# Block Sorting Text Compression

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

*peter-f@cs.auckland.ac.nz*

## Abstract.

*A recent development in text compression is a "block sorting" algorithm which permutes the input text according to a special sort procedure and then processes the permuted text with Move-To-Front and a final statistical compressor. The technique is fast, with a compression performance ranking it among the best of the known compressors.*

*This paper describes work on the block sorting algorithm, especially establishing its relation to other compressors and attempting to improve its compression performance. It is already known to be a form of statistical compressor with unbounded contexts; we show that the contexts are so completely restructured by the sorting that many standard file compression techniques are no longer appropriate. Various approaches are investigated in an attempt to improve the compression, most of which involve a hierarchy of coding models to allow fast adaptation to local contexts. The better coding techniques include one derived from work of Shannon in 1951 in establishing the entropy of English text, while the best employs a novel model especially designed for skewed probability distributions.*

*The work in this paper confirms block-sorting as a viable text compression technique, with a compression approaching that of the currently best compressors while being much faster than many other compressors of comparable performance.*

**Keywords** Text compression, statistical compression, block sorting

## 1. Introduction.

A very recent development in text compression is a "Block Sorting" algorithm (or "block reduction"), reported by Burrows and Wheeler[4]. It considers the text in blocks, which may be as large as 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. The realisation is quite different from traditional text compressors and raises some interesting questions, including its relation to other compressors, the statistics of the symbols to be compressed, and the possibility of alternatives to the MTF and Huffman stages.

**Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31 – February 2 1996.**

Much of the material of this paper has been presented in two Technical Reports [7, 8], to which readers are referred for more detailed information, tables of results, and logs of actual output.

Most good 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 [1] that these two apparently different compression techniques are in fact equivalent. More recently, Cleary et al [6] 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[2] has examined the structure of the Dynamic Markov Coding compressor (DMC). These results have been coordinated with those of Cleary to demonstrate an equivalence of all the established text compression techniques. Block sorting is therefore a member of a well-established family of text compressors, even if its precise position and qualities remain uncertain.

## 2. The block-sorting algorithm

Burrows and Wheeler [4] 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 correlation 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. By contrast, the block sorting algorithm considers each symbol in relation to its *following* context, rather than the more conventional *preceding* context.

### 2.1 Forward transformation

The symbols of a text block to be compressed (part or all of the file) are first sorted, using as a sort key for each symbol the immediately following symbols, to whatever length is needed to resolve the comparison, and wrapping from the end of the file back to the beginning. The output of this stage is a permutation of all of the symbols of the original file, together with the position of the last symbol of the file. 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 before 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 Reverse transformation

Recovery of the data requires first a decompression to recover the output of the original sorting permutation. Reversing the permutation of these symbols depends on the observations that

(a) the recovered (or transmitted) data contains all of the original symbols, and

(b) 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 and obtain its corresponding context symbol as the next symbol. The position of the symbol corresponding to the first context is needed to locate the last symbol of the output. From there we can traverse the entire transmitted data to recover the original text.

| symbol | context | Index | symbol | context | link |
|---|---|---|---|---|---|
| p | imississip | 1 | p | i… | 5 |
| s | ippimissis | 2 | s | i… | 7 |
| s | issippimis | 3 | s | i… | 10 |
| m | ississippi | 4 | m | i… | 11 |
| → i | mississipp | 5 | i | m… | 4 |
| p | pimississi | 6 | p | p… | 1 |
| i | ppimississ | 7 | i | p… | 6 |
| s | sippimissi | 8 | s | s… | 2 |
| s | sissippimi | 9 | s | s… | 3 |
| i | ssippimiss | 10 | i | s… | 8 |
| i | ssissippim | 11 | i | s… | 9 |

Figure 1. The forward and reverse transformations

To illustrate the operations of coding and decoding we consider the text "`mississippi`" as shown in Figure 1. The first context is "`imississip`" for symbol "`p`", the second is "`ippimissis`" for symbol "`s`", and so on. The permuted text is then "`pssmipissii`", and the initial index is 5 (marked with "→"), because the fifth context corresponds to the original text.

To decode we take the string "`pssmipissii`", sort it to build the initial letters of the contexts ("`iiiimppssss`") and then build the links shown in the last column. The four "`i…`" contexts link to the four "`i`" input symbols, first to first, second to second, and so on (to 5, 7, 10 and 11). The "`m…`" context links to the only "`m`" symbol, and the two "`p…`"s and four "`s…`"s link to their partners in order.

To finally recover the text, we start at the indicated position (5) and immediately link to 4. The sorted received symbol there yields the desired symbol "`m`". We then link to 11 get the "`i`", and so on for the rest of the data, stopping on a symbol count or return to the start of the file.

## 3. Order-0 implementation.

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. The three stages are therefore —

- the initial sorting transformation, yielding a permuted version of the input text,
- a Move-To-Front processing of the permuted text,
- a final statistical compression, using order-0 arithmetic coding.

The immediate results are given in Table 1, testing on the Calgary Corpus [1] and using PPMC [9] as a reference compressor. (All of the compression values are given in output bits per input byte.)

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 the Move-To-Front list and are emitted as zeros. The number of string comparisons is a useful measure of the work required by the initial sort. Finally we have the average number of data comparisons for each of the compares of the previous column. (This value could not be obtained from this initial version, but came from later versions with word comparisons and run-length encoding of the input. The number of bytes compared is about 4 times the value shown,

[1] The files of the Calgary compression corpus are available by anonymous FTP from `ftp.cpsc.ucalgary.ca:/pub/projects/text.compression.corpus/textcompression.corpus.tar.Z`

| File | size bytes | PPMC (1990) | bs Order0 | MTF dist non-0 | frac 0 MTF | compares | Avg. compare length (words) |
|---|---|---|---|---|---|---|---|
| **Bib** | 111,261 | 2.11 | 2.13 | 5.50 | 66.8% | 1,704,645 | 1.65 |
| **Book1** | 768,771 | 2.48 | 2.52 | 3.88 | 49.8% | 15,974,996 | 1.25 |
| **Book2** | 610,856 | 2.26 | 2.20 | 4.20 | 60.8% | 12,139,721 | 1.42 |
| **Geo** | 102,400 | 4.78 | 4.81 | 55.63 | 35.8% | 1,435,323 | 1.07 |
| **News** | 377,109 | 2.65 | 2.68 | 7.65 | 57.9% | 6,789,971 | 1.66 |
| **Obj1** | 21,504 | 3.76 | 4.23 | 46.82 | 50.6% | 231,884 | 1.51 |
| **Obj2** | 246,814 | 2.69 | 2.71 | 30.22 | 68.1% | 3,900,488 | 1.83 |
| **Paper1** | 53,161 | 2.48 | 2.61 | 6.45 | 58.4% | 734,781 | 1.31 |
| **Paper2** | 82,199 | 2.45 | 2.57 | 5.06 | 55.4% | 1,274,109 | 1.24 |
| **Pic** | 513,216 | 1.09 | 0.92 | 3.39 | 87.4% | 47,889,672 | 1.36 |
| **ProgC** | 39,611 | 2.49 | 2.67 | 8.32 | 60.3% | 511,849 | 1.37 |
| **ProgL** | 71,646 | 1.90 | 1.84 | 4.63 | 72.9% | 1,055,533 | 2.06 |
| **ProgP** | 49,379 | 1.84 | 1.82 | 5.54 | 74.0% | 673,846 | 3.25 |
| **Trans** | 93,695 | 1.77 | 1.60 | 5.66 | 79.2% | 1,329,847 | 3.51 |
| **AVG** | | 2.48 | 2.52 | 13.78 | 62.7% | | |

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

plus 2 for those implicit in the bucket sort.)

The most obvious result is that the compressor is already very good – within about 1.6% 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. 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 have 87% and 79% respectively. A detailed log of these tests is given in [7] Appendix II.

With the compressor implemented, there were many matters to be investigated and understood while attempting to improve the performance. These included –

- Were there variations on the sorting permutation or the MTF operation which would improve compression?
- Would final stages other than MTF with Huffman or arithmetic coding provide better compression?
- What is really happening to the data as it moves through the stages of sorting, MTF and final coding?
- Compressors rely on the underlying structure of their input data. What is the structure of the MTF codes to the final, compression, stage ?
- What coding methods are appropriate to that structure? It was apparent from very early work that the techniques of conventional compression we unsuitable; it was equally unclear just what methods were more suitable.
- Some files, and PIC in particular, needed extremely long times to sort. Why was this, and how could their speed be improved?

These topics are discussed in the remainder of the paper. The next few sections are devoted to detailed examinations of some of these areas, and a "new" compression technique is then given before actual compression improvements are discussed in section 8.

In all of this discussion it is important to remember that the initial implementation of block sorting already gives excellent compression and that any improvements are relatively small. The best methods of this paper improve the compression by only about 7%; another 3% improvement would rank it with the best compressors published to date.

## 4. Initial Results

Compression depends on both the frequencies of the symbols being compared and on the contexts or other

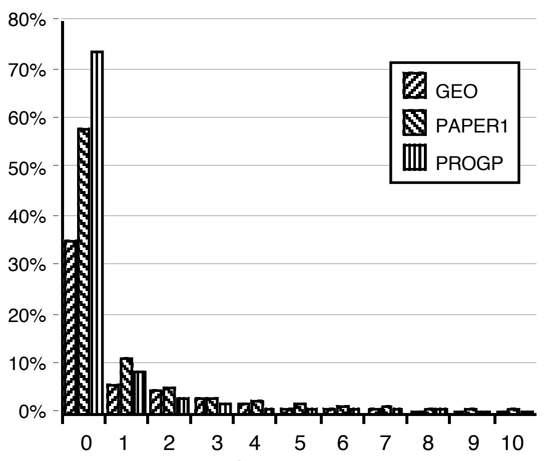

Figure 2.
Order-0 probabilities of MTF symbols

interrelations between those symbols. For now we look at only the symbol frequencies. The distribution of the Move-To-Front frequencies is shown in Figure 2 for three of the files – GEO (less compressible), PAPER1 (representative text), and PROGP (quite compressible).

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

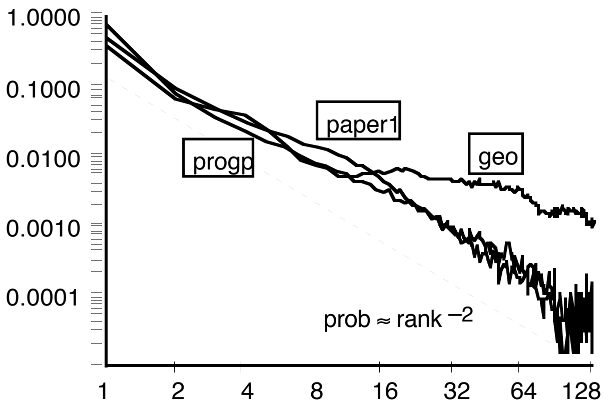

Figure 3.
Probabilities of the first 128 MTF codes for three files

Figure 3 shows the MTF code probabilities, with logarithmic scales and for the first 128 codes (which includes all symbols of ASCII text files). The rank scale is 1-origin to allow the first value to be represented. The probabilities for PAPER1 and PROGP are approximately proportional to $rank^{-2}$, and GEO is similar for low ranks. This may be compared with Zipf's law for natural language, which has a $rank^{-1}$ dependency, indicating the effect of the sorting and MTF operations. There is little difference between PAPER1 and the more-compressible PROGP on this scale, but PROGP has a higher frequency for the rank=1 and generally lower probabilities for higher ranks. With GEO the less frequent symbols have probabilities of about 0.001 – 0.005, much as would be expected for an approximately uniformly distributed population of 256 symbols.

Other measurements, presented in [7], show that MTF_code=0 is generally emitted at about 0.5 bit/symbol, rising to about 3 bits for MTF_code=1, with a gradual increase thereafter. For the more compressible files, about 15% of the bits are emitted for MTF_codes of 0 and 1, with about half of the bits being emitted for codes less than 5 or 6.

Much of the improvement 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 %. Some improvement in coding high ranks comes from the "structured model" in section 9.

One of the motivations for this work was the realisation that the original algorithm achieved excellent compression with a Move-To-Front and Huffman 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. Applying the output of the MTF stage to various good text compressors showed that the "better" the compressor the worse the final compression! The reason is that all good compressors (whether dictionary or statistical) rely on symbol contexts for their operation and that the sorting phase destroys the context structure on which those compressors rely. This aspect will be discussed later.

Another possibility (mentioned by Burrows and Wheeler) is that the MTF operation might be tuned, perhaps by moving symbols to near the head of the MTF list rather than the very head. This was found to be unsuccessful. The least-compressible files improve by about 0.1% and other files give poorer compression.

## 5. Improving the sort performance

A crucial step in the compression algorithm is the sort phase to reorder the input text. While Burrows and Wheeler devote a great deal of their report [4] to sort optimisation, the present experience is that adequate results can be achieved with the standard C *qsort* routine and a 65,536-way radix sort based on the first two bytes of each comparand. (The present work has concentrated on understanding the technique, rather than achieving high speed.)

The file PIC is not handled well by this scheme — it has long runs of 0s and over 80% of it is placed in a single radix-sort bucket! Another major problem is that the long runs of 0s may require comparisons of thousands of bytes to resolve, leading to very long execution times (9 minutes for PIC). These problems are solved by first run-encoding the input text, following each run of 4 identical symbols with an in-line count of the remaining symbols in the run (the count may be 0). The very compressible runs are thereby turned into an almost incompressible combination of random codes and most files have their compression reduced by about 0.1%. About 75% of PIC is absorbed in the runs and its compression improves by about 10%. More importantly, its sort time is reduced from about 9 minutes to 12 seconds. Similar problems of slow sorting arise from files with strings of very long repetitious sequences such as "…aaaabaaaabaaaabaaaab…". It is shown in [8] how sorting speed can be handled by an LZ-77 type of preprocessor, but at a considerable cost in compression for most files.

Some further improvements are still possible. The radix sort buckets have very uneven loading —

for example a text file with an alphabet of 100 symbols will use only 10,000 of the available 65,536 buckets, leaving 85% of the buckets empty. With about 3% of all digraphs going into the single "e " bucket, that bucket is considerably overloaded. A second level of radix sorting could be useful, especially using the techniques of Chen and Rief [5] to expedite the sorting of low-entropy files.

The number of comparison operations is reduced by collecting 4 8-bit bytes into 32-bit words, striping bytes across successive words and striding across 4 words between comparisons. Each raw comparison operation therefore compares 4 bytes, usually taking no longer than a simple 1-byte character compare.

Burrows and Wheeler go to considerable trouble in their reports [4, 13] to provide a fast sort and it is interesting to note that Burrows [3] reports that he has abandoned techniques as described here in favour of improvements to those techniques. Nevertheless, the combination of qsort and radix sorting seems generally adequate and has been retained here.

Wheeler's latest implementation [13] ("*bred*", see section 10) uses some other interesting sorting techniques. He uses a 256-way radix sort and initially stripes bytes across 32-bit words, much as described above. The sort buckets (or "groups") are then sorted in order of size, smaller groups first. As each group is completed the low-order 24 bits of each newly-sorted word are replaced by its index in the sort group. This ensures that each sorted symbol is uniquely tagged and comparison strings which include it are resolved immediately. With a sort routine specifically designed for the application, rather than calling on a compare procedure for each comparison, the result of these improvements is a very fast sort indeed.

## 6. The arithmetic coding routines

This work was started with the traditional "CACM" arithmetic coding routines[1,14]. More recently, improved versions of arithmetic coding routines have been described[10], which will be referred as the "DCC95" routines. These routines are faster and much better for large alphabets and also include optimisations for binary alphabets, which is useful in some of the coders to be described later.

Unfortunately, when the DCC95 routines were included and tested on the Order-0 coding described above, they gave markedly worse compression than did the older CACM routines! The reason lies in the handling of the symbol frequencies and in the nature of contexts in different compression algorithms.

In both types of arithmetic coding we keep integer frequency counts for each symbol – when the total count exceeds some maximum value all of the counts are halved to keep the total within range. In the older CACM routines the increment remains constant (usually 1) and older counts are effectively decayed and lose significance at each halving. As the maximum count is small compared with the size of many files, there is frequent rescaling and a measure of adaptation to the more recent part of the data.

In the DCC95 routines, the maximum counts are much larger (typically 200 million rather than 16,000). The increment is also much larger (initially comparable to the maximum count) and is halved when the data counts are halved. Older and newer counts are thus treated uniformly, with no bias in favour of more recent symbols. This behaviour is appropriate for PPM-style compressors where statistics are reasonably assumed constant over the whole file, and we may be developing many models in parallel, one for each context. Equal treatment of all symbols is quite inappropriate for handling block-sorted text where considerable local adaptation is essential. Changing the DCC95 routines to allow increments to be handled as for the older CACM routines restored the expected performance.

The arithmetic compressor 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.

## 7. Shannon's coding model

In one of the first papers relating to text compression, Shannon in 1951 [11] used a coding model in which a test subject guessed at the next symbols, given a block of preceding text. While this paper gave the well-known limit of 0.6 – 1.3 bits/letter as the entropy of English text, we are more interested in his coder.

The coder, shown in Figure 4, 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

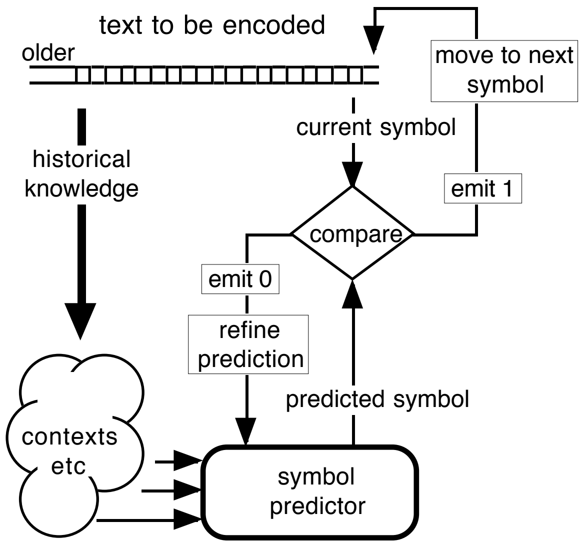

Figure 4. Shannon's symbol encoder

according to the transmitted instructions, is able to track the coder predictor and eventually arrive at the correct symbol. The prediction is an ordered list of symbols, from most probable to least probable. The number of wrong estimates is just the position of the symbol in the ordered list – in effect a measure of the error in the estimate. To recognise its historical importance we propose the term "Shannon model" for the technique. (The technique has obvious parallels to the well-known predictive coding or delta coding methods of analogue data encoding.)

Prediction of symbol ranking is essentially what the block sorting algorithm does, although with a permutation of the input text to increase locality effects. The MTF list approximates an ordering in symbol frequency, and the emitted index is simply an error indication.

Thus the block-sorting compressor with MTF processing is very close to the proposed "Shannon" compression mechanism.

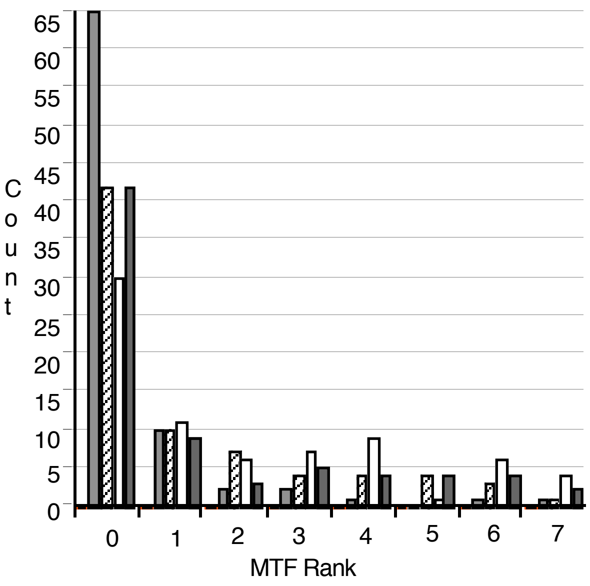

Figure 5. MTF counts in successive samples

## 8. Compression improvements

The "order-0" statistical model of the MTF output is at best only an approximation or averaging-out of the local contexts, probably with considerable local deviation from that model. Figure 5 shows the frequencies of the first few MTF ranks for four successive samples of 100 symbols after MTF coding the file PAPER1. There are significant differences between adjacent samples; even in these few samples, the counts vary from 30 to 65 for 0, 3 to 7 for 2 and 1 to 9 for 4, showing the large local variations from the overall "average distribution". There is little correlation between the frequencies of differing ranks. It is never greater than 65% and usually less than 20%. The differences are a natural consequence of having quite unrelated contexts following one another in quick succession, each with its own "signature" or combination of MTF probabilities.

Improving compression over that achieved by the order-0 model requires models which can adapt quickly to local changes in frequency, especially for the first half dozen or so MTF codes. With adaptive arithmetic coding this requires a model containing only a few symbols and with a small count limit, so that statistics are sensitive to just a few added symbols and there is frequent rescaling to provide locality.

One approach is to use a small "cache" model which holds only the first few, or most probable, MTF codes, escaping to a complete, background, model for other values. These results are reported in [8]; the coders achieved 2.42 bits/byte, which is very little better than the 2.43 of the original block-sorting [4] or the 2.52 of the simple order-0 model.

With much of the MTF output being simply runs of zeros, it seems reasonable to try run length coding of the zero values. Unless the coding is done very carefully indeed, experience is that there may be little advantage over a simple arithmetic coding of the run itself. In very general terms for a run of length $N$, coding the length requires about $\log_2 N$ bits, plus any overhead for signalling the run. With a fully-adapted arithmetic coder, coding the symbols of the run requires $N\log_2 (N+1)/N$ bits, while the run termination requires about $\log_2 N$ bits. Both cases are dominated by the $\log_2 N$ term and the two approaches have a similar cost. Neither is clearly superior and this is borne out in practice. Wheeler [13] describes an interesting method of run-length encoding which does seem to be beneficial and will be described later.

A possible disadvantage of the MTF coding is that symbols lose their identity; it was felt that there might be some advantage in directly coding the sorted output. The best of these coders had the normal alphabet extended by one symbol to denote a repetition of the previous symbol. It achieved 2.51 bits/byte, or essentially the same as the simple order-0 coding of the MTF output. (It may be regarded as an implementation of a variant of the Shannon coder, described in Shannon's paper, where the response to the suggestion is either "yes" or the correct symbol.)

Good compression was achieved with coders based on the Shannon coding scheme described earlier. The complete development is given in [8]; here we describe only the best of the compressors. It consists of a series of interacting arithmetic coding models, most of which handle the MTF output regarded as a simple unary code.

1. If the last MTF code was a 0, a binary (2 symbol) model emits either a 0 (ie as part of a run of 0s) or the first 1 of the next unary code.
2. The first few 1s of the unary code, and the terminating 0 if it is a small value, are emitted by

| MTF | zero | "one" models | | | | full | emitted |
|---|---|---|---|---|---|---|---|
| code | model | 1 | 2 | 3 | 4 | model | code |
| 0 | 0 | | | | | | 0 |
| 4 | 1 | | 1 | 1 | 1 | 4 | 1111<4> |
| 2 | | 1 | 1 | 0 | | | 110 |
| 19 | | 1 | 1 | 1 | 1 | 19 | 1111<19> |
| 3 | | 1 | 1 | 1 | 0 | | 1110 |
| 1 | | 1 | 0 | | | | 10 |
| 1 | | 1 | 0 | | | | 10 |
| 0 | | 0 | | | | | 0 |
| 6 | 1 | | 1 | 1 | 1 | 6 | 1111<6> |
| 4 | | 1 | 1 | 1 | 1 | 4 | 1111<4> |
| 2 | | 1 | 1 | 0 | | | 110 |
| 3 | | 1 | 1 | 1 | 0 | | 1110 |
| 1 | | 1 | 0 | | | | 10 |
| 0 | | 0 | | | | | 0 |
| 0 | 0 | | | | | | 0 |
| 0 | 0 | | | | | | 0 |

Figure 6.
Sample codes emitted by the Shannon encoder

a series of binary models, one for the first digit, another for the second digit, and so on.

3. For larger values (4 or greater), the actual coded value is emitted from a full-alphabet model. A value in this range will be coded as 1111xxxxx…

Figure 6 shows the sequence of codes emitted for each of a set of MTF codes, and the models from which those codes are emitted. Each of the binary models (the "zero" and "one" models) has a small limit and relatively large increment to force rapid adjustment to the local environment. This coder compresses the Calgary corpus to an average of 2.36 bits/byte.

## 9. A structured coding model

One problem with the simple arithmetic coding model is that it must represent a range of symbol probabilities covering 4 or 5 orders of magnitude. Simpler arithmetic coders cannot cope at all with such a range, while others may take several file life-times to adjust properly to the statistics.

A second point is that the MTF code distribution is quite different from that expected by conventional compressors. Most coders are expected to handle probability distributions which resemble a line spectrum, having a few arbitrarily-placed "spikes" of the more probable symbols rising above a general background "noise". The MTF distribution is generally much more regular, with the zero value being most probable and a usually monotonic decrease in probabilities going to larger values.

To recognise these aspects, we use a hierarchical model where the first level entries cover approximately octave ranges of symbol rank, as shown in Figure 7. At the low-rank end it tends to act as a cache with unique values or few values in each entry of the first-level model. At the high-rank end it is sensitive to the

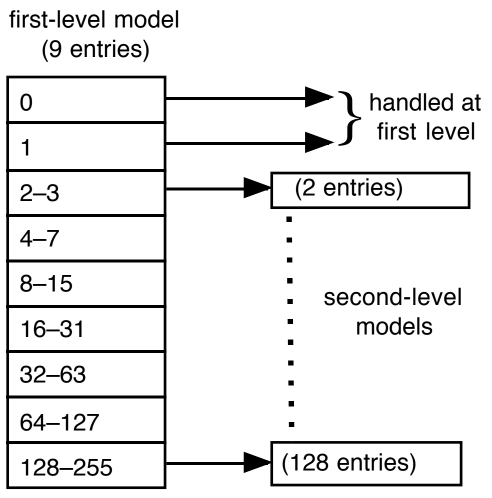

Figure 7. Structured coding model

large-scale features of the symbol frequency distribution, such as the higher frequencies of GEO (Figure 3) or the absence of symbols above 127 for most text files.

The foreground level of the two-level hierarchy divides the 256 MTF codes into 9 ranges, of about equal probabilities if Zipf's Law applied. Most entries act as escapes into "background" models to resolve the symbols in each group. When used directly on the MTF output this coding model compresses the corpus at 2.37 bit/byte, which is nearly as good as the best of the more complex coding models. When the runs of zeros are encoded with Wheeler's run-length code (described later) the performance improves to 2.34 bit/byte, the best of all of the compressors tested so far. Detailed results for the entire corpus are included in Table 2.

The encoded symbols do not obey Zipf's Law, falling off much more rapidly at higher ranks than the law would predict. Adjusting the range boundaries to give a more uniform distribution gives less than 0.1% improvement and is judged to be unnecessary.

## 10. Other work

When this work was almost completed, some more results were released by Wheeler[13], including implementations of routines very similar to the original BW94, and also a short report on some improved compressors. The implementation ("*bred*" for compression and "*bexp*" for expansion) includes some improved sorting techniques and, with a fast Huffman coding stage, runs at very impressive speeds. The sorting techniques of *bred* have been discussed earlier in section 5.

His other coders (not *bred*) use a version of the hierarchical coding models described above, with run-length encoding for runs of zeros. The first-level of the hierarchy has 4 codes; 0 and 1 code the digits of the run length, 2 is used for MTF codes of 1, and 3 is

|  | order-0 MTF | Shannon | Structured model |  | PPMC (1990) | BW94 | PPM* | PPMD+ | BW95 6/2 arith |
|---|---|---|---|---|---|---|---|---|---|
| **Bib** | 2.13 | 1.98 | 1.95 |  | 2.11 | 2.07 | 1.91 | 1.86 | 2.02 |
| **Book1** | 2.52 | 2.40 | 2.39 |  | 2.48 | 2.49 | 2.40 | 2.30 | 2.48 |
| **Book2** | 2.20 | 2.06 | 2.04 |  | 2.26 | 2.13 | 2.02 | 1.96 | 2.10 |
| **Geo** | 4.81 | 4.55 | 4.50 |  | 4.78 | 4.45 | 4.83 | 4.73 | 4.73 |
| **News** | 2.68 | 2.53 | 2.50 |  | 2.65 | 2.59 | 2.42 | 2.35 | 2.56 |
| **Obj1** | 4.20 | 3.93 | 3.87 |  | 3.76 | 3.98 | 4.00 | 3.73 | 3.88 |
| **Obj2** | 2.71 | 2.49 | 2.46 |  | 2.69 | 2.64 | 2.43 | 2.38 | 2.53 |
| **Paper1** | 2.61 | 2.48 | 2.46 |  | 2.48 | 2.55 | 2.37 | 2.33 | 2.52 |
| **Paper2** | 2.57 | 2.44 | 2.41 |  | 2.45 | 2.51 | 2.36 | 2.32 | 2.50 |
| **Pic** | 0.83 | 0.77 | 0.77 |  | 1.09 | 0.83 | 0.85 | 0.80 | 0.79 |
| **ProgC** | 2.68 | 2.52 | 2.49 |  | 2.49 | 2.58 | 2.40 | 2.36 | 2.54 |
| **ProgL** | 1.85 | 1.73 | 1.72 |  | 1.90 | 1.80 | 1.67 | 1.68 | 1.75 |
| **ProgP** | 1.84 | 1.72 | 1.70 |  | 1.84 | 1.79 | 1.62 | 1.70 | 1.74 |
| **Trans** | 1.61 | 1.50 | 1.50 |  | 1.77 | 1.57 | 1.45 | 1.47 | 1.52 |
| **AVG** | 2.52 | 2.36 | 2.34 |  | 2.48 | 2.43 | 2.34 | 2.28 | 2.40 |

Table 2. Summary of results : best compressor of each type

an escape to the full, second-level, model. This result is then encoded using Huffman or arithmetic codes and a form of trigraph context encoding, to give results which are quite competitive with other compressors. His best results are shown in Table 2 as "BW95 6/2 arith". (There is some doubt as to whether his use of context models is really useful – see section 12.)

His run-length coding is interesting, and apparently unpublished. The alphabet is expanded by one symbol, using 0 and 1 for runs and with all other values increased by 1. The run is encoded as a binary number, but with the digits 0 and 1 having weights of 1 and 2, instead of the more usual 0 and 1. A value of 0 cannot be represented, but most values require one less bit than a simple binary coding would imply. He uses a variant of this coding to handle runs in the original input to improve the sorting speed.

## 11. Final results

Table 2 presents the detailed results for three compressors developed here, together with several other representative compressors. These compressors are —

| | |
|---|---|
| **order-0 MTF** | block sorting, with order-0 arithmetic compression of the MTF output (the first version in this paper) |
| **Shannon** | The best compressor using the Shannon coding model |
| **Structured Model** | The compressor with a structured coding model |
| **PPMC** | the established standard for quality compression. (A referee has pointed out that the compression of PPMC has now been improved to 2.34 bit/byte, which is similar to the best result with block sorting and PPM*. The older "1990" values have been retained in this paper.) |
| **BW94** | the original block-sorting compressor, as published by Burrows and Wheeler |
| **PPM*** | a recently published unbounded context version of PPM [6], |
| **PPMD+** | a further-improved version of PPM [12] |
| **BW95 6/2 arith** | the best of the compressors in Wheeler's latest report [13] |

Thus the Shannon model compressor and especially the "structured model" compressor represent a considerable improvement on PPMC which has for some years been regarded as the benchmark quality compressor. Their performance is very close to that of PPM* (and the most recent version of PPMC) and within 4% of PPMD+, the best published to date.

In the present implementation the block-sorting compressors require about 9 bytes of overhead for each data byte, plus a constant 700 kByte. Most files of the Calgary Corpus can be compressed in 2 MByte of storage and all can be processed in 8 MByte. (Burrows' latest sorting techniques require only about 5 bytes per input byte, but do rely on packing information into words [3].)

The complete corpus (3.15 MByte) compresses in 460 seconds on a Macintosh Powerbook 540C (66 MHz 68040LC), or 135 seconds on a HP 755 workstation (99 MHz PA-RISC). These times and memory requirements compare well with those for other compressors of comparable performance. For example PPMD+[12] on a 50 MHz SPARC server requires about 1,200 seconds to compress the Corpus (965 seconds without PIC), with storage of 12 – 20 data bytes per input byte on most files.

Wheeler [13] quotes "*bred*" (corresponding to

BW94) as requiring 27 seconds to compress the entire Calgary Corpus on a DEC5000/133; his routine is optimised for speed with the fast sorting routines mentioned earlier and Huffman coding which is faster than the arithmetic used here. Compiled with maximum optimisation on a HP-755 workstation, *bred* compresses the entire corpus in about 12 seconds, corresponding to a speed of 250 kByte/s. (On the Powerbook 540C, *bred* compresses the corpus in 93 seconds.)

## 12. The context structure

The techniques described so far have used only the simple order-0 frequencies and runs of zeros, ignoring most of the higher-order Markov structure on which most compressors depend. It is clear that the original sorting phase uses this Markov structure in grouping symbols according to their contexts, but that the MTF output structure is quite different. A fundamental question is just what is this structure.

The poor behaviour with high-order PPM compressors has been noted already. The detailed structure is the subject of ongoing study, but some preliminary observations are appropriate.

- Changing between two coding models as a function of the distance from the last occurrence of a particular MTF value gives negligible change in compression.
- The distribution of distances between occurrences of a particular MTF code approximates a negative exponential.
- Choosing coding models according to the last few codes emitted (a constant-order Markov or PPM coding model) has negligible effect on the compression.

The tentative conclusion is that a particular MTF code value occurs as a set of independent random events, distributed across the file according to the expected probability. In other words there may be no structure at all, apart from the skew symbol distribution, preponderance of zero values and rapid local changes in frequencies, all of which have been exploited already. If this conclusion is correct, it means that there is little chance of any significant improvement in the compression of block sorting.

## 13. What block-sorting actually does

Block-sort compression is a sequence of processes, 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. The permutation has created strong locality because 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, even though it is at best an approximation to the true local contexts.
3. The final compression stage exploits the highly skewed frequency distribution from the second stage to produce efficiently-compressed output.

The first two stages are both transformations, the first a permutation and the second a recoding, which effect no compression in themselves. Between them they completely restructure the original text and that is why the “good” compressors do not work well. All efficient text compressors (whether dictionary, statistical, etc — all are equivalent) exploit the high-order historical 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. Any structure which remains is quite different from that on which normal compressors depend, as was discussed in section 12.

The rearrangement of the contexts also explains a significant weakness in block-sorting as compared with PPM compression, even though both rely on the high-order context structure of the input data. In PPM compression the multiple contexts develop in parallel as compression proceeds; we have exact knowledge of all possible contexts and can use them in the coding and decoding.

In block-sorting compression similar contexts are collected together in a region of the reordered input. As we proceed with coding we develop one context and, when it is completely processed, move on to another context, possibly similar, but possibly quite different. Thus whereas PPM develops its contexts in parallel, block sorting develops the same contexts sequentially, but we see only the emitted symbols and not the associated context. Changes in the emitted symbol are a very poor indicator of changes in context and we simply cannot infer context changes from the pattern of emitted symbols. All that the coder and decoder can do is respond to local changes in what are seen as the likely symbols. The knowledge of contexts is much less precise and the compression (which ultimately depends on details of the contexts) is that much poorer.

Data compression is traditionally considered a combination of two activities, modelling and coding, but traditional compressors always treat data in its natural form. Block sorting introduces an extra preliminary step and we must now consider the three steps of transformation, modelling and coding. In this regard it resembles image compression techniques which include an initial step such as a discrete cosine or wavelet transform. It raises the question as to whether other transformations on the input might achieve even better results. (Traditional MTF compression also fits into this model of {*transformation*, *modelling*, *coding*}.)

The discussion so far has tacitly assumed that sorting is the fundamental operation, with an output which is especially suited to MTF compression. It may be more accurate to regard the sorting step as a preprocessor which improves the operation of the MTF compression.

## 14. Conclusions

It is clear that block sorting compression (or "block reduction" to use the term in Wheeler's latest report) is a very competitive compressor. It can be very fast indeed and, with suitable final coding, can give performance comparable with the very best compressors.

In comparison with PPM it has the advantage that there is no need to code escapes to move between orders, with the possibility that a bad estimate of the escape probability will impair compression. In comparison with PPM it has the disadvantage that there is no knowledge of the actual contexts; the coder can respond to changes in the output statistics but cannot anticipate them as easily from prior knowledge. Perhaps more importantly, it does not know when to forget a previously active context. For these reasons block sorting is likely to remain inferior to the very best PPM compressors, although it is certainly competitive with most.

The statistics of the data to the final coder are quite different from those of conventional statistical compression and considerable effort has gone into developing models which can exploit those statistics. Overall though there seems to be relatively little structure in the data, and possibly little further improvement possible in compression.

## 15. Acknowledgements

This work was supported by research 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, the University of Wisconsin–Madison and the University of Western Australia. The author acknowledges the contributions of all of these institutions.

The assistance and ideas of Prof David Wheeler are especially acknowledged. He developed the original algorithm and provided many useful comments on this work. His own work, especially that of his second report, acted as a considerable impetus to what is described here. Other important contributions came from Profs Richard Brent and Clarke Thomborson, and from Alistair Moffat (who provided the new arithmetic coding routines), Bill Teahan (the PPM+ results), Mike Burrows and Charles Bloom.

## References

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

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

[3] M. Burrows, 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 `gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-124.ps.Z`

[5] 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

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

[7] P.M. Fenwick, "Experiments with a Block-Sorting Text Compression Algorithm", The University of Auckland, Department of Computer Science, Technical Report 111, March 1995. `ftp.cs.auckland.ac.nz /out/peter-f/report111.ps`

[8] P.M. Fenwick, "Improvements to the Block-Sorting Text Compression Algorithm", The University of Auckland, Department of Computer Science, Technical Report 120, July 1995. `ftp.cs.auckland.ac.nz /out/peter-f/report120.ps`

[9] A. Moffat, "Implementing the PPM Data Compression Scheme", *IEEE Trans. Comm.*, Vol 38, No 11, p1917–1921, Nov 1990

[10] A. Moffat, R. Neal, I.H. Witten, "Arithmetic Coding Revisited", *Data Compression Conference, DCC-95*, Snowbird Utah, March 1995

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

[12] W.J. Teahan, private communication

[13] D.J. Wheeler, private communication. (Oct '95) [This result was also posted to the `comp.compression.research` newsgroup. The files are available by anonymous FTP from `ftp.cl.cam.ac.uk /users/djw3`]

[14] I. Witten, R. Neal, and J. Cleary, "Arithmetic coding for data compression", *Communications of the ACM*, Vol 30 (1987), pp 520-540.