PAPER Special Section on Foundations of Computer Science

Scalable Detection of Frequent Substrings by Grammar-Based Compression

Masaya NAKAHARA[†], **Shirou MARUYAMA**[††], ***Nonmembers***, **Tetsuji KUBOYAMA**[†††], ***and*** **Hiroshi SAKAMOTO**[†,††††a)], ***Members***

SUMMARY A scalable pattern discovery by compression is proposed. A string is representable by a context-free grammar deriving the string deterministically. In this framework of *grammar-based compression*, the aim of the algorithm is to output as small a grammar as possible. Beyond that, the optimization problem is approximately solvable. In such approximation algorithms, the compressor based on *edit-sensitive parsing* (ESP) is especially suitable for detecting maximal common substrings as well as long frequent substrings. Based on ESP, we design a linear time algorithm to find all frequent patterns in a string approximately and prove several lower bounds to guarantee the length of extracted patterns. We also examine the performance of our algorithm by experiments in biological sequences and other compressible real world texts. Compared to other practical algorithms, our algorithm is faster and more scalable with large and repetitive strings.

key words: *pattern discovery, grammar-based compression, edit-sensitive parsing*

1. Introduction

This paper tackle the problem of finding all frequent substrings in the input string, which requires square time for input length by a trivial algorithm, where we call a pattern is frequent if it appears in the string at least twice. We propose a rapid approximation algorithm based on *grammar-based compression* where a small context-free grammar (CFG) is required to represent an input string uniquely. The preliminary results were partially presented in [1] and this paper is its extended version.

We first outline the framework of grammar-based compression and recent work related to our study. Any CFG G is assumed to be *admissible* [2], i.e., G derives just one string w and a derivation tree for w is unique. Here, the set of production rules of G is regarded as a compression of w. For instance, a string $w = abab \cdots ab = (ab)^{16}$ is represented by an admissible CFG with $D = \{S_0 \to S_1S_1,\ S_1 \to S_2S_2,\ S_2 \to S_3S_3,\ S_3 \to S_4S_4,\ S_4 \to ab\}$, which is the set of production rules. Because we can assume that any production rule is of the form $A \to BC$ for some two symbols B and C, the set D is equivalent to the decodable sequence $S_1S_1S_2S_2S_3S_3S_4S_4ab$, which is shorter than the original string w. Such a data structure to access D is called *dictionary*.

For a string w of length u, finding a smallest CFG is NP-hard; it is also known that this problem is hard to approximate within a constant factor [3]. A good news is that several compression algorithms have been proposed to guarantee $O(\log u)$-approximation ratio to the optimum compression [4]–[6], and this bound is indeed considered to be *tight* because of the relationship with the difficult optimization problem mentioned in [3].

Data compression is closely related to string comparison and clustering. Li et al. [7] and Cilibrasi and Vitanyi [8] introduced clustering by compression based on *normalized compression distance* (NCD). Using the grammar-based compression, NCD briefly expresses that two strings w_1, w_2 are considered to be similar if the difference of D_1 for w_1 and D_2 for w_2 is sufficiently small with respect to a reasonable compression algorithm. Although this measure is computable by several practical compressors, there seems to be a trade-off between the accuracy of clustering and the consumption of memory due to the claim that two occurrences of the same substring α in $x\alpha y\alpha z$ should be replaced by the same variable A, which is associated with the production rule $A \to \alpha$. This claim is intended to prevent the number of production rules, i.e., the size of CFG, from increasing, and such variables A are expected to encode a frequent pattern. Indexing like suffix tree and similar techniques [9], [10] makes it possible to replace most occurrences of the same substring by the same variable. Such methods, however, require $\Omega(u)$ work space, so it is impractical with gigabytes of strings because of a large constant factor hidden in $\Omega(u)$.

To develop a practical pattern detection algorithm based on grammar-based compression, we focus on the technique of *edit-sensitive parsing* (ESP) [11] proposed to approximate a variant of edit distance where a moving operation for any substring with unit cost is permitted. For instance, xyz is transformed to yxz by a single operation for any strings x, y, z. The edit distance *with move* problem is NP-hard, and the distance was proved to be $O(\log u)$-approximable [12]. Moreover, the harder problem, edit distance *matching* with move, also proved to be almost $O(\log u)$-approximable by the embedding of

Manuscript received March 28, 2012.
Manuscript revised July 28, 2012.

[†]The authors are with Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
[††]The author is with Kyushu University, Fukuoka-shi, 819–0395 Japan.
[†††]The author is with Gakushuin University, Tokyo, 171–8588 Japan.
[††††]The author is with JST PRESTO, Kawaguchi-shi, 332–0012 Japan.
a) E-mail: hiroshi@ai.kyutech.ac.jp

DOI: 10.1587/transinf.E96.D.457

the string into L_1 vector space using ESP [11]. The most important characteristic of ESP in measuring the similarity of strings is roughly explained as follows: an ESP of a string w is a derivation tree for w, which is represented by a dictionary D. Let us consider any two strings w_1, w_2 to be compared. An algorithm computes D_1, D_2 for w_1, w_2 respectively. We assume that all variables are *appropriate*, i.e., $A \to \alpha$, $B \to \alpha \in D_1 \cup D_2$ implies $A = B$. The distance of w_1, w_2 is then approximable by the number of variables appearing in exactly one of D_1 and D_2 computed by ESP. Conversely, any variable in $D_1 \cap D_2$ encodes a common substring in w_1 and w_2. Because ESP minimizes the difference of D_1 and D_2 approximately, a practical algorithm for grammar-based compression is expected to find sufficiently long common substrings as frequent variables in $D_1 \cap D_2$. ESP was diverted to a space-efficient compression algorithm [13] within a good approximation ratio, which is, however, not implemented. We thus modify this compression algorithm to fit our problem of detecting long common substrings, and we show a lower bound to guarantee the length of the extracted pattern.

ESP has a potentially compact representation because it is essentially equal to a binary tree. Moreover, if w is compressible, the number of different variables in D is significantly smaller than the length of w. Thus, compared to other data structures (e.g., suffix tree), grammar-based compression is suitable for our approximation problem. The modified algorithm requires two passes: the first compresses input w and generates D for w; the second locates the positions of any substring encoded by a frequent variable in D.

We summarize our contribution in this paper. As mentioned above, our algorithm outputs an approximation for the occurrences of all maximal frequent substrings. We prove an exact lower bound of the length of such frequent substrings extracted by the algorithm with respect to a reasonable condition. Moreover, as an extension, we prove another lower bound for the general case. The efficiency is also shown by experiments. The proposed algorithm outputs all frequent variables generated in ESP with the occurrence positions and the encoded string lengths. We implement this algorithm and examine its performance for detecting frequent substrings in biological strings and large documents in real world data. Our algorithm is also compared to two detection algorithms: one is SACHICA proposed in [14] and the other is Suffix Array with LCP proposed in [15]. SACHICA is an approximation algorithm, and Suffix Array is an exact algorithm for this problem. We confirm that our algorithm is significantly faster with long strings as well as compressible strings. This scalability is an advantage of our method over other algorithms [14], [15].

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^k with $k \geq 2$ and a symbol a is called a *repetition of* a, and a^+ is its abbreviation when the length is omissible. More generally, α^k is also called repetition of a string α ($|\alpha| \geq 1$). For a string S, $S[i]$ and $S[i, j]$ denote the i-th symbol of S and the substring from $S[i]$ to $S[j]$, respectively. The expression $\log^* n$, the inverse Ackermann function $\alpha_3(n)$, indicates the maximum number of logarithms satisfying $\log\log\cdots\log n \geq 1$. For instance, $\log^* n \leq 5$ for any $n \leq 2^{65536}$. We treat $\log^* n$ as a constant for any n.

We assume that any context-free grammar G is *admissible*, i.e., G derives just one string. For a production rule $X \to \gamma$, symbol X is called *variable*. The set of variables and production rules are denoted by V and D. We also assume that any variable is *appropriate*, i.e., for any $\alpha \in (\Sigma \cup V)^*$, at most one $X \to \alpha \in D$ exists. 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 $S(D) = abcabca$. If G derives a string w, the derivation is represented by a rooted ordered tree. The *size of* G is the total length of strings on the right hand sides of all production rules, and is denoted by $|G|$.

For a derivation tree T and a node x in T, let $L(x)$ denote the label of x. If $v_1, \ldots, v_k$ are the children of x, the concatenation $L(v_1)\cdots L(v_k)$ called the block of x. If x, y are nodes in T and the subtrees on x, y are adjacent in this order, the x, y are called to be adjacent. Similarly, $x_1, \ldots, x_k$ are called to be adjacent if each x_i, x_{i+1} are adjacent. Then, the concatenation of blocks of x_is is called a metablock. We next outline edit-sensitive parsing (ESP) [11]. By this, all metablocks are categorized into three types as follows.

Throughout this paper, n denotes $|\Sigma \cup V|$ for a current set V of variables, where initially $V = \emptyset$ and clearly $n \leq |S|$ for any input string $S \in \Sigma^*$. The string S 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 repetitions. Each a_i^+ is called Type1 metablock, w_i is called Type2 metablock if $|w_i| \geq \log^* n$, and any 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. Thus, any metablock is longer than or equal to two symbols.

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. When 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 $t_i \to S[2i-1, 2i] \in D'$ and $t \to S[k-2, k] \in D'$ where t_0 denotes the empty string for $k = 3$.
2. When 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 $t_i \to s_i \in D'$.
3. For any case in the above, all blocks are replaced by appropriate variables, i.e., all occurrences of a same substring are replaced by a same variable.

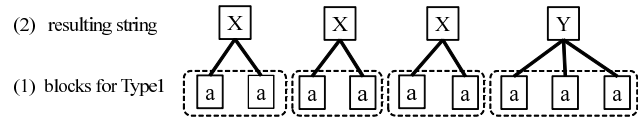


Fig. 1 Parsing for Type1 metablock: Line (1) is an original Type1 metablock $S = a^9$ with its blocks. Line (2) is the resulting string XXXY, and the production rules X → aa and Y → aaa. Any Type3 string is parsed analogously.

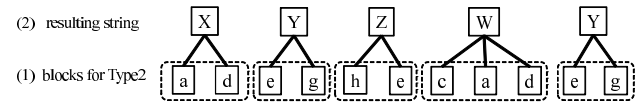


Fig. 2 Parsing for Type2 metablock: Line (1) is an original Type2 metablock 'adeghecadeg' with its blocks by alphabet reduction, where the definition is omitted in this paper. Line (2) is the resulting string XYZWY, and the production rules X → ad, Y → eg, etc.

Cases (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 $S = a^9$, $S' = x^3y$ and $x \to a^2$, $y \to aaa \in D'$. In Case (c), we omit the description of alphabet reduction [11] because the details are unnecessary in this paper.

Case (b) is illustrated in Fig. 1 for a Type1 string, and the parsing manner in Case (a) is obtained by ignoring the last three symbols in Case (b). Parsing for Type3 is analogous. Case (c) for Type2 is illustrated in Fig. 2.

Finally, we define ESP for the general case of $S \in (\Sigma \cup V)^*$ 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 $|S| = 1$. 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 the bases of our study.

Lemma 1: (Cormode and Muthukrishnan [11]) $ET(S)$ can be computed in time $O(u \log^* u)$ time for $u = |S|$, and the height of $ET(S)$ is $O(\log u)$.

Lemma 2: (Cormode and Muthukrishnan [11]) Let S be Type2 partitioned into $S = s_1 s_2 \cdots s_k$ 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 $1 \le i \le k - 5$. 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 $0 \le j \le k - \log^* |S| - 5$.

3. Approximation of Pattern Detection

We first define the approximation problem of pattern detection by grammar-based compression.

Definition 1: Let D be a set of production rules whose derivation tree is T deriving $S \in \Sigma^*$. A substring P of S is said to be approximated by a node label A in T within $\delta = \frac{|A(D)|}{|\alpha|}$ iff for each occurrence $S[i, j] = P$, there exists a node v of T with $L(v) = A$ deriving $S[i + k, j - \ell] = P'$ such that $P = xP'y$ for some $k = |x|$ and $\ell = |y|$.

We show that any substring P of S is approximated by a variable in D generated by ESP within a sufficiently large δ depending on $|P|$ and offset positions k, ℓ are fixed for each P. For $T = ET(S)$, an internal node label is a variable and a leaf label is an alphabet symbol. For a node v in T, let $yield(v)$ denote the substring of S derived from $L(v)$. We call that a node v in T derives $S[n, m]$ if $yield(v) = S[n, m]$. Using this notation, we define a measure for a maximal common subtree in $T = ET(S)$. A node label A in T is called a *core* of P if the condition is satisfied for some fixed k, $\ell \ge 0$: when $S[n, m] = P$, there exists a node v in T such that $L(v) = A$ and v derives $S[n + k, m - \ell]$. Intuitively, a core is a necessary condition for an occurrence of P. Note that at least one core exists for any P because we can take any $P[i]$ as a core. The length of core A means the length of string derived from A. We show that for any substring P in S, $ET(S)$ has a maximal core that approximates any occurrence of P within a sufficiently large δ. The following lemma guarantees a lower bound of δ with a restriction.

Lemma 3: Assuming S contains no repetition longer than $\log^* n$, there exists a constant $\delta \ge \frac{1}{24(\log^* n + 10)}$ such that for any occurrence of substring P in S, the length of maximal core of P is longer than $\delta|P|$.

proof. Taking any substring P of S, P is uniquely divided into $P = \alpha\beta\gamma$ for the first metablock α, the last metablock γ, and the concatenation β of all other metablocks. When α is Type1, by the assumption, $|\alpha| \le \log^* n$. When α is Type3, $|\alpha|$ is also at most $\log^* n$. When γ is also Type1 or 3, any occurrence of β inside $P = \alpha\beta\gamma$ is transformed to the same string β'.

When α/γ is Type2, let $\alpha = \alpha_1 \cdots \alpha_\ell$ partitioned by alphabet reduction for some α_i ($2 \le |\alpha_i| \le 3$), and let $\gamma = \gamma_1 \cdots \gamma_r$ similarly. By Lemma 2 and the definition of ESP, for any xzy such that $|x| \ge \log^* n + 5$ and $|y| \ge 5$, any occurrence of z inside xzy is transformed to the same string z' by a single iteration of ESP. By this characteristic, for $i = \min(\log^* n + 5, \ell)$ and $j = \min(5, r)$, any occurrence of $\alpha_i \cdots \alpha_\ell \beta \gamma_1 \cdots \gamma_j$ inside $\alpha\beta\gamma$ is transformed to the same string β'.

In all cases, we can conclude that for any occurrence of P, a substring β of P satisfying $P = \alpha\beta\gamma$ and $|\alpha|$, $|\gamma| \le \log^* n + 5$ is transformed to the same string. Moreover, this analysis holds on the resulting string β recursively, since the input string contains no repetition s^k of a string $s \in \Sigma^*$ for some $k > \log^* n$ and $|s| \ge 1$.

When $ESP^k(S, D)$ is completed for some k, the resulting same string becomes a variable X. We estimate the length of encoded string by X. At each $(S', D') = ESP^i(S, D)$, we note that any variable in S' encodes a string of length at least 2^i because $T = ET(S)$ is a balanced 2-3 tree. Thus, as the above k, we can choose any integer k satisfying

$$(2\log^* n + 10)(1 + 3 + \cdots + 3^{k-1}) < (\log^* n + 10)\, 3^{k+1} \le |P|,$$

that is,

$$k = \left\lfloor \log_3 \frac{|P|}{6(\log^* n + 10)} \right\rfloor \geq \log \frac{|P|}{24(\log^* n + 10)}.$$

Any variable generated in k-th iteration of ESP derives a string of length at least 2^k. We can, therefore, take a $\delta \geq \dfrac{1}{24(\log^* n + 10)}$ such that a variable X generated in $ESP^k(S, D)$ satisfies $|D(X)| \geq \delta|P|$. *Q.E.D.*

Generally, we can obtain the following bound for core length.

Lemma 4: For any occurrence of substring P in S, the length of maximal core of P is longer than $\frac{|P|}{2\log^2 |P|}$.

proof. Let $|P| = m$ and $P = \alpha\beta\gamma$ for the first metablock α and the last metablock γ. For an input string S, let $ESP(S, D) = (S_1, D \cup D_1)$. By the definition of edit-sensitive parsing, P occurs in S iff $S_1[\ell_1, r_1](D_1) = \beta$, $S_1[\ell_1 - k_1, \ell_1 - 1](D_1)$ contains α as its suffix, and $S_1[r_1 + 1, r_1 + k'_1](D_1)$ contains γ as its prefix for some ℓ_1, r_1, k_1, k'_1. Let $p_1 = \alpha$, $s_1 = \gamma$, and $S_1[\ell_1, r_1] = \beta'$. By iterating the above parsing until $|\beta'| = 1$, we obtain a sequence $p_1, p_2, \ldots, p_h, s_h, \ldots, s_2, s_1$ of strings consisting of cores of P and $h \leq \log m$.

If p_i (or s_i) is a repetition X^k for some $X \in \Sigma \cup V$ and $k \geq 2$. By a single iteration of the left aligned parsing, p_i is transformed into $X_1^{\lfloor k/2 \rfloor} X$ if k is odd. Thus, in worst case, p_i is transformed into $\bar{p}_i = X_\ell X_{\ell-1} \cdots X_1 X$ such that $|\bar{p}_i| = \ell + 1 \leq \log |p_i| \leq \log m$. If p_i is not repetition, $|p_i| \leq \log^* n + 5 \leq \log m$, since $\log^* n$ is regarded as a constant. We suppose any p_i is a repetition and p_i is transformed into $\bar{p}_i$ such that $|\bar{p}_i| \leq \log m$ in worst case.

Thus, the sequence $\bar{p}_1, \bar{p}_2, \ldots, \bar{p}_h, \bar{s}_h, \ldots, \bar{s}_2, \bar{s}_1$ satisfies that any symbol in $\bar{p}_i$ and $\bar{s}_i$ is a core of P and $|\bar{p}_i|$, $|\bar{s}_i| \leq \log m$. Since the total length of strings encoded by all $\bar{p}_i$ and $\bar{s}_i$ is m, by *the pigeonhole principle*, at least one symbol in the sequence encodes a substring S' of P such that $|S'| \geq \dfrac{m}{2\log^2 m}$. *Q.E.D.*

For our pattern detection problem, we assume that any pattern has a sufficiently large core in $T = ET(S)$. We show in the next section, however, that such restriction is reasonable by experiments with large strings. We propose two algorithms summarized as follows: the first algorithm *make_dic*(S, D) in Fig. 3 computes $ESP^*(S, D)$ as well as the hash function H to store the length of encoded string by any variable in D. Decoding D with H, the second algorithm *locate_pat* in Fig. 4 records the *position of* any node v such that the variable $L(v)$ is frequent, i.e., $L(v)$ appears in T at least twice where the position of v is the integer n such that $yield(v) = S[n, m]$.

Lemma 5: The time and space complexity of the algorithm *make_dic*(S, D) is $O(u)$ and $O(n)$ for $u = |S|$ and $n = |D|$, respectively.

proof. The time bound is directly derived from Lemma 1.

```
make_dic(S, D)
/* S is the input string and
       D = ∅ is the initial dictionary */
initialize the hash function H(i) ← 1 for 1 ≤ i ≤ |Σ|;
while(|S| > 1){
    (S', D' ∪ D) ← ESP(S, D);
    foreach(X_k → X_i X_j ∈ D'){H(k) ← H(i) + H(j);}
    S ← S', D ← D' ∪ D, D' ← ∅;
}
output (D, H);
```

Fig. 3 Dictionary construction algorithm.

```
locate_pat(S, D)
(D, H) ← make_dic(S, D);
initialize the array Pos[1, n] for n = |D|;
/* Pos[i] indicates the list to store all occurrences of
       substring encoded by X_i */
let T = ET(S) represented by D;
foreach (node v with L(v) = X_j){
    let v' be the lowest left ancestor of v
        with L(left(v')) = X_i;
    let p be the position of v';
    if(X_j is frequent) append p + H(i) to Pos[j];
}
output Pos;
```

Fig. 4 Substring location algorithm.

In ESP, random access is not required for the input string. Thus, the space complexity depends on D, H, and the reverse function of D to access the variable Z from a given digram XY associated with $Z \to XY$. The sizes of D and H are both $O(n)$. Using the Karp-Miller-Rosenberge labeling algorithm [16], we can refer to the variable Z from a string γ with $Z \to \gamma \in D$ in $O(1)$ time with linear space[†]. *Q.E.D.*

Any $ET(S)$ is a 2-3 tree where any internal node and its children correspond to either $Z \to X_i X_j$ or $Z \to X_i X_j X_k$. We can simulate $ET(S)$ by a labeled binary tree introducing an intermediate variable Y to represent $Z \to X_i Y$ and $Y \to X_j X_k$ instead of $Z \to X_i X_j X_k$. Thus, we adopt this binary tree representation as $ET(S)$. Additionally, we define some notation for the binary tree. The parent, left child, and right child of v are denoted by *parent*(v), *left*(v), and *right*(v), respectively. An edge connecting $(v, left(v))$ is called a *left edge*, and a right edge is analogous. Node v is called a *left ancestor* of x if v is an ancestor of x satisfying that the path from v to x contains at least one right edge. These notions are illustrated in Fig. 5. Next we analyze algorithm *locate_pat*(S, D) in Fig. 4.

Lemma 6: The time and space complexity of the algorithm *locate_pat*(S, D) is $O(u)$ and $O(n)$ for $u = |S|$ and $n = |D|$, respectively.

[†]Practically, we use a hash function to access the reverse dictionary.

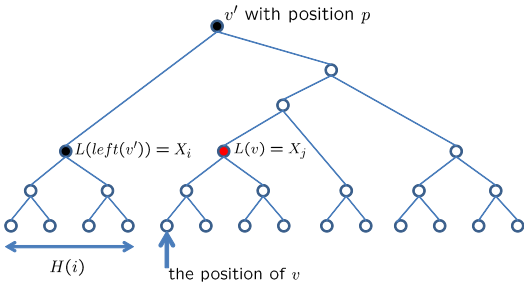


Fig. 5 Relation of nodes and their positions in binary tree.

proof. The times to compute $ESP(S, D)$ and find all frequent variables are both $O(u)$. Because the lowest left ancestor v' of any v is unique, we can compute the position of v from v' and $left(v')$ in $O(1)$. Thus, the time complexity of $locate\_pat(S, D)$ is $O(u)$. On the other hand, this algorithm requires random access to only D, H and the bit sequence to indicate at most n frequent variables. Therefore, the space complexity is $O(n)$. *Q.E.D.*

Theorem 1: The algorithm $locate\_pat(S, D)$ finds all occurrences of core whose length is at least $\delta|P|$ in $O(u)$ time and $O(n)$ space where $u = |S|$, $n = |D|$, $\delta \geq \frac{1}{24(\log^* n+10)}$ if S contains no repetition longer than $\log^* n$ and $\delta \geq \frac{1}{2\log^2 |P|}$ otherwise.

Since $T = ET(S)$ is balanced by Lemma 1, given two nodes v, v' in T with the same label, we can compute a maximal common substring $S[k, \ell] = S[k', \ell'] = P$ in $O(\log u + m)$ time such that $S[k, \ell]$ contains $yield(v)$ and $S[k', \ell']$ contains $yield(v')$ where $u = |S|$ and $m = |P|$. The next section examines the performance of our algorithm and estimates the length $\delta|P|$ for real world data.

4. Computational Experiments

We implemented our algorithm and compared its performance with the practical algorithm SACHICA3.4[†] proposed in [14] based on radix sort as well as SA (suffix array) with LCP[††] (Longest Common Prefix) [15] to extract all pattern occurrences.

The environment of experiment is 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.

As benchmark data, we obtained highly repetitive strings from repetitive corpus (Real)[†††], which consists of DNA sequences (i.e., *E_Coli, Para, Cere, influenza)*, source codes (i.e., *coreutils, kernel*), and natural language texts (i.e., *einstein.de, einstein.en, world leaders*). From these, as natural language texts, we selected *einstein.en* (200 MB) with 139 different characters consisting of Wikipedia articles about Albert Einstein, and *world_leaders* (40 MB) with 89 different characters consisting of CIA World Leaders from January 2003 to December 2009. We also selected *E_Coli* (100 MB) as DNA sequence with 15 different characters. Additionally, we obtain the *rice chromosome 1*[††††] (40 MB) as a DNA sequence, which is expected to have long frequent substrings. For DNA sequences, all occurrences of unknown character 'N', which is a wildcard for any of 'A, C, G, T', were deleted as preprocessing.

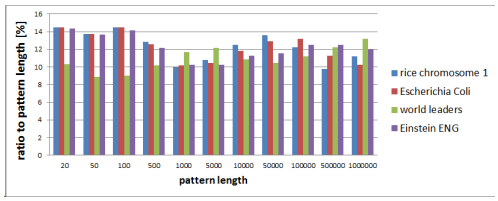


Fig. 6 Ratio between the length of extracted patterns and original patterns in text.

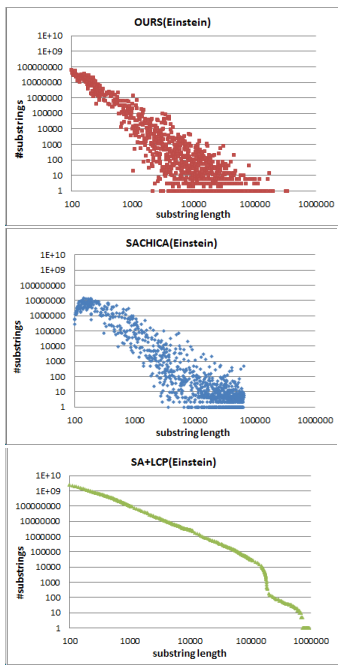


Fig. 7 Comparison of detected patterns in *einstein.en* ($L = 100$).

We start with the evaluation of extracted pattern in aspect of the length. In Fig. 6, the X-axis is the length of pattern occurring in text, and the Y-axis is the ratio of the length of extracted pattern corresponding to the pattern occurrence. A pattern is randomly selected as $S[i, j]$ for the input string S. The result shows 1000 times iteration for each specified length. For any types of benchmark text, our algorithm extracts sufficiently long substring of any pattern (approximately, the ratio is from 8% to 15%). This strengthens our theoretical results.

Next, we report the performance of our algorithm compared with SACHICA and SA. The parameter L denotes the minimum length of detected patterns. Fig. 7 shows the scatter diagram for *einstein.en* (200 MB). The X-axis is

[†]http://research.nii.ac.jp/ uno/code/sachica.html
[††]http://homepage3.nifty.com/wpage/
[†††]http://pizzachili.dcc.uchile.cl/repcorpus.html
[††††]http://rgp.dna.affrc.go.jp/E/IRGSP/Build5/build5.html

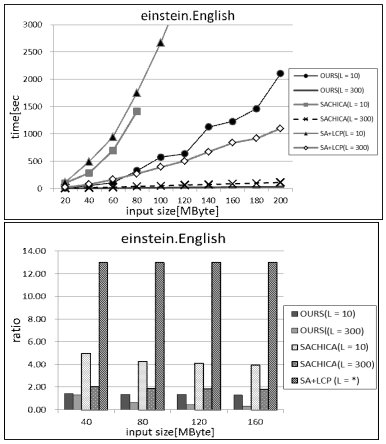


Fig. 8 Comparison of time/space consumption in *einstein.en*.

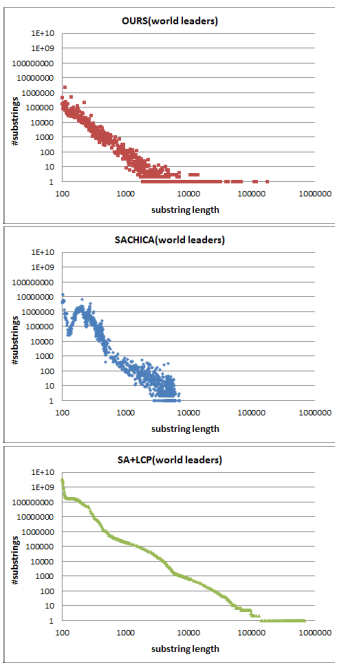


Fig. 9 Comparison of detected patterns in *world_leaders* ($L = 100$).

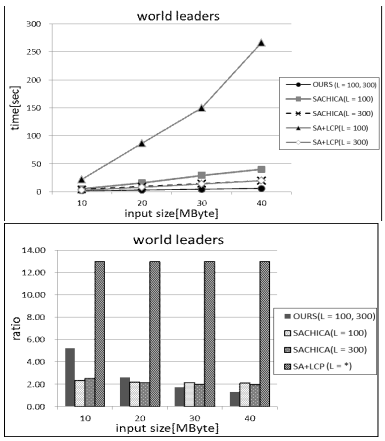


Fig. 10 Comparison of time/space consumption in *world_leaders*. The results for $L = 10, 50$ are not shown because SACHICA did not terminate within the time scale.

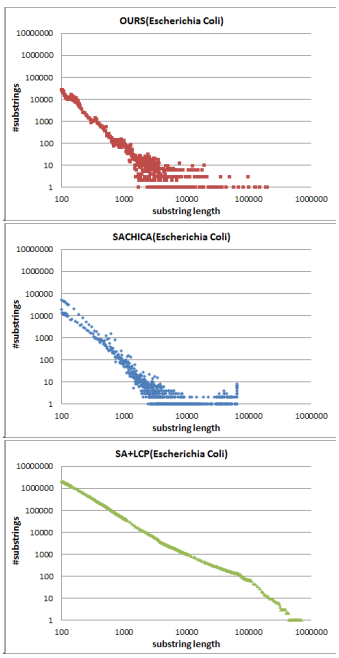


Fig. 11 Comparison of detected patterns in *E_Coli* ($L = 100$).

the length of detected patterns by three algorithms, and the Y-axis is the number of detected patterns, where both axes use log-scales. We note that only SA is the exact algorithm, and other two are approximation algorithms. We should also note that SACHICA can find similar patterns with any Hamming distance, which is not yet computable by our algorithm, which is one of our important future works.

Figure 8 shows the time and space consumption. The length given to SACHICA is set to $L = 10, 50, 100, 300$ and the Hamming distance is set to zero. In Fig. 8, only the case $L = 10, 300$ are shown for readability and the mark $L = *$ in SA denotes that the space consumption of SA does not depend on L, since the indexes of all suffixes are stored in memory.

There is a trade-off between the quality of solution and the consumption of resource. By this result, it is hard to execute SA for huge texts, and so is in SACHICA depending on the parameter to indicate the minimum length of detected patterns. However, our algorithm performs well on all parameters. In particular, we can confirm that the performance of our algorithm is weaker than SACHICA in short patterns, and is stronger in very long patterns.

In order to exemplify the characteristics, we display the results for other types of text in Fig. 9 and Fig. 10 for *world_leaders*, in Fig. 11 and Fig. 12 for *E_Coli*, and in Fig. 13 and Fig. 14 for *rice chromosome 1*. These demonstrate the scatter diagram and the time and space consumption, respectively, and we conclude that our algorithm works well on input texts of varied alphabet sizes.

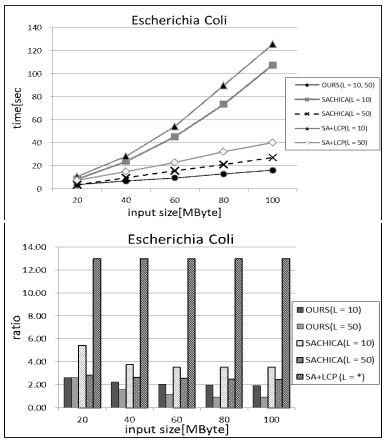


Fig. 12 Comparison of time/space consumption in *E_Coli*.

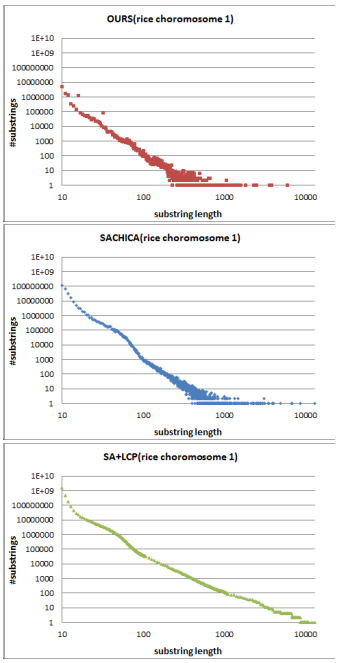


Fig. 13 Comparison of detected patterns in *rice chromosome 1* ($L = 10$).

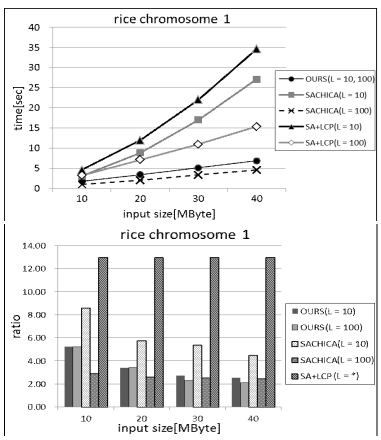


Fig. 14 Comparison of time/space consumption in *rice chromosome 1*.

5. Conclusion and Future Work

We proposed a linear time algorithm for frequent pattern discovery in the preliminary paper [1], based on grammar-based compression and edit-sensitive parsing. In this paper we extended our result in both theoretical and practical aspects. Originally, ESP was intended to solve an intractable variant of the edit distance problem between two strings. A weak point of our method is that many patterns are likely ignored because our algorithm counts only the occurrences of patterns that are specifically named by variables in the derivation tree. In compensation for the shortage of detected patterns, our algorithm abstracts input strings in linear time by the length of the lower bound for detected patterns. Experiments show that this characteristic is especially-pronounced in large and compressible texts, which are expected to contain long frequent patterns. Thus, with more work, our method will make it possible to detect plagiarism from large document collections and interesting patterns from biological sequences using a novel technique in [17]. An important future work is to improve our method to handle the stream data, which is partially developed by an online compression algorithm for grammar-based compression in [18] we recently proposed.

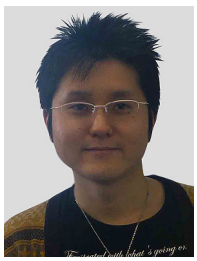
Acknowledgment

We thank anonymous referees for their comments.

References

[1] M. Nakahara, S. Maruyama, T. Kuboyama, and H. Sakamoto, "Scalable detection of frequent substrings by grammar-based compression," DS 2011, pp.236–246, Espoo, Finland, Oct., Springer Verlag, Berlin, 2011.
[2] J.C. Kieffer and E.-H. Yang, "Grammar-based codes: A new class of universal lossless source codes," IEEE Trans. Inf. Theory, vol.46, no.3, pp.737–754, 2000.
[3] E. Lehman and A. Shelat, "Approximation algorithms for grammar-based compression," SODA 2002, pp.205–212, San Francisco, CA, ACM-SIAM, USA, 2002.
[4] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," IEEE Trans. Inf. Theory, vol.51, no.7, pp.2554–2576, 2005.
[5] W. Rytter, "Application of Lempel-Ziv factorization to the approximation of grammar-based compression," Theor. Comput. Sci., vol.302, pp.211–222, 2003.
[6] H. Sakamoto, "A fully linear-time approximation algorithm for grammar-based compression," J. Discrete Algorithms, vol.3, no.2-4, pp.416–430, 2005.
[7] M. Li, X. Chen, X. Li, B. Ma, and P.M.B. Vitanyi, "The similarity metric," IEEE Trans. Inf. Theory, vol.50, no.12, pp.3250–3264, 2004.
[8] R. Cilibrasi and P.B.M. Vitanyi, "Clustering by compression," IEEE Trans. Inf. Theory, vol.51, no.4, pp.1523–1545, 2005.
[9] K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array," ISAAC 2000, pp.410–421, Taipei, Taiwan, Dec., LNCS 1969, Springer Verlag, Berlin, 2000.
[10] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit, "Engineering a

compressed suffix tree implementation," ACM J. Experimental Algorithmics, vol.14, article no.2, 2009.
[11] G. Cormode and S. Muthukrishnan, "The string edit distance matching problem with moves," ACM Trans. Algor., vol.3, no.1, article no.2, 2007.
[12] D. Shapira and J.A. Storer, "Edit distance with move operations," J. Discrete Algorithms, vol.5, no.2, pp.380–392, 2007.
[13] H. Sakamoto, S. Maruyama, T. Kida, and S. Shimozono, "A space-saving approximation algorithm for grammar-based compression," IEICE Trans. Inf. & Syst., vol.E92-D, no.2, pp.158–165, Feb. 2009.
[14] T. Uno, "An efficient algorithm for finding similar short substrings from large scale string data," PAKDD 2008, pp.345–356, Osaka, Japan, May, LNCS 5012, Springer Verlag, Berlin, 2008.
[15] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," CPM 2001, pp.181–192, Jerusalem, Israel, July, LNCS 2089, Springer Verlag, Berlin, 2001.
[16] R.M. Karp, R.E. Miller, and A.L. Rosenberg, "Rapid identification of repeated patterns in strings, trees and arrays," STOC 1972, pp.125–136, Denver, Colorado, ACM, USA, 1972.
[17] K. Lanctot, M. Li, and E.-H. Yang, "Estimating DNA sequence entropy," SODA 2000, pp.409–418, San Francisco, CA, Jan., ACM-SIAM, USA, 2000.
[18] S. Maruyama, M. Takeda, M. Nakahara, and H. Sakamoto, "An online algorithm for lightweight grammar-based compression," CCP 2011, pp.19–28, Palinuro, Italy, June 2011.



Masaya Nakahara received the B.S. and M.S. degree in Computer Science and Systems Engineering from Kyushu Institute of Technology in 2010 and 2012. He is now working at NaU Data Institute, Inc.

Shirou Maruyama received the B.S. degree in Engineering from Fukuoka University in 2006, the M.S. degree in Computer Science and Systems Engineering from Kyushu Institute of Technology in 2008, and Ph.D. in Information Science from Kyushu University in 2012. He is now working at Preferred Infrastructure, Inc.

Tetsuji Kuboyama received the B.Eng. and M.Eng. from Kyushu University in 1992 and 1994, and the Ph.D. from University of Tokyo in 2007. From 1997 to 2008, he was a research associate at Center for Collaborative Research, University of Tokyo. From 2008 to present, he is an associate professor at computer centre, Gakushuin University. He is a member of JSAI and IPSJ.

Hiroshi Sakamoto received the B.S. degree in Physics, the M.S. and Dr. Sci. degree in Information Systems all from Kyushu University in 1994, 1996, and 1998, respectively. He received JSPS Research Fellowships for Young Scientists from 1996 to 1998. From Jan. 1999 to Oct. 2003, he was a research associate of Department of Informatics, Kyushu University. He is currently an associate professor of Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology. From Oct. 2009, he is concurrently a researcher on JST PRESTO program. He is a member of JSAI and DBSJ.