Symbol Ranking Text Compression

Peter Fenwick

Technical Report 132
ISSN 1173-3500
6 June 1996

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand
peter-f@cs.auckland.ac.nz

Abstract In his work on the information content of English text in 1951, Shannon described a method of recoding the input text, a technique which has apparently lain dormant for the ensuing 45 years. Whereas traditional compressors exploit symbol frequencies and symbol contexts, Shannon's method adds the concept of "symbol ranking", as in 'the next symbol is the one 3rd most likely in the present context'. This report describes an implementation of his method and shows that it forms the basis of a good text compressor.[1] The recent "acb" compressor of Buynovsky is shown to belong to the general class of symbol ranking compressors.

Keywords text compression, Shannon, symbol ranking

[1] This report has been submitted as a paper to the *Journal of Universal Computer Science.* It is available by anonymous ftp from `ftp.cs.auckland.ac.nz /out/peter-f/TechRep132`

1. Introduction

In 1951 C.E. Shannon published his classic paper on the information content of English text, establishing the well-known bounds of 0.6 – 1.3 bits per letter [Shannon 51]. What is perhaps less recognised is the method by which he obtained those results, and it is that which is used here as the basis of a text compressor.

Shannon actually describes two methods. In both of them a person is asked to predict letters of a passage of English text. (Some of the preceding text may be made available, but the text to be predicted must be unfamiliar to the subject.) Shannon also shows that the *responses* to the predictions are equivalent to the original text and that an "identical twin" (or its mathematical equivalent) could be used to recover the original input. In both cases the person effectively prepares a ranked list of the probable symbols, most probable first, and presents this list to the comparator (or examiner!)

1. In the first method, the person predicts the letter and is then told "correct", or is told the correct answer.
2. In the second method, the person must continue predicting until the correct answer is obtained. The output is effectively the position of the symbol in the list and the sequence of "NO" and the final "YES" responses is a unary-coded representation of that rank or position.

A third method is a hybrid of the two given by Shannon. After some small number of failures (typically 4 – 6) the response is the correct answer, rather than "NO". With some types of coding for the prediction values this may give a more compact code.

This algorithm is actually a transformation or recoding of the original text, with an output symbol for every input symbol. For his Method 2, Shannon gives the results reproduced in Table 1.

Guesses, or symbol ranking	1	2	3	4	5	> 5
Probability	79%	8%	3%	2%	2%	5%

Table 1. Shannon's original prediction statistics

The distribution is very highly skewed, being dominated by only one value. This implies a low symbol entropy, which in turn implies excellent compressibility .

The technique used by Shannon is an example of the little known method of "symbol ranking”. Statistical compressors usually rely on “symbol frequency”, to assign shorter codes to more frequent symbols, and “symbol contexts”, to restrict the choice of probable symbols and enhance the symbol frequency encoding. Symbol ranking simply takes the current context (or any other aid to compression) and, based on that, prepares a list of all possible symbols, ordered from most likely to least likely. The recoding of the symbol is its position in the ordered list. The sequence of operations is then *contexts* $\rightarrow$ *ranked-list* $\rightarrow$ *encoded output*. Because of its historical antecedents, the coding into the ranked list will be called a “Shannon coding”. This is not to be confused with the well known Shannon variable-length code, which may of course be used to finally encode the output.

“All that is needed” to implement a compressor is some algorithm which can produce a symbol list ranked according to the expected probability of occurrence, with a following statistical compressor.

2. The algorithm

Bloom has recently produced a family of compressors based on the one simple observation that the longest earlier context which matches the current context is an excellent predictor of the next symbol [Bloom 96]. (The meaning will become clearer later.) His compressors follow Shannon’s first method in that he flags a prediction as “correct” or “incorrect” and follows an incorrect flag by the correct symbol. His compressors differ in their manner of encoding the flags and of presenting the correct symbol, with some giving moderate compression at very high speeds and others giving exceedingly good compression (as good as any reported) albeit with slower performance.

His method is essentially one of statistical sampling. The most recent matching context acts as a randomly chosen context when the whole file is considered, although probably biased by recency or locality effects. Thus while his method does not guarantee to deliver the most probable symbol, it is quite likely to deliver it or, failing that, will deliver one of the other more probable symbols.

The algorithm presented here extends Bloom’s method to offer possible symbols in the approximate order of the probability of their occurring in the present context. Although based on probabilistic sampling it is completely deterministic and is equally applicable to both compression and decompression. The visible technique is exactly that of Shannon’s second method — symbols are offered as candidates in the order of their estimated likelihood and the number of unsuccessful offers is encoded. The algorithm proceeds in several different stages —

1. The preceding data is searched for the longest string matching the current context (the most-recently decoded symbols). The symbol immediately following this string is offered as the most probable candidate. So far this is precisely Bloom's algorithm. The search may be terminated as soon as strings match to some predetermined length, or *order*, or the search may proceed to an unlimited match length (unbounded order).
2. If the first offer is rejected, the search continues at the original order, looking for more matches which are *not* followed by the first-offered symbol. The first such following match is the second offered symbol. If that suggestion is rejected, the search continues along earlier contexts of the same order. As the search proceeds, symbols which have been rejected are added to the *exclusion list* as candidates which are known to be unacceptable.
3. When all available contexts have been searched at an order, the order is reduced by one and the search repeated over the whole of the preceding text, from most recent to oldest. Matches followed by an excluded symbol are ignored and any offered symbol is added to the exclusion list.
4. When the order has dropped to zero, the remaining alphabet is searched, again with exclusions. This copy of the alphabet is kept in a Move To Front list, rearranged according to all converted symbols to give some preference to the more recent symbols.

The algorithm is given here primarily as a "proof of concept" rather than a production-level compressor. It works well on smaller and more compressible files but is very slow and gives poor results on larger or less compressible files where there is more text to search and more samples are likely to be rejected.

The decompression algorithm is the obvious converse. An initial statistical decoder recovers the sequence of symbol ranks and the symbol prediction mechanism is then called, rejecting as many estimates as indicated by the rank.

2.1. Comparison with other compression methods

There is little prior work on compressors which rank symbols in order of likelihood. The first example is the Move-To-Front compressor of [Bentley et al 86], but that uses words as its fundamental coding unit. A preliminary discussion of a symbol-ranking compressor is found in [Fenwick 95a], where it is described as a "PPMδ" compressor.

A more important example is the "block sorting" technique described recently by [Burrows and Wheeler 94], and extended by [Wheeler 95] and [Fenwick 96a, Fenwick 96b]. It uses a context dependent permutation of the input text to bring together similar contexts and therefore the relatively few symbols which appear in each of those contexts. A Move-To-Front transformation then ranks the symbols according to their recency of occurrence. Overall, the result is very similar to what is obtained here, except that the input is permuted in block sorting, whereas here it is processed in its natural order.

[Fenwick 96b] shows that block sorting is a symbol-ranking compressor, with the Move-To-Front list acting as a good estimate of symbol ranking. In collecting together similar contexts, the preceding sort phase also brings together the symbol rankings of those contexts; because the contexts are similar so are the ranking lists and the list for one symbol is usually a good prediction of that for the next symbol.

More importantly, the initial transformations of the block-sorting compressor and the symbol recoding of the new method both produce highly skewed symbol distributions. Methods which were developed for the efficient coding of the block sorting compressors are applicable to the new method as well. It will be seen that the frequency distributions of the recoded symbols are very similar for the two cases.

In comparison with PPM, the other major family of context-dependent compressors, the major differences are that here there is no attempt to assign probabilities to explicit symbols in each context and no need to use an escape symbol to move between context orders.

3. Implementation

The symbol ranking algorithm is implemented largely using techniques derived from LZ-77 parsing and with a fast string-matcher similar to that devised by Gutmann [Fenwick 95b]. Before describing that however, it is necessary to discuss the efficient implementation of exclusion.

3.1 Exclusion.

Exclusion is one of the techniques developed to improve PPM compression and consists of ignoring any symbol which has been already considered at a higher order of the present context. Exclusion is equally important in a ranking compressor because any symbol which has been rejected for the current position should not be offered again.

An obvious implementation of exclusion is to build a table of excluded symbols and to search the list for each possible candidate. (The exclusion list is set to empty when moving to a new context.) This implementation may be quite inefficient when considering less probable symbols, with high rank numbers. Every possible symbol must be tested against a list of reasonable length, and most symbols will be rejected.

The chosen implementation inverts the problem. Instead of asking “Is this one of the list of symbols excluded from the present context?”, we ask “When was this symbol last excluded?” A context is easily and uniquely identified by the sequential index of its last symbol. We maintain an array indexed by the candidate symbol and whenever a symbol is excluded place its context index in `Array[symbol]`. Testing a symbol for exclusion involves just accessing its array element and seeing whether that corresponds to the current context. When the current symbol is completely processed and the compressor advances, the context identification changes and the old exclusions are immediately forgotten without clearing or modifying the table.

3.2 Finding the initial offering

The context discovery mechanism uses a technique derived from a fast LZ-77 type of string comparison. The known text is saved in a wrap-round LZ-77 buffer, with pairs of “similar” digraphs linked as a list to facilitate fast traversal; the lists are accessed via a hash table on the last two symbols. The mechanism is shown in Figure 2.

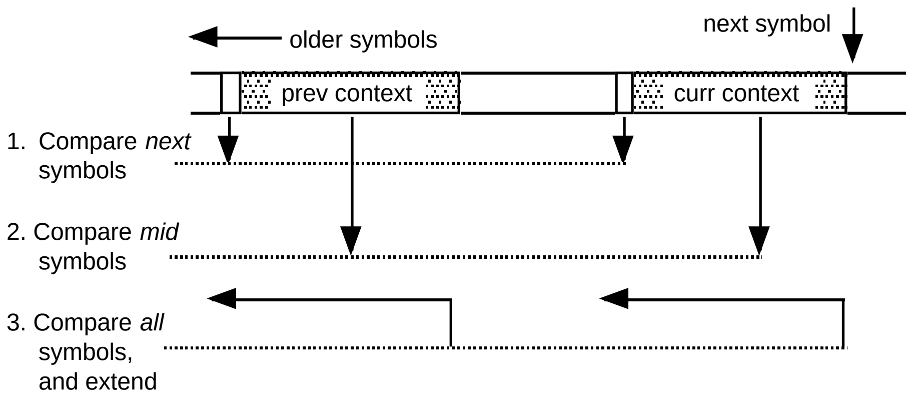

Figure 2. The Gutmann LZ-77 context scan.

Assume at some stage that we know a context which matches “curr context” to some length and have linked to the next possible context (“prev context”).

- The rightmost (most recent) symbols of the two contexts probably match because they share the same hash value.
- As we wish to *extend* the match to find a longer context we first test the symbol beyond the known best order. If this differs the context cannot possibly extend. Choosing this symbol also uses one which is less correlated with the most recent symbols.
- Having confirmed that the extension symbols match, we compare two symbols near the midpoint of the two contexts, as a further quick filter on the contexts. On some files it is better to compare low probability symbols at about the midpoint, but it is usually simpler to use the actual midpoint.
- Finally we do a complete string comparison, from the most recent symbols, for as long as the contexts match. If this extends the contexts the test context becomes the best-known and its order is saved as the best order.

Experience is that 50% of the possible contexts are eliminated by the first, extension, test and that fewer than 10% survive the midpoint comparison and need a full string compare.

From this string comparison algorithm we find the longest preceding context which matches the most recent context; the symbol following that context is offered as the first choice. We also enter this symbol in the exclusion table in case a longer search is needed.

3.3 Continuing within the order and to lower orders

The algorithm for continuing the search is similar in spirit but different in detail. The search is now at a known order, so the initial test is on the end-symbol of the context, not on the one beyond the end, and the comparison never extends outside the limits defined by the current order. At an early stage of the tests, and certainly before the complete string comparison, the following symbol is tested against the exclusion table. The next symbol with a matching context and which is not excluded is offered as the next candidate symbol.

When the oldest available context has been tested at a particular order, the order is reduced by one and the search repeated from newest to oldest. At no stage is the order actually released to the coding mechanism. All that is released is the possible symbols, in the expected order of likelihood.

3.4 Handling order–1 contexts

With two symbols used in forming the hash value, the above methods do not work for order–1 contexts. These are handled by a completely different mechanism. For each context (or most recent symbol) there is a series of links along the buffer, linking occurrences of that symbol, but bypassing occurrences where the following symbol has been seen already in that list.

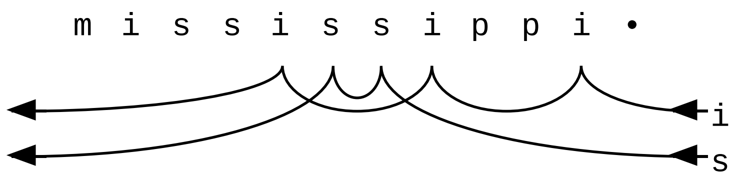

Figure 3. Illustration of order-1 context handling

For tracing an order–1 context the appropriate list is selected according to the context symbol and its list traversed. The next symbol which is not excluded is offered as the next candidate (the list structure automatically maintains order–1 exclusions.) The technique is illustrated in Figure 3.

Two lists are shown, for the context symbols “i” and “s”. (The lists for “p” and “m” add nothing to the explanation.) In order from the rightmost (most recent) symbol, the “i” list links occurrences followed by, in order “•”, “p” and “s”. The leftmost “i” of the string is followed by an “s”. This symbol has already been seen and is therefore bypassed. The “s” list similarly bypasses the leftmost two “s”s, both of whose following symbols have been already seen.

Whenever a new context is created the list is traced to find the occurrence of its following symbol. The list is then altered to bypass that older occurrence because it has been superseded by the more recent occurrence. This implements exclusion within the list and also limits the length to be searched.

4. Compression Results

Results for the compressor are shown in Table 2, tested on the Calgary compression corpus [2] . All of these results include two techniques developed for block sorting compression.

- The input text is subject to run compression, with runs of length 6 or greater being replaced by the initial 6 symbols and then a length count coded into following symbols. This is intended primarily to increase the speed of encoding files such as PIC, which have many long runs and compress very slowly. A threshold of 6 means that most files

[2] 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`

are not affected. This run compression does improve the compression of PIC by about 10%.

- The output of the Shannon coder has a preponderance of 0s (first prediction correct) and most of these occur in runs. These are run-length encoded using a method due to Wheeler [Wheeler 95, Fenwick 96b] to give a small improvement in the compression of all files.

The block sorting compressor is used as a comparison because of its close relationship to the new Shannon compressor. Both are "symbol ranking" compressors, but block sorting permutes the input file whereas the Shannon compressor works on the file in natural order.

The rightmost columns are for various versions of the new "Shannon" compressor.

1. **64 K buffer, Order-0, maxorder=20.** The "Ziv-Lempel" type buffer is 64 Kbytes long (the most recent 65,536 bytes enter into context determination), the output is encoded with a simple order-0 arithmetic coder, and the maximum context order is 20. This is intended as a reference version of the new Shannon compressor.
2. **1024 K buffer, Order-0, maxorder=20.** The buffer size is now adequate to hold the largest files and allows contexts from the whole file rather than just the most recent 64 K bytes. As compared with the previous column, the improvement is up to 8% for the

File	Block Sort, Order-0	Block Sort, structured	64 K buffer Order-0 maxord=20	1024 K buffer Order-0 maxord=20	64 K buffer structured maxord=20	1024 K buffer structured maxord=20	64 K buffer structured maxord=10	Buynovsky ACB
BIB	2.31	1.95	2.31	2.26	2.27	2.22	2.29	1.95
BOOK1	2.52	2.39	3.06	2.85	3.03	2.82	3.03	2.34
BOOK2	2.20	2.04	2.52	2.35	2.48	2.32	2.49	1.96
GEO	4.81	4.50	5.72	5.71	5.51	5.49	5.51	4.69
NEWS	2.68	2.50	2.89	2.66	2.84	2.62	2.86	2.34
OBJ1	4.23	3.87	3.94	3.94	3.79	3.79	3.81	3.54
OBJ2	2.71	2.46	2.63	2.50	2.55	2.43	2.57	2.24
PAPER1	2.61	2.46	2.63	2.63	2.59	2.59	2.60	2.37
PAPER2	2.57	2.41	2.72	2.71	2.69	2.68	2.69	2.37
PIC	0.92	0.77	0.84	0.83	0.84	0.82	0.82	0.75
PROGC	2.67	2.49	2.60	2.60	2.55	2.55	2.55	2.35
PROGL	1.84	1.72	1.74	1.74	1.70	1.70	1.73	1.53
PROGP	1.82	1.70	1.73	1.73	1.69	1.69	1.75	1.52
TRANS	1.60	1.50	1.54	1.51	1.50	1.48	1.54	1.31
Average	**2.53**	**2.34**	**2.63**	**2.57**	**2.57**	**2.51**	**2.59**	**2.23**

Table 2. Results in compressing Calgary Corpus

larger files. (Many of the smaller files fit, completely or nearly, into the smaller buffer and show little or no benefit from the larger buffer. The initial run-encoding of PIC transforms it into a file of little more than 100 Kbyte and it behaves here as medium-sized file.)

3. **64 K buffer, structured, maxorder=20.** The coder has been replaced by one which was developed for the block-sorting compressor and is better at handling very skew symbol distributions [Fenwick 96b].
4. **1024 K buffer, structured, maxorder=20.** This case combines the benefits of the larger file for more context information and the improved final coder.
5. **ACB** This is Buynovsky's ACB compressor, among the best compressors at present, working at maximum compression. (It is discussed in Section 8.)

A test with a 64 K buffer, structured coder, and *maxorder*=10 gave somewhat somewhat faster operation, but slightly poorer compression than the similar one with maximum *order* = 20 (2.59 bit/byte).

The results are generally similar to those with block sorting, which is of course to be expected from the similarity of the two methods. In general the new compressor seems to be better on the object files and the more compressible text files, while block sorting is better on most text files and much better on GEO.

The difference probably arises from the sampling symbol predictor in the Shannon compressor. While it works well for compressible files where there is little doubt as to the correct symbol, for less compressible files there tends to be a much greater selection of "reasonable" symbols and a correspondingly greater chance of making a poor prediction.

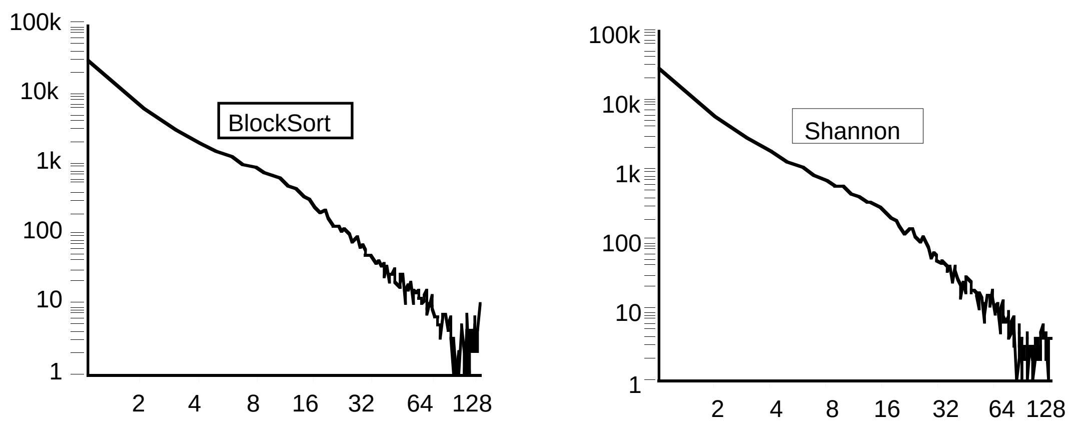

Figure 4. Frequencies of different code ranks for block sorting and Shannon compression.

5. Block sorting and symbol ranking compared.

While block sorting and symbol ranking compression may be regarded as generally equivalent, they do give slightly different results. Figure 4 shows the frequencies of the symbol ranks for the file PAPER1 using the two methods. At this scale the two are essentially identical, except for minor differences for ranks beyond about 16 where the symbol probabilities are quite low.

Rank	0	1	2	3	4	5	6	7	8	9
BlockSort	58.3%	11.3%	5.5%	3.7%	2.8%	2.3%	1.8%	1.6%	1.4%	1.3%
Shannon	58.9%	11.6%	5.8%	3.8%	2.7%	2.0%	1.6%	1.4%	1.2%	1.1%

Table 2. Frequencies of different code ranks for block sorting and Shannon compression.

More detail can be seen in Table 2 which shows the relative frequencies of symbol ranks for the two compressors, for the relative frequencies greater than about 1%. The more-probable values (ranks 0, 1 and 2) are slightly more frequent with Shannon compression; this lowers the skewness of the distribution and degrades the compression slightly.

Another effect which is not visible from these results comes from the locality effects of block sorting. Block sorting collects together similar contexts and emits the symbols from those contexts as a group. A suitable final coder (such as is used here) can adapt to the local statistics of the contexts. The Shannon compressor by comparison must switch between quite different contexts for successive symbols and cannot adapt to any of them. This is probably the major reason for the difference in the results. It should be possible to improve the performance of the Shannon compressor by using final coders which are sensitive to the current context. This is being investigated.

6. The prediction process

Some output from an actual encoding of a version of this paper is shown in Figure 5. There is a 2-symbol overlap between the two sequences. The actual text is written in bold face, with the output value just below it (this is the number of wrong estimates for the symbol). Above each symbol are the predictions for that symbol, with the first always at the top. The eighth and subsequent bad estimates are replaced by a single ⊗. Below the output code is the order at which that code is determined, often with an obvious relation to the preceding text.

What is not so easily conveyed is the way that the order changes during prediction. For example, in predicting the final “e” of “Shannon code”, the unsuccessful “i” is predicted at

	4				c							i							t									
	W				a							s							p								v	
					t							t							s								i	
	F				i							1							c								g	
	3				r							b	c						l								a	
	E				s							d	l						r								p	
	H				p							a	r						b								d	
	A				⊗							⊗	n						⊗	o						’	r	
text	**T**	**h**	**e**		**o**	**u**	**t**	**p**	**u**	**t**		**o**	**f**		**t**	**h**	**e**		**S**	**h**	**a**	**n**	**n**	**o**	**n**		**c**	**o**
output	8	0	0	0	17	0	0	0	0	0	0	10	4	0	0	0	0	0	50	1	0	0	0	0	0	1	7	0
order	2	3	4	5	4	5	6	7	8	9	10	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	8	9

	v												f											k				
	i												l									–	t	s				
	g				d		o						“									-	l	g				
	a				s		“						s			d						,	c	t				
	p				.		f						t			c	a	r			d	.	v	d			i	
	d						a				d	l	c			s	l	s		s	x	s	p	n			t	
	r			i	,	a	i				b	p	v			f	e	i			n	⊗	y	⊗		s	b	
text	**c**	**o**	**d**	**e**	**r**		**h**	**a**	**s**		**a**		**p**	**r**	**e**	**p**	**o**	**n**	**d**	**e**	**r**	**a**	**n**	**c**	**e**		**o**	**f**
output	7	0	0	1	5	1	5	0	0	0	2	2	7	0	0	4	3	3	0	2	3	9	6	12	0	1	3	0
order	8	9	10	4	3	3	4	5	6	7	5	4	4	5	6	4	3	2	3	2	2	3	2	2	3	4	4	5

Figure 5. Illustration of coder symbol prediction

order 11, but the next prediction (successful) is at order 4, with no external indication of the change in order. This ability to change order invisibly is believed to be a strength of this technique — no escapes are needed.

While there is often some difficulty in establishing a symbol, the correct text then often proceeds with no trouble for several symbols. A human predictor would get “of” with little difficulty, and should also get “Shannon” almost immediately from the overall theme. Again, the latter part of “preponderance” should be predictable (there is is no other reasonable word “prepon…”) The prediction is often almost eerily like that expected from a person, but with a limited vocabulary and largely ignorant of idiom! (This text, of about 21,000 bytes, compresses to 2.77 bit/byte.)

7. Final Comments

The algorithm is given here primarily to illustrate the principle rather than as a production-level compressor. Its technique is one of statistical sampling, with the overwhelming majority of candidate symbols being rejected, especially on large or less compressible files. This leads to very slow operation on those files. An implementation which holds contexts and their symbols in more-conventional data structures should improve both the compression speed and the accuracy of symbol prediction. It should also allow explicit symbol frequencies to be associated with each context, again improving compression. This work is proceeding.

8. Addendum—Buynovsky's "ACB" text compressor

Another recent compressor is the "acb" algorithm of [Buynovsky 94]. It gives excellent performance, the most recent versions of acb being among the best known text compressors. Explanations of the acb algorithm by Leo Broukhis and Charles Bloom have been combined with those of Buynovsky and then further developed below to describe the essence of the methods, but not the fine details.

The input text is processed sequentially; as compression proceeds we build a sorted dictionary of all encountered *contexts*. Following each context, from the input text, is its corresponding *content* string. Prefixes of these content strings are tested against to-be-processed characters to find matching phrases (as in LZ–77 compression). In practise the dictionary may be an appropriately ordered table of pointers back into the input text; going back from a pointer gives the context and going forward gives its content.

To process a character we first find its best matching context, using this context index as an *anchor* into the context dictionary. (The actual position in the input text is irrelevant.) Having found the anchor, we search neighbouring (similar) contexts for the longest matching phrase from the contents. If this best phrase has length λ and occurs a distance δ contexts from the anchor (δ may be negative), the phrase is represented by the couple $\{\delta, \lambda\}$. The final coding of $\{\delta, \lambda\}$ is rather more subtle and largely determines the final quality of the compression. As an example of one such optimisation the chosen phrase will probably match some earlier phrase (closer to the anchor) to some length μ and we encode not λ but $(\lambda - \mu)$. Another alternative is given in the next paragraph. The most recent portion of the context may be used as a conditioning class for the final coding models.

In some ways the operation is best envisaged using the sorted contexts from the block sorting context algorithm; in other ways it is related to LZ-77 (δ corresponds to a displacement and the use of contexts reduces the range of δ and thereby improves the coding), or it resembles LZ-78, regarding the neighbouring contexts as a dictionary and emitting indices into that dictionary. Here we prefer to interpret the method as a derivative of a symbol ranking compressor. The anchor and its content provides the best estimate of the phrase to be emitted;

the distance of the correct phrase from the anchor is a measure of its ranking with respect to the anchor context. We then have not a *symbol-ranking* compressor, but a *phrase-ranking* compressor. The principle follows immediately from Shannon's technique if we allow the predictions to be phrases rather than single symbols.

An implementation follows from the compressor of this paper. Accompanying its input "LZ buffer" are pointers which allow us to build an ordered list of contexts. As each character is processed it is linked into this list, probably by an insertion sort, to maintain the correctly ordered context table. The best matching context is found by Bloom's method and forms the anchor. Searching the list around the anchor context allows us to find the best matching content and the $\{\delta, \lambda\}$ couple. In rejecting phrases as we move away from the anchor it is quite possible that we will encounter some which are identical to ones already rejected. These repeated phrases should be omitted from the count, giving a technique of *phrase exclusion* (analogous to the more usual *symbol exclusion*). Phrase exclusion requires that we emit, not $\{\delta, \lambda\}$, but $\{\varepsilon, \lambda\}$ where ε is the number of unique phrases of length λ encountered between the anchor and the content with the best matching phrase.

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 Western Australia. The author acknowledges the contributions of these institutions. Thanks are also due to Prof. David Wheeler, whose work on the block sorting compression algorithm led the author into the present work, and to Charles Bloom whose LZP compressors provided the direct inspiration. George Buynovsky, Leo Broukhis and Stig Valentini provided essential information for Section 8.

References

[Bloom96] C. Bloom, "LZP: a new data compression algorithm", *Data Compression Conference, DCC'96*

[Bentley etal 86] 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

[Burrows, Wheeler 94] 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`

[Buynovsky 94] G. Buynovsky, "Associativnoe Kodirovanie", ("Associative Coding", in Russian), "Monitor", Moscow, No 8, 1994, pp. 10–19.

[Fenwick 95a] P.M. Fenwick, "Experiments with a Block Sorting Text Compression Algorithm", The

University of Auckland, Department of Computer Science, Technical Report 111, May 1996. `ftp.cs.auckland.ac.nz /out/peter-f/report111.ps`

[Fenwick95b] P.M. Fenwick, "Differential Ziv-Lempel Text Compression", *J.UCS* Vol 1, No 8 pp 587–598 Aug 1995

[Fenwick96a] 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`

[Fenwick 96b] P.M. Fenwick, "Block-Sorting Text Compression — Final Report", The University of Auckland, Department of Computer Science, Technical Report 130, March 1996. `ftp.cs.auckland.ac.nz /out/peter-f/report130.ps`

[Shannon 51] C.E. Shannon, "Prediction and Entropy of Printed English", *Bell System Technical Journal*, Vol 30, pp 50–64, Jan 1951

[Wheeler 95] 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`]