Enhanced Word-Based Block-Sorting Text Compression

R. Yugo Kartono Isal **Alistair Moffat** **Alwin C. H. Ngai**

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

http://www.cs.mu.oz.au/~yugo
http://www.cs.mu.oz.au/~alistair
http://www.cs.mu.oz.au/~achn

Abstract

The Block Sorting process of Burrows and Wheeler can be applied to any sequence in which symbols are (or might be) conditioned upon each other. In particular, it is possible to parse text into a stream of words, and then employ block sorting to identify and so exploit any conditioning relationships between words. In this paper we build upon the previous work of two of the authors, describing several further recency rank transformations, and considering also the role of the entropy coder. By combining the best of the new recency transformations with an entropy coder that conditions ranks upon gross characteristics of previous ones, we are able to obtain improved compression on typical text files.

Keywords: Burrows Wheeler transformation, text compression, word-based modelling, recency ranking, arithmetic coding.

1 Introduction

The Burrows Wheeler transform (BWT) is a surprisingly simple, but surprisingly elegant, mechanism for permuting a block of characters so as to reveal any inter-character dependencies that might be present. Its use in text compression programs is now well known (see, for example, Witten et al. [1999, page 65]), and implementations such as BZIP2 [Seward, 1999] are in wide use for general purpose compression.

What is less widely appreciated is that the BWT can be applied to any sequence of symbols, and if there are conditioning dependencies, will assist in making them obvious. In previous work we have shown that text interpreted as a stream of word numbers can also benefit from the use of the BWT [Isal and Moffat, 2001a,b], with the result being improved compression effectiveness for text files.

In this paper we continue our exploration of word-based parsing in conjunction with the use of the BWT. We focus on two particular aspects of the compression chain: the ranking transformation used to convert the BWT output into a low-entropy stream of ranks that can then be entropy coded; and on the entropy coder itself. Several new ranking transformations are described for use in large-alphabet situations, and their implementation merits (or otherwise) discussed. We also show that a coder that conditions upon gross characteristics of previous ranks can obtain slightly better compression.

By combining the best of the new recency transformations with the conditioning coder, we are able to obtain improved compression on typical text files.

The paper is organised as follows. In the remainder of this section we briefly remark upon some of the related literature. Section 2 then summarises the previous work of the first two authors with regard to the word-based parsing regime used, approximate MTF transformations, and the structured arithmetic coder SINT. Section 3 extends the approximate ranker, and describes several other transformations that have the required properties. Entropy coding is then considered in Section 4, and a conditioning coder introduced that exploits gross properties of the ranked symbol stream.

In related work, Grabowski [1999] proposes that text files be preprocessed by converting capital letters to lower case equivalents coupled with a flag to indicate the conversion. Space stuffing – or prepending with a space symbol – is used to handle symbols which are hard to predict, such as newline characters and punctuation marks. Grabowski also suggests assigning unused symbols in the alphabet to static phrases, primarily frequently used 2-, 3-, and 4-grams in English. The objective of these preprocessing techniques is to increase the skewness of the distributions of letters while keeping the set of symbols small, and they result in slightly better compression.

In a somewhat parenthetical remark, Sadakane [1999, page 93] also proposes that block sorting and word-based modelling be coupled. Our work can be regarded as being an elaboration of that observation.

Finally in this section we describe the five test files we have used in this investigation. Table 1 summarises their attributes. The first file is taken from the Canterbury Corpus (see http://www.corpus.canterbury.ac.nz for

File	Size (in MB)	Symbols
asyoulik.txt	0.119	51,463
world192.txt	2.359	658,212
bible.txt	3.860	984,495
E.coli	4.424	4,510,733
wsj20	20.000	4,794,109

Table 1: Test files and sizes, in MB and in terms of symbols created by the spaceless words parsing process.

a description of the Canterbury Corpus files, and compression results relating to them); the next three comprise the Large Canterbury Corpus (LCC, available from the same location); and the fifth file – `wsj20` – consists of 20 MB of SGML-tagged text taken from the *Wall Street Journal*. While file `E.coli` is not a text file, it is included so that the complete LCC is covered by our tests.

Where experimental times are reported, they represent the CPU time required on an unloaded 650 MHz Pentium III with 256 MB RAM and 256 kB on-die cache.

2 Spaceless Words and the BWT

The spaceless words parsing model we use throughout this paper is due to de Moura et al. [2000]. They use the model in a semi-static manner to build an external dictionary of words that is separately transmitted, whereas here we use the same model adaptively, transmitting the dictionary implicitly by "spelling out" each new word the first time it occurs.

In the spaceless words model the tokens are based upon alphanumeric and non-alphanumeric character types. Contiguous strings of like characters are isolated, and transformed into integers via the use of the dictionary of strings; with the additional proviso that if the non-word string between two word strings consists of a single blank character, it is elided. Isal and Moffat [2001b] give an example showing the action of the parsing strategy.

Using this first transformation, a message of text is reduced to a stream of integer values, some of which (the ones with values less than 256) represent primitive characters, and the remainder of which represent either alphanumeric strings (the "words") or strings of punctuation and white space characters ("non-words"). The original message can be uniquely reconstructed from this stream of integers. For example, when file `bible.txt` is processed, it is reduced to a sequence of $984{,}495$ four-byte integers (which, by coincidence, is almost the same size as the initial file), of which $94{,}068$ or 9.6% are less than 256 and represent primitive characters rather than longer strings.

This process is not suited to non-text files. For example, on file `E.coli`, which contains no white space characters, the parsing process generates a sequence of "words" in which there are almost no repetitions, meaning that each word is spelt out in full and never reused. That is, the stream of symbol identifiers generated is dominated by the characters of the original file as words are spelt out. This is why (Table 1) the number of symbols generated for this file is disproportionately large.

The second stage of the process is the BWT. The transform permutes the set of symbols in the input message, and has the effect of bringing into physical proximity the symbols following like prior contexts. Hence, if some prior context – as a very simple example, the single letter "q" – is typically followed by one of just a small number of symbols, then the BWT output will include a segment that includes those possible following symbols. For example, if the integer stream contains n "q" characters (that is, integer "113"), then after the BWT there will be a contiguous section in the permuted message that contains all of the n integer symbols that follow the n "q"s.

The same is also true for longer conditioning strings, and if there are n appearances in the integer stream of the four-integer string "$81, 117, 101, 101$" (representing "Quee"), then there will be a contiguous section of length n characters in the permuted message that contains the set of following integers. In the case of the latter example, we would imagine that all n following symbols would be either "110" (for "n") or "114" (for "r"), and that coding that segment could be economically accomplished by listing which of the two appears using perhaps just one bit per symbol.

The examples in the previous paragraph were character based and motivated. But at a word-level, the same relationships continue to hold. Once a word has been encountered for the first time and spelled out as a character string, subsequent appearances are represented by an integer word number. Hence, if integer "1615" is the word "Burrows", and integer "2795" is the word "Wheeler", then after the BWT all of the different words that follow integer "1615" will form a contiguous segment, and, at least in this paper, we would expect that segment to be dominated by the integer "2795".

Again, we refer the reader to other sources for a detailed description of the process involved, and the computation of the inverse transform.

Because the BWT output can be thought of as being the concatenation of segments, each of which has a distinctive nature because of conditioning, it makes sense to condense each segment down to a small set of distinct symbols, and use a specific entropy code within each segment. Hence, in the segment corresponding to a prior context of "q", the probability of a "u" might be determined to be very high, and its resulting codeword very short. While such direct post-BWT coding techniques are possible [Wirth and Moffat, 2001], they suffer from the considerable difficulty that the segments are not explicitly known while the inverse BWT is being carried out.

As an alternative, the subalphabets within each segment can be left implicit, and the segments themselves allowed to segue seamlessly through the use of the third transformation in the standard chain – recency ranking.

The simplest ranking technique is the well-known MTF transformation, in which each value is replaced by one plus the count of the number of distinct symbols encountered since the last appearance of that symbol. For example, the integer sequence

$$97, 97, 97, 101, 101, 97, 101, 117, 117, 101, 97, 117, 117$$

would be replaced by

$$1 + M_{97}, 1, 1, 1 + M_{101}, 1, 2, 2, 1 + M_{117}, 1, 2, 3, 3, 1$$

where M_x indicates the number of distinct symbols coded since the immediately preceding occurrence of x. As can be seen, when a segment of like symbols is encountered – in the case of the example, a segment containing only the letters "a", "e", and "u" ("97", "101", and "117" respectively) – the MTF transformation reduces the sequence of initially large integers to a sequence of very small ones.

On the same `bible.txt` file discussed above, after the MTF transformation fully 87.8% of the integers in the stream are less than 256. Indeed, 23.6% of the integers in the post-MTF sequence are "1", a strong corroboration of the effect of the BWT, and of the extent to which both characters and words are conditioned upon their predecessors.

Figure 1 shows graphically the probability distribution of symbols on file `bible.txt` (the grey bars) and the corresponding distribution of MTF ranks (the dotted line). The

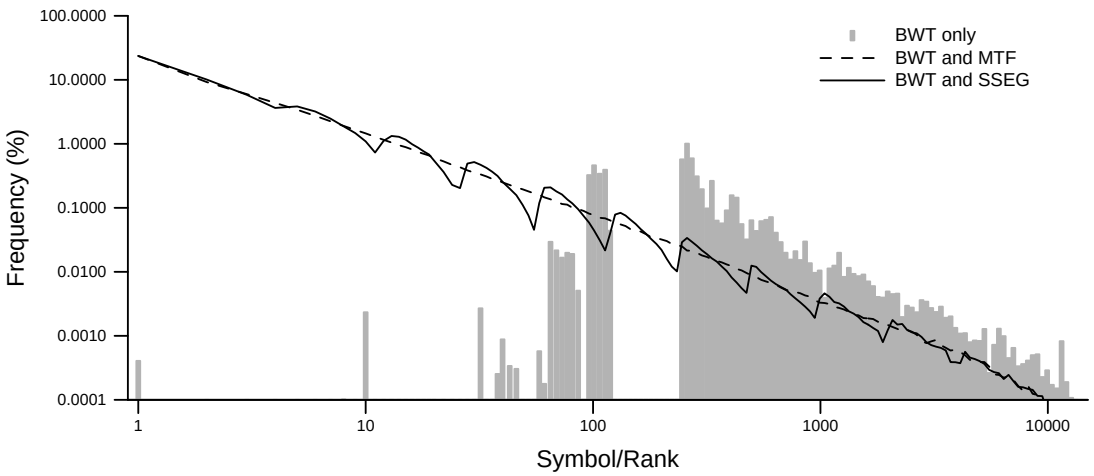

Figure 1: Probability distribution of symbols, MTF ranks, and SSEG ranks for `bible.txt`. All of the probability distributions were smoothed for presentation, by generating buckets each 5% larger than the previous one, and reporting the average probability for the symbols in each bucket.

graph was created by averaging over frequency buckets, with each bucket approximately 5% larger than the previous one. For example, all symbols in the range 1400 to 1470 inclusive were collected into one bucket, and their average frequency plotted on the graph. Note the gap between symbol 128 and 255, which arises because the first word number allocated is always 256, in case any non-ASCII characters appear as primitives in the input file.

As can be seen, the rather chaotic distribution of integer values in the BWT sequence is reduced to a much smoother – and lower entropy – sequence by the MTF. But note also that there is a definite trend amongst the word numbers, and that is that the greater the word number (and hence the later in the text it is first encountered), the lower its probability. This effect will be exploited below. The solid line in Figure 1 will also be discussed shortly.

To implement the MTF transformation for character-based BWT streams is easy – a simple array or linked list will suffice. But for large-alphabet applications, linear search in an array or linked list is hopelessly inefficient, and a better structure is required. In our earlier work a Splay tree [Sleator and Tarjan, 1985] is used to implement the MTF transformation, a possibility originally noted by Bentley et al. [1986]. Use of a Splay tree allows the item of rank t to be identified in $O(\log t)$ amortised time (compared with the $O(t)$ time required in a linked list implementation) and makes the process tractable. For details of the implementation, see Isal and Moffat [2001b].

Once the MTF transformation has been applied, an entropy coder is used to reduce the new sequence into a bitstream. Any coder can be used, with the usual assumption being that all conditioning has now been recognised, and that there is no remaining correlation between consecutive symbols. In the case of a character-based parser, BWT, and MTF, the post-MTF sequence will again be over an alphabet of 256 symbols, so a zero-order character-based coder can be used. In the case of the word-based model a larger alphabet must be handled, but this poses few problems from a coding point of view, and either minimum-redundancy (Huffman) or arithmetic coding can be used. The subject of entropy coding will be returned to in Section 4 below.

3 Approximate Ranking

In their original paper on block-sorting compression, Burrows and Wheeler [1994] noted that a slight improvement on compression might be achieved if symbols that had not been seen in a very long time are moved only part-way to the front rather than into position number one. Other variants on the MTF strategy – such as moving the symbol to the second location in the list, unless it was already in the second location, have also been explored [Balkenhol et al., 1999, Chapin, 2000]. These variations illustrate an important freedom – any reversible transformation may be used to convert the BWT sequence into an equivalent sequence, and there is nothing sacrosanct about the use of the MTF.

One alternative that we have explored is based upon the use of a forest $\{T_i\}$ of search trees. The ith tree in the forest contains S_i values, with the most recently accessed items being stored in tree T_0, the next most recently accessed items being stored in T_1, and so on. Within each tree the items are stored in the usual key-based ordering (that is, based upon the integer value being represented); but within each tree the items are also doubly threaded in a conventional MTF list. Figure 2, taken from Isal and Moffat [2001b], shows this arrangement.

To calculate a rank for an item, its tree is determined, and then its ordinal location within that tree used as an offset. For example, if the tree sizes are $S_0 = 1$, $S_1 = 3$, $S_2 = 7$, and so on, then a search for the 6th most recently accessed item will generate a rank of between 5 and 11, depending upon the key values of the other items in tree T_2.

A pseudo-MTF transformation can then be achieved if, once an item is accessed, it is removed from its tree and inserted into T_0, with the oldest item in each intervening tree being "bumped" down one tree, to become the newest item in the next tree.

If the tree depths double from one to the next, with $S_i \approx S_{i-1}^2$, then $O(\log t)$ time per operation can be achieved. We call this approach the DSEG transformation.

However, this double-powers sequence results in a relatively small number of trees ($S_5 = 2^{16} - 1$, and $S_6 = 2^{32} - 1$), and the movements of items between trees can be quite dramatic. Use of a slower-growing sequence $S_i = 2^{i+1} - 1$ in which $S_i = 2S_{i-1} - 1$ increases the

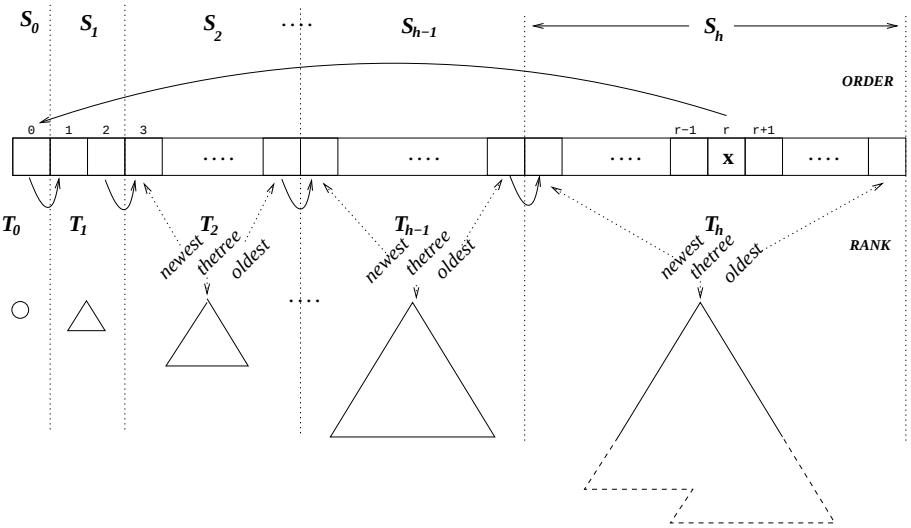

Figure 2: Recency computation using a forest of trees of increasing size (from Isal and Moffat [2001b]).

execution time to $O(\log^2 t)$ per operation, but results in a more accurate approximation to the exact MTF mechanism.

We call this latter option the SSEG transformation [Isal and Moffat, 2001b]. The sawtooth line in Figure 1 shows effect of applying the SSEG transformation to the file `bible.txt`. The repeated pattern shown in the graph arises because, within each tree, objects are stored (and thus assigned ranks) in their integer order. The triangular grey region of Figure 1 already demonstrated that there is a strong correlation between symbol number and frequency, and this relationship is partially preserved by the SSEG transformation, with each tooth of the pattern corresponding to one of the trees in the ranking forest.

In the remainder of this section we consider a number of variations of the SSEG approach, and consider the implementation advantages of each. Experimental results comparing the various methods are given at the end of this section.

To allow an accurate comparison between alternative methods, and to insulate our results (at this stage) from any vagaries of the particular coders used, we calculate the zero-order self-entropy of the transformed streams. That is, for a stream in which symbol i appears f_i times, we calculate

$$H = -\sum_{i=1}^{n} \frac{f_i}{N} \cdot \log_2 \frac{f_i}{N} \ ,$$

where $N = \sum_{j=1}^{n} f_j$, as the average number of bits per symbol required to represent that message, assuming an ideal coder and no cost for the coding prelude. The resultant values are somewhat idealised and possibly not attainable; nevertheless, the numbers so computed provide a basis for comparing ranking methods, since in all cases the alphabet size is approximately the same, and the eventual probability distribution similar.

Neighbour

In our quest to balance compression effectiveness and compression efficiency, the simplest technique is to only partially promote the item accessed. For example, if the symbol is currently in tree T_j, removing it from T_j and inserting it as the newest item in tree T_{j-1}, while simultaneously pushing the least recently used object in tree T_{j-1} into tree T_j, will only require $O(\log t)$ time when the symbol in question has rank t. Indeed, any variant of this strategy that cycle objects within a fixed number of the trees in the forest has the same asymptotic cost.

Because each tree is ordered by integer symbol number, promoting an object from T_j to T_{j-1} on average halves its rank. While this is a substantial decrease, it is possible that symbols will move too slowly towards T_0 and its low-rank high-probability zone compared to MTF and SSEG, and that compression effectiveness will degrade.

We call this first approach the "neighbour" strategy, and denote it as SSEG-N.

Halfway

A more balanced compromise between the slow shuffle of SSEG-N and the rapid response of the base SSEG approach is to promote each accessed item from its current tree T_j to tree $T_{\lfloor j/2 \rfloor}$, reducing its rank to approximately the square root of what it was prior to being accessed. The least recently accessed symbol in each of the intervening trees is then percolated downward to restore the tree sizes, and so total execution time is $O(\log^2 t)$ to process the symbol ranked in position t. Nevertheless, the number of trees handled is halved compared to SSEG, and a practical speedup might result.

We call this approach the "halfway" strategy, and denote it as SSEG-H. Because we number the trees from zero, this strategy (as does the neighbour heuristic) only allows symbols in tree T_1 to move into T_0 and rank 1, and has the useful benefit of mirroring the "move-one-from-front" strategy, in which the only symbol that can displace the one currently in rank 1 is the symbol currently ranked 2.

Skipping

The halfway strategy also suggests another, in which the trees between T_j and $T_{\lfloor (j-1)/2 \rfloor}$ are skipped over rather than transitted. Extending this idea, we have the "skipping" mechanism, in which the accessed item is promoted into T_0, the least recent item in T_0 demoted into T_1, the least recent symbol in T_1 moved into either T_2 or T_3, and

File	Ranking method				Variations				
	None	MTF	DSEG	SSEG	SSEG-N	SSEG-H	SSEG-S	SSEG-T	SSEG-C
asyoulik.txt	7.34	6.46	6.55	6.40	6.39	**6.31**	6.62	6.40	6.34
world192.txt	9.12	5.65	5.78	**5.62**	5.95	5.63	5.82	**5.62**	5.71
bible.txt	8.46	6.50	6.60	6.39	6.41	**6.28**	6.66	6.39	6.33
E.coli	2.06	2.08	2.08	2.08	**2.05**	**2.05**	2.09	2.07	**2.05**
wsj20	10.21	7.52	7.66	7.37	7.40	**7.22**	7.69	7.37	7.28

Table 2: Zero-order self-entropy of test files using different recency ranking techniques, after conversion using the implicit dictionary spaceless words approach, and after the BWT transform is applied. All values are in bits per symbol, where each symbol is an integer generated by the parsing transformation. The parameter τ used in SSEG-T was set at $\tau = 16$.

so on, following the binary path back down to T_j. For example, if an object is T_{11} is being promoted, it is inserted into T_0, then the oldest symbol in T_0 moved to T_1, the oldest symbol in T_1 moved to T_2, the oldest object in T_2 moved to T_5, and the oldest symbol in T_5 moved to T_{11}, to complete the cyclic update.

Because $\log_2 S_{\lfloor j/2 \rfloor} \approx (\log_2 S_j)/2$ when $S_i = 2^{i+1} - 1$, the total running time for the $\log_2 j$ tree insertions and deletions is $O(\log t)$, and asymptotic efficiency is regained.

In the results below this approach is denoted as the SSEG-S technique.

Threshold

If the objective is to reduce the number of trees affected by each symbol promotion, another obvious method is to presume that high-numbered symbols will not reappear soon, even in the BWT sequence, and so not promote them. That is, the forest of trees, and the symbols therein, is separated into two parts. In the first part, in trees numbered up to and including a threshold τ, symbol occurrence is followed by a promotion to tree T_0, and percolation back down of the least recent item in each tree until stability is regained. But in the second part of the forest, occurrence of a symbol in a tree numbered greater than τ is not followed by promotion, and the forest is left unchanged. Because the symbols are assigned to trees in order of discovery, this approach effectively means that symbols guessed to occur frequently are always promoted, while symbols guessed (by virtue of their high ordinal symbol number) to be infrequent are never promoted. Hence, a stable one-to-one entropy code will be used in the second part of the forest.

We call this strategy the "threshold" approach, and denote it by SSEG-T.

Counting

Another possibility is to realise that we do in fact know something about each of the symbols being ranked – its frequency in the stream to date can easily be monitored and used as a guide to how far or quickly it should be promoted. For example, there might be little benefit to be gained by promoting into tree T_0 after the very first occurrence of a symbol.

Our proposal here is that, if the symbol accessed has frequency f_i in the section of the sequence processed so far, then it should be promoted from tree T_j to tree $T_{j'}$, where

$$j' = \left\lfloor j \cdot \left(1 - \frac{\log f_i}{\log N}\right) \right\rfloor$$

and N is the total number of symbols processed so far. For example, if this is the symbol's first occurrence, then it is promoted to tree T_{j-1}. On the other hand, when a symbol in T_1 is processed, or when a very common symbol from another tree is processed, it is promoted to tree T_0.

We denote this "counting" approach as SSEG-C.

Results

Table 2 shows, for each of the five test files, and each of the recency ranking methods, the zero-order self-entropy of the ranked sequence.

As can be seen, there is a considerable variation in effectiveness. Overall the best method appears to be SSEG-H. It slightly outperforms SSEG-C on all of the files and is also consistently better than the original SSEG heuristic. Compared to SSEG-H, it appears that promoting symbols all the way to the front (SSEG) is too dramatic; promoting them by just one tree (SSEG-N) too slow to adapt; demoting by skipping down the forest (SSEG-S) insufficiently smooth; and SSEG-C perhaps just too complex. With the exception of SSEG-S, all of the SSEG-family methods outperform both the standard MTF ranking rule, and the DSEG heuristic, which uses tree sizes that grow as double powers of two.

Figure 3 plots compression effectiveness on file wsj20 as a function of the cost of the recency transformation for the various mechanisms discussed. The fastest transformation is MTF, which makes use of just a single tree and is asymptotically efficient; the slowest is SSEG, which requires $O(\log^2 t)$ time per update. Between these extremes, the different heuristics listed above offer varying compromises in terms of execution speed, and (measured by the self-entropy of the streams) do not necessarily trade away compression effectiveness to attain that speed. The thresholding approach SSEG-T can be made very fast by setting a small part of the symbol set in which promotions are permitted, but compression effectiveness suffers considerably. The best choices appear to be MTF, if speed is paramount; SSEG-H if compression is more important than speed; and SSEG-N if a balance is required between efficiency and effectiveness.

We also experimented with a version of SSEG-N that promoted by one tree if the accessed symbol was in T_1 or T_2, and by two trees otherwise. For file wsj20 the self-entropy of the ranked stream was slightly lower than shown in Table 2, at 7.29 bits per symbol. It requires slightly more time than SSEG-N.

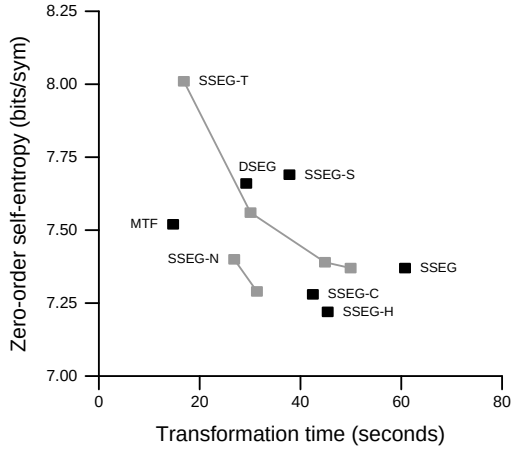

Figure 3: The relationship between time spent undertaking the transformation (average of forwards and reverse transformation time, in CPU seconds) and the zero-order self-entropy of the transformed sequence, for different recency ranking transformations, and for file `wsj20` when parsed using the spaceless words mechanism. The parameter τ used in SSEG-T was varied between $\tau = 10$ and $\tau = 16$ in steps of 2. Two versions of SSEG-N were tested, that promoted by either one tree or two trees.

4 Entropy Coders

The symbol stream produced by the ranking transformation must still be entropy coded, and we now turn our attention to this step.

Fenwick [1996] showed that better compression could be obtained if a structured arithmetic coder was used in preference to a uniform one. Both types of coder are nominally zero-order, and as such should not be able to obtain compression effectiveness better than the zero-order self-entropies described in Table 2.

But the streams generated by the BWT and then ranking process are not completely random, and having a further level of adaptation is beneficial. To understand why, recall that the ranks that are being coded correspond to segments in the BWT sequence representing the following symbols of like contexts in the source text. Some of these segments have localised alphabets that are relatively large. Others are very small. For example, of the exactly 50 occurrences in `wsj20` of the word "Margaret", there are only 12 different successors, and of those 12, eleven only appear once. There are, of course, 39 occurrences of the word "Thatcher". In this case the set of following symbols is essentially of size 1. The next segment in the BWT output may well correspond to the successors of the word "Thatcher", but in this case the segment is 75 symbols long, and contains 50 different symbols (the most common successor is the "is", which still only appears 6 times). It clearly makes no sense to code the ranks in these two segments using the same probability distribution – markedly different distributions are required.

In the structured arithmetic coder each rank is broken into two components. The first is a *selector*, which indicates the magnitude of the rank. The probability distribution for the selector is set so that adaptation at the selector level is fast, and that localised patterns in the selector statistics will be rapidly absorbed, over sequences of just a handful of consistently small or consistently large selector values. The second component of each rank is an offset within the bucket of values corresponding to the selector. In most of the buckets there is a much greater range of values than there is at the selector level, and the adaptation is much more conservative, so that accurate statistics are maintained within each bucket.

In our structured coder SINT [Isal and Moffat, 2001a] the buckets are determined by a geometric sequence with radix 1.5. Rounded to integers, this puts symbols 1 and 2 in buckets of their own; $[3, 4]$ in the third bucket; $[5, 6, 7]$ in the fourth; $[8, 9, 10, 11]$ in the fifth, and so on.

Another way to understand the benefits of the structured coder is to consider the effect of coding (say) the symbol with rank 100 – when the selector that covers the bucket containing rank 100 is used, its frequency count is adapted, and all symbols in that bucket get an effective boost to their probabilities. This makes sense, since (Figure 1) the ranks form a continuous distribution.

Balkenhol et al. [1999] have also worked with selectors and secondary distributions in their BWT implementation. One critical difference between their work and that of Fenwick is that they condition each selector upon previous selectors. It is that thread of development that we exploit in this section, as we describe three versions of a conditioning structured arithmetic coder CINT.

Full conditioning

In seeking to condition ranks it was clear that a full first-order compressor for ranks would be ineffective, as there would be a large number of conditioning states in use, and in the majority of the states the statistics gathered would, of necessity, be imprecise. Instead, we conditioned the sequence of selectors, in the same manner as proposed by Balkenhol et al. [1999]. That is, the bucket offsets continued to be coded as zero-order values, conditioned only upon knowledge of the bucket to be used, while the selector component of each rank is conditioned on the selector component of the immediately preceding rank. This alternative is shown in Figure 4, and is denoted by CINT-F. For the non-`E.coli` test files, the largest number of selector values was 28, and so the selector matrix (that is, a vector of states, each of which involves a vector of integer probability estimates) required only just a few kilobytes of memory.

We denote this alternative as CINT-F.

Quintuple conditioning

Even if only the selector component is conditioned, there are still a non-trivial number of conditional probabilities to be maintained, and it is hard to arrange for the adaptation to be as speedy as required.

To further reduce state dilution, we also experimented with a reduced version, in which just five classes of selector value were considered: selector 1 (which only represents rank 1) as class A; selectors 2 and 3 (which together account for ranks 2 to 4) as class B; selectors 4 to 7 (ranks 5 to 26) as class C; selectors 8 to 15 (ranks starting at 27) as class D; and all other selectors (ranks 710 and greater) as class E.

Use of the class associated with the previous rank – in a sense, a value calculated as $\log_2 \log_{1.5} t$ for a given rank t – then gave five conditioning states and a more compact set of statistics to be maintained.

We denote this strategy as CINT-Q, where the "Q" stands for "quin".

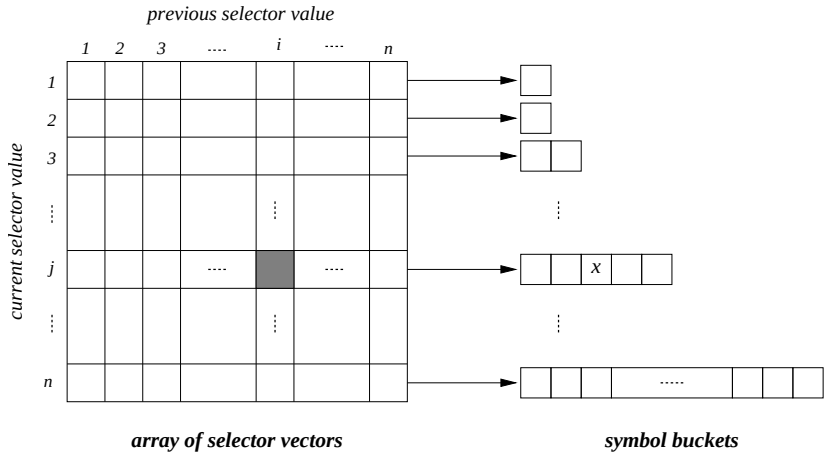

Figure 4: Using a matrix of selectors to condition the probability estimates of symbols. The current symbol x in bucket j is coded in the context of the bucket number i of the previous rank.

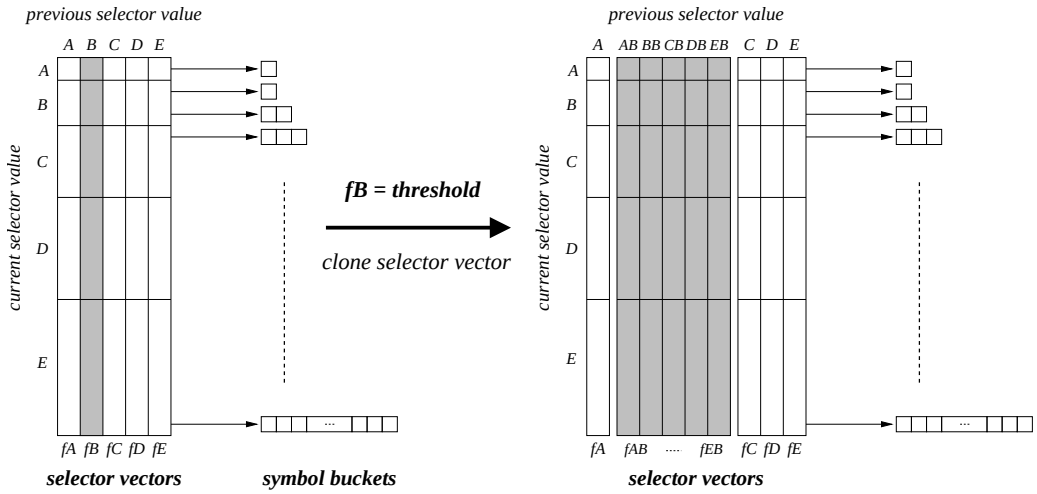

Figure 5: Cloning in the state machine that represents conditioning classes. State B, with total frequency f_B, reaches the cloning threshold, so five new states are created, representing the two-symbol classes *AB*, *BB*, *CB*, *DB*, and *EB*, and each is given one fifth of the integer-valued probability estimates of state *B*. The statistics used in these five classes will then diverge if it is appropriate for them to do so; and each of the new states starts with a frequency counter of zero. No new buckets are created, and no adjustment made to the probability estimates recorded in the buckets.

Adaptive conditioning

Having obtained a slight gain through the use of first order conditioning on selectors, our thoughts naturally turned to second and higher order predictions, to see if further slight gains could be garnered. But we were also wary of state explosion, even if selector classes were used as the control rather than selectors or raw ranks. While a long context might be more effective for commonly used values, a short context is to be preferred for less frequently occurring items so that the probability estimates converge appropriately.

To balance these concerns, we implemented an adaptive context splitting mechanism that has some parallels to the DMC compression mechanism [Witten et al., 1999, page 69]. Initially five selectors are used, one for each of the selector classes *A*, *B*, *C*, *D*, and *E*. Each state tracks the number of times it has been used, and if its usage reaches some threshold, it is split into five new states, the existing frequency counts are split five ways amongst the new states, and coding continues as before, except that an additional quantum of prior history is used for some subset of the states.

Figure 5 shows the process of state cloning. In the figure, the state corresponding to a previous context of class *B* is assumed to have reached the cloning threshold, and so is split to make five new states representing prior class contexts of *AB*, *BB*, *CB*, *DB*, and *EB*. State *BB* might then continue to be heavily used, and subsequently cloned to make further new states *ABB*, *BBB*, *CBB*, *DBB*, and *EBB*.

After preliminary trials we settled on a threshold of 2^{16} to control the cloning. It appeared to represent a plausible balance between over-aggressive cloning, which is too busy to settle down and be useful; and more conservative cloning, which is too sluggish in its adaptation.

This adaptive approach is designated in the results below as CINT-A.

Results

Table 3 summarises the compression effectiveness of the various coders at our disposal. Each compresses a sequence of integer values to a bitstream. The SHUFF program is a minimum-redundancy (Huffman) coder included

File	Self Entropy	Zero-order			Conditioned		
		SHUFF	UINT	SINT	CINT-F	CINT-Q	CINT-A
asyoulik.txt	2.59	2.67	2.70	**2.48**	2.49	**2.48**	**2.48**
world192.txt	1.50	1.51	1.52	1.42	1.38	1.38	**1.37**
bible.txt	1.53	1.55	1.54	1.49	1.49	**1.48**	**1.48**
E.coli	1.99	2.20	2.06	2.01	**2.00**	**2.00**	2.01
wsj20	1.65	1.67	1.66	1.56	**1.55**	**1.55**	**1.55**

Table 3: Compression results using different entropy coders. In all cases the spaceless words model, followed by BWT, followed by SSEG-H is used. The second column shows the zero-order self-entropy from Table 2, converted to bits per character relative to the original source file; other columns show the performance of actual entropy coders, again expressed in bits per character.

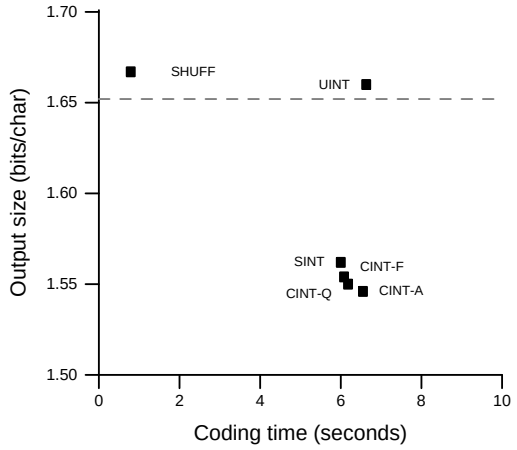

Figure 6: The relationship between time spent coding (average of encoding and decoding times, in CPU seconds) and actual compressed size of the output file (bits per character of the original file) for different entropy coding mechanisms, and for file wsj20 when parsed using the spaceless words mechanism and ranked using SSEG-H. The dotted horizontal line shows the zero-order self-entropy of the transformed file in bits per character.

for completeness; the others are all arithmetic coders. The UINT program is an adaptive uniform coder with a long and stable "memory", and so is unable to flexibly distinguish between the distributions required for, say, "Margaret"-like symbols and "Thatcher"-like symbols. Program SINT is the coder used in the results previously reported [Isal and Moffat, 2001a].

As can be seen, the use of CINT-A yields a very slight gain in compression effectiveness compared to SINT.

Figure 6 shows how compression efficiency and effectiveness are related during the coding phase. The arithmetic coders are slower than the minimum-redundancy coder, but give better compression. The structured and conditioning coders are the same speed as the uniform coder – they require a little more memory space during execution, but use of that space does not slow them down.

Table 4 summarises the cost of encoding file wsj20 using the four transformations necessary to a word-based BWT implementation. None of our programs are especially well tuned, and there is probably scope for speed improvements. On the other hand, it is unlikely that we would be able to match the speed of BZIP2, which employs an elegant BWT implementation [Seward, 2000] and a minimum-redundancy coder. Note that we deliberately do not compare our final compression effectiveness with

Transformation	Time (CPU seconds)	
	Encoding	Decoding
Word parsing	11.6	4.6
BWT	18.0	2.8
SSEG-H	50.3	41.0
CINT	6.8	6.7
Total	86.7	55.1

Table 4: Overall cost of forwards and inverse transformations on file wsj20. All times are CPU seconds on a 650 MHz Pentium III with 256 MB RAM and 256 kB on-die cache. As a benchmark, on the same file the highly tuned [Seward, 2000, 2001] character-based BWT implementation BZIP2 requires 25.0 seconds for encoding and 7.5 seconds for decoding.

BZIP2 – the comparison would be unfair, as BZIP2 operates on blocks of 900 kB at a time and within a total memory allocation of around 5 MB, whereas our process uses more than 30 MB during its BWT stage. A fairer comparison is with the PPMD mechanism (Witten et al. [1999, page 61]), which, when constrained to execute in 32 MB and allowed to build a fifth-order model, attained 1.58 bits per character. On the small file asyoulik.txt BZIP2 attains 2.52 bits per character, and the fifth-order PPMD attains 2.49 bits per character, compared with the best result obtained in this paper of 2.48 bits per character.

5 Conclusion

We have experimented with a number of variations on the four-phase word-based compression mechanism described by Isal and Moffat [2001a,b]. Compared to the previous results, we have described further approximate ranking heuristics that provide improved compression effectiveness in reduced time; and have also slightly improved the effectiveness of the coding stage. Overall, we have improved our previous best result of 1.58 bits per character for file wsj20 [Isal and Moffat, 2001a] by approximately 2%, and can now report compression of 1.55 bits per character, with similar gains also being achieved for the other four test files. In the grand scheme of things these improvements are admittedly small; nevertheless they reflect a better understanding of the processes involved, and represent the attainment of what to a sportsperson would certainly be valued as a "personal best".

We still have a number of aspects of this project to explore. One area not yet considered is the use of heuristics based upon a single Splay tree. The promotion mechanism

need not assign to the current symbol the next timestamp, and can employ any re-weighting it likes when re-inserting the accessed symbol. It may thus be possible to implement an SSEG-H-like strategy for promotion, but with the speed of the current MTF process (see Figure 3).

Another obvious exploration path is to combine the logarithmic selector used for the coding and the logarithmic tree index used in ranking into a single purpose. That is, the ranking process should be organised so that an input stream of symbol numbers is converted to an output stream of "(tree number, rank within tree)" tuples that are coded by a tailored coder. The adaptive conditioning would then apply to the tree number stream, while the "rank within tree" component would be coded in a single set of states, one for each tree number. Such an arrangement would almost certainly execute faster, even if there was no compression gain.

Acknowledgement This work was supported by the Australian Research Council and by the QUE Project of the Fasilkom, Universitas Indonesia.

References

B. Balkenhol, S. Kurtz, and Y. M. Shtarkov. Modifications of the Burrows and Wheeler data compression algorithm. In Storer and Cohn [1999], pages 188–197.

J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.

B. Chapin. Switching between two on-line list update algorithms for higher compression of Burrows-Wheeler transformed data. In Storer and Cohn [2000], pages 183–192.

E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

P. Fenwick. The Burrows-Wheeler transform for block sorting text compression: Principles and improvements. *The Computer Journal*, 39 (9):731–740, September 1996.

S. Grabowski. Text preprocessing for Burrows-Wheeler block sorting compression. In *VII Konferencja Sieci i Systemy Informatycne – Teoria, Projekty, Wdrozenia*, Lodzkiej, 1999.

R. Y. K. Isal and A. Moffat. Parsing strategies for BWT compression. In J. A. Storer and M. Cohn, editors, *Proc. 2001 IEEE Data Compression Conference*, pages 429–438. IEEE Computer Society Press, Los Alamitos, California, March 2001a.

R. Y. K. Isal and A. Moffat. Word-based block-sorting text compression. In M. Oudshoorn, editor, *Proc 24th Australian Computer Science Conference*, pages 92–99, Gold Coast, Australia, February 2001b. IEEE Computer Society, Los Alamitos, CA.

K. Sadakane. *Unifying Text Search and Compression: Suffix Sorting, Block Sorting and Suffix Arrays*. PhD thesis, The University of Tokyo, December 1999.

J. Seward. Bzip2 program and documentation, 1999. `http://sourceware.cygnus.com/bzip2/`.

J. Seward. On the performance of BWT sorting algorithms. In Storer and Cohn [2000], pages 173–182.

J. Seward. Space-time tradeoffs in the inverse B-W transform. In J. A. Storer and M. Cohn, editors, *Proc. 2001 IEEE Data Compression Conference*, pages 439–448. IEEE Computer Society Press, Los Alamitos, California, March 2001.

D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

J. A. Storer and M. Cohn, editors. *Proc. 1999 IEEE Data Compression Conference*, March 1999. IEEE Computer Society Press, Los Alamitos, California.

J. A. Storer and M. Cohn, editors. *Proc. 2000 IEEE Data Compression Conference*, March 2000. IEEE Computer Society Press, Los Alamitos, California.

A. I. Wirth and A. Moffat. Can we do without ranks in Burrows Wheeler transform compression? In J. A. Storer and M. Cohn, editors, *Proc. 2001 IEEE Data Compression Conference*, pages 419–428. IEEE Computer Society Press, Los Alamitos, California, March 2001.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, second edition, 1999.