A Boyer-Moore type algorithm for compressed pattern matching

Shibata, Yusuke
Department of Informatics, Kyushu University

Matsumoto, Tetsuya
Department of Informatics, Kyushu University

Takeda, Masayuki
Department of Informatics, Kyushu University

Shinohara, Ayumi
Department of Informatics, Kyushu University

他

https://hdl.handle.net/2324/3029

出版情報：DOI Technical Report. 170, 1999-01. Department of Informatics, Kyushu University
バージョン：
権利関係：

DOI-TR-170

𝔻𝕆𝕀 Technical Report

A Boyer-Moore type algorithm for compressed pattern matching

by

Yusuke Shibata, Tetsuya Matsumoto, Masayuki Takeda, Ayumi Shinohara and Setsuo Arikawa

January 1999

Department of Informatics
Kyushu University
Fukuoka 812-8581, Japan

Email: tetsuya@i.kyushu-u.ac.jp Phone: +81-92-642-2697

A Boyer-Moore type algorithm for compressed pattern matching

Yusuke Shibata Tetsuya Matsumoto Masayuki Takeda
Ayumi Shinohara
Setsuo Arikawa
Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{ yusuke, tetsuya, takeda, ayumi, arikawa } @i.kyushu-u.ac.jp

January 14, 2000

Abstract

Recently the compressed pattern matching problem has attracted special concern, where the goal is to find a pattern in a compressed text without decompression. In previous work, we proposed an Aho-Corasick (AC) type algorithm for searching in text files compressed by the so-called byte pair encoding (BPE). The searching time is reduced at the same rate as the compression ratio compared with AC. In this paper, we show a Boyer-Moore (BM) type algorithm for pattern matching in BPE compressed files. Experimental results show that the algorithm runs about 1.5 $\sim$ 3.0 times faster than the exact match routines based on the BM algorithm in the software package `Agrep`, which is known as the fastest pattern matching tool.

1 Introduction

The problem of compressed pattern matching is to find pattern occurrences in compressed text without decompression. It has been extensively studied for various compression methods by many researchers in the last decade. For recent developments, see the survey [17].

Let n and N denote the compressed text length and the original text length, respectively. Theoretically, the best compression has $n = \sqrt{N}$ for the Lempel-Ziv-Welch (LZW) encoding [21], and $n = \log N$ for LZ77 [24]. Thus an $O(n)$ time algorithm for searching directly in compressed text is considered to be better than a simple $O(N)$ time algorithm for searching in the original text.[1] However, in practice n is linearly proportional to N for real text files. For this reason, an elaborate $O(n)$ time

[1] The $O(n)$ time algorithm requires an extra $O(r)$ time in order to report all pattern occurrences, where r is the number of them, and r could be linear in N. But, we here ignore the $O(r)$ factor.

algorithm for searching in compressed text is often slower than a simple $O(N)$ time algorithm running on the original text. For example, as shown in [12, 16], searching in LZW compressed files is slower than searching in the original files though it is fast in comparison with a regular decompression followed by a simple search. In order to speed up pattern matching by text compression, we have to choose an appropriate compression method paying attention to the constant factors hidden behind the O-notation.

The first attempt was made by Manber [15]. The approach is to encode a given pattern and to apply any search routine in order to find the encoded pattern within compressed files. The problem in this approach is that the pattern may have more than one encoding. A compression scheme based on the pattern-substitution [10] is introduced in which the number of possible encodings of any string is restricted. The reductions in file size and searching time, however, are not so good (only about 30%).

In a recent work [18], we focused on the compression method called the byte pair encoding (BPE) [8], and showed that it is very suitable for speeding up pattern matching. BPE is also based on the pattern-substitution.[2] The basic operation is to substitute a single character which did not appear in the text for a pair of consecutive two characters which frequently appears in the text. This operation will be repeated until either all characters are used up or no pair of consecutive two characters appears frequently. We proposed in [18] an algorithm for searching in BPE compressed files, which runs in $O(n)$ time after some preprocessing. It is indeed faster than the famous $O(N)$ time algorithms, such as the Knuth-Morris-Pratt (KMP) algorithm [13] and the Shift-Or algorithm [23, 3]. The searching time is reduced at nearly the same rate as the compression ratio. Thus we have shown that text compression by BPE speeds up these $O(N)$ time algorithms.

However, there are sublinear time algorithms, such as the Boyer-Moore (BM) algorithm [4], which skip many characters of text and run faster than the $O(N)$ time algorithms on the average. The software package `Agrep`, known as the fastest pattern matching tool, uses the Horspool variation [9] of the BM algorithm. Our algorithm in [18] is not better than `Agrep` in the case of searching for a long pattern in text files that are not highly compressible by BPE, whereas it defeats `Agrep` for highly compressible text files such as biological data. Then a question arises: *Does text compression speed up such a sublinear time algorithm?*

In this paper, we give an affirmative answer to this question. We show a BM type algorithm for searching in BPE compressed files. To our best knowledge, this is the first attempt to develop such an algorithm in compressed text. Recall that in the dictionary-based methods a compressed text can be viewed as a pair of a dictionary $\mathcal{D}$ and a sequence $\mathcal{S}$ of tokens, each of which represents a phrase in defined in $\mathcal{D}$. The proposed algorithm runs on the sequence $\mathcal{S}$, with skipping some tokens. The token-wise processing has two advantages in comparison with the usual character-

[2] The compression method named Re-Pair [14] is considered as a generalization of BPE.

wise processing. One is quick detection of a mismatch at each stage of the algorithm, and the other is larger shift depending upon one token (not upon one character) to align the phrase with its occurrence within the pattern. Disadvantage is that the shift value is divided by C, the maximum phrase length, assuming that the phrases corresponding to the skipped tokens are all of length C. Therefore the value of C is a crucial factor. We observed that putting a restriction on C makes no great sacrifice of compression ratio even for $C = 3, 4$. Experimental results show that the proposed algorithm defeats the BM type algorithms running on the original text files. Especially, it is about $1.5 \sim 3.0$ times faster than `Agrep`.

It should be emphasized that Moura et al. [7] proposed a compression scheme that uses a word-based Huffman encoding with a byte-oriented code. They presented an algorithm which runs twice faster than `Agrep` [22]. However, the compression method is not applicable to such texts as biological sequence data, which cannot be segmented into words. Both of our previous and new algorithms can deal with such texts.

2 Efficiency of compressed pattern matching

In order to achieve a fast search in compressed files, we have to re-estimate the existing compression methods in the light of the new criterion: *Efficiency of compressed pattern matching*.

As an effective tool for such re-estimation, we introduced in [11] a unifying framework, named *collage system*, which abstracts various dictionary-based compression methods, such as the Lempel-Ziv family, BPE, and the static dictionary methods. In the framework, a text string is described by a pair of a dictionary $\mathcal{D}$ and a sequence $\mathcal{S}$ of tokens, each of which represents a phrase defined in $\mathcal{D}$. The dictionary $\mathcal{D}$ is given as a sequence of assignments where the basic operations are concatenation, repetition, and prefix (suffix) truncation. We developed in [11] a general compressed pattern matching algorithm for texts described in terms of collage system. Consequently, any of the compression methods that can be described within the framework has a compressed pattern matching algorithm as an instance. We denote by $t.u$ the phrase represented by a token t. Let $\mathcal{S} = \mathcal{S}[1:n]$. The original text is thus $\mathcal{S}[1].u \cdot \mathcal{S}[2] \cdots \mathcal{S}[n].u$. Let $\|\mathcal{D}\|$ and $\mathit{height}(\mathcal{D})$ respectively denote the number of assignments in $\mathcal{D}$ and the maximum dependency in $\mathcal{D}$. Let $\pi = \pi[1:m]$ be a given pattern. Let r be the number of all occurrences of π in the text.

Theorem 1 (Kida et al. 1999) *The problem of compressed pattern matching can be solved in $O((\|\mathcal{D}\| + n) \cdot \mathit{height}(\mathcal{D}) + m^2 + r)$ time using $O(\|\mathcal{D}\| + m^2)$ space. If $\mathcal{D}$ contains no truncation, the time complexity becomes $O(\|\mathcal{D}\| + n + m^2 + r)$.*

Figure 1 gives an overview of the algorithm presented in [11]. It is an on-line algorithm in the sense that it processes $\mathcal{S}$ token-by-token. The algorithm simulates the move of the KMP automaton running on the original text, by using two functions *Jump*

```
Input:    Pattern $\pi$ and compressed text consisting of $\mathcal{D}$ and $\mathcal{S} = \mathcal{S}[1:n]$.
Output:   All occurrences of $\pi$ in the original text.
begin
/* Preprocessing for computing Jump and Output. */
    Preprocess the pattern $\pi$ and the dictionary $\mathcal{D}$;
/* Main routine */
    $state := 0$;    $\ell := 0$;
    for $i := 1$ to $n$ do begin
        for each $d \in Output(state, \mathcal{S}[i])$ do
            Report a pattern occurrence that ends at position $\ell + d$;
        $state := Jump(state, \mathcal{S}[i])$;    $\ell := \ell + |\mathcal{S}[i].u|$
    end
end.
```

Figure 1: On-line algorithm for searching in compressed text.

and *Output*, both take as input a state and a token. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton caused by each of the phrases, and the latter is used to report all pattern occurrences found during the state transitions. This idea is essentially based on the algorithm for searching in LZW compressed text due to Amir et al. [2] which finds only the leftmost pattern occurrence. The extension to find all pattern occurrences was achieved by Kida et al. in [12], together with an extension to the multiple pattern problem.

The above idea can also be applied to compressed pattern matching for other compression methods that are not contained in the collage system. For instance, we presented in [19] an algorithm, based on the similar idea, for searching in texts compressed using anti-dictionaries [5].

Theorem 1 suggests that a compression method described as a collage system with no truncation might be suitable for the speed-up of pattern matching. In fact the collage system for LZ77 has truncation and LZ77 is not suitable as shown in [16]. The collage system for LZW has no truncation. However, the dictionary $\mathcal{D}$ is not encoded explicitly: it will be incrementally re-built from $\mathcal{S}$. The preprocessing of $\mathcal{D}$ is therefore merged into the main routine. This is one of the reasons why compressed pattern matching for LZW is slow. Another reason is as follows. Although *Jump* can be realized using only $O(\|\mathcal{D}\| + m)$ space so that it answers in constant time, the constant is relatively large. But the two-dimensional array realization requires $O(\|\mathcal{D}\| \cdot m)$ space, which is unrealistic because $\|\mathcal{D}\|$ is linear with respect to n in the case of LZW.

BPE is a collage system with no truncation. Unlike LZW and LZ77, BPE has the following good properties.

- The dictionary $\mathcal{D}$ is encoded separately from the sequence $\mathcal{S}$.

```
$T[0] := \$$; /* $\$$ is a character that never occurs in pattern */
$i := m$;
while $i \leq N$ do begin
    $state := 0$;    $\ell := 0$;
    while $g(state, T[i - \ell])$ is defined do begin
        $state := g(state, T[i - \ell])$;
        $\ell := \ell + 1$
    end;
    if $state = m$ then report a pattern occurrence;
    $i := i + shift(state, T[i - \ell])$
end
```

Figure 2: BM type algorithm on uncompressed text.

- The size of $\mathcal{D}$ is small enough (i.e. $\|\mathcal{D}\| \leq 256$).

- The tokens of $\mathcal{S}$ are encoded using a fixed length code.

We can take the two-dimensional array realization of *Jump* as in [18], which is realistic since the array size is only $256 \cdot (m + 1)$.

The maximum phrase length C is crucial for a sublinear time search in compressed text. Experimental result shows that we can put a restriction on C with little sacrifice of compression ratio, e.g. $C = 3$ or 4. Thus, BPE is considered suitable for our purpose.

3 BM type algorithm for compressed search

We first briefly sketch the BM algorithm, and then present a BM type algorithm for searching in BPE compressed files.

3.1 BM type algorithm on uncompressed text

The BM type algorithm performs the character comparisons in the right-to-left direction, and slides the pattern to the right using the so-called shift function when a mismatch occurs. The algorithm for searching in text $T[1 : N]$ is shown in Fig. 2. Note that the function g is the state transition function of the (partial) automaton that accepts the reversed pattern, in which state j represents the length j suffix of the pattern. Such an automata-oriented description of the character comparisons is for convenience of explanation of the algorithm in Section 3.2.

Although there are many variations of the shift function, they are basically designed to shift the pattern to the right so as to align a text substring with its rightmost occurrence within the pattern. Let

$$rightmost\_occ(w) = \min\left\{ \ell > 0 \;\middle|\; \begin{array}{l} \pi[m-\ell-|w| : m-\ell] = w, \text{ or} \\ \pi[1 : m-\ell] \text{ is a suffix of } w \end{array} \right\}.$$

The following definition, presented in [20] (for multiple pattern case), is the one which utilizes all information gathered in one stage.

$$shift(j, a) = rightmost\_occ(a \cdot \pi[m-j+1 : m]).$$

3.2 BM type compressed pattern matching

Figure 3 gives an overview of our algorithm. For each iteration of the **while**-loop, we determine in Step 1 the pattern occurrences that end within the phrase represented by the token we focus on, and then shift our focus to the right by Δ obtained in Step 2. In the following we discuss how to realize Step 1 and Step 2.

Figure 4 illustrates pattern occurrences that end within the focused phrase. Note that we assume the pattern length m is sufficiently larger than the maximum phrase length C. A candidate for pattern occurrence is a prefix of the focused phrase that is also a suffix of the pattern. There may be more than one candidate to be checked. Naive method is to check all of them independently, but we take here another approach. We shall start with the longest one. For the case of uncompressed text, we can do it by using the partial automaton for the reversed pattern stated in Section 3.1. When a mismatch occurs, we change the state by using the failure function and try to proceed into the left direction. The process is repeated until the pattern does not have an overlap with the phrase at which we started. In order to perform such processing in compressed text, we use the two functions *Jump* and *Output* defined in the sequel, which differ from those mentioned in Section 2.

Let *lps*(w) denote the longest prefix of a string w that is also a suffix of the pattern π. Extend the function g into the domain $\{0, 1, \ldots, m\} \times \Sigma^*$ by $g(j, aw) = g(g(j, w), a)$, if $g(j, w)$ is defined; undefined, otherwise, where $w \in \Sigma^*$ and $a \in \Sigma$. Let $f(j)$ be the largest integer k $(k < j)$ such that the length k suffix of the pattern is a prefix of the length j suffix of the pattern. Note that f is the same as the failure function of the KMP automaton. The functions *Jump* and *Output* are defined as

```
        $focus := \lceil m/C \rceil$;
        while $focus \le n$ do begin
Step 1:     Find all pattern occurrences that end within the string $\mathcal{S}[focus].u$;
Step 2:     Compute a possible shift $\Delta$ based on information gathered in Step 1;
            $focus := focus + \Delta$
        end
```

Figure 3: Overview of sublinear time algorithm.

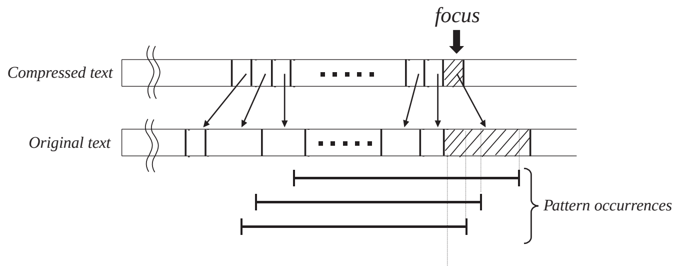


Figure 4: Pattern occurrences.

.

```
procedure $Find\_pattern\_occurrences(focus : \textbf{integer})$;
begin
    if $Jump(0, \mathcal{S}[focus])$ is undefined then return;
    $state := Jump(0, \mathcal{S}[focus])$;
    $d := state$;    $\ell := 1$;
    repeat
        while $Jump(state, \mathcal{S}[focus - \ell])$ is defined do begin
            $state := Jump(state, \mathcal{S}[focus - \ell])$;
            $\ell := \ell + 1$
        end;
        if $Output(state, \mathcal{S}[focus - \ell]) = true$ then report a pattern occurrence;
        $d := d - (state - f(state))$;
        $state := f(state)$
    until $d \leq 0$
end;
```

Figure 5: Finding pattern occurrences in Step 1.

follows.

$$Jump(j,t) \;=\; \begin{cases} g(j, t.u), & \text{if } j \neq 0; \\ g(j, lps(t.u)), & \text{if } j = 0 \text{ and } lps(t.u) \neq \varepsilon; \\ \text{undefined}, & \text{otherwise.} \end{cases} ,$$

$$Output(j,t) \;=\; \begin{cases} true, & \text{if } g(j,w) = m \text{ and } w \text{ is a proper suffix of } t.u; \\ false, & \text{otherwise.} \end{cases}$$

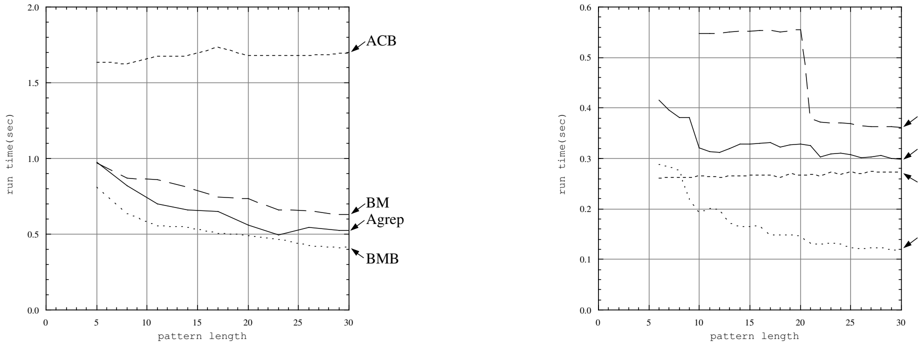
Theorem 2 *The tables Jump and Output can be built in $O(\|\mathcal{D}\| \cdot m)$ time and space.*

The procedure for Step 1 is summarized in Fig. 5.

We use the shift function $\Delta(j,t)$, where j is a state and t is a token, in order to shift the focus on a token to align the text substring $t.u \cdot \pi[m-j+1 : m]$ with its rightmost occurrence in π. The definition is as follows.

$$\Delta(j,t) = \begin{cases} \lceil rightmost\_occ(t.u)/C \rceil, & \text{if } j = 0; \\ \lfloor rightmost\_occ(t.u \cdot \pi[m-j+1 : m])/C \rfloor, & \text{otherwise.} \end{cases}$$

Note that this is a version of the function *shift* stated in Section 3.1. When returning at the first **if-then** statement of the procedure in Fig. 5. we can shift the focus by



(a) Medline (b) GenBank

Figure 6: Running times.

$\Delta(0, \mathcal{S}[\mathit{focus}])$ to the right. Otherwise, we use the function $\Delta(\cdot,\cdot)$ for the values of *state* and $\mathcal{S}[\mathit{focus} - \ell]$ just after the execution of the **while**-loop at the first iteration of the **repeat-until** loop of the procedure.

A simplified version is also possible in which the value $\Delta(0, \mathit{focus})$ is used independently of the result of the procedure of Fig. 5.

Theorem 3 *The table Δ can be built in $O(\|\mathcal{D}\| \cdot m)$ time and space.*

Proof. We can fill the entries of the table in the bottom-up manner by using the directed acyclic word graph [6] for the reversed pattern. □

4 Experimental results

We estimated the running times of the following four programs: `Agrep`; the BM algorithm with shift function due to [20]; the AC type algorithm on BPE compressed files due to [18] (abbreviated as ACB); and the BM type algorithm on BPE compressed files (abbreviated as BMB). The text files we used are: (1) a clinically-oriented subset of **Medline**, consisting of 348,566 references. The original file size is 60.3 Mbyte, and the BPE compression ratio is 59.4% for $C = 3$; and (2) the file obtained by removing all fields other than accession number and nucleotide sequence from a data set from **GenBank**. The original file size is 17.1 Mbyte, and the BPE compression ratio is 32.8% for $C = 4$. The machine used is SunMicrosystems Ultra Enterprise 3000 running Soralis 2.5.1 operating system. The results are shown in Fig. 6, where we excluded the preprocessing times, which are negligible compared with the running times. The proposed algorithm (BMB) is faster than all the others. Especially, it runs about 1.5 times faster than `Agrep` for Medline, and about 3 times faster for GenBank.

5 Conclusion

For searching a very long pattern (e.g., $m > 30$), a simplified version of the backward-dawg-matching algorithm [6] is very fast as reported in [1]. To develop its compressed matching version will be our future work.

References

[1] C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle, suffix oracle. Technical Report IGM-99-08, Institut Gaspard-Monge, 1999.

[2] 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.

[3] R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):62–72, 1977.

[5] M. Crochemore, F. Mignosi, A. Restivo, and S. Salemi. Text compression using antidictionaries. In *Proc. 26th Internationial Colloquim on Automata, Languages and Programming*, pages 261–270. Springer-Verlag, 1999.

[6] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York, 1994.

[7] E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th International Symp. on String Processing and Information Retrieval*, pages 90–95. IEEE Computer Society, 1998.

[8] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

[9] R. N. Horspool. Practical fast searching in strings. *Software-Practice and Experience*, 10:501–506, 1980.

[10] G. C. Jewell. Text compaction for information retrieval. *IEEE SMC Newsletter*, 5, 1976.

[11] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symp. on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

[12] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference '98*, pages 103–112. IEEE Computer Society, 1998.

[13] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1977.

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

[15] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching*, pages 113–124. Springer-Verlag, 1994.

[16] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 14–36. Springer-Verlag, 1999.

[17] W. Rytter. Algorithms on compressed strings and arrays. In *Proc. 26th Ann. Conf. on Current Trends in Theory and Practice of Infomatics*. Springer-Verlag, 1999.

[18] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, , T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. the 4th Italian Conference on Algorithms and Complexity*. Springer-Verlag, 2000. to appear.

[19] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. Pattern matching in text compressed by using antidictionaries. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 37–49. Springer-Verlag, 1999.

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

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

[22] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.

[23] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, October 1992.

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