Pattern Matching Machine for Text Compressed Using Finite State Model

Takeda, Masayuki
Department of Informatics Kyushu University

出版情報：DOI Technical Report. 142, 1997-10. Department of Informatics, Kyushu University
バージョン：
権利関係：

DOI-TR-142

DOI Technical Report

Pattern Matching Machine for Text Compressed Using Finite State Model

by

M. Takeda

October 1997

Department of Informatics
Kyushu University
Fukuoka 812-81, Japan

Email: takeda@i.kyushu-u.ac.jp Phone: +81-92-642-2692

Pattern Matching Machine for Text Compressed Using Finite State Model

Masayuki Takeda
Department of Informatics
Kyushu University
Fukuoka 812-81
Japan
takeda@i.kyushu-u.ac.jp

Abstract

The classical pattern matching problem is to find all occurrences of patterns in a text. In many practical cases, since the text is very large and stored in the secondary storage, most of the time for the pattern matching is dominated by data transmission of the text. Therefore the text compression can speed-up the pattern matching. In this framework it is required to develop an efficient pattern matching algorithm for searching the compressed text directly without decoding. In 1992, Fukamachi et al. proposed a method of constructing pattern matching machine that runs on Huffman coded text, based on the Aho-Coracick algorithm. However, since the Huffman code is optimal only under the assumption of the memoryless source model, the compression ratio is not very high. On the other hand, it is known that English text can be highly compressed by the compression method based on the Markov model. In this paper, we focus our attention on the finite-state model, which subsumes the Markov model as an important special case, and show an algorithm for constructing pattern matching machine for text compressed under the assumption of this model. We also give a proof of the correctness of the algorithm.

1 Introduction

The classical pattern matching problem is to find all occurrences of patterns in a text. In many practical cases, since the text is large and stored in the secondary storage, most of the time for the pattern matching is dominated by the data transmission of the text. It is not easy to decrease the processing time only by improving pattern matching algorithm itself. The data transmission time can be reduced by using text compression. In other words, the text compression can speed-up the pattern matching. In this framework, it is required to develop an efficient pattern matching algorithm for searching directly the compressed text without decoding.

The compressed pattern matching problem has been studied by many researchers [6, 5, 1, 2, 3, 7, 9, 4, 10, 13, 11], mainly from theoretical viewpoints. Most of the

compression methods dealt with are the adaptive compression methods such as the LZ77 compression [15] and the LZW compression [14]. Since in such compression methods the encoding of text substring depends on the previous part of the text, it is needed to keep track of some information about the compression together with the pattern matching. For this reason we shall focus our attention on the non-adaptive compressions in this paper.

Now, the problems to be overcome in the compressed pattern matching are summarized as follows.

(1) It is needed to determine the first bits of codewords in a compressed text when searching for a compressed pattern. That is, a syncronization mechanism is needed to avoid misdetection of the pattern.

(2) Bitwise processing slows down the pattern matching.

(3) The number of the encodings of pattern can grow exponentially, especially in a dictionary based compression method.

In 1994 Manber [11] proposed a simple method that assigns the unused area of the ASCII code to some frequent pairs of characters. A compressed pattern is searched for within a compressed text by an arbitrary algorithm. Since all the codewords are of 8 bits, the problems (1) and (2) do not arise. To avoide the problem (3), he proposed a method of decreasing the number of the encodings of a pattern. The compression ratio is about 70%, and then the running time is reduced to nearly the same rate.

On the other hand, in 1992 Fukamachi et al. [8] presented an algorithm for constructing a pattern matching machine that runs on the Huffman coded text. It is an extension of the Aho-Corasick algorithm. The problem (1) arises since the Huffman code is a variable length code. They solved this problem by incorporating an automaton accepting the set of codewords into the pattern matching machine. This method avoids misdetection of patterns without extra works. Whereas the problem (3) does not arise, the problem (2) is crucial. In fact, we must make state transitions and checks of outputs bit-by-bit. This problem can be overcome by a simple method: Substitute one state-transition for 4 consecutive state-transitions caused by 4 bits of the Huffman coded text. The experimental results on the Brown corpus showed that the running time is reduced to 64% of the uncompressed case [12].

To highly speed-up the pattern matching, we shall consider to improve the compression ratio. The Huffman code is optimal only when the memoryless source model is assumed. It is known that English texts show a good compression ratio when compressed under an assumption of the Markov model. In this paper, we use the finite-state model in which the probablity distribution is conditioned by the *current state*. The Markov model is an important special case of this model.

The compression based on this model can be formalized as a finite-state encoder (FSE, for short). An FSE reads each character of text, emits the corresponding codeword, and then changes its state. That is, an FSE is a kind of Mealy type generalized sequential machine. In this paper we show an algorithm for constructing pattern matching machine that runs on the text coded by FSE. We also give a correctness proof of the algorithm.

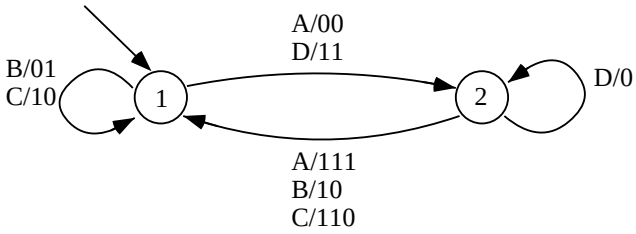
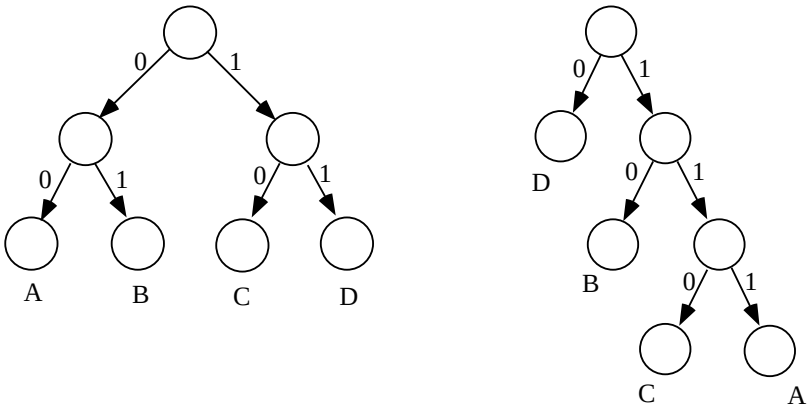


Figure 1: Finite-state encoder with two states.



for state 1 for state 2

Figure 2: Code-trees.

2 Finite-State Encoder

Figure 1 shows an example of an FSE with two states, where the source alphabet is $\{A, B, C, D\}$, and the code alphabet is $\{0, 1\}$. The code-trees for the two states 1 and 2 are shown in Fig. 2.

Formally, an FSE is a 6-tuple $(Q, q_0, \Sigma, \Delta, \delta, \lambda)$, where Q is a finite set of states; q_0 in Q is the initial state; Σ is a source alphabet; Δ is a code alphabet; $\delta : Q \times \Sigma \to Q$ is a state-transition function; and $\lambda : Q \times \Sigma \to \Delta^*$ is a coding function which satisfies the condition that, for all q in Q and all a, b in Σ, if $\lambda(q, a)$ is a prefix of $\lambda(q, b)$, then $a = b$. Define for each state q in Q the function $\varphi_q : \Sigma \to \Delta^*$ by $\varphi_q(a) = \lambda(q, a)$ $(a \in \Sigma)$. Then, the above condition means that, for any state q, φ_q is a one-to-one mapping and the set of codewords $C_q = \{\varphi_q(a) | a \in \Sigma\}$ has the prefix property.

input:		D		B		B		D		C		A		B	
state:	1	→	2	→	1	→	1	→	2	→	1	→	2	→	1
output:		11		10		01		11		110		00		10	

Figure 3: Move of FSE.

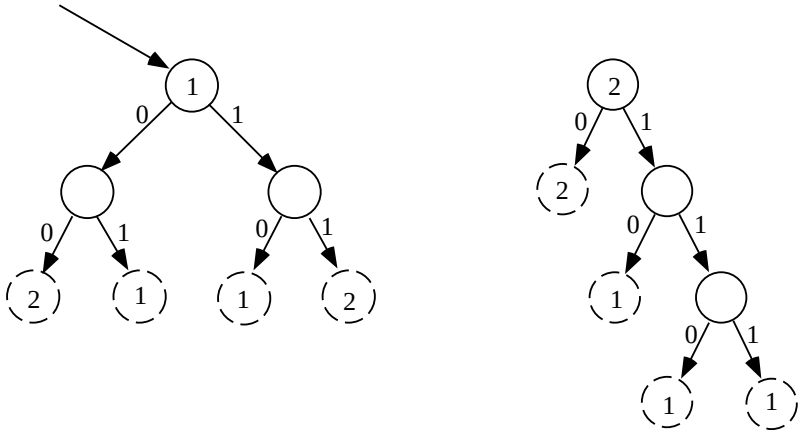


Figure 4: Automaton that accepts the set of encoded strings.

The broken line circles indicate the root node with the same number.

Extend δ into the function from $Q \times \Sigma^*$ to Q by

$$\begin{cases} \delta(q, \varepsilon) = q, \\ \delta(q, xa) = \delta(\delta(q, x), a), \end{cases}$$

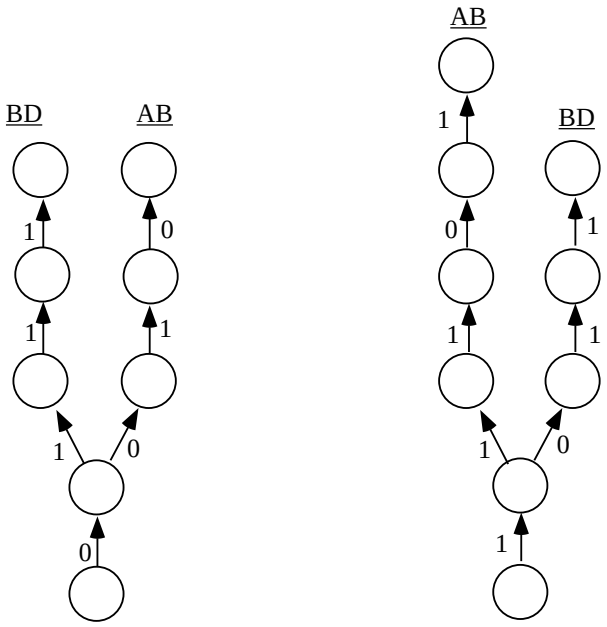
and then extend λ into the function from $Q \times \Sigma^*$ to Δ^* by

$$\begin{cases} \lambda(q, \varepsilon) = \varepsilon, \\ \lambda(q, xa) = \lambda(q, a) \cdot \lambda(\delta(q, x), a), \end{cases}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$. The encoding of a text $T \in \Sigma^*$ by an FSE is then defined to be the string $\lambda(q_0, T)$.

For example, the FSE of Fig. 1 takes as input a text $DBBDCAB$, makes state-transitions of $1 \to 2 \to 1 \to 1 \to 2 \to 1 \to 2 \to 1$, and emits string $\lambda(1, DBBDCAB) =$ 11 10 01 11 110 00 10. See Fig. 3.

To find keywords in a compressed text without decoding, we have to scan the text with recognizing the state changes of the FSE. Figure 4 shows the automaton accepting the set of strings encoded by the encoder of Fig. 1. The automaton is obtained from the code-trees of Fig. 2 in a simple way. Notice that every leaf of the code-trees is replaced by the root of some code-tree. The replacement should be done according to the state-transition function δ of the encoder. For example, since $\delta(2, C) = 1$ in Fig. 1, the leaf representing the symbol C in the code-tree for state 2 is replaced by the root of the code-tree for state 1.



for state 1 for state 2

Figure 5: Keyword-trees for AB and BD.

The thick line circles correspond to the first bit of code-words.

Suppose the keywords are AB and BD. In state 1 we shall search for the strings $\lambda(1, AB) = 00\,10$ and $\lambda(1, BD) = 01\,11$, and in state 2 the strings $\lambda(2, AB) = 111\,01$ and $\lambda(2, BD) = 10\,11$. The keyword-trees for states 1 and 2 are shown in Fig. 5.

PMM for text coded by FSE is constructed from the automaton of this kind and the keyword-trees. The next section presents a method of constructing such PMM.

3 PMM for text coded by FSE

A PMM is composed of the goto, the failure, and the output functions, which are denoted by g, f, and o, respectively. Figure 6 shows PMM for the running example. The solid and the broken arrows represent the goto and the failure functions, respectively. The underlined strings adjacent to the nodes mean the outputs from them. The goto function is obtained by combining the keyword-trees of Fig. 5 with the automaton of Fig. 4. Note that the leaves of the code-trees represented by broken circles mean the root nodes of the keyword-trees with the same number.

The computation of the failure function are almost the same as Aho-Corasick's. It is computed for all nodes in the keyword-trees. Define the depth of a node in a keyword-tree to be the length of the path from the root. We shall compute the failure function for all nodes of depth 0, then for all nodes of depth 1, and so on, until the failure function has been computed for all nodes in the keyword-trees. The difference

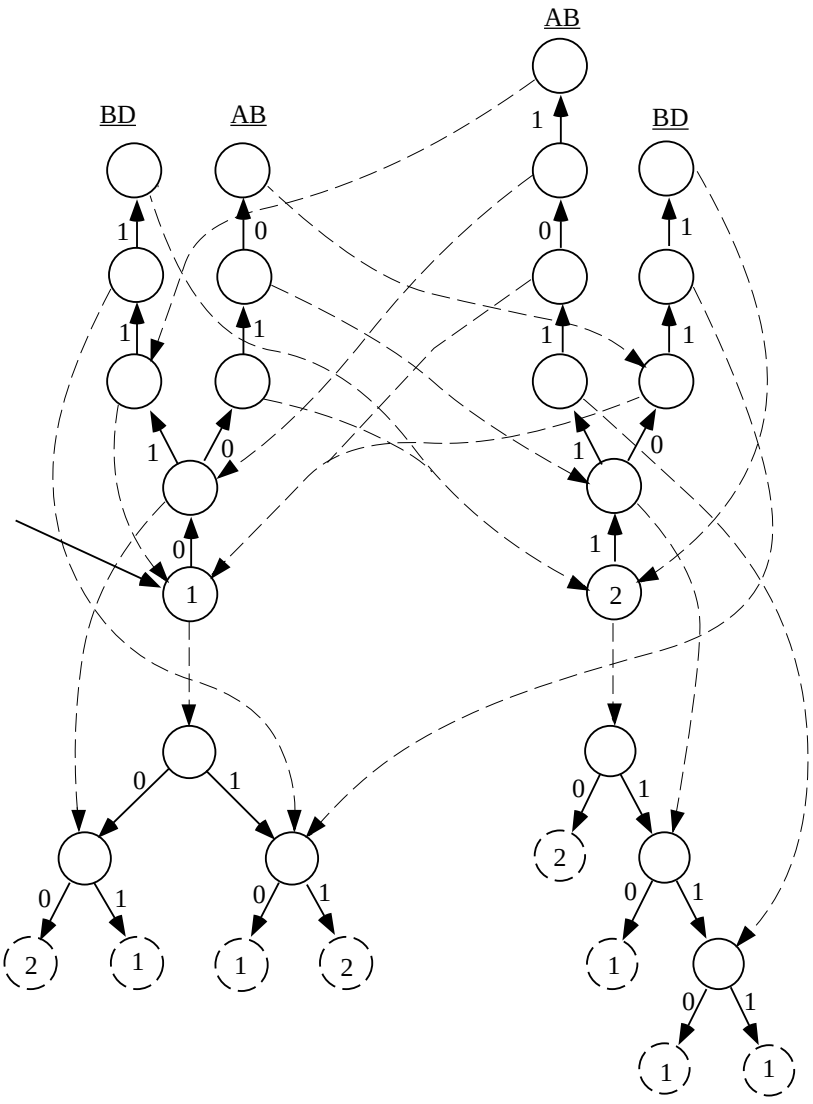


Figure 6: Pattern matching machine

is on the values of the failure function for the nodes of depth 0. The computation is as follows: Let n_i be the root node of the keyword-tree corresponding to a state i of the encoder, and set $f(n_i)$ to be the root node of the code-tree corresponding to the state i. The values for the nodes of depth $d > 0$ are computed from the falilure function for the nodes of depth less than d, in exactly the same manner as Aho-Corasick's algorithm. That is, we consider each node r of depth $d-1$ and perform the following actions.

1. If $g(r,a) = fail$ for all a in Δ, do nothing.
2. Otherwise, for each a in Δ with $g(r,a) = s$, do the following:
   (a) Set $t = f(r)$.
   (b) Execute the statement $t \leftarrow f(t)$ zero or more times, until a value for t is obtained with $g(t,a) \neq fail$.
   (c) Set $f(s) = g(t,a)$.

The computation of the output function is the same as Aho-Corasick's.

4 Correctness of Construction of PMM

A PMM is said to be *valid* for a set of patterns Π if it reports that pattern $\pi \in \Pi$ ends at position i of text T if and only if $T = u\pi v$ and $|u\pi| = i$. This section gives a correctness proof of the construction algorithm of our PMM.

Let

$$S = \left\{ \langle p,x,u\rangle \in Q \times \Sigma^* \times \Delta^* \;\middle|\; \begin{array}{l} u \text{ is a proper prefix of} \\ \text{the codeword } \lambda(\delta(p,x),a) \\ \text{for some } a \in \Sigma. \end{array} \right\}.$$

Let us define an equivalence relation $\equiv$ on S by

$$\langle p,x,u\rangle \equiv \langle q,y,v\rangle \quad \Leftrightarrow \quad \delta(p,x) = \delta(q,y) \quad \text{and} \quad u = v.$$

For a subset X of Σ^+, put

$$S_X = \left\{ \langle q,x,u\rangle \in S \;\middle|\; \begin{array}{l} \lambda(q,x)\cdot u \text{ is a prefix of} \\ \text{the encoded string } \lambda(q,w) \\ \text{for some } w \in X \text{ such that} \\ x \text{ is a prefix of } w. \end{array} \right\}.$$

Let K be the set of keywords. Then there is a natural one-to-one correspondance between S_K and the set of nodes in the keyword-trees. Every node in the keyword-trees is represented by an element in S_K. Similarly, every node in the code-trees is represented by an element in S_Σ. Of course, an element in $S_K \cap S_\Sigma$ represents both a node in the keyword-trees and a node in the code-trees.

Definition 1 *For any $\langle p,x,u\rangle$ in S_K with $x \neq \varepsilon$, let $\Phi(p,x,u)$ be the tupple $\langle q,y,v\rangle$ in $S_K \cup S_\Sigma$ such that y is the longest proper suffix of x with $\langle p,x,u\rangle \equiv \langle q,y,v\rangle$.*

Note that $\Phi(p, x, u)$ is always defined for $x \neq \varepsilon$ because of $\langle p, x, u\rangle \equiv \langle \delta(p, x), \varepsilon, u\rangle \in S_\Sigma$.

Lemma 1 *Let s be a node in the keyword-trees that is represented by $\langle p, x, u\rangle$.*

(1) When $x = \varepsilon$, $f(s)$ is the node in the code-trees that is represented by the same tupple $\langle p, x, u\rangle$.

(2) When $x \neq \varepsilon$, if $\Phi(p, x, u)$ is in S_K, $f(s)$ is the node in the keyword-trees that is represented by $\Phi(p, x, u)$; otherwise, $f(s)$ is the node in the code-trees that is represented by $\Phi(p, x, u)$.

Proof. By the induction on depth of s. It is easy to show (1). To prove (2), we consider the following two cases.

a) When $u \neq \varepsilon$, u is written as $u = wa$ for some w in Δ^*, a in Δ. Then there exist a sequence of states $p_0, p_1, \ldots, p_m$ and a sequence of strings $x_0, x_1, \ldots, x_m$ such that

$$
\begin{array}{l}
m > 0; \\
p_0 = p; x_0 = x; \\
\langle p_i, x_i, w\rangle \in S_K \cup S_\Sigma \quad (0 \leq i \leq m); \\
\Phi(p_{i-1}, x_{i-1}, w) = \langle p_i, x_i, w\rangle \quad (1 \leq i \leq m); \\
\langle p_i, x_i, wa\rangle \notin S_K \cup S_\Sigma \quad (1 \leq i < m); \\
\langle p_m, x_m, wa\rangle \in S_K \cup S_\Sigma.
\end{array}
$$

By the definition of Φ we have

$$\Phi(p, x, wa) = \langle p_m, x_m, wa\rangle.$$

By the induction hypothesis and the computation of the failure function, we can see that the node $f(s)$ is represented by $\Phi(p, x, wa)$.

b) When $u = \varepsilon$, we can prove in a similar way that $f(s)$ is represented by $\Phi(p, x, \varepsilon)$.

Lemma 2 *Let s be a node in the keyword-trees that is represented by $\langle p, x, u\rangle$. Then, the state s has $\alpha \in K$ as output if and only if $u = \varepsilon$ and α is a suffix of x.*

From Lemmas 1 and 2, we have the following theorem.

Theorem 1 *PMM constructed by the algorithm presented in the previous section is valid.*

5 Concluding remarks

In this paper, we addressed the problem of compressed pattern matching in which texts are compressed by finite state encoders (FSE). We presented an algorithm for constructing pattern matching machine that runs on a text compressed by an FSE. We also gave a correctness proof of the algorithm.

One of the future directions of this study is to establish a way of finding a 'good' model for a given text.

References

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

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 440–452, 1992.

[3] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. In *Proc. 21st International Colloquium on Automata, Languages and Programming*, pages 215–226, 1994.

[4] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.

[5] A. Amir, G. M. Landau, and U. Vishkin. Efficient pattern matching with scaling. *Journal of Algorithms*, 13(1):2–32, 1992.

[6] T. Eilam-Tzoreff and U. Vishkin. Matching patterns in strings subject to multi-linear transformations. *Theoretical Computer Science*, 60(3):231–254, 1988.

[7] M. Farach and M. Thorup. String-matching in Lempel-Ziv compressed strings. In *27th ACM STOC*, pages 703–713, 1995.

[8] S. Fukamachi, T. Shinohara, and M. Takeda. String pattern matching for compressed data using variable length code — Efficient retrieval of Genome information. In *Proc. Symposium on Infomatics 1992*, pages 95–103, 1992.

[9] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encoding. In *Proc. 4th Scandinavian Workshop on Algorithm Theory*, volume 1097 of *Lecture Notes in Computer Science*, pages 392–403. Springer-Verlag, 1996.

[10] M. Karpinski, W. Rytter, and A. Shinohara. Pattern-matching for strings with short descriptions. In *Proc. Combinatorial Pattern Matching*, volume 637 of *Lecture Notes in Computer Science*, pages 205–214. Springer-Verlag, 1995.

[11] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 113–124. Springer-Verlag, 1994.

[12] M. Miyazaki. Speeding-up of pattern matching machines for compressed texts by using look-ahead techniques. Diploma thesis (in Japanese), Kyushu Institute of Technology, 1996.

[13] M. Miyazaki, A. Shinohara, and M. Takeda. An improved pattern matching algorithm for strings in terms of straight-line programs. In *Proc. Combinatorial Pattern Matching*, volume 1264 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1997.

[14] T. A. Welch. A technique for high performance data compression. *IEEE Comput.*, 17:8–19, June 1984.

[15] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inform. Theory*, IT-23(3):337–349, May 1977.