Processing Text Files as Is: Pattern Matching over Compressed Texts, Multi-byte Character Texts, and Semi-structured Texts

Masayuki Takeda[1,2], Satoru Miyamoto[1], Takuya Kida[3], Ayumi Shinohara[1,2], Shuichi Fukamachi[4], Takeshi Shinohara[4], and Setsuo Arikawa[1]

[1] Department of Informatics, Kyushu University
Fukuoka 812-8581, Japan
{takeda,s-miya,ayumi,arikawa}@i.kyushu-u.ac.jp
[2] PRESTO, Japan Science and Technology Corporation (JST)
[3] Kyushu University Library
Fukuoka 812-8581, Japan
kida@lib.kyushu-u.ac.jp
[4] Department of Artificial Intelligence, Kyushu Institute of Technology
Izuka, 820-8502, Japan
{fukamati,shino}@ai.kyutech.ac.jp

Abstract. Techniques in processing text files "as is" are presented, in which given text files are processed without modification. The compressed pattern matching problem, first defined by Amir and Benson (1992), is a good example of the "as-is" principle. Another example is string matching over multi-byte character texts, which is a significant problem common to oriental languages such as Japanese, Korean, Chinese, and Taiwanese. A text file from such languages is a mixture of single-byte characters and multi-byte characters. Naive solution would be (1) to convert a given text into a fixed length encoded one and then apply any string matching routine to it; or (2) to directly search the text file byte after byte for (the encoding of) a pattern in which an extra work is needed for synchronization to avoid false detection. Both the solutions, however, sacrifice the searching speed. Our algorithm runs on such a multi-byte character text file at the same speed as on an ordinary ASCII text file, without false detection. The technique is applicable to *any* prefix code such as the Huffman code and variants of Unicode. We also generalize the technique so as to handle structured texts such as XML documents. Using this technique, we can avoid false detection of keyword even if it is a substring of a tag name or of an attribute description, without any sacrifice of searching speed.

1 Introduction

In the standard ASCII, alphabetic, numeric, and other special characters are represented in 7 bits and thus there are only 128 codewords. An extra 128 suffices to represent other characters for most western languages. However, an oriental

A.H.F. Laender and A.L. Oliveira (Eds.): SPIRE 2002, LNCS 2476, pp. 170–186, 2002.

language such as Japanese, Korean, Chinese, and Taiwanese, has often much more characters and multi-byte character code is therefore used to represent them. 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. This destroys a comfortable property: one-to-one correspondence between the characters in a text string and the bytes in its representation.

Consider the classical string matching problem over such a text. One naive solution would be to convert a text file into a sequence of two byte codewords by padding each single byte codeword and then apply a favoured string matching algorithm. The method, however, requires an additional time for the conversion which is linearly proportional to the text length, and this is not desirable from the practical viewpoint despite the theoretical complexity does not get worse. We want to process a text file without modification. Another solution would be to directly search the text file for the encoding of a pattern by using a Boyer-Moore type string matching algorithm. An extra work for synchronization is needed to avoid false-detection. Simple and quick verification is possible for some specific codes. For example, we have only to go backward to find a byte in the range of `00`h to `7f`h, not used to represent multi-byte characters in the Japanese EUC. However, we do not adopt such approach by the following reasons: (1) A set of keywords should be simultaneously searched for. Although there are several multipattern variants of the BM algorithm (e.g. [12]), the performance gets worse if the keyword set contains at least one short keyword. (2) Keywords in Japanese are often of length two or three (as the number of characters is very large), and therefore they are represented by 4 or 6 bytes. (3) The above mentioned synchronization technique depends upon a code being used. In general, synchronization requires reading all of the symbols of a text encoded with a prefix code, and therefore it decelerates the over-all task even if we adopt a BM-type algorithm that allows skipping most of the symbols of the encoded text.

Our idea is to merge the synchronization and the string searching tasks into one. This enables us to perform string pattern matching over a multi-byte character text without sacrifices of searching speed. The method is to build a pattern matching machine by embedding a DFA recognizing the set of codewords into an ordinary Aho-Corasick machine and then make it run over a text file byte after byte. Although this synchronization technique has not been published yet, it was adopted as the main engine of the general purpose text database management system SIGMA [4], which has been exploited e.g. for literary analysis of whole volume of novel writers. Also, the technique can be generalized to handle *any* prefix code including the Huffman code, variants of Unicode, and so on. This technique can be generalized to handle *any* prefix code including the Huffman code, variants of Unicode, and so on.

Thus, researchers who must process such multi-byte character text files are forced to be aware of the difference between a text string and its representation. “Processing text files as is” has been one of the most important research topics

for such researchers. The *compressed pattern matching problem*, which is first defined by Amir and Benson [2] and the aim of which is to efficiently perform string pattern matching over a compressed text without decompressing it, can be recognized as a topic along the "as-is" principle.

Here, we present another example along the "as-is" principle. In the query processing for XML documents, it is usual to convert each XML document into a tree structure and then process a user-specified query by node operations over the trees. However, this conversion often consumes much CPU time and a huge amount of memory. It is desirable to process XML documents without conversion, namely, "as is". The problems to be overcome are: (1) to avoid false-detection of keyword when it is a substring of a tag name or of attribute descriptions; and (2) to detect all occurrences of the start and the end tags even when many attribute descriptions are included in a start tag. A naive solution exists which requires an extra work to distinguish the inside and outside of a tag expression, but it sacrifices the searching speed. To resolve these problems, we generalize the synchronization technique so as to handle tag strings in structured texts such as XML documents. Using this method with a stack for storing tag names, we develop a fast pattern matching algorithm for structured texts. Experimental results show that it is approximately $7 \sim 10$ times faster than sgrep [5], an alternative tool for processing structured texts as strings.

2 Preliminaries

Let Σ be a finite set of characters, called an *alphabet*. A finite sequence of characters is called a *string*. The empty string is denoted by ε.Let Σ^* be the set of strings over Σ, and let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string $u = xyz$, respectively. A prefix, factor, and suffix of a string u is said to be *proper* if it is not u. Let $\mathit{Prefix}(u)$ be the set of prefixes of a string u, and let $\mathit{Prefix}(S) = \bigcup_{u \in S} \mathit{Prefix}(u)$ for a set S of strings.

2.1 Prefix Code

Definition 1. *A subset L of Σ^+ is said to have the* prefix property *if no string in L is a proper prefix of another string in L.*

Let Σ be a source alphabet and Δ be a code alphabet. The code alphabet Δ is usually a binary alphabet $\{0, 1\}$, but can be an arbitrary finite set of symbols. A mapping $\mathbf{code} : \Sigma \to \Delta^+$ is called a *prefix code* if and only if:

1. For any $a, b \in \Sigma$, $\mathbf{code}(a) = \mathbf{code}(b)$ implies $a = b$.
2. The set $\mathbf{code}(\Sigma) = \{\mathbf{code}(a) \mid a \in \Sigma\}$ has the prefix property.

An element of $\mathbf{code}(\Sigma)$ is called a *codeword*. With a prefix code **code**, a string $a_1 \dots a_n$ over the source alphabet Σ is encoded as concatenation of the codewords of characters $a_1, \dots, a_n$, namely, $\mathbf{code}(a_1 \dots a_n) = \mathbf{code}(a_1) \dots \mathbf{code}(a_n)$.

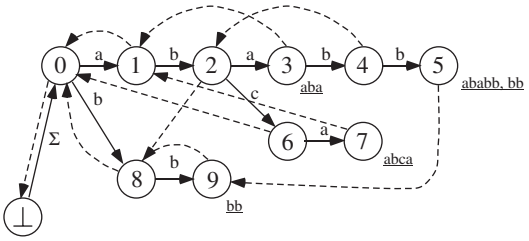

Fig. 1. Aho-Corasick machine for $\Pi = \{aba, ababb, abca, bb\}$. The circles denote states. The solid and the broken arrows represent the goto and the failure functions, respectively. The underlined strings adjacent to the states mean the outputs from them

The Huffman code is a typical example of prefix code, where the code alphabet is $\Delta = \{0, 1\}$. Multi-byte character encoding schemes for oriental languages such as Japanese, Korean, Chinese, and Taiwanese are also prefix codes, where the code alphabet is $\Delta = \{\texttt{00}, \texttt{01}, \ldots, \texttt{ff}\}$, i.e., a set of one byte integers (represented here in hexadecimal) rather than the binary alphabet $\{0, 1\}$.

2.2 Aho-Corasick Pattern Matching Machine

The Aho-Corasick pattern matching machine (AC machine for short) [1] is a finite state machine which simultaneously recognizes all occurrences of multiple patterns in a single pass through a text. The AC machine for a finite set $\Pi \subseteq \Sigma^+$ of patterns consists of three functions: *goto*, *failure*, and *output*. Figure 1 displays the AC machine for the patterns $\Pi = \{aba, ababb, abca, bb\}$.

There is a natural one-to-one correspondence between the states of the AC machine and the pattern prefixes. For example, the initial state 0 corresponds to the empty string ε and the state 4 corresponds to the string *abab* in Fig. 1. Based on this correspondence, the move of the AC machine is characterized as the following theorem.

Theorem 1 ([1]). *Let $a_1 \ldots a_n$ be the text ($n > 0$). After reading a text prefix $a_1 \ldots a_j$ ($1 \leq j \leq n$), the AC machine is in state corresponding to the string $a_i \ldots a_j$ ($1 \leq i \leq j$) such that $a_i \ldots a_j$ is the longest suffix of $a_1 \ldots a_j$ that is also a prefix of some pattern in Π.*

3 Pattern Matching over Variable-Length Encoded Texts

Suppose we are given a text string encoded with a variable-length prefix code, and want to find a pattern within the text. One solution would be to apply a favoured string searching algorithm to find the encoded pattern, with performing an extra work to determine the beginning of codewords within the encoded text. This work can be done if we build a DFA that accepts the language

$\mathbf{code}(\Sigma)^* \subseteq \Delta^*$ and make it run on the encoded text. An occurrence of the encoded pattern that begins at position i is a false match if the DFA is not in accepting state just after reading the $(i-1)$th symbol. However, even if we adopt a BM-type algorithm that allows skipping most symbols of the encoded text, the DFA reads all the symbols in it and therefore this decelerates the over-all task.

Klein and Shapira [6] proposed a probabilistic method: When the encoded pattern is found at index i, jump back by some constant number of symbols, say K, and make the DFA run on the substring of length K starting at the $(i-K)$th symbol. They showed that if K is large enough the probability of false match is low. However, the probability is not zero unless $K = i$.

Our idea is to merge the DFA and the AC machine into one pattern matching machine (PMM). This section illustrates the construction of such PMMs which never report false matches. The technique was originally developed for processing Japanese texts, and then generalized for prefix codes including variants of Unicode, other multi-byte encodings, and the Huffman code. However, the algorithm and its correctness proof have not been published yet. For this reason, the authors present the algorithm and give a correctness proof, together with presenting its applications.

3.1 PMM Construction Algorithm

Let us illustrate the algorithm with an example of the Huffman code. Suppose that the source alphabet $\Sigma = \{A, B, C, D, E\}$, the code alphabet $\Delta = \{0, 1\}$, and the encoding is given by the Huffman tree shown in Fig. 2. Let EC and CD be the patterns to be searched for. Their encodings are, respectively, $\mathbf{code}(EC) = 1001$ and $\mathbf{code}(CD) = 00101$. The AC machine for these bit strings is shown in Fig. 2. This machine, however, leads to false detection.

The way for avoiding false detection is quite simple. We build a DFA that accepts the set of codewords and then embed it into the AC machine. Such a DFA is called the *codeword DFA*. The smallest codeword DFA for the Huffman tree of Fig. 2 is shown in Fig. 3.

Proposition 1. *Let $L \subseteq \Delta^+$ be a regular language over Δ. If L has the prefix property, the smallest DFA accepting L has only one final state.*

From the proposition, if **code** is a prefix code, then the smallest codeword DFA has only one final state. The construction of the smallest codeword DFA is (1) building a codeword DFA as a trie representing the set $\mathbf{code}(\Sigma)$ and (2) then minimizing it. The minimization can be performed in only linear time with respect to the size of the codeword trie, by using the minimization technique [10].

We construct a PMM by embedding the codeword DFA into the ordinary AC machine. The algorithm is summarized as follows. (We omit the construction of the output function.)

Construction of the goto function.

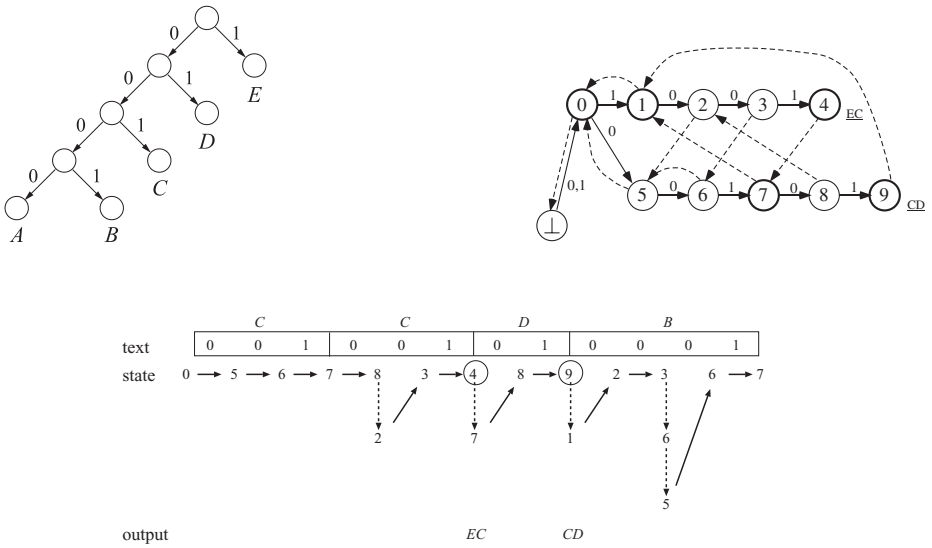

Fig. 2. On the upper-left a Huffman tree is displayed. On the upper-right the ordinary AC machine built from 1001 and 00101 that are the encodings of patterns EC and CD, respectively, is displayed. The move of this machine on the encoding 001001010001 of text $CCDB$ is illustrated on the lower, in which a false detection of pattern EC occurs

1. Build the smallest codeword DFA for **code**, which has a unique final state without outgoing edges. (See the DFA on the upper-left in Fig. 3.)
2. For the given pattern set $\Pi = \{w_1, \ldots, w_k\} \subseteq \Sigma^+$, build a trie for the set $\mathbf{code}(\Pi) = \{\mathbf{code}(w_1), \ldots, \mathbf{code}(w_k)\} \subseteq \Delta^+$. We call the trie *pattern trie*.
3. Replace the unique final state of the codeword DFA by the root node of the pattern trie. (See the state 0 of the PMM on the upper-right in Fig. 3.)

Construction of the failure function.

1. (Basis) Create a failure link from the root of the pattern trie to the initial state of the codeword DFA. (See the broken arrow from state 0 to state 10 in the PMM on the right in Fig. 3.)
2. (Induction Step) Consider a node s of depth $d > 0$ in the pattern trie. Assume the failure links are already computed for all nodes of depth less than d. Let r be the father of the node s and a be the label of the edge from r to s. Start from r and repeat failure transitions until a node r' is found that has an outgoing edge labeled a to some node s'. Create a new failure link from s to s'.

It should be noted that the induction step of the failure function construction is exactly the same as that of the original AC algorithm. The only difference is in the codeword DFA attached to the root node.

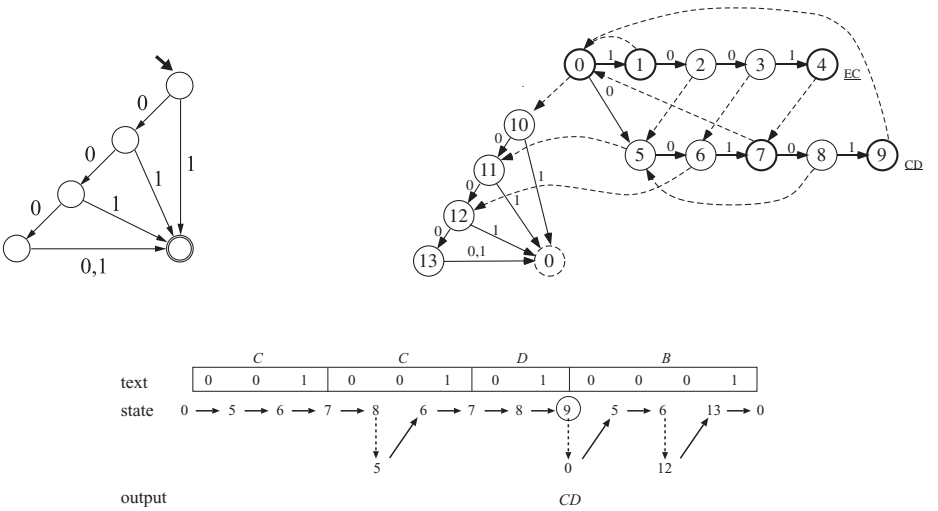

Fig. 3. On the upper-left the smallest DFA that accepts the set of codewords is shown. On the upper-right the PMM obtained by embedding this DFA into the AC machine of Fig. 2. The move of this PMM on the encoding 001001010001 of text $CCDB$ is displayed on the lower. No false detection occurs when using this PMM

3.2 Correctness of the Algorithm

Now, we give a characterization of the PMM constructed by the above mentioned algorithm. Let $\Pi \subseteq \Sigma^+$ be the given pattern set. Since there is a natural one-to-one correspondence between the nodes of the pattern trie for Π and the strings in $\mathit{Prefix}(\mathbf{code}(\Pi))$, a node of the pattern trie can be referred to as a string in $\mathit{Prefix}(\mathbf{code}(\Pi))$. A string u in $\mathit{Prefix}(\mathbf{code}(\Pi))$ can be specified by a pair of a string $x \in \mathit{Prefix}(\Pi)$ over Σ and a string $v \in \Delta^*$ such that $u = \mathbf{code}(x) \cdot v$. Such a reference pair $\langle x, v \rangle$ is said to be *canonical* if x is the longest one. Hereafter, we identify a canonical reference pair $\langle x, v \rangle$ with the corresponding node of the pattern trie, if no confusion occurs. For the pattern set $\Pi \subseteq \Sigma^+$, let denote by S_Π the set of canonical reference pairs to the nodes of the pattern trie for Π. Note that, for every $\langle x, v \rangle$ in S_Π, there exists a character a in Σ such that v is a proper prefix of the codeword $\mathbf{code}(a)$ for a.

Definition 2. *For any pair $\langle x, v \rangle \in S_\Pi$, let y be the longest proper suffix of x such that $\langle y, v \rangle \in S_\Pi$, and denote by $\Phi(x, v)$ the pair $\langle y, v \rangle$. If there is no such pair, let $\Phi(x, v) = nil$.*

The failure function computed by the above algorithm has the following property.

Lemma 1. *Let s be any node of the pattern trie for Π, and $\langle x, v \rangle$ be its canonical reference pair.*

1. *If* $\Phi(x, v) = \langle y, v\rangle$ *is defined, the failure link from* s *goes to the node* $\langle y, v\rangle$.
2. *If* $\Phi(x, v) = nil$, *the failure link from* s *goes to the state of the codeword DFA to which the string* v *leads starting from the initial state.*

Proof. By induction on the depth of the nodes in the pattern trie. When s is of depth 0, that is, $\langle x, v\rangle = \langle \varepsilon, \varepsilon\rangle$, then $\Phi(x, v) = nil$ and the lemma trivially holds. We now consider the nodes of depth > 0.

1. When $v \in \Delta^+$: The string v can be written as $v = wb$ with $w \in \Delta^*$ and $b \in \Delta$. Then, there is a sequence $x_0, x_1, \ldots, x_m \in \Sigma^*$ with $m \geq 0$ such that

$$x_0 = x; \qquad \langle x_i, w\rangle \in S_\Pi \quad (0 \leq i \leq m);$$
$$\Phi(x_{i-1}, w) = \langle x_i, w\rangle \quad (1 \leq i \leq m); \qquad \Phi(x_m, w) = nil.$$

   (a) If there is an integer ℓ $(0 < \ell \leq m)$ such that $\langle x_\ell, wb\rangle \in S_\Pi$, take ℓ as small as possible. By the definition of Φ, $\Phi(x, wb) = \langle x_\ell, wb\rangle$. By the induction hypothesis, for each $i = 1, \ldots, \ell$, the failure link from the node $\langle x_{i-1}, w\rangle$ goes to $\langle x_i, w\rangle$. By the construction of the failure function, the failure link from s goes to the node $\Phi(x, wb) = \langle x_\ell, wb\rangle$.
   (b) If there is no such integer ℓ, $\Phi(x, wb) = nil$. On the other hand, since $\Phi(x_m, w) = nil$, by the induction hypothesis, the failure link from the node $\langle x_m, w\rangle$ goes to the state of the codeword DFA to which the string w leads from the initial state. By the construction of the failure function, the failure link from s goes to the state of the codeword DFA to which the string $v = wb \in \Delta^+$ leads from the initial state.
2. When $v = \varepsilon$: If $x = \varepsilon$, s is the root of the pattern trie. This contradicts the assumption that s is of depth > 0. Hence we have $x \in \Sigma^+$. We can write x as $x'a$ with $x' \in \Sigma^*$ and $a \in \Sigma$. Since there uniquely exist $w \in \Delta^*$ and $b \in \Delta$ with $wb = \mathbf{code}(a)$, we can prove this case in a way similar to 1. □

Theorem 2. *Let* $a_1 \ldots a_n$ *be the text* $(n > 0)$. *After reading* **the encoding of** *a text prefix* $a_1 \ldots a_j$ $(1 \leq j \leq n)$, *PMM is in state corresponding to* **the encoding of** $a_i \ldots a_j$ $(1 \leq i \leq j)$ *such that* $a_i \ldots a_j$ *is the longest suffix of* $a_1 \ldots a_j$ *that is also a prefix of some pattern in* Π.

The boldfaced portions are the difference in comparison with Theorem 1.

3.3 Applications

Searching in Multi-byte Character Texts. The ASCII (American Standard Code for Information Interchange) code, which is the basic character code in the world, represents characters with a 7-bit code. Variants of ISO646 such as BS4730 (England version), DIN66 003 (Germany version), and NF Z62-010 (France version) are also 7-bit codes. A string encoded with such a code is a sequence of bytes, and therefore people who use such codes are hardly conscious of the codeword boundaries. The boundaries, however, must be distinguished in multi-byte encoding scheme for oriental languages, such as Japanese, Korean, Chinese, and Taiwanese. The requirements of such a scheme are as follows.

1. Per byte encoding.
2. Single byte characters in range of `00h`–`7fh`, which is compatible with ASCII.

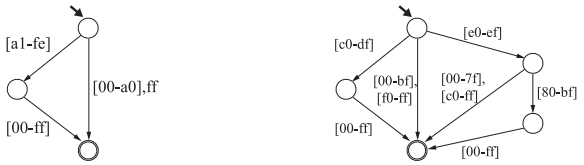

Fig. 4. The codeword DFAs for the Japanese EUC and for UTF-8 (Unicode) are shown on the left and on the right, respectively

3. Multi-byte characters do not begin with `00`h-`7f`h.

The codes are therefore prefix codes.

For example, in the Japanese Extended-Unix-Code (EUC), both the bytes of each two-byte codeword fall into the range of `a1`h to `fe`h. The smallest codeword DFA for the Japanese EUC is shown on the left of Fig. 4, where the code alphabet is $\Delta = \{\texttt{00}, \ldots, \texttt{ff}\}$. By using the codeword DFA, we can process Japanese EUC texts in a byte-by-byte manner. The DFA can also be used for CN-GB (Chinese), EUC-KR (Korean), and BIG5 (Taiwanese).

Both Unicode and ISO10646 are devised to put all the characters in the world into one coded character set. The former is a 16-bit code, and the latter is a 32-bit code. UCS-2 and UCS-4, which are respectively the encoding schemes of Unicode and ISO10646, are fixed-length encodings. Hence we can process texts encoded by such schemes byte after byte by using the codeword DFAs for them (although omitted here). UTF-8, an alternative encoding of Unicode is rather complex. The smallest codeword DFA for UTF-8 is shown on the right in Fig. 4.

Searching in Huffman Encoded Texts. The PMM for a Huffman code makes one state-transition per one bit of an encoded text, and such bit-wise processing is relatively slow. This problem can be overcome by converting PMM into a new one so that it runs in byte-by-byte manner, as shown in [9]. The new PMM runs on a Huffman encoded file faster than the AC machine on the original file. The searching time is reduced at nearly the same rate as the compression ratio.

Dividing a Codeword ([3]). Assume that the number of the states of PMM is ℓ. The size of a two-dimensional table storing the state transition function is then $|\Delta| \cdot \ell$. If a target text is encoded with a byte code, i.e., $\Delta = \{\texttt{00}, \texttt{01}, \ldots, \texttt{ff}\}$, the table size becomes $256 \cdot \ell$. This space requirement could be unrealistic when a great deal of patterns are given. We can reduce the table size with the help of the syncronization technique. Consider dividing each byte of input text into two half bytes and replacing one state transition by two consecutive transitions. Namely, we use the code alphabet $\Delta = \{\texttt{0}, \texttt{1}, \ldots, \texttt{f}\}$ instead of $\{\texttt{00}, \texttt{01}, \ldots, \texttt{ff}\}$, and regard a text and a pattern as sequences of half bytes. The table size is reduced to 1/8, that is, it becomes $|\Delta| \cdot (2\ell) = 32\ell$ even in the worst case. False detection never occurs thanks to the syncronization technique.

3.4 Comparing with BM Algorithm Followed by Quick Verification

In Japanese EUC, bytes in the range of `00`h to `7f`h do not appear in the codewords representing two-byte characters. This provides us with a simple verification technique fast on the average: If an occurrence of the encoded pattern that begins at position i is found, start at the $(i-1)$th byte and go backward until a byte in the range of `00`h to `7f`h is found. The occurrence is false match if and only if the number of bytes not in the range is odd. For codes for which such a quick verification is possible, the "search-then-verify" method might be faster than the PMM-based method. Here, we compare the performance of the "search-then-verify" method with that of the PMM-based method.

We implemented a multipattern variant [12] of the BM algorithm so that the quick verification routine is invoked whenever the encoded pattern is found. On the other hand, the PMM was built from the encoded patterns and converted into a deterministic finite automaton by eliminating failure transitions as described in [1], and the state-transition function was implemented as a two-dimensional array, with the technique so-called *table-look-at* [7].

Our experiment was carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667 MHz running Tru64 UNIX operating system V4.0F. The text file we used is composed of novels written by Soseki Natsume, a Japanese famous novelist, and of size 53.7Mb. The number k of patterns being searched for varied from 1 to 57. Each pattern set consists of uniform length patterns. The pattern length m varied from 2 to 10 bytes. The patterns were text substrings randomly chosen from the text file. For each combination of k and m, we generated 50 pattern sets and made the two algorithms run on the text file for each pattern set. The searching time reported in Fig. 3.4 is the average over the 50 runs. The solid lines show the total time, whereas the broken lines exclude the verification time. The verification time varies depending upon the number of occurrences of the encoded patterns and upon the expected number of bytes inspected in a single call of verification routine. The latter depends only on the nature of a text file. For the text file we used, this value was approximately 369 bytes. It should be stated that almost all occurrences of the single-byte characters in the file are those of the "newline" code. For this reason, almost every verification task for this file ended with encountering it.

Let us focus on the case of $m = 4$, which occurs frequently in Japanese language. The PMM-based method outperforms the "search-then-verify" method for $k > 2$. Even if we exclude the verification time, it is faster for $k > 10$. We observe that the "search-then-verify" method shows a good performance only when (1) all the patterns are relatively long and (2) the number of patterns is relatively small.

4 Generalization of Prefix Codes

The synchronization technique presented in the previous section is applicable to any prefix code. In this section, we generalize the scheme of prefix codes so as to

allow the characters have one or more (possibly infinite) codewords. This may sound strange from the viewpoint of data compression, but has rich applications.

4.1 Generalized Prefix Codes

Recall that a prefix code to express the characters in a source alphabet Σ can be represented by a code tree in which every leaf is associated with a character a in the source alphabet Σ and the path from the root to the leaf spells out the codeword over Δ for a. See again the Huffman tree of Fig. 2. As a generalization of code tree, we consider a DFA called *code automaton* where:

1. there are exactly $|\Sigma|$ final states each corresponding to a character in Σ, and
2. every final state has no outgoing edge.

An example of the code automaton is shown on the left of Fig. 6, where the source alphabet is $\Sigma = \{A, B, C, D, E\}$ and the code alphabet is $\Delta = \{0, 1\}$.

Let denote by f_a the final state associated with a character a in Σ. For any $a \in \Sigma$, there can be more than one string in Δ^+ that leads to the state f_a starting at the initial state. Let such strings be the codewords to express the character a. Let **code** be the encoding determined by a code automaton. Then, **code** maps a character in Σ to a regular set over Δ. In the case of Fig. 6, it is easy to see that $\mathbf{code}(A) = 01^*000$, $\mathbf{code}(B) = 01^*001$, $\mathbf{code}(C) = 01^*01 + 100$, $\mathbf{code}(D) = 101$, and $\mathbf{code}(E) = 11$. One of the encodings of text $ACED$ is 01000 01110 11 101. Since a prefix code is a special case where $|\mathbf{code}(a)| = 1$ for all $a \in \Sigma$, this encoding scheme is a generalization of prefix codes. We call such a code *generalized prefix code*. The subgraph induced by the paths from the initial state to the final state f_a for $a \in \Sigma$ is called the *partial code automaton for* a, and denoted by $\mathbf{PCA}(a)$. Similarly, we define $\mathbf{PCA}(S)$ for a subset S of Σ. Note that $\mathbf{PCA}(S)$ is a DFA that accepts the language $\mathbf{code}(S) = \bigcup_{a \in S} \mathbf{code}(a) \subseteq \Delta^+$.

Our algorithm for constructing PMMs can be extended to generalized prefix codes. The main difference is in the goto function construction.

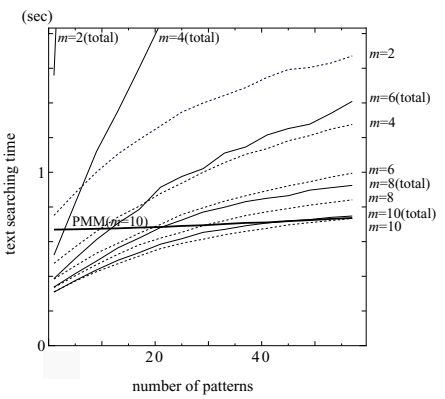

Fig. 5. Comparing the two methods. For the "search-then-verify" method, the solid lines show the total searching time, whereas the broken lines exclude the verification time. For the PMM-based method, we show the searching time only for $m = 10$

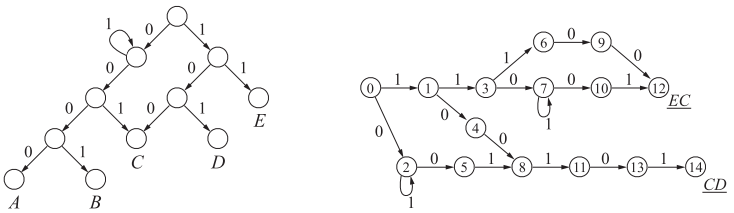

Fig. 6. An example of the code automaton is displayed on the left, and the pattern graph for patterns EC and CD is shown on the right, where the source alphabet is $\Sigma = \{A, B, C, D, E\}$ and the code alphabet is $\Delta = \{0, 1\}$

1. Create a pattern trie from a given pattern set Π.
2. Replace each edge of the pattern trie labeled a with the graph $\mathbf{PCA}(a)$. More precisely, if the node represented by a string $v \in \mathit{Prefix}(\Pi)$ has k outgoing edges labeled $a_1, \ldots, a_k \in \Sigma$, respectively, replace these edges with $\mathbf{PCA}(\{a_1, \ldots, a_k\})$. We call the obtained graph the *pattern graph*. The node corresponding to the root node of the pattern trie is referred to as the *source* node. (An example of the pattern graph is shown on the right in Fig. 6.)
3. Build the smallest codeword DFA that accepts the language $\mathbf{code}(\Sigma) \subseteq \Delta^+$. It can be obtained from the code automaton.
4. Replace the unique final state of the codeword DFA with the root node of the pattern graph.

The construction of the failure function is essentially the same as that for prefix codes.

4.2 Applications

One application is to make no distinction between some characters. People who use a western language might be often faced with the need for ignoring the distinction between upper-case and lower-case of alphabetic characters. More complex situation happens in the case of multi-byte encoding scheme for oriental languages. A two-byte character is usually twice of the width compared to single-byte characters. Wider characters are called “zen-kaku” - meaning full width, narrower characters are called “han-kaku” - meaning half width. The “zen-kaku” characters are usually fixed width. For the sake of proportion to multi-byte characters displayed in the “zen-kaku” form, there are “zenkaku” version of the ASCII characters such as alphabetic, numeric, and other special characters. Thus each of the characters has two representations of different length. Our technique described above is applicable in order to identify the two representations.

In the next section, we mention applications of our techniques to query processing for XML documents.

```
<db>
   <virus>
      <name> W32.Badtrans.B </name>
      <type> worm </type>
      <distribution> high </distribution>
   </virus>
   <virus>
      <name> W32.Zoek </name>
      <type> Trojan horse </type>
      <distribution> low </distribution>
   </virus>
</db>
```

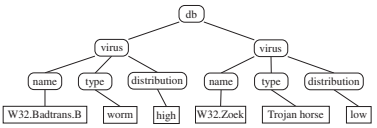

Fig. 7. On the left an example of XML documents is displayed and on the right the tree structure it represents is shown

5 Applications to Query Processing for XML Documents

XML (Extensible Markup Language) is emerging as a standard format for data and documents on the Internet. An example of XML documents and the corresponding tree structure are displayed in Fig. 7.

Several query languages for XML have been proposed, e.g., XQL, Xquery, XPath. Most of them are based on the so-called *path expressions*. An example of the path expressions is "`db/virus/type`", where "`db`", "`virus`", and "`type`" are tag names which are node labels in the tree structure, and "`/`" means a direct descendant. We can also use "`//`" to indicate a descendant (not necessarily direct descendant). In processing path expression based queries, XML documents are usually converted into tree structures, and query processing is performed as node operations over the trees. Another approach stores all possible paths from the root in a tree structure into a relational database [13]. We shall adopt a different approach along our "as-is" principle.

5.1 XML Documents and What They Represent

A tag string in XML documents such as "`<virus>`" represents an "abstract" symbol. We shall put an explicit distinction between such abstract symbols and their representation. From this viewpoint, an XML document represents a string of abstract symbols, and this abstract string is generated by some unambiguous context-free grammar. The tree structure represented by an XML document is essentially the same as the parse tree of the abstract string.

Let Σ_0 be an alphabet. Let $\Gamma = \{ [_1 ,]_1 , \ldots , [_\ell ,]_\ell \}$, where $[_i$ (resp. $]_i$) is the ith left (resp. right) bracket for $i = 1, \ldots, \ell$. Assume $\Sigma_0 \cap \Gamma = \emptyset$, and let $\Sigma = \Sigma_0 \cup \Gamma$ be the source alphabet. The elements of Σ are "abstract" symbols. Consider the context-free grammar $G = (\Sigma, N, P, A_1)$, where N is the set of nonterminals $A_1, \ldots, A_\ell$; A_1 is the start symbol; and P is the set of

productions $A_1 \to [_1\, \alpha_1\,]_1\,, \quad \cdots, A_\ell \to [_\ell\, \alpha_\ell\,]_\ell$, where α_i is a regular expression over $\Sigma_0 \cup N$.

An XML document is an encoding of some element of the language $L(G) \subseteq \Sigma^*$ generated by G. The encodings of the left brackets $[_i$ are called the *start tags*, and those of the right brackets $]_i$ the *end tags*. The start tags are enclosed by the angle brackets "<", ">", and the end tags are enclosed by "</" and ">". The symbols in Σ_0 are encoded with the standard ASCII (or its extension).

For example, consider the grammar with productions $S \to [_S\, A^*\,]_S\,, A \to [_A\, BCD]_A, B \to [_B\, \Sigma_0^*]_B, C \to [_C\, \Sigma_0^*]_C, D \to [_D\, \Sigma_0^*]_D$, where S, A, B, C, D are nonterminals and $[_S,]_S, [_A,]_A, [_B,]_B, [_C,]_C, [_D,]_D$ are four pairs of brackets. Suppose these bracket symbols are encoded as "<db>", "</db>", "<virus>", "</virus>", "<name>", "</name>", "<type>", "</type>", "<distribution>", "</distribution>", respectively. The XML document of Fig. 7 can then be seen as an encoding of some string generated by this grammar.

5.2 Processing XML Documents as Is

We process path expression based queries by using a PMM with stack. Namely, a PMM searches an XML document for tag strings as well as the keywords appearing in the queries. If a start tag is found, then push the corresponding nonterminal into the stack. If an end tag is found, then pop an element from the stack. The stack content is therefore the sequence of nonterminals on the path from the root node to the node corresponding to the current position within the XML document being processed. Path expressions can be regarded as limited regular expressions over the nonterminals, for which DFAs can be built only in linear time. We perform pattern matching of path expressions against the stack content using DFAs. In this way, we can process XML documents with recognizing its structure, without explicitly constructing tree structures.

We are faced with two difficulties in this approach. One is a problem of false-detections which occur when a keyword in the queries appears inside a tag. Of course, this problem is easily resolved if we keep a boolean value which indicates whether the current point is inside the tag enclosed with the angle brackets "<", ">". However, such an extra work requires an additional time. By using the synchronization technique mentioned in the previous section, we resolve the problem without sacrifices of running speed. Since we regard each tag string as an encoding of one symbol from the source alphabet $\Sigma = \Sigma_0 \cup \Gamma$, the above-mentioned false-detection never occurs.

The other difficulty is as follows. A start tag can contain descriptions of attribute values, e.g., as in "<section␣number="1.2">", where "␣" is a blank symbol. Even when we can omit the attribute values because of the irrelevance to the query, we have to identify the start tags with various attribute values whenever they have a unique tag name in common. One naive solution would be to eliminate all the attribute descriptions inside the tags. This, however, contradicts the "as-is" principle and sacrifices the processing speed. Let us recall that the generalized prefix coding scheme allows us to assign more than one codeword to each left-bracket symbol. With this scheme, for example, every

string of the form "<section␣" · w · ">" can be regarded as one of the possible encodings of the left bracket represented simply as "<section>", where w is a string over $\Delta - \{<, >\}$. The difficulty is thus resolved by using the PMMs for generalized prefix codes presented in the previous section.

Moreover, we have only to distinguish the start tags appearing in the path expressions in the queries. The start tags not appearing in the path expressions should not necessarily distinguished. We can take them as encodings of one left bracket symbol. Similarly, the end tags irrelevant to the queries can be treated as different encodings of one right bracket symbol.

5.3 Experimental Results

First, we estimated the performance of PMM with synchronization technique in searching XML documents for keywords, in comparison with that of the ordinary AC machine with extra work for recognizing the inside/outside of a tag. We generated XML documents of various size by using the tool "XMLgen" (XML-benchmark project)[1]. The keywords were chosen from the XML documents. We performed this experiment on the AlphaStation XP1000 mentioned in Section 3.4. Although we omit here the detailed results for lack of space, they demonstrate that the PMM with synchronization runs approximately $1.2 \sim 1.5$ times faster than the ordinary AC machine with extra work.

Next, we implemented a prototype system for processing relatively simple path expression based queries for XML documents, based on the PMM with stack. We then compared its performance with that of sgrep [5]. This experiment was carried out on a Personal Computer with Celelon processor at 366 MHz running Kondara/MNU Linux 2.1 RC2. We used one of the XML documents generated with "XMLgen". It is of size 110 Mb and contains 77 different tag names. The corresponding tree is of height 21. The running times for each of the queries are shown in Table 1. We observe that our prototype system is approximately $7 \sim 10$ times faster than sgrep concerning these queries. Moreover a simultaneous processing of the three queries by our prototype system is only about 15% slower than single query case.

6 Conclusion

Along the "as-is" principle, we have presented a synchronization technique for processing multi-byte character text files efficiently, on the basis of Aho-Corasick pattern matching machine. The technique is applicable to any prefix code. We also generalize the scheme of prefix codes and then extend our algorithm to cope with text files encoded with the generalized scheme. One important application is query processing for XML documents. Experimental results demonstrate that our method is approximately $7 \sim 10$ times faster than sgrep in processing XML documents for some path expression based queries.

[1] http://monetdb.cwi.nl/xml/Benchmark/benchmark.html

Table 1. Running time for query processing (sec). We compared the two methods for the three queries shown in this table. The first query, for example, means "Find the tree nodes labeled `text` from which a text node containing an occurrence of the keyword `summers` is directly descended." We remark that the query language of sgrep is based on the region algebra, and therefore we translated the three queries appropriately

query	PMM with stack	sgrep
`//text/"summers"`	4.794	31.826
`//text//"summers"`	5.056	32.036
`/site/regions/africa/item/location/"United␣States"`	4.880	52.102
simultaneous processing of the three queries	5.614	—

The "as-is" principle also leads us to the question: Which form should be better in storing text? Shibata et al. [11] gave an answer to this question: they showed that a BM-type string pattern matching algorithm running over text files compressed by Byte-Pair Enconding (BPE) — which is a restricted version of Re-Pair [8] — is approximately $1.2 \sim 3.0$ times faster than that in the ASCII format. Namely, the BPE compressed format is one good format from the viewpoint of string pattern matching. Problems of representation of text strings from a viewpoint of processing speed thus attract special concerns.

References

[1] A. V. Aho and M. Corasick. Efficient string matching: An aid to bibliographic search. *Comm. ACM*, 18(6):333–340, 1975. 173, 179

[2] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. Data Compression Conference*, page 279, 1992. 172

[3] S. Arikawa and T. Shinohara. A run-time efficient realization of Aho-Corasick pattern matching machines. *New Generation Computing*, 2(2):171–186, 1984. 178

[4] S. Arikawa et al. The text database management syste SIGMA: An improvement of the main engine. In *Proc. of Berliner Informatik-Tage*, pages 72–81, 1989. 171

[5] J. Jaakkola and P. Kilpeläinen. A tool to search structured text. University of Helsinki. (In preparation). 172, 184

[6] S. T. Klein and D. Shapira. Pattern matching in Huffman encoded texts. In *Proc. Data Compression Conference 2001*, pages 449–458. IEEE Computer Society, 2001. 174

[7] D. E. Knuth. *The Art of Computer Programing, Sorting and Searching*, volume 3. Addison-Wesley, 1973. 179

[8] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference '99*, pages 296–305. IEEE Computer Society, 1999. 185

[9] M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998. (in Japanese). 178

[10] D. Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992. 174

[11] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 181–194. Springer-Verlag, 2000. 185

[12] N. Uratani and M. Takeda. A fast string-searching algorithm for multiple patterns. *Information Processing & Management*, 29(6):775–791, 1993. 171, 179

[13] M. Yoshikawa and T. Amagasa. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology*, 1(1):110–141, August 2001. 182