PAPER Special Section on Foundations of Computer Science

Context-Sensitive Grammar Transform: Compression and Pattern Matching

Shirou MARUYAMA[†a)], **Youhei TANAKA**[††b)], ***Nonmembers***, **Hiroshi SAKAMOTO**[††c)], ***Member***, ***and*** **Masayuki TAKEDA**[†d)], ***Nonmember***

SUMMARY A framework of context-sensitive grammar transform for speeding-up compressed pattern matching (CPM) is proposed. A greedy compression algorithm with the transform model is presented as well as a Knuth-Morris-Pratt (KMP)-type compressed pattern matching algorithm. The compression ratio is a match for gzip and Re-Pair, and the search speed of our CPM algorithm is almost twice faster than the KMP-type CPM algorithm on Byte-Pair-Encoding by Shibata et al. [18], and in the case of short patterns, faster than the Boyer-Moore-Horspool algorithm with the stopper encoding by Rautio et al. [14], which is regarded as one of the best combinations that allows a practically fast search.

key words: *compressed pattern matching, grammar-based compression, KMP automaton*

1. Introduction

In this paper, we propose a framework of context-sensitive grammar (CSG) transform for good compression ratio and fast compressed pattern matching. For this objective, we introduce a subclass of CSGs and construct an effective compression algorithm with a special case of the grammar transform model. We also implement Knuth-Morris-Pratt (KMP) pattern matching automaton on the compressed strings and show its performance by experiments. We thus refer to related work in both grammar-based compression and compressed pattern matching.

The framework of grammar-based compression was proposed by Kieffer and Yang [8] to obtain good compression as context-free grammar (CFG) transform, and we would expand the grammar transform by using powerful CSG. In this problem, we assume that any grammar is defined by a unique derivation for a single string, in other words, any CSG transform can be expressed by a CFG. Thus, the size of minimum CSG for a string is closely related to that of CFG. So, we mention briefly the theoretical results on the smallest grammar-based compression problem, which is known to be NP-hard. Three $O(\log n)$-approximation algorithms [4], [15], [16] were proposed for this problem, and the space efficiency was improved in [17] within almost log-scale approximation ratio. Particularly, the technique of LZ-factorization in [15] is useful for grammar size analysis in this study.

On the other hand, a large number of practical algorithms have been proposed. We specially refer to Re-Pair [9] since our compression algorithm is also based on the *recursive pairing*. In order to develop an $O(n)$-time/space algorithm, the input string is represented by a linked-list so that any occurrence of a digram is connected to its successor and predecessor of the same digram. The Byte-Pair-Encoding (BPE) is considered as simple implementation of Re-Pair with grammar symbols at most 256.

Such effective compression algorithms are closely related to fast *compressed pattern matching* (CPM). Amir et al. [2] introduced an algorithm of finding the first occurrence of a pattern on LZW compression in $O(n + m^2)$ time, where n and m are the lengths of text and pattern, respectively. Navarro and Raffinot [13] developed more general technique, which abstracts both LZ77 and LZ78 and runs in $O(nm/w + m + occ)$, where w is the machine word length and occ is the number of pattern occurrences. Kida et al. [7] proposed the *collage systems*: a formal system to represent a string by dictionary $\mathcal{D}$ and sequence $\mathcal{S}$ of variables, which unifies various dictionary-based compressions such as LZ family (LZ77, LZSS, LZ78, LZW), CFG transform based compressions, Run-Length encoding, etc. They also presented a general CPM algorithm on collage systems which runs in $O(h \cdot (d + s) + m^2 + occ)$ time, where d, h are the size and the maximum dependence of $\mathcal{D}$, respectively, and s is the length of $\mathcal{S}$, and the factor h disappears for the class of truncation-free collage systems that subsumes the CFG transform.

For practical speed-up of CPM, compressions with byte code are attractive since we can avoid any bitwise processing. BPE limits the number of the grammar symbols by 256 in order to represent in one byte each of them, and allows a fast search [18], [19]. The compression ratio is, however, very poor. Matsumoto et al. [12] recently proposed to represent a large number of grammar symbols by byte-oriented Huffman code and improved both the compression and the search performances. However, the space requirement for the finite-state machine used grows linearly proportional to the number of grammar symbols.

Along this line of researches, our study is motivated

Manuscript received March 23, 2009.
Manuscript revised June 30, 2009.

[†]The authors are with the Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka-shi, 819–0395 Japan.

[††]The authors are with the Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology, Fukuoka-shi, 820–8502 Japan.

a) E-mail: shiro.maruyama@i.kyushu-u.ac.jp
b) E-mail: t_youhei@donald.ai.kyutech.ac.jp
c) E-mail: hiroshi@ai.kyutech.ac.jp
d) E-mail: takeda@i.kyushu-u.ac.jp

DOI: 10.1587/transinf.E93.D.219

by improving the present CPM performance in both theoretical and practical sense. Let us express our strategy for grammar-based compression by an intuitive example. If a text contains many occurrences of a digram AB, we can replace all of them by a single variable X which is associated with AB like $X \to AB$. The text is thus compressed to a shorter one according to the frequency of AB. However the variables are incompatible among different digrams, i.e., we must produce k different variables for each k different digrams. Since this restriction is not avoidable in CFG transform, we relax the grammar class to context-sensitive grammars (CSGs) and introduce the *CSG transform*.

The introduced grammars CSGs are the Σ-*sensitive grammars* such that each of the production rules is of the form $aA \to a\gamma$ or $A \to \gamma$, where a is a symbol in the alphabet, A is a variable and γ is a non-empty string of symbols and variables. The production rules of the former form is also called Σ-sensitive. This grammar transform is related to the *context-dependent grammar* (CDG) by [20]. Indeed, a subclass of the Σ-sensitive grammars produced by our compression algorithm is included in the CDG transform.

Our contribution in this paper is as follows. We first analyze the expressiveness of Σ-sensitive grammars compared with CFGs by proving the upper/lower bound of grammar size. We next give a compression algorithm by recursive pairing. In our method, digram AB is replaced by a variable X for every occurrence of trigram aAB with yielding a new production rule $aX \to aAB$. This strategy is potentially better than the standard recursive pairing since different digrams like $AA, AB, \ldots$ can be replaced by a same variable if they are appearing in different contexts. While the compression model is a special case of the Σ-sensitive grammars, we can show that, even when the number of grammar symbols is bounded by 256, the compression performance is a match for other practical methods such as Gzip and Re-Pair. Finally we develop a CPM algorithm on the compression model and show that it runs almost twice faster than the KMP type CPM algorithm on BPE by Shibata et al. [18], and that in the case of short patterns, it runs faster than the CPM technique of Rautio et al. [14], which is based on a variant of the Boyer-Moore-Horspool algorithm and the stopper encoding (SE), regarded as one of the best combinations of compression scheme and pattern matching technique that allow a fast search in practice.

2. Preliminaries

We assume a finite set Σ of alphabet symbols. The set of all strings over Σ is denoted by Σ^*, and $\Sigma^+ = \Sigma^* - \{\varepsilon\}$ for the empty string ε. The expression Σ^i denotes the set of all strings of length i. The length of a string $w \in \Sigma^*$ is denoted by $|w|$, and also for a set $\mathcal{S}$, the notation $|\mathcal{S}|$ refers to the size of $\mathcal{S}$.

We recall the definition of context-free grammars (CFGs) and context-sensitive grammars (CSGs). A CFG is defined by $G = (V, \Sigma, P, S)$ with disjoint finite sets Σ and V, a finite set $P \subseteq V \times (V \cup \Sigma)^*$ of *production rules*, and the *start symbol* $S \in V$. Symbols in V are called *variables*. Here we define the derivation of G. For string $\alpha, \beta \in (V \cup \Sigma)^*$, $\alpha A \beta$ *derives* $\alpha\gamma\beta$ if $(A \to \gamma) \in P$. This relation is expressed by $\alpha A \beta \Rightarrow \alpha\gamma\beta$. The reflexive, transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$, and $S \overset{*}{\Rightarrow} \alpha$ is called a *derivation* of α from the start symbol.

A CSG is defined by $G = (V, \Sigma, P, S)$ such that any production rule is of $\alpha A \beta \to \alpha\gamma\beta$ for $A \in V$ and $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$. In case of CSG, we can replace an occurrence of variable $A \in V$ with a string γ by a production rule $(\alpha A \beta \to \alpha\gamma\beta) \in P$ only if A appears together with the contexts α, β. The derivation of CSG is thus similarly defined, and we also use the notation $\overset{*}{\Rightarrow}$ for CSGs.

The *size* of a grammar, $|G|$, is the total length of all production rules. Without loss of generality, we can regard the size of a CFG G as $|V|$ in G, since there is an equivalent CFG G' in Chomsky normal form such that $|G'| \leq 2|G|$.

3. CSG Transform and Grammar Size Analysis

In this section, we give the definition of a very restricted class of CSGs and show that the class is powerful enough to handle the *CSG transform* defined as follows. We assume that any CFG G is restricted to be an *admissible grammar*: G derives exactly one string $w \in \Sigma^+$. This notion leads us to the CFG transform: G is an encoding of w and $S \overset{*}{\Rightarrow} w$ is the decoding to w. The notion of *CSG transform* is directly obtained by the same condition, i.e. any CSG derives exactly one string. We then assume that any CSG is also an admissible grammar.

3.1 Σ-Sensitive Grammars

For CSG transform, we introduce restricted CSGs each of which production rules is either of the forms:

$$aA \to a\gamma \quad \text{and} \quad A \to \gamma,$$

where $a \in \Sigma$, $A \in V$, and $\gamma \in (V \cup \Sigma)^+$. Such a grammar is said to be Σ-*sensitive*. The preceding symbol a of an occurrence of A is called its *context*, and the definition of derivation follows the case of general CSG. Moreover, in order to obtain the uniqueness of derivation, we assume the *leftmost derivation*, i.e. leftmost variable must be replaced by possible production rule at all times.

The grammars are naturally extended to the Σ^n-sensitive grammars, where a production rule $\alpha A \to \alpha\gamma$ for $\alpha \in \Sigma^n$ is allowed. Such a production $\alpha A \to \alpha\gamma$ can be reduced to a short expression $xA \to x\gamma$, where x is a variable in a CFG which derives α.

Next we mention a normal form of Σ-sensitive grammar. Any production rule $aA \to a\gamma$ with $\gamma = A_1 \cdots A_k$ can be simulated by $aA \to aA_1B_1$, $B_i \to A_{i+1}B_{i+1}$ and $B_{k-2} \to A_{k-1}A_k$. Thus, without loss of generality, we can assume that the length of right hand of any production rule is bounded by two.

3.2 Grammar Size Analysis

We compare the size of CFG with that of Σ-sensitive grammar over k-letter alphabet as well as Σ^n-sensitive grammar over three-letter alphabet, respectively. We begin with the following upper bound.

Theorem 1: Let G_s be a minimum Σ-sensitive grammar for a string and G_f be a minimum CFG for same string. If $|\Sigma| = k$, then $\dfrac{|G_f|}{|G_s|} = O(k)$.

proof. We show that a CFG satisfying the bound can be constructed from G_s. By the assumption, G_s has a unique leftmost derivation of w. Let T_s be a corresponding derivation tree. If all variables appearing in T_s are different, this bound is trivial. Assume that T_s contains two occurrences of a variable A, and let T_1, T_2 be the corresponding subtrees rooted by the two As. Let $a \in \Sigma$ be the context of the root A of T_1, that is, the preceding leaf of the leftmost leaf of T_1, and let $b \in \Sigma$ be the context of the root A of T_2. Then, T_1 is identical to T_2 iff $a = b$, since the derivations from the two As are decided by their contexts a, b only. Thus, there are at most k different subtrees rooted by A. Replacing all such roots by a CFG variable A_a, a derivation tree T of a CFG G is produced. The number of different variables in T is denoted by $|T|$. Then, $|T_f| \le |T| \le k|T_s| \le k|G_s|$ by $|T_s| \le |G_s|$. Therefore, we obtain the bound $|G|/|G_s| = O(k)$ by $|G| = O(|T|) = O(|V|)$. *Q.E.D.*

On the other hand, the upper bound does not tell us the existence of such a *difficult* string to compress. Thus, by proving two theorems, we would show that CSG transform is indeed powerful compared with CFG transform. The first one is the comparison between Σ-sensitive grammars and CFGs, and the second is an extension for Σ^n-sensitive grammars allowed to take long contexts.

Theorem 2: Let G_s be a minimum Σ-sensitive grammar for a string and G_f be a minimum CFG for same string. For $|\Sigma| = k$ and any sufficiently large n, there exists $w \in \Sigma^n$ satisfying $\dfrac{|G_f|}{|G_s|} \ge c\left(\dfrac{k \log \frac{n}{k}}{k + \log \frac{n}{k}}\right)$, where c is a constant independent of w.

proof. For $m \ge 1$ and $\Sigma = \{a_i, b_j \mid 0 \le i \le k, 0 \le j < k\}$, let us consider the following string.

$$
\begin{aligned}
w &= \left(\prod_{i=0}^{k-1} a_i^m b_i b_i\right) a_k a_k \\
&= (\overbrace{a_0 \cdots a_0}^{m} b_0 b_0)(\overbrace{a_1 \cdots a_1}^{m} b_1 b_1) \cdots \\
&\quad (\overbrace{a_{k-1} \cdots a_{k-1}}^{m} b_{k-1} b_{k-1}) a_k a_k
\end{aligned}
$$

For this w we can construct the Σ-sensitive grammar defined by

$$
3k \text{ productions} \begin{cases} a_i A & \to\ a_i a_i \\ a_i B & \to\ a_i b_i b_i \\ b_i A & \to\ b_i a_{i+1} \end{cases} \quad (0 \le i < k)
$$

and $(\lceil \log m \rceil + \lceil \log k \rceil + 1)$ production rules for the derivations $X \overset{*}{\Rightarrow} A^m B$ and $S \overset{*}{\Rightarrow} a_0 X^k A$. On the other hand, it is clear that G_f must contain at least $k\lceil \log m \rceil$ production rules for deriving w. Thus, we obtain the bound by the relation $n = km + 2k + 2$. *Q.E.D.*

Here we specify that the bound in Theorem 2 is smaller than k, which is the upper bound in Theorem 1. Next we show another bound in case that the length of contexts are unlimited. We recall the rich class of CSGs, called Σ^n-sensitive grammars, i.e. a context is possibly equal to the string itself. Note that this case is incomparable to the upper bound of $|G_f|/|G_s|$ since the length of context in G_s is just one.

For this purpose, we begin with the notion of *LZ-factorization*. The factors $f_1, f_2, \ldots, f_k$ is called the LZ-factorization of a string w if $w = f_1 f_2 \cdots f_k$, $f_1 = w[1]$, and f_i is the longest prefix of $f_i \cdots f_k$ appearing in $f_1 \cdots f_{i-1}$. By $\#LZ(w)$, we denote the number of the factors. For example, if $w = ababaaba$, the LZ-factorization is a, b, ab, a, aba, and $\#LZ(w) = 5$.

Theorem 3: (Rytter [15]) For any string w and its admissible CFG G, it holds that $\#LZ(w) \le |G|$.

Here we show a bound of the ratio CFG/CSG on a constant alphabet. For this proof, we use the infinitely long *square-free string* over a three-letter alphabet.

A string is said to be *square-free* if it contains no squares like α^2. For example, *abcacb* is square-free but *ababc* is not square-free. It is known (see e.g. [10]) that for a three-letter alphabet $\Sigma = \{a, b, c\}$, there exists the infinite square-free string

$$abcbacbcabcbabcacbacabcacbcabacbabcabac \cdots .$$

Using the infinitely many prefixes of the string, we prove the following bound of the ratio CFG/CSG on the three-letter alphabet.

Theorem 4: Let G_s^n be a minimum Σ^n-sensitive grammar for a string and G_f be a minimum CFG for same string. For a 3-letter alphabet $\Sigma = \{a, b, c\}$ and any sufficiently large n, there exists $w \in \Sigma^n$ satisfying $\dfrac{|G_f|}{|G_s^n|} \ge c\left(\dfrac{\sqrt[3]{n} \log n}{\sqrt[3]{n} + \log n}\right)$, where c is a constant independent of w.

proof. Let p_i be the i-th prefix of the infinite square-free string, that is, $p_0 = a, p_1 = ab, p_2 = abc, p_3 = abcb, \ldots$ For any even number m, we define the following string w.

$$
\begin{aligned}
w &= \left(\prod_{i=0}^{\frac{m}{2}} p_{2i}^m p_{2i+1}^2\right) p_{m+2} p_{m+2} \\
&= (\overbrace{p_0 \cdots p_0}^{m} p_1 p_1)(\overbrace{p_2 \cdots p_2}^{m} p_3 p_3) \cdots \\
&\quad (\overbrace{p_m \cdots p_m}^{m} p_{m+1} p_{m+1}) p_{m+2} p_{m+2}
\end{aligned}
$$

We first analyze the size of G_f. Consider the LZ-factorization of w. If p_i^m contains a period[†] shorter than i, it must be of $(\alpha\beta)^k\alpha$ for some $k \geq 2$, which is not square-free. Thus, p_i^k is not appearing in p_j^m for any $i < j$ and $k \geq 2$. Hence, the LZ-factorization for p_i^m contains $\Omega(\log m)$ factors, that is, $\#LZ(w) = \Omega(m \log m) \leq |G_f|$.

On the other hand, we can construct a CFG deriving all strings $p_1, p_2 \ldots, p_{m+2}$ by m production rules defined by variables $P_1, P_2 \ldots, P_{m+2}$. Then, this grammar encodes the string to the following string.

$$P(w) = (\overbrace{p_0 \cdots p_0}^{m} P_1 P_1)(\overbrace{P_2 \cdots P_2}^{m} P_3 P_3) \cdots (\overbrace{P_m \cdots P_m}^{m} P_{m+1} P_{m+1}) P_{m+2} P_{m+2}$$

By the similar technique in Theorem 2, we can construct a Σ^n-sensitive grammar which derives $P(w)$ within $O(\log m^2) = O(\log m)$ production rules. Thus, $|G_s^n| = O(m + \log m)$.

Therefore, since $|p_m| = m$ and $|w| = n = \Theta(m^3)$, we can obtain the bound for the string.

$$\frac{|G_f|}{|G_s^n|} \geq c\left(\frac{m \log m}{m + \log m}\right) \geq c\left(\frac{\sqrt[3]{n} \log n}{\sqrt[3]{n} + \log n}\right)$$

Q.E.D.

From the analysis in this section, we can conclude that our CSG transform is powerful compared with the standard CFG transform.

4. Greedy Compression Algorithm

Re-Pair [9] is one greedy CFG transform algorithm based on the most-frequent-first strategy. It replaces every occurrence of a most frequent digram AB in the input string by a new variable symbol X and generates the production rule $X \to AB$. This process is repeated until no digram appear more than once.

We extend this algorithm to the CSG transform. Let $\Sigma = \{a_1, \ldots, a_k\}$ with $|\Sigma| = k$. The key idea is to select a digram A_iB_i that occurs most frequently just after a_i for every $i = 1, \ldots, k$, and generate the following production rules

$$a_1X \to a_1A_1B_1, \quad \ldots, \quad a_kX \to a_kA_kB_k,$$

where X is a new variable symbol and A_i, B_i are either symbols in Σ or variable symbols introduced so far. Every occurrence of A_iB_i preceded by a context a_i in the input string is replaced by one symbol X independently of i. We remark that rewriting the input string yields occurrences of digrams preceded by a variable symbol (not a symbol in Σ), but their contexts remain unchanged and can be kept.

The extended Re-Pair algorithm is presented in Algorithm 1, and the outline of this algorithm is illustrated in Fig. 1. As is shown in this example, although a straightforward extension of the algorithm of [9] requires $\Omega(|\Sigma||w|)$ time and space, we can reduce the time/space complexity to $O(|w|)$. The key technique is a data structure which returns in constant time one of the most frequent digram for each context. Let $c \in \Sigma$, and let $L_c(f)$ be the list of active digrams with context c having frequency f. We use a specialized priority queue which stores $\mathcal{L}_c = L_c(f_1), \ldots, L_c(f_k)$, where $f_1, \ldots, f_k$ are the positive integers (priorities) in the increasing order such that $L_c(f_i)$ is not empty for all $i = 1, \ldots, k$. We maintain the priority queues $\mathcal{L}_c$ for all contexts c. An update of the priority queues takes constant time per a replacement operation. The total time and space complexity is thus $O(|w|)$.

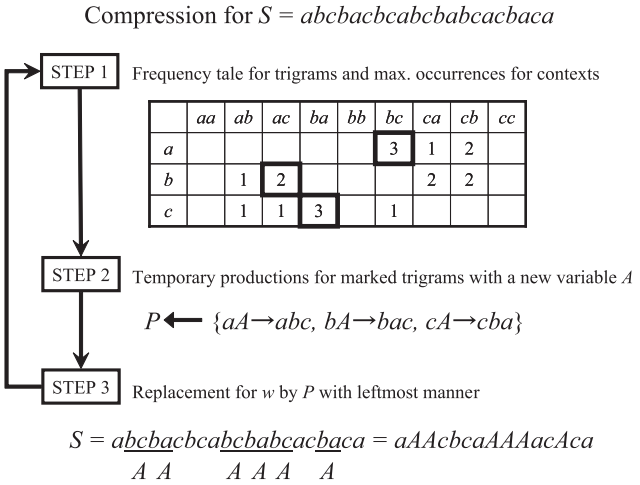


Fig. 1 An example run of the extended Re-Pair.

5. Compressed Pattern Matching

One goal of the CPM problem (Goal 1) is a faster search in a compressed text, compared with decompression followed by an ordinary search [1]. A more ambitious goal (Goal 2) is a faster search in a compressed text in comparison with

Algorithm 1 Extended Re-Pair

```
input a string S.
repeat
    scanning S from left to right, get frequency of digrams with a context a ∈ Σ.
    Introduce a new variable X.
    for each a ∈ Σ do
        get the lexicographically first AB (A, B ∈ (Σ∪V)) that is the most frequent digram with context a,
        generate aX → aAB, and update P.
    end for
    scanning S, replace the leftmost AB with context a by X for some aX → aAB ∈ P, and
    update S to the replaced string.
until no digram to replace appear in S
output (S, P).
```

[†]A positive integer p is called a *period* of a string x if $x[i] = x[i + p]$ for $i = 0, 1, \ldots, |x| - p - 1$, where $x[i]$ is the i-th symbol of x.

an ordinary search in the original text [11]. The aim of compression in the context of Goal 2 is not only to reduce disk storage requirement or data transmission cost but also to speed up string searching. In this section, we consider the CPM problem for restricted Σ-sensitive grammars and show a CPM algorithm based on [7]. We then discuss a Goal 2-oriented implementation of it.

Definition 1: CompPatMatch
Input: *A pattern* $\pi \in \Sigma^+$ *and a* Σ*-sensitive grammar* $G = (V, \Sigma, P, S)$ *generating a string* $w \in \Sigma^+$ *such that every production rule in* P *takes either of the forms:* $aA \to a\gamma$ *and* $A \to \gamma$ *(*$a \in \Sigma, A \in V, \gamma \in (V \cup \Sigma)^+$*).*
Output: *All occurrences of* π *within* w*.*

The claim for CompPatMatch is to find all occurrences of π without expansion of G. Let us consider the production rules

$$S \to bBBaA,\ aA \to aBB,\ aB \to abb,\ bB \to bab,$$

and pattern $P = ababb$. Since the original string is *bababababb*, the output for this input is the occurrence position 4. Such an answer can be reported by a pattern automaton over (π, w). Thus, our goal is to construct a compact data structure for simulating such automata over (π, G). Fortunately, the techniques of CPM is established for CFG, our strategy is focused on convert a Σ-sensitive grammar to an equivalent CFG preserving the compactness of automata.

The production rule with lhs S is called the *start rule*, and the set of other rules in P is denoted by $P^{\#}$. Let $S \to b\mu$ be the start rule of G with $b \in \Sigma$ and $\mu \in (V \cup \Sigma)^*$. Similar to the collage system, we regard $P^{\#}$ as a dictionary and $b\mu$ as a variable sequence although the rules of $P^{\#}$ are not context-free. Denote by $\|P^{\#}\|$ the total length of the rhs's of rules in $P^{\#}$. Let $V^{\#} = V - \{S\}$.

For $a \in \Sigma$ and $X \in V$, let $\xi(a, X)$ denote the string u in Σ^+ such that $aX \stackrel{*}{\Rightarrow} au$. If no such a string u exists, $\xi(a, X)$ is undefined. For $X = c \in \Sigma$, let $\xi(a, X) = c$. Let $\lambda(a, X)$ be the rightmost symbol of $\xi(a, X)$.

Lemma 1: The function $\lambda : \Sigma \times (V^{\#} \cup \Sigma) \to \Sigma$ can be constructed in $O(\|P^{\#}\|)$ time so that it responds in constant time.

5.1 Application of Algorithm by Kida et al.

The input Σ-sensitive grammar $G = (V, \Sigma, P, S)$ is equivalent to the CFG $G' = (V_f \cup V_s \cup \{S\}, \Sigma, P_f \cup P_s \cup \{S \to \psi(b, \mu)\}, S)$, where

$$V_f = \{A \mid A \in V^{\#},\ A \to \gamma \in P\},$$
$$V_s = \{A_a \mid A \in V,\ a \in \Sigma,\ aA \to a\gamma \in P\},$$
$$P_f = \{A \to \gamma \mid A \in V_f,\ A \to \gamma \in P\},$$
$$P_s = \{A_a \to \psi(a, \gamma) \mid A_a \in V_s,\ aA \to a\gamma \in P\},$$

where $\psi(a, \gamma)$ denotes the string over $(V_f \cup V_s \cup \Sigma)$ obtained from γ by replacing every occurrence of $A \in V - V_f$ such that $\gamma = \alpha A \beta$ and $\alpha, \beta \in (V \cup \Sigma)^*$ by $A_c \in V_s$ such that $c = \lambda(a, Y)$ where Y is the rightmost symbol of $a\alpha$. Conversion of G into G' takes $O(\|P\|)$ time by using the function λ.

A naive solution to CompPatMatch would be to convert G into G' and apply the algorithm of Kida et al. [7]. The algorithm first preprocesses π and the rules in $P_f \cup P_s$ to build a finite-state machine M and then makes M run over the symbols of $\psi(b, \mu)$. The machine M consists of state-transition and output functions defined on the domain $Q \times (V_f \cup V_s \cup \Sigma)$, where Q is the set of states of the KMP automaton for π. It can be implemented in $O(|\pi|^2 + \|P_f \cup P_s\|) = O(|\pi|^2 + \|P^{\#}\|)$ time and space and runs in $O(|\mu| + occ)$ time over $\psi(b, \mu)$.

Theorem 5: CompPatMatch can be solved in $O(|\pi|^2 + \|P\| + occ)$ time using $O(|\pi|^2 + \|P^{\#}\|)$ space.

5.2 Practically-Fast Implementation

For practical speed-up, we want to implement the state-transition function as a two-dimensional array of size $|Q| \times |V_f \cup V_s \cup \Sigma|$ as in [18]. However, this is unrealistic for a large V since $|V_f \cup V_s|$ can be $|V^{\#}||\Sigma|$. In what follows, we describe how to reduce the domain size.

Consider the set $Q = \{0, 1, \ldots, |\pi|\}$ of states of the KMP automaton for a pattern π, where j corresponds to the j-length prefix of π. The idea is to modify the KMP automaton by adding $|\Sigma|$ distinct states so that it memorizes the symbol read previously. Let $Q_\Sigma = \{q_a \mid a \in \Sigma\}$ be the set of these states. The state-transition function $\delta' : (Q \cup Q_\Sigma) \times \Sigma \to Q \cup Q_\Sigma$ of the modified KMP automaton is defined as follows.

$$\delta'(q, a) = \begin{cases} \delta(0, a), & \text{if } q = q_c \in Q_\Sigma \text{ for some } c; \\ q_a, & \text{if } q \in Q \wedge \delta(q, a) = 0; \\ \delta(q, a), & \text{otherwise,} \end{cases}$$

where δ is the state-transition function of the original KMP automaton. The function δ is extended for strings by $\delta(p, aw) = \delta(q, w)$ if $\delta(p, a) = q$, $a \in \Sigma$, and $w \in \Sigma^+$, and so is the function δ'.

The function δ' is computed from the modified version of the goto and the failure functions. The modified goto function differs from the original one in that the arrows from the auxiliary state $\perp$ to q_a and the arrows from q_a to state 1 are added for all $a \in \Sigma$. The inductive computation of the modified failure function is performed in exactly the same way as the original one.

An example of the modified KMP automaton is shown in Fig. 2, together with the original one.

Based on the modified KMP automaton, we define functions *Jump* and *Output* on the domain $(Q \cup Q_\Sigma) \times (V^{\#} \cup \Sigma)$ by

$$Jump(q, X) = \delta'(q, \xi(a, X)),$$
$$Output(q, X) = \left\{ |\xi(a, X)| - |w| \;\middle|\; \begin{array}{l} w \text{ is a non-empty prefix of } \xi(a, X) \\ \text{such that } \delta'(q, w) \text{ is the final state.} \end{array} \right\}$$

where $q \in (Q - \{0\}) \cup Q_\Sigma$, $X \in V^{\#} \cup \Sigma$, and $a \in \Sigma$ is the

context memorized by state q. For $q = 0$, $Jump(q, X)$ and $Output(q, X)$ are defined only for $X \in \Sigma$.

Figure 3 shows the functions *Jump* and *Output* built from the modified KMP automaton of Fig. 2 for the production rules $aA \to aBB, aB \to abb, bB \to bab$. Figure 4 shows the move of the machine over the rhs of $S \to bBBaA$.

We note that $|V_f \cup V_s|$ can be $|V^{\#}||\Sigma|$ for a large V. This means $|Q \times (V_f \cup V_s \cup \Sigma)| \geq |V^{\#}||Q||\Sigma|$. On the other hand, $|(Q \cup Q_\Sigma) \times V^{\#}| \leq |V^{\#}|(|Q|+|\Sigma|)$. Thus, the domain $(Q \cup Q_\Sigma) \times V^{\#}$ can be much smaller than the domain $Q \times (V_f \cup V_s \cup \Sigma)$. This is a big advantage in the sense that we can adopt the standard two-dimensional array implementation of *Jump*.

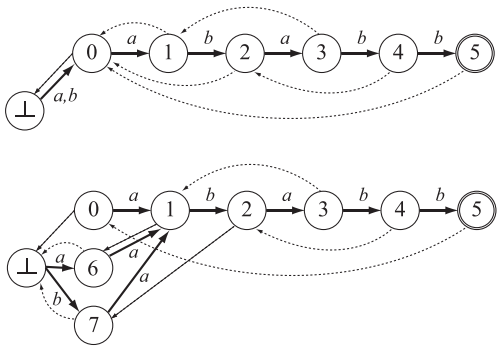


Fig. 2 A standard KMP automaton for $\pi = ababb$ is displayed on the upper, and the modified one is displayed on the lower, where $\Sigma = \{a, b\}$ and the solid and the broken arrows represent the goto and the failure functions. We note that the values of the failure function for states 1 and 2 differ between the two automata.

$Jump(q, X)$	a	b	A	B
0	1	7	—	—
1 (a)	1	2	2	7
2 (b)	3	7	—	4
3 (a)	1	4	2	5
4 (b)	3	5	—	4
5 (b)	1	7	—	2
6 (a)	1	7	2	7
7 (b)	1	7	—	2
$Output(q, X)$	a	b	A	B
0	$\emptyset$	$\emptyset$	—	—
1 (a)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2 (b)	$\emptyset$	$\emptyset$	—	$\emptyset$
3 (a)	$\emptyset$	$\emptyset$	$\{2\}$	$\{0\}$
4 (b)	$\emptyset$	$\{0\}$	—	$\emptyset$
5 (b)	$\emptyset$	$\emptyset$	—	$\emptyset$
6 (a)	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
7 (b)	$\emptyset$	$\emptyset$	—	$\emptyset$

Fig. 3 *Jump* and *Output* functions built from the modified KMP automaton of Fig. 2 for production rules $aA \to aBB, aB \to abb, bB \to bab$. Each parenthesized symbol following state s means the symbol read immediately before reaching s. Variable A represents $bbab$ in context a, while it represents nothing in context b. Also variable B means bb in context a, while it means ab in context b.

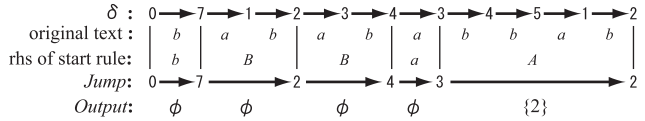


Fig. 4 Move of the machine of Fig. 3 over the rhs of $S \to bBBaA$.

Theorem 6: We can build in $O(|\pi||V| + ||P^{\#}||)$ time a two-dimensional table storing the values of *Jump* and a data structure for *Output* which responds in time linear in the answer size.

If $|V^{\#} \cup \Sigma| \leq 256$, we can encode symbols in $V^{\#} \cup \Sigma$ in one byte. Compared to the CPM algorithm on BPE presented in [18], the number of production rules can be $|\Sigma|$ times larger whereas the table size of *Jump* is larger just by $256 \times |\Sigma|$ table entries. Thus, both the compression and the search performances are expected to be improved drastically.

6. Computational Experiments

We implemented in C language the compression algorithm presented in Sect. 4 where the grammar symbols are bounded by 256 and encoded in one byte, and the CPM algorithm presented in Sect. 5.2. We evaluated their performances by a series of computational experiments. All the experiments were carried out on a SUN Ultra 20 M2 Workstation with a 2.2 GHz Dual Core AMD Opteron 1214 and 2.0 GB RAM running Solaris 10. The text files we used are as follows.

Medline. A clinically-oriented subset of Medline, consisting of 348,566 references. The file size is 60.3 Mbytes. $|\Sigma| = 87$.

Genbank. The file consisting of accession numbers and nucleotide sequences taken from a data set in Genbank. The file size is 17.1 Mbytes. $|\Sigma| = 59$.

DBLP. A set of DBLP XML records. The file size is 130.7 Mbytes. $|\Sigma| = 96$.

Sources. The concatenation of all the .c, .h, .C, .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions. The file size is 52.4 Mbytes. $|\Sigma| = 227$.

Pitches. A sequence of pitch values obtained from a myriad of MIDI files freely available on Internet. The file size is 52.4 Mbytes. $|\Sigma| = 133$.

Table 1 compares the compression ratios of our method

Table 1 Compression ratio comparison (%).

	standard compressors		
	gzip	bzip2	Re-Pair
Medline	33.29	23.57	33.83
Genbank	21.98	22.17	31.32
DBLP	17.48	11.66	17.67
Sources	23.29	19.79	31.07
Pitches	30.27	35.73	58.23
	Goal 2-oriented compressors		
	SE [14]	BPE	ours
Medline	66.50	56.41	32.94
Genbank	51.74	31.37	28.22
DBLP	70.05	40.83	20.24
Sources	71.93	80.54	55.56
Pitches	74.77	78.34	63.36

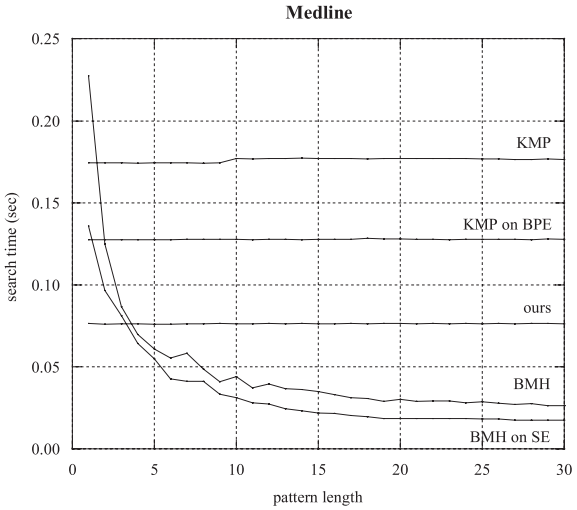


Fig. 5 Search-time comparison for MEDLINE.

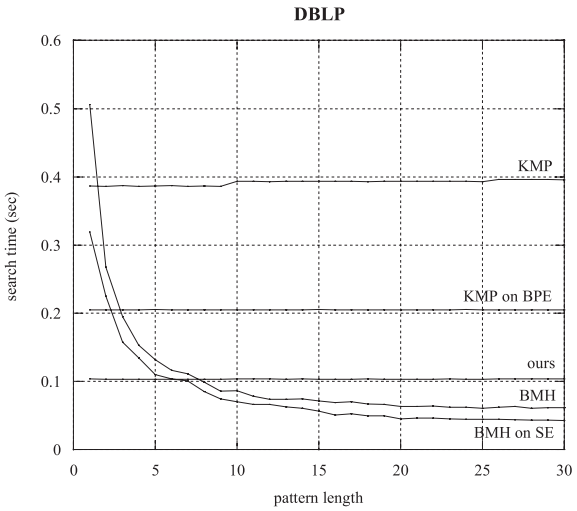


Fig. 6 Search-time comparison for DBLP.

and other compressors, where SE denotes the stopper encoding with 4-bit base symbols. Despite using byte codes, the compression ratio of our method is competitive to or slightly worse than the standard compressors for Medline, Genbank and DBLP. It is also much better than the other Goal 2-oriented compressors[†]. On the other hand, the performance of our method is poor for Sources and Pitches. For Pitches, the performance of Re-Pair is also poor. Although Pitches is a mixture of pitch data with different nature, our method as well as Re-Pair depends on the substring statistics over the whole data and therefore shows poor performance. In fact, the performance of our method was improved by partitioning the file into fragments and then compressing them separately. The poor performance for Sources is mainly due to the large alphabet size ($|\Sigma| = 227$). We note that the number of production rules generated is upper-bounded by $|\Sigma|(256 - |\Sigma|)$, and the bounds for Medline, Genbank, DBLP, Sources and Pitches are, respectively, 14703, 11623, 15360, 6583 and 16359. Thus our compression scheme is suited when $|\Sigma|$ is closed to 128.

We compared the search time of our method with the KMP algorithm (KMP) and the BMH algorithm (BMH) over uncompressed text as well as existing Goal 2-oriented CPM methods: the KMP algorithm over BPE compressed text (KMP on BPE) [19] and the BMH algorithm over SE compressed text (BMH on SE) [14]. Figures 5 and 6. displays the search times (including the preprocessing times) for Medline and DBLP. Our method runs faster than BMH on SE for short patterns.

7. Conclusion

We proposed a special case of CSGs called Σ-sensitive grammars for effective grammar transform and fast compressed pattern matching. While the Σ-sensitiveness is strong restriction, we show that this grammars is powerful enough to represent a compact formal model. Using a small subclass of this class, we obtained a sufficient compression ratio competitive with other practical models. Moreover we implemented the CPM algorithm on the compressed texts and confirmed its performance. In particular, compared to the BMH algorithm and the stopper encoding, regarded as one of the best combinations that allows a practically fast search, our method achieves much better compression and a faster search for short patterns.

In practice our method as well as BPE assumes that a text file is a sequence of characters in Σ which are encoded in 8-bit codewords and compresses it by exploiting the unused codewords for encoding variables. Thus at most 256 different characters/variables are allowed. However, oriental languages such as Japanese, Korean, and Chinese have much more characters and multi-byte character code is used to represent them. One solution would be to compress such a text by simply regarding it as a sequence of bytes. Since the ASCII characters are still expressed with a single byte for compatibility, there are single byte characters and multi-byte characters in one text file. For example, a text in Japanese Extended-Unix-Code (EUC) is a mixture of single byte characters and two byte characters. Therefore a straightforward application of the CPM technique presented in Sect. 5 causes false-detection of pattern. Fortunately, this problem can be resolved by using the synchronization technique [3], [6].

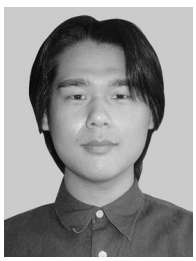
In [3], the technique of dividing characters into halves is presented for a space-economical implementation of the Aho-Corasick (AC) automata: We build AC automaton by regarding each pattern as a sequence of half-bytes, and then make it run over the half-bytes sequence of a text file. Again the synchronization technique is used to avoid the false-detection of patterns. The number of states is at most twice larger than the ordinary AC automaton since we need intermediate states. But the size of two-dimensional array of the state-transition function is reduced to 1/8 since the alphabet size becomes 1/16. The technique, however, is not

[†]Compression methods for achieving faster search than uncompressed pattern matching algorithms.

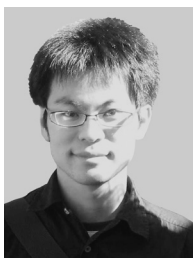
applicable to our method. If we regard a text file as a sequence of half-bytes, then most of the 16 half-bytes are already used up, and hence a BPE-type compression method does not work well. Suppose we regard the text file as a sequence of bytes and compress it by using a BPE-type compression method. The rhs of the start rule is a sequence of bytes representing either characters or variables. We have to create intermediate states for the first half-bytes of not only characters but also variables. Thus the number of states can grow more than twice as large as that of the ordinary AC automaton.

References

[1] A. Amir and G. Benson, "Efficient two-dimensional compressed matching," Proc. Data Compression Conference'92 (DCC'92), p.279, 1992.
[2] A. Amir, G. Benson, and M. Farach, "Let sleeping files lie: Pattern matching in Z-compressed files," J. Comput. Syst. Sci., vol.52, no.2, pp.299–307, 1996.
[3] S. Arikawa and T. Shinohara, "A run-time efficient realization of Aho-Corasick pattern matching machines," New Generation Computing, vol.2, no.2, pp.171–186, 1984.
[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] M. Crochemore, C. Hancart, and T. Lecroq, Algorithms on Strings, Cambridge University Press, 2007.
[6] S. Arikawa et al., "The text database management system SIGMA: An improvement of the main engine," Proc. Berliner Informatik-Tage, pp.72–81, 1989.
[7] T. Kida, T. Matsumoto, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa, "Collage systems: A unifying framework for compressed pattern matching," Theor. Comput. Sci., vol.298, no.1, pp.253–272, 2003.
[8] 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.
[9] N.J. Larsson and A. Moffat, "Off-line dictionary-based compression," Proc. IEEE, vol.88, no.11, pp.1722–1732, 2000.
[10] M. Lothaire, Combinatorics on Words, Cambridge University Press, 1983.
[11] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file," ACM Trans. Inf. Syst., vol.15, no.2, pp.124–136, 1997.
[12] T. Matsumoto, K. Hagio, and M. Takeda, "A run-time efficient implementation of compressed pattern matching automata," Proc. 13th International Conference on Implementation and Application of Automata (CIAA'08), pp.201–211, 2008.
[13] G. Navarro and M. Raffinot, "Practical and flexible pattern matching over Ziv-Lempel compressed text," J. Discrete Algorithms, vol.2, no.3, pp.347–371, 2004.
[14] J. Rautio, J. Tanninen, and J. Tarhio, "String matching with stopper encoding and code splitting," Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM'02), vol.2373 of LNCS, pp.42–52, 2002.
[15] W. Rytter, "Application of lempel-ziv factorization to the approximation of grammar-based compression," Theor. Comput. Sci., vol.302, no.1-3, pp.211–222, 2003.
[16] H. Sakamoto, "A fully linear-time approximation algorithm for grammar-based compression," J. Discrete Algorithms, vol.3, no.2-4, pp.416–430, 2005.
[17] 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.
[18] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, "Speeding up pattern matching by text compression," Proc. 4th Italian Conference on Algorithms and Complexity (CIAC'00), vol.1767 of LNCS, pp.306–315, 2000.
[19] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa, "A Boyer-Moore type algorithm for compressed pattern matching," Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'00), vol.1848 of LNCS, pp.181–194, 2000.
[20] E.-H. Yang and D.-K. He, "Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform - part two: With context models," IEEE Trans. Inf. Theory, vol.49, no.11, pp.2874–2894, 2003.



Shirou Maruyama received his B.S. in 2006 in Engineering from Fukuoka University and his M.S. degree in 2008 in Information Science from Kyushu Institute of Technology. He is currently a doctoral student in Graduate School of Information Science and Electrical Engineering, Kyushu University. His research interests include data compression and string pattern matching algorithms.



Youhei Tanaka received his B.S. in 2007 and his M.S. degree in 2009 in Information Science from Kyushu Institute of Technology. He is currently working in TOPPAN PRINTING CO., LTD.

Hiroshi Sakamoto received his B.S. in 1994 in Physics, his M.S. in 1996, and Dr. Sci. degree in 1998 in Information Systems all from Kyushu University. 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. His research interests include algorithms in data compression and web mining. He is a member of JSAI, and DBSJ. He is currently a JST PRESTO researcher.

Masayuki Takeda received his B.S. in 1987 in Mathematics, his M.S. in 1989 in Information Systems, and his Dr. Eng. degree in 1996 all from Kyushu University. From April 1989 to March 1996, he was a research associate of Department of Engineering, From April 1996 to March 2000, an associate professor of Graduate School of Information Science and Electrical Engineering, Kyushu University. He is currently a professor of Graduate School of Information Science and Electrical Engineering, Kyushu University. His research interests include algorithms in string pattern matching, text compression, and string data mining. He is a member of IPSJ, JSAI, and DBSJ.