An Order-2 Context Model for Data Compression With Reduced Time and Space Requirements

Debra A. Lelewer and Daniel S. Hirschberg

Technical Report No. 90-33

Abstract

Context modeling has emerged as the most promising new approach to compressing text. While context-modeling algorithms provide very good compression, they suffer from the disadvantages of being quite slow and requiring large amounts of main memory in which to execute. We describe a context-model-based algorithm that runs significantly faster and uses less space than earlier context models. Although our algorithm does not achieve the compression performance of competing context models, it does provide a significant improvement over the widely-used Unix utility *compress* in terms of both use of memory and compression performance.

Introduction

The most widely used data compression algorithms, including the Unix utility *compress*, are based on the work of Ziv and Lempel [ZL78]. These are dynamic algorithms that build a dictionary representative of the input text and code dictionary entries using fixed-length codewords. *Compress* typically reduces a file to 40–50% of its original size. *Compress* is extremely fast, but has a large memory requirement (450 Kbytes). An updated version of the Ziv-Lempel algorithm requires less memory (186 Kbytes for encoding and 130 Kbytes for decoding) and achieves better compression (compressing files by an additional 30% on average) [FG89].

Newer approaches to data compression tend to focus on files of one particular type, and text files are most commonly studied. The most promising new methodology is one that predicts successive characters taking into account the context provided by characters already seen. What is meant by *predict* here is that previous characters are used in determining the number of bits used to encode the current character. A method of this type is referred to as a *context model* and, if the number of previous characters used to make a prediction is constant, an *order-i context model*. When $i = 0$, no context is used and the text is simply coded one character at a time. This is the model most commonly discussed in connection with Huffman coding ([H52]; [G78]) and with arithmetic coding [WNC87]. When $i = 1$, the previous character is used in encoding the current character; when $i = 2$, the previous two characters are used, and so on.

A context model is generally combined with arithmetic coding to form a data compression system. The model provides a frequency distribution for each context (each character in the order-1 case and each pair of characters in the order-2 case). Each frequency distribution forms the basis of an arithmetic code and these are used to map events into code bits. Huffman coding may be used in concert with finite-context models but will generally perform less effectively. This is because Huffman coding is constrained to represent every event (character) using an integral number of bits. While information theory tells us that an event with probability $\frac{4}{5}$ contains $\lg \frac{5}{4}$† bits of information content and should be coded in $\lg \frac{5}{4} \approx .32$ bits, Huffman coding will assign 1 bit to represent this event. Finite-context models frequently construct skewed distributions, often with one or two very high probability events.

† lg denotes the base 2 logarithm

Arithmetic codes are able to represent an event using $\lg p$ bits where p is the probability of the event and are thus able to code events with probability greater than 1/2 in less than 1 bit. Context modeling is usually used adaptively so that, like the Lempel-Ziv model and dynamic Huffman models, it requires only a single pass over the data to be compressed.

A disadvantage of context models is that the memory requirement of the model frequently exceeds the size of the file being compressed. Bell et al. report compression ratios (compressed file size divided by original file size) for an order-3 context model of approximately 30% on average, but the model requires 500 Kbytes of memory to achieve this compression [BWC90]. While this quantity of memory may be available on research or production machines, it is not generally available. In particular, microcomputer implementations must greatly reduce memory utilization. Another disadvantage of context models is that they tend to be much slower than the Lempel-Ziv style of compression. Bell et al. report encoding and decoding speeds of 2000 characters per second (cps) for the order-3 context model as compared with 12000 cps for *compress* and 6000 cps for the updated Ziv-Lempel method of Fiala and Green.

The algorithm we describe improves the practicality of the context modeling concept. Our modifications of the basic finite-context model improve its speed and decrease its memory requirements. We are willing to sacrifice some compression efficiency to achieve the speed and memory improvements as long as our system also provides advantages over the state-of-the-art *compress*. We present our algorithm from the point of view of the encoder, describing the way in which the encoder maintains a context model and uses corresponding frequency values to code characters. As in any adaptive data compression algorithm, the decoder must maintain the same model and use the frequency information in a compatible way so as to correctly interpret data received from the encoder.

Finite-Context Modeling

In this section we present an introduction to context modeling. We include only that information needed to understand our approach to the use of context, however, and a more comprehensive discussion can be found in the book by Bell et al. [BCW90]. In order to simplify our introduction to the concept of finite-context

modeling, we defined an order-i context model to be one in which the previous i characters are always used to code the current character. Such a model should more accurately be referred to as a *pure* order-i model (or a model of *fixed* order i) to distinguish it from the more common *blended* context model. A blended model of order i is one in which the order i model is blended with models of orders $i-1, i-2, \ldots, 0$. Blending is desirable and essentially unavoidable in an adaptive setting where the model is built from scratch as encoding proceeds. When the first character of a file is read, the model has no history on which to base predictions. Larger contexts become more meaningful as compression proceeds.

In a typical blended order-i model, the number of bits used to code character c will be dictated by the preceding i characters if c has occurred in this particular context before. Otherwise, models of lower order are consulted, beginning with order $i-1$, until one of them supplies a prediction. When the context of order i fails to predict the current character, the encoder must emit an *escape* character, a signal to the decoder that the lower model is being consulted. The order-0 model may be initialized to provide a prediction for each character so that the process of consulting lower-ordered models terminates. Alternatively, the order-0 model may be used only for characters that have appeared before but are now appearing in a novel context. In this case, a model of order -1 is used for predicting characters when they occur for the first time. The -1 model is initialized so that each of the unused characters is equally likely. When a character occurs in a novel context, this new information is added to the model being constructed.

We will call an order-i context model *full* if for all $k < i$, every *k-gram* (sequence of k contiguous characters) that occurs in the file being encoded forms an order-i context in the model being constructed. A full model of even order 3 is rare since the space required to store context information for every 3-gram, 2-gram, 1-gram, and single character in the file is prohibitive. The PPMC algorithm of Bell et al. is a full context model of order 3 stored in a tree data structure that is allowed to grow to 500 Kbytes [BCW90]. The model is rebuilt from scratch when it reaches this limit. We consider strategies that use less space and execute faster but that still achieve better compression than the state-of-the-art *compress*.

Previous Work on Context Models with Modest Memory Requirements

Langdon and Rissanen describe a algorithm (LR) that uses a subset of the order-1 model [LR83]. Algorithm LR uses a model consisting of z order-1 contexts and an order-0 context (z is a parameter associated with the algorithm and determines its memory requirements). When encoding begins, the order-0 model is used since no characters have yet occurred in any order-1 context. In a full order-1 model, when a character occurs for the first time it becomes an order-1 context. In algorithm LR, only z contexts will be constructed: for the first z characters to occur at least N times in the text being encoded (N is another parameter of the algorithm). The suggested values $z = 31$ and $N = 50$ provide approximately 50% compression with a very modest space requirement and very good speed [BCW90].

Abrahamson presents an order-1 context model with very modest memory requirements. He describes his model as follows:

> If, for example, in a given text, the probability that the character h follows the character t is higher than that for any other character following a t and the probability of an e following a v is higher than that for any other character following a v, then the same symbol should be used to encode an h following a t as an e following a v. It should be noted that this scheme will also increase the probability of occurrence of the encoded symbol. ... the source message *abracadabra* can be represented by the sequence of symbols *abracadaaaa*. Notice how a b following an a and an r following a b (and also an a following an r) have all been converted into an a, the most frequently occurring source character [A89].

We believe a simpler description of Abrahamson's model characterizes it as an order-1 context model that employs a single frequency distribution and that codes symbol y following symbol x as symbol k, where k can be thought of as the position of y on x's list of successors and where successor lists are maintained in frequency count order. Thus we think of *bra* as being coded by 111 rather than *aaa*. The other characters in the string *abracadabra* will also be coded as list positions, but these positions cannot be inferred from the example. While this characterization

may not be obvious from the description given above, it becomes clear from the implementation details given in the paper [A89].

The data structures in Abrahamson's method consist of two-dimensional arrays *char_to_index*, *index_to_char* and *count*, and one-dimensional frequency and cumulative-frequency arrays. The frequency count array stores in $count[x, y]$ the number of times that character y has appeared in context x (i.e., following character x). The *char_to_index* array is used by the encoder to map characters to frequency values and the *index_to_char* array is used by the decoder to map frequency values to characters. The value of $char\_to\_index[x, y]$ gives the position of y on x's successor list and this position is used to index into the frequency distribution. The single frequency distribution may be thought of as representing the frequencies of occurrence of the various list positions (k values) and this distribution is used for arithmetic coding of the events modeled.

Thus, we recognize that Abrahamson is modifying the basic order-1 model by a) employing a single frequency distribution rather than a distribution for each 1-character context and b) employing self-organizing lists to map characters to frequency values. Abrahamson's model is a *pure* order-1 context model. That is, it is always possible to predict the next character given its predecessor. For any pair x, y of successive characters, we code y using the k^{th} frequency value where $char\_to\_index[x, y] = k$. There is intuitive appeal in the use of the frequency count list organizing strategy in Abrahamson's algorithm since the coding technique employed is based on frequency values. On the other hand, the frequency values used are aggregate values. Character y in context x is coded not using $count[x, y]$, but $frequency[k]$ where k is the position of y on the self-organizing list for context x. That is, the frequency used for encoding is not the frequency with which y has occurred after x, but the number of times that position k has been used to encode an event.

We have investigated the performance of other self-organizing list strategies in connection with Abrahamson's model (for a survey of list-organizing strategies, see [HH85]). We tested the performance of move-to-front, transpose, and move-p%-of-the-way-to-front (for $p = 33, 50, 67, 70$, and 75). For most files we tested, frequency count provided the best compression ratios, but the differences in performance were not dramatic. Our results agree with research by Horspool and Cormack in which

a variety of list organizing strategies are used in connection with an order-0 context model based on words rather than characters. They also report no significant performance differences among list organizing methods [HC87]. The use of transpose or move-to-front obviates the need for frequency counts in Abrahamson's algorithm and reduces the memory requirement from 200 Kbytes to 68 Kbytes when $n = 256$.

Order-2 Context Models in Limited Memory

When used to encode text files (where the alphabet size is typically in the range 90–128), Abrahamson's algorithm provides a speed advantage over the context model PPMC and a space advantage over PPMC and *compress*. However, the compression performance, approximately 54%, compares poorly with that provided by PPMC and *compress*. Abrahamson's algorithm provides about the same throughput as that of the updated Lempel-Ziv algorithm by Fiala and Green (algorithm FG, [FG89]), and this is significantly slower than *compress*. The space required is less than than of algorithm FG only for small alphabets ($n \leq 200$). Using the transpose list organizing strategy instead of frequency count improves the space requirements of Abrahamson's algorithm, but provides the same mediocre compression performance.

We consider finite-context models of order 2. Abrahamson's technique of using a single frequency distribution would provide some memory reduction, but to maintain a self-organizing list of size n (where $n = 128$ or 256) for each two-character context is prohibitive. We use a blended order-2 model and maintain a self-organizing list of size s, where $s \ll n$, for each two-character context. When z occurs in context xy and z is not on the xy list we must leave the order-2 model and code z on some other basis. There are several candidates for this alternative basis. We may employ a pure order-1 model at this point, or an order-0 model, or we may maintain short order-1 lists and resort to order-0 when z appears on neither the xy list nor the y list.

We have considered each of these alternatives. Our experiments indicate that an order-2-and-1 model is the least successful of the three options. Models based on orders 2 and 0 and models based on orders 2, 1, and 0 have produced similar compression results. The order-2-and-0 model allows faster encoding/decoding since it consults at most two contexts per character. We discuss the order-2-and-0

model that displays the best compression performance. We refer to this model as *partially blended* since it does not consult all models of lower order.

In our order-2-and-0 model, we maintain a self-organizing list of size s for each two-character context. We encode z in context xy by event k if z is in position k of list xy. When z does not appear on list xy we encode z itself. The order-0 part of the model consists of frequency values for n characters. Encoding entails mapping the event (k or z) to a frequency and employing an arithmetic coder. To complete the description of the model, we need to specify a list-organizing strategy and the method of maintaining frequencies. The frequency count list-organizing strategy is inappropriate because of the large number of counts required. We use the transpose strategy because in addition to not requiring frequency counts it also provides faster update than move-to-front.

In order to conserve memory we do not use a frequency distribution for each context. Instead, we maintain a frequency value for each feasible event. Since there are $s + 1$ values of k (the s list positions and the escape value) and $n + 1$ values for z (the n characters of the alphabet and an end-of-file character), the number of feasible events is $s + n + 2$. We can maintain the frequency values either as a single distribution or as two distributions, an order-2 distribution to which list positions are mapped and an order-0 distribution to which characters are mapped. Our experiments indicate that the two-distribution model is slightly superior. When z occurs in context xy we use the two frequency distributions in the following way: if list xy exists and z occupies position k, we encode k using the order-2 distribution. If list xy exists but does not contain z, we encode an escape code (using the order-2 distribution) as a signal to the decoder that an order-0 prediction (and the order-0 frequency distribution) is to be used, and then encode the character z. When list xy has not been created yet, the decoder knows this and no escape code is necessary; we encode z using the order-0 distribution.

The escape code must be chosen so that the decoder recognizes it as a signal rather than a list position. If viewed as just another list position, there are two reasonable choices for the value of the escape. The value $s + 1$ will never represent a list position; or we may use $size + 1$, where $size$ is the current size of list xy (and ranges from 1 to s). In the first case, the escape code is the same for every context and all of the counts for the escape code accrue to a single frequency

value; in the second case, the value of the escape code depends on the context and generates counts that accrue to multiple frequency values. The two escape strategies produce similar compression results. The algorithm we describe here uses the second alternative.

We also need to specify how the self-organizing lists and frequency distributions are updated. A list is updated for each character encoded. That is, when z occurs in context xy, the xy list is updated, either by transposing z with its predecessor or by adding it to the list. Similarly, we update a frequency distribution when it is used. Thus, when list xy exists, the order-2 distribution is updated after it is used to encode either a list position or an escape. The order-0 distribution is used and updated when z is not predicted by context xy.

The data stored for our method includes frequency and cumulative frequency lists of size $s+2$ (for order 2) and $n+1$ (for order 0), and *pos_to_freq* and *freq_to_pos* arrays of size $s+1$ and $n+1$, as well as the self-organizing lists of size s. The *pos_to_freq* and *freq_to_pos* arrays play the role of Abrahamson's *char_to_index* and *index_to_char* arrays, mapping list positions to frequencies in the order-2 context and characters to frequencies in the order-0 context. When the self-organizing lists are implemented as arrays, the total memory requirement of our method is $n^2(s+1)+5(s+n+2)+3$ bytes. With an s value as low as 2, our method is faster than Abrahamson's and provides better compression with less storage required. Based on empirical data, $s = 7$ provides the best average compression over a suite of test files. With $s = 7$ we use approximately three times as much memory as Abrahamson's method but achieve compression that is 21% better on average (3.3 bits per character as opposed to 4.2) and in slightly less execution time. Our method also provides better compression than Unix's *compress* (approximately 15% better with $s = 7$) using essentially the same memory requirement for $n = 256$ and far less for $n = 128$.

Using dynamic memory allocation to implement the self-organizing lists results in a far more efficient use of space. We allocate an array of n^2 pointers to potential lists, and allocate space for list xy only if xy occurs in the text being compressed. The memory requirement becomes $n^2 + u(s+1) + 5(s+n+2) + 3$ bytes, where u represents the number of distinct character pairs occurring in the text. In our suite of test files, the maximum value of u was 4721; this value was

encountered in file **windows**, a 0.69 megabyte file of messages extracted from the bulletin board *comp.windows.x*. Even in this worst case, the dynamic-memory version of the order-2-and-0 algorithm results in a 95 Kbyte space savings over Abrahamson's method (when both methods use $k = 256$ and with $s = 7$, our space requirement is ≈ 104 Kbytes and his ≈ 199 Kbytes). The compression performance is, of course, the same as that provided by an array-based implementation. The dynamic-memory implementation is slightly slower than the static version due to overhead incurred by dynamic allocation, but this algorithm is still faster than Abrahamson's algorithm.

File type	Order 2-and-0	Unix Compress	Abrahamson's Order-1
bboard	45.96	47.69	51.61
doc	38.76	42.85	48.98
TEX	40.98	43.09	50.14
source	33.66	41.30	45.30
non-text	51.24	55.94	57.08
all	41.09	48.06	51.89

Table 1 Performance by category.

Experimental Results

We compare the performance of the order-2-and-0 method to that of *compress* and Abrahamson's method on a suite of 34 files selected to include a variety of file types and sizes. Since *compress* is available under Unix and source code for Abrahamson's method appears in [A89], we are able to run each of these methods against our test suite. Where possible, we include files used by other researchers to compare with competing compression algorithms. The files we use can be grouped into categories: *bboard* files consisting of electronic bulletin board entries, *doc* files of on-line program documentation/user's manuals, TEX-formatted versions of technical papers, *source* files in C and Pascal, *non-text* files including a *dvi* file and a binary file, and *miscellaneous* file types. The miscellaneous file

category includes files **alphabet** (enough copies of the 26-letter alphabet to fill out 100,000 characters) and **skewstat** (10,000 copies of the string *aaaabaaaac*) described by Witten et al. [WNC87] and the Unix dictionary **/usr/dict/words** described by Williams [W88]. Table 1 presents a performance comparison of our method with *compress* and Abrahamson's order-1 method. Data reported are average compression ratios by category and overall.

In Table 2 we display results for some specific files. These are: **alphabet**, **skewstat**, and **/usr/dict/words** described above; **fcsh**, the formated manual entry for the *csh* command in Unix; **ocsh**, the object code for the *csh* command; **compress20** through **compress500**; and **compress**. **Compress** is the C source code for the Unix utility *compress* and **compress20** contains the first 20 lines of **compress**. Original file sizes are listed in column two.

File	Original Size	Order 2-and-0	Unix Compress	Abrahamson's Order-1
alphabet	100000	0.05	3.05	0.58
skewstat	100000	9.06	1.80	12.09
/usr/dict/words	201089	38.52	51.10	49.33
fcsh	77844	37.84	38.10	44.30
ocsh	118784	59.82	65.35	62.68
comp20	578	69.38	83.56	80.45
comp50	1234	59.32	68.80	66.94
comp100	2292	49.87	59.82	58.38
comp200	4877	49.33	58.48	58.60
comp500	13314	47.30	54.35	55.88
compress	35382	41.75	47.67	51.81

Table 2 Performance on selected files.

Witten et al. describe **alphabet** and **skewstat** and give results for an order-0 context model [WNC87]. This model achieves compression ratios of 59.29 and 12.09 respectively. Table 2 shows that our order-2-and-0 model outperforms the

order-0 model, *compress*, and the order-1 model of Abrahamson on **alphabet** and that only *compress* performs better on **skewstat**. These files are not typical of text, however, so performance on them is of little interest. Williams reports results on **/usr/dict/words** and the various **compress** files [W88]. The values he gives for original file sizes are slightly different from ours since local copies of the files contain minor differences. Williams' dynamic-history compression technique achieves a compression ratio of 58.3 on **/usr/dict/words** and ratios of 69.9, 57.3, 45.4, 49.2, 40.1, and 42.24 on the versions of the *compress* source. Williams' motivation in considering subsets of the *compress* source was to emphasize the fact that his model 'learns' the characteristics of a file as it compresses. Thus, a larger file provides more opportunity for learning and greater compression is achieved. Any dynamic data compression scheme learns characteristics of a source as compression proceeds. Compression performance improves with file size to the point at which the limit on available memory is reached. When the algorithm can no longer store new information, performance may degrade. The *compress* files and the *source* category in Table 1 demonstrate that the order-2-and-0 method performs particularly well on source program files. Cormack and Horspool report results for files **fcsh** and **ocsh** [CH87]. The values for original file size differ from ours substantially, so comparisons are unreliable. Cormack and Horspool report that an order-4 context model achieves compression ratios of 26.5 and 69.4 on **fcsh** and **ocsh** respectively and that a dynamic Markov model (which is essentially a finite context model) provides ratios of 27.2 and 54.8. Our results for **ocsh** compare favorably to theirs. In comparing ratios, however, we must keep in mind that the files may be quite different and, more importantly, that the models they discuss have unlimited memory requirements.

Using Hashing to Improve Memory Use

We have described an algorithm that allocates n^2 self-organizing lists of size s and another that uses dynamic memory to allocate lists of size s only when they are needed. The second algorithm, however, statically allocates n^2 pointers, one for each of the n^2 possible contexts. In this section we describe an order-2-and-0 strategy that uses hashing rather than dynamic memory. This algorithm employs a hash table into which all n^2 contexts are hashed. Each hash table entry is a self-organizing list of size s. An implementation of this strategy provides better average compression than the earlier methods and requires much less memory.

Encoding and decoding proceed as in the earlier algorithms: when z occurs in context xy and no xy list exists we encode z using the order-0 frequency distribution; when an xy list exists but does not contain z, we emit an escape code and then code z using the order-0 distribution; when z is contained on the list for xy we code its position. An obvious disadvantage of the use of hashing is the possibility of collision. If two contexts hash to the same table position, the lists for these contexts are coalesced into a single list. This does not affect the correctness of the approach, but may impact its compression performance. We mitigate the negative effects of hashing in two ways. First, we use linear probing to resolve collisions. In order to maintain reasonable running time we perform only a small number of probes (four in the implementation we describe here). Second, we use some of the space gained by eliminating n^2 pointers to provide $m > 1$ order-2 frequency distributions. The value of m is is significantly smaller than the size of the hash table (H) so that we are coalescing H/m lists into each frequency distribution. Thus the cost is less than that of providing a frequency distribution for each context while compression results are better than those achieved when we use a single frequency distribution for all lists.

An implementation of the hash-based algorithm with $H = 4800$, $m = 70$, $s = 7$, and $n = 256$ provides approximately 4% more compression than the order-2-and-0 algorithm described above and uses only 48 Kbytes of memory. We provide empirical comparisons with the pointer-based algorithm in Table 3.

Future Research

Our algorithm provides an order-2 model that makes efficient use of internal storage. The use of main memory is particularly critical for microcomputer compression programs. We will compare our results with state-of-the-art microcomputer compression utilities. We offer better compression than that achieved by *compress*; it seems likely that our work represents an even more valuable improvement over limited-memory microcomputer versions of Lempel-Ziv coding.

We are also investigating limited-memory models based on order-3 context. These models must be implemented using dynamic memory, and the 'array of

File/Type	Order 2-and-0 (Dynamic)	Order 2-and-0 (Hashing)
bboard	45.96	44.46
doc	38.76	36.44
TEX	40.98	39.70
source	33.66	32.92
non-text	51.24	50.00
all	41.09	39.49
/usr/dict/words	38.52	36.94
fcsh	37.84	35.45
ocsh	59.82	58.23
comp20	69.38	69.55
comp50	59.32	57.29
comp100	49.87	47.47
comp200	49.33	47.16
comp500	47.30	45.26
compress	41.75	40.10

Table 3 Performance comparison - dynamic memory and hashing.

pointers' dynamic strategy discussed for the order-2-and-0 model is not appropriate. For an alphabet of 256 symbols, storing even a single pointer for each 3-character context is prohibitive.

Summary

We present an order-2-and-0 finite context model that provides compression performance better than that of the state-of-the-art Unix utility *compress* and has memory requirements far more modest than those of *compress*. Our algorithm provides improved compression performance on files of many types and performs particularly well on program source codes. We believe that our model is conceptually simple and easy to implement.

References

[A89] Abrahamson, D. M. An adaptive dependency source model for data compression. *Commun. ACM 32*, 1 (Jan., 1989), 77–83.

[BCW90] Bell, T., Cleary, J. G., and Witten, I. H. *Text Compression*, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[CH87] Cormack, G. V. and Horspool, R. N. S. Data compression using dynamic Markov modeling. *Comput. J. 30*, 6 (Dec., 1987), 541–550.

[FG89] Fiala, E. R. and Greene, D. H. Data compression with finite windows. *Commun. ACM 32*, 4 (Apr., 1989), 490–505.

[G78] Gallager, R. G. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory 24*, 6 (Nov., 1978), 668–674.

[HH85] Hester, J. H. and Hirschberg, D. S. Self-organizing linear search. *ACM Comput. Surv. 17*, 3 (Sept., 1985), 295–311.

[HC87] Horspool, R. N. and Cormack, G. V. A locally adaptive data compression scheme. *Commun. ACM 16*, 2 (Sept., 1987), 792–794.

[H52] Huffman, D. A. A method for the construction of minimum- redundancy codes. *Proc. IRE 40*, 9 (Sept., 1952), 1098–1101.

[LR83] Langdon, G. G. and Rissanen, J. J. A double-adaptive file compression algorithm. *IEEE Trans. Comm. 31*, 11 (Nov., 1983), 1253–1255.

[W88] Williams, R. N. Dynamic-history predictive compression. *Information Systems 13*, 1 (Jan., 1988), 129–140.

[WNC87] Witten, I. H., Neal, R. M., and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June, 1987), 520–540.

[ZL78] Ziv, J. and Lempel, A. Compression of individual sequences via variable- rate coding. *IEEE Trans. Inf. Theory 24*, 5 (Sept., 1978), 530–536.