数理解析研究所講究録
第 1744 巻 2011 年 115-122

A Searchable Compressed Edit-Sensitive Parsing *

Naoya Kishiue[1], Masaya Nakahara[1], Shirou Maruyama[2], Hiroshi Sakamoto[1,3]
[1] Kyushu Institute of Technology, 680-4 Kawazu, Iizuka-shi, Fukuoka 820-8502,
[2] Kyushu University, 744 Motooka, Nishi-ku, Fukuoka-shi, Fukuoka 819-0395,
[3] JST PRESTO, 4-1-8 Honcho Kawaguchi, Saitama 332-0012, Japan

Abstract

Practical data structures for the edit-sensitive parsing (ESP) are proposed. Given a string S, its ESP tree is equivalent to a context-free grammar G generating just S, which is represented by a DAG. Using the succinct data structures for trees and permutations, G is decomposed to two LOUDS bit strings and single array in $(1+\varepsilon)n\log n+4n+o(n)$ bits for any $0<\varepsilon<1$ and the number n of variables in G. The time to count occurrences of P in S is in $O(\frac{1}{\varepsilon}(m\log n+occ_c(\log m\log u))$, whereas $m=|P|$, $u=|S|$, and occ_c is the number of occurrences of a maximal common subtree in ESPs of P and S. The efficiency of the proposed index is evaluated by the experiments conducted on several benchmarks complying with the other compressed indexes.

1 Introduction

The edit distance is one of the most fundamental problems with respect to every string in dealing with the text. Exclusively with the several variants of this problem, the *edit distance with move* where moving operation for any substring with unit cost is permitted is NP-hard and $O(\log u)$-approximable [14] for string length u. With regard to the matching problem whose approximate solution can be obtained by means of *edit-sensitive parsing* (ESP) technique [4], utilization of detected maximal common substrings makes it possible to expect application of the problem to plagiarism detection and clustering of texts. As a matter of fact, a compression algorithm based on ESP has been proposed [13], which results in exhibition of its approximation ratio for the optimum compression.

In this work, we propose a practical compressed index for ESP. Utilization of a compressed index makes it possible to search patterns rapidly, which is regarded as a specific case of maximum common substrings of the two strings where one is entirely in the other. Comparison of the compressed index proposed in this work with the indexes dealt with in the other methods reveals that sufficient performance is provided in accordance with the proposed method. On the other hand, it is shown from theoretical analysis of ESP that thanks to the proposed method, a long enough common substring of the two strings of the text and pattern can be found rapidly from the compressed index.

Edit distance is closely related to optimum compression. Particularly with one of approximation algorithms, assigning a same name to common subtrees allows approximately optimum parsing tree, i.e. approximately optimum CFG. This optimization problem is not only NP-hard but also $O(\log n)$-approximable [1, 10, 12]. As a consequence, compressing two strings and finding occurrences of a maximal subtree from these parsing trees make it possible to determine with great rapidity whether one string appears in another as a substring.

Our contributions are hereunder described. The proposed algorithm for indexed grammar-based compression outputs a CFG in Chomsky normal form. The said CFG, which is equivalent to a DAG G where every internal node has its left and right children, is also equivalent to the two spanning trees. The one called the left tree is exclusively constructed by the left edges, whereas the one called the right tree is exclusively constructed by the right edges. Both the left and the right trees are encoded by LOUDS [5], one of the types of the succinct data structure for ordered trees. Furthermore the correspondence among the nodes of the trees is memorized in an array. Adding the data structure for the permutation [7] over the array makes it possible to traverse the G. Meanwhile it is possible for the size of the data structure to be constructed with $(1+\varepsilon)n\log n+4n+o(n)$ bits for arbitrary $0<\varepsilon<1$, where n is the number of the variables in the G.

At the next stage, the algorithm should refer to a function, called *reverse dictionary* for the text when compression of the pattern is executed. For example, if a production rule $Z\to XY$ is

*The full paper is available from `http://arxiv.org/abs/1101.0080` included in G, an occurrence of the digram XY in a pattern, which is determined to be replaced, should be replaced without fail by the same Z. Taking up the hash function $H(XY) = Z$ for the said purpose compels the size of the index to be increased. Thus we propose the improvement for compression so as to obtain the name Z directly from the compression. It is possible to calculate the number of occurrences of a given pattern P from a text S in $O(\frac{1}{\varepsilon}(m \log n + occ_c(\log m \log u))$ time in accordance with the contrivance referred to above together with the characteristics of the ESP, where $m = |P|$ and $u = |S|$. On the other hand, occ_c is the occurrence number of maximal common subtree called a core in the parsing tree for S and P. The core is obtained from ESP for S and P, and it is understood that a constant α is in existence to show the lower bound that a core encodes a substring longer than αm.

At the final stage, comparison is made between the performance of our method and that of other practical compressed indexes [8, 9, 11]. Compressed indexes to comply with text corpus formed from large English texts and DNA sequences are constructed. Thereafter comparison is made with the search time to count occurrences of patterns to correspond to the pattern length. As a result, it is ascertained that the proposed index is efficient enough among these benchmarks in case the pattern is long enough to accomplish the construction of the indexes.

2 Preliminaries

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 $\{a\}^*$ 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 $\log^* n$ indicates the maximum number of logarithms satisfying $\log \log \cdots \log n \geq 1$. For instance, $\log^* n = 5$ for $n = 2^{65536}$. We thus treat $\log^* n$ as a constant.

We assume that any context-free grammar G is *adimissible*, i.e., G derives just one string. For a production rule $X \to AB \cdots C$, symbol X is called *variable*. If G derives a string w, the derivation is represented by a rooted ordered tree, called the *parsing tree* of G. The *size of G* is the total length of strings in the right hand sides of all production rules, and is denoted by $|G|$. The optimization for the *grammar-based compression* is to minimize the size of G deriving a given string w. For the approximation ratio of this problem, see [1, 10, 12, 13].

We consider a special parsing tree of CFG constructed by *edit sensitive parsing* by [4], which is based on a transformation of string called *alphabet reduction*. A string $S \in \Sigma^*$ of length n is partitioned into maximal nonoverlapping substrings of three types; Type1 is a maximal repetition of a symbol, Type2 is a maximal substring longer than $\log^* n$ not containing any repetition, and Type3 is any other short substring. 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 ℓ. Then $label(S[i]) = 2\ell + bit(\ell, S[i])$. By this, a string $label(S)$ is obtained as the sequence of such $label(S[i])$.

For the resulting $label(S)$, $label(S[i]) \neq label(S[i+1])$ if $S[i] \neq S[i+1]$ for any i (See the proof by [4]). 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.

After the final iteration of alphabet reduction, the alphabet size is reduced to at most 6 like $\{0, \cdots, 5\}$. 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. This process is illustrated for a concrete string S in Fig. 1.

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 maximal, i.e., $label(S[i]) > label(S[i-1]), label(S[i+1])$. Following this, we select any position j as a landmark if $label(S[j])$ is minimal and both $j-1, j+1$ are not selected yet. We also display this selection of landmarks in Fig. 1.

Edit sensitive parsing: After computing final string $label(S)$ and its 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.

	a	d	e	g	h	e	c	a	d	e	g
(1) string in binary	000	011	100	110	111	100	010	000	011	100	110
(2) label	–	001	000	011	001	000	011	010	001	000	011
(3) label as integer	–	1	0	3	1	0	3	2	1	0	3
(4) final label & landmark	–	$\boxed{1}$	0	$\boxed{2}$	1	$\boxed{0}$	1	$\boxed{2}$	1	0	$\boxed{2}$

Figure 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.

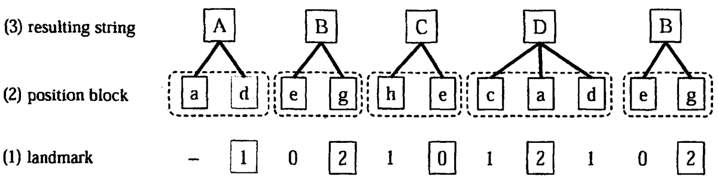

Figure 2: Single iteration of ESP: The line (1) is the computed final labels and landmarks. (2) shows the groups of all positions in s having two or three around the landmarks. (3) is the resulting string $ABCDB$, and the production rules $A \to ad$, $B \to eg$, etc.

Since $label(S) \in \{0, 1, 2\}^*$ contains no repetition, 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 for a same substring must be replaced by a same symbol. This replacement is called *edit sensitive parsing* (ESP). We illustrate single iteration of ESP for determined blocks in Fig. 2.

Finally, we mention 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 remaining 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.

If S is partitioned into $S_1, \ldots, S_k$ of Type1, Type2, or Type3, after parsing them, all the transformed strings S_i' are concatenated together. This process is iterated until a tree for S is constructed. By the parsing manner, we can obtain a balanced $2-3$ tree, called *ESP tree*, in which any internal node has two or three children.

3 Algorithms and Data Structures

3.1 Basic notions

A set of production rules of a CFG is represented by a directed acyclic graph (DAG) with the root labeled by the start symbol. In Chomsky normal form hereby taken up, each internal node has respectively two children called the left/right child, and each edge is also called the left/right edge. An internal node labeled by X with left/right child labeled by A/B is corresponding to the production rule $X \to AB$. We note that this correspondence is one-to-one so that the DAG of a CFG G is a compact representation of the parsing tree T of G. Let v be a node in T, and the subtree of v is the induced subgraph by all descendant of v. The parent, left/right child, and variable on a node v is denoted by $parent(v)$, $left(v)/right(v)$, and $label(v)$, respectively.

With respect to an ordered binary tree T, a node v is called the *lowest right ancestor* of a node x and is denoted by $lra(x)$, provided that v is the lowest ancestor so that the path from v to x will contain at least one left edge. If x is a node in the right most path in T, $lra(x)$ is undefined. Otherwise, $lra(x)$ is uniquely decided. The subtree of x is *left adjacent* to the subtree of y provided that $lra(x) = lla(y)$, thus the *adjacency in the right* is similarly defined.

***fact* 1** For an ordered binary tree, a node y is right adjacent to a node x iff y is in the left most path from $right(lra(x))$, and y is left adjacent to x iff y is in the right most path from $left(lla(x))$.

```
Algorithm ESP-COMP
Input: a string S.
Output: a CFG represented by D deriving S.
   initialize D;
   while(|S| > 1)
      for-each(X_k → X_i X_j produced in same level of ESP)
         sort all X_k → X_i X_j by (i,j);
         rename all X_k in S by X_ℓ, the rank of sorted X_k → X_i X_j;
         update D for renovated X_ℓ → X_i X_j;
   return D;

procedure ESP(S,D)
   compute one iteration of ESP for S;
   update D;
   return the resulting string;
```

Figure 3: The compression algorithm to output a dictionary D for a string S. We assume the reverse dictionary D^R.

3.2 Pattern embedding on parsing tree

For two parsing trees of strings P and S, if there is a common subtree for them, then its root variable is called a *core*. It is shown that with respect to each of strings P and S, these ESP trees concerning a same naming function contain a sufficiently large core X provided S contains P. This property is available as a necessary condition in searching P. In other words, any occurrence of P in S is restricted in a region around X.

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 $\alpha|P|$.

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.

Two algorithms are developed for compression and search based on Lemma 1 and 2. At first, since any ESP tree is balanced $2-3$ tree, each production rule is of $X \to AB$ or $X \to ABC$. The latter is identical to $X \to AB'$ and $B' \to BC$. Assumption is hereby made exclusively with Chomsky normal form. A data structure D to access the digram XY from a variable Z associated by $Z \to XY$ is called a *dictionary*. In the meantime, another data structure D^R to compute the reverse function $f(XY) = Z$ is called a *reverse dictionary*.

ESP-COMP is described in Fig. 3 with a view to computing the ESP tree of a given string. This algorithm outputs the corresponding dictionary D. The reverse dictionary D^R is required to replace different occurrences of XY by means of a common variable Z. This function, which can be developed by a hash function with high probability [6], requires large extra space regardless of such a circumstance. In the next subsection, we propose a method to simulate D^R by D. The improvement brought about as above makes it possible to compress a given pattern for the purpose of obtaining the core exclusively by D.

ESP-SEARCH is described in Fig. 4 to count occurrences of a given pattern P in S. To extract the sequence of cores, P is also compressed by *ESP-COMP* referring to D^R for S. Furthermore if XY is undefined in D^R, a new variable is produced and D^R is updated. Then *ESP-SEARCH* gets the sequence of cores, $X_1, \ldots, X_k$ to be embedded on the parsing tree of S. The algorithm checks if X_i is left adjacent to X_{i+1} for all $i = 1, \ldots, k-1$ from a node v labeled by X_1. As we propose several data structures in the next subsection, we can access to all such v randomly. Thus, the computation time is faster than the time to traverse of the whole ESP tree, which is proved by the time complexity.

Lemma 3 If we assume the reverse dictionary D^R with constant time access, the running time of *ESP-COMP* is $O(u)$ and the height of the ESP tree is $O(\log u)$ for the length of string, u.

Lemma 4 *ESP-SEARCH* correctly counts the occurrences of a given pattern in the ESP tree of a text.

Algorithm ***ESP-SEARCH***
Preprocess: $D \leftarrow$*ESP-COMP*(S) for text S.
Input: a pattern P.
Output: the number of occurrences of P in S

```
count ← 0 and (X_1,...,X_k) ← FACT(P,D);
for-each(v satisfying label(v) = X_1)
   i ← 2, t ← right(lra(v)), and type ← true;
   while(i ≤ k)
      if(a left descendant v' of t satisfies label(v') = X_i)
         v ← v', t ← right(lra(v)), and i ← i + 1;
      else type ← false, and break;
   if(type = true), count ← count + 1;
return count;
```

procedure $FACT(P, D)$

```
compute the variable by CORE(P,D) which encodes P[i,j];
recursively compute the variables
   CORE(pre(P),D) for pre(P) = P[1,i-1] and
   CORE(suf(P),D) for suf(P) = P[i+1,|P|];
return all variables from the left occurrence;
```

procedure $CORE(P, D)$

```
ℓ ← 1 and r ← |P| = m;
while(|P| > 1 and ℓ < r)
   P ← ESP(P,D)
   ℓ ← (ℓ + ⌈log*n⌉ + 5) and r ← r − 5;
return the symbol P[1];
```

Figure 4: The pattern search algorithm from the compressed text represented by a dictionary D. We assume the reverse dictionary D^R again.

3.3 Compact representation for ESP

We propose compact data structures used by the algorithms. These types of improvement are achieved by means of two techniques: one is the decomposition of DAG representation into left/right tree, and the other is the simulation of the reverse dictionary D^R by the dictionary D with an auxiliary data structure. First the decomposition of DAG is considered. Let G be a DAG representation of a CFG in Chomsky normal form. By introducing a node v together with addition of left/right edges from any sink of G to v, G can be modified to have the unique source and sink.

***fact* 2** Let G be a DAG representation with single source/sink of a CFG in Chomsky normal form. For any in-branching spanning tree of G, the graph defined by the remaining edges is also an in-branching spanning tree of G.

An in-branching spanning tree of G, which is called the *left tree* of G, is concurrently denoted T_L provided that the tree consists exclusively of the left edges. Thus the complementary tree is called the *right tree* of G to be denoted T_R. A schematic of such trees is given in Fig. 5.

When a DAG is decomposed into T_L and T_R, the two are represented by succinct data structures for ordered trees and permutations. Brief description concerning the structures is hereunder made. The bit-string by LOUDS [5] 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. A schematic of the LOUDS representations for T_L and T_R is also given in Fig. 5. For n node tree, LOUDS uses $2n + o(n)$ bits to support the constant time access to the parent, the i-th child, and the number of children of a node, which are required by our algorithm.

For traversing the DAG, we also need the correspondence of the set of nodes in one tree to the one in the other. For this purpose, we employ the succinct data structure for permutations by [7]. For a given permutation P of $N = (0, \ldots, n-1)$, using $(1+\varepsilon)n\log n + o(1)$ bits space, the data structure supports to access to $P[i]$ in $O(1)$ time and $P^{-1}[i]$ in $O(1/\varepsilon)$ time. For instance, if $P = (2, 3, 0, 4, 1)$, then $P[2] = 0$ and $P^{-1}[4] = 3$, that is, $P[i]$ is the i-th member of P and

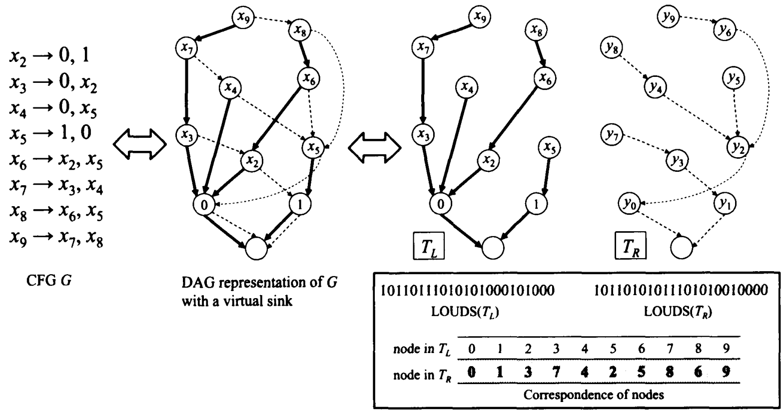

Figure 5: A DAG representing a CFG in Chomsky normal form and its decomposition into two ordered trees with their succinct representations.

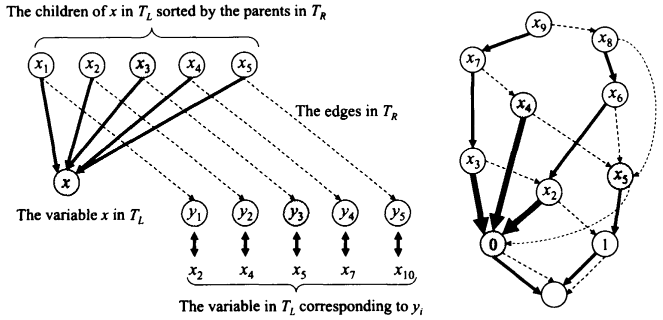

Figure 6: The simulation of D^R using binary search over the nodes of T_L. For each node x in T_L, the children x_is of x are already sorted by the variables in T_L corresponding to the parents of x_is in T_R.

$P^{-1}[i]$ is the position of the member i. For each node i in $LOUDS(T_L)$, the corresponding node j in $LOUDS(T_R)$ is stored in $P[i]$. These are also illustrated in Fig. 5.

In the compression algorithm in Fig. 3, all variables produced in a same level are sorted by the left hands of production rules[1], and these variables are renamed by their rank. Thus, the i-th variable in a DAG coincides with node i in T_L since they are both named in level-order. In accordance with the improvement referred to above, storage can be made with the required correspondence in almost $n \log n$ bits. Devoid of these characteristics, $2n \log n$ bits are required to traverse G.

At the final stage, a method is proposed with a view to simulating the reverse dictionary D^R from the data structures referred to above. Adapting this technique makes it possible to reduce the space for the hash function to compress a pattern. Preprocessing causes the X_k to denote the rank of the sorted X_iX_j by $X_k \to X_iX_j$. Conversely being given a variable X_i, the children of X_i in T_L are already sorted by the indexes of their parents in T_R. Thus the variable X_k associated to X_iX_j can be obtained by using binary search on the children of X_i in T_L, of which depiction is made in Fig. 6. Since LOUDS supports the number of the children and i-th child, access can be made to the middle child X_i in $O(1)$ time. Thus we obtain the following lemma.

Lemma 5 The function $f(XY) = Z$ is computable in $O(\frac{1}{\varepsilon}\log k) = O(\frac{1}{\varepsilon}\log n)$ time for the maximum degree of T_L, k, bounded by the number of variables, n.

Theorem 1 A grammar-based compression G for any string S is represented in $(1+\varepsilon)n\log n + 4n + o(n)$ bits, where n is the number of variables in G. With any pattern P, the number of its occurrence in S is computable in $O(\frac{1}{\varepsilon}(m \log n + occ_c(\log m \log u)))$ time for any $0 < \varepsilon < 1$, where $u = |S|$, $m = |P|$, and occ_c is the number of occurrences of a maximal core of P for S.

[1]In [3], similar technique was proposed, but variables are sorted by encoded strings.

4 Experiments

The experiments are conducted in the environment shown below. 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 are obtained from the text collection in Pizza&Chili Corpus[2] to compare hereto referred method called ESP with other compressed indexes called LZ-index (LZI)[3], Compressed Suffix Array, and FM-index (CSA and FMI)[4]. These implementations are based on [8, 9, 11]. Due to the trade-off in the construction time and the index size, the index referred to above and other methods for reasonable parameters are examined. In our algorithm, setting is made with $\varepsilon = 1, 1/4$ for the permutation. In CSA, the option (-P1:L) means that 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 Huffman-shaped wavelet tree with compressed bit-vectors and L specifies the sampling rate for storing rank values, and (-P7:L) is the uncompressed version. In addition these CSA and FMI do not make indexes for occurrence position. Setting up is made with 200MB texts for each DAN and ENGLISH to evaluate construction time, index size, and search time.

The results in construction time are shown in Fig. 7. It is deduced from these results that the method dealt with at this stage is comparable with FMI and CSA in the parameters in construction time, and slower than LZI. Furthermore it is understood that none of conspicuous difference is seen in construction time so long as the value of ε stand still from 1 to 1/4.

The results of index size are shown in Fig. 8. The results reveal that the index is furthermore compact enough and comparable to CSA(-P1:64). The size of LZI contains the space to locate patterns.

The indexes in Fig. 9 show the time to count all occurrences of a given pattern in the text. The indexes are aligned to accomplish the maximum texts in DNA and ENGLISH (200MB each). Random selection of pattern from the text is made 1000 times for each fixed pattern length, and the search time indicates the average time. In this implementation, we modified our search algorithm so that the core is extracted by a short prefix of a given pattern P and an occurrence of P in S is decided by the single core and the exact match of the remaining substrings by partial decoding of the compressed S. To determine length or the short prefix, the rate 1% of the pattern by preliminary experiments is taken up. In DNA and ENGLISH, our method is faster searchable than LZI and CSA in the parameters for long patterns. The proposed method is liable to be behind the pattern with short length in case of searching, which might be for the reason why the occurrence number is relatively made multiplied, and comparison of variables are executed for the individual occurrences.

From the results referred to above, it is ascertained that the proposed method, which is believed to be subject to settlement of pattern length or parameter settlement, can acquire sufficient performance as index for pattern searching.

References

[1] 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.

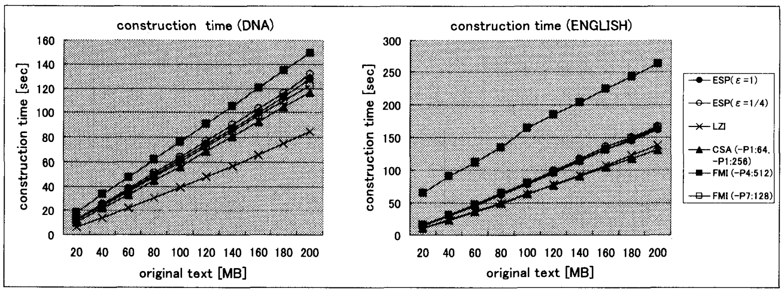

Figure 7: Construction Time.

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

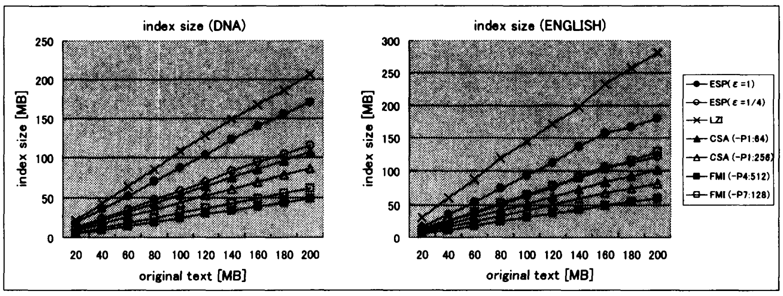

Figure 8: Index Size.

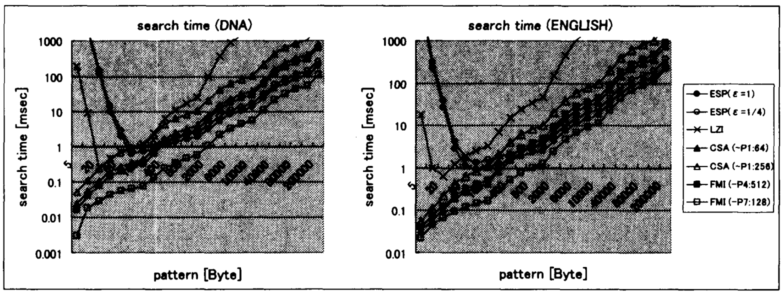

Figure 9: Search Time.

[2] R. Cilibrasi and P.M.B. Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[3] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *MFCS09*, pages 235–246, 2009. to appear in Fundamenta Informaticae.

[4] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. *ACM Trans. Algor.*, 3(1):Article 2, 2007.

[5] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *WEA2006*, pages 134–145, 2006.

[6] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[7] J.I. Munro, R. Raman, V. Raman, and S.S. Rao. Succinct representations of permutations. In *ICALP03*, pages 345–356, 2003.

[8] G. Navarro. Indexing text using the ziv-lempel tire. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[9] G. Navarro and V. Makinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article 2, 2007.

[10] W. Rytter. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.

[11] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[12] H. Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.

[13] 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.

[14] D. Shapira and J.A. Storer. Edit distance with move operations. *J. Discrete Algorithms*, 5(2):380–392, 2007.