UNIVERSITY OF JOENSUU

DEPARTMENT OF COMPUTER SCIENCE AND STATISTICS

Report Series A

Compressed representation of sequences with constant time random access

Kimmo Fredriksson and Fedor Nikitin

A-2006-3

ACM E.1, E.2, E.4, F.2, H.3.1

ISBN 952-458-873-0

ISSN 0789-7316

October 17, 2006

Compressed representation of sequences with constant time random access

Kimmo Fredriksson* and Fedor Nikitin
Department of Computer Science
University of Joensuu
{kfredrik,fnikitin}@cs.joensuu.fi

Abstract

Given a sequence S of n symbols over some alphabet Σ, we develop a new compression method that is (i) extremely simple to implement; (ii) gives good compression ratio; (iii) provides $O(1)$ time random access to any symbol of the original sequence. Our simplest solution uses at most $2h + o(h)$ bits of space, where $h = n(H_0(S) + 1)$, and $H_0(S)$ is the zeroth-order empirical entropy of S. This can be improved to take only $h + h' + o(n) + O(\log\log(h))$ bits of space, where $h' = h(-\frac{n}{h}\log_2(\frac{n}{h}) - \frac{h-n}{h}\log_2(\frac{h-n}{h}))$. $H_0(S)$ can be replaced by the k-order empirical entropy $H_k(S)$ for constant size alphabets, for $k = o(\log_\sigma(n))$. We discuss a number of improvements and trade-offs (we obtain e.g. $n(H_0(S)+2\sqrt{H_0(S)}+1)+o(n(\sqrt{H_0(S)}+1))$ bits) and give potential applications for the method.

Keywords: algorithms, compression, succinct sequences, self-delimiting integers

ACM Classification: E.1 [Data structures]; E.2 [Data storage representations]; E.4 [Coding and information theory] — *Data compaction and compression*; F.2 [Analysis of algorithms and problem complexity]; H.3.1 [Information storage and retrieval]: Content Analysis and Indexing — *Dictionaries, Indexing methods*

1 Introduction

In this paper we sketch a new compression method for sequences. The main traits of the method are its extreme simplicity, good compression ratio and that it provides constant time random access to any symbol of the original sequence. This work was inspired by [6, 14]. Our work is in a very preliminary phase, we plan to extend and improve it in a number of ways, and to provide experimental results.

2 Preliminaries

Let $S[0 \ldots n-1] = s_0, s_1, s_2, \ldots, s_{n-1}$ be a sequence of symbols over an alphabet Σ of size $\sigma = |\Sigma|$. For a binary sequence $B[0 \ldots n-1]$ the function $\texttt{rank}_b(B, i)$ returns the number of times the bit b occurs in $B[0 \ldots i]$. Function $\texttt{select}_b(B, i)$ is the inverse, i.e. it gives the index of the ith bit that has value b. Note that for binary sequences $\texttt{rank}_0(B, i) = i + 1 - \texttt{rank}_1(B, i)$. Both `rank` and `select` can be computed in $O(1)$ time with only $o(n)$ bits of space in addition to the original sequence taking n bits [8, 10]. It is also possible to achieve $nH_0(B) + o(n)$ total space, where $H_0(B)$ is the zero-order entropy of B [12, 13], while retaining the $O(1)$ query times.

The zeroth-order empirical entropy of the sequence S is defined to be

$$H_0(S) = -\sum_{s \in \Sigma} \frac{f(s)}{n} \log_2 \left(\frac{f(s)}{n} \right), \tag{1}$$

*Supported by the Academy of Finland, grant 207022.

Table 1: Coding scheme

symbol	codeword
s_{i_0}	0
s_{i_1}	1
s_{i_2}	00
s_{i_3}	01
s_{i_4}	10
s_{i_5}	11
s_{i_6}	000
s_{i_7}	001
s_{i_8}	010
s_{i_9}	011
$s_{i_{10}}$	100
$s_{i_{11}}$	101
$s_{i_{12}}$	110
$s_{i_{13}}$	111
$s_{i_{14}}$	0000
...	...

where $f(s)$ denotes the number of times s appears in S.

The k-th order empirical entropy is

$$H_k(S) = -\sum_{i=k+1}^{n} p_i \log_2(p_i), \tag{2}$$

where $p_i = \text{Probability}(s_i \mid s_{i-k}, \ldots, s_{i-1})$. In other words, the symbol probabilities depend on the context they appear on, i.e. on which are the previous k symbols in S. Obviously, $H_k(S) \leq H_0(S)$.

3 Simple compression scheme

Our compression scheme first computes the frequencies of each alphabet symbol appearing in S. Assume that the symbol $s_i \in \Sigma$ occurs $f(s_i)$ times. The symbols are then sorted by their frequency, so that the most frequent symbol comes first. Let this list be $s_{i_0}, s_{i_1}, \ldots, s_{i_{\sigma-1}}$, i.e. $i_0 \ldots i_{\sigma-1}$ is a permutation of $\{0, \ldots, \sigma - 1\}$.

The coding scheme assigns binary codes with different lengths for the symbols as follows. We assign 0 for s_{i_0} and 1 for s_{i_1}. Then we use all binary codes of length 2. In that way the symbols $s_{i_2}, s_{i_3}, s_{i_4}, s_{i_5}$ get the codes $00, 01, 10, 11$, correspondingly. When all the codes with length 2 are exhausted we again increase length by 1 and assign codes of length 3 for the next symbols and so on until all symbols in the alphabet get their codes, see Table 1.

Theorem 1 *For the proposed coding scheme the following holds:*

1. *The binary code for the symbol $s_{i_j} \in \Sigma$ is of length $\lfloor \log_2(j+2) \rfloor$.*
2. *The code for the symbol $s_{i_j} \in \Sigma$ is binary representation of the number $j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}$ of $\lfloor \log_2(j+2) \rfloor$ bits.*

Proof. Let a_ℓ and b_ℓ be indices of the first and the last symbol in alphabet Σ, which have the binary codes of length ℓ. We have $a_1 = 0$ and $b_1 = 1$. The values a_ℓ and b_ℓ for $\ell > 1$ can be

defined by recurrent formulas

$$a_\ell = b_{\ell-1} + 1, \quad b_\ell = a_\ell + 2^\ell - 1. \tag{3}$$

In order to get the values a_ℓ and b_ℓ as functions of ℓ, we first substitute the first formula in (3) to the second one and have

$$b_\ell = b_{\ell-1} + 2^\ell. \tag{4}$$

By applying the above formula many times we have a series

$$\begin{aligned} b_\ell &= b_{\ell-2} + 2^{\ell-1} + 2^\ell, \\ b_\ell &= b_{\ell-3} + 2^{\ell-2} + 2^{\ell-1} + 2^\ell, \\ &\dots \\ b_\ell &= b_1 + 2^2 + 2^3 + \ldots + 2^\ell. \end{aligned}$$

Finally, b_ℓ as a function of ℓ becomes

$$b_\ell = 1 + \sum_{k=2}^{\ell} 2^k = \sum_{k=0}^{\ell} 2^k - 2 = 2^{\ell+1} - 3. \tag{5}$$

Using (3) we get

$$a_\ell = 2^\ell - 3 + 1 = 2^\ell - 2. \tag{6}$$

If j is given the length of the code for s_{i_j} is defined equal to ℓ, satisfying

$$a_\ell \le j \le b_\ell. \tag{7}$$

According to above explicit formulas for a_ℓ and b_ℓ we have

$$2^\ell - 2 \le j \le 2^{\ell+1} - 3 \quad \Longleftrightarrow \quad 2^\ell \le j + 2 \le 2^{\ell+1} - 1, \tag{8}$$

and finally

$$\ell \le \log_2(j+2) \le \log_2(2^{\ell+1} - 1), \tag{9}$$

whose solution is easily seen to be $\ell = \lfloor \log_2(j+2) \rfloor$.

For the setting the second statement it is sufficient to observe that the code for the symbol $s_j \in \Sigma$ is $j - a_\ell$. By applying simple transformations we have

$$j - a_\ell = j - (2^\ell - 2) = j + 2 - 2^\ell = j + 2 - 2^{\lfloor \log_2(j+2) \rfloor}.$$

So, the second statement is also proved. □

The whole sequence is then compressed just by concatenating the codewords for each of the symbols of the original sequence. We denote the compressed binary sequence as $S' = S'[0 \dots h-1]$, where h is the number of bits in the sequence. Fig. 1 illustrates.

3.1 Constant time random access to the compressed sequence

The seemingly fatal problem of the above approach is that the codes are not *prefix codes*, and we have not used any delimiting method to mark the codeword boundaries, and hence the original sequence would be impossible to obtain. However, we also create an auxiliary binary sequence $D[0 \dots h-1]$, where h is the length of S' in bits. $D[i] = 1$ iff $S'[i]$ starts a new codeword, and 0 otherwise, see Fig. 1. We also need a symbol table T, such that for each different codeword length we have table of the possible codewords of the corresponding length. In other words, we have a table $T[0 \dots \lfloor \log_2(\sigma+1) \rfloor - 1]$, such that table $T[i][0 \dots 2^{i+1} - 1]$ lists the codewords of length i. Then, given a bit-string r, $T[|r|-1][r]$ gives the decoded symbol for codeword r. This

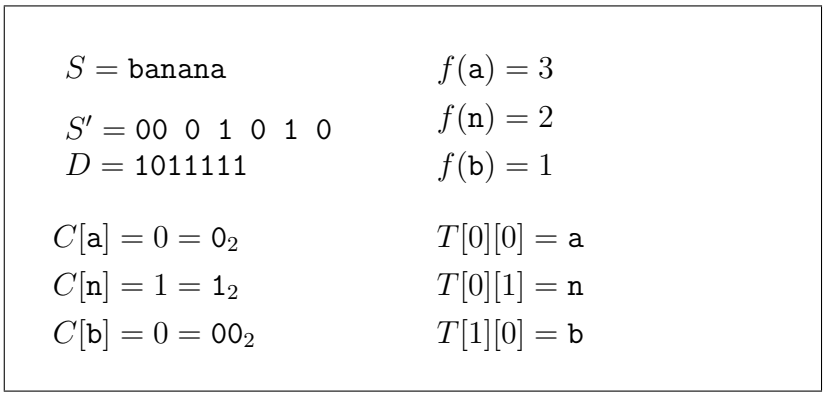

Figure 1: Example of compressing the string `banana`.

information is enough for decoding. However, D also gives us *random access to any codeword of* S'. That is, the ith codeword of S' starts at the bit position $\texttt{select}_1(D, i)$, and ends at the position $\texttt{select}_1(D, i+1) - 1$. This in turn allows to access any symbol of the original sequence S in constant time. The bit-string

$$r = S'[\texttt{select}_1(D, i) \ldots \texttt{select}_1(D, i+1) - 1] \tag{10}$$

gives us the codeword for the ith symbol, and hence

$$S[i] \;=\; T[|r| - 1][r], \tag{11}$$

where $|r|$ is the length of the bitstring r. Note that in RAM model of computation $|r| = O(\log(n))$, and hence r can be extracted in $O(1)$ time.

3.2 Space complexity

The number of bits required by S' is

$$h = \sum_{j=0}^{\sigma-1} f(s_{i_j}) \; \lfloor \log_2(j+2) \rfloor, \tag{12}$$

and hence the average number of bits per symbol is h/n.

Theorem 2 *The number of bits required by S' is at most $n(H_0(S) + 1)$.*

Proof. The zero-order empirical entropy of S is

$$-\sum_{j=0}^{\sigma-1} \frac{f(s_{i_j})}{n} \log_2 \left(\frac{f(s_{i_j})}{n} \right), \tag{13}$$

and thus

$$n(H_0(S) + 1) = n \sum_{j=0}^{\sigma-1} \frac{f(s_{i_j})}{n} \log_2 \left(\frac{n}{f(s_{i_j})} \right) + n = \sum_{j=0}^{\sigma-1} f(s_{i_j}) \left(\log_2 \frac{n}{f(s_{i_j})} + 1 \right). \tag{14}$$

We will show that the inequality

$$\lfloor \log_2(j+2) \rfloor \leq \log_2(j+2) \leq \left(\log_2 \left(\frac{n}{f(s_{i_j})} \right) + 1 \right) = \log_2 \left(\frac{2n}{f(s_{i_j})} \right) \tag{15}$$

holds for every j, which is the same as

$$j+2 \leq \frac{2n}{f(s_{i_j})} \iff (j+2)f(s_{i_j}) \leq 2n. \tag{16}$$

Note that for $j=0$ the maximum value for $f(s_{i_j})$ is $n-\sigma+1$, and hence the inequality holds for $j=0,\ \sigma \geq 2$. In general, we have that $f(s_{i_{j+1}}) \leq f(s_{i_j})$, so the maximum value for $f(s_{i_1})$ is $n/2$, since otherwise it would be larger than $f(s_{i_0})$, a contradiction. In general $f(s_{i_j}) \leq n/(j+1)$, and the inequality becomes

$$(j+2)f(s_{i_j}) \leq 2n \iff (j+2)n/(j+1) \leq 2n \iff (j+2)/(j+1) \leq 2, \tag{17}$$

which holds always. □

In general, our coding cannot achieve $H_0(S)$ bits per symbol, since we cannot represent fractional bits (as in arithmetic coding). However, if the distribution of the source symbols is not very skewed, it is possible that $h/n < H_0(S)$. This does not violate the information theoretic lower bound, since in addition to S' we need also the bit sequence D, taking another h bits. Therefore the total space we need is $2h$ bits, which is at most $2n(H_0(S)+1)$ bits. However, this can be improved.

Note that we do not actually need D, but only a data structure that can answer $\texttt{select}_1(D,i)$ queries in $O(1)$ time. This is possible using just $h' = hH_0(D)+o(n)+O(\log\log(h))$ bits of space [13]. Therefore the total space we need is only $h+h'$ bits. $H_0(D)$ is easy to compute as we know that D has exactly n bits set to 1, and $h-n$ bits to 0. Hence

$$H_0(D) \;=\; -\frac{n}{h}\log_2\left(\frac{n}{h}\right) \;-\; \frac{h-n}{h}\log_2\left(\frac{h-n}{h}\right). \tag{18}$$

Note that $\frac{n}{h} \approx \frac{1}{H_0(S)+1}$ and $\frac{h-n}{h} \approx 1-\frac{1}{H_0(S)+1}$. This means that $H_0(D)$ is maximized when $\frac{n}{h}=\frac{1}{2}$, but on the other hand h' depends also on h. Thus, h'/h shrinks as h grows, and hence for increasing $H_0(S)$ (or for non-compressible sequences, in the worst case $H_0(S)=\log_2(\sigma)$) the contribution of $hH_0(D)$ to the total size becomes more and more negligible.

Finally, the space for the symbol table T is $\sigma\lceil\log_2(\sigma)\rceil$ bits, totally negligible in most applications. However, see Sec. 3.5 and Sec. 3.6 for examples of large alphabets.

3.3 Trade-offs between h and h'

So far we have used the minimum possible number of bits for the codewords. Consider now that we round each of the codeword lengths up to the next integers divisible by some constant u, i.e. the lengths are of the form $i \times u$, for $i = \{1,2,\ldots,\lceil\log_2(\sigma)\rceil/u\}$. So far we have used $u=1$. Using $u>1$ obviously only increases the length of S', the compressed sequence. But the benefit is that each of the codewords in S' can start only at positions of the form $j \times u$, for $j=\{0,1,2,\ldots\}$. This has two consequences:

1. the bit sequence D need to store only every uth bit;
2. every removed bit is a 0 bit.

The item (2) means that the probability of 1-bit occurring increases to $\frac{n}{h/u}$. The extreme case of $u=\log_2(\sigma)$ turns D into a vector of n 1-bits, effectively making it (and S') useless. However, if we do not compress D, then the parameter u allows easy optimization of the total space required. Notice that when using $u>1$, the codeword lenght becomes

$$\lfloor\log_2((2^u-1)j+2^u)\rfloor_u \leq \log_2((2^u-1)j+2^u) \tag{19}$$

bits, where $\lfloor x\rfloor_u = \lfloor x/u\rfloor u$. Then we have the following:

Theorem 3 *The number of bits required by S' is at most $n(H_0(S)+u)$.*

Proof. The theorem is easily proved by following the steps of the proof of Theorem 2. □

The space required by D is then at most

$$\frac{n(H_0(S)+u)}{u}. \tag{20}$$

bits. Summing up, the total space is optimized for $u=\sqrt{H_0(S)}$, which leads to total space of

$$n\left(H_0(S)+2\sqrt{H_0(S)}+1\right)+o\left(n\left(\sqrt{H_0(S)}+1\right)\right) \tag{21}$$

bits, where the last term is for the $\texttt{select}_1$ data structure [10].

Note that $u=7$ would correspond to byte based End Tagged Dense Code (ETDC) [1] if we do not compress D. By compressing D our space is smaller and we also achieve random access to any codeword, see Sec. 3.5.

3.4 Context based modelling

We note that we could easily use context based modelling to obtain h of the form $nH_k(S)$. The only problem is that for large alphabets k must be quite small, since the symbol table size is multiplied by σ^k. This can be controlled by using k that depends on S. For example, using $k=\frac{1}{2}\log_\sigma(n)$ the space complexity is multiplied by $\sqrt{n}$, negligible for constant size alphabets.

3.5 Word alphabets

Our method can be used to obtain very good compression ratios for natural language texts by using the σ distinct words of the text as the alphabet. By Heaps' Law [7], $\sigma=n^\alpha$, where n is the total number of words in the text, and α is language dependent constant, for English $\alpha=0.4\ldots0.6$. These words form a dictionary $T[0\ldots\sigma-1]$ of σ strings, sorted by their frequency. The compression algorithm then codes the jth most frequent word as an integer j using $\lfloor\log_2(j+2)\rfloor$ bits. Again, the bit-vector D provides random access to any word of the original text.

As already mentioned, using $u=7$ corresponds ETDC method [1]. ETDC uses 7 bits in each 8 bit byte to encode the codewords. The last bit is saved for a flag that indicates whether the current byte is the last byte of the codeword. Our benefit is that as we store these flag bits into a separate vector D, we can compress D as well, and simultaneously obtain random access to the original text words.

We quickly estimated the size of the compressed text using our method. We used two dictionaries, one for the text words and the other for "separators", where separator is defined to be any substring between two words [9]. As there is strictly alternating order between the two, decompressing is easy as far as we know whether the text starts with a word or a separator. We used a 58,482,621 characters long text (a concatenation of various Project Gutenberg[1] files). The number of words and separators was $n=10,529,843$ for both, while the number of distinct words was $\sigma=169,847$, and the number of distinct separators was $\lambda=9,790$. We build separate S' and D vectors for words and separators. Our method gives compression ratio of 33.1%, including the two dictionaries (compressed using `zlib` library[2]) and an estimate of size of the data structures for $\texttt{select}_1$ queries for words ($h_WH_0(D_W)+n$ bits, where the subscript W refers to words). For separators we store only the (compressed) D vector ($h_SH_0(D_S)$ bits, where the subscript S refers to separators), since we are not interesting in random access to them, but

[1] `http://www.gutenberg.org/`

[2] `www.zlib.org`

store them only for decompression purposes. Using `select`$_1$ data structures for separators as well, increases the space to about 36.5% (additional n bits). If we are not interesting in random access, but just want to compress the text (and hence `select`$_1$ is not needed at all), we obtain about 30.8%. Using `gzip -9` (a variant of LZ compression [15]) gives 37.5%, slightly more than ours. However, using `bzip2 -9` (a variant of block-sorting compression [2]) gives about 28.5%. The zeroth-order empirical entropy of the text using the model of words and separators is about 6.36 bits per word/separator, while our method gives about 6.85 bits per word/separator (without `select`$_1$ structures).

Note that the so called "spaceless" model should give somewhat better compression results [9] for our method too, especially for $u > 1$. Using ASCII alphabet instead of word alphabets gives compression ratio of about 74.8%, including the `select`$_1$ data structure.

3.6 Self-delimiting integers

Assume that S is a sequence of integers in range $\{0, \ldots, \sigma - 1\}$. Note that our compression scheme can be directly applied to represent S succinctly, even without assigning the codewords based on the frequencies of the integers. In fact, we can just directly encode the number $S[i]$ with $\lfloor \log_2(S[i]+2) \rfloor$ bits, and again using the auxiliary sequence D to mark the starting positions of the codewords. This approach does not need any symbol tables, so the space requirement does not depend on σ. Still, if σ and the entropy of the sequence is small, we can resort to codewords based on the symbol frequencies.

This method can be used to replace e.g. Elias δ-coding [3], which achieves

$$\lfloor \log_2(x) \rfloor + 2\lfloor \log_2(1 + \lfloor \log_2(x) \rfloor) \rfloor + 1 \tag{22}$$

bits to code an integer x. Elias codes are self-delimiting prefix codes, so the sequence can be uniquely decompressed. However, Elias codes do not provide constant time random access to the ith integer of the sequence.

Again, we can use u to tune the space complexity of our method.

4 Concluding remarks

We have presented a simple compression scheme that allows constant time access to any symbol of the original sequence. The main benefit over previous methods is that the new representation is extremely simple.

Note that basically our compression scheme can be directly used in place of the arithmetic coder used in [6], and our D table (or actually the `select`$_1$ data structures built from it) replace their tables R_g and R_l. The difference that remains is that they retrieve any substring of length $b = \lfloor \log_\sigma(n) \rfloor$ in constant time, while we have shown only how to handle the case $b = 1$. However, extending our method to $b = \lfloor \log_\sigma(n) \rfloor$ is simple and straight-forward.

Our method could be used as a building-block in Burrows-Wheeler transform [2] -based compressed self-index [4, 11] to the text using word alphabets. This needs also `rank` queries for word alphabets. Using [5], `rank` takes $H_0(S) + O(n)$ space and $O(\log\log(n))$ time, as the alphabet size is of the form $\sigma = O(n^\alpha)$, for a constant $0 < \alpha < 1$.

We are currently working on extending our method in several ways.

References

[1] N. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proceedings of the 25th European Conference on Information Retrieval Research (ECIR'03)*, LNCS 2633, pages 468–481, 2003.

[2] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[3] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[4] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science (FOCS 2000)*, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

[5] A. Golynski, I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm (SODA'06)*, pages 368–373. ACM Press, 2006.

[6] R. González and G. Navarro. Statistical encoding of succinct data structures. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM 2006)*, LNCS 4009, pages 295–306, 2006.

[7] H. S. Heaps. *Information retrieval: theoretical and computational aspects.* Academic Press, New York, NY, 1978.

[8] G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of Foundations of Computer Science (FOCS 1989)*, pages 549–554. IEEE Computer Society Press, 1989.

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

[10] J. Ian Munro. Tables. In *Proceedings of Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, LNCS 1180, pages 37–42. Springer, 1996.

[11] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 2006. To appear.

[12] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP 1999)*, pages 595–604. Springer-Verlag, 1999.

[13] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 233–242. SIAM, 2002.

[14] K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239. ACM Press, 2006.

[15] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.