Context-dependent Stopper encoding

Jussi Rautio

Laboratory of Information Processing Science
Helsinki University of Technology
Espoo, Finland

e-mail: Jussi.Rautio@hut.fi

Abstract. A character-based encoding method is presented for natural-language texts and genetic data. Exact string matching from the encoded text is faster than from the original text, with medium and longer patterns. A compression ratio of about 50% is achieved as a by-product. The method encodes characters with variable-length *codewords* of 2-bit *base symbols*. An advanced variant is context-dependent, using information from the previous character. The method supersedes the previous comparable methods in compression ratio, and is comparable to the best such methods in search speed.

Keywords: compressed matching, accelerator encoding, Stopper encoding

1 Introduction

As the amount of available information is constantly growing, fast information retrieval is becoming more and more important; it is a key concept in many applications, especially on-line ones. The *string matching problem* is about locating all the *occurrences* of a specific *pattern* from the *full text*. Within this paper, I will concentrate on exact string matching, requiring an exact match with the pattern and the occurrence in the full text.

A common solution to the string matching problem is to build an external *index* with pointers to the full text [15]. With an index, string matching can be done in logarithmic time. The disadvantage of these methods is an increase in space consumption. For static files, it is possible to compress the index only [15], or to compress the index separately from the full text [15]. The FM index [5] applies Burrows-Wheeler transformation [4] to the text before compression, drastically reducing the amount of necessary index data. These methods allow both an excellent compression ratio and fast string matching. However, they do not support on-line updates or approximate matching.

An alternative to indexing, it is possible to encode the full text with a local scheme specificly designed to improve search speed. A Boyer-Moore [2, 9] type string matching algorithm can be used with the encoded text, improving search speed by a constant factor. Some of these schemes even offer a significant compression ratio as a by-product. In the absence of a common term for this class of schemes, I will use the term *accelerator encoding* for all such schemes.

Although accelerator encoding falls behind compressed indexing in both compression ratio and search speed, it allows on-line updates and on-line decoding. It is suited for documents which are queried, retrieved or updated often, for example text databases or log files.

There are two types of accelerator encoding schemes: word-based and character-based. Word-based schemes [3, 12] work with whole words at a time, allowing a better compression ratio for large files but requiring a large dictionary. Their use is limited to natural-language texts where words are separated by delimiters (unlike Japanese, for example), and string matching is possible only with whole words and combinations of subsequent words. Character-based schemes [6, 13, 14] work with a fixed number of characters at a time. String matching is possible with a more free range of patterns, possibly including errors and classes of characters.

I will present a novel character-based accelerator encoding scheme and an exact string matching algorithm which works with it. Variants of the new scheme, *flexible stopper encoding*, can be used either with genetic data or with natural-language texts. The scheme is based on encoding each character of the text with a *codeword* of one or more 2-bit *base symbols*. With pure DNA code (with the alphabet `acgt`), exactly one base symbol is used for each base pair, leading to a trivial encoding. With natural-language texts, the compression ratio of the scheme is optimized with methods including context dependence of the first order. String matching from the encoded text is done with an algorithm resembling Tuned Boyer-Moore.

Compared with existing character-based schemes, the new scheme can be useful. For previous character-based schemes, the compression ratio (size of compressed file divided by size of original file) of natural-language texts was about 50% for the slowest methods and 60% for the faster ones. For my example texts, compression ratio of the new scheme is $0.3 - 2$ percentage units better than the best previous schemes, and the search speed is comparable to the fastest previous methods.

2 Background

Let $T[0, n-1]$ denote a text over the alphabet Σ. An encoding is a transformation from the text $T[0, n-1]$ to the *encoded text* $T'[0, n'-1]$, in the alphabet Σ'. String matching in an encoded text means locating all occurrences $L(P)$ of a given *pattern* $P[0, m-1]$ from the original text, with only the encoded text available. Throughout this paper, I assume that it is sufficient to locate all occurrences of the encoded pattern $L'(P')$ in the encoded text.

Accelerator encoding methods use either static or semi-static encoding. In the latter case, a small dictionary containing all necessary information for matching and decoding is saved along with the encoded text. Dynamic dictionary methods cannot be used, because the dictionary cannot be kept up to date without reading every character of the text, which would be disastrous to the performance of the algorithm.

An important property of any compression algorithm is *compression ratio*, here denoted as the size of the encoded text divided by the size of the original text, the smaller the better. For the sake of uniformity, a character of the original text is always calculated as one 8-bit byte, even with genetic data.

Byte-pair encoding (BPE) exploited the variable frequencies of consecutive character pairs. For BPE, $\Sigma' = \Sigma$. T' is a copy of T, except that the most common pairs of consecutive characters are replaced with characters of Σ with no occurrences in T. The Manber variant [11] limits possible pairs, sacrificing compression ratio for search speed. The original variant [6] does not have this limitation, allowing a better compression ratio. Both variants have an efficient Boyer-Moore type string matching algorithm. A partially recursive version of the compression algorithm was recommended for the original variant [14], where one character in Σ' can represent one to three characters in Σ. Another variant of Byte-pair encoding called Repair [10] is even more optimized to compression ratio, but lacks an efficient search algorithm.

Our earlier comparison of these two variants [13] suggested that the Manber variant supported faster exact string matching, however its compression ratio was only 70-75% with natural-language texts. For the original variant, and especially its recursive version, the compression ratio could be as good as 45% with the same texts. However, the better the compression ratio, the slower the string matching. The scheme with the best compression ratio only allowed a slower string matching than with the original text.

Stopper encoding [13] is related to an earlier word-based method by de Moura et al. [12]. I will describe here only the 4-bit variant $\mathrm{SE}_4, 0$. The basic unit of the encoded text was the *base symbol*. It consisted of four bits, $\Sigma' = [0, 15]$. When encoding, every character of T was replaced with a corresponding *codeword*, a sequence of one or more base symbols. No codeword could be a prefix of another. More common characters were given shorter codewords than less common ones. This resembled Huffman coding. [8]

To allow faster string matching, base symbols were divided into two classes called *stoppers* and *continuers*. Let s denote the number of stoppers, such as all c in Σ' less than s are stoppers. A legal codeword $C'[0, r-1]$ consisted of zero or more symbols of the continuer class, followed by exactly one symbol of the stopper class, that is: $C'[i] \geq s$ holds for all $i < r-1$, and $C'[r-1] < s$. This made it possible to recognize codewords when starting at an arbitary location in the text, including after jumps made by a Boyer-Moore type algorithm.

The 4-bit encoded text T' was stored into the 8-bit form, two base symbols per a 8-bit computer byte, so that it could be used efficiently. String matching in the encoded text was done with a Boyer-Moore type algorithm called BM-SE, which handles whole bytes instead of individual base symbols. The algorithm operates by encoding pattern P and then locating the occurrences of the encoded pattern P' from T'. Naturally, the possible occurrences were not restricted to byte boundaries, but could start or end at either the first or the second base symbol in the byte. Because of this, two possible *alignments* of the encoded pattern must be produced by using a shift operation. The actual search algorithm was a multi-pattern version of Tuned Boyer-Moore [2, 9]. It only made one pass of the encoded text, trying to locate both of the alignments at the same time. When a presumed match was found, the preceding base symbol was checked. If it is a stopper, the match was confirmed, otherwise it was discarded.

Word-based methods resembling Stopper encoding have been used to encode whole words at a time. In schemes by de Moura et al. [12] and Brisaboa et al. [3], whole words were encoded at a time. Each was given a representation of one to three base symbols, in this case 8-bit bytes. These base symbols were divided into the continuer and stopper classes. This algorithm produced an excellent compression ratio with natural language (currently the best one seen in accelerator encoding). De Moura's scheme used a fixed number of stoppers (128), while Brisaboa allowed free determination of the number. Search speed was only discussed by de Moura. Both schemes had the same disadvantages. They allowed matching with whole words only and could not support approximate matching. In addition, the size of the required dictionary was large compared to other methods in the field.

Our earlier comparison between Byte-pair encoding and Stopper encoding [13] suggests that Stopper encoding is superior in search speed (probably partially because of implementation issues) and that some variants of Byte-pair encoding provide a better compression ratio.

3 New solution

The new solution, *flexible stopper encoding*, is an extension to stopper encoding [13]. Stopper encoding used 4- or 6-bit base symbols depending on the variant, which had theoretical limits for the compression ratio at 50%, and 75%, respectively. Flexible stopper encoding uses 2-bit base symbols, and its theoretical limit for compression ratio is 25%. Some limitations of stopper encoding are relaxed to achieve an efficient compression ratio for this scheme. I will start by describing the basic method, and continue by discussing improvements one at a time.

Pure DNA data (of the symbols `acgt` only) can be encoded with a trivial encoding. The alphabet has only four different characters, so let us denote $\Sigma' = \{0, 1, 2, 3\}$. This means 2-bit base symbols, four of which can be stored in a 8-bit byte. This encoding gives an exact compression ratio of 25%.

Stopper encoding from the previous section can also be introduced to 2-bit base symbols. A constant s, $0 < s < 4$ is determined, dividing the encoded alphabet into two classes: stoppers and continuers. According to the definition in the previous section, for a valid codeword of length r, denoted by $C'[0, r-1]$, for all $i < r-1$ holds $C'[i] \geq s$, and $C'[r-1] < s$. The more common characters are represented by shorter codewords than the less common ones.

Note that only three legal values exist for s. 0 is impossible since no stoppers means that codewords would never end, and 4 is only valid when there are four or fewer characters in the alphabet. The best compression ratio is usually achieved with $s = 2$, but generally this scheme is too strict and must be relaxed.

Flexibility introduced to the previous scheme produces Flexible stopper encoding (FSE). The base symbols are divided into two classes as before, but the definition of the classes is changed. Stoppers function as before, but continuers are replaced with flexers, base symbols that may act either as stoppers or continuers, depending on their position in the codeword. Usually flexers act as continuers near the beginning of a codeword, and as stoppers after that.

More formally, assign values to s_i, $0 < s_i < 4$, for all reasonable i. Now, a valid codeword has exactly such base symbols that $C'[i] \geq s_i$ holds for all $i < r-1$, and $C'[r-1] < s_{r-1}$. Consequently, $s = \min s_i$.

Presumed matches preceded by flexer can be confirmed by locating the first stopper symbol preceding the presumed occurrence, and decoding after that until the identity of the flexer can be confirmed.

Context dependence allows a better compression ratio than possible if all characters are encoded separately. The context-dependent variant is called Context-dependent FSE, or CFSE. The meanings of codewords change according to the meanings of their preceding characters. This only applies to codeword allocation, not their structure. Context dependence may be implemented in conjunction with flexibility, or independently from it.

To allow on-line locating and decoding, delimiters (spaces) are fixed always to have the same encoding. It could be possible to choose any character, but the space is chosen because it appears regularly and the most often.

A separate *successor table* S_c is constructed for each different character c occurring in the text. $S_c[0]$ is fixed to the space character, and $S_c[i]$ is the i:th common non-space successor of c. In addition, a *codeword table* is constructed, containing the $|\Sigma|$ shortest valid codewords sorted by increasing length. When encoding a character $T[s]$, its index i is located from the successor table such as $S_{T[s-1]}[i] = T[s]$, and the i:th codeword from the codeword table is put in the output stream.

Encoding and decoding algorithms are straight-forward to implement. To encode, the entire text is first scanned to count relative frequencies of characters. Then, the base symbol configuration (number of stoppers s_n) is decided, and the codeword table built. Another pass of the text is required to encode the characters one by one. Finally, the save file is built, including the base symbol configuration, the successor table, and the encoded text.

The optimal number of stoppers s_i can be calculated with an exhaustive search for small values of i. After preliminary tests, I decided to test all value combinations for all $i < 4$, and to set $s_i = s_4$ for all $i \geq 4$. The best general values for s_i for natural-language texts seem to be $1, 3, 3, \ldots$. With the context-dependent variant, all preliminary tests with natural-language texts seemed to work almost optimally with the values $s_i = 2, 3, 3, \ldots$, so this value set is automatically used with this variant.

The successor table takes $O(n^2)$ space. The list of all characters in the text is saved first. Then, for each character, its successors are saved in descending order of frequency. This takes about $4k$ space with 64 different characters in the text. Improvements are possible. The data structure used by the non-context-dependent variant is the list of characters ordered by frequency.

Decoding is done by building either a single decoding tree (non-context-dependent variant) or a separate decoding tree for each preceding character. This works exactly the same way as Huffman [8] decoding.

String matching means locating an occurrence of the pattern in the text. With FSE, it is sufficient to locate an occurrence of the encoded pattern in the encoded text, preceded by a stopper symbol. In the context-dependent variant, the first character varies according to the preceding one, but its successors do not vary and are used for the search.

The exact string matching algorithm BM-CFSE is developed from the 2-bit exact string matching algorithm used with stopper encoding, BM-SE$_{6,2}$, which was in turn influenced by Tuned Boyer-Moore [9]. The algorithm is basically a multi-pattern version of Tuned Boyer-Moore, locating all four possible alignments of the encoded pattern P' in a single pass through the text. It consists of a *preprocessing phase* and a *search phase*. The search phase alternates between a *fast loop*, which quickly weeds out most locations, and a more precise *slow loop*, which is used to confirm presumed matches found by the fast loop.

The search algorithm needs two data structures to work. The slow loop uses a *multi-mask table* S, resembling the mask table of the shift-or algorithm [1]. The fast loop uses a *jump table* D constructed from the multi-mask table, resembling the occurrence heuristic jump table from Boyer-Moore type algorithms.

To construct the multi-mask table, some definitions are required. Let P' be the encoded pattern, and P'_0, P'_1, P'_2, and P'_3 its alignments (in any order). The alignments are filled with *wild card* symbols where no base symbol is available (before the beginning or after the end of the encoded pattern). Each character in the encoded text c' consists of four base symbols c'_0, c'_1, c'_2, and c'_3. The encoded characters c' and d' are said to *unify* if and only if for all a, either c'_a and d'_a are equal, or one of them is a wild card symbol $*$.

The multi-mask table is constructed with a simple rule. Let l' be such that for all i, $P'_i[l]$ is the last full character (one not containing any wild card symbols) of P'_i. Now, $S[c,i]_a = 1$ if and only if $P'_a[i]$ unifies with c, and 0 otherwise. The value of the multi-mask is now $S[c,i] = S[c,i]_0 + 2S[c,i]_1 + 4S[c,i]_2 + 8S[c,i]_3$.

Algorithm 1 Constructing the multi-mask table S

```
fill $S$ with 0
q ← {1,2,4,8}
**for** $c \leftarrow 0$ to 256, $i \leftarrow 0$ to $m$, $a \leftarrow 0$ to 4 **do**
  **if** P'$_a$[i] unifies with $c$ **then**
    $S[c,i] \leftarrow S[c,i] + q[a]$
```

When the multi-mask table has been constructed, making the jump table is a trivial matter. The l'th encoded character of the pattern is always the last full encoded character of each alignment. For other encoded characters in the pattern at the location i, the possible jump length is $l - i$. The jump table construction and the fast loop are direct adaptations from Tuned Boyer-Moore. After preliminary tests, I decided to use triple loop unrolling as recommended by Hume and Sunday. A md_2 step-after-match heuristic can also be used instead of direct incrementation.

Algorithm 2 Constructing the jump table D

```
fill $D$ with $l$
**for** $i \leftarrow 0$ to $l$, $c \leftarrow 0$ to 256 **do**
  **if** $S[c,i] \neq 0$ **then**
    $D[S[c,i]] \leftarrow l - i$
```

The slow loop of the actual search algorithm works as a mask automaton, recognizing all 4 patterns at a time. Starting from the suspectedly first encoded character of the pattern and a state variable q positive for all masks, a bitwise-or operation is repeatedly applied to the state for each character. When the state variable reaches zero, all chances of an occurrence are lost and the fast loop can be resumed. If having gone through all the characters in the suspected pattern the state variable still has one or more positive bits, the match can be confirmed by locating a stopper symbol immediately preceding the suspected pattern.

Algorithm 3 Search algorithm: text scan phase

```
copy pattern $P'$ to end of text $T[n], T[n+1], \ldots$
$s \leftarrow l$
for ever do
  $k \leftarrow D[T[s]]$
  while $k \neq 0$ do
    $s \leftarrow s + k$
    $k \leftarrow D[T[s]]$
  $i \leftarrow 0; q \leftarrow 15$
  while $i < l$ and $q \neq 0$ do
    $q \leftarrow q$ bitwise-or $S[T[s - l + 1 + i], i]$
    $i \leftarrow i + 1$
  if $q \neq 0$ then
    if $s = n$ then
      end
    else
      confirm and report occurrence(s)
  $s \leftarrow s + 1$
```

4 Experiments

The most important properties of accelerator encoding algorithms are search speed and compression ratio, in that order. Compression and decompression times are reported in the final version.

In the experiments, FSE and CFSE are pitted against the leading uncompressed and compressed matching algorithms. As reference algorithms, I have my earlier implementations of $\mathrm{SE}_4, 0$ and the 6-bit Stopper encoding $\mathrm{SE}_6, 2$, Tuned Boyer-Moore by courtesy of Hume and Sunday, and BM-BPE by courtesy of Takeda. BM-BPE comes in three versions, *fast* limiting maximum compression to two original characters per encoded character, *rec* (recommended) limiting it to three, and *max* being without limitation.

I use the Canterbury Corpus version of the King James Bible for test data. I run two separate tests with separate sets of patterns. In the first test, all patterns are whole words or beginnings of words, including the space before the beginning. Using them is a common scenario, and CFSE can search them faster than other patterns. In the second test, the patterns are unrestricted. Experiments with genetic data will be included in the final version.

All experiments are run on a 650 MHz AMD Athlon machine with 384 megabytes of main memory, running Debian Linux in single-user mode. All the programs are compiled using `gcc` with maximum optimization (flag `-O6`).

In the experiment, command-line versions of all test programs, all of them performing exactly one search per execution of program, are run several times. The programs measure their own execution time by inserting calls to the C function `clock()` into the code. This clocked time includes everything except program argument parsing and reading the file from disk.

The compression ratios are shown in Table 1. CFSE provides a better compression ratio than any of the other algorithms in all these examples. Differences between it

	KJV Bible (3.86M)
BPE$_{\text{max}}$	47.8%
BPE$_{\text{rec}}$	51.0%
BPE$_{\text{fast}}$	56.2%
SE$_{4,0}$	58.9%
FSE	55.6%
CFSE	47.5%

Table 1: Compression ratios.

and the maximal-compression version of BPE are 0.3 − 2.1 percentage units. However, it provides an over 10 percentage units better compression ratio than the generally fastest of the other algorithms, the 4-bit Stopper Encoding.

Table 2 describes the search speed from the Bible with whole words or word beginnings, and Table 3 repeats the same test with freely chosen patterns. The performance of BM-CFSE is about the same as that of BM-SE$_{4,0}$, being somewhat faster with longer patterns and somewhat slower with shorter patterns. However, it is about twice faster than the algorithms which offer a similar compression ratio, BM-BPE$_{\text{rec}}$ and BM-BPE$_{\text{fast}}$. With pattern length 5 in Table 3, the poor performance of BM-CFSE is probably because of an implementation anomaly. CFSE is minimally better with whole-word patterns than with free ones.

5 Conclusions

I have presented new accelerator encoding schemes called Flexible stopper encoding FSE and the context-dependent version CFSE, and an exact string matching algorithm for them, called BM-FSE. The new schemes produce a better compression ratio than any of the the existing accelerator encoding methods for the example natural-language text. The string matching algorithm is comparable to the fastest existing methods with both uncompressed and compressed texts.

With pure genetic data, FSE reduces to a trivial encoding with a compression ratio of exactly 25%. Compression and decompression are straight-forward operations, and mapping from the encoded text to the original is trivial. FSE can be used to store

pattern length	3	4	5	6	8	12	20
TBM	99	116	131	142	159	173	193
BM-BPE$_{\text{max}}$	61	63	66	68	73	81	122
BM-BPE$_{\text{rec}}$	56	90	95	97	128	155	212
BM-BPE$_{\text{fast}}$	80	84	110	113	138	177	226
BM-SE$_{4,0}$	112	152	177	203	241	301	330
BM-SE$_{6,2}$	95	160	166	219	281	398	566
BM-FSE	83	123	159	184	228	289	352
BM-CFSE	100	136	165	190	246	322	399

Table 2: Search speed for KJV Bible (word beginnings only) in kB/ms.

pattern length	5	6	8	12	20
TBM	143	148	165	189	213
BM-BPE$_{\text{max}}$	67	69	73	81	120
BM-BPE$_{\text{rec}}$	136	138	165	214	293
BM-BPE$_{\text{fast}}$	111	115	138	169	214
BM-SE$_{4,0}$	186	220	247	358	361
BM-SE$_{6,2}$	184	213	273	435	794
BM-FSE	161	196	227	309	385
BM-CFSE	67*	139	223	307	355

Table 3: Search speed for KJV Bible (free patterns) in kB/ms.

large files of pure genetic data for efficient retrieval.

With natural-language texts, CFSE is efficient because of its good compression ratio. Its worst limit is that it relies on frequent occurrences of delimiters in the text. Unlike word-based accelerator compression schemes, CFSE still allows exact string matching with any pattern, and requires a smaller dictionary.

CFSE's advantage over BPE in compression ratio comes from the fact that BPE divides text into units encoded separately from one another. CFSE, however, always encodes according to the previous character.

In search speed, BM-CFSE is similar to BM-SE. There seems to be no fundamental difference between 4-bit base symbols and 2-bit ones. BM-CFSE benefits from its compression ratio and suffers from the omission of the first character from the fast loop.

The earlier accelerator encoding schemes had trade-offs, being either good in compression ratio and bad in speed, (BPE$_{\text{max}}$), or the other way round (SE$_{4,0}$). It can be noted that CFSE has no such trade-off, having both a superior compression ratio and an excellent search speed. The inclusion of disk read times favors it even more. The only exception is searching with short patterns (less than 5 characters), where SE$_{4,0}$ is better.

A better compression ratio could be obtained by introducing a higher order context dependence. However, there would be problems with dictionary size, and for each pattern, two first characters would become unstable instead of one, further reducing search speed. Another interesting question is how well approximate string matching could be performed with stopper encoding or CFSE.

References

[1] Baeza-Yates, R., Gonnet, G., *A new approach to text searching*, Communications of the ACM, 35(10):74–82, 1992.

[2] Boyer, R. and Moore, J. *A fast string searching algorithm.* Communications of the ACM, 20(10):762–772, 1977.

[3] Brisaboa, N., Fariña A., Navarro, G., and Esteller, M. *(S,C)-Dense Coding: An Optimized Compression Code for Natural Language Text Databases.* Proceedings of the SPIRE conference, pages 122-136, 2003.

[4] Burrows, M. and Wheeler, D. *A block-sorting lossless data compression algorithm.* DEC SRC Research Report 124, 1994.

[5] Ferragina, P. and Manzini, G. *An experimental study of an opportunistic index.* Proceedings of the 12th ACM-SIAM Symposium of Discrete Algorithms (SODA), 2001.

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

[7] Golomb, S. *Run-length encoding.* IEEE Transactions on Information Theory, 12(3), 1966.

[8] Huffman, D. *A method for the construction of minimum-redundancy codes.* Proceedings of the IRE 40, 1098-1101. David Applegate et al, 1952.

[9] Hume, A. and Sunday, S. *Fast string searching.* Software Practice and Experience, 21:1221–1248, 1991.

[10] Larsson, N., Moffat, A. *Offline dictionary-based compression.* Proc. IEEE, 88(11), 1722–1732, 2000.

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

[12] de Moura, E., Navarro, G., Ziviani, N. and Baeza-Yeates, R. *Fast and flexible word searching on compressed text.* ACM Transactions on Information Systems, 18(2):113–139, 2000.

[13] Rautio, J., Tanninen, J., and Tarhio, J. *String matching with stopper encoding and code splitting.* Proc. CPM '02, Combinatorial Pattern Matching (ed. A. Apostolico, M. Takeda), Lecture Notes in Computer Science 2373, Springer, 2002, 42-52.

[14] Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A. and Arikawa, S. *A Boyer-Moore type algorithm for compressed pattern matching.* Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (LNCS 1848), pages 181–194. Springer-Verlag, 2000.

[15] Witten, I., Moffat, A., Bell, T. *Managing gigabytes.* Morgan Kaufmann Publishers, Academic Press, 1999.

[16] Ziv, J. and Lempel, A. *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23:337–343, 1977.