Multiple-Dictionary Compression Using Partial Matching

Dzung T. Hoang* Philip M. Long† Jeffrey Scott Vitter‡

Department of Computer Science
Duke University
Box 90129
Durham, NC 27708–0129

Abstract

Motivated by the desire to find text compressors that compress better than existing dictionary methods, but run faster than PPM implementations, we describe methods for text compression using multiple dictionaries, one for each context of preceeding characters, where the contexts have varying lengths. The context to be used is determined using an escape mechanism similar to that of PPM methods. We describe modifications of three popular dictionary coders along these lines and experiments evaluating their efficacy using the text files in the Calgary corpus. Our results suggest that modifying LZ77 along these lines yields an improvement in compression of about 4%, that modifying LZFG yields a compression improvement of about 8%, and that modifying LZW in this manner yields an average improvement on the order of 12%.

1 Introduction

Text compression methods are usually divided into two categories: statistical methods, in which previously coded characters are used to estimate probabilities for the character to be coded, and these probabilities are used in conjunction with arithmetic coding; and dictionary methods, in which sections of text are replaced with pointers into a dictionary. The PPM (prediction by partial matching) methods [1,3] are an example of the former, and the Ziv-Lempel methods [4,13,14,15,16] are examples of the latter. Statistical coders typically give better compression while dictionary coders run faster. One successful approach to making PPM methods run faster—by a factor

*Affiliated with Brown University. Supported in part by an NSF Graduate Fellowship and by Air Force Office of Strategic Research grants F49620–92–J–0515 and F49620–94–1–0217.

†Supported in part by Air Force Office of Strategic Research grants F49620–92–J–0515 and F49620–94–1–0217.

‡Supported in part by Air Force Office of Scientific Research grants F49620–92–J–0515 and F49620–94–1–0217 and by a Universities Space Research Association/CESDIS associate membership.

of about 2—is the PPM method developed in [7]. In this paper we describe a different approach to developing methods that give better compression than LZ methods while running faster than PPM and report on experiments evaluating the new methods.

At a high level, for one setting of its parameters, the PPMC algorithm works approximately as follows. For each sequence σ of three characters, the algorithm maintains statistics concerning the number of times each character c occurred when the previous three characters were σ. The same is done for each sequence of two characters, and one character.[1] When the encoder is at a particular position in the file encoding a given character c, it looks at the previous three characters (called the *context*), and encodes the current character using arithmetic coding with a probability estimate given by the fraction of time c has occurred given that the context was the previous three characters. If c has not occurred previously in the given three character context, the encoder sends a special escape character, and recurses to using a two character context.

It has been shown [2,9], that some of the Ziv-Lempel algorithms are, in a sense, equivalent to corresponding statistical coders. Loosely speaking, a statistical method of this sort uses a varying number of previous characters in order to calculate its estimated probability distribution for the next character in a manner similar to the PPM method, except that counts used for probability estimates are updated less frequently. It cycles, first using a context of 0 previous characters, then 1 and continuing for a varying amount of time, typically to around 7 characters but sometimes longer, then returns to a context of 0 characters. The proofs of these results lead to the intuition that a similar phenomenon exists with regard to the other Ziv-Lempel algorithms.

The zero-order entropy of typical English text is known to be quite high, and therefore we cannot get a good estimate for the probability of the next character without using a context. Even using a context of one character has been shown not to be enough to give good compression. Thus, it appears that some of the degradation in compression exhibited by the dictionary coders is due to what corresponds to the characters encoded using contexts of 0, 1 and possibly 2 characters.

When a statistical coder corresponding to some dictionary coder is making its probability estimates according to the highest-order contexts, often these (conditional) probability estimates are based on very small samples. Thus, another possible source for the relatively poor compression of the dictionary coders is in what corresponds to the dubious probability estimates obtained using high-order, say order 6 and higher, contexts. This effect is ameliorated by the escape mechanism in the PPM methods.

Recent work has attempted to improve the compression of dictionary-based methods without slowing them down too much by having separate dictionaries for each previous character [5,10]. The idea in both is, loosely, to have a corresponding equivalent statistical algorithm start its cycle by using a context of 1 previous character, then 2, and so on, instead of starting at 0. The work of [5] placed a heavy emphasis on speed, and therefore it is unfair to compare our compression results with theirs. The compression results of the CSD algorithm [10], which was based in the LZW [14] dictionary method, are given in Table 1 for comparison.

The currently best known statistical method, the PPMD method, usually bases its probability estimates on contexts of three characters. In this paper, we investigate

[1] Actually, these statistics are only sometimes updated.

the use of separate dictionaries for each context up to some maximum context order. We describe implementations of coders of this type behaving, for each context, in a manner analogous to a basic LZ77 [15] coder, and in a manner analogous to the LZFG [4] variant of the basic Ziv-Lempel coder, and the LZW [14] coder. In order to address the second possible problem with the dictionary methods—the use of very long contexts, which result in the use of insufficient statistics—the first two implementations use the following simple idea: A context of a given length, say 2, is not used until is has been seen a given number, say 50, times. Since the decoder has this information, no escape must be sent. This has an indirect "trickle down" effect concerning what corresponds to the use of very high order contexts, resulting in improved compression. We found this tool not to be useful in the LZW-based coder.

We describe experiments using the Calgary corpus which show that, for a simple instantiation of the LZ77 ideas, based on the A1 coder from [4], context-length management of this type leads to an improvement in compression of about 4%. For LZFG [4], the improvement is around 8%, and for LZW, the improvement is around 12%. While our implementations are not yet tuned for speed, analysis suggests that the use of variable-length contexts will not dramatically slow down any of the dictionary coders, as discussed in the following section.

2 The dictionary PM coders

In this section, we describe the multiple-dictionary coders studied in this paper.

2.1 Dictionary partial matching

At a high level, all of the new methods of this paper, which we call dictionary partial matching methods, or DPM for short, work as follows. The coder maintains a list of contexts, which are sequences of characters, of length 0 up to some upper limit, typically about 2. For each context in the list, the coder maintains a dictionary, which is a list of strings. To encode a character in a given position in a file, the coder first looks for the longest context in its collection that is the same as the characters immediately before the position to be coded: call it σ. It then looks for the longest string in σ's dictionary which is the same as the string obtaining by reading forward in the file, starting at the position to be coded. If this "match" has length greater than zero, the identity of this element of σ's dictionary is transmitted somehow, otherwise a special escape symbol is transmitted, and the process is repeated with the next longest context preceding the position to be coded. If even the length-0 context fails to find a string of length greater than 0, a final escape is sent, and the character in the position to be coded is sent literally, that is, as it appears in the file. If the identity of a string in some dictionary is sent, the position to be coded is of course advanced to the first position in the file past those "matching" the given string. If a literal is coded, the position is moved up one. In either case, certain strings are added to dictionaries matching various contexts. A variant of this idea which we have examined is for a context only to be considered for use at a given position when that context has occured at least a certain number of times earlier in the file. If a context fails such a test, no escape needs to be sent, since the decoder can perform the same test.

The differences in the coders of this paper lie mainly in which dictionary method is used: the choice of a dictionary method involves choosing which strings are added to the dictionary and how often, how dictionary elements and escapes are encoded, and what data structures are used for maintaining the dictionaries. The coder described in Section 2.2 makes these decisions in a manner analogous to the A1 instantiation [4] of the LZ77 method [15]. The coder of Section 2.3 uses as a subroutine a dictionary coder based on Slyz's implementation [12] of the C2 coder [4]. The coder of Section 2.4 is based on LZW [14].

2.2 LZ77-PM

We begin by describing a PM modification of a simple LZ77 coder that is almost identical to the A1 coder from [4].

The baseline coder to which we compare our results works as follows. First, it divides the file into buffers of size 256K, and codes each buffer as if it were a separate file. While coding a particular buffer, the dictionary at any given time consists of all strings preceding the position to be coded that are of length at least 2 and at most 16. The encoder then finds the longest string in the dictionary that is identical to the string of the same length obtained by reading forward in the buffer starting at the position to be coded. If p is the buffer position being coded (counting from 0), the encoder tells the decoder which string "matched" by giving the position earlier in the buffer where the match occurred along with the length of the match. The position is encoded with a flat code, using $\lceil \log_2 p \rceil$ bits. The length is encoded in binary using four bits. If there is no matching string in the dictionary, the remaining 4-bit codeword (the first fifteen were used to encode lengths 2 through 16) is transmitted, and the character at position p is coded literally, i.e., as it appears in the file.

Our PM modification of the coder is the same, except that a length 1 context, together with a PPM-style escape mechanism, is used. As in the baseline coder, the file is coded in 256K buffers. Within a buffer, for each character c a dictionary containing all strings from length 2 to length 16 that occur immediately following occurrences of c is maintained. When coding at a particular position in the buffer, if the number n of times the previous character occurs in the buffer is at least 70, the encoder finds the longest entry in the dictionary that "matches" the characters starting with the position to be coded. If a dictionary entry is found, the starting position from among the n possible starting positions for the match is transmitted using a flat code, with $\lceil \log_2 n \rceil$ bits, and the length (between 2 and 16) of the match is transmitted using four bits. If there is no match, the coder uses the remaining 4-bit code to inform the decoder of this fact, and it recurses to behaving like the baseline coder, i.e., to using a length-0 context. If the previous character has occurred fewer than 70 times, the encoder immediately recurses to behaving like the baseline coder. As discussed in Section 2.1, no "escape" must be sent in this case, since the decoder can make this test as well.

While our implementation of the LZ77-PM coder is not tuned for speed, analysis suggests that it should be nearly as fast, if not faster, than a corresponding implementation of the baseline coder. The amount of time spent coding any position is easily seen to be at most roughly twice that of a baseline coder using similar data structures. Further, for fairly long files, where time is an issue, most of the time there

is a match using the order-1 context. For example, on the `book1` file from the Calgary Corpus, 91% of the positions coded resulted in matches using the order-1 context. This includes those positions coded using order-0 because a given order-1 context had occurred less than 70 times previously. The use of a smaller threshold would make this proportion greater with a small (about 1%) degradation in compression. If an implementation like *gzip* is used, most of the time is spent looking for a match, and the fact that LZ77-PM's context length-1 dictionary is smaller than the order-0 dictionary, coupled with the fact that roughly 90% of the time LZ77-PM does only one search, suggests that its time requirements should be competitive.

The results for the baseline coder and the PM modification on the 10 text files from the Calgary Corpus are given in Table 1.

The A1 coder from [4], instead of restarting from time to time, restricts pointers earlier into the file to be to positions at most 64K from the position to be coded. The distance to the position where the match begins is then transmitted using a flat code. To implement a PM generalization of such a "sliding window" algorithm, we must maintain counts, for each context, of the number of times that context occured in the previous 64K characters, which results in some overhead. If we do this, a window of 64K characters is, loosely speaking, not long enough for the high order statistics to build up enough for the use of contexts to help significantly. (A similar effect, without the use of escapes, was reported in [10,11].) Using a sliding window of size 256K improves the compression of both the LZ77 and LZ77PM coders, and the relative performance is about the same: the LZ77PM coder is about 4% better on average over the test files. We report the results for the "restarting" PM coder because it has greater potential to be useful in practice.

2.3 LZFG-PM

In the LZ77 coder, for a given match, several positions in the buffer may contain the same match. Therefore, LZ77 coders reserve codespace for pointers that will never be coded, which is wasteful. The C2 coder of Fiala and Green [4] (referred to in [1] as LZFG) eliminates pointer redundancy by coding a pointer into a trie, which stores unique parsed substrings that have occurred previously in the text. Each substring stored in the trie has a unique identifier. The improvement in compression with LZFG comes at a price of a more complex decoder, which has to maintain the same trie as the encoder.

LZFG uses a space-efficient implementation of a trie called the PATRICIA trie [8]. A PATRICIA trie is a trie in which sequences of unary branches are compressed into a single branch. A PATRICIA trie saves space by not storing every character explicitly in the nodes. Instead, each node stores a position/length pointer (p, ℓ) to a string, which is stored externally in a buffer. For a leaf node, the corresponding string extends indefinitely; i.e., $\ell = \infty$. We can view the arcs between nodes in a PATRICIA trie as being labeled with strings such that the concatenation of the labels on the path from the root to a node gives the string pointed to by that node. Each internal node stores a string previously matched by the coder. In searching the PATRICIA trie, a match may terminate along an arc. The effective dictionary therefore consists of all strings (within length constraints) in the buffer that start at positions where previous matches began.

LZFG identifies an element s of the dictionary encoded by its PATRICIA trie as follows. First, it informs the decoder, using one bit, whether the first node *following* the position in the tree corresponding to s is a leaf or an internal node. If it is an internal node, the identity of the "following" node is transmitted using a flat code,[2] and the position of the string in the arc leading to the given node is also transmitted using a flat code. In Slyz's implementation, leaves are ordered in an (approximate) LRU queue, and if the "following" node is a leaf, its position in this queue is coded using another static variable-length code. Unlike internal nodes, the substrings represented by a leaf can extend indefinitely. The number of characters in the extension that matches the parsed phrase is coded using yet another static variable-length code. All the static variable-length codes used belong to a parameterized class called start-step-stop codes which are quickly encoded and decoded.

We constructed a PM coder which we call LZFG-PM based on an existing LZFG implementation [12]. Following the outline of Section 2.1, LZFG-PM constructs separate tries for each context, including a context of order 0. A limited number of internal nodes and leaves is globally allocated. Nodes and leaves are ordered in LRU lists, with new nodes and leaves replacing old ones when their allotted number has been reached.

Coding of literal characters are handled differently in LZFG-PM than in LZFG. Whereas LZFG groups consecutive literals together and sends them in one chunk, LZFG-PM codes each literal individually. The reasoning is that since LZFG-PM allows pointers to code phrases of length 1 efficiently, literals are less likely to occur consecutively than in LZFG.

Escapes can be coded in several ways. We implemented two methods. In one method, we code an escape as a special node, namely the root node. The second method treats the escape symbol as a special entry in the LRU list of leaves: an escape is coded just like a leaf, using the same self-organizing-list heuristic. We find that coding escapes as a node gives better compression than coding them as a leaf.

We use a heuristic to determine the appropriate context order to begin parsing the next phrase. If the previous context order is C and the length of the previous phrase is L, we start parsing with a context of order $L + C - 1$. Starting at an order higher than $L + C - 1$ would not likely be fruitful since that would mean repeating a search that has failed in the past. This heuristic saves us in the cost of coding escapes.

A new dictionary entry is added to a context when an escape from that context is coded. A new entry is also added to the context used for coding. Contexts of order less than the coding context could also be updated. In our experiments we find that updating lower-order contexts results in marginal improvement (about 0.01 bits/char) in compression at the expense of more overhead. Therefore we adopt the strategy of updating only the coding context and the contexts that are escaped from.

Experimental results for the base LZFG and LZFG-PM coders are given in Table 1. For both coders, the buffer size is 44K. Both coders limit the total number of internal nodes to 16K and the number of leaves to 32K. This allows for practically the best compression with the given buffer size. We experimentally determined the best settings for the start-step-stop codes based on a subset of the test files. We also adjusted the maximum context order and obtained the best results for a maximum context order of one. The tabulated results, given in Table 1, are for a maximum

[2] Actually a nearly flat code [1] is used if the number of internal nodes is not a power of 2.

context order of one, coding escapes as nodes.

2.4 LZW-PM

In the LZW algorithm [14], which is used in UNIX *compress*, the dictionary is initialized to contain all strings of length 1, and a flat code is used to encode the dictionary entry that is matched. After the match, a single string consisting of the matched string concatenated with the first unmatched character is added to the dictionary. In the *compress* implementation of LZW, the size of the dictionary is restricted. In the baseline coder to which we compare our results in this section, this restriction is removed. Also, instead of initializing the dictionary to contain all one character strings, the dictionary is initialized to contain only the empty string. When the longest match is the empty string, the next character in the file is encoded literally.

In the LZW-PM method evaluated in our experiments, we maintain a dictionary like the LZW dictionary for each previously seen context of length at most 2. An important difference is that the dictionary is initially empty (or can be thought of as containing only the empty string, as discussed below), in order to make use of the escape mechanism. The maximum context length of 2 was experimentally found to yield the best compression. After an escape, the string consisting only of the next character is added to the dictionary corresponding to the context escaped from. After a match, the string consisting of the match plus the next character is added to the dictionary corresponding to the matched context. No strings are added to lower-order contexts. Escapes are handled by having each dictionary have a special "escape" element. This can be thought of as having a dictionary element corresponding to the empty string, which is always matched. As in LZW, the identity of the longest dictionary element matched is transmitted using a flat code: if the number of dictionary elements, including the empty string which corresponds to an escape, is d, then $\lceil \log_2 d \rceil$ bits are used. This method of encoding escapes approximately corresponds to the method used in PPMA [1], since it (approximately) involves the implicit assumption that the probability of seeing a previously unseen character in a given context is the inverse of the number of times the context has been seen. While other probability models for escapes performed better as part of PPM (see [1,6]) and are therefore likely to yield better compression here, we chose the method above for simplicity and computational efficiency.

The compression results given by applying LZW-PM to the Calgary corpus, together with those of straight LZW, and the CSD coder [10], which adapts LZW by having a separate context for each single previous character, with no escapes to an order-0 context, are given in Table 1. Analysis of the untimed method suggests that its tuned speed should be roughly comparable to the baseline coder; certainly less than a factor of 2 slower. First, while early in the file, multiple dictionaries must be updated, later almost all second-order contexts will result in a match. For example, while coding `book1` from the Calgary corpus, this happens 95% of the time. The time required for a match is roughly the same per character regardless of the length of the match, and the time to insert a string in the dictionary after the match, if a trie is used to represent the dictionary, is a small constant. Since we expect the length of matches in LZW-PM to be a couple of characters shorter than in LZW, there will be more dictionary additions, resulting in some extra time. Also, searching

for the context to be used requires some time not used by LZW, although it should be significantly less than the time required for the subsequent match, since matches are usually somewhat longer than two characters.

The reason we compared our encoder to the baseline coder described above, instead of *compress*, was to be sure that the improvement obtained was due to the addition of the PM ideas, rather than just due to using larger dictionaries.

3 Conclusions and ongoing work

In this paper, we have described a new approach to dictionary-based text compression, and we have demonstrated that it results in improved compression. The improvement is particularly significant when used with LZW, which is the fastest of the dictionary methods. We expect that the use of the dictionary PM approach also yields practically fast coders. The fact that this approach succeeded in conjunction with the three coders studied in this paper suggests that the high-level specification of this approach can be combined with any dictionary method, including those that might be developed in the future.

Why did the use of contexts improve compression significantly more in conjunction with some dictionary methods than with others? One plausible explanation is as follows. The LZ77 coder is the most liberal about adding entries to its dictionary, followed by LZFG and LZW. Having a larger dictionary results in longer matches, but requires more bits per match. As discussed in the introduction, intuition from the correspondence between (non-PM) dictionary methods and statistical methods suggests that, as a match continues, the probability estimates used in the corresponding encoding of individual characters tend to get better for a while, then get worse. The effect of the use of a context is, loosely speaking, to "eat up" the early characters with relatively poor probability estimates, thereby increasing the average quality of the remaining probability estimates.

A method that liberally updates its dictionary, all else being equal, makes longer matches at the expense of requiring more bits to identify which match was made. By tending to have long matches, such a method has a relatively greater share of its coding inefficiency due to the later characters in its matches. For such a method, removing the use of low-order contexts will not yield much of an improvement. Further, even if, for example, using a context length 2 to encode a single character tends to be worse than using context length 3, it is possible that the dictionaries corresponding to context length 2 tend to be better than the dictionaries for context length 3. To see why this might be true, consider what happens in the corresponding statistical method when we use context length 2 rather than context length 3. Loosely speaking, instead of cycling from contexts of length 3 up to some large number, we cycle from 2 through some large number. Adding the order 2 to the cycle will result in improved compression if, loosely speaking, using order-2 contexts is better than the average of the use of contexts of order 3 and higher.

A method that is very stingy about adding elements to its dictionary, like LZW, has a greater share of its inefficiency due to the early characters in a match, and therefore we would expect it to be helped more by the addition of contexts, as is consistent with our experiments.

file	LZ77	LZ77-PM	LZFG	LZFG-PM	LZW	CSD	LZW-PM
bib	2.85	2.68	2.69	2.44	3.34	3.22	2.82
book1	3.44	3.32	3.57	3.26	3.27	3.59	2.91
book2	2.98	2.84	3.05	2.78	3.14	3.37	2.70
news	3.51	3.43	3.42	3.26	3.75	4.13	3.42
paper1	3.24	3.14	2.94	2.75	3.76	3.43	3.35
paper2	3.20	3.05	3.02	2.81	3.51	3.28	3.18
progc	3.28	3.24	2.89	2.74	3.85	3.42	3.45
progl	2.44	2.30	1.95	1.80	3.03	2.70	2.54
progp	2.42	2.32	1.91	1.82	3.10	2.66	2.58
trans	2.31	2.11	1.79	1.67	3.26	2.91	2.48
total bits/char	3.18	3.06	3.15	2.91	3.34	3.51	2.93
(bits/char)/file	2.97	2.84	2.72	2.53	3.40	3.27	2.94

Table 1: Experimental results on the ten text files in the Calgary corpus. The LZ77 coder is almost identical to the A1 coder from [4]. The LZFG implementation used is from [12]. The total bits per character is obtained taking the total number of bits in the compressed representations of the files in the corpus and dividing by the total number of characters. The bits per character per file is calculated as the average of the bits per character over the test files. CSD results are copied from [10]. CSD used a limited dictionary size (16K), while the LZW and LZW-PM algorithms did not; the adverse affect of this is most notable on the long files.

We are presently working on improving our results in a number of ways. First, we are working on a variant of the LZFG-PM coder that, instead of using separate tries for each context, uses one trie where the dictionary for a particular context is stored in a subtree of the global trie. One advantage of this approach is that more strings can be stored given the same global allocation of nodes and leaves, because storage is shared among contexts with the same prefix. In this approach, adding a string to a given context's dictionary results in strings effectively being added to the dictionaries of contexts that are prefixes of the given context. This results in faster growth of dictionaries, but some dictionary additions may not be as well justified as in the separate-trie coder discussed elsewhere in this paper. Since in the one-trie coder, a particular node or leaf might represent entries in several dictionaries, the use of one trie presents implementation difficulties with the numbering of nodes and leaves.

Second, we are in the process of tuning our PM implementations for speed, which would enable us to make more complete comparisons with existing dictionary techniques. Finally, using the fact the higher-order contexts were not matched and that earlier matches were not longer, the decoder can sometimes infer that certain entries in a given dictionary will not be used. The encoder can then use fewer bits to transmit which dictionary entry was chosen. Taking advantage of these possibilities in a computationally efficient manner appears challenging.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[2] T. C. Bell and I. H. Witten, "The relationship between greedy parsing and symbolwise text compression," *Journal of the ACM* (In press).

[3] J. G. Cleary and I. H. Witten, "Data Compression using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communication* 32 (April 1984), 396–402.

[4] E.R. Fiala and D.H. Greene, "Data compression with finite windows," *Communications of the ACM* 32, 490–505.

[5] P.C. Gutmann and T.C. Bell, "A hybrid approach to data compression," *Proceedings of the 1994 Data Compression Conference* (1994), 225–233.

[6] P. G. Howard and J. S. Vitter, "Practical Implementations of Arithmetic Coding," in *Images and Text Compression*, Kluwer Academic Publishers, 1992, invited paper.

[7] P. G. Howard and J. S. Vitter, "Design and Analysis of Fast Text Compression Based on Quasi-Arithmetic Coding," *Proc. 1993 IEEE Data Compression Conference* (March–April 1993), 98–107.

[8] D. Knuth, in *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[9] G. G. Langdon, "A note on the Ziv-Lempel model for compressing individual sequences," *IEEE Transactions on Information Theory* 29 (March 1983), 284–287.

[10] Y. Nakano, H. Yahagi, Y. Okada, and S. Yoshida, "Highly efficient universal coding with classifying to subdictionaries for text compression," *Proceedings of the 1994 Data Compression Conference*, 234–243.

[11] M. Ohmine and H. Yamamoto, "Universal data compression algorithms with multiple dictionaries," *Tech. Repo. of Inst. of Elec. Inf. Comm. Eng.* (1993), 1–6.

[12] M. Slyz, "Image Compression using a Ziv-Lempel type Coder," University of Michigan School of Engineering, Master's Thesis, 1991.

[13] J. A. Storer, *Data Compression: Methods and Theory*, CS Press, New York, 1988.

[14] T.A. Welch, "A technique for high performance data compression," *Computer* (1984), 8–19.

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

[16] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," *IEEE Transactions on Information Theory* 24 (September 1978), 530–536.