactly two children. For any in-branching spanning tree of G, the graph defined by the remaining edges is also an in-branching spanning tree of G.*

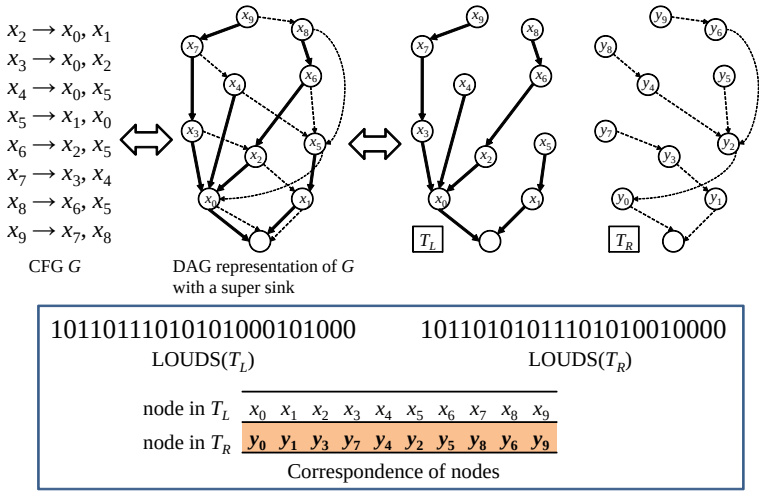

Fig. 3. Succinct representation of a CFG by left tree and right tree in LOUDS bit-string with a permutation array.

In-branching spanning tree of G constructed by left edges only is called *left tree* of G and denoted by T_L. Thus the complementary tree is called *right tree* of G and denoted by T_R. An example of G and its left/right tree is shown in Fig. 3 with its succinct representation proposed hereafter.

When a DAG is decomposed into T_L and T_R, the two are represented by succinct data structures for ordered trees and permutations. The bit-string by LOUDS in [4] for an ordered tree is defined as shown below. We visit any node in level-order from the root. As we visit a node v with $d \geq 0$ children, we append 1^d0 to the bit-string beginning with the empty string. Finally, we add 10 as the prefix corresponding to an imaginary root, which is the parent of the root of the tree. For the n-node tree, LOUDS uses $2n + o(n)$ bits to support constant time access to the parent, the i-th child, and the number of children of a node.

To traverse G equivalent to T_S, we also need the correspondence of nodes in one tree to the other. For this purpose, we employ the succinct data structure for permutation in [5]. For a given permutation π of $N = (0, \ldots, n-1)$, using $(1+\varepsilon)n \log n + o(n)$ bits of space, this data structure supports access to $\pi[i]$ in $O(1)$ time and $\pi^{-1}[i]$ in $O(1/\varepsilon)$ time. For instance, if $\pi = (2, 3, 0, 4, 1)$, then $\pi[2] = 0$ and $\pi^{-1}[4] = 3$; 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 relation by $\pi[i] = j$ and $\pi^{-1}[j] = i$.

We introduce another preprocess for G. For each iteration $ESP(S, D) = (S', D \cup D')$, we rename new variables in D' and S' by sorting[1] all production rules $X \to X_i X_j \in D'$ by (i, j): If the rank of $X \to X_i X_j$ is k, all occurrences of X in D' and S' are renamed to X_k. For example, $D' = \{X_1 \to ab, X_2 \to$

[1] A similar technique was proposed in [2] but variables are sorted by encoded strings.

$bc, X_3 \to ac, X_4 \to aX_2\}$ and $S' = X_1X_2X_3X_4$ are renamed to $D' = \{X_1 \to ab, X_2 \to ac, X_3 \to aX_4, X_4 \to bc\}$ and $S' = X_1X_4X_2X_3$. Thus, variable X_i in G coincides with node i in LOUDS of T_L because they are both named in level-order. The G in Fig. 3 is already renamed. By this improvement, the size of the array required for node correspondence is reduced to $n \log n$ bits of space.

Finally we simulate the reverse dictionary using G for pattern compression. When $ESP^*(S, D)$ is computed, the naming function $H_S(XY) = Z$ defined by $Z \to XY \in D$ is realized by a hash function. However, because our index does not contain this data structure, we must simulate H_S by only G to obtain an evidence of P. By preprocessing, 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. Because LOUDS supports referring to the number of children and i-th child, $H_S(X_iX_j) = X_k$ is obtained by binary search in the following time complexity.

Lemma 5. The function $H_S(XY) = Z$ is computable in $O(\frac{1}{\varepsilon} \log k) = O(\frac{1}{\varepsilon} \log n)$ time, where k is the maximum degree and n is the number of nodes of T_L.

Theorem 1. The size of ESP-index for string S is $(1+\varepsilon)n \log n + 4n + o(n)$ bits of space, where n is the number of variables in T_S. With pattern P, the number of its occurrence in S is computable in $O(\frac{\log^* u}{\varepsilon}(m \log n + occ_c(\log m \log u)))$ time for any $0 < \varepsilon < 1$, where $u = |S|$, $m = |P|$, and 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. Thus, starting with v_ℓ labeled by q which encodes a longest string, we can find z by $(v_1, \ldots, v_\ell) \prec z_1$, $(v_\ell, \ldots, v_k) \prec z_2$, and $(v_1, \ldots, v_k) \prec lca(z_1, z_2) = z$. This derives the time bound. □

Locating and displaying are realized by an additional array to store the length of each variable. Since ESP tree is balanced, we obtain the bound below.

Corollary 1. With additional $n \log u + o(n)$ bits of space, ESP-index supports locating P and displaying $S[i, j]$. The time to locate is the same as the case of counting, and the time to display a substring of length m is $O(m + \log u)$.

4 Experiments

The environment is OS:CentOS 5.5 (64-bit), CPU:Intel Xeon E5504 2.0GHz (Quad)×2, Memory:144GB RAM, HDD:140GB, and Compiler:gcc 4.1.2. Datasets of English texts and DNA sequences of 200MB each are obtained from the text collection in Pizza&Chili Corpus.[2]

We first show how a long string is encoded by evidence of pattern in Fig. 4. This figure shows the maximum length of a string encoded by a symbol in evidence Q according to the pattern length. We call this symbol in Q a *core*. By

[2] http://pizzachili.dcc.uchile.cl/texts.html

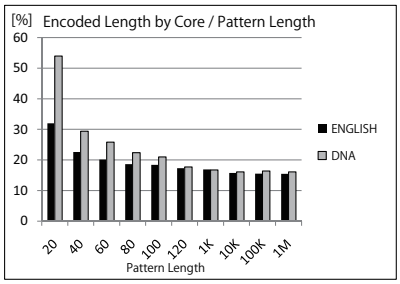

Fig. 4. Encoded String Length by Core / Pattern Length.

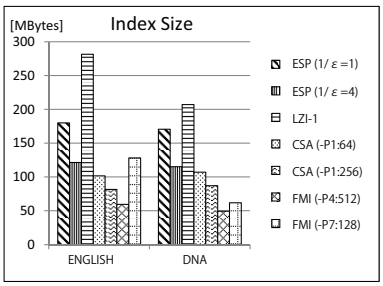

Fig. 5. Index Size

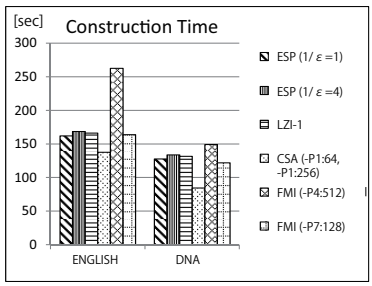

Fig. 6. Construction Time

this result, sufficiently long common substrings in S_1 and S_2 are extracted as common variables in $ET(S_1)$ and $ET(S_2)$.

We next compare our ESP-index with other compressed indexes referred to as LZ-index (LZI)[3], Compressed Suffix Array, and FM-index (CSA and FMI)[4]. These implementations are based on [6–8]. Due to the trade-off between construction time and index size, the indexes referred to above are examined with respect to reasonable parameters.

For ESP-index, we set $\varepsilon = 1, 1/4$ for 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 that BW-text is represented by a Huffman-shaped wavelet tree with compressed bit-vectors and L specifies the sampling rate for storing rank values; (-P7:L) is the uncompressed version. In addition, these CSA and FMI do not make indexes for occurrence position.

The result of index size is shown in Fig. 5, where the space for locating is removed in all indexes except LZI; total index size including the space for locating/displaying is shown in the last two tables. Fig. 5 reveals that ESP-index is compact enough and comparable to CSA(-P1:64). The result of construction time is shown in Fig. 6. It is deduced from this result that ESP-index is comparable with FMI and CSA in the parameters in construction time, and slower than LZI. Further, a conspicuous difference is not seen in construction time.

[3] http://pizzachili.dcc.uchile.cl/indexes/LZ-index/LZ-index-1
[4] http://code.google.com/p/csalib/

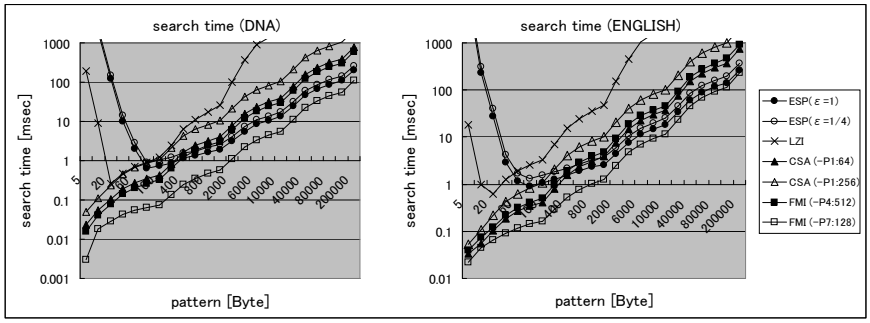

Fig. 7. Counting Time.

Fig. 7 shows the time to count the occurrences of pattern for both types of English and DNA. Random[5] selection of pattern from the text was made 1000 times for each fixed pattern length; the search time indicates the average time. In this implementation, we modified our search algorithm so that the core q is extracted by preprocessing a short prefix of P. And an occurrence of P in S is examined by finding q and the exact match of the remaining substrings by partial decoding of the compressed S. By preliminary experiments, we determine the length of preprocessed prefix to be 1% of $u = |P|$ in practice. In DNA and ENGLISH, our method is faster than LZI and CSA in case of long patterns.

Finally, we show the results of locating and displaying in Table 1 and Table 2 respectively. Locating is achieved with an additional array to store the length of the encoded string for each variable. Because ESP tree is balanced, the size of this array is reduced by level-wise sampling; in this experiment, the array is developed only for variables produced in odd level of ESP. In CSA/FMI, option -I:{D}:{D2} indicates sampling parameter D for suffix array and D2 for the reverse array. We get better results for $m = 100$ than for $m = 1000$. This is because the frequency of longer pattern becomes one and the search time is thus proportional to m. For comparison of theoretical time/space bound, see e.g. [2].

5 Discussion

We have another 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 [1] to compare two long strings directly. Improving $\log m \log u$ in time complexity is also an important work.

[5] The result for random *generated* pattern is omitted because the search time immediately converges under milliseconds in ESP, CSA, and FMI due to its rare occurrence.

Table 1. Locating Time

	ENGLISH				DNA			
	index size	locating time [sec]			index size	locating time [sec]		
	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	223292	84.27	0.93	2.44	212847	1923.45	0.63	1.73
ESP ($1/\varepsilon = 1$)	282646	60.76	0.67	1.81	269424	1271.55	0.46	1.33
ESP sparse ($1/\varepsilon = 4$)	181756	88.91	0.97	2.47	170954	1701.10	0.77	1.78
ESP sparse ($1/\varepsilon = 1$)	241110	73.09	0.67	1.81	227532	1400.90	0.46	1.32
LZI-1	290915	0.61	2.00	30.12	214161	7.23	0.77	16.34
CSA (-P1:64 -I:4:0)	308927	0.81	0.67	3.36	314529	1.79	0.66	3.56
CSA (-P1:64 -I:256:0)	107327	53.65	0.96	3.76	112929	168.83	1.02	3.17
CSA (-P1:256 -I:4:0)	288307	1.21	1.32	8.35	293865	3.02	1.41	8.67
CSA (-P1:256 -I:256:0)	86707	82.41	1.94	7.81	92265	314.05	1.43	8.81
FMI (-P4:512 -I:4:0)	265706	2.24	0.55	3.46	255483	3.94	0.36	2.54
FMI (-P4:512 -I:256:0)	64106	180.51	1.22	4.26	53883	412.22	0.64	2.59
FMI (-P7:128 -I:4:0)	336193	0.80	0.33	1.43	268264	1.17	0.19	0.67
FMI (-P7:128 -I:256:0)	134593	55.48	0.78	1.77	66664	96.22	0.25	0.59

Table 2. Displaying Time

	ENGLISH				DNA			
	index size	displaying time [sec]			index size	displaying time [sec]		
	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$	[Kbytes]	$m = 10$	$m = 100$	$m = 1000$
ESP ($1/\varepsilon = 4$)	223292	0.09	0.37	1.63	212847	0.12	0.21	1.08
ESP ($1/\varepsilon = 1$)	282646	0.07	0.25	1.18	269424	0.05	0.16	0.80
ESP sparse ($1/\varepsilon = 4$)	181756	0.16	0.47	2.58	170954	0.14	0.34	2.20
ESP sparse ($1/\varepsilon = 1$)	241110	0.10	0.37	1.99	227532	0.09	0.30	1.80
LZI-1	290915	0.01	0.04	0.27	227532	0.01	0.03	0.20
CSA (-P1:64 -I:0:4)	308927	0.04	0.28	1.20	314529	0.03	0.27	1.16
CSA (-P1:64 -I:0:256)	107327	0.30	0.47	1.22	112929	0.31	0.34	1.28
CSA (-P1:256 -I:0:4)	288307	0.04	0.31	1.83	293865	0.05	0.21	1.69
CSA (-P1:256 -I:0:256)	86707	0.25	0.53	2.17	92265	0.22	0.60	2.01
FMI (-P4:512 -I:0:4)	265706	0.09	0.39	2.52	255483	0.05	0.27	1.41
FMI (-P4:512 -I:0:256)	64106	0.09	0.41	2.27	53883	0.04	0.27	1.67
FMI (-P7:128 -I:0:4)	336193	0.05	0.30	0.94	268264	0.05	0.18	0.58
FMI (-P7:128 -I:0:256)	134593	0.08	0.37	1.10	66664	0.05	0.16	0.44

References

1. R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.
2. F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *MFCS09*, pages 235–246, 2009. to appear in Fundamenta Informaticae.
3. G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algor.*, 3(1):Article 2, 2007.
4. O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *WEA2006*, pages 134–145, 2006.
5. J.I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *ICALP03*, pages 345–356, 2003.
6. G. Navarro. Indexing text using the ziv-lempel tire. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.
7. G. Navarro and V. Makinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.
8. K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.
9. D. Shapira and J.A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007.