Information Processing & Management Vol 25, No. 2, pp 151–160, 1989
Printed in Great Britain

0306-4573/89 $3.00 + .00

PREDICTIVE ENCODING IN TEXT COMPRESSION

TIMO RAITA and JUKKA TEUHOLA
Department of Computer Science, University of Turku
Lemminkäisenkatu 14, SF-20520 Turku, Finland

Abstract—In predictive text compression the characters are encoded one by one on the basis of a few preceding characters. The usage of contextual knowledge makes the compression more effective than the plain coding of characters independently of their neighbors. In the simplest case we merely try to guess the next character, and the success/failure is encoded. Generally, the preceding substring determines the probability distribution of the successor, providing a basis for encoding. In this article, three compression methods of increasing power are presented. Special attention is paid to the trade-off between compression gain and processing time. As for speed, hashing turns out to be an ideal technique for maintaining the prediction information. The best gain is achieved by applying the optimal arithmetic coding to the successor information, extracted from the dependencies between characters.

1. INTRODUCTION

Data compression means a reversible manipulation technique that reduces the size of the data in such a way that the original form of the data can be later recovered. The reduction is possible if the data involve *redundancy* in some form. In the case of textual data, redundancy is partly caused by a nonminimal character domain and their nonuniform distribution within the text, and partly by the *dependencies* between characters.

There are two important application areas of compression. The first is the storage of large amounts of data in secondary storage. The second expanding area is data communication in computer networks, where substantial savings of time and money can be achieved through compression. With specialized hardware the encoding/decoding times can be kept tolerable, especially if the applied algorithms are simple enough.

Compression methods can be classified in several ways. The features a method should have are primarily dictated by the data type and the usage environment. For communication purposes, the most appropriate is an *online* method, which accomplishes the compression in one pass. For textual data, the characters can be thought to be generated sequentially by a source and encoded on the fly. In *offline* methods the entire input string may be processed several times before the final encoding strategy is determined. If the strings to be compressed have stable features (e.g. English text), the coding scheme can be *fixed* in advance and embedded in the coding procedure.

Another categorization can be made to *static* and *adaptive* compression schemes. In static schemes the rules used to encode the string are kept fixed during the process, but may vary from text to text. This implies that a static method is either offline or fixed. Adaptive methods change the rules according to the local characteristics of the text. This is normally implemented as an online *learning* process.

In the following section, we concentrate upon adaptive online compression methods for text. The article is organized as follows. Section 2 introduces the prediction principle and reports some previous works on the subject. In Section 3 we present three predictive compression methods of increasing power and complexity, together with argumentation for the choices made. Test results are given in Section 4 and the article is closed with conclusions in Section 5.

An early version of this work was presented during the New Orleans ACM SIGIR meeting, June 3–5, 1987, and appeared as "Predictive text compression by hashing" on pages 223–233 in *Proceedings of the Tenth Annual International ACM SIGIR Conference on Research & Development in Information Retrieval*, edited by C.T. Yu and C.J. van Rijsbergen. This final version was submitted March 31, 1988.

2. PREDICTION PRINCIPLE

One way to take advantage of the dependencies between successive characters in a text is to apply so-called *prediction* in the compression. The prediction principle was studied already by Shannon [1], whose test persons tried to guess successive characters of English sentences. The experiments showed that the information content of a character is very low; with a 26-character alphabet there is only about one bit per character. Thus, the potential savings in compression are considerable, though we cannot load all the contextual knowledge of a human into the compression program.

In predictive compression the characters are encoded sequentially, using the knowledge of a few preceding characters, the *prediction block*. The ideal is that we can uniquely guess what the next character will be, because then it need not be stored (assuming that the decompressor is equally good in guessing). In practice this is not always possible, so the idea must somehow be generalized. Some knowledge must be maintained about the distribution of possible successors for each block. These statistics must be gathered from the source text, unless they are generally known (the fixed case). In an online method the statistics are gathered along with the actual encoding, starting from an empty state and then gradually learning the properties of the text. The *decompression* also proceeds sequentially. At each step the characters of the prediction block have been decoded earlier, providing a basis for the current guess, identical to the encoding phase. Hence, the compression scheme is reversible, provided that the coding method itself is unique.

From an information-theoretic point of view, the input string to the encoder can be thought to be generated by a Markov process. In a simple approach the order (length of the prediction block) of the process is fixed and only one character is created at a time. This view will be taken in the methods to be presented.

An early application of Shannon's ideas to predictive compression was presented by Ott [2]. His Markov model can be used in a situation where the probabilities of the input symbols are fixed and known in advance. A more practical solution to the problem was suggested by Mommens and Raviv [3]. They took a first-order Markov process as their model and generated different Huffman codes for the successors of each character. The disadvantage of this approach is the large number of coding trees to be stored within the compressed data.

Teuhola and Raita [4,5] use a fixed-order Markov process, together with fast but nonoptimal encoding of the successor. Cleary and Witten [6] apply a variable-order model in which the prediction block length (with a predetermined maximum) can be adapted for each position. Combined with Arithmetic coding, Cleary and Witten have attained a result where only 2.2 bits per character are needed on the average for English text. The processing times are not reported, but they are presumably rather high, due to the complexity of the method.

Cormack and Horspool [7] manipulate the data on the bit level by trying to group bits to form logical units. The model is presented by a graph structure, which enables one to extract far-reaching dependencies between input symbols, for example those resulting from the syntax rules of the language.

The compression gain is high when the distribution of the successors is skew. As stated in [4], the number of *different* character blocks of length i ($i = 1, 2, 3, \ldots$) grows very slowly with the length of the source text, compared to all possible combinations. Correspondingly, the average number of different successors for a block decreases rapidly when the block length grows. Moreover, the proportions of unique and most probable successors were found to be very high, making the distribution extremely skew. The dependencies between successive characters may be stronger at some positions than others. According to Guiasu [8], they can extend even to 30 characters, but it is evident that they weaken noticeably at word boundaries. The varying amount of redundancy suggests that it could be advantageous to adapt the order of the model according to the position in the text.

It is not easy to analyze the performance of predictive methods because the Markov model does not give any results unless we can fix the distribution of the conditional suc-

cessor probabilities. There exist some excellent papers that deal with the theoretical analysis of string compression [9–14]. However, they mainly discuss the effectiveness of methods that substitute fixed or variable length substrings for code words. It seems that the derived results cannot be directly applied to the Markov model-based schemes. A good overview on the use of predictive encoding for text and digital images is given in [15].

3. THREE PREDICTION ALGORITHMS

Let us start with a motivation and argumentation for the methods to be presented. Almost all compression methods in the literature have a single objective: maximization of compression gain. However, in practice the *space* is not always the only target of optimization. *Time* is often equally important, especially in data communication, where the total time consists of the encoding/decoding time plus the actual transfer time. Because of the trade-off between these factors, it is obvious that we should look for simple and fast methods offering a reasonable, not necessarily the best possible, compression gain.

The first decision to be made regards the data structure maintained for representing the prediction information, in our case the prediction blocks and their successors. Most of the available methods use a *trie* structure for storing text fragments, see e.g. [4,6]. Here we apply a much faster technique, namely *hashing*, proposed already in [5]. The prediction block is converted to a number, from which the home address is calculated. The successor information is stored in the home address (or more generally, in a structure referenced by a pointer in the home address). The technique of using hashing can be regarded as a generalization of the substring test suggested in [16]. It also resembles the idea of *signatures* [17], used mainly for partial-match retrieval purposes.

There are two aspects that make hashing especially suitable here:

1. *Collisions can be ignored.* Their only effect is to deteriorate the compression gain slightly because the successors of two or more different prediction blocks are mixed. However, if the table is not too full, collisions are so rare that they do not play a significant role. For online methods (such as the ones below) we can choose the size of the hash table arbitrarily because the table need not be stored in the compressed data. This implies that the proportion of collisions can be made arbitrarily small.
2. The hash table is *more compact* than the trie because the prediction blocks need not be represented explicitly. In offline methods the prediction information must be stored within the compressed data and therefore its size must be carefully optimized. The hash table has a clear advantage, although nice techniques for compacting the trie have also been devised [4,18]. In online methods the size of the table/trie is not so important but, as noticed in [6], for very long blocks the trie may be too large to fit in the main memory.

We now give detailed descriptions of three hash-based algorithms of increasing compression gains but also increasing execution times. Further details related to the methods are discussed in Section 4.

3.1 *Method A*

Given a string $S_1 \ldots S_i$ (prefix of the input string $S = S_1 \ldots S_n$), a block length $k \leq i$ and a block $P = S_{i-k+1} \ldots S_i$, we predict that S_{i+1} is the same as the successor of the previous occurrence of P before this. Thus, there should be $j < i$ such that $P = S_{j-k+1} \ldots S_j$, and the predicted successor of S_i is S_{j+1}. The success/failure is encoded as a bit within the string $B = B_{k+1} \ldots B_n$. In the failure cases we store the actual successor in a list L of *left-over characters*. If P has not occurred before, we have no basis for guessing (except perhaps a default guess, e.g. space). This situation is interpreted as a normal failure. B and L constitute the compressed file.

A hash table $T = T_0 \ldots T_{m-1}$ (array of characters) is maintained to keep track of the

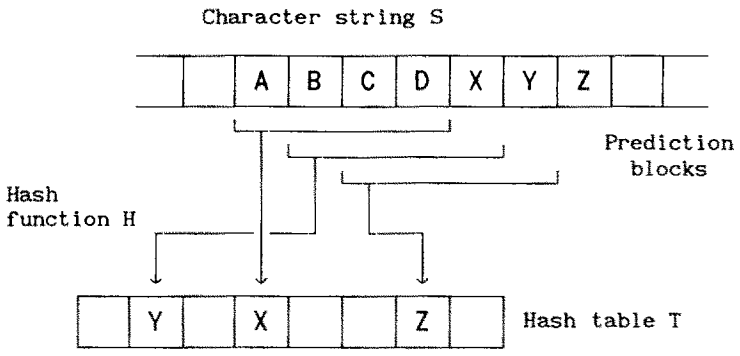

Fig. 1. The data structure for maintaining the latest successors of blocks in Method A.

successors. For each i from k to $n-1$ we store S_{i+1} into the slot indexed by $H(S_{i-k+1} \ldots S_i)$, where H is the hash function (see Fig. 1).

Algorithm A

A1. Initialize T with default characters.
A2. Store the first k characters (starting block) of S in L.
A3. For the rest ($i = k+1, \ldots, n$) do the following:
  A3.1. Calculate the hash address $a := H(S_{i-k} \ldots S_{i-1})$.
  A3.2. The predicted character is T_a.
    If $T_a = S_i$, then prediction succeeds; set $B_i := 1$.
    Otherwise, set $B_i := 0$, store S_i in L and set $T_a := S_i$ (update of successor).
A4. Output B and L as the compressed data.
End of A.

The algorithm was presented as if the results were collected in the main storage during the processing and then thrown out to the compressed file at the end. In practice we minimize the main storage consumption and internal moves. Especially in data communication applications the output must be generated on the fly, i.e. the success/failure bits are interleaved with the leftover characters in the same order as they are generated.

The decompression is obvious, but we present it here for completeness. Note that the hash table is maintained identically to the compression phase.

Algorithm A′

A′1. Retrieve B and L from the compressed data.
A′2. Initialize T with default characters.
A′3. Fetch the first k characters from L to S.
A′4. Determine the rest of S ($i = k+1, \ldots, n$) as follows:
  A′4.1. Calculate the hash address $a := H(S_{i-k} \ldots S_{i-1})$.
  A′4.2. If $B_i = 1$, then set $S_i := T_a$ (prediction had succeeded).
    Otherwise, get S_i from L and set $T_a := S_i$ (update successor).
End of A′.

Algorithm A does not necessarily predict the most probable global successors, but behaves in a more local manner; it is based on the hypothesis that two successive occurrences of the same block also have the same successors. This is often a reasonable strategy because different parts of the text may have different characteristics and the method accommodates these variations dynamically.

3.2 *Method B*

Method A is truly predictive in the sense that at each point an explicit guess is made, which may succeed or fail. We now take a more general view, where we actually predict

that the successor conforms to a certain distribution, containing all possible successors for the related block. The distribution is here presented very roughly: we only want to estimate the relative probability of the possible successors. An ordered list is maintained, and the order number of the actual successor in this list is recorded in the compressed form. As stated in the introduction, the distribution of successors is very skew. Hence, the compression task is reduced to *encoding of small integers.*

Technically, we implement the above idea as follows (see Fig. 2). A hash table is again used, now to maintain the *successor lists* of blocks. Each list contains the successors of the relevant block (and its synonyms) that have occurred up to the current character. Because no frequencies are counted, the lists are maintained by applying the *move-to-front* principle [19,20], i.e. the latest successor is always made to be the most probable guess for the successor of the next occurrence of the same block. This keeps the lists roughly in probability order. Note that this is in fact a generalization of Method A, which also used the move-to-front strategy.

The actual encoding of the successor is done by encoding its order number in the list. The coding technique should have the following property: if $i < j$ then length(code(i)) $\leq$ length(code(j)), because the more probable characters should get shorter codes. The coding cannot be guaranteed to be optimal because we do not know the exact distribution. If the successor has not occurred before, it cannot be found in the list. Therefore we assume a *virtual continuation* for each list, containing all other characters in the alphabetic order. The order numbers are here encoded using a modification of the Golomb method [21], presented by Teuhola [18]. It is a fast data-independent technique for encoding very skewly distributed items.

Algorithm B

B1. Initialize the hash table T with NIL pointers.
B2. Store the starting block of k characters in the compressed file.

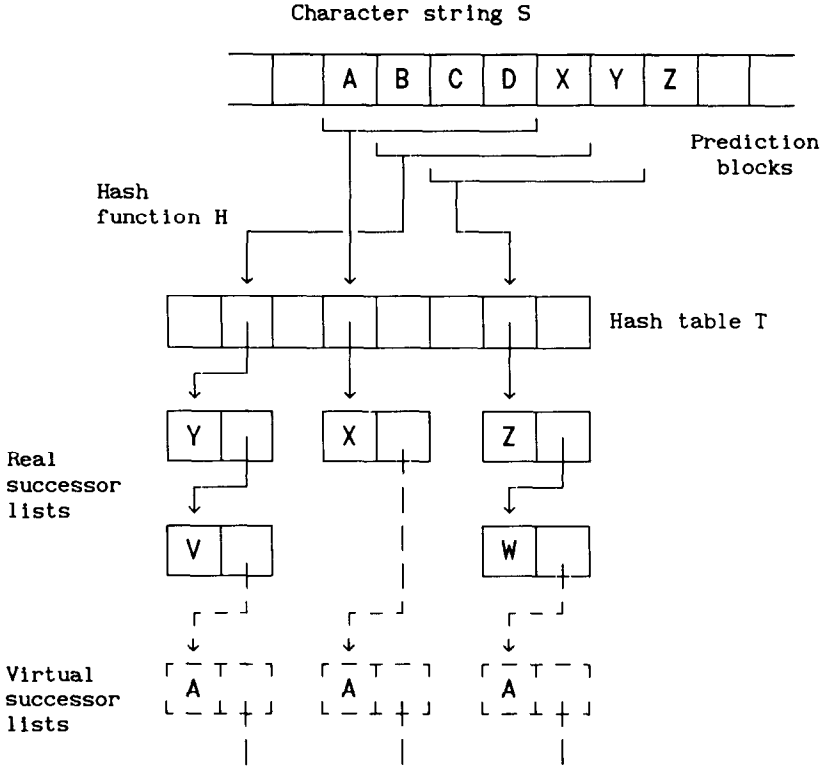

Fig. 2. The data structure for maintaining the successors of blocks. The virtual lists represent the rest of the alphabet and are not stored anywhere.

B3. For $i = k + 1, \ldots, n$ do the following:
  B3.1. Calculate the hash address $a := H(S_{i-k} \ldots S_{i-1})$.
  B3.2. Search S_i from the list starting at T_a. If it was found, determine its order number O_i in the list, otherwise do the following:
    - Scan the virtual list up to character S_i.
    - $O_i :=$ length of the "real" list plus the order number of S_i in the virtual list.
    - Create a new real list item containing S_i. (S_i disappears from the virtual list.)
  B3.3. Encode O_i (e.g. using the extended Golomb method) and store the code in the compressed file.
  B3.4. Move the list element containing S_i to the front of the list starting at T_a.
End of B.

The decompression algorithm for B is analogous to compression and so is omitted.

3.3 *Method C*

By maintaining the frequencies of successors we get a more precise estimate of the actual successor probabilities for each block. The frequencies can be stored in the list elements together with the corresponding character. The total frequency of the list items is stored in the list head (Fig. 3). The same data-independent coding of successor numbers as in Method B could be applied again, by keeping the lists in frequency order. However, because we now have more information available, we should also take advantage of it. We propose here using *arithmetic coding* [22] because it is optimal and can easily be applied to a dynamic case, where the probabilities of characters change.

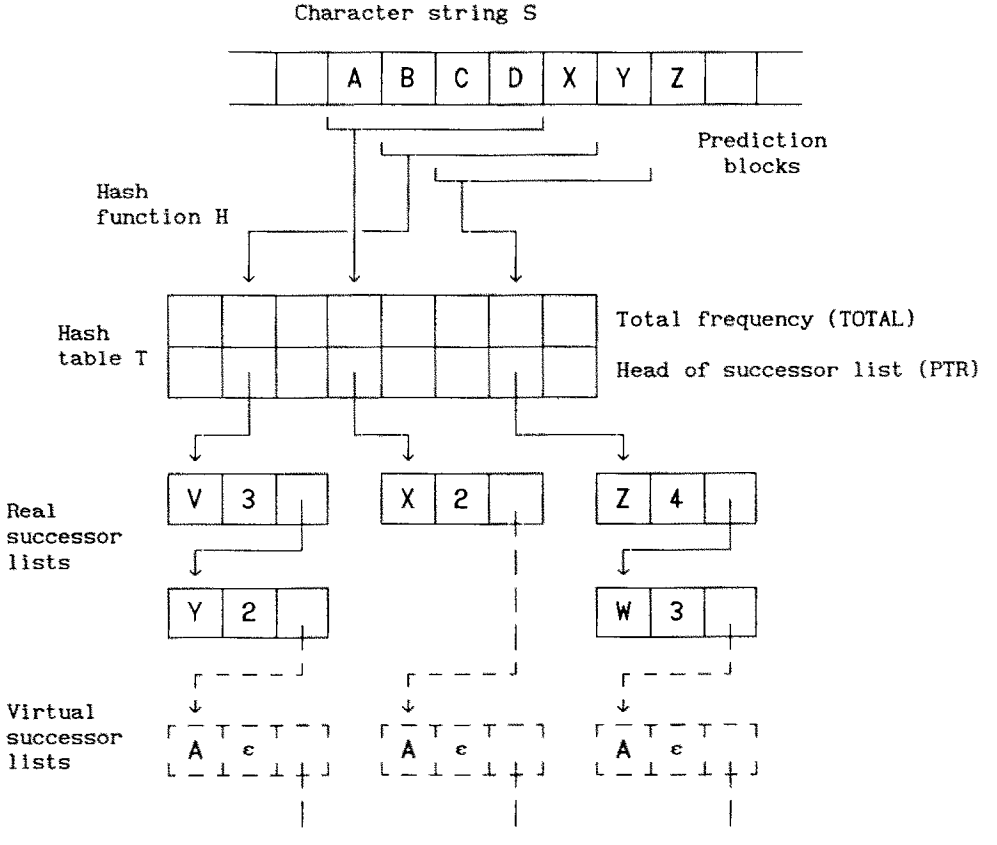

Fig. 3. The data structure for Method C. The successor nodes contain the characters and their frequencies.

With these choices we come very close to the method presented by Cleary and Witten [6]. The differences express our endeavor to make the technique fast enough for practical purposes. Besides hashing, the main difference is the treatment of successors that have not occurred earlier in the text. Cleary and Witten shorten the prediction block until the successor is found in the earlier statistics (or the block is reduced to null). Each shortening step is expressed with an escape character, which is assigned an estimated probability. We, instead, stick to the fixed block length, but apply the same technique as in Method B: behind the real successor list we assume a virtual list containing the rest of the alphabet. However, we cannot assign them a zero frequency because then the arithmetic coding could not be applied. Thus, a small nonzero frequency ϵ is assumed for all characters in each virtual list, distorting the distribution somewhat in the beginning. The initial frequency (ϵ) is a parameter of the method and should be optimized empirically. In order to be able to use integers in arithmetic coding [23], we choose $\epsilon = 1$ and apply a suitable weight for the frequency increments.

Algorithm C

C1. Initialize the hash table T by setting: $T_i.TOTAL := \epsilon *$ (size of alphabet) and T_i.PTR := NIL ($i = 0, \ldots, m-1$).
C2. Store the starting block of k characters in the compressed file by using arithmetic coding for uniformly distributed characters.
C3. For $i = k+1, \ldots, n$ do the following:
  C3.1. Calculate the hash address $a := H(S_{i-k} \ldots S_{i-1})$.
  C3.2. Scan the list starting at T_a.PTR until the character S_i is found (either from the real or the virtual list). Denote the list element by E. During scanning, maintain the cumulative frequency C of the successors (excluding S_i).
  C3.3. Apply arithmetic coding to the interval [C/T_a.TOTAL, ($C + E$.FREQ)/T_a.TOTAL] and move the solved bits to the compressed file.
  C3.4. If E was virtual, create a real node for it. (E disappears from the virtual list.)
  C3.5. Increase T_a.TOTAL and E.FREQ by the increment value.
  C3.6. Move element E to its correct position in the frequency order within the successor list.
C4. Finish the arithmetic encoding.
End of C.

Step C3.6 is not necessary, but it reduces the scanning of lists by keeping them in the order of probability. Compared to Method B, the above algorithm is, though still simple, much slower due to the maintenance of frequencies and the abundance of arithmetic operations in the coding.

The decompression for Method C is again analogous to compression and is therefore omitted.

4. IMPLEMENTATION AND RESULTS

There are still some open technical details to be decided. Related to hashing, a good hash function should be found. Fortunately, the simple *remainder-of-division* technique has proved to be the most successful, so we apply it also here (see [24,25]). For prediction blocks longer than a machine word, some kind of *folding* technique could be used to shrink the block to fit into one word. Fortunately, the optimal block size almost never exceeds 4. When performing successive hashing of partially overlapping blocks, a *window* containing the current block should be implemented to pass over S. Clearly, some assembler-level *shift* operation would be the most efficient implementation for this purpose.

The most critical factor in the methods is the size (k) of the prediction block. The actual optimum is method- and text-specific, and can be determined only experimentally, as in [5], where $k = 3$, or in some cases $k = 2$, was found to give the best results for files of moderate size. For small files even $k = 1$ is sufficient, due to the quicker convergence of the online methods. A sophisticated compression utility program should be able to

choose a good block size on the basis of the file size and file type. It is clear that very small files cannot be compressed much without some prior knowledge about the properties of their contents.

The other important parameter to be decided is the size (m) of the hash table. Because all three methods are of the online type, the table can be discarded after compression. Thus, we should choose a "sufficiently" large table to reduce collisions. On the other hand, for too large a table the initialization takes a long time. As stated in [4], the average number of blocks of size 3 is about 2,000–3,000 in a typical English text, so a hash table of size 10,000 should be sufficient in most cases. From [5] we see that even the size 2,000 is sufficient, deteriorating the compression gain only by a couple of percent. In the tests we used the size 49,999, for security.

Concerning Method C, various frequency increments were tested to find an "optimum." It is evident that small values lead slowly to a steady state in the lists. On the other hand, large values result in excessive bit strings for characters that have not occurred before as successors (i.e. are found from the virtual list). In the test runs, we started from the initial value 1 and increased the weighted frequency of a successor by 100 for each occurrence. In terms of the earlier description this means that the value of ϵ was 0.01.

In the algorithms above we used the variable n for the length of the string, but did not tell where its value comes from. For disk files the system maintains the length in the directory. Because this holds for both original and compressed files, the length need not be stored within the compressed data. However, in a pure online application we do not know the length of the string at the start. The problem is peculiar to data transmission in general, not only to compression. The normal solution is to collect and send the data in fixed-size packets, equipped with a frame containing a.o., the amount of data in the packet.

There are two main categories of text: formal and natural languages. For our experiments we chose one from each to be compressed: Pascal and English. Methods A, B, and C were tested for different sizes of files and for different lengths of prediction blocks. An adaptive online version of the arithmetic coding [23] was also used, for comparison, but applying the normal frequency increment 1. The results are gathered in Table 1. The measure of compression is the average number of bits required per source character.

There are remarkable differences between the compression gains for Pascal and English texts: Methods A–C perform much better for Pascal. This can be explained by the

Table 1. Performance of the compression algorithms (the results are expressed in bits per source character)

Type of file	Length of file (char.)	Prediction block length	Method			
			A	B	C	Arith.
Pascal	10,000	2	3.71	2.87	2.52	5.09
		3	3.46	3.07	2.61	
		4	3.64	3.41	3.00	
	20,000	2	3.66	2.51	2.09	4.77
		3	3.21	2.49	2.01	
		4	3.18	2.65	2.21	
	30,000	2	2.98	2.26	1.76	4.27
		3	2.75	2.36	1.73	
		4	2.85	2.56	1.94	
English	10,000	2	5.13	3.70	3.38	4.63
		3	4.58	3.86	3.71	
		4	4.73	4.29	4.23	
	20,000	2	5.51	3.78	3.30	4.60
		3	4.68	3.75	3.45	
		4	4.73	4.19	4.01	
	30,000	2	5.42	3.62	3.07	4.60
		3	4.58	3.48	3.14	
		4	4.56	3.88	3.64	

regularity of the character sequences in a formal language. The Arithmetic coding cannot take advantage of this; its compression gain is dependent only on the character distribution.

Because we are also interested in execution times, we show them in Table 2, measured using a DEC-2060 computer, with programs written in Pascal. The source programs were written in a straightforward manner, without special optimization, except that the "optimize" option of the compiler was used. The decompression times are also shown in Table 2 and they are, as expected, very close to the compression times. This is due to the online nature of the methods, because the same data structures must be maintained and identical steps performed in both phases. For compactness, Table 2 shows the times only for optimal choices of the prediction block length k.

Method A, though having the lowest compression gain, seems to be the most promising candidate for data communication purposes. From Algorithm A we see that the inner loop (A3.1–A3.2) contains only the following basic operations:

1. Move the window to the next block, i.e. shift left the register containing the window and put the next character to the rear of the window.
2. Calculate the hash address, i.e. remainder of division.
3. Send one bit.
4. In unsuccessful cases send one character and update one table element.

Step 1 was implemented with the modulo operation, ord function, multiplication, and addition. If the inner loop is carefully programmed using a low-level language and machine registers, its execution takes only a few microseconds per character. Evidently, a hardware realization would be faster still.

Methods B and C, though not slow, can be better recommended for the compression of files in secondary storage. Nevertheless, the speed is important because the compression/decompression time should not be much higher than the time saved in the reduced data transfer.

5. CONCLUSIONS

Three text compression algorithms were presented, exemplifying different approaches to the prediction principle. An essential technical feature in all of the methods was hashing, used as a search method for the successors of prediction blocks.

The methods illustrate nicely the observation of Rissanen and Langdon [12]: encoding consists of two separate actions. First, a model and its parameters must be created from the source data. Our model is pure Markovian and the parameters are extracted either from the order of the nodes of the successors or from the weights attached to them. Second, the actual coding method can be chosen arbitrarily to adapt to the parameters. We have taken two examples: the extended Golomb and the arithmetic coding. The latter was combined with the most space-efficient version, but from the results we can conclude that a large proportion of the compression gain is due to the dependencies between successive characters.

Table 2. CPU times (s) of compression and decompression experiments

Type of file	Length of file	Method							
		Compression time				Decompression time			
		A	B	C	Arith.	A	B	C	Arith.
Pascal	10,000	0.9	3.6	9.8	6.0	0.9	3.6	9.8	6.5
	20,000	1.4	5.6	13.5	11.3	1.3	5.9	13.6	12.3
	30,000	1.8	7.8	19.2	16.1	1.8	8.1	18.9	17.0
English	10,000	0.9	4.6	9.8	5.5	0.9	4.5	9.9	5.8
	20,000	1.5	9.1	17.2	10.8	1.4	8.5	17.0	11.4
	30,000	2.0	11.8	22.6	16.2	2.0	12.1	22.4	17.2

Obviously, the superiority of arithmetic coding with respect to the nonoptimal methods is marginal for many distributions.

Speed was an important goal in developing the methods. The observation that collisions can be ignored makes the hashing technique especially fast. Also, the usage of a nonoptimal final coding in Methods A and B was a deliberate choice aiming at reduction of the execution time.

Further variations of the same basic ideas can be found in [5], exemplifying also offline methods. In [26] the length of the prediction block and the number of predicted characters are made variable, producing further improvements to the compression gain.

REFERENCES

1. Shannon, C.E. Prediction and entropy of printed English. Bell System Technical Journal, 30: 50–64; 1951.
2. Ott, G. Compact encoding of stationary Markov sources. IEEE Transactions on Information Theory, IT-13(1): 82–86; 1967.
3. Mommens, J.H.; Raviv, J. Coding for data compaction. IBM Research Report RC5150, T.J. Watson Research Center, Yorktown Heights, N.Y.; 1974.
4. Teuhola, J.; Raita, T. Text compression using prediction. ACM 1986 international conference on research and development in information retrieval; September 1986; 97–102; Pisa, Italy.
5. Raita, T.; Teuhola, J Predictive text compression by hashing. ACM 1987 international conference on research and development in information retrieval, June 1987; 223–233; New Orleans.
6. Cleary, J.G.; Witten, I.H. Data compression using adaptive coding and partial string matching. IEEE Transactions on Communications, COM-32(4): 396–402; 1984.
7. Cormack, G.V.; Horspool, R.N.S. Data compression using dynamic Markov modelling. Research report CS-86-18. Waterloo, Ontario: University of Waterloo; 1986.
8. Guiasu, S. Information theory with applications. London: McGraw-Hill; 1977.
9. Lempel, A.; Ziv, L On the complexity of finite sequences. IEEE Transactions on Information Theory, IT-22(1): 75–81; 1976.
10. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression IEEE Transactions on Information Theory, IT-23(3): 337–343; 1977.
11. Storer, J.A. Data compression: Methods and complexity issues. Ph.D. Thesis. Princeton, NJ: Department of Electrical Engineering and Computer Science, Princeton University; 1978.
12. Rissanen, J.; Langdon Jr., G.G Universal modeling and coding. IEEE Transactions on Information Theory, IT-27(1): 12–23; 1981.
13. Cleary, J.G.; Witten, I.H. A comparison of enumerative and adaptive codes. IEEE Transactions on Information Theory, IT-30(2): 306–315; 1984.
14. Storer, J.A. Textual substitution techniques for data compression. In: Apostolico, A , Galil, Z , editors Combinatorial algorithms on words. Heidelberg: Springer, 1984: 111–129.
15. Witten, I.H.; Cleary, J.G. Foretelling the future by adaptive modeling. ABACUS, 3(3): 16–73; 1986.
16. Harrison, M.C. Implementation of the substring test by hashing. Communications of the ACM, 14(12): 777–779; 1971
17. Faloutsos, C.; Christodoulakis, S. Signature files: An access method for documents and its analytical performance evaluation. ACM Transactions OIS, 2(4): 267–288; 1984.
18. Teuhola, J A compression method for clustered bit-vectors. Information Processing Letters, 7(6): 308–311; 1978.
19. Hester, J.H.; Hirschberg, D.S. Self-organizing linear search. ACM Computing Surveys, 17(3): 295–311; 1985.
20. Sleator, D.D.; Tarjan, R.E. Amortized efficiency of list update and paging rules. Communications of the ACM, 28(2): 202–208; 1985.
21. Golomb, S.W. Run-length encoding. IEEE Transactions on Information Theory, IT-12: 399–401; 1966.
22. Rissanen, J.; Langdon Jr., G.G. Arithmetic coding. IBM Journal of Research and Development, 23(2): 149–162; 1979.
23. Witten, I.H.; Neal, R.M.; Cleary, J.G. Arithmetic coding for data compression. Communications of the ACM, 30(6): 520–540; 1987.
24. Knuth, D.E. The art of computer programming, vol. 3: Sorting and searching. Reading, MA: Addison-Wesley; 1973.
25. Maurer, W.D.; Lewis, T.G. Hash table methods. ACM Computing Surveys, 7(1): 5–19; 1975.
26. Raita, T. Generalized coding algorithms for predictive text compression In preparation; 1988.