String Matching with Stopper Encoding and Code Splitting*

Jussi Rautio[1], Jani Tanninen[2], and Jorma Tarhio[1]

[1] Department of Computer Science and Engineering
Helsinki University of Technology
P.O. Box 5400, FIN-02015 HUT, Finland
{jrautio,tarhio}@cs.hut.fi
[2] Department of Computer Science, University of Joensuu
P.O. Box 111, FIN-80101 Joensuu, Finland
jtanni@cs.joensuu.fi

Abstract. We consider exact string searching in compressed texts. We utilize a semi-static compression scheme, where characters of the text are encoded as variable-length sequences of base symbols, each of which is represented by a fixed number of bits. In addition, we split the symbols into two parallel files in order to allow faster access. Our searching algorithm is a modification of the Boyer-Moore-Horspool algorithm. Our approach is practical and enables faster searching of string patterns than earlier character-based compression models and the best Boyer-Moore variants in uncompressed texts.

1 Introduction

The *string matching problem*, which is a common problem in many applications, is defined as follows: given a pattern $P = p_1 \dots p_m$ and a text $T = t_1 \dots t_n$ in an alphabet Σ, find all the occurrences of P in T. Various good solutions [6] have been presented for this problem. The most efficient solutions in practice are based on the Boyer-Moore approach [5].

Recently the compressed matching problem [1] has gained much attention. In this problem, string matching is done in a compressed text without decompressing it. Researchers have proposed several efficient methods [12, 2, 15, 13] based on Huffman coding [9] or the Ziv-Lempel family [20, 21].

One of the most efficient approaches has been developed by Shibata et al. [17]. They present a method called BM-BPE which finds text patterns faster in a compressed text than Agrep [18] finds the same patterns in an uncompressed text. Their search engine is based on the Boyer-Moore algorithm and they employ a restricted version of byte pair encoding (BPE) [7] achieving a saving of 40% in space. BPE replaces recursively the most

* This work has been supported by the National Technology Agency (Tekes).

common character pair by an unused character code. According to their experiments BM-BPE is faster than most of the earlier methods.

Two other works apply the Boyer-Moore approach in compressed texts. Manber [11] presents a non-recursive coding scheme related to BPE. No character can be both the right character of one pair and the left character of another. His method achieves a saving of 30% in space and the search speed is 30% faster than Agrep. Because of special coding, Manber's approach works poorly with short patterns. Moura et al. [12] present a method with a better compression ratio than BPE and with a faster search than Manber, but their search works only with words.

We present a new method, which is faster than the best variation of BM-BPE with a comparable compression ratio. In our method characters are encoded as variable-length sequences of base symbols, where each base symbol is represented by a fixed number of bits. Our coding approach is a generalization of that of Moura et al. [12], where bytes are used as base symbols for coding words. In addition, we split the base symbols into two parallel files in order to allow faster access. Our search algorithm is a variation of the Boyer-Moore-Horspool algorithm [8]. The shift function is based on several base symbols in order to enable longer jumps than the ordinary occurrence heuristic.

We tested our approach with texts of natural language. Besides outperforming BM-BPE, our approach was clearly more efficient than the best Boyer-Moore variants of Hume and Sunday [10] in uncompressed texts for $m > 3$. Our approach is efficient also for short patterns, which are important in practice. For example, our approach is 20% faster than BM-BPE and an efficient Boyer-Moore variant for patterns of four characters.

Our approach is not restricted to exact matching nor the Boyer-Moore algorithm, but it can be applied to string matching problems of other types as well.

2 Stopper Encoding

2.1 Stoppers and Continuers

We apply a semi-static coding scheme called *stopper encoding* for characters, where the codewords are based on frequencies of characters in the text to be compressed. The frequencies of characters are gathered in the first pass of the text before the actual coding in the second pass. Alternatively, fixed frequencies based on the language and the type of the text may be used.

A codeword is a variable-length sequence of *base symbols* which are represented as k bits, where k is a parameter of our scheme. Because the length of a codeword varies, we need a mechanism to recognize where a new one starts. A simple solution is to reserve some of the base symbols as *stoppers* which can only be used as the last base symbol of a codeword. All other base symbols are *continuers* which can be used anywhere but in the end of a codeword. If $u_1 \ldots u_j$ is a codeword, then $u_1, \ldots, u_{j-1}$ are continuers and u_j is a stopper.

Moura et al. [12] use a fixed coding scheme related to our approach. They apply 8-bit base symbols to encode words where one bit is used to describe whether the base symbol is a stopper or a continuer. Thus they have 128 stoppers and 128 continuers.

2.2 Number of Stoppers

It is an optimization problem to choose the number of stoppers to achieve the best compression ratio (the size of the compressed file divided by that of the original file). The optimal number of stoppers depends on the number of different characters and the frequencies of the characters. Let F_i be the frequency of the i^{th} character in the decreasing order according to frequency. When encoding with k-bit base symbols, s stoppers, and $2^k - s$ continuers, the compression ratio C for a fixed division to stoppers and continuers can be calculated with the following formulas, where $L_{s,x}$ is the number of different codewords with x or less base symbols when there are s stoppers.

$$L_{s,x} = \sum_{t=0}^{x-1} s(2^k - s)^t$$

$$Q_i = \frac{kx}{8} F_i, \quad \text{where } x \text{ is the smallest such that } i \leq L_{s,x}.$$

$$C = \frac{\sum_i Q_i}{\sum_i F_i}$$

Let us consider 3-bit base symbols as an example. Table 1 shows how many characters at most can be processed optimally with s stoppers, when the frequency distribution of the characters is uniform or follows Zipf's law.

As another example, let us consider the bible.txt of the Canterbury Corpus [3]. For this text, 14 is the best number of stoppers, when base symbols of four bits are used. Then each of the 14 most common characters

Table 1. *Optimal stopper selection.*

Stoppers	*Uniform*	*Zipf*
7	15	16
6	19	44
5	66	79
4	87	437
3	480	15352

of the text (63.9% of all characters) is encoded with one base symbol of 4 bits and the next 28 characters with two base symbols. In this scheme, 56 of the next characters could be encoded with three base symbols, but there are only 21 of them left (0.6% of all characters.)

Moura et al. [12] use 128 stoppers for 8-bit base symbols. This number is not optimal. More stoppers produce a better compression ratio—the gain is about 5% in the case of the words of the bible.txt. However, the difference is marginal in the case of longer texts.

Perhaps the easiest method of finding the optimal number of stoppers is to calculate first the cumulative frequencies of characters. Let $F'(i)$ be the cumulative frequency of the i^{th} character in the decreasing order of frequency such that $F'(0) = 0$. Then $S_{s,x} = \frac{kx}{8}(F'(L_{s,x}) - F'(L_{s,x-1}))$ is the total coding space for all k-bit base symbols of width x. Then we examine which value of s minimizes the sum $\sum_{x=1}^{x_N} S_{s,x}$ where N is the number of different characters and x_N is the largest x such that $L_{s,x} \leq N$ holds.

2.3 Building the Encoding Table

After the number of stoppers (and with it, the compression ratio) has been decided, an encoding table can be created. The average search time is smaller if the distribution of base symbols is as uniform as possible. We present here a heuristic algorithm, which produces comparable results with an optimal solution in the average case.

The procedure depends on the width of base symbols. We present here the 4-bit version. Let us assume that the characters are ordered in a decreasing order of frequency. If the number of stoppers is s, we allocate the s first base symbols as one-symbol codewords for the s most common characters. The next $s(16 - s)$ characters will have two-symbol codewords, starting with a continuer (the index of the base symbol is $s + (c - s) \bmod (16 - s)$), and ending with a stopper (the index of the base symbol is $(c - s) \operatorname{div} (16 - s)$), where c is the index of the character

in turn. Further symbols are encoded with more continuers and a stopper according to the same idea.

The encoding table is stored with the compressed text. We need $N+2$ bytes to encode the table for N 8-bit characters. One byte is reserved for the number of characters and another byte for the number of stoppers. After these bytes, all the characters present in the text are given in the decreasing order of frequency. With this information, the encoding table can be reconstructed before decompression.

3 Code Splitting

We made an experiment of accessing 100 000 bytes from a long array. For each $g = 0, 1, ..., 7$ we run a test where the bytes were accessed with repetitive gaps of g, i.e. we access bytes $g \cdot i$, $i = 0, 1, ..., 99999$.

Table 2 shows the results of the experiment which was run on a 500 MHz Celeron processor under Linux. The times are relative execution times with a fixed gap width. The main task of the test program was access the array, but it did some additional computation to make the situation more realistic. This extra computation was the same for each run.

Table 2. *An access experiment.*

Gap	0	1	2	3	4	5	6	7
Time	1.00	1.56	2.18	2.70	2.02	2.39	2.55	2.72

According to Table 2, dense accessing is clearly more efficient than sparse accessing, although the total time does not grow monotonously. This dependency is a consequence of the hierarchical organization of memory in modern computers. We tested the same program also with other processors and the results were rather similar. This phenomenon suggests that string matching of the Boyer-Moore type could be made faster by splitting the text to several parallel files.

Combining code splitting with stopper encoding. The splitting of the text could be done in many ways. We apply the following approach. Let the text be represented as k-bit base symbols. We concatenate the h high bits of the base symbols to a file and the l low bits to another file,

$h + l = k$. In practice these files could be still concatenated, but here we consider two separate files for clarity. We call this method *code splitting*.

We denote stopper encoding with the division to h high bits and l low bits by $\mathrm{SE}_{k,h}$. The version without code splitting is denoted by $\mathrm{SE}_{k,0}$. The plain code splitting without compression is denoted by $\mathrm{SE}_{8,h}$. Note that $\mathrm{SE}_{8,h}$ can be seen a representative of stopper encoding: in $\mathrm{SE}_{8,h}$ all the 256 base symbols of eight bits are stoppers.

We consider mainly three versions of stopper encoding: $\mathrm{SE}_{4,0}$, $\mathrm{SE}_{8,4}$, and $\mathrm{SE}_{6,2}$. Note that $\mathrm{SE}_{4,0}$ applies compression, $\mathrm{SE}_{8,4}$ code splitting, and $\mathrm{SE}_{6,2}$ both of them.

4 The Searching Algorithm

The key point of the searching algorithm is that the pattern is encoded in the same way as the text. So we actually search for occurrences of a string of base symbols, or low bits of them, if code splitting is applied. In the latter case, the search in low bits produces only potential matches which all should be checked with high bits.

After finding an occurrence of the encoded pattern, we simply check the base symbol preceding the occurrence. If the base symbol is a stopper (or the occurrence starts the text), we report a match, otherwise we ignore this alignment and move on.

4.1 Searching in an Alphabet of 16 Characters

Let us assume that we have a text with 16 or less different characters. Then all the characters of the text can be represented with four bits, and we can store two consecutive characters in one 8-bit byte.

The basis of our searching algorithm is ufast.fwd.md2, a fast Boyer-More-Horspool variant presented by Hume and Sunday [10]. This algorithm employs an unrolled skip loop and a fixed shift in the case of the match of the last character of the pattern. The shift is based on the text character under the rightmost character of the pattern. It is straightforward to modify ufast.fwd.md2 to our setting.

Because one byte holds two characters, there are two different byte alignments of an occurrence of the pattern. Therefore there are two acceptable bytes which may start the checking phase of the algorithm, corresponding to these two alignments.

The shift is based on a character pair in the terms of the original text. This approach in ordinary string matching has been studied by Baeza-Yates [4] and Zhu and Takaoka [19]. Zhu and Takaoka take the shift as a

minimum of shifts based on match and occurrence heuristics like in the original Boyer-Moore algorithm [5]. However the mere occurrence heuristic is faster in practice for natural language texts.

This searching algorithm works fine with the variant $SE_{4,0}$, where 4-bit base symbols are used and thus the size of alphabet is just 16.

4.2 Searching in $SE_{8,4}$

Recall that no compression is involved with $SE_{8,4}$. All the bytes of the text are split in two parts: the four high bits to one part and the four low bits to the other. These parts are stored in separate files, where new bytes are made from two half-bytes. For example the text "`Finland!`", which is `46-69-6e-6c-61-6e-64-21` in hexadecimal, will have its high bits stored as `46-66-66-62` and the low bits as `69-ec-1e-41`.

Now suppose we wanted to search the pattern `land`, which is `6c-61-6e-64`, in the encoded text. We start by searching the low bits for the corresponding combination of low bits which we will call the encoded pattern, namely `c1-e4`. Then the pattern could start from the beginning of a byte in the low bits (`c1-e4`) or from the middle of a byte (`*c-1e-4*`), where the asterisk represents any hexadecimal digit.

Now we use some method to search the low bits for all occurrences (of both variants) of this encoded pattern. When and only when a match is found in the low bits, the corresponding high bits are checked. So if there are no matches in the low bits, the high bits can be ignored.

An advantage of this method is that only a fraction of the characters of the text are inspected. *False matches* (substrings of the text where the low bits match with the pattern and the high bits do not) are rare in most texts of natural language, so we seldom need to check the high bits at all. Another advantage is that two characters are accessed at a time while scanning the text.

4.3 Searching in $SE_{6,2}$

This 6-bit variant sacrifices space for speed. The ideal compression ratio is 75%, when there are 64 or fewer different characters in the text. Since it is difficult to store sequences of 6-bit base symbols into 8-bit bytes, code splitting is applied. We store four low bits and two high bits separately.

This allows two variations. The first variation goes through the 4-bit part and checks the 2-bit part only when a match in the 4-bit part is found. This is what we will call the 4+2 searching algorithm. The second variation, 2+4, does the same thing vice versa. The 2+4 variation is

generally faster, because it only needs to take $\frac{1}{3}$ of all data into account on the first pass, while the 4+2 takes $\frac{2}{3}$ of it. However, the overhead of having to search 4 patterns simultaneously and inefficiency in the case of patterns of 7 or less characters, also make the 4+2 variation usable on the side of the 2+4 one. The best algorithm is obviously a combination. Based on our experiments, we decided to use the 2+4 variation for $m \geq 8$ and the 4+2 one for $m < 8$.

5 Experimental Results

When Boyer-Moore string searching described above is combined with stopper encoding $\mathrm{SE}_{k,h}$, the total method is denoted by BM-$\mathrm{SE}_{k,h}$.

We tested the performance of the algorithms BM-$\mathrm{SE}_{4,0}$, BM-$\mathrm{SE}_{8,4}$, and BM-$\mathrm{SE}_{6,2}$. Recall that $\mathrm{SE}_{4,0}$ applies compression, $\mathrm{SE}_{8,4}$ code splitting, and $\mathrm{SE}_{6,2}$ both of them. We compared them with four other searching algorithms. We used Tuned Boyer-Moore or ufast.fwd.md2 [10] denoted by TBM as the searching algorithm for uncompressed texts. Three versions of the BM-BPE algorithm (a courtesy from M. Takeda) for compressed texts were tested: one with maximal compression ratio and no upper limit for the number of characters represented by a byte (max), another with optimal search speed where a byte can represent at most two characters (fast), and the third one where a byte can represent at most three characters and which was recommended by the authors (rec). All the algorithms were modified to read first the whole text to the main memory and then to perform the search. All the tests were run on on a 500 MHz Celeron processor with 64 MB main memory under Linux.

The compression ratio was measured with four texts (Table 3): the bible.txt [3], the CIA World Factbook of 1992, Kalevala, the national epic of Finland (in Finnish), and E.coli, the genome of *Escherichia coli*, entirely composed of the four DNA symbols. As explained earlier, there is no compression involved with $\mathrm{SE}_{8,4}$, only a different encoding. The compressed files include the encoding tables which are necessary to uncompress them. As a reference, we give also the compression ratios achieved with Gzip.

To make a fair comparison with BM-BPE, the version BM-$\mathrm{SE}_{4,0}$ is the right choice, because its compression rate is similar to that of the fast BM-BPE.

The compression and decompression algorithms of BM-SE are very fast (17 MB/s) due to the lightweight encoding and decoding schemes.

We tested the search speed with two texts: bible.txt (Table 4) and E.coli (Table 5). We used command-line versions of all the algorithms.

Table 3. Compression ratio.

	bible.txt 3.86 MB	CIA1992 2.36 MB	Kalevala 0.52 MB	E.coli 4.42 MB
BM-BPE max	47.8%	56.8%	51.9%	31.3%
BM-BPE fast	56.2%	63.0%	55.1%	50.0%
BM-SE$_{4,0}$	58.9%	68.2%	58.1%	50.0%
BM-SE$_{6,2}$	75.0%	75.8%	75.1%	75.0%
Gzip	29.4%	29.3%	36.3%	28.9%

We measured the processor time in milliseconds required by the search. Although the excluding of the I/O time slightly favors poorer compression methods, we wanted to measure the efficiency of the pure algorithms without any disturbance due to buffering. The same test was repeated for 500 different strings of the same length randomly chosen in the text.

Table 4. Search times (ms), bible.txt, $3 \leq m \leq 20$.

	3	4	5	6	8	10	12	16	20
TBM	53.4	47.4	42.8	40.4	37.0	35.2	35.0	33.8	31.8
BM-BPE max	68.4	66.2	63.4	61.2	57.6	55.2	53.2	39.4	38.6
BM-BPE rec	71.8	51.2	45.0	44.2	35.0	31.2	30.6	26.8	25.8
BM-BPEfast	52.4	46.4	38.2	36.2	31.0	27.8	26.4	24.2	23.6
BM-SE$_{8,4}$	59.4	38.4	32.0	26.2	22.6	20.0	18.8	17.2	17.0
BM-SE$_{4,0}$	49.0	37.2	31.6	28.0	24.2	22.2	21.0	20.0	19.4
BM-SE$_{6,2}$	66.8	38.0	32.6	26.6	18.8	15.2	13.6	12.0	10.4

In the bible.txt, the versions BM-SE$_{4,0}$ and BM-SE$_{6,2}$ of Boyer-Moore with stopper encoding are clearly faster than BM-BPE for all pattern widths shown in Table 4. However, they are also faster than TBM excluding very short patterns $m < 4$. Even the version without compression, BM-SE$_{8,4}$ is faster than TBM and BM-BPE for $m > 3$. None of the BM-SE algorithms is distinctly the fastest one. BM-SE$_{4,0}$ is the fastest for $m < 6$, BM-SE$_{8,4}$ for $m = 6$, BM-SE$_{6,2}$ for $m > 6$. The times of four algorithms are shown graphically in Figure 1.

The advantage of BM-SE is smaller in the DNA text, because the average length of shift is shorter. According to Table 5, BM-SE$_{4,0}$ is the fastest for short patterns $m \leq 12$ and BM-BPE rec for longer ones. Note that BM-BPE rec is now clearly faster than BM-BPE fast. As one may expect, BM-SE$_{6,2}$ is very poor in the DNA text and so we left it out from this comparison. Probably BM-SE$_{2,0}$ (which has not yet been implemented) will be even better than BM-SE$_{4,0}$ for DNA data.

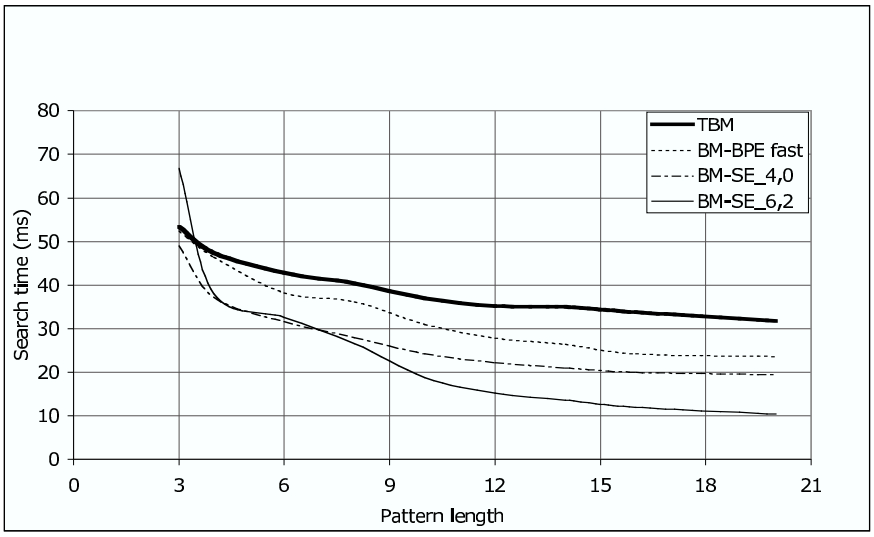

Fig. 1. Search times in bible.txt.

TBM is not a good reference algorithm for DNA matching. BNDM [14] would be more appropriate, because is the fastest known algorithm for patterns $m \leq w$, where w is the number of bits in the computer word. Other alternatives would have been ufast.rev.gd2 [10] or algorithms based on alphabet transformations [4, 16].

Table 5. Search times (ms), E.coli, $6 \leq m \leq 48$.

	6	12	24	48
TBM	67.0	61.2	60.0	60.2
BM-BPE max	52.8	34.2	26.0	23.0
BM-BPE rec	43.2	28.0	22.0	21.0
BM-BPE fast	52.4	36.8	31.4	30.4
BM-SE$_{8,4}$	37.8	27.4	23.6	22.0
BM-SE$_{4,0}$	35.8	26.2	23.0	21.4

6 Concluding Remarks

We have presented a new practical solution for the compressed matching problem. According to our experiments the search speed of our BM-SE

is clearly faster than that of BM-BPE for natural language texts. The version BM-$SE_{4,0}$ has similar compression ratio to the fast BM-BPE. In DNA texts there is no significant difference in the search speed.

Moreover our BM-SE is faster than TBM for patterns longer than three characters.

It would be interesting to compare BM-SE with Manber's method [11], because he reports a gain of 30% in search times. It is clear that this gain is not possible for short patterns because of Manber's pairing scheme. A part of the gain is due to the save in I/O time which was excluded in our measurements.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, 52(2):299–307, 1996.
3. R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proc. DCC '97, Data Compression Conference*. IEEE, 1997.
4. R. Baeza-Yates. Improved string searching. *Software – Practice and Experience*, 19(3):257–271, 1989.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, 1977.
6. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.
7. P. Gage. A new algorithm for data compression. *C/C++ Users Journal*, 12(2), 1994.
8. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.
9. D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.
10. A. Hume and D. Sunday. Fast string searching. *Software – Practice and Experience*, 21(11):1221–1248, 1991.
11. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. on Information Systems*, 15(2):124–136, 1997.
12. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems*, 18(2):113–139, 2000.
13. G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 459–468, 2001.
14. G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5, 2000.
15. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. 11st Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 166–180, 2000.

16. H. Peltola and J. Tarhio. String matching in the DNA alphabet. *Software – Practice and Experience*, 27:851–861, 1997.
17. Y. Shibata, T. Matsumoto, M. Takeda, A. Shiohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11st Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 181–194, 2000.
18. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, Berkeley, CA, USA, 1992.
19. R. Zhu and T. Takaoka. On improving the average case of Boyer-Moore string matching algorithm. *Journal of Information Processing*, 10:173–177, 1987.
20. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
21. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.