Experiments with a Block Sorting Text Compression Algorithm

Peter Fenwick

Technical Report 111

ISSN 1173-3500

17 May 1995

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand

peter-f@cs.auckland.ac.nz

Abstract.

This report presents some preliminary work on a recently described "Block Sorting" lossless or text compression algorithm. While having little apparent relationship to established techniques, it has a performance which places it definitely among the best-known compressors. The original paper did little more than present the algorithm, with strong advice for efficient implementation. Here, the algorithm is restated in data compression terms and various measurements are made on aspects of its operation.

Consideration of the possible efficiency of text compression leads to the revival of ideas by Shannon as the basis of a text compressor and then to the classification of the Block Sorting compressor as an example of this "new" type. Finally, this work leads to a reconsideration of the meaning of escape codes in PPM-style compressors and a suggested technique for better estimating escape probabilities.

This report is available by anonymous FTP from

`ftp.cs.auckland.ac.nz /out/peter-f/report111.ps`

1. Introduction.

A very recent development in text compression is a “Block Sorting” algorithm, published by Burrows and Wheeler[4]. It considers the text in blocks, which may be as large the entire file, reorders the text according to an apparently bizarre algorithm and then compresses that text with a Move-to-Front and Huffman compressor. The compression performance is comparable with that of the best high-order statistical compressors. Cleary et al [5] have shown that the overall algorithm is equivalent to a PPM-style compressor, operating with unbounded order. The realisation is however utterly different from any of the traditional text compressors and raises some interesting questions, including —

- how does the compressor relate to other, better-known, compressors,
- are MTF and Huffman the best operations in this situation,
- might they be replaced by alternatives,
- what are the statistics of the symbols which are actually compressed?

Work has been done on improving the basic Block-Sorting algorithm, and it was originally intended to be included in this report. However, that work is not complete, and the initial investigations and other related work justified a preliminary report. The present report emphasises observations on the nature of the compressor and some statistics of its coding parameters, together with some general comments which followed from thinking about the compressor. While results could have been presented which incorporate some recent improvements (especially in the sorting phase), it was felt best to defer those and report on only the simpler, direct results.

Detailed discussions on improving Block-Sort compression will be reported later.

2. The Block-Sorting algorithm

Burrows and Wheeler present their algorithm in terms of matrix operations, an approach which has a certain elegance, but is far removed from the usual conventions of text compression. In this section we present the algorithm in text compression terms.

In normal statistical compression we consider each symbol of the file in relation to its preceding symbols or context. The inter-relations between symbols in the file means that it is possible to predict most symbols with a high degree of confidence. The limited choice of possible symbols within the context means that few bits are needed for the encoding and considerable compression is achieved. In general, increasing the context (or number of preceding symbols being considered) narrows the choice of possible symbols and improves the compression. A maximum context of about 4–8 symbols is appropriate for most files. Above that length, any improvement in actually coding the symbols themselves tends to be offset by the overheads in controlling and specifying the context; the compression remains constant or even deteriorates slightly.

The Block Sorting algorithm actually considers each symbol in relation to its *following* context, rather than the more conventional *preceding* context. (There is no reason why block sorting should not use a preceding context, but the following context is a natural consequence of usual sorting conventions.) Each symbol is then considered in relation to its following context; near the end of the file we can either wrap round cyclically to the beginning, or use a special EOF terminator. (Burrows and Wheeler introduce the method with a cyclic wrap-round, but later imply an EOF symbol.)

2.1 Compression

The text block to be compressed (part or all of the file) is first sorted according to the context of each symbol. (The sort key for a symbol is its following symbols, to whatever length is needed to resolve the comparison.) The output of this stage is a permutation of all the symbols of the original file, together with the position of the symbol whose context is the original input. (This position is required for the decoding step, as explained below.) At this stage we have done no compression at all, but we have collected together similar contexts. Because these contexts restrict the choice of preceding symbols, any region of the permuted file contains sequences of just the few symbols which appear within the similar contexts, the actual symbols of course varying according to the context. There is strong locality; if we have recently seen a symbol there is a high probability that that symbol will recur in the near future.

In their original paper Burrows and Wheeler capture this locality with a Move-To-Front compressor, with Huffman and perhaps run-length encoding of the output.

2.2 Decompression

Recovery of the data requires first a decompression to recover the output of the original sorting transformation. Reversing the permutation of these symbols depends on the observations that the recovered (or transmitted) data contains all of the original symbols and that sorting these transmitted symbols gives the first character of each of the sorted contexts. But the transmitted data is ordered according to the contexts, so the n-th symbol transmitted corresponds to the n-th ordered context, of which we know the first symbol. So, given a symbol s in position i of the transmitted text, we find that position i within the ordered contexts contains the j-th occurrence of symbol t; this is the next emitted symbol. We then go to the j-th occurrence of t in the transmitted data, occurring in position *k*, and obtain its corresponding context symbol as the next symbol. The need for the position of the symbol corresponding to the first context is now obvious; it locates the last symbol of the output and from there we can traverse the entire transmitted data to recover the original text.

	The forward transformation		the reverse transformation			
	symbol	context	Index	symbol	context	link
	e	compressiond	1	e	c..	8
First →	n	decompressio	2	n	d..	3
	d	ecompression	3	d	e..	1
	r	essiondecomp	4	r	e..	13
	s	iondecompres	5	s	i..	9
	o	mpressiondec	6	o	m..	10
	o	ndecompressi	7	o	n..	2
	c	ompressionde	8	c	o..	6
	i	ondecompress	9	i	o..	7
	m	pressiondeco	10	m	p..	11
	p	ressiondecom	11	p	r..	4
	s	siondecompre	12	s	s..	5
	e	ssiondecompr	13	e	s..	12

Figure 1. The forward and reverse transformations

To illustrate the operations of codingand decoding we consider the text “decompression” as shown in Fig 1[1] . (“compression” is a more obvious choice, but the initial index is 1, which tends to obscure some details of the data recovery.) The lexicographically first context is “`compressiond`” for symbol “`e`”, the second is “`decompressio`” for symbol “`n`”, and so on. The transformed text is then “`endrsoocimpse`”, and the initial index is 2, because the second context corresponds to the original text.

To decode we take the string “`endrsoocimpse`”, sort it to build the contexts and then build the links shown in the last column. In the links, the context “`c..`” links to the first (and only) occurrence of “`c`” in the input, here the 8-th symbol. Within the “`e..`” context, the first occurrence links to the first “`e`” in the input (index = 1) and the second links to the second occurrence (at 13). Similarly the two “`o`”s and the two “`s`”s link to their respective positions.

---

1 Burrows and Wheeler proceed by writing the text as the first row of a matrix, and then writing all possible cyclic rotations of the text as the other matrix rows. They sort the matrix by rows and use the last column of the sorted matrix as the to-be-compressed text. Putting the ‘symbol’ column to the right of the ‘contexts’ in Fig 1 yields the Burrows and Wheeler matrix (sorted).

To finally recover the text, we start at the indicated position (2) and immediately link to 3. The sorted received string there yields the desired symbol “d”. We then link to 1 get the “e”, and so on for the rest of the data stopping on a symbol count or EOF symbol.

3. Order-0 implementation and results

The algorithm was implemented very much as described by Burrows and Wheeler, but with an order-0 arithmetic coder replacing the Huffman coder of the original. Some aspects of the implementation are given in Appendix I; in particular it retains a relatively simple sort phase.

The immediate results are given in Table 1, testing on the Calgary Corpus and using PPMC as a reference compressor. (The compression values are in output bits per input byte.)

File		PPMC	bs Order0	MTF dist non-0	frac 0 MTF	compares	short	run	long
Bib	111,261	2.110	2.133	5.50	66.8%	953,647	499,267	0	423
Book1	768,771	2.480	2.523	3.88	49.8%	10,066,830	5,134,120	400	43
Book2	610,856	2.260	2.198	4.20	60.8%	7,295,435	3,649,435	386	2,295
Geo	102,400	4.780	4.812	55.63	35.8%	710,791	654,944	8,588	2,860
News	377,109	2.650	2.677	7.65	57.9%	3,794,914	2,220,768	101,264	33,458
Obj1	21,504	3.760	4.227	46.82	50.6%	120,686	47,001	49,775	39
Obj2	246,814	2.690	2.710	30.22	68.1%	2,012,735	1,107,942	16,611	10,586
Paper1	53,161	2.480	2.606	6.45	58.4%	381,283	233,918	634	77
Paper2	82,199	2.450	2.571	5.06	55.4%	711,529	424,242	6	29
Pic	513,216	1.090	0.919	3.39	87.4%	11,150,747	654,959	8,957,076	2,312,587
ProgC	39,611	2.490	2.666	8.32	60.3%	253,062	150,910	1,458	225
ProgL	71,646	1.900	1.839	4.63	72.9%	621,697	295,551	37,115	18,836
ProgP	49,379	1.840	1.821	5.54	74.0%	379,732	157,582	21,142	11,547
Trans	93,695	1.770	1.601	5.66	79.2%	736,952	338,303	24,215	20,575
AVG		2.482	2.522	13.78	62.7%				

Table 1. Results on Calgary Corpus, with arithmetic order-0 final encoder

The first columns give the file name, its size in bytes, and then the results for the PPMC compressor and the new “Block-Sort, order-0” compressor. The next column gives the average Move-To-Front distance, for those symbols which move, ie are not already at the head of the list. Then we have the fraction of the symbols which are already at the head of of the Move-To-Front list and are emitted as zeros. The final 4 columns give statistics on the comparisons of the sorting phase, and can be understood only by reference to Appendix I. The arithmetic compressor for these results was tuned by adjusting the frequency increment and limit to give a relatively fast response to changes; the final values are *increment*=16 and *limit*= 8192.

The most obvious result is that the compressor is already very good – within about 1.5% of the PPMC performance on average, and better on many of the more compressible files. The compression is clearly related to the proportion of symbols which are emitted as zeros, but the correlation is not exact. Nevertheless, most "text" files have around 60% of their symbols emitted as zeros; the relatively incompressible GEO has only 36%, while the more compressible PIC and TRANS are at 87% and 79%. A detailed log of these tests is given in Appendix II.

The distribution of the Move-To-Front frequencies is shown in Figure 2 for three of the files – GEO (incompressible), PAPER1 (representative text), and PROGP (quite compressible).

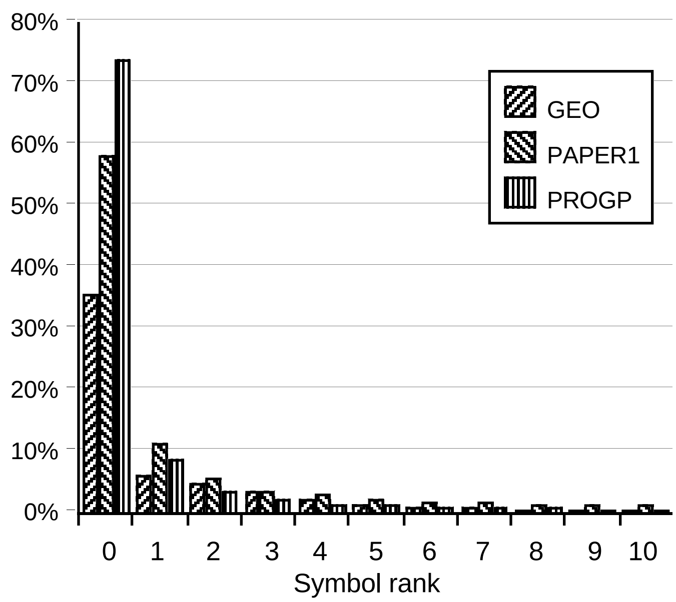


Fig 2. Order-0: probabilities of MTF symbols

We have already noted the preponderance of rank-0 symbols in the output; this Figure shows that the others have frequencies almost always less than 10% and usually less than 5%.

Figures 3 and 4 also show the rank frequencies, but with logarithmic probability scales and for the first 20 ranks and the first 128 (which includes all symbols of text files). PAPER1 and PROGP both show a steady and rapid decline in probability as the MTF rank increases, but the behaviour of GEO is quite different with many less frequent symbols having rank probabilities of about 0.002 – 0.004. This is in line with the expected probability of 0.0039 for a uniformly distributed population of 256 symbols. (GEO has, on average, a local context of perhaps 6–10 active symbols. Symbols of higher rank are chosen essentially at random.) The very low probabilities of Fig 4 are to some extent spurious. Not only are they very small to start with but the graph was drawn with 1 added to all counts to handle those counts which were actually zero.

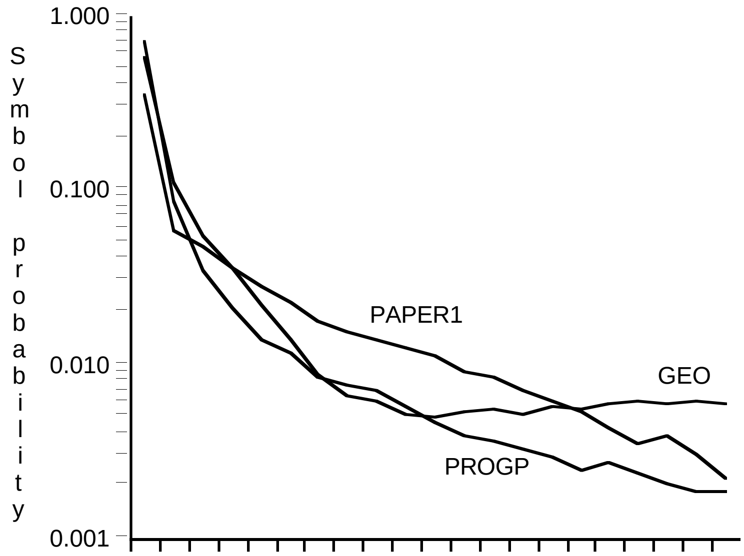


Fig 3. Probabilities of first 20 MTF codes

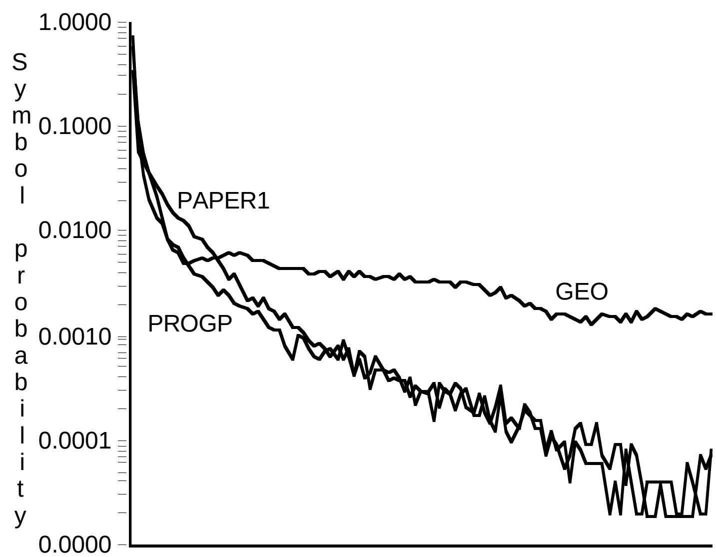


Fig 4. Probabilities of first 128 MTF codes

It is also instructive to look at the costs of emitting data from the various ranks, as shown in Fig 5. There is very little difference between the three files in the costs of emitting a byte at a particular order. A byte at Order 0 requires about 0.5 bits, and at Order 1 requires about 3 bits.

The cost thereafter increases at about 1 bits per byte, with smaller increases at higher orders. The actual proportion of bits emitted at each order decreases as the order increases, but the change is not nearly as dramatic as either the number of bytes handled at each order, or the bits/byte at each order.

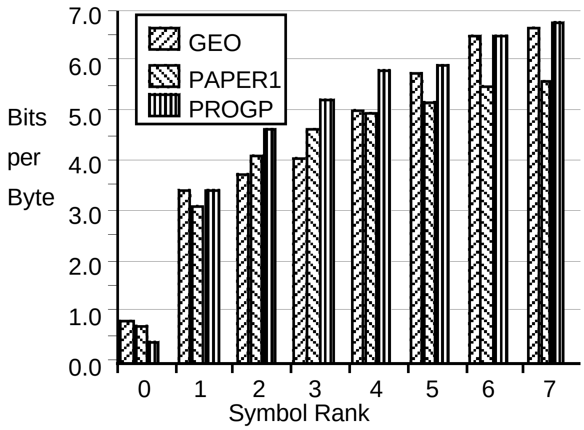


Fig 5 a
Cost of emitting bytes for symbol rankings

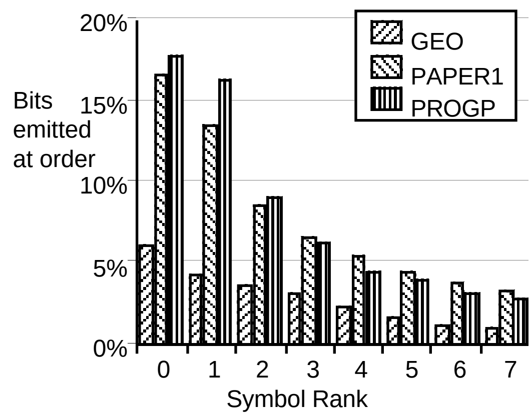


Fig 5 b
Fraction of bits at different orders

It is obvious that improvements in compression must come from decreasing the cost at the low ranks, simply because high-rank symbols are relatively rare. Halving the cost of coding ranks 0 or 1 would, in each case, improve compression by about 7–8 %. To some extent too, the various effects compensate one for the other. Thus PROGC emits over 70% of its symbols at Rank=0, at a cost of about 0.4 bit/byte. PAPER1 emits about 60% of its codes at that rank, but at about 0.6 bits/byte. Both emit about 16–17% of their total bits at Rank=0. (This effect has been observed in most attempts at improving compression; the arithmetic coding models are remarkably resilient and largely compensate against any attempt to 'improve' coding performance.)

One simple optimisation is possible, and has been applied in the results above. Many files use only a portion of the full alphabet of 256 symbols, such as ASCII text files which use only the first half of the available codes. During the initial processing it is easy to examine the file and determine its maximum code value, allowing text files to be encoded with a reduced alphabet of 128 symbols rather than the full 256 symbols. The effect of this change is to improve the coding of text files by about 0.04 bits/byte. The reduced alphabet has a serious effect on the sort phase, which uses two symbols to form a 65,536-way radix sort. As text files have an alphabet of 90 – 100 symbols, we find that only 8,000 – 10, 000 sort "buckets" are actually used (say 12–16% of the total).

3.1 Tests with "better' compressors

One of the motivations for this work was the realisation that the original algorithm achieved excellent compression with a Move-To-Front compressor, which is generally regarded as having only moderate performance. It was thought interesting to test the algorithm with compressors which approach the state of the art.

The Block-Sort compressor was altered to write out the codes after the MTF operation and these files were then used as inputs to other compressors. Results are given in Table 2 for Nelson's "COMP-4" compressor[6] which has the advantage of being publicly available and of being able to run at different maximum orders.

File	bs Order0	COMP4 order 2	COMP4 order 3	COMP4 order 4
BIB	2.022	2.171	2.286	2.375
OBJ1	4.011	4.588	4.639	4.669
PAPER1	2.513	2.754	2.915	3.308
PAPER2	2.445	2.635	2.791	2.923
PROGC	2.595	2.859	2.989	3.094
PROGL	1.846	1.916	2.004	2.082
PROGP	1.859	1.928	2.000	2.077
TRANS	1.644	1.695	1.761	1.829

Table 2. Compression at high arithmetic orders

It is clear that the "better" the compressor the worse the results! Consistent results were found for a variety of dictionary compressors as well. Not all files are given — for many files the COMP4 compressor terminated abnormally with failures of its memory management. Quite clearly there is little hope of improvement here, for reasons which are given later.

3.2 Adjusting the Move-to-Front operation

One possible improvement, which is mentioned by Burrows and Wheeler, is to move symbols only most of the way toward the front of the MTF list, rather than to the very front. The intent is that a new symbol does not immediately displace the currently active symbols at the head of the list. The smaller files of the corpus were therefore compressed with varying MTF distance, with the results shown in Table 3. We show a reference order-0 result[2], and then with move-

[2] These tests were done with a differently tuned compressor from that used in the other results, giving slightly different values in the reference "order-0" column.

ments of $^{31}/_{32}$ and $^{7}/_{8}$ of the original symbol displacement. The first movement, to $^{1}/_{32}$ of the original displacement is almost as strong as a move to the very head, while the second, to $^{1}/_{8}$, is rather weaker.

For most files, movement to anywhere except the very front leads to a definite reduction in performance. The two less-compressible files GEO and OBJ1 showed modest improvements with the weaker movement.

File	bs Order0	Move to 1/32	Move to 1/8	Move avg/2	Move avg/8
BIB	2.133	2.135	2.165	2.133	2.133
GEO	4.812	4.808	4.807	4.808	4.809
OBJ1	4.229	4.221	4.260	4.250	4.229
PAPER1	2.606	2.613	2.646	2.609	2.606
PAPER2	2.571	2.571	2.576	2.571	2.571
PROGC	2.667	2.674	2.716	2.669	2.667
PROGL	1.839	1.842	1.887	1.840	1.839
PROGP	1.822	1.836	1.906	1.824	1.822
TRANS	1.602	1.626	1.734	1.608	1.602

Table 3. Compression with varying Move-to-Front distance

The two last columns show the results of an attempt to vary the movement in accordance with the file statistics. Every 1000 symbols the average MTF distance is calculated and the “move to” position set to $^{1}/_{2}$ or $^{1}/_{8}$ of that distance. Most files are at near their previous best values, except for OBJ1. In all cases though the change is less than 0.1% and judged to be of little real benefit. We therefore retain, at least for the present, the simple Move-to-Front operation, without any tuning of its operation.

4. What Block-Sorting actually does

The block-sort compression is actually a sequence of processes as shown in Fig 5, the first two of which transform data without compression and only the last performs compression.

The three stages are —

1. The initial, sorting, stage permutes the input text so that similar symbol contexts are grouped together. Every input symbol is still present and identifiable in the output and no compression has occurred (in fact there is a very slight expansion because a few bytes are

needed to hold the initial index). The permutation has however created strong locality as the grouping of the (invisible) contexts has collected together the few symbols likely to occur in each context.

2. The Move-to-Front phase then converts the various *locally valid* contexts into a single *globally valid* context. The most likely symbol in each neighbourhood converts to a 0, the next most likely to a 1, and so on. Whereas the local contexts are fairly dynamic and fast-changing, the global one is much more stable with relatively constant statistics. There is still no compression; each symbol is simply replaced by a Move-to-Front index, with a maximum value equal to the alphabet size.

3. The final compression stage exploits the highly skewed frequency distribution from the second stage to produce efficiently-compressed output.

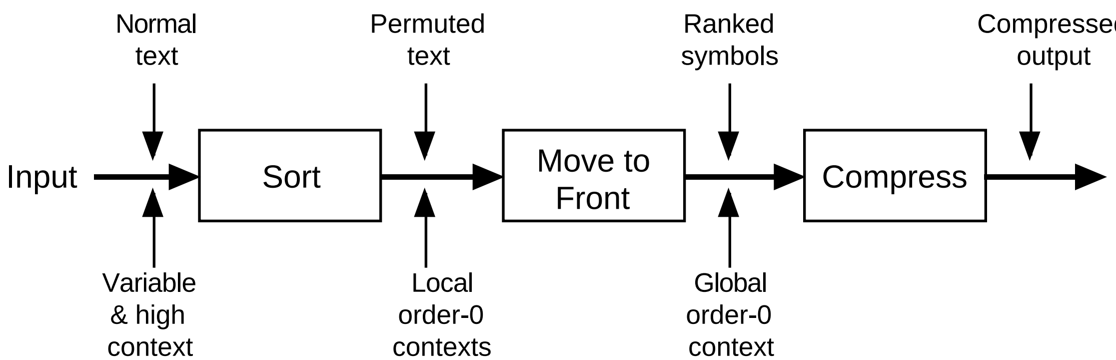


Fig. 6 Data flow in Block Sorting Compression

It is immediately obvious why the "good" compressors do not work well. All efficient text compressors (whether dictionary, statistical, etc — all are equivalent) exploit the high-order context structure of the input text. That structure has been destroyed by the sorting and transformed into the much simpler local and then global order-0 contexts. Thus a conventional compressor tries very hard to detect non-existent structure in the data and may even wastefully keep signalling that there is no high-order structure to use! A similar effect has been observed by Bell et al [ref 2, p 270].

4.1 The Move-to-Front operation

An impression remains that Move-to-Front might not be the best operation in middle part of the compressor. It has a possible disadvantage that characters lose their identity; all that remains is their rank. We will see later that a more appropriate ordering may be by probability or likelihood of occurrence, whereas MTF orders by recency. The two are similar, but certainly not identical. Another approach may be to encode with a straight arithmetic coding of the symbols, but with a statistics model which can respond very rapidly to changes in statistics. This is to be investigated.

5. The limits to compression

One of the major frustrations in text compression is the discrepancy between what is believed to possible (described here as the *experimental* results, as the values are derived from experiments with human subjects) and what has been achieved by the best compressors (the *practical* results). Bell et al [2] have a good discussion of work on the entropy of English text, leading to the "best experimental" value of about 1.3 bits/letter. The classic paper by Shannon [7] is certainly worth reading (and becomes more relevant later in this report!) By contrast, the best text compressors seem to be tending to a limit of about 2 bits/letter (the apparent "practical" limit). Why the difference? There are several possible reasons.

1. The experimental work uses a smaller alphabet of about 30 characters, whereas practical compressors usually work with about 80–90 characters in typical formatted text. From experience in coding with reduced alphabets, this may contribute about 0.1 bits/symbol.

2. The practical compressors generally use contexts of only 4–6 characters, whereas the experimental results imply contexts of perhaps 10^8 characters for adult subjects with some decades of language experience. Human subjects can also use experience on grammar, syntax, semantics and subject matter to direct the estimation of characters. Thus the actual contexts are very large, and perhaps not even measurable in terms of visible symbols.

3. The final, and probably most important, reason is that the experimental and practical results apply to different situations. In the experimental situation, human subjects somehow estimated a likely letter, and were then told whether it was correct or incorrect. If incorrect, they then retried until successful. The practical results on the other hand require a coder to emit all the information for the decoder to successfully decode the symbol.

   In information terms, the symbol encoder is somewhat unreliable, adding noise to the nominally correct prediction. The experimental results apply to a system with *error detection and retry* (ARQ in communications terminology) where the encoder can compare the prediction with the original; the emitted information corresponds to the comparison success/failure. The practical results however apply to a system with *forward error correction* where the emitted code contains enough information to overcome all expected errors. It is well known that forward error correction has a much lower information rate than error detection and retry.

To illustrate, we can consider the unreliable predictor as analogous to a noisy information channel. The "noisy channel" is internal to the predictor (and can be precisely duplicated at the receiver) so we can observe the channel output and signal its status by a reliable reverse channel. A binary symmetric channel, encoding equiprobable symbols and with an error probability of $^{5}/_{6}$ (1.2 symbols must be received for each correct one received) has a channel capacity of only 0.35. Thus to convey one bit of information, over 2.8 bits must be transmitted over the channel. The poorer results of practical compressors therefore correspond precisely to what we would expect from forward correction over a noisy channel, as compared with feedback detection over that same channel.

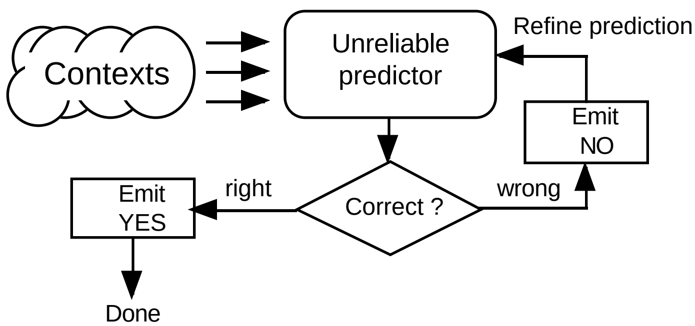


Fig. 7 Action of "experimental" compression

This leads to a possible "Prediction/Correction" compressor, which mirrors exactly the "experimental" situation. The coder, shown in Fig 7, contains a "predictor" which somehow estimates the next symbol and is then told whether to revise its estimate; the revision instructions constitute the coder output. The decoder contains an identical predictor which, revising according to the transmitted instructions, is able to track the coder predictor and eventually arrive at the correct symbol.

While apparently novel in text compression, these techniques are well-established in analogue data transmission and include techniques such as delta modulation and linear predictive encoding. A major problem is that the idea of "error" is usually clear in an analogue environment, but not in a text symbol. We will return later to consider an actual design for this proposed compressor.

6. The classification of compressors

Most best-performing text compressors have been either of the dictionary type, and especially

Ziv-Lempel (either LZ-77 or LZ-78), or statistical, as exemplified by PPM and its derivatives. It is well known [2] that these two apparently different compression techniques are in fact equivalent. More recently, Cleary et al [5] have shown that Block Sorting can be implemented with the data structures used in their PPM* compressor and that it too is equivalent to the general dictionary/statistical compressors. Again, Bunton[1] has examined the structure of the Dynamic Markov Chaining compressor (DMC). These results have been coordinated with those of Cleary [3] to demonstrate at least a formal equivalence of all the established text compression techniques.

We now return to the results of the previous section and propose an actual implementation of the prediction/compression text compression. The encoder contains a PPM-style mechanism which examines the known preceding context and produces a list of possible symbols, ranked according to their expected likelihood. The initially predicted symbol is the most likely and the "error" is its distance in the list from the actual symbol to be encoded. The encoder therefore just emits the position of the symbol in the ranked list. (The position is clearly the number of estimates to arrive at the correct answer.) The decoder has an identical predictor, producing the same list, and can use the received "error" to read the correct symbol. The proposed name is a PPMδ compressor, from its combination of PPM symbol prediction with "delta" coding of the error value.

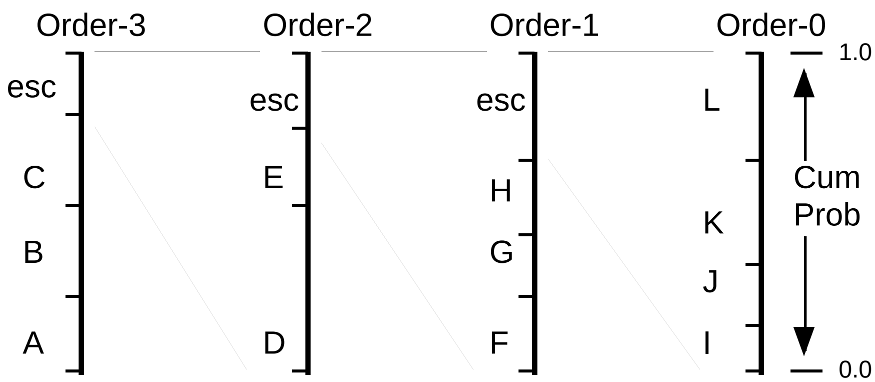
But this is essentially what the Block Sorting algorithm does, although with a permutation of the input text to facilitate the prediction from contexts. The MTF list approximates an ordering in symbol frequency, and the emitted index is simply an error indication.

The PPMδ compressor examines the symbols in their *natural* order (in contrast to the permuted order of block sorting), generating the contexts from the history of what has been encountered already in the file. The most likely symbols are those in the highest order context, ranked in probability order. These are followed by the ranked symbols in the next non-empty order, and so on. The code at each stage is of course just the rank of the symbol. Note that there is no calculation of escape probabilities — escapes do not exist! Neither are we concerned about the actual symbol probabilities, just their rankings within the contexts.

Finally we must recognise that the proposed compressor is not at all new in text compression terms. The original paper by Shannon on the entropy of printed English [7] describes what is essentially the same system, with results which mirror those here. It is interesting too that for contexts of about 6 or 8 letters he obtains entropies which are quite close to those of the best current compressors. His subjects generally achieved a success rate of 60–70% with their letter predictions, compared with about 60% on the text files here. This difference is easily accounted for by the much greater contexts available to the human subjects.

7. Coding of Escapes

An important aspect of PPM and similar compressors is their use of “escapes” to move from an assumed, but unusable, higher order to some lower order from which the symbol can be emitted. One of the major problems in the design of PPM-style compressors has been deriving a suitable frequency for the escape code. A frequency which is too high penalises the symbols which do exist within the context, but an escape with too low a frequency lowers the efficiency of emitting symbols from the lower order. The work of the previous sections on Prediction-Correction Coding and the PPMδ compressor gives some additional insight into the handling of escapes.



The total range of each lower order is scaled into the ESC probability of the next higher order.

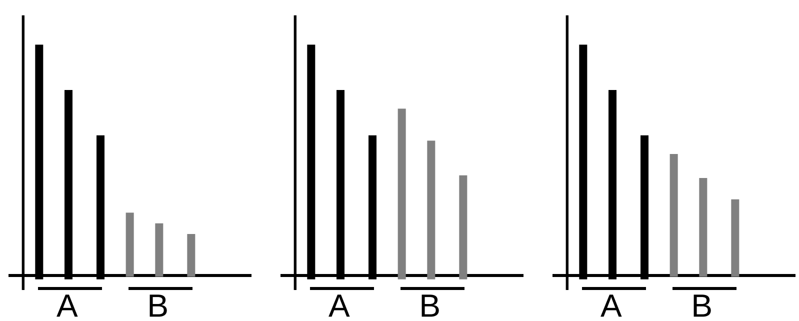
Fig. 8 Scaling from escapes in PPM-style compression

Considering the proposed PPMδ compressor as a particular example, we rank the symbols according to their predicted probabilities and can assign a probability distribution to the symbol alphabet. The most probable symbols come from the highest order contexts; when that is exhausted the next symbols come from the next non-empty lower-order context, and so on down to the order-0 context. The transition from one context to the next corresponds to the emission of an escape in PPM. If we go back to PPM and its arithmetic coding, we see that an escape at one order provides the “space” for the entire frequency spectrum of the lower model, as shown in Fig. 8. The escape probability therefore acts as a scaling factor for the lower-order probabilities. A group of low-order probabilities tend, when arithmetically coded, to share the same initial prefix, and this prefix, with escapes, is simply the escape coding itself. Thus the action of the escape is to scale the “escaped-to” probabilities and provide the appropriate prefix corresponding to that scaling.

In the PPMδ compressor, we are combining the partial probability distributions from the various contexts into a single composite distribution. (Remember that the symbols are not in their natural order, but are ranked in order of probability). The action of the escape probability is to scale the lower order portion with respect to its higher-order neighbour. Presumably the two portions should match to give a reasonably smooth distribution — the matching can be adjusted by varying the escape probability. The calculation of the escape frequency is thereby reduced to a problem of curve fitting. (While the assumption of a “smooth distribution” may be questionable, it is probably as justifiable as any other rationale for choosing the escape frequency!)

We work from the lower orders to the higher, adjusting the escape frequency of the higher order so that it matches its lower neighbour with minimal discontinuity. This is in contrast to the usual methods in PPM where we use only evidence from within a particular order to estimate the likelihood of escaping from that order. In classical coding terms the older escape coding techniques correspond to top-down Shannon-Fano coding, while the suggested technique corresponds to bottom-up Huffman coding.

The effect is shown in Fig 9, combining the two partial distributions A and B, with B encoded as an escape from A. In Fig 9(a), the escape probability is clearly too low, while in Fig 9(b) it is too high. In Fig 9(c), the relatively smooth continuous distribution shows that the escape probability is about correct.



(a) A *esc prob too low* *(b)* A *esc prob too high* *(c)* A *esc prob correct*

Distributions for 2 models, with B coded into escape from A

Fig. 9 Effect of escape probability when merging two distributions

A simple heuristic may be to scale so that the total probability of the lower orders is about the same as the probability of the least-frequent symbol of the higher order. (This might not work well with highly-skew distributions.) Remember too that the escape is to the next non-empty order. The reason is obvious when one considers the amalgamation of frequency distributions as here, but less obvious when the escape is considered as a movement between coding orders.

This approach also helps understand the need for exclusions. Without exclusions a symbol which is already included in some higher-order context is repeated in a lower-order context and uses probability space which is completely wasted. Blending, another important concept in PPM compression, simply implies the merging of the various partial probability distributions in a systematic, but tidy, manner.

A reading of most descriptions of PPM implies that the escape is at best a necessary evil, being an extraneous code which is needed to force entry to a lower-order model. The above discussion shows that this is not so — the escape is an essential adjunct to combining several models from different orders into the single frequency distribution from which any symbol can be emitted.

8. A "δ-coded" Block Sort compressor

Following from the preceding discussions, it seemed sensible to try a compressor with the simple "Yes/No" output coding. The output index is simply emitted as a unary-coded value — 0 ↠ 0, 1 ↠10, 2 ↠110, 3 ↠1110, and so on. A preliminary test of feasibility used a single arithmetic coding model and is shown as "BS-delta 1 model" in Table 4.

This Table also includes the results for the best published compressors — PPMC (the usual reference), Cleary's new PPM*[5] (the best to date), "BS original" (the original Block Sorting compressor), together with the order-0 arithmetically coded Block Sort from earlier. We see that the "BS-delta 1 model" is a reasonable compressor, but not as good as any of the other compressors in the table.

An improvement follows from recognising that the compressed output tends to consist of alternating regions. Most obviously there are many long runs of zeros, while the desired symbol is at the head of MTF list and the MTF mechanism is idle. Interspersed with these zero runs are bursts of MTF activity as hitherto rare symbols become active, establishing a new context. We therefore use two models —

- The "Zero" model is used primarily for emitting the runs of 0s. It also emits the first '1' of the code for a new symbol; the coder then switches to the other model.
- The "One" model is used primarily in emitting the codes for new symbols. It emits the 1s

and the final 0 of each unary code. It will emit the first 0 of a run, after which the coder switches to the "Zero" model.

File		PPMC	B S Order0	PPM*	B S original	BS-delta 1 model	BS-delta 2 models	BS-delta 3 models
Bib	111,261	2.110	2.133	1.91	2.07	2.214	2.036	2.045
Book1	768,771	2.480	2.523	2.40	2.49	2.612	2.449	2.452
Book2	610,856	2.260	2.198	2.02	2.13	2.238	2.103	2.106
Geo	102,400	4.780	4.812	4.83	4.45	5.337	5.448	4.499
News	377,109	2.650	2.677	2.42	2.59	2.807	2.631	2.610
Obj1	21,504	3.760	4.227	4.00	3.98	4.458	4.508	4.044
Obj2	246,814	2.690	2.710	2.43	2.64	2.851	2.816	2.589
Paper1	53,161	2.480	2.606	2.37	2.55	2.750	2.571	2.567
Paper2	82,199	2.450	2.571	2.36	2.51	2.661	2.490	2.502
Pic	513,216	1.090	0.919	0.85	0.83	0.949	0.832	0.828
ProgC	39,611	2.490	2.666	2.40	2.58	2.820	2.649	2.614
ProgL	71,646	1.900	1.839	1.67	1.80	1.949	1.809	1.806
ProgP	49,379	1.840	1.821	1.62	1.79	1.937	1.822	1.798
Trans	93,695	1.770	1.601	1.45	1.57	1.807	1.633	1.612
AVG		2.482	2.522	2.34	2.43	2.671	2.557	2.434
Rel. to BS-delta 3		102%	104%	96%	100%	110%	105%	100%

Table 4. Predictor-Corrector Compressor Performance

The results are shown in the next-to-last column of Table 4. The 2-model BlockSortδ compressor is, in most cases, quite comparable to the other compressors. In fact considering its apparently naive coding, the results are quite remarkable.

A further improvement comes from considering the probability distribution of the symbol ranks. Figure 10 below repeats the earlier Fig 3, but with added lines having slopes of 2^{-n}.

If we consider Huffman coding of alphabets with very skewed probability distributions, we find that in regions where the frequency is decreasing by more than a factor of 2 between successive symbols, we often get a unary code where each symbol adds one more digit. Groups of symbols with similar probabilities share a common prefix to a binary coding. Thus areas in Fig 9 where the distributions are steeper than the added lines are coded as unary codes, and less steep areas tend to be coded as groups of symbols.

In this case we see that for low ranks (less than about 5 – 7) all of the represented codes are handled well by a unary code. For higher values (where the distributions flatten out), a more conventional arithmetic coding is appropriate.

This has been done in the last column of Table 4. Values up to 6 are represented by the unary code as before, but larger values are coded through a standard 256-symbol arithmetic coder, with a prefix of 6 1s (from the unary coding). The improvement for the binary codes (those with large average MTF distances) is quite dramatic. Other files show a slight decrease in performance, but there is an overall improvement from the change. The importance of shape of the frequency distribution will be explored further in the second report of this sequence.

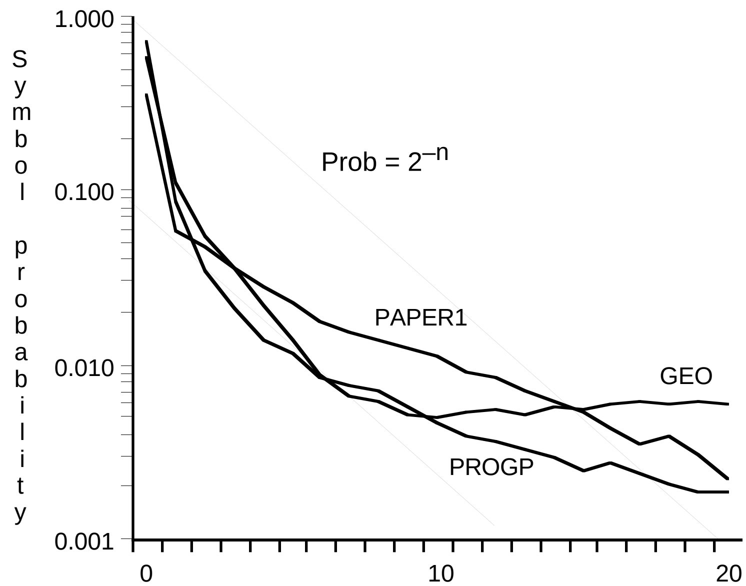


Fig 10. Probabilities of first 20 MTF codes

9. Computing requirements.

Most high performance statistical compressors require considerable computing resources, both memory and processing. (One experimental one, compressing PIC, apparently requires 160 Mbyte to build its context models, and takes 7 hours on a workstation.) The block sorting compressor needs about 10 – 12 Mbyte, in the current implementation, allocated as follows –

- The data buffers need a byte for the data, a long (4 bytes) to link bytes which belong to the radix-sort bucket, and another long for information on runs. There is actually a second data buffer, giving an “input” and an “output” buffer for most stages, but this is not really necessary. With a total of 10 bytes per input symbol, and a buffer of 800 kbytes for the largest files in the Corpus, the overall data storage is 8 byte.
- The 65,536 buckets of the radix sort need 12 bytes each to hold the first and last indices of the symbol list and the bucket size. This is nearly 800 kbytes. When a bucket is being sor-

ted its symbol indices are collected in a tag array, which needs 4 bytes per symbol. Space is allocated for the largest bucket, which is about 450,000 elements for PIC (over 85% of the file is zero bytes, and most of those go into one bucket!) This array requires another 1.8 Mbyte.

The total memory is thus about 10.6 Mbyte. From Appendix II we see an average speed of about 50μs per byte (20,000 bytes/second) for a good workstation. A good ‘notebook’ computer (a Macintosh Powerbook 540C, 66MHz 68040LC) runs at about 150μs per byte, or about a third of the workstation speed. This speed is somewhat degraded because of extensive statistics collection and the unoptimised sort routines. The memory and processing requirements are quite compatible with readily available computers.

10. Conclusions

This report has presented some preliminary measurements on the new Block-Sorting text compressor. It is initially implemented with a simple order-0 arithmetic compressor in the final stage and various parameters are obtained. In particular it shown that there is no advantage in substituting “better” compressors, and little advantage in tuning the Move-to-Front operation. The symbols which are finally encoded have a frequency distribution which can exploited in designing an optimal compressor.

A general discussion on the limits of compression suggests reasons why practical compressors do not reach the expected performance limits, and leads to a proposal for a new type of text compressor. The Block-Sorting compression technique is shown to be an example of this suggested type.

Finally, these discussions lead to a new interpretation of the place of escape codes in PPM compression, including a suggested new method for estimating escape probabilities.

11. Acknowledgements

This work was supported by grant A18/XXXXX/62090/F3414032 from the University of Auckland and performed while the author was on Study Leave at the University of California – Santa Cruz and the University of Wisconsin – Madison. The author acknowledges the contributions of all of these institutions.

Many of the later ideas reported here followed from discussions at the Data Compression Conference, Snowbird, Utah, March 1995. The contributions of Suzanne Bunton, John Cleary and Victor Miller are acknowledged — they might not recognise any of their ideas in this work, but

their conversations certainly provided an initial stimulus!

References

[1] S. Bunton, “The Structure of DMC”, *Data Compression Conference*, *DCC-95*, Snowbird Utah, March 1995

[2] T.C. Bell, J. G. Cleary, and I. H. Witten, “*Text Compression*”, Prentice Hall, New Jersey, 1990

[3] S. Bunton and J. G. Cleary, private communication.

[4] M. Burrows and D.J. Wheeler, “A Block-sorting Lossless Data Compression Algorithm”, SRC Research Report 124, Digital Systems Research Center, Palo Alto, May 1994

[5] J. G. Cleary, W.J. Teahan, I. H. Witten, “Unbounded Length Contexts for PPM”, *Data Compression Conference, DCC-95*, Snowbird Utah, March 1995

[6] M. Nelson. “Arithmetic coding and statistical modelling”, *Dr Dobbs Journal*, Feb 1991. Anonymous FTP from `wuarchive.wustl.edu/systems/msdos/msdos/ddjmag/ddj9102.zip`

[7] C.E. Shannon, “Prediction and Entropy of Printed English”, *Bell System Technical Journal*, Vol 30, pp 50–64, Jan 1951

Appendix I. The Block-Sort Implementation

The present work has concentrated on the compression ability, rather than speed, investigating various types of compression which might be appropriate to the transformed data from the sorting stage. We will find that most files are handled well by relatively simple sorting techniques, as described later, and that a simple MTF implementation is adequate for most files.

Sorting

Because of the emphasis on compression rather than speed, the sorting techniques have been developed only to the extent that is necessary to handle files in a reasonable time. The sorting is done using a standard C *qsort* routine[3], after first performing a 65,536-way radix sort on the input, ie radix-sorting on symbol digraphs. The *compare* procedure for the sort proceeds in several stages -

1. The third and fourth symbols of the two contexts are compared as a coarse initial filter (the first two symbols are of course equal from the radix sort). Comparisons resolved at this stage are the "short compares" of Table 1 and Appendix II.
2. The standard C *memcmp* function is used to compare about 100 symbols.
3. Strings which survive step 2 proceed to a "*longCompare*" with three stages of *memcmp*. The first compares until one string reaches the end of the input, then (wrapping round the first) until the second reaches the end, and finally until the starting point is reached. (If an End-File symbol is used instead of the wrap-round, the compare should stop after the first stage, reporting the shorter comparand as higher.) These are the "long compares".
4. Some files contain long runs of symbols. These are handled by building an auxiliary array in parallel with the input buffer, containing the length of the run following this symbol. If, after surviving step 1, both symbols are found to have following runs, the "*runCompare*" function compares the bytes at the end of the shortest run. The *longCompare* routine is called if the symbols compare equal. These are the "run compares".

This combination works well for most files, (in fact all but PIC) but better handling is needed for runs, possibly along the lines described by Burrows and Wheeler. The following description is largely a restatement of some of their points. In the first case an equal comparison from *runCompare* should clearly recurse back into the main compare structure, rather than the lazy fall-

[3] Even so, be warned that this supposedly standard routine is far from standard! In the course of this work, the author tested 5 different versions of qsort. On a test array of 10 random integers, the 5 versions required 28, 28, 29, 36 and 40 comparisons. The version with 36 comparisons also required 6 tests of an element against itself!

through to the *longCompare*.

Of more importance though is the fact that few symbols within a run need a full set of sorting comparisons anyway. If we have a run "*…ssssss…r…*", where $r < s$, then strings starting later in the run must sort lower than earlier-starting strings. Similarly, a run "*…ssssss…t…*", where $t > s$, must have later strings sorting higher than earlier and longer ones. Thus the contexts from a run of length N can be ordered in time $O(N)$, whereas a full sort can be expected take $O(NlogN)$ operations. It is necessary to merge together the outputs from all runs, but this is another simple, linear-time, task.

In summary, the radix-sort bucket for symbol "`s`" can be sub-classified into four categories -

1. Contexts of the form "`sr…`", where `r` < `s`.
2. Contexts of the form "`ss…r`", where `r` < `s`.
3. Contexts of the form "`ss…t`", where `s` < `t`.
4. Contexts of the form "`st…`", where `s` < `t`.

These categories can be sorted individually and in the order given, emitting the preceding byte as each sort proceeds.

Move-to-Front

Another area of possible optimisation is the Move-to-Front action. This has been implemented as a simple array containing the symbols in MTF order, with a matching index array to find the position of any given symbol. Moves are done by actually shuffling the MTF list with corresponding changes to the mapping table. It sacrifices speed for simplicity, but given that the average MTF distance is only 3–4 symbols for most files, the penalty is small for files other than GEO, OBJ1 and OBJ2.

Sorting; a postscript

While all of the preceding work has been done using the sort techniques as described above, it is appropriate to mention some sort refinements which have been tested recently and will be used in future versions of the block-sorting routines.

Burrows and Wheeler describe how the sort can be accelerated by using an array of long-words to hold the input text, one word per symbol, and "striping" bytes across preceding words. With 4 bytes per word, a symbol goes into the leftmost byte of "its" word, the second byte of the preceding word, and so on for the two previous words. It is then possible for a word compare to compare 4 bytes at a time, with about the same overhead as for a single character compare. (The comparison uses a stride of 4 words between steps. A 64-bit word can hold 8 bytes and has a stride of 8.) In the actual sort routine the main comparison loop is preceded by a single long-word compare as a preliminary "short compare" filter. The tests can be offset by two positions from the start of the nominal comparands (the first two symbols are known to be the same, because they are in the same radix-sort bucket). The initial comparison then tests the first 6 symbols, or 10 symbols with 64-bit long words.

One of the main remaining problems is dealing with runs. While the text above has described some clever approaches to accelerating sorting with runs, a better approach is to eliminate the runs completely by preprocessing the text with run-length coding, and then sorting and compressing this resulting file. Note that the run-encoding is intended only to improve the sorting speed. Most files have few runs and the slight improvement from run-elimination is balanced by the penalty of eliminating some contexts.

The combination of word sorting and run-encoding reduces the sorting time for PIC from about 10 minutes to about 10 seconds; about 80% of the original file is absorbed into the compressed runs. The compression for PIC improves by about 5%.

The final major problem is that files of the form "…`aaabaaabaaabaaab`…", ie with exact periodic structure, still sort very slowly because many of the comparisons must proceed to the very end of the file. This structure can be handled by another form of preprocessing, based on LZ-77 parsing techniques. It uses an LZ-77 scanner which allows recursive matching into the look-ahead buffer and encodes as a {displacement, length} couple those strings which do give a recursive match. Matches wholly within the history part of the LZ-77 buffer are ignored. (Run encoding is really just a special case of periodic coding.)

A lesser problem is that the symbol digraphs are not distributed evenly across the radix-sort buckets. This may be handled by a further level of radix sorting on over-large buckets. An alternative technique by Chen and Reif[4] involves sampling the file to estimate an appropriate allocation of buckets among the sort keys so that the bucket sizes are approximately balanced.

While word-sorting and run encoding have been implemented, neither the LZ-77 parse for handling periodic structure nor any method for handling over-full radix-sort buckets has been investigated so far.

---

[4] Shenfeng Chen, John H. Rief "Using Difficulty of Prediction to Decrease Computation: Fast Sort, Priority Queue and Convex Hull on Entropy Bounded Limits", *34th Symposium on the Foundations of Computer Science,* pp 104–112, 1993

Appendix II. Logs for the Calgary Corpus

This Appendix presents the the output log for a run of the simple order-0 compressor over the files of the Calgary Corpus. The more important values have been summarised elsewhere, but there is still much information to be gleaned from a close observation of these records. The results were obtained on a Hewlett-Packard 755 workstation with a 125MHz PA-RISC processor and 64MB of RAM.

The comparison counts, which have been included in Table 1 earlier, are very sensitive to the idiosyncrancies of *qsort*, as mentioned in the footnote to the previous Appendix. The version of qsort with self-comparisons requires about 10% more comparisons on most files, but over 4 times as many comparisons on PIC! The number of comparisons increases from about 11 million to over 47 million. The overall compression is not affected.

These results all use the original character-based sort routines, rather than the newer improved ones with word sorting and run encoding. The word sort routines have no “run” comparisons and more “short” comparisons.

```
Block-sorting algorithm, after Burrows & Wheeler

M T F encoding of permuted input
Order-0 compressor; data limit = 8192, increment = 16
Run - 13 April 1995 at 14:12


Compress "bib"

111261 input bytes, 237300 output bits  (2.133 bit/byte),   46.74 us/byte
    5.20 seconds
953647 compares, (499267 short, 0 run, 423 long)

Average MTF distance =   2.50; non-zero =   5.50
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  74297  9555  4404  3217  2491  2053  1788  1594  1459  1276
Ratio    66.8%  8.6%  4.0%  2.9%  2.2%  1.8%  1.6%  1.4%  1.3%  1.1%


Compress "book1"

768771 input bytes, 1939874 output bits  (2.523 bit/byte),   57.40 us/byte
   44.13 seconds
10.066830M compares, (5.134120M short, 400 run, 43 long)

Average MTF distance =   2.45; non-zero =   3.88
Dist.        0      1     2     3     4     5     6     7     8     9
Counts  382507 118027 60885 40517 29196 22381 18103 15285 12927 11182
Ratio     49.8%  15.4%  7.9%  5.3%  3.8%  2.9%  2.4%  2.0%  1.7%  1.5%
```

Compress "book2"

610856 input bytes, 1342713 output bits (2.198 bit/byte), 53.61 us/byte
32.75 seconds
7.295435M compares, (3.649435M short, 386 run, 2295 long)

Average MTF distance = 2.25; non-zero = 4.20

Dist.	0	1	2	3	4	5	6	7	8	9
Counts	371490	76671	36148	23014	16884	12832	10541	8966	7471	6322
Ratio	60.8%	12.6%	5.9%	3.8%	2.8%	2.1%	1.7%	1.5%	1.2%	1.0%

Compress "geo"

102400 input bytes, 492706 output bits (4.812 bit/byte), 57.52 us/byte
5.89 seconds
710791 compares, (654944 short, 8588 run, 2860 long)

Average MTF distance = 36.05; non-zero = 55.56

Dist.	0	1	2	3	4	5	6	7	8	9
Counts	36619	6189	4905	3776	2280	1436	898	684	647	529
Ratio	35.8%	6.0%	4.8%	3.7%	2.2%	1.4%	0.9%	0.7%	0.6%	0.5%

Compress "news"

377109 input bytes, 1009478 output bits (2.677 bit/byte), 49.99 us/byte
18.85 seconds
3.794914M compares, (2.220768M short, 101264 run, 33458 long)

Average MTF distance = 3.80; non-zero = 7.65

Dist.	0	1	2	3	4	5	6	7	8	9
Counts	218517	38585	20737	14020	10609	8526	6970	5886	5038	4435
Ratio	57.9%	10.2%	5.5%	3.7%	2.8%	2.3%	1.8%	1.6%	1.3%	1.2%

Compress "obj1"

21504 input bytes, 90896 output bits (4.227 bit/byte), 58.59 us/byte
1.26 seconds
120686 compares, (47001 short, 49775 run, 39 long)

Average MTF distance = 23.47; non-zero = 46.51

Dist.	0	1	2	3	4	5	6	7	8	9
Counts	10884	1257	695	511	416	321	297	232	234	193
Ratio	50.6%	5.8%	3.2%	2.4%	1.9%	1.5%	1.4%	1.1%	1.1%	0.9%

Compress "obj2"

246814 input bytes, 668972 output bits (2.710 bit/byte), 48.09 us/byte
11.87 seconds
2.012735M compares, (1.107942M short, 16611 run, 10586 long)

Average MTF distance = 10.26; non-zero = 30.00

Dist.	0	1	2	3	4	5	6	7	8	9

```
Counts  168023 14928  7657  5162  3854  3005  2380  2006  1743  1448
Ratio    68.1%  6.0%  3.1%  2.1%  1.6%  1.2%  1.0%  0.8%  0.7%  0.6%


Compress "paper1"

53161 input bytes, 138537 output bits  (2.606 bit/byte),   43.26 us/byte
    2.30 seconds
381283 compares, (233918 short, 634 run, 77 long)

Average MTF distance =   3.27; non-zero =   6.45
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  31021  5994  2906  1959  1503  1221   970   830   742   679
Ratio    58.4% 11.3%  5.5%  3.7%  2.8%  2.3%  1.8%  1.6%  1.4%  1.3%


Compress "paper2"

82199 input bytes, 211330 output bits  (2.571 bit/byte),   45.62 us/byte
    3.75 seconds
711529 compares, (424242 short, 6 run, 29 long)

Average MTF distance =   2.81; non-zero =   5.06
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  45512 10091  5273  3642  2688  2105  1765  1543  1259  1171
Ratio    55.4% 12.3%  6.4%  4.4%  3.3%  2.6%  2.1%  1.9%  1.5%  1.4%


Compress "pic"

513216 input bytes, 471848 output bits  (0.919 bit/byte),  116.09 us/byte
   59.58 seconds
11.150747M compares, (654959 short, 8.957076M run, 2.312587M long)

Average MTF distance =   1.30; non-zero =   3.39
Dist.        0     1     2     3     4     5     6     7     8     9
Counts  448525 14829  6799  4658  3617  3049  2718  2406  2119  1883
Ratio     87.4%  2.9%  1.3%  0.9%  0.7%  0.6%  0.5%  0.5%  0.4%  0.4%


Compress "progc"

39611 input bytes, 105614 output bits  (2.666 bit/byte),   42.92 us/byte
    1.70 seconds
253062 compares, (150910 short, 1458 run, 225 long)

Average MTF distance =   3.90; non-zero =   8.32
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  23905  4428  1854  1241   934   724   587   499   447   380
Ratio    60.3% 11.2%  4.7%  3.1%  2.4%  1.8%  1.5%  1.3%  1.1%  1.0%


Compress "progl"

71646 input bytes, 131742 output bits  (1.839 bit/byte),   46.62 us/byte
    3.34 seconds
621697 compares, (295551 short, 37115 run, 18836 long)
```

```
Average MTF distance =   1.99; non-zero =   4.63
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  52201  6196  2803  1606  1173   920   740   684   511   464
Ratio    72.9%  8.6%  3.9%  2.2%  1.6%  1.3%  1.0%  1.0%  0.7%  0.6%


Compress "progp"

49379 input bytes, 89940 output bits  (1.821 bit/byte),   46.58 us/byte
    2.30 seconds
379732 compares, (157582 short, 21142 run, 11547 long)

Average MTF distance =   2.18; non-zero =   5.54
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  36554  4292  1756  1046   686   591   431   376   350   285
Ratio    74.0%  8.7%  3.6%  2.1%  1.4%  1.2%  0.9%  0.8%  0.7%  0.6%


Compress "trans"

93695 input bytes, 150020 output bits  (1.601 bit/byte),   44.72 us/byte
    4.19 seconds
736952 compares, (338303 short, 25215 run, 20575 long)

Average MTF distance =   1.96; non-zero =   5.65
Dist.       0     1     2     3     4     5     6     7     8     9
Counts  74304  5331  2459  1524  1221   918   787   718   586   532
Ratio    79.3%  5.7%  2.6%  1.6%  1.3%  1.0%  0.8%  0.8%  0.6%  0.6%
```

Appendix III. Detailed log for PAPER1

This appendix contains a trace of the actual output coding for a portion of the file PAPER1. The columns are, in order —

- The sequential position of the symbol in the *sorted* file.
- Its position in the MTF list — this is the value actually emitted
- Three columns giving the symbol in decimal, hexadecimal and as a text literal
- The 4-symbol *following* context for the symbol
- The number of bits emitted for this symbol
- The arithmetic coder frequencies, in the form [symbol_freq in total_freq]
- The Move-to-Front list, *after* the symbol is emitted

The literal symbols and context symbols are translated into printable codes. In particular, a Carriage Return is printed as “©”.

We see in this file the typical alternation of runs of zeros and groups of non-zero positions as new symbols are introduced into the current context.

```
symbol          MTF   – symbol –    context    emitted   coding frequencies     consequent MTF list
number         index  dec hex lit               bits
20000 from   8 101 65 'e' "caus"  9 bits [  18 in 4496]  "eino -©s(Sa$.\" ..
20001 from   0 101 65 'e' "caus"  0 bits [3300 in 4512]  "eino -©s(Sa$.\" ..
20002 from   0 101 65 'e' "caus"  1 bits [3316 in 4528]  "eino -©s(Sa$.\" ..
20003 from   0 101 65 'e' "caus"  0 bits [3332 in 4544]  "eino -©s(Sa$.\" ..
20004 from   0 101 65 'e' "caus"  0 bits [3348 in 4560]  "eino -©s(Sa$.\" ..
20005 from   0 101 65 'e' "caus"  1 bits [3364 in 4576]  "eino -©s(Sa$.\" ..
20006 from   4  32 20 ' ' "caus"  8 bits [  23 in 4592]  " eino-©s(Sa$.\" ..
20007 from   1 101 65 'e' "caus"  3 bits [ 415 in 4608]  "e ino-©s(Sa$.\" ..
20008 from   0 101 65 'e' "caus"  1 bits [3380 in 4624]  "e ino-©s(Sa$.\" ..
20009 from   0 101 65 'e' "caus"  0 bits [3396 in 4640]  "e ino-©s(Sa$.\" ..
20010 from   0 101 65 'e' "caus"  0 bits [3412 in 4656]  "e ino-©s(Sa$.\" ..
20011 from   1  32 20 ' ' "caus"  4 bits [ 431 in 4672]  " eino-©s(Sa$.\" ..
20012 from   0  32 20 ' ' "caus"  0 bits [3428 in 4688]  " eino-©s(Sa$.\" ..
20013 from   1 101 65 'e' "caut"  3 bits [ 447 in 4704]  "e ino-©s(Sa$.\" ..
20014 from  10  97 61 'a' "ccel"  8 bits [  19 in 4720]  "ae ino-©s(S$.\" ..
20015 from   0  97 61 'a' "ccen"  1 bits [3444 in 4736]  "ae ino-©s(S$.\" ..
20016 from   0  97 61 'a' "cces"  0 bits [3460 in 4752]  "ae ino-©s(S$.\" ..
20017 from   0  97 61 'a' "cces"  1 bits [3476 in 4768]  "ae ino-©s(S$.\" ..
20018 from  14 117 75 'u' "cces" 11 bits [   2 in 4784]  "uae ino-©s(S$." ..
20019 from   1  97 61 'a' "cces"  4 bits [ 463 in 4800]  "aue ino-©s(S$." ..
20020 from   1 117 75 'u' "cces"  3 bits [ 479 in 4816]  "uae ino-©s(S$." ..
20021 from   1  97 61 'a' "ccom"  3 bits [ 495 in 4832]  "aue ino-©s(S$." ..
20022 from   0  97 61 'a' "ccom"  1 bits [3492 in 4848]  "aue ino-©s(S$." ..
20023 from   0  97 61 'a' "ccor"  0 bits [3508 in 4864]  "aue ino-©s(S$." ..
20024 from   0  97 61 'a' "ccor"  0 bits [3524 in 4880]  "aue ino-©s(S$." ..
20025 from   0  97 61 'a' "ccor"  1 bits [3540 in 4896]  "aue ino-©s(S$." ..
```

```
20026 from   0  97 61 'a' "ccor"  1 bits [3556 in 4912]  "aue ino-©s(S$." ..
20027 from   0  97 61 'a' "ccor"  0 bits [3572 in 4928]  "aue ino-©s(S$." ..
20028 from   0  97 61 'a' "ccou"  0 bits [3588 in 4944]  "aue ino-©s(S$." ..
20029 from   6 111 6f 'o' "ccup"  7 bits [  54 in 4960]  "oaue in-©s(S$." ..
20030 from   0 111 6f 'o' "ccup"  0 bits [3604 in 4976]  "oaue in-©s(S$." ..
20031 from   0 111 6f 'o' "ccur"  1 bits [3620 in 4992]  "oaue in-©s(S$." ..
20032 from   0 111 6f 'o' "ccur"  1 bits [3636 in 5008]  "oaue in-©s(S$." ..
20033 from   0 111 6f 'o' "ccur"  0 bits [3652 in 5024]  "oaue in-©s(S$." ..
20034 from   0 111 6f 'o' "ccur"  1 bits [3668 in 5040]  "oaue in-©s(S$." ..
20035 from   0 111 6f 'o' "ccur"  0 bits [3684 in 5056]  "oaue in-©s(S$." ..
20036 from   0 111 6f 'o' "ccur"  0 bits [3700 in 5072]  "oaue in-©s(S$." ..
20037 from   0 111 6f 'o' "ccur"  1 bits [3716 in 5088]  "oaue in-©s(S$." ..
20038 from   0 111 6f 'o' "ccur"  0 bits [3732 in 5104]  "oaue in-©s(S$." ..
20039 from   0 111 6f 'o' "ccur"  0 bits [3748 in 5120]  "oaue in-©s(S$." ..
20040 from   1  97 61 'a' "ccur"  3 bits [ 511 in 5136]  "aoue in-©s(S$." ..
20041 from   0  97 61 'a' "ccur"  1 bits [3764 in 5152]  "aoue in-©s(S$." ..
20042 from   0  97 61 'a' "ccur"  0 bits [3780 in 5168]  "aoue in-©s(S$." ..
20043 from   0  97 61 'a' "ccur"  1 bits [3796 in 5184]  "aoue in-©s(S$." ..
20044 from   1 111 6f 'o' "ccur"  4 bits [ 527 in 5200]  "oaue in-©s(S$." ..
20045 from   0 111 6f 'o' "ccur"  0 bits [3812 in 5216]  "oaue in-©s(S$." ..
20046 from   6 110 6e 'n' "ce©."  6 bits [  70 in 5232]  "noaue i-©s(S$." ..
20047 from  13  46 2e '.' "ce©A" 11 bits [   4 in 5248]  ".noaue i-©s(S$" ..
20048 from   0  46 2e '.' "ce©A"  0 bits [3828 in 5264]  ".noaue i-©s(S$" ..
20049 from   0  46 2e '.' "ce©I"  1 bits [3844 in 5280]  ".noaue i-©s(S$" ..
20050 from   1 110 6e 'n' "ce©T"  3 bits [ 543 in 5296]  "n.oaue i-©s(S$" ..
20051 from   0 110 6e 'n' "ce©a"  1 bits [3860 in 5312]  "n.oaue i-©s(S$" ..
20052 from   0 110 6e 'n' "ce©c"  0 bits [3876 in 5328]  "n.oaue i-©s(S$" ..
20053 from   7 105 69 'i' "ce©i"  8 bits [  19 in 5344]  "in.oaue -©s(S$" ..
20054 from   1 110 6e 'n' "ce©w"  3 bits [ 559 in 5360]  "ni.oaue -©s(S$" ..
20055 from   0 110 6e 'n' "ce $"  0 bits [3892 in 5376]  "ni.oaue -©s(S$" ..
20056 from   0 110 6e 'n' "ce $"  1 bits [3908 in 5392]  "ni.oaue -©s(S$" ..
20057 from   0 110 6e 'n' "ce $"  0 bits [3924 in 5408]  "ni.oaue -©s(S$" ..
20058 from   0 110 6e 'n' "ce ("  1 bits [3940 in 5424]  "ni.oaue -©s(S$" ..
20059 from   0 110 6e 'n' "ce \"  1 bits [3956 in 5440]  "ni.oaue -©s(S$" ..
20060 from   0 110 6e 'n' "ce a"  0 bits [3972 in 5456]  "ni.oaue -©s(S$" ..
20061 from   0 110 6e 'n' "ce a"  0 bits [3988 in 5472]  "ni.oaue -©s(S$" ..
20062 from   4  97 61 'a' "ce a"  8 bits [  39 in 5488]  "ani.oue -©s(S$" ..
20063 from   2 105 69 'i' "ce b"  5 bits [ 177 in 5504]  "ian.oue -©s(S$" ..
20064 from  19 114 72 'r' "ce b" 13 bits [   1 in 5520]  "rian.oue -©s(S" ..
20065 from   3 110 6e 'n' "ce b"  5 bits [  98 in 5536]  "nria.oue -©s(S" ..
20066 from   1 114 72 'r' "ce c"  3 bits [ 575 in 5552]  "rnia.oue -©s(S" ..
20067 from   1 110 6e 'n' "ce c"  3 bits [ 591 in 5568]  "nria.oue -©s(S" ..
20068 from   0 110 6e 'n' "ce e"  1 bits [4004 in 5584]  "nria.oue -©s(S" ..
20069 from   6 117 75 'u' "ce e"  6 bits [  86 in 5600]  "unria.oe -©s(S" ..
20070 from   4  97 61 'a' "ce i"  6 bits [  55 in 5616]  "aunri.oe -©s(S" ..
20071 from   2 110 6e 'n' "ce i"  6 bits [ 193 in 5632]  "nauri.oe -©s(S" ..
20072 from   3 114 72 'r' "ce i"  5 bits [ 114 in 5648]  "rnaui.oe -©s(S" ..
20073 from   1 110 6e 'n' "ce i"  3 bits [ 607 in 5664]  "nraui.oe -©s(S" ..
20074 from   0 110 6e 'n' "ce i"  0 bits [4020 in 5680]  "nraui.oe -©s(S" ..
20075 from   0 110 6e 'n' "ce n"  1 bits [4036 in 5696]  "nraui.oe -©s(S" ..
20076 from   0 110 6e 'n' "ce o"  1 bits [4052 in 5712]  "nraui.oe -©s(S" ..
20077 from   0 110 6e 'n' "ce o"  0 bits [4068 in 5728]  "nraui.oe -©s(S" ..
20078 from   0 110 6e 'n' "ce o"  1 bits [4084 in 5744]  "nraui.oe -©s(S" ..
20079 from   0 110 6e 'n' "ce o"  0 bits [4100 in 5760]  "nraui.oe -©s(S" ..
20080 from   0 110 6e 'n' "ce o"  0 bits [4116 in 5776]  "nraui.oe -©s(S" ..
20081 from   0 110 6e 'n' "ce o"  1 bits [4132 in 5792]  "nraui.oe -©s(S" ..
20082 from   0 110 6e 'n' "ce o"  0 bits [4148 in 5808]  "nraui.oe -©s(S" ..
```

```
20083 from   1 114 72 'r' "ce p"  4 bits [ 623 in 5824]  "rnaui.oe -©s(S" ..
20084 from   3 117 75 'u' "ce t"  6 bits [ 130 in 5840]  "urnai.oe -©s(S" ..
20085 from   3  97 61 'a' "ce t"  5 bits [ 146 in 5856]  "aurni.oe -©s(S" ..
20086 from   3 110 6e 'n' "ce t"  5 bits [ 162 in 5872]  "nauri.oe -©s(S" ..
20087 from   0 110 6e 'n' "ce t"  0 bits [4164 in 5888]  "nauri.oe -©s(S" ..
20088 from   0 110 6e 'n' "ce t"  1 bits [4180 in 5904]  "nauri.oe -©s(S" ..
20089 from   0 110 6e 'n' "ce t"  0 bits [4196 in 5920]  "nauri.oe -©s(S" ..
20090 from   1  97 61 'a' "ce t"  4 bits [ 639 in 5936]  "anuri.oe -©s(S" ..
20091 from   1 110 6e 'n' "ce t"  3 bits [ 655 in 5952]  "nauri.oe -©s(S" ..
20092 from   1  97 61 'a' "ce t"  3 bits [ 671 in 5968]  "anuri.oe -©s(S" ..
20093 from   2 117 75 'u' "ce t"  4 bits [ 209 in 5984]  "uanri.oe -©s(S" ..
20094 from   0 117 75 'u' "ce t"  1 bits [4212 in 6000]  "uanri.oe -©s(S" ..
20095 from   2 110 6e 'n' "ce t"  5 bits [ 225 in 6016]  "nuari.oe -©s(S" ..
20096 from   1 117 75 'u' "ce t"  3 bits [ 687 in 6032]  "unari.oe -©s(S" ..
20097 from   1 110 6e 'n' "ce t"  3 bits [ 703 in 6048]  "nuari.oe -©s(S" ..
20098 from   0 110 6e 'n' "ce t"  1 bits [4228 in 6064]  "nuari.oe -©s(S" ..
20099 from   0 110 6e 'n' "ce t"  0 bits [4244 in 6080]  "nuari.oe -©s(S" ..
20100 from   0 110 6e 'n' "ce t"  1 bits [4260 in 6096]  "nuari.oe -©s(S" ..
20101 from   2  97 61 'a' "ce t"  5 bits [ 241 in 6112]  "anuri.oe -©s(S"
20102 from   1 110 6e 'n' "ce w"  3 bits [ 719 in 6128]  "nauri.oe -©s(S"
20103 from   0 110 6e 'n' "ce w"  0 bits [4276 in 6144]  "nauri.oe -©s(S"
20104 from   0 110 6e 'n' "ce w"  1 bits [4292 in 6160]  "nauri.oe -©s(S"
20105 from   0 110 6e 'n' "ce w"  1 bits [4308 in 6176]  "nauri.oe -©s(S"
20106 from   0 110 6e 'n' "ce w"  0 bits [4324 in 6192]  "nauri.oe -©s(S"
20107 from   0 110 6e 'n' "ce w"  0 bits [4340 in 6208]  "nauri.oe -©s(S"
20108 from   0 110 6e 'n' "ce w"  1 bits [4356 in 6224]  "nauri.oe -©s(S"
20109 from   0 110 6e 'n' "ce"©"  0 bits [4372 in 6240]  "nauri.oe -©s(S"
20110 from   0 110 6e 'n' "ce) "  1 bits [4388 in 6256]  "nauri.oe -©s(S"
20111 from   0 110 6e 'n' "ce, "  0 bits [4404 in 6272]  "nauri.oe -©s(S"
20112 from   0 110 6e 'n' "ce, "  1 bits [4420 in 6288]  "nauri.oe -©s(S"
20113 from   3 114 72 'r' "ce, "  5 bits [ 178 in 6304]  "rnaui.oe -©s(S"
20114 from   1 110 6e 'n' "ce, "  3 bits [ 735 in 6320]  "nraui.oe -©s(S"
20115 from   4 105 69 'i' "ce, "  6 bits [  71 in 6336]  "inrau.oe -©s(S"
20116 from   1 110 6e 'n' "ce, "  3 bits [ 751 in 6352]  "nirau.oe -©s(S"
20117 from   1 105 69 'i' "ce.©"  4 bits [ 767 in 6368]  "inrau.oe -©s(S"
20118 from   1 110 6e 'n' "ce.©"  3 bits [ 783 in 6384]  "nirau.oe -©s(S"
20119 from   5  46 2e '.' "ce4©"  6 bits [  62 in 6400]  ".nirauoe -©s(S"
20120 from   2 105 69 'i' "ce:©"  5 bits [ 257 in 6416]  "i.nrauoe -©s(S"
20121 from   0 105 69 'i' "ce;©"  1 bits [4436 in 6432]  "i.nrauoe -©s(S"
20122 from   2 110 6e 'n' "ce; "  4 bits [ 273 in 6448]  "ni.rauoe -©s(S"
20123 from   4  97 61 'a' "ced "  6 bits [  87 in 6464]  "ani.ruoe -©s(S"
20124 from   5 117 75 'u' "ced "  7 bits [  78 in 6480]  "uani.roe -©s(S"
20125 from   1  97 61 'a' "ced "  2 bits [ 799 in 6496]  "auni.roe -©s(S"
20126 from   0  97 61 'a' "ced "  1 bits [4452 in 6512]  "auni.roe -©s(S"
20127 from   1 117 75 'u' "ced "  4 bits [ 815 in 6528]  "uani.roe -©s(S"
20128 from   0 117 75 'u' "ced "  0 bits [4468 in 6544]  "uani.roe -©s(S"
20129 from   7 101 65 'e' "cede"  7 bits [  35 in 6560]  "euani.ro -©s(S"
20130 from   7 111 6f 'o' "cedu"  8 bits [  51 in 6576]  "oeuani.r -©s(S"
20131 from   0 111 6f 'o' "cedu"  0 bits [4484 in 6592]  "oeuani.r -©s(S"
20132 from   0 111 6f 'o' "cedu"  1 bits [4500 in 6608]  "oeuani.r -©s(S"
20133 from   0 111 6f 'o' "cedu"  0 bits [4516 in 6624]  "oeuani.r -©s(S"
20134 from   0 111 6f 'o' "cedu"  0 bits [4532 in 6640]  "oeuani.r -©s(S"
20135 from   0 111 6f 'o' "cedu"  1 bits [4548 in 6656]  "oeuani.r -©s(S"
20136 from   0 111 6f 'o' "cedu"  1 bits [4564 in 6672]  "oeuani.r -©s(S"
```

```
20137 from   0 111 6f 'o' "cedu"  0 bits [4580 in 6688]  "oeuani.r -©s(S"
20138 from   0 111 6f 'o' "cedu"  1 bits [4596 in 6704]  "oeuani.r -©s(S"
20139 from   0 111 6f 'o' "cedu"  0 bits [4612 in 6720]  "oeuani.r -©s(S"
20140 from   0 111 6f 'o' "cedu"  1 bits [4628 in 6736]  "oeuani.r -©s(S"
20141 from  45 120 78 'x' "ceed" 13 bits [   1 in 6752]  "xoeuani.r -©s("
20142 from   0 120 78 'x' "ceed"  1 bits [4644 in 6768]  "xoeuani.r -©s("
20143 from   0 120 78 'x' "ceed"  0 bits [4660 in 6784]  "xoeuani.r -©s("
20144 from   0 120 78 'x' "ceed"  1 bits [4676 in 6800]  "xoeuani.r -©s("
20145 from   1 111 6f 'o' "ceed"  2 bits [ 831 in 6816]  "oxeuani.r -©s("
20146 from   0 111 6f 'o' "ceed"  0 bits [4692 in 6832]  "oxeuani.r -©s("
20147 from   2 101 65 'e' "ceiv"  5 bits [ 289 in 6848]  "eoxuani.r -©s("
20148 from   0 101 65 'e' "ceiv"  1 bits [4708 in 6864]  "eoxuani.r -©s("
20149 from   0 101 65 'e' "ceiv"  1 bits [4724 in 6880]  "eoxuani.r -©s("
20150 from   0 101 65 'e' "ceiv"  0 bits [4740 in 6896]  "eoxuani.r -©s("
20151 from   0 101 65 'e' "ceiv"  0 bits [4756 in 6912]  "eoxuani.r -©s("
20152 from  28  99 63 'c' "cele" 13 bits [   2 in 6928]  "ceoxuani.r -©s"
20153 from   3 120 78 'x' "cell"  4 bits [ 194 in 6944]  "xceouani.r -©s"
20154 from   0 120 78 'x' "cell"  1 bits [4772 in 6960]  "xceouani.r -©s"
20155 from   2 101 65 'e' "cenc"  5 bits [ 305 in 6976]  "excouani.r -©s"
20156 from  13 115 73 's' "cend"  8 bits [  20 in 6992]  "sexcouani.r -©"
20157 from   1 101 65 'e' "cent"  3 bits [ 847 in 7008]  "esxcouani.r -©"
20158 from   0 101 65 'e' "cent"  1 bits [4788 in 7024]  "esxcouani.r -©"
20159 from   3  99 63 'c' "cent"  5 bits [ 210 in 7040]  "cesxouani.r -©"
20160 from   7 110 6e 'n' "cept"  7 bits [  67 in 7056]  "ncesxouai.r -©"
20161 from   4 120 78 'x' "cept"  6 bits [ 103 in 7072]  "xncesouai.r -©"
20162 from   3 101 65 'e' "cept"  4 bits [ 226 in 7088]  "exncsouai.r -©"
20163 from   0 101 65 'e' "cept"  1 bits [4804 in 7104]  "exncsouai.r -©"
20164 from   0 101 65 'e' "cept"  1 bits [4820 in 7120]  "exncsouai.r -©"
20165 from   2 110 6e 'n' "cept"  4 bits [ 321 in 7136]  "nexcsouai.r -©"
20166 from   0 110 6e 'n' "cept"  1 bits [4836 in 7152]  "nexcsouai.r -©"
20167 from   6 117 75 'u' "cera"  6 bits [ 102 in 7168]  "unexcsoai.r -©"
20168 from   0 117 75 'u' "cera"  0 bits [4852 in 7184]  "unexcsoai.r -©"
20169 from   1 110 6e 'n' "cern"  4 bits [ 863 in 7200]  "nuexcsoai.r -©"
20170 from   3 120 78 'x' "cerp"  5 bits [ 242 in 7216]  "xnuecsoai.r -©"
20171 from   0 120 78 'x' "cerp"  0 bits [4868 in 7232]  "xnuecsoai.r -©"
20172 from  11  32 20 ' ' "cert"  8 bits [  28 in 7248]  " xnuecsoai.r-©"
20173 from   2 110 6e 'n' "ces©"  4 bits [ 337 in 7264]  "n xuecsoai.r-©"
20174 from   0 110 6e 'n' "ces©"  1 bits [4884 in 7280]  "n xuecsoai.r-©"
20175 from   3 117 75 'u' "ces "  5 bits [ 258 in 7296]  "un xecsoai.r-©"
20176 from   1 110 6e 'n' "ces "  3 bits [ 879 in 7312]  "nu xecsoai.r-©"
20177 from   8  97 61 'a' "ces "  8 bits [  34 in 7328]  "anu xecsoi.r-©"
20178 from   2 117 75 'u' "ces "  4 bits [ 353 in 7344]  "uan xecsoi.r-©"
20179 from   0 117 75 'u' "ces "  0 bits [4900 in 7360]  "uan xecsoi.r-©"
20180 from   0 117 75 'u' "ces "  1 bits [4916 in 7376]  "uan xecsoi.r-©"
20181 from   0 117 75 'u' "ces "  0 bits [4932 in 7392]  "uan xecsoi.r-©"
20182 from   2 110 6e 'n' "ces""  5 bits [ 369 in 7408]  "nua xecsoi.r-©"
20183 from   8 111 6f 'o' "cess"  7 bits [  50 in 7424]  "onua xecsi.r-©"
20184 from   7  99 63 'c' "cess"  7 bits [  83 in 7440]  "conua xesi.r-©"
20185 from   0  99 63 'c' "cess"  1 bits [4948 in 7456]  "conua xesi.r-©"
```