Context Modeling for Text Compression

Daniel S. Hirschberg[†] and Debra A. Lelewer[‡]

Abstract

Adaptive context modeling has emerged as one of the most promising new approaches to compressing text. A *finite-context model* is a probabilistic model that uses the context in which input symbols occur (generally a few preceding characters) to determine the number of bits used to code these symbols. We provide an introduction to context modeling and recent research results that incorporate the concept of context modeling into practical data compression algorithms.

1. Introduction

One of the more important developments in the study of data compression is the modern paradigm first presented by Rissanen and Langdon [RL81]. This paradigm divides the process of compression into two separate components: modeling and coding. A model is a representation of the source that generates the data being compressed. *Modeling* is the process of constructing this representation. *Coding* entails mapping the modeler's representation of the source into a compressed representation. The coding component takes information

[†] Department of Information and Computer Science, University of California, Irvine

[‡] Computer Science Dept., California State Polytechnic University, Pomona

supplied by the modeler and translates this information into a sequence of bits. Recognizing the dual nature of compression allows us to focus our attention on just one of the two processes.

The problem of coding input symbols provided by a statistical modeler has been well studied and is essentially solved. Arithmetic coding provides optimal compression with respect to the model used to generate the statistics. That is, given a model that provides information to the coder, arithmetic coding produces a minimal-length compressed representation. Witten et al. provide a description and an implementation of arithmetic coding [WNC87]. The well-known algorithm of Huffman is another statistical coder [H52]. Huffman coding is inferior to arithmetic coding in two important respects. First, Huffman coding is constrained to represent every event (e.g., 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. The accuracy of arithmetic coding is limited only by the precision of the machine on which it is implemented. The second advantage of arithmetic coding is that it can represent changing models more effectively. Updating a Huffman tree is much more time consuming. Researchers continue to refine arithmetic coding for purposes of efficiency.

Given the existence of an optimal coding method, modeling becomes the key to effective data compression. The selection of a modeling paradigm and its implementation determine the resource requirements and compression performance of the system. Context modeling is a very promising new approach to statistical modeling for text compression. Context modeling is a special case of *finite-state modeling*, or *Markov modeling*, and in fact the term Markov modeling is frequently used loosely to refer to finite-context modeling. In this section we describe the strategy of context modeling and the parameters involved in implementing context models.

† lg denotes the base 2 logarithm

1.1. Basics of Context Modeling

A *finite-context model* uses the context provided by characters already seen to determine the encoding of the current character. The idea of a context consisting of a few previous characters is very reasonable when the data being compressed is natural language. We all know that the character following *q* in an English text is all but guaranteed to be *u* and that given the context *now is the time for all good men to come to the aid of*, the phrase *their country* is bound to follow. One would expect that using knowledge of this type would result in more accurate modeling of the information source. Although the technique of context modeling was developed and is clearly appropriate for compressing natural language, context models provide very good compression over a wide range of file types.

We say that a context model predicts successive characters taking into account the context provided by characters already seen. What is meant by *predict* here is that the frequency values used in encoding the current character are determined by its context. The frequency distribution used to encode a character determines the number of bits it contributes to the compressed representation. When character x occurs in context c and the model for context c does not include a frequency for x we say that context *c fails to predict x*.

A context model may use a fixed number of previous characters in its predictions or may be a *blended* model, incorporating predictions based on contexts of several lengths. A model that always uses i previous characters to predict the current character is a *pure order-i context model.* When $i = 0$, no context is used and the text is simply coded one character at a time. 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 *blended* model may use the previous three characters, the previous two characters when the three-character context fails to predict, and one predecessor if both the order-3 and order-2 contexts fail. A blended model is composed of two or more submodels. An order-i context model consists of a frequency distribution for each i-character sequence occurring in the input stream. In the order-1 case, this means that the frequency distribution for context *q* will give a very high value to *u* and very little weight to any other letter, while the distribution for context *t* will have high frequencies for *a, e , i, o, u*, and *h* among others and very little weight for letters like *q, n* and *g*.

A blended model is *fully blended* if it contains submodels for the maximum-length context and *all* lower-order contexts. That is, a fully-blended order-3 context model bases its predictions on models of orders 3, 2, 1, 0, and -1 (the model of order -1 consists of a frequency distribution that weights all characters equally). A *partially-blended* model uses some, but not all, of the lower-order contexts.

Context modeling may also be either static or dynamic. A model is *static* if the information of which it consists remains unchanged throughout the encoding process. A *dynamic*, or *adaptive*, model modifies its representation of the input as encoding proceeds and it gathers information about the characteristics of the source. Most static data compression models have adaptive equivalents and the adaptive counterparts generally provide more effective compression. In fact, Bell et al. prove that over a large range of circumstances there is an adaptive model that will be only slightly worse than *any* static model, while a static model can be arbitrarily worse than an adaptive counterpart [BCW90]. We will confine our discussion to adaptive context models.

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 is not appropriate for use with adaptive context models for the reasons given above.

1.2. Methods of Blending

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. The general mechanism of blending, *weighted blending*, assigns a probability to a character by weighting probabilities (or, more accurately, frequencies) provided by the various submodels and computing the weighted sum of these probabilities. This method of blending is too slow to be practical and has the additional disadvantage that there is no theoretical basis for assigning weights to the models of various orders. In a simpler and more practical blended order-i model, the number of bits used to code character c is dictated by the preceding i characters if c has occurred in this particular context before. In this case, only the order-i frequency distribution is used. Otherwise, models of lower orders are consulted until one of them supplies a prediction. When the context of order i fails to predict the current character, the encoder emits an *escape* code, a signal to the decoder that the model of lower order is being consulted. Some lowest-order model must be guaranteed to supply a prediction for every character in the input alphabet.

The frequencies used by the arithmetic coder may be computed in a number of ways. One of the more straightforward methods is to assign to character x in context c the frequency f where f is the number of times that context c has been used to predict character x. Alternatively, f may represent the number of times that x has occurred in context c. An implementation may also require x to occur in context c some minimal number of times before it allocates frequency to the event.

1.3. Escape Strategy

In order for the encoder to transmit the escape code, each frequency distribution in the blended model must have some frequency allocated to *escape*.

A simple strategy is to treat the escape event as if it were an additional symbol in the input alphabet. Like any other character, the frequency of the escape event is the number of times it occurs. Other strategies involve relating the frequency of the escape code to the total frequency of the context and the number of different characters occurring in the context. On one hand, as the number of different characters increases, the probability of prediction increases and the use of the escape code becomes less likely. On the other hand, if a context has occurred frequently and predicted the same character (or small number of characters) every time, the appearance of a new character (and the need to escape) would seem unlikely. There is no theoretical basis for selecting one of these escape strategies over another. Fortunately, empirical experiments indicate that compression performance is largely insensitive to the selection of escape strategy.

1.4. Exclusion

The blending strategy described in Section 1.3 has the effect of excluding lower-order predictions when a character occurs in a higher-order model. However, it does not exclude as much lower-order information as it might. For example, when character x occurs in context abc for the first time the order-2 context bc is consulted. If character y has occurred in context abc it can be excluded from the order-2 prediction. That is, the fact that we escape from the order-3 context abc informs the decoder that the character being encoded is not y. Thus the bc model need not assign any frequency to y in making this prediction. By excluding y from the order-2 prediction x may be predicted more accurately. Excluding characters predicted by higher-order models can double execution time. The gain in compression performance is on the order of 5%, which hardly justifies the increased execution time [BCW90]. Another type of exclusion that is much simpler and has the effect of decreasing execution time is *update exclusion*. Update exclusion means updating only those models that contribute to the current prediction. Thus if, in the above example, context bc

predicts x, only the order-3 model for abc and the order-2 model for bc will be updated. The models of lower order remain unchanged.

1.5. Memory Limitations

We call an order-i context model *complete* if for every character x occurring in context c, the model includes a frequency distribution for c that contains a count for x. That is, the model retains all that it has learned. Complete context models of even order 3 are rare since the space required to store all of the context information gleaned from a large file is prohibitive. There are two obvious ways to impose a memory limit on a finite context model. The first is to monitor its size and freeze the model when the size reaches some maximum. When the model is frozen, it can no longer represent characters occurring in novel contexts, but we can continue to update the frequency values already stored in the model. The second approach is to rebuild the model rather than freeze it. The model can be rebuilt from scratch or from a buffer representing recent history. The use of the buffer may lessen the degradation in compression performance due to rebuilding. On the other hand, the memory set aside for the buffer causes rebuilding to occur earlier. A third approach, which is not strictly a solution to the problem of limited memory, is to monitor compression performance as well as the size of the data structure. Rebuilding when compression begins to degrade may be more opportune than waiting until it becomes necessary. We will say more about the data structures used to represent context models in later sections. The representation of the model clearly impacts the amount of information it can contain and the ease with which it can be consulted and updated.

1.6. Measuring Compression Performance

Compression performance is generally presented in one of two forms: *compression ratio* and *number of bits per character in the compressed representation.* Compression ratio is the ratio (size of compressed representation)/(size of original input) and may be expressed as a percentage of the original input remaining after compression. Each of these definitions has the virtue that it makes no assumptions (e.g., ergodicity) about the input stream, the model, or the coding component of the algorithm. Reporting compression in terms of number of bits per character also factors out the representation of the symbols in the original input.

The performance of a data compression technique clearly depends on the type of data to which it is applied, and further, on the characteristics of a particular data file. In order to measure the effectiveness of a data compression system, it should be applied to a large corpus of files representing a variety of types and sizes. A reliable comparison of techniques can be made only if they are applied to the same data.

2. Early Context-Modeling Algorithms

A variety of approaches to the use of context modeling have been developed. The algorithms differ in the selection of the parameters described in Section 1 and in terms of the data structures used to represent the models. We describe the most successful of the early context modeling algorithms in Sections 2.1 through 2.6. The compression performance, memory requirements, and execution speeds of the methods are compared in Section 2.7.

2.1. Algorithm LOEMA

The earliest use of context modeling for text compression is the Local Order Estimating Markov Analysis method developed by Roberts in his dissertation [R82]. Algorithm LOEMA is fully-blended and uses weights on the models to form a single prediction for each character in the text. The weights represent confidence values placed on each prediction. The data structure used to represent the model is a backward tree stored in a hash table. Roberts investigates methods of growing and storing the models so as to conserve memory. As discussed in Section 1, weighted blending is inefficient and algorithms that employ the escape strategy of blending provide a much more practical alternative to LOEMA.

2.2. Algorithm DAFC

Langdon and Rissanen's Double-Adaptive File Compression algorithm is one of the first methods that employs blending in an adaptive data compression scheme [LR83]. DAFC is a partially-blended 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 complete order-1 model, when a character occurs for the first time it becomes an order-1 context. In algorithm DAFC, only z contexts will be constructed: corresponding to the first z characters that 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 percent compression with a very modest space requirement and very good speed [BCW90].

DAFC employs neither explicit blending nor exclusion. Explicit blending is not required because once a model is activated it can predict any character in the input. That is, every model contains a frequency value for every character in the input alphabet. If character x occurs in context c and context c has occurred N times then x is predicted by c. Otherwise x is predicted by the order-0 model.

DAFC also employs run-length encoding when it encounters a sequence of three or more repetitions of the same character. This is equivalent to employing an order-2 model for this special case. Coding is performed by decomposition using the simple binary arithmetic code [LR82].

2.3. The PPM Algorithms

The PPM, or Prediction by Partial Match, algorithms are variable-order adaptive methods that implement the escape mechanism for blending. A PPM algorithm stores its context models in a single forward tree. The maximum number of characters of context (the order of the model) is a parameter of the PPM paradigm. The optimal order varies with file type and size. For text files, three or four appears to be the best choice. PPM algorithms are fully blended, so that an order-3 model uses submodels of order 3, 2, 1, 0, and -1. Members of the PPM family differ in terms of exclusion strategy and memory management. PPMC, the best of the PPM family, employs only update exclusion and imposes a limit on model size. Other PPM algorithms use full exclusion and allow the data structure to grow without bound.

The implementation of PPMC that proves most effective is an order-3 model that permits the tree to grow to a size of 500 Kbytes. The model is rebuilt using the previous 2048 characters when it reaches this limit.

2.4. Algorithm WORD

WORD is an algorithm that employs context models based on words and non-words, where a word is a sequence of alphabetic characters and a non-word a sequence of non-alphabetic characters [Mo89]. Each of the word and non-word submodels is a partially-blended model of orders 1 and 0. Words are predicted by preceding words if possible, and if not they are predicted without context. Similarly, non-words are predicted in the context of non-words. A model of order -1 is inappropriate for words that have not been seen before since we cannot allocate frequency to all possible words. Instead, when a word (respectively, non-word) occurs for the first time, its length is encoded using an order-0 model for lengths and then its letters (non-letters) are coded using a fully-blended model of order 1. WORD employs 12 submodels in total. For each mode (word and non-word) there are: order-1 submodels for words and letters; order-0 submodels for words, letters, and lengths; and a model of order -1 for letters. The escape mechanism is used for blending. Update exclusion is performed while prediction exclusion is rejected for reasons of efficiency. Hash tables are used to store words and non-words, and the arithmetic codes are represented by tree structures.

Moffat also experimented with including word (and non-word) models of order 2 and found that this did not provide significant improvement over the order 1 model described above. WORD provides good compression and is reasonably fast; its speed derives from the fact that the arithmetic encoder is only invoked about 20% as often as in a character-based compression system (the average length of an English word being about five characters).

2.5. Algorithm DHPC

Williams's Dynamic-History Predictive Compression technique is similar to the PPM algorithms. The two major differences explored by Williams are the use of a backward tree and the use of thresholds[W88]. Whereas PPM extends its tree every time it is presented with new context information, DHPC grows a branch only if the occurrence count of the parent node is sufficiently high. In addition, DHPC avoids rebuilding its tree by setting a limit on its size and applying no changes to the tree once the maximum size has been reached. DHPC is also similar to algorithm DAFC in that both use thresholding to limit the size of the data structure representing the context information. However, the threshold in DAFC requires that the child node occur sufficiently often to justify its inclusion as a submodel while DHPC requires the parent node to have sufficient frequency before it spawns any children. DHPC provides better compression than DAFC due to its use of higher-order context information. DHPC is faster than the PPM algorithms but yields poorer compression.

2.6. Algorithm ADSM

Abrahamson's Adaptive Dependency Source Model is a pure order-1 context model with modest memory requirements. Abrahamson 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, p 78]."

In simpler terms, Abrahamson's model is an order-1 context model that employs a single frequency distribution and encodes symbol y following symbol x as symbol k, where k is the position of y on x's list of successors and where successor lists are maintained in frequency count order. Thus, in the example above we think of *bra* as being encoded by 111 rather than *aaa*. The other characters in the string *abracadabra* will also be encoded 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 Abrahamson's article [A89].

Thus, Abrahamson is modifying the basic order-1 model by:

(1) employing a single frequency distribution rather than a distribution for each 1-character context and

(2) employing self-organizing lists to map characters to frequency values.

ADSM employs a *pure* order-1 context model. For any pair x, y of successive characters, y is coded using the k^{th} frequency value where y is the k^{th} most frequent successor of x. There is intuitive appeal in the use of the frequency count list organizing strategy in ADSM since the coding technique employed is based on frequency values. On the other hand, the frequency values used for coding are aggregate values. The frequency used for encoding character y in context x 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.

2.7. Comparison of the Methods

Recently, progress has been made toward standardizing methods of reporting compression performance. In the past, researchers reported performance of their approaches using only a few private data files. This provided no reliable basis for comparing algorithms. In addition, measures of compression varied widely and were occasionally either ill-defined or multiply defined (i.e., a single term was used to mean different things by different authors). Today the majority of researchers have settled on the two measures defined in Section 1.6, compression ratio and number of bits per character. In addition, a corpus of files of a variety of types and sizes is now available in the public domain to facilitate legitimate comparisons. Descriptions of the files in the corpus and methods of accessing them are given in [BCW90]. The data reported here is drawn from the books by Bell et al. and Williams[BCW90, W91A].

Algorithm PPMC represents the state of the art in context modeling. PPMC provides very good compression (approximately 30 percent on average), but has a large memory requirement (500 Kbytes) and executes slowly (encoding and decoding approximately 2000 cps on a 1MIP machine. Over the compression corpus PPMC achieves average compression of 2.48 bits per character (bpc). DAFC and ADSM fare worst among the methods described here, providing average compression of 4.02 bpc and 3.95 bpc, respectively. DAFC's performance in terms of both compression and use of memory falls between that of an order-0 and an order-1 method. ADSM also uses far less memory than a complete order-1 model, and the information lost results in a loss of compression performance. WORD provides compression close to that achieved by PPMC, 2.76 bits per character over the corpus. WORD is the fastest of the methods and achieves the 2.76 bpc performance with approximately the same memory resources as PPMC. DHPC is neither as fast nor as effective as WORD. The average number of bits per character for the corpus when compressed by DHPC is 2.98. Compression results are not available for LOEMA. Williams reports only that LOEMA provides compression comparable to that of PPMC[W91A]. LOEMA is of little practical interest, however, because it executes very slowly.

2.8. Other Competing Methods

At present, the most commonly-used data compression systems are not context-modeling methods. The UNIX utility *compress* and other Ziv-Lempel algorithms represent the current state of the art in practical data compression. Ziv-Lempel algorithms employ dictionary models and fixed codes. The books by Storer and Bell et al. provide complete descriptions of the Ziv-Lempel family of algorithms[St88, BCW90]. We mention the Ziv-Lempel methods only as competitors that must be reckoned with. *Compress* encodes and decodes using 450 Kbytes of internal memory and at a rate of approximately 15,000 cps [BCW90]. Its speed is its primary virtue and its compression performance is less than stellar. Over the corpus *compress* reduces files to an average of 3.64 bpc. A more recent Ziv-Lempel technique, algorithm FG, provides superior compression performance (2.95 bpc over the corpus) using less memory (186 Kbytes for encoding and 130 Kbytes for decoding), but executes only about half as fast as *compress* (encoding 6,000 cps and decoding 11,000 cps)[FG89]. Research continues on the Ziv-Lempel technique. A recent article by Williams proposes techniques for combining the compression performance of algorithm FG with the throughput of *compress*[W91b].

3. Context Modeling in Limited Memory

While context-modeling algorithms provide very good compression, they suffer from the disadvantages of being relatively slow and requiring large amounts of main memory in which to execute. Algorithm PPMC achieves an average compression ratio of approximately 30 percent. However, PPMC uses 500 Kbytes to represent the model it employs. As memory becomes less expensive and more accessible, machines are increasingly likely to have 500 Kbytes of internal memory available for the task of data compression. There are applications, however, for which it is unreasonable to require this much memory. Examples of these applications include mass-produced special-purpose machines, modems, and software systems in which compressors/decompressors are embedded in larger application programs. Algorithm PPMC has the additional disadvantage of executing at only 2000 characters per second. In the next two sections we describe efforts to improve the practicality of the finite-context modeling concept by streamlining the representation of the model.

Algorithms DAFC and ADSM described in Section 2 employ simplified order-1 context models that require less run-time memory than algorithm PPMC and execute faster. These methods sacrifice a great deal of compression performance in achieving gains in memory requirement and execution speed. In Section 4 we describe an order-2 context modeling algorithm that requires far less memory and executes faster than ADSM while achieving better compression performance. In Section 5 we extend our work on context modeling in limited memory to context models of order 3. The algorithm we describe achieves much of the compression performance of PPMC using far less space and executing much faster.

The improvements provided by the algorithms described in Sections 4 and 5 are achieved through the use of self-organizing lists and a restrained approach to blending. Both ADSM and DAFC apply the strategy of limited blending. ADSM uses self-organizing lists as well, applying the frequency count list organizing strategy. Self-organizing lists have also been used as the basis of non-context-based data compression systems. A move-to-front scheme was independently invented by Ryabko, Horspool and Cormack, Bentley et al. and Elias [BCW90]. Each of these authors evaluates several variations on the basic idea. The paper by Bentley et al. provides a great deal of data structure detail for maintaining a move-to-front list of words efficiently[BSTW86]. Horspool and Cormack investigated a variety of list organizing strategies for use in a word-based model[HC87]. Finally, the common implementation of arithmetic coding uses frequency count organization for faster access.

4. Space-Limited Context Models of Order 2

The algorithm we describe in this section employs a blended order-2 context model. It can be implemented so as to provide compression performance that is better than that provided by *compress* and much better than that provided by algorithm ADSM, using far less space than either of these systems (10 percent as much memory as *compress*). In Section 4.1 we describe the method of blending we employ, and in Section 4.2 we provide more detail on our use of self-organizing lists. In Section 4.3 we describe the frequency distributions maintained by our algorithm, and Section 4.4 presents our escape strategy. We discuss the memory requirement of our algorithm and its execution speed in Section 4.5, and consider the use of dynamic memory to improve the memory requirement. In Section 4.6 we show that hashing is a much more effective means to this end. We present some experimental data on the performance of our order-2-and-0 method in Section 4.6. A more detailed comparison with competing algorithms is given in [L91], and a comparison with our order-3 algorithm appears in Section 5.

4.1. Blending Strategy

One of the ways in which we conserve on both memory and execution time is by blending only models of orders 2 and 0, rather than orders 2, 1, 0, and -1. Thus we refer to our model as an order-2-and-0 context model. We have experimented with order-2-and-1 and order-2-1-and-0 models. The order-2-and-1 model did not provide satisfactory compression performance and the order-2-1-and-0 model produces compression results that are very close to those of our order-2-and-0 algorithm. The order-2-and-0 model allows faster encoding and decoding since it consults at most two contexts per character. We provide more details on the models of orders 2 and 0 and how they are blended in Section 4.2.

4.2. Self-Organizing Lists

In our order-2-and-0 model, we maintain a self-organizing list of size s for each two-character context (s is a parameter of the algorithm). We encode z when it occurs 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 using the order-0 model. Encoding entails mapping the event (k or z) to a frequency and employing an arithmetic coder. To complete the description of the algorithm, 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 employ the transpose strategy because it provides faster update than move-to-front.

When character z occurs in context xy and z appears on the context list for xy, the list is updated using the transpose strategy. If z does not appear on the xy list, it is added. If the size of list xy is less than s ($size < s$), the item currently in position $size$ moves into position $size + 1$ and z is stored in position $size$. If the list is full when z is to be added, z will replace the last item. An obvious disadvantage to fixing the size of the order-2 context lists is that the lists are likely to be too short for some contexts and too long for others. When an order-2 list (say, list xy) contains s items and a new character z occurs in context xy, we delete the bottom item (call it t) from the list and add z. Context xy no longer predicts t. This does not affect the correctness of our algorithm. When t occurs again in context xy it will be predicted by the order-0 model. The fact that encoder and decoder maintain identical models ensures correctness. In addition, the rationale behind the use of self-organizing lists is that we expect to have the s most common successors on the list at any point in time. As characteristics of the file change, successors that become common replace those that fall into disuse. The method of maintaining frequencies and using them to encode is described in Section 4.3.

4.3. Frequency Distributions

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 code) 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 simply encode z using the order-0 distribution. Our limited use of frequency distributions is similar to that of algorithm ADSM.

4.4. Escape Strategy

We adopt the strategy of treating the escape event as if it were an additional list position. Given this decision, there are two reasonable choices for the value of *escape*. One choice is to use the value $s + 1$, as it will never represent a list position. The second choice is to use the value $size + 1$, 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 *escape* accrue to a single frequency value while in the second case, the value of *escape* 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 first alternative.

We apply update exclusion in dealing with both lists and frequency distributions. That is, a list or frequency distribution is updated when it is used. Thus, when list xy exists, both the list and the frequency distribution are updated after being used to encode either a list position or an escape. The order-0 distribution is used and updated each time context xy fails to predict.

4.5. Memory Requirement and Execution Speed

The data stored for our method is a function of the list size s and the alphabet size n. When the self-organizing lists are implemented as arrays, the total memory requirement of our method is $O(n^2 s)$. With an s value as low as 2, our method is faster than ADSM 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 ADSM but achieve compression that is 20 percent better on average (3.16 bits per character as opposed to 3.94) and execute faster. Our method also provides better compression than *compress* (approximately 15 percent 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 much 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 $O(n^2 + us)$, 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 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 ADSM (when both methods use $k = 256$ and with $s = 7$, our space requirement is approximately half that of ADSM). 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 implementation of ADSM. C-language versions of ADSM and our dynamic-memory implementation compress approximately 1900 and 3000 characters per second, respectively. We estimate that if our implementation were optimized, its speed would be competitive with that of algorithm FG. The execution time of our algorithm is determined by the size of the input file, the size of the output file, and the lengths of the self-organizing lists. For each input character we consult and update one or both models, and use and update the corresponding frequency distribution(s). The time contribution due to the order-2 model consists of the time required to search for the current character on an order-2 list and the time required to update the order-2 list and corresponding frequency distribution. The time to search and update the model is limited by the current size of the self-organizing list (which is in turn bounded by the maximum list size, s). We expect the frequently-occurring characters to be near the front of the context lists, so that the average time spent in manipulating the order-2 model should be much less than the maximum list length, s. When the order-2 model does not supply a prediction, the order-0 model must be consulted. Consulting the order-0 model requires very little time since the order-0 model is stored in frequency-count order. Updating the order-0 model involves maintaining frequency-count order, so that a single update could require n operations. However, the frequency-count strategy maintains popular characters near the front of the list so that the average cost is again much less than the maximum.

When an order-2 list contains fewer than s items, we are subject to the criticism that we are not putting our memory resources where we need them. In fact, fixing the number of successors represents a tradeoff of the ability to predict any character against the ability to predict quickly while using a reasonable amount of space. Fixing the number of successors suggests the use of an array data structure rather than a linked structure; thus we avoid the space required for links and the time involved in creating and updating linked nodes. The links in a linked structure may also be viewed as consuming memory without directly representing information needed for prediction. Another disadvantage of the linked structure is that it is more difficult to control its growth. In algorithm PPMC, the tree is simply allowed to grow until it reaches a limit and then is discarded and rebuilt. Rebuilding can result in loss of prediction accuracy. When rebuilding takes place, all of the information constructed from the prefix of the file is lost. By contrast, our model loses only the ability to predict certain successors in certain contexts, and only when they have ceased to occur frequently. Finally, we must keep in mind that a dynamic data compression system attempts to "hit a moving target". When characteristics of the file being compressed change, it may be advantageous to lose some of the data collected in compressing the early part of the file. Unfortunately, we can only make an intelligent guess at what information to collect and when to discard it.

4.6. Using Hashing to Improve Memory Requirement

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 or more contexts (say xy and ab) hash to the same table position, the lists for these contexts are coalesced into a single self-organizing list used to represent both contexts. We can view this as xy's successors vying with those of ab for position on the list. Intuitively, it would seem that our predictions are more accurate when xy and ab are represented by separate lists. However, we repeat our admonitions on the unreliability of intuition in compressing text. It is possible that when we expect it least the characteristics of our file change and our good statistics become bad. Thus, we can be optimistic and hope that coalescing two lists will a) happen infrequently, b) not degrade performance, or c) will actually improve performance.

Hash conflicts have no impact on the correctness of the approach; they may, however, impact compression performance. We mitigate the negative effects of hashing in three ways. First, we select the hash function so as to minimize the occurrence of collisions. Second, we use double hashing to resolve collisions. In order to resolve collisions, we must be able to detect them. We detect collisions by storing with the self-organizing list an indication of the context to which it corresponds. When context xy hashes to position h but the check value at position h does not correspond to context xy, we know that we have collision. In order to maintain reasonable running time we perform only a small number of probes. If the short probe sequence does not resolve the hash conflict, we allow the two lists to coalesce.

The third way in which we minimize the negative effects of hashing is to 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. The disadvantages of coalescing frequency distributions are the same as those of coalescing lists except that coalescing lists is likely to cause loss of the ability to predict some characters (since two contexts now have s list positions between them instead of s each), and limiting the number of frequency distributions does not cause this problem. Because the number of frequency distributions is very small relative to the number of contexts of order 2 we consider hash collisions to be inevitable and do not attempt to resolve them.

An implementation of the hash-based algorithm with $H = 4800$, $m = 70$, $s = 7$, and $n = 256$ provides approximately 6 percent more compression than the order-2-and-0 algorithm described above and uses only 45 Kbytes of memory (less than half of the requirement of the dynamic-memory method). The use of hashing provides improved compression performance overall.

5. Space-Limited Context Models of Order 3

In this section we extend our work on context modeling in limited memory to context models of order 3†. The use of hashing to store context information permits the extension of the strategy developed in Section 3 to blended models of arbitrary order. The primary problem in designing an order-3 algorithm with modest memory requirements is that of deciding which lower-order models to blend with the order-3 model. We concentrate our discussion on the blended order-3 context model that gives the best overall results. Our algorithm has a much more modest memory requirement than competing algorithms FG and PPMC and provides compression performance that is superior on average to that provided by FG. In addition, it runs much faster than PPMC. When tuned, we expect encode speed comparable to that of the faster algorithm FG. In Section 5.1 we discuss the method of blending we employ, and in Section 5.2 the data structures used. Section 5.3 details the way in which the predictions

† A preliminary version of these results was presented at the 1991 Data Compression Conference[LH91]

supplied by our model are coded, and in Section 5.4 we discuss the memory requirements and execution speed of our order-3 algorithm. Section 5.5 contains experimental results comparing our order-3 method with PPMC and FG.

5.1. Blending Strategy

The best algorithm in our family is based on an order-3-1-and-0 context model. That is, we construct a prediction for the character being encoded by blending predictions based on the previous three characters, the previous character, and unconditioned character counts. We considered order-3-and-0 models and order-3-2-and-0 models as well as the order-3-1-and-0 approach that we describe here. The addition of order-2 context information to the order-3-and-0 model generally did not improve compression performance, while the addition of contexts of order 1 does provide significantly better results. Eliminating some of the models of lower order contributes to both the decreased memory requirement and increased speed of our methods. Thus we limited the total number of contexts to be blended to three, and did not consider models that blended orders 3, 2, 1, and 0, for example. In Section 5.2 we describe the way in which we store context information.

5.2. Data Structures

We use self-organizing lists to maintain the order-3 and order-1 context information. As in the order-2-and-0 model, we employ the transpose list organizing strategy. The order-3 context information is stored in two hash tables, table $H3$ of size $h3$ whose elements are self-organizing lists of size $s3$, and table $F3$ containing $f3$ frequency distributions. Thus, each trigram (i.e., context of order 3) appearing in the file being compressed is mapped to a position in table $H3$, where a list of $s3$ successor characters is stored. A second hash function maps the trigram to a position in table $F3$ that stores the frequency distribution corresponding to the $s3$ successor characters. Since there are only n order-1 lists (where n is the size of the alphabet), hashing is not used to store the self-organizing lists of order 1. We maintain a list of $s1$

successors for each single character (context of order 1). However, we maintain just $f1$ (where $f1 < n$) frequency distributions for the collection of order-1 lists. Thus while the order-3 model is a essentially a two-level hashing scheme, where a context hashes first to a position in the table of self-organizing lists and then to a smaller table of frequency distributions, the order-1 model employs just one level of hashing, mapping the n contexts to $f1$ frequency distributions. The order-0 data consists of frequency data for the n symbols of our alphabet.

5.3. Coding the Model

The models of order 3, 1, and 0 are used to form a prediction of the current character in much the same way as we used them in the order-2-and-0 algorithm. We encode character z occurring in context wxy by event k if z occurs in position k of the list for context wxy. If z does not appear on wxy's list, we code an escape and consult the list for the order-1 context y. An order-3 frequency distribution is used to code either k or $escape$. When the order-1 model is consulted, an order-1 frequency distribution is used to code either j (if z occurs in position j of list y) or $escape$. When neither context wxy nor context y predicts z we follow the two escape codes with an order-0 prediction (i.e., we code the character itself). If the list for context wxy (likewise context y) is empty, the corresponding escape code is not necessary.

The escape codes are represented as list positions $s3 + 1$ and $s1 + 1$, respectively. As in our order-2-and-0 algorithm we apply update exclusion so that lists and frequency distributions are updated only when they contribute to the prediction of the current character, z. If list wxy exists, we update it using the transpose heuristic. If no wxy list exists one will be created. If context wxy does not predict z, then the y list is updated using the transpose method. If list y is not used in the prediction, it is not updated. When list wxy exists, the wxy frequency distribution is updated after it is used to encode either a list position or an escape. When context wxy does not predict and list y exists, the y frequency distribution is updated. The order-0 frequency distribution is updated whenever the character itself is coded.

5.4. Memory Requirement and Execution Speed

Our order-3-1-and-0 algorithm is in fact a family of algorithms where each algorithm in the family corresponds to a different set of values for the parameters $s3$, $h3$, $f3$, $s1$, and $f1$. The space requirements, speed, and compression performance of a particular algorithm depend on the values of these parameters. We report results in Section 5.5 for an algorithm that executes in 100 Kbytes of memory and encodes and decodes approximately 2800 cps. Bell et al. report compression speeds for competing algorithms running on a 1-MIP VAX 11/780 [BCW90]. In order to provide a meaningful comparison of running times, we execute on our research machine the order-3-1-and-0 algorithm and the version of *compress* used by Bell et al. Using the execution time of *compress* as a baseline, we adjust the running time of our algorithm to reflect the difference in machines. While this approach is obviously imperfect, it provides a reasonable basis for comparison. Our programs are part of a research testbed and have not been optimized for speed. We believe that with some attention to optimization they can be tuned to compress at approximately the same rate as algorithm FG.

5.5. Experimental Results

We compare the performance of our order-3-1-and-0 model to that of *compress* and the 45-Kbyte method of Section 3 on the corpus used by Bell et al. to measure the performance of a collection of data compression methods [BCW90]. The files represent a variety of sizes and types: **obj1** and **obj2** are executable files for two different machines, **geo** is a file of 32-bit numbers representing seismic data, **pic** is a bit map of a black and white facsimile picture. The remaining files are ASCII files of various types including program source files (the **prog** files). In Table 1 we display compression ratios for the order-2-and-0 model, the order-3-1-and-0 models, and *compress*. The order-3-1-and-0 model used here has parameter settings: $s3 = 3$, $h3 = 12000$, $f3 = 900$,

File	Order 2-0	Order 3-1-0	Unix Compress
bib	3.27	2.49	3.35
book1	3.39	3.03	3.46
book2	3.23	2.69	3.28
geo	5.18	5.14	6.08
news	3.68	3.22	3.86
obj1	4.28	4.10	5.23
obj2	3.55	3.33	4.17
paper1	3.34	2.87	3.77
paper2	3.27	2.85	3.52
pic	0.91	0.90	0.97
progc	3.24	2.87	3.87
progl	2.58	2.06	3.03
progp	2.63	2.12	3.11
trans	2.71	2.02	3.27
Averages	3.23	2.84	3.64
Memory (Kbytes)	45	186	500

Table 1

Comparison of order-3-1-0, order-2-and-0, and *compress*

$s1 = 20$, $f1 = 256$ and uses less than 100 Kbytes of internal memory. The performance of the order-3-1-and-0 model is significantly better than that of the order-2-and-0 method which, in turn, provides significant gains over the state-of-the-art *compress*. The order-3-1-and-0 algorithm reduces a file to an average of 35 percent of its original size, while the order-2-and-0 method reduces

File	**Original** **Size**	**Order** **3-1-0**	**FG**	**PPMC**
bib	111261	2.49	2.90	2.11
book1	768771	3.03	3.62	2.48
book2	610856	2.69	3.05	2.26
geo	102400	5.14	5.70	4.78
news	377109	3.22	3.44	2.65
obj1	21504	4.10	4.03	3.76
obj2	246814	3.33	2.96	2.69
paper1	53161	2.87	3.03	2.48
paper2	82199	2.85	3.16	2.45
pic	513216	0.90	0.87	1.09
progc	39611	2.87	2.89	2.49
progl	71646	2.06	1.97	1.90
progp	49379	2.12	1.90	1.84
trans	93965	2.02	1.76	1.77
Averages		2.84	2.95	2.48
Memory (Kbytes)		100	186	500

Table 2

Comparison of order-3-1-0, FG, and PPMC

files to 39 percent of original size on average, and *compress* leaves 45 percent of the original size.

We compare the performance of our order-3-1-and-0 model to the performance of algorithms FG and PPMC in Table 2. The data for FG and PPMC is taken from [BCW90]. The compression performance of our method is superior on average to that provided by algorithm FG. The order-3-1-and-0 model requires only 20 percent as much internal memory as algorithm PPMC and half as much as FG. Without tuning, the speed of the order-3-1-and-0 algorithm is superior to that of PPMC, and we expect to achieve speed comparable to that of FG when we take advantage of optimization techniques such as the use of registers and incorporating assembly-language code.

6. The Future of Context Modeling

Context modeling is a relatively new and very promising method for data compression. Early context modeling algorithms require large amounts of runtime memory and execute slowly. Algorithm PPMC, for example, provides excellent compression when 500 Kbytes of memory are available and speeds of 2000 cps are adequate. Our work provides an alternative to PPMC for applications in which 500 Kbytes of internal memory is not a reasonable requirement. Our order-3-1-and-0 method achieves much of the compression performance of PPMC without the large memory requirement (in fact it requires only one-fifth as much run-time memory). Our method has the additional advantage of executing much faster. We are able to achieve compression factors of less than 2.4 bits per character for source code files and less than 2.8 bits per character for a large variety of file types using less than 100 Kbytes of internal memory. The order-2-and-0 algorithm of Section 3 achieves respectable compression using only 48 Kbytes of internal memory. The compression performance of this method is superior to that of *compress* and uses less than 10 percent as much memory.

The work we describe in Sections 3 and 4 is applicable to context models of any order. The use of self-organizing lists and hashing provides a means of representing context models of any order in any available amount of memory. The restriction on internal memory and the values of the parameters (list sizes, hash table sizes, blending method, etc.) must be carefully balanced so as to achieve satisfactory performance in terms of compression ratio and execution speed. We have conducted limited experiments in order-4 context modeling. We have not yet identified a combination of parameters that provides performance that is consistently superior to that of our order-3-1-and-0 algorithm, given approximately the same restrictions on the use of internal memory. It is possible that with a different selection of parameter values an order-4 model may provide improved performance.

Finite-state modeling is an extension of finite-context modeling that permits exploitation of characteristics of the input that cannot be represented in finite-context models. For instance, finite-state models can represent information such as "every fourth character in the file is a zero" or "every sequence of *a*'s has even length". Horspool and Cormack describe an adaptive finite-state model DMC (for dynamic Markov compression) [HC86, CH87]. Due to the way in which states are added to the model, however, DMC does not attain the potential power of finite-state models. Bell and Moffat show that the DMC model is equivalent to a finite-context model [BM89].

Thus, the increased power of the finite-state model is attractive but, to date, successful use of this increased potential has eluded researchers. Developing methods of constructing finite-state models dynamically is an open problem the solution to which has considerable value. It is likely that straightforward methods of representing finite-state models, when they are developed, will consume large amounts of memory like the early context models. Representing finite-state models in limited memory is another challenging open problem.

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.

[BM89] Bell, T. and Moffat, A. A note on the DMC data compression scheme. *Comput. J. 32*, 1 (Feb., 1989), 16–20.

[BSTW86] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr., 1986), 320–330.

[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.

[HC86] Horspool, R. N. and Cormack, G. V. Dynamic Markov modelling — a prediction technique. *Proc. International Conference on the System Sciences*, Honolulu, HA. (Jan., 1986).

[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.

[LR82] Langdon, G. G. and Rissanen, J. J. A simple general binary source code. *IEEE Trans. Inf. Theory 28*, 5 (Sept., 1982), 800–803.

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

[L91] Lelewer, D. A. *Data Compression on Machines with Limited Memory*, Ph. D. dissertation, Department of Information and Computer Science, University of California, Irvine, 1991.

[LH91] Lelewer, D. A. and Hirschberg, D. S. Streamlining Context Models for Data Compression. *Proc. Data Compression Conference*, Snowbird, Utah (Apr., 1991), 313–322.

[Mo89] Moffat, A. Word-based text compression. *Software - Practice and Experience 19*, 2 (Feb., 1989), 185–198.

[RL81] Rissanen, J. J. and Langdon, G. G. Universal modeling and coding. *IEEE Trans. Inf. Theory 27*, 1 (Jan., 1981), 12–23.

[R82] Roberts, M. G. *Local Order Estimating Markovian Analysis for Noiseless Source Coding and Authorship Identification*, Ph. D. dissertation, Computer Science Dept., Stanford Univ.,, Stanford, 1982.

[St88] Storer, J. A. *Data Compression: Methods and Theory*, Computer Science Press, Rockville, Md., 1988.

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

[W91a] Williams, R. N. *Adaptive Data Compression*, Kluwer Academic Publishers, Boston, 1991.

[W91b] Williams, R. N. An extremely fast Ziv-Lempel data compression algorithm. *Proc. Data Compression Conference*, Snowbird, Utah (Apr., 1991), 362–371.

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