Block Sorting Text Compression — Final Report

Peter Fenwick,

Technical Report 130

ISSN 1173-3500

23 April 1996

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 combines good speed with excellent compression performance.

This report investigates the block sorting compression algorithm, in particular trying to understand its operation and limitations. Various approaches are investigated in an attempt to improve the compression with block sorting, most of which involve a hierarchy of coding models to allow fast adaptation to local contexts. The best technique involves a new "structured" coding model, especially designed for compressing data with skew symbol distributions. Block sorting compression is found to be related to work by Shannon in 1951 on the prediction of English text.

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

Preface

This is third report[1] of a series on block sorting text compression (previous members Technical Reports 111, 120, Refs 10 & 11). A shorter version was presented as a paper at ACSC'96 (ref [12]). This present report was prepared as a comprehensive account of my experience with block sorting compression, including several interesting curiosities, but grew too large for publication as a paper. Important material will be extracted to form probably two journal papers, but this text remains as the comprehensive and coherent picture of the work.

While it largely replaces the two earlier reports, those do contain some useful material which is omitted from here. For example Tech Report 111 includes extensive logs of the compression process and output which could be useful for people interested in the details of the operation.

[1] The report is available by anonymous FTP from `ftp.cs.auckland.ac.nz /out/peter-f/TechRep130.ps`

1. Introduction

Text compression of data, or lossless compression, is an area which has experienced a resurgence in the last few years. For a very long time it appeared that Huffman coding had closed the subject once and for all, but work by Ziv and Lempel in the mid-1970s and the development of arithmetic coding and PPM in the 1980s showed that compression well beyond that achieved by Huffman codes was easily achievable. A good summary of text compression as at 1990 is contained in the book by Bell, Cleary and Witten [1].

Since then there has been continuing progress in improving the known techniques such as PPMC[13, 19] and variants of PPM such as PPM*[7], PPMD+[17]. Thus whereas the state of the art in 1990 was PPMC compressing the Calgary Corpus [6] to an average of 2.48 bit/byte, several techniques now achieve 2.34 bit/byte and some better 2.29.

A recent report by Burrows and Wheeler [5] describes an apparently quite different method which permutes the entire input and compresses that permutation. The authors state that their "algorithm achieves speed comparable to algorithms based on the techniques of Lempel and Ziv, but obtains compression close to the best statistical modelling techniques". It is this new method (known variously as "block sorting", "block reduction" or the "Burrows-Wheeler Transform") which is investigated more fully in the present report. Other material on the new technique is contained in work by Burrows and Wheeler [5, 18], and Fenwick [10, 11, 12].

1.1 Statistical compressors

Text compressors are generally divided into two classes —

1. Dictionary compressors (such as LZ–77, LZ–78 and their derivatives) build explicit or implicit dictionaries of strings and replace entire strings or groups of symbols.

2. Statistical compressors develop models of the statistics of the input text and use those statistical models to control the final compression.

Statistical text compressors are traditionally regarded as a combination of a *modelling stage* and a following *coding stage*. The model is constructed from the already-known input and used to facilitate efficient compression within the coder (and matching decompression in the decoder). A good model will contain a few symbols with high probability (and preferably one dominant symbol), thus allowing very compact coding of those probable symbols.

There are three basic modelling techniques, with most good compressors combining two or even three of the techniques. The symbols of the input data are usually considered individually, with differing models for successive symbols, although some methods use words as the coding elements.

Symbol frequency The possible symbols have expected frequencies associated with them, allowing the coder to use shorter codes for the more frequent symbols and longer codes for the less frequent ones. In the simplest cases the model just accumulates the frequencies of all of the symbols which it has seen, or even works with preassigned frequencies. Frequency modelling underlies all statistical compressors; the more complex methods use correspondingly more complex modelling and prediction techniques to increase the probability of likely symbols and allow them to be coded more efficiently.

Symbol context The strong dependency between adjacent symbols of normal text is usually expressed as a Markov model, with the probability of the occurrence of a particular symbol being expressed as a function of the preceding n symbols (an "order n" Markov model, or a context of n symbols). The most frequent context modelling technique is "Prediction by Partial Matching", or PPM[8, 13, 19]. For each symbol to be encoded, a typical PPM compressor will consider say the preceding 4 symbols (an "order 4" context) and determine the probabilities of the symbols which it has already seen in that context. If the desired symbol has not been seen in the order 4 context, the coder will "escape" down to lower orders until it determines a context from which the symbol can be emitted.

An escape to a lower order appears as a symbol to the higher order, but one whose probability is not easily determined. Various heuristics are available to determine the escape probability and these largely differentiate the different PPM compressors and determine their relative qualities. However the handling of escapes or notification of the correct order remains one of the major problems in PPM compression. The data structures to represent the known contexts may be quite complex and subtle and some may require large amounts of data storage and computing time.

Some of the more recent PPM techniques (PPM*[7] and PPMD+[17]) allow contexts of unbounded order. In all cases though the modelling stage prepares, for each symbol, a model which is unique to the current context and individually tuned to its characteristics.

Symbol ranking This is a relatively little-known technique derived from work by Shannon in 1951[16]. A symbol "predictor" determines a probable next symbol, which may be accepted or rejected by a comparator which can see the incoming text. Shannon presents two alternative methods.

1. The response to an offered symbol is either "CORRECT", or "THE CORRECT VALUE IS …".

2. The predictor must keep suggesting symbols until the correct one is found. The effect is that the predictor ranks the symbols in order of decreasing likelihood and the sequence of "NO"s and the final "YES" constitutes a unary coded value of that ranking.

Shannon's second method is essentially a recoding of the input symbols, with the recoding usually dynamic according to the symbol context, although he does not specify how it might be done. There is an output symbol for each input symbol, and compression relies on the very skew output distribution, with most output symbols being 0 and able to be emitted with very short codes.

The main example of symbol ranking is found in "Move-to-Front" compression as presented by Bentley et al[2], but with words as the fundamental coding unit — here we use 8-bit bytes as the unit of MTF coding. MTF coding assumes that symbols can be ranked according to their "recency", or the closeness of their last occurrence. The compressor maintains a list of symbols; when a symbol is encountered its position or rank in the list is emitted as the value to the coder and the symbol is then moved to the front of the list. The effect is that more frequently-used symbols stay closer to the front of the list, while less frequently-used symbols drift to the back of the list. Smaller list ranks tend to be more frequent and can be emitted with shorter codes, while the less frequent larger ranks require longer codes.

A third version of the Shannon coding model is essentially a hybrid of the two types above and will be used later in this paper. It allows several unsuccessful attempts, but after about half a dozen failures supplies the correct symbol. (Shannon's Type 1 allows only one bad prediction.)

Symbol ranking is not really a modelling technique at all, but is rather an example of a *transformation* which converts the raw input into a more-compressible form. Where there is no reordering of the symbols the transformation becomes a *recoding* (as with MTF compression). In general the transformation (or recoding) produces an output symbol for each input symbol, with the output alphabet being dominated by a few symbols. It is the responsibility of the following modeller and coder to handle the transformed data as efficiently as possible. Thus the traditional picture of $\{modelling \rightarrow coding\}$ has been replaced by $\{transformation \rightarrow modelling \rightarrow coding\}$. We will see later that the "block sorting" or "block reduction" algorithm of Burrows and Wheeler is reasonably interpreted as a symbol ranking compressor with an initial permutation of the input text.

2. Block sorting transformations

Block sorting is based upon two quite separate transformations, a forward transformation which permutes the input data into a form which is easily compressed, and a matching reverse transformation which recovers the original input from the permuted data. The data is always considered in blocks which may be as large as the entire file or may be a few tens or hundreds of kilobytes.

2.1 The block sorting forward transformation

There are two approaches to describing the forward transformation. Burrows and Wheeler describe it as the steps –

1. Write the input as the first row of a matrix, one symbol per column

2. Form all cyclic permutations of that row and write as them as the other rows of the matrix

3. Sort the matrix rows according to the lexicographical order of the elements of the rows

4. Take as output the final column of the sorted matrix, together with the number of the row which corresponds to the original input

In terms which are more familiar to workers in data compression, the transformation may be described as –

1. Sort the input symbols, using as a key for each symbol the symbols which immediately follow it, to whatever length is needed to resolve the comparison. The symbols are therefore sorted according to their *following* contexts, whereas conventional data compression uses the *preceding* contexts. (Some implementations do use preceding contexts, but the discussion is easier with following contexts.)

2. Take as output the sorted symbols, together with the position in that output of the last symbol of the input data.

In either case the effect of the sorting is to collect together similar contexts. The (assumed) Markov structure of the input implies that only a few symbols are likely to occur in association with adjacent contexts. Any region of the permuted file will probably contain only a very few symbols and this locality can be captured with a Move-to-Front compressor.

2.2 The block sorting reverse transformation

Perhaps the most surprising thing about the forward transformation described above is that it is actually reversible! The reverse transformation depends on two observations –

1. The transformed input data is a permutation of the original input symbols

2. Sorting the permuted data gives the first symbol 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 $\boldsymbol{s}$ in position i of the transmitted text, we find that position i within the ordered contexts contains the j-th occurrence of symbol $\boldsymbol{t}$; this is the next emitted symbol. We then go to the j-th occurrence of $\boldsymbol{t}$ 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 and from there we can traverse the entire transmitted data to recover the original text.

2.3 Illustration of the 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 contexts ("`iiiimppssss`") and then build the links shown in the last column. The four "`i`" contexts link to the four "`i`" input symbols in order (to 5, 7, 10 & 11 respectively). The "`m`" context links to the only "`m`" symbol, and the two "`p`"s and four "`s`"s link to their partners in order.

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 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`" and its immediately following context symbol, the fourth "`i`". We then link to 11 get the "`i`", and so on for the rest of the data, stopping on a symbol count or the return to the start of the file.

2.4 Algorithms for the reverse transformation

The forward transformation is largely concerned with sorting and will be described later. Here we describe the steps of the reverse transformation, assuming the notation –

`N` = symbols in file	
`n` = symbols in alphabet	
`S` = array of received symbols	`S[1..N]`
`T` = array of recovered symbols	`T[1..N]`
`K` = counts of each symbol	`K[0..n-1]`
`L` = links to resolve permutation	`L[1..N]`
`M` = mapping array for symbols	`M[0..n-1]`

The received (permuted) symbols are assumed in `S`, with counters etc appropriately initialised. The first step is to just count the occurrences of each symbol; it may be performed as the file is read in and decompressed.

```
for (i = 1; i <= N; i++)
   K[S[i]] ++;                     /* count input symbols */
```

The next step involves building the table of the initial positions of each context. Remember that these are ordered according to the lexicographical ordering of the symbols and that there are as many contexts starting with "`a`" as there are occurrences of "`a`" in the input.

```
M[0] = 1;
for (j = 1; j < n; j++)
   M[j] = M[j-1]+K[j-1];
```

We are now in a position to build the links which will be used to traverse the received data. The mapping array `M` initially points to the first positions of the contexts. As a symbol `s` is used from a context, `M[s]` is incremented so that it tracks the next context starting with "`s`".

```
for (i = 1; i <= N; i++)
   {
   s = S[i];                    /* get current symbol */
```

```
    L[i] = M[s];                    /* set link from mapping table */
    M[s] ++;                        /* increment mapping */
    }
```

Although we never actually prepare an explicit list of the contexts (or rather their initial symbols), the steps so far are equivalent to using such a sorted list. Finally we can recover the original text.

```
ix = initial_position;          /* start at the correct position */
for (i = 1; i <= N; i++)
   {
   ix = L[ix];                  /* step on via the links array */
   T[i] = S[ix];                /* copy symbol to output */
   if (ix == initial_position)
      break;                    /* an alternative termination */
   }
```

There are three passes through the entire file (one of which may be combined with reading) and one pass through the alphabet. The operations are all quite simple and usually add little to the input/output costs of handling the file or, in most cases, to the cost of decompressing the file.

3. Initial implementation

None of the present work was aimed at providing an especially fast compressor, or one that required little memory; some approaches to fast or memory efficient versions are described by Burrows and Wheeler [5, 18]. The emphasis here was on an implementation that was fast enough to be useful and could be used for investigations on the fundamental properties of block sorting compression.

1. The sorting phase uses a 65,536-way radix sort, using the Unix *qsort* routine and with the buckets selected by the first two symbols of each context[2]. More details of the sort phase are given later, including modifications to improve the speed on some files.

2. The Move-to-Front phase uses a simple pair of arrays, one holding the current symbol image and one mapping symbols to their indices into that image. Most files require quite small symbol movement and the complex data structures and housekeeping overheads of an "efficient" implementation are not justified.

3. The final compression stage uses a simple order-0 arithmetic coder, initially based on the well-known "CACM" routines [20].

The first two stages are essentially standard and are used in all of the coders which have been tried for this report. It is only the final compression stage which was the subject of detailed experimentation.

File decompression uses the obvious steps of arithmetic decoder, MTF decoder and the reverse permutation described earlier. This document will say little about the decompression performance. In general decompression is very fast, with its performance set by the arithmetic (or Huffman) decoder and possibly the input/output system.

Wheeler has released a more recent report [18] including an implementation ("bred") which is comparable to his original "BW94" or the current "bsOrder0", but with Huffman coding for the final compression stage. That report will be referred to at various parts of this present report.

4. Initial results

The initial results are shown in Table 1, testing on the Calgary corpus[6] and showing PPMC as a reference compressor.

[2] 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!

File	size bytes	PPMC (1990)	bs Order0	MTF dist non-0	frac 0 MTF	compares	Avg. compare length (words)
BIB	111,261	2.110	2.133	5.50	66.8%	1,704,645	1.65
BOOK1	768,771	2.480	2.523	3.88	49.8%	15,974,996	1.25
BOOK2	610,856	2.260	2.198	4.20	60.8%	12,139,721	1.42
GEO	102,400	4.780	4.812	55.63	35.8%	1,435,323	1.07
NEWS	377,109	2.650	2.677	7.65	57.9%	6,789,971	1.66
OBJ1	21,504	3.760	4.227	46.82	50.6%	231,884	1.51
OBJ2	246,814	2.690	2.710	30.22	68.1%	3,900,488	1.83
PAPER1	53,161	2.480	2.606	6.45	58.4%	734,781	1.31
PAPER2	82,199	2.450	2.571	5.06	55.4%	1,274,109	1.24
PIC	513,216	1.090	0.919	3.39	87.4%	47,889,672	1.36
PROGC	39,611	2.490	2.666	8.32	60.3%	511,849	1.37
PROGL	71,646	1.900	1.839	4.63	72.9%	1,055,533	2.06
PROGP	49,379	1.840	1.821	5.54	74.0%	673,846	3.25
TRANS	93,695	1.770	1.601	5.66	79.2%	1,329,847	3.51
AVG		2.482	2.522	13.78	62.7%		

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

The columns of the table are, in order –

1. **The file size in bytes.** It may be noted here that some tests used only the 9 or 10 small files (about 100 kbyte or less, and possibly including PIC after the sort optimisation). Most of the general testing, tuning and algorithm validation was done on the file PAPER1.

2. **The compression of each file using PPMC.** The compression performance is always given here as output bits per input byte. PPMC was until recently the state of the art and this column is intended as a reference against which other compressions may be measured. (The performance of PPMC has been improved recently [19], but the better known 1990 results are retained here.)

3. **The compression of each file with the new block sorting compressor.** For most files the performance is quite similar to that with PPMC, with the more compressible files showing slightly better results.

4. **The average Move-to-Front distance**, for those symbols which move. (Symbols already at the head of the MTF list are excluded.) Except for the three binary files, the average movement is only about 5 or 6 positions, justifying the use of a simple MTF algorithm.

5. **The fraction of symbols encoded as zero by the MTF step.** In general the more compressible the file, the more of its output is represented by zeros in the MTF output. Most text files have 50–60% of the codes as zero, while the less-compressible GEO has only 36% and the highly compressible PIC 87%.
[As a general rule the values from the MTF step will be referred as *ranks*. *Rank*_0 will refer to a value of zero from the MTF step, *rank*_1 to a value of 1, and so on.]

6. **Number of compares.** This is the number of string comparisons needed to reorder the input file. It is generally about 10 – 20 per input byte, but rises to 80 for PIC, for reasons which will be given later.

7. **Average comparison length.** This is the average length of each string comparison. This column is taken from later work as described in Section 6.1 and gives the number of *word* comparisons (4 bytes per word). It shows that for most files a comparison needs to extend to only about 6 symbols, plus the 2 used in forming the sort bucket. (The earliest implementations could not reveal the comparison length.)

The obvious conclusion is that block sorting is already a good compressor, being within about 1.6% of the overall compression of PPMC. (Following the usual practice, we take an unweighted mean of the compressions for the individual files of the corpus.)

Another important result should be quoted from Burrows and Wheeler [5]. The sorted blocks can be smaller than a complete file, although the work here always uses the entire file. Their results for the compression of two files as a function of the size of the sorted blocks are presented here as Table 2. One file is BOOK1 form the Calgary Corpus. The other is the "Hector Corpus" a collection of over 100 MByte of English text.

file	1k	4k	16k	64k	256k	750k	1M	4M	16M	64M	103M
book1	4.34	3.86	3.43	3.00	2.68	2.49					
Hector	4.35	3.83	3.39	2.98	2.65		2.43	2.26	2.13	2.04	2.01

Table 2. Compression (bit/byte) v blocksize for "BOOK1" and the Hector Corpus

The simple result is that increasing the blocksize allows more contexts to be considered and improves the compression. It is interesting to see the close agreement between the two results, up to the size of BOOK1.

5. Symbol statistics

An early motivation for this work was the realisation that as block sorting achieves good results with MTF and Huffman coding, a compressor of only moderate performance, a "better" final compressor might give correspondingly better compression. Early tests with "good" compressors were singularly unsuccessful and are reported later. These tests did however indicate that block sorting is fundamentally different from other compression algorithms and that a completely new understanding might be required. The output of the MTF stage appears to contain little of the context or historical structure on which most compressors depend.

Remember too, that initial sorting and MTF stages are not compressors, but recode symbols into a peculiarly skewed alphabet, which can be compressed using an appropriate compressor. In the apparent absence of context structure or history effects, we must assume that all of the compression arises from the frequencies of the MTF output symbols. As a first stage to understanding the operation of block sorting, we therefore consider the statistics of the MTF ranks across entire files.

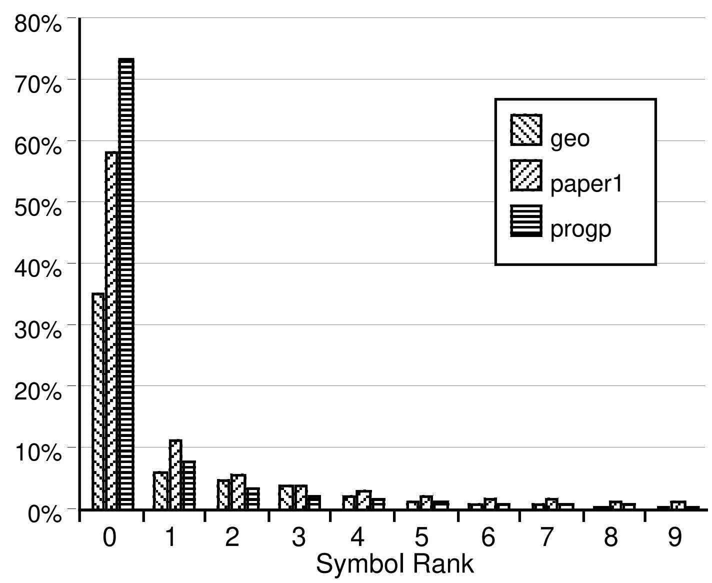

Figure 2s. Probabilities of MTF symbols

The distribution of ranks from the MTF stage is shown in Figures 2 and 3 for three representative files. GEO is relatively incompressible, PAPER1 is a typical text file, while PROGP is more compressible. The major differences in Figure 2 are in the probabilities of rank_0, and that PROGP (more compressible) has a greater skewness.

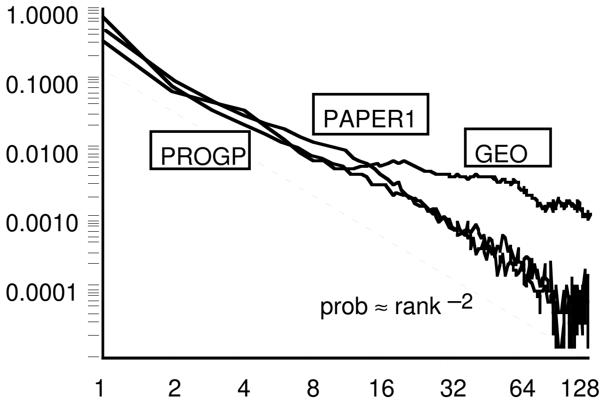

Figure 3. Probabilities of the first 128 MTF ranks for three files

Some more useful information comes in Figure 3, drawn with log-log scales. The two files PAPER1 and PROGP (and GEO for small values) display an almost linear behaviour, with probabilities proportional to $rank^{-2}$. This may be compared with Zipf's law for natural language, where the symbol probability varies as $rank^{-1}$. The higher exponent corresponds to a more highly skewed distribution, a lower entropy or higher compressibility and follows from the action of the block sorting and MTF operations. (The relation between skewness and compression is explored later.)

Although GEO approximates the $rank^{-2}$ dependency for small ranks, its quite different behaviour for large ranks implies that it is much less compressible, as is indeed the case.

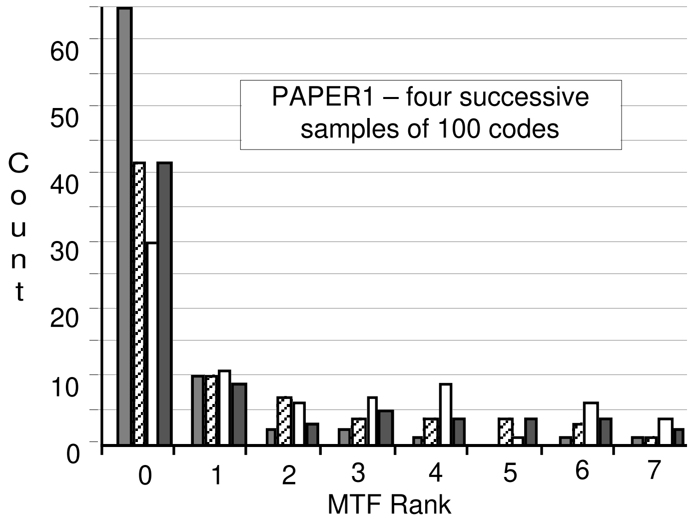

Figure 4. MTF ranks in successive samples

A very important aspect is the local variability of the code statistics, shown in Figure 4, which is simply the count of each rank in four successive 100-code samples of the file PAPER1. For example, the rank_0 probability varies from 30% to 65%, and rank_4 from 1% to 9%, even over this small region. Thus not only must the final modelling and coding stages handle very wide ranges of values, they must also have fast adaptation to handle the rapid local fluctuations.

6. Various improvements

While the initial implementation certainly worked, it was slow and gave possibly poor compression performance. The following sections detail work to overcome these known deficiencies. The three main areas covered are the initial sorting routines, the final coding routines and then modelling for the MTF ranks.

6.1 Improving the sort

The current work has concentrated on understanding the principles of block sorting compression and achieving the best possible compression. The sorting or permutation step was regarded as an essentially fixed component; any method that was fast enough for viable compression was acceptable.

The basic "sort engine" is the standard C library *qsort*. While simple to get going, every comparison requires calling a user-supplied comparison procedure and this involves a considerable overhead which does slow the operation. Sorting uses a 65,536-way radix sort, based on the first two symbols of the comparand string. To accelerate comparisons, the input text is placed in an array of 32-bit words with 4 bytes to a word and bytes striped across successive words. (A byte initially appears in the left-most position of "its" word, the second position of the next word and so on for the next two words.) A word comparison therefore compares 4 bytes and a stride of 4 words steps to the next 4 bytes to be compared. This is found to give adequate performance on most files, despite the performance penalties of the generic sort.

Sorting can be very slow though on files with long runs of identical symbols. Not only are there many comparisons between quite similar strings, but the comparisons themselves may be long because the shorter comparand must reach the end of its run before the comparison is resolved. Within the Calgary corpus, the file PIC is especially bad in this regard with many runs of thousands of zeros and in fact 80% of the file being zero. In their original report Burrows and Wheeler outline several approaches to improving the sorting of runs, essentially based on relationships between the run symbol, the terminating symbols, and the distance to those terminators. A much more direct solution though is to eliminate the runs.

In the author's implementation, a sequence of 6 identical symbols signals a run and is always followed by a length code (which may be zero). The encoded run length is presented to the sort in place of the original data run. The run length is encoded into the next few symbols in groups of 6 bits, with each symbol having a 7th bit set if more length digits follow. Although run-length encoding is normally regarded as a compression technique that is not its intention here. It is meant only to accelerate sorting. Most files have few or no runs; any compressible runs are replaced by generally incompressible length counts and the compression may deteriorate very slightly. PIC is an exception. About 75% of its symbols disappear into the encoded runs and the overall compression improves by about 10% with run encoding. More importantly though, the sorting time typically improves from 9 minutes to 12 seconds on one computer.

A similar problem arises in files with a "picket fence" structure such as "…`aaaabaaaabaaaabaaaab`…". These files can be handled by a technique similar to an LZ-77 string match, preferably allowing a recursive match ahead into the "picket fence" string. While this method does improve the speed on these admittedly pathological cases, it hinders the compression of ordinary files and has not been used in general.

In his latest report, Wheeler [18] describes a sorting technique which achieves very impressive speeds. Firstly, he uses a quicksort which "knows" its data structures and need not call a user comparison procedure, with all of the associated procedure-calling overheads. His sort starts off as a radix-256 sort, based on the initial symbols of the contexts and with symbols striped across successive long words, as described above. He then sorts the buckets into increasing size, and sorts the smaller ones first. After sorting each bucket, each of the long-words referred to by that bucket has its rightmost 24 bits replaced by its ordinal position in the sorted bucket, leaving the symbol itself in the leftmost 8 bits. Thus each symbol in the bucket is uniquely tagged according to its lexicographical ordering and whenever two already processed symbols are compared the comparison is resolved immediately.

6.2 Arithmetic coding routines

This work was started with the traditional "CACM" arithmetic coding routines [1, 20]. More recently, improved versions of arithmetic coding routines have been described[14], 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 CACM routines the increment is small and remains constant (usually 1); older counts are 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 we develop separate models for each context and the statistics of each context are reasonably assumed constant over the whole file. In block sorting some of the compression comes from adaptation to local variations in the statistics and equal treatment of all symbols is quite inappropriate. Changing the DCC95 routines to allow increments to be handled as for the older CACM routines restored the expected performance.

A CACM-type of coding model has as parameters the count limit and the count increment. The *limit* governs the range of frequencies or probabilities which the model can handle. A limit of 8,191 or 16,383 allows a frequency range of about 4 orders of magnitude. The *increment*, or rather the ratio of ${}^{limit}/_{increment}$ controls the rate of adaptation because after ${}^{limit}/_{increment}$ events the counts must be rescaled. For most situations, even with multiple models, good values were found to be — *increment*=16 and *limit*= 8192.

One aspect which became very clear was the overall resilience of arithmetic coding. If one aspect of the compression is seen to be relatively expensive and the model is tuned to allow for that, the general experience is that the rest of the model usually adjusts itself to compensate and there is seldom much overall benefit! Significant improvements in compression usually require major changes to the structure or operation of the coding model.

6.3 Adjusting the Move-to-Front operation

We must consider possible improvements to the Move-to-Front operation, and perhaps even in the sorting permutation itself. Move-to-Front is a quite drastic rearrangement of the alphabet of recoded symbols, and often brings up a symbol which is needed only once and is immediately replaced by the previous MTF “head” symbol.. Work with self-organising lists, of which Move-to-Front is one example, had shown that other, weaker, movement algorithms are sometimes helpful [9].

File	move to 0	move to 1/32	move to 1/8
BIB	2.133	2.135	2.165
GEO	4.812	4.808	4.807
OBJ1	4.229	4.221	4.260
PAPER1	2.606	2.613	2.646
PAPER2	2.571	2.571	2.576
PROGC	2.667	2.674	2.716
PROGL	1.839	1.842	1.887
PROGP	1.822	1.836	1.906
TRANS	1.602	1.626	1.734
Average	2.698	2.703	2.744

Table 3. Performance with varying MTF distances

At one time it was thought that varying the sort and altering the sort key might improve the compression, but that is not the case. A very careful examination of the reconstruction algorithm shows that it assumes that all contexts and their sub-contexts are in sorted order. Anything other than a standard sort upsets the detailed ordering and prevents recovery of the data. An interpretation given later in which the Move-to-Front list becomes the current symbol ranking list implies that the combination of sorting and simple MTF is necessary and cannot be altered.

6.4 Improving compression – caches and run-length coding

All of the better block-sorting compressors use a hierarchy of coding models, although in several variants. The first or "foreground" level may be termed a "cache" and handles the first few rank values directly, with a special code signalling an escape to the "background" level for the remaining, individually less-frequent, ranks. Such a hierarchy solves two problems with encoding the skew distribution from the MTF output.

1. The coder must handle an enormous range of symbol probabilities — perhaps 4 or 5 orders of magnitude. Simple arithmetic coders just cannot handle this range efficiently and certainly cannot handle it while remaining sensitive to short-term local fluctuations. With the "CACM" routines, the possible range of symbol probabilities is governed by the maximum permitted count, or 16,383 for 32 bit operands. The range of symbol probabilities or frequencies is then 16383 : 1, or approximately 4 orders of magnitude.

   With a hierarchical model, all of the background probabilities are scaled by the escape probability. The low probabilities are shared between the escape probability and the background model and neither model has to cope with the entire range of possible probabilities.

2. If the most frequent symbol has a probability of 0.5, its count will vary between 4,000 and 8,000 (with the CACM routines) as the count is scaled and several thousand occurrences will be needed to effect much change in the overall statistics. Coding will become more sensitive to change if the counting increment is much greater than 1, but experience is that this should not be taken too far. It will in any case lead to frequent and costly rescaling of the coding model. The foreground cache, handling a small symbol alphabet, can be given a relatively low count limit and large increment so that it responds quickly to local changes in symbol statistics.

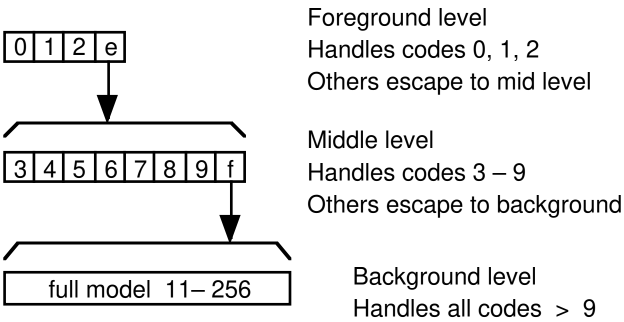

Figure 5. Three-level hierarchical coding model

A typical three-level hierarchy is shown in Figure 5. Values of 0, 1 or 2 (the most probable values with a skew distribution) are handled entirely within the foreground level. For values of 3 or greater an initial escape "*e*" is emitted using the foreground model. Values of 3 to 9 are then handled by the middle model; a value of 8 will be encoded, in total, as "*e*8". Values greater than 9 will be passed on to the background model after emitting the "escape to background" code "*f*". A large *n* will be encoded as the three symbols "*efn*".

The two best cache models each had a hierarchy of three levels. One used caches of 4 and then 16 elements (ie the first 4 and the next 16 MTF ranks) ahead of the full background model. The other used a unary run-encoding (0 for runs, 1 for end-run) ahead of an 8 entry cache and then the full model. Both achieved 2.42 bit/byte, or little better than the original block-sorting compressors.

Wheeler [18] also uses a hierarchy, but with four entries at the first level. Codes 0 and 1 encode runs, code 2 is Rank_1, and 3 is the escape to the full, background, model. He achieves 2.43 bit/byte with a final Huffman coder, or 2.40 bit/byte with an arithmetic final coder. In retrospect, it is clear that his improved run-length encoding is of considerable benefit; it is discussed later in section 11.3.

7. Shannon coder

In one of the first papers relating to the compressibility of text, Shannon in 1951 [16] used a coding model in which a test subject guessed at the next symbols, given a block of preceding text. While his 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. He gives two techniques, both of which have been discussed above. Here we are more interested in his second technique, in which the predictor is required to continue until the correct symbol is found.

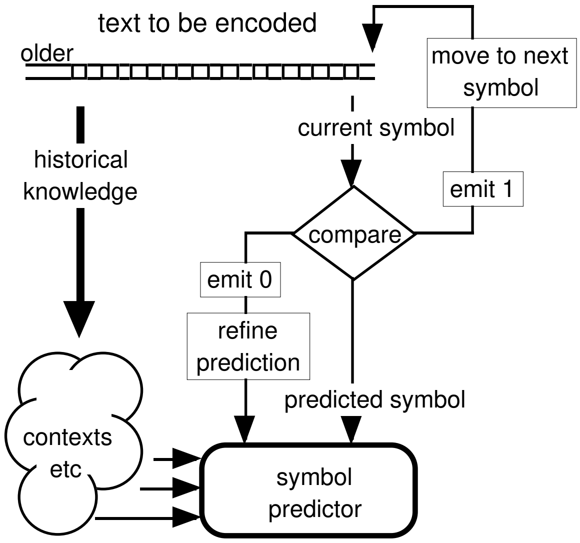

Figure 6. Shannon's symbol encoder

The coder, shown in Figure 6, contains a "predictor" which somehow estimates the next symbol and is then told whether to revise its estimate; the revision instructions (a sequence of "NO" and "YES" responses to the offered symbols) 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. The prediction is an ordered list of symbols, from most probable to least probable and the emitted code is the rank of the symbol in this list.

Shannon does not of course state just how predictions are made, other than "…Familiarity with the words, idioms, clichés and grammar …", but it is obvious that he is implying the contextual knowledge which would be available to a human predictor. In conventional computer text compression only words and phrases are available, and even that knowledge has to be accumulated afresh for each file. The ignorance of idioms etc is one good reason why the best text compressors achieve only about 2 bits/letter on English text, whereas the measured limit is closer to 1.3 bit/letter [1].

We can however postulate a text compressor based on Shannon's technique. It would perform a contextual analysis of the preceding text and predict the most likely next symbol, the second most likely, and so on. The sequence of output codes 1 (most likely), …, n(n-th most likely) constitutes a *recoding* of the input. If the predictor is good, most of its initial predictions will be correct and any failure will be corrected very quickly.[3]

It is interesting to compare Shannon's 1951 human results with those of the current block-sorting compressor. Shannon presents results for both forward and reverse prediction and they are compared here with those for the file BOOK1 as a similar example of English prose.

symbol rank	1	2	3	4	5	6
Shannon — forward	69%	10%	7%	2%	2%	3%
— reverse	65%	7%	4%	4%	6%	2%
Block Sort — Book1	50%	15%	8%	5%	4%	3%

Table 4. Comparison of Shannon's and block-sort prediction accuracy

Table 4 presents the three sets of results, and the general similarity is quite striking. Shannon's predictions are generally better, simply because of the much larger general context available to his predictor. His first-symbol

3 Experiments are being conducted with just such a compressor.

prediction accuracy of 69% is generally similar to that of PROGL or PROGP of the Calgary Corpus with block-sorting and those files are compressed to 1.7 bit/byte, whereas BOOK1 is generally compressed to about 2.3 bit/letter with the best compressors.

7.1 Applying the Shannon coder to block-sorting compression

This pattern of mostly-accurate predictions is just what we see in the output of the block-sort/MTF transformation; it is clearly a realisation of the Shannon recoder, albeit with an initial permutation. What we can now do is to code the output as though it came from the Shannon prediction mechanism. In other words, the MTF position is encoded as a sequence of bits, say 0 ⇒ CORRECT, 1 ⇒ WRONG, or as a unary code.

Several different versions of the Shannon coding model were tested

1. A single binary model, encoding 0s and 1s as described above, compressed the Corpus to 2.67 bit/byte. This is not too much worse than the initial "order-0" implementation, and quite surprising considering the simplicity of the model.
2. The unary coded output has a considerable amount of simple context dependency. If a 0 has been emitted it is quite probable that another 0 will follow. If a 1 has been emitted, a following 1 is quite likely. Using two models, selected according to the last-emitted bit, improved compression to 2.58 bit/byte.
3. Although they are relatively rare, the codes for large ranks are quite long and therefore expensive to emit. Stopping the unary code at say 6 and then emitting the correct value improves the compression to 2.43 bit/byte. In human terms this corresponds to helping the estimator who is probably frustrated by this time; it is difficult to give any sort of accurate prediction after the first few failures. In terms of coding theory, we show later that it is indeed appropriate to switch coding techniques in this way and at about this position.
4. The probability of emitting a 0 (or a 1) changes after successive failures, or according to the number of immediately preceding failures. Allowing separate models as a function of the present unary code position (with an escape to a general background model) improves compression to 2.36 bit/byte. Complete results for this compressor are given later.

Thus the Shannon coding technique is useful not only for helping to understand the fundamental operation of the block-sorting compressor, but can also be used to develop an efficient method of encoding its output.

8. Structured coding model

For most forms of compression, the modelling stage must be able to handle a few probable symbols, arbitrarily placed amidst a background of less-probable symbols. The distribution resembles a line spectrum with a noisy background. With block-sorting (and with symbol- ranking compressors in general), the symbol-frequency distribution is much more regular with the first element being the most-probable and an approximately monotonic decrease for later elements. Although there are significant differences in the details of the distribution (which leads to different compressibilities for different files), the general form is quite predictable. We can exploit that predictability to design a compression model especially for highly skewed symbol distributions.

Compression is most efficient if we can equalise the probabilities of the symbols to be encoded. If the symbol distribution obeyed Zipf's law, we could equalise the coding probabilities by grouping symbols so that the n-th group included the next 2^n symbols. The resultant coding model is illustrated in Figure 7.

As with all of the better coding models investigated here it has a hierarchical structure, but one that is rather more complex than has been used before. Most entries at the first level handle groups of symbols, each covering about one octave of values; two entries handle just one value each. Each entry which handles more than one symbol has its own second-level model to resolve those symbols.

We have previously considered the use of a "cache" to assist compression. This new model can be considered an extended cache — the 0 and 1 entries function precisely as in the earlier caches, while the other entries allow adaptation to the larger scale variations of the less-probable symbols.

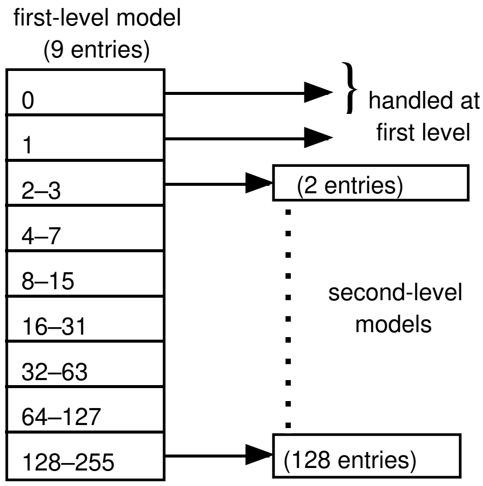

Figure 7. Structured coding model

When this coding model is used directly on the MTF output, the compression for the whole Corpus is 2.37 bit/byte, which is nearly as good as the best of the more complex coding models. When runs of zeros are encoded with Wheeler's run-length coding, the compression improves to 2.34 bit/byte which is the best result achieved with block sorting. Results for the complete corpus are given later.

The design assumes a Zipf's Law distribution, which is not the actual distribution of the symbols being encoded. Adjusting the range boundaries to give a more uniform distribution of frequencies across the first-level model gives less than 0.1% improvement and is judged to be unnecessary.

9. Final results

Table 5 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 report, see Section 4 above)

Shannon The best compressor based on the Shannon coding model (section 7.1 above)

Structured Model The compressor with a structured coding model (section 8 above)

PPMC the established standard for quality compression. (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 report.)

BW94 the original block-sorting compressor, as published by Burrows and Wheeler [5]

PPM* a recently published unbounded context version of PPM [7],

PPMD+ a further-improved version of PPM [17]

BW95 6/2 arith the best of the compressors in Wheeler's latest report [18]. This version uses his improved run-length coding, described in Section 11.3 later, and also some historical context from recent codings.

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 compressor published to date.

	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 5. Summary of results : best compressor of each type

10. Resource requirements

In the present implementation the block-sorting compressors require about 9 bytes of memory 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 [4].)

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), using the "structured model" compressor. These times and memory requirements compare well with those for other compressors of comparable performance. For example PPMD+[17] 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 [18] 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.)

11. Various topics

We now present a variety of topics which arose during the work on block sorting. While they include some important results, they do not fit into the main development of this report and are given here in a separate section.

11.1 Compression of skewed probability distributions.

We can consider coding the value as either a unary code or as an "entropy code", where the codeword length is $\log_2(1/P)$, where P is the probability of a symbol and using arithmetic coding as an entropy code. Figure 8 shows the two codelengths against rank for a typical symbol distribution, using values as discussed later in

Section 11.4. A simple unary code is more efficient than the entropy code for ranks of 6 or less. With different files the curves move vertically with relation to one another, but the values here are typical for text files. In fact the threshold between codings is not at all critical because of the ability of the arithmetic coder to tune itself to its environment.

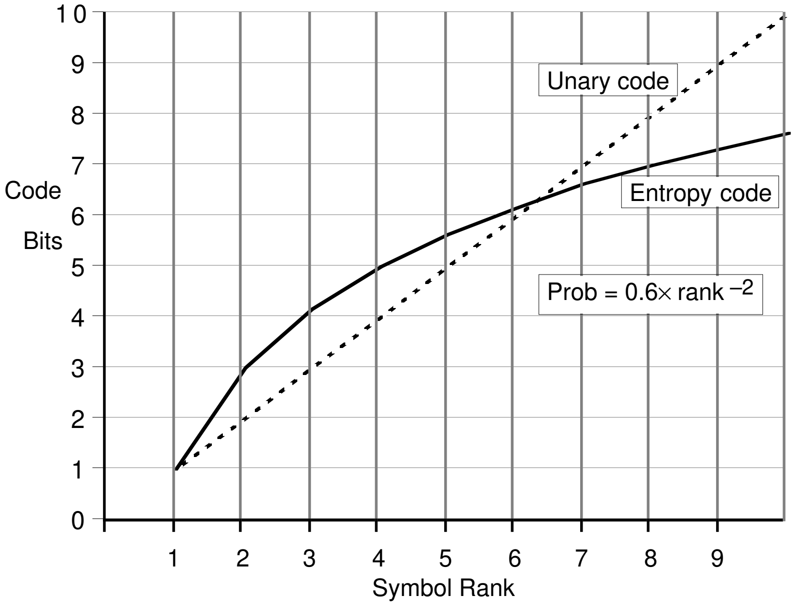

Figure 8. Unary and entropy coding of a source

An alternative approach to the coding of a skewed distribution is most easily seen by considering Huffman coding. If, when adding the *j*-th symbol of an *N* symbol alphabet with individual symbol probabilities $P(S_i)$, we have

$$P(S_j) > \sum_{j+1}^{N} P(S_i)$$

then the whole of the preceding sub-tree (represented by the right-hand side) becomes one branch of the new tree and the new symbol becomes the entire other branch. More specifically, if

$$P(S_j) > 2P(S_{j+1}) \qquad \forall \ j$$

or the probabilities form a geometric series of common ratio less than (–2), the Huffman tree is degenerate, as shown in Figure 9.

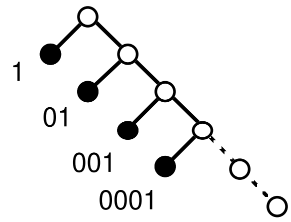

Figure 9. Degenerate Huffman tree

But we have already seen that the *n*-th symbol here has a probability proportional to n^{-b}, where b is about 2. Combining these two results, the Huffman coding tree is found to be degenerate for the first few most probable symbols, with those symbols represented as simple unary codes.

This result can be applied back to Shannon's coding transformation to show that a good coding is to allow a few wrong estimates (to give the initial unary code) but then to give the correct value after perhaps 5 or 6 failures. This is in effect a compromise between his first method (give the correct value immediately) and his second method (the predictor must deliver the correct value).

11.2 The context structure of the MTF output

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.

File	bs Order0	COMP-4 order 2	COMP-4 order 3	COMP-4 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 6. Compression at high arithmetic orders

The Block-Sort compressor was altered to write out the MTF ranks for use as inputs to other compressors. Results for the smaller files of the Calgary Corpus are given in Table 6 using Nelson's "COMP-4" compressor[15] which has the advantage of being publicly available and of being able to run at different maximum orders.

The "better" the compressor the worse the results! Consistent results were found for a variety of dictionary compressors as well. It is clear that the statistical and context structure on which these rely has been completely destroyed by the sorting and MTF transformations. There is *some* contextual structure in the runs of zeros (and the runs of ones in the Shannon coders) which has been exploited in run-length coding and the hierarchy of "1-coding" models in the Shannon coders, but that seemed to be all. A particular form of structure arises from the skew frequency distribution of the ranks and this has been exploited in the various hierarchical coding models and especially the "structured" model.

The question remains as to whether there is any form of context or other position-dependent structure which could be used to improve compression. This possible 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 rank approximates a negative exponential.
- Choosing coding models according to the last few values emitted (a constant-order Markov or PPM coding model) has negligible effect on the compression.

The tentative conclusion is that a particular rank 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.

Wheeler's latest compressors [18] do attempt to use contextual information. His compressor "arith 6/2", encodes initially into 2-bit groups (0, 1 for runs, 2 for Rank_1 and 3 for escape) and uses then uses 6 of these bits (3 groups) as context information. The author's experience is that the context gives little if any improvement in compression.

11.3 Run length coding

With many of the MTF ranks of most files being zeros, there are often runs of zeros and it is tempting to try to improve compression by using some form of run length coding[4]. Although run length coding is a well established technique, experience here is that it can be very difficult to do it efficiently. There are two basic approaches to handling runs of frequent symbols, with the simplification here that we need consider only runs of zeros. For analysis we consider runs of $L = 2^N$ symbols, with an average run length of 7 or 8.

1. **Direct arithmetic coding**. The arithmetic coder is just allowed to adapt to the frequent run symbols. To determine the cost of a run we must assume that the coder is fully adapted to the run statistics. A run of nominal length L consists of $L+1$ symbols, the L run symbols each with probability $L/(L+1)$ and one terminator with probability $1/(L+1)$. For a run of length 8, the run symbols encode into $8\times\log_2((L+1)/L) =$ 1.36 bits, the terminator encodes into $\log_2(L+1) = 3.17$ bits, or a total of 4.53 bits. Over 75% of the cost is in the termination and is of order $O(\log_2(L))$.

2. **Length-count encoding.** A run of length $L = 2^N$ symbols requires N bits to encode the length, not counting the overheads of signalling and terminating the run. The costs of signalling and terminating the run are now much more difficult to quantify but encoding the length again has a cost of order $O(\log_2(L))$. The implementations used here signal runs by a sequence of 4 zeros and immediately follow that sequence by the bits of the count and a terminating escape.

3. **Wheeler's length-count encoding.** In his latest report, Wheeler [18] uses a novel representation for the length. The method is apparently unpublished and is considerably more efficient than more obvious methods.

   Runs are *always* of the value "0", and the values 0 and 1 are used to encode the run length, but using digit weights of 1 and 2 instead of the more usual 0 and 1. A sequence of bits $x_0x_1x_2\ldots$ then represents the value

$$\sum_{i=0}^{n-1}(1+x_i)2^i = \sum_{i=0}^{n-1}2^i + \sum_{i=0}^{n-1}x_i 2^i$$
$$= (2^n - 1) + \sum_{i=0}^{n-1}x_i 2^i$$

For most values the most significant bit is implied and need not be encoded; the value is represented in one fewer bits than might be expected. A value of 0 cannot be represented, but that does not matter here. The coding may be generated in two ways (least significant bit first) —

- Increment the value by one and encode that modified value as an ordinary binary number, but ignoring the most significant 1 bit.

- Encode much as usual for a binary number, emitting the low-order bit and then shifting right to eliminate that bit. However, before emitting each bit, decrement the current value by 1.

Ranks greater than 0 are incremented by 1 to give an $(N+1)$ symbol alphabet. There is no special code to introduce a run and the run is terminated by any "non-run" code. When combined with the shorter coding for the length itself, the Wheeler length code is an especially efficient method.

Experience is that there is very little to choose between direct arithmetic coding and the more obvious run-

[4] Run length coding here should not be confused with run length coding before the sort. There it is intended to expedite the sorting of some files, and incidentally may give some extra compression. Here run-length coding is purely an aid to compression.

length encoding. Wheeler's modified run-length coding is better than either and is used in the best of the compressors which are described in this report.

[Wheeler uses a variation of this method to encode runs prior to sorting. A sequence of two identical characters "…*cc*" signals a run. The run length is encoded as above, using the two symbols c and $c\oplus 1$, where $\oplus$ denotes an exclusive-OR. If the symbol following the run is either $c\oplus 1$ or $c\oplus 2$, an extra $c\oplus 2$ is inserted as a terminator — ie on decoding ignore any symbol $c\oplus 2$ which terminates the run.]

11.4 Compressibility and rank_0 probability

We saw earlier that the probability of the MTF codes follows an approximate power law. In this section we examine the effect of that exponent on the entropy of the code.

If we assume an initial probability P_1 for the first rank probability and that the probability falls off as a power of the rank, we can adjust the power a to obtain a total probability of 1 for the first 256 symbols. We can also calculate the entropy of that code considered as a source. Mathematically

$P_1 =$	Probability of first symbol, $r = 1$ (1 – origin)
$P_r = P_1 r^a$	Probability of rank r symbol
$\sum_{r=1}^{256} P_r = 1$	
$H = \sum_{r=1}^{256} P_r \log\left(\frac{1}{P_r}\right)$	Entropy of assumed source

The power law dependency on symbol rank makes it appropriate to use 1–origin numbering for the rank, rather than 0–origin as used elsewhere. As a test compressor we use Wheeler's "bred", with a block size of 128 Kbyte.

File	frac 0 MTF	predicted power	predicted entropy
BIB	66.8%	-2.19	1.92
BOOK1	49.8%	-1.71	3.06
BOOK2	60.8%	-2.00	2.30
GEO	35.8%	-1.39	4.11
NEWS	57.9%	-1.91	2.49
OBJ1	50.6%	-1.78	3.00
OBJ2	68.1%	-2.23	1.84
PAPER1	58.4%	-1.92	2.46
PAPER2	55.4%	-1.84	2.66
PIC	87.4%	-3.36	0.75
PROGC	60.3%	-1.98	2.33
PROGL	72.9%	-2.42	1.56
PROGP	74.0%	-2.47	1.49

Table 7. Predicted entropies assuming power-law distribution of rank frequencies

Table 7 shows the exponent required to match the initial (rank_1) probability and the consequent entropy for the files of the corpus. A better view is in Figure 10, where we plot the predicted entropy as a function of the initial rank_1 probability and with it the points corresponding to the files of the Calgary corpus for the "bred" compressor.

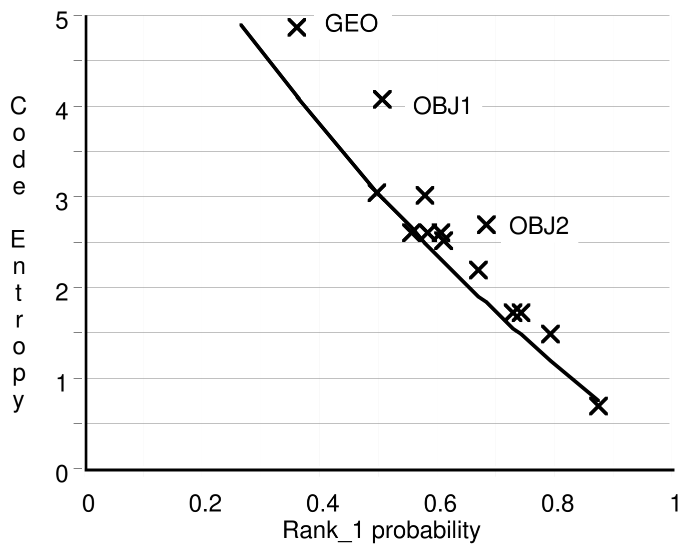

Figure 10 Code entropy v Rank_1 probability – comparison with experimental results

The fit is excellent, considering that some files show considerable divergence from a strict power law behaviour and that the values are based on only the probability of the Rank_1 (elsewhere Rank_0) symbol. Three of the binary files (GEO, OBJ1 and OBJ2 – all labelled) are well above the line and less compressible than the theoretical model might indicate. They have more high-rank symbols than the model expects, as indicated by their larger average MTF distance. OBJ1 also suffers because it is too small to develop good context statistics. PIC (the bottom rightmost point) has its compression improved by its initial run-encoding, an effect wicjh is not allowed for. The general trend is for compression to be slightly worse than predicted, but it is certainly in good agreement with the predictions from the empirical model.

If we used a full-file block on the two large text files (BOOK1 and BOOK2) they would improve considerably (2.93 to 2.60, and 2.46 to 2.19 respectively) purely because of the availability of more context information. The files were compressed with bred and a small block size to eliminate this effect and allow a more-equal comparison of files.

11.5 Following and preceding contexts

The lore of text compression states that using following contexts should give the same results as preceding contexts — this is in fact proved by Shannon in his 1951 paper. Despite this some indications of asymmetric compression results were reported in [12]. A more systematic study, comparing results on the normal Calgary Corpus with compression of reversed copies of those files, is shown in Table 8. Results are shown for the compressors — the simple “Order-0” and “structured model” of this report, Nelson’s COMP2, Bell’s LZB (the standard good LZ-77 compressor), and Wheeler’s “bred”. Bold-face entries show results which better by about 1% compared with the file in the other direction. The legends “fwd” and “rev” refer to the directions of the contexts; with “BS Order-0” and “BS Struct” a forward context refers to the reversed file!

There is just no systematic effect. Most differences may be ascribed to “noise” in measuring the file compression. The one significant difference is for the file GEO and the block-sorting compressors (but not bred). The file consists entirely of 4-byte floating point numbers. All of the numbers end in a zero byte and most have this preceded by a few zero bits. There are relatively few likely values in this third byte and all imply a next value of zero with forward or preceding contexts. With the file reversed, the zero byte cannot predict the next byte nearly as accurately. The compression of 4.31 bit/byte for the “BS Struct” compressor on the reversed file is believed to be the best known compression of GEO.

program	BS Order-0		COMP2		LZB		bred		BS Struct	
context	Fwd	Rev	Fwd	Rev	Fwd	Rev	Fwd	Rev	Fwd	Rev
bib	2.18	**2.13**	2.15	2.15	3.17	**3.15**	2.07	**2.04**	1.97	1.95
book1	2.55	**2.52**	2.47	2.47	3.86	3.86	2.60	**2.57**	2.40	2.39
book2	2.22	2.20	2.28	2.28	3.28	3.28	2.19	2.17	2.05	2.04
geo	**4.75**	4.81	5.05	5.06	6.17	**6.09**	4.87	4.87	**4.31**	4.50
news	2.69	2.68	2.72	2.72	3.56	3.56	2.62	2.60	2.51	2.50
obj1	4.20	4.20	4.03	**3.97**	**4.27**	4.31	**3.91**	3.94	**3.82**	3.87
obj2	2.73	2.71	**2.77**	2.80	**3.14**	3.19	2.58	**2.52**	2.49	**2.46**
paper1	2.63	2.61	2.54	2.55	2.23	3.23	2.58	**2.55**	2.48	2.46
paper2	2.59	2.57	2.48	2.48	3.43	3.41	2.58	**2.53**	2.44	**2.41**
pic	0.83	0.83	0.87	0.87	1.03	1.03	0.83	0.82	0.77	0.77
progc	2.68	2.68	2.61	2.60	3.08	3.09	2.58	2.56	2.50	2.49
progl	1.85	1.85	1.94	1.94	2.12	2.11	1.78	1.78	1.71	1.72
progp	1.84	1.84	1.89	1.88	2.09	2.10	1.77	1.76	1.70	1.70
trans	1.62	1.61	1.79	1.84	2.12	2.13	1.56	1.54	1.50	1.50
Average	2.53	**2.52**	2.54	2.54	**3.11**	3.18	2.47	**2.45**	**2.33**	2.34

Table 8. Compression with forward and reversed files of the Calgary corpus.

12. Interpretation of block sorting compression

Cleary etal [7] showed that the permutation step of block sorting can be achieved using the data structures of their PPM* compressor, so that block sorting is in a sense equivalent to context modelling compressors such as the PPM family. Context modelling compressors are already known to be equivalent to the dictionary compressors of Ziv and Lempel. Bunton [3] has analysed DMC compression (Dynamic Markov Coding) and with Cleary has shown that it too uses data structures similar to those of PPM*. Thus despite the initial appearance of the block sorting algorithm being quite different from other data compression algorithms, it definitely belongs as a new member of the family of established techniques

12.1 The Burrows and Wheeler approach

Following on from the original report by Burrows and Wheeler, we can regard block-sorting compression as a sequence of three processes —

1. The initial, sorting, stage permutes the input text so that similar symbol contexts are grouped together. The permutation creates strong locality because the grouping of the (invisible) contexts collects together the few symbols likely to occur in each context.
2. The Move-to-Front phase then converts the various local symbol groupings into a single global structure. 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 11.2.

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, probably 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.

12.2 The symbol ranking approach

An alternative interpretation considers it as a symbol ranking compressor. In a sequential symbol-ranking compressor each context has its own individual symbol ranking list and the compressor must maintain an appropriate complex of contexts and lists, switching contexts as individual symbols are processed. In block-sorting compression the Move-to-Front list is used as the current estimate of the symbol ranking. The initial sorting transformation, by collecting together similar contexts, ensures that the preferred symbols usually change relatively slowly and that a single ranking list is adequate for all contexts.

The heart of the operation is therefore the Move-to-Front list, with the initial sorting transformation a preprocessor which ensures that the list remains close to its optimum ordering. As the sort performs a context analysis of the input text the new algorithm therefore bridges two of the techniques described above (and employs the third as well in the final, statistical compressor). In comparison with simple Move-to-Front (MTF) we see that it includes contextual information, whereas MTF works entirely from symbol recency — it may be most accurate to regard block sorting as a preprocessor for assisting a traditional MTF "compressor".

13. Conclusions

Block sorting data compression is an interesting and novel technique which has been confirmed as a practical data compression scheme, combining high speed with good compression performance. It is already established as a member of the general family of statistical compressors, being realisable in terms of data structures used with more-conventional statistical compressors. The crucial "block-sorting" initial step acts as a preprocessor to the already known Move-to-Front transformation. Together these two operations provide an implementation of a context-dependent "symbol ranking" transformation first used by Shannon in 1951. Although provably equivalent to PPM and similar statistical modelling techniques, block sorting is best regarded as a member of this long-established though little-recognised compression method.

The compression achieved with block sorting is comparable with that of some PPM-style schemes, but not quite as good as the best of these compressors. In comparing the two, block-sorting has the advantage that it does not need to encode escapes between models (an advantage which it shares with symbol ranking compressors in general), but has the grave disadvantage that the original context structure is unavailable to the statistical coder and decoder.

The sorting and MTF transformations on the input data appear to remove most, if not all, of the contextual structure on which traditional text compression depends. Improving compression over that achieved with very simple coding models has required the development of statistical models and coders which are especially designed for handling skew symbol frequency distributions. Further study is needed to decide whether statistical predictive methods can improve the compression, but preliminary results indicate that improvement may be unlikely.

13. 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] J.L. Bentley, D.D. Sleator, R.E. Tarjan, V.K. Wei. "A locally adaptive data compression algorithm", *Communications of the ACM*, Vol 29, No 4, April 1986, pp 320–330

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

[4] M. Burrows, private communication

[5] 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`

[6] 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`

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

[8] J.G. Cleary, I.H. Witten, "Data compression using adaptive coding and partial string matching", *IEEE Trans Communications*, COM-32, vol 4, pp 396–402 April 1984.

[9] P.M. Fenwick, "A New Technique for Self Organising List Searches", *Computer Journal*, pp 450–454, Oct. 1991.

[10] P.M. Fenwick, "Block sorting text compression", *Australasian Computer Science Conference, ACSC'96*, Melbourne, Australia, Feb 1996. `ftp.cs.auckland.ac.nz /out/peter-f/ACSC96.ps`

[11] 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`

[12] 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`

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

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

[15] M. Nelson, "Arithmetic coding and statistical modelling", *Dr Dobbs Journal*, Feb 1991.
`wuarchive.wustl.edu /systems/msdos/msdos/ddjmag/ddj9102.zip`

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

[17] W.J. Teahan, private communication

[18] 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`]

[19] I.H. Witten, A. Moffat and T.C. Bell, "*Managing Gigabytes : Compressing and indexing documents and images*", van Nostrand Reinhold, 1994

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