Searching BWT compressed text with the Boyer-Moore algorithm and binary search

Tim Bell[1] Matt Powell[1] Amar Mukherjee[2] Don Adjeroh[3]

November 2001

Abstract: This paper explores two techniques for on-line exact pattern matching in files that have been compressed using the Burrows-Wheeler transform. We investigate two approaches. The first is an application of the Boyer-Moore algorithm (Boyer & Moore 1977) to a transformed string. The second approach is based on the observation that the transform effectively contains a sorted list of all substrings of the original text, which can be exploited for very rapid searching using a variant of binary search. Both methods are faster than a decompress-and-search approach for small numbers of queries, and binary search is much faster even for large numbers of queries.

1 Introduction

This paper investigates on-line exact pattern matching in files compressed with the Burrows-Wheeler transform (Burrows & Wheeler 1994). By 'on-line' pattern matching, we refer to methods that do not require a pre-computed index—all the work of pattern matching is done at query time. They are particularly suitable for texts that are not likely to be searched frequently, where the overhead of constructing an index (for an 'off-line' method) is not justified.

On-line searching in the *uncompressed* domain requires a scan of the entire input file. The simplest 'brute force' algorithm, a linear search which aligns the pattern at each position in the text and checks for a match, has time complexity $\mathcal{O}(mn)$, where m is the length of the pattern and n the length of the text. Other, more sophisticated algorithms such as the Knuth-Morris-Pratt algorithm (Knuth, Morris & Pratt 1977) and the Boyer-Moore algorithm (Boyer & Moore 1977), are able to identify portions of the text that cannot possibly contain a match, and can consequently be 'skipped'. In this way, the Knuth-Morris-Pratt algorithm achieves a worst case time complexity of $\mathcal{O}(m+n)$, and the Boyer-Moore algorithm, although in the worst case it degenerates to the linear algorithm, has a best case time complexity of $\mathcal{O}(\frac{n}{m})$. In general, the Boyer-Moore algorithm manages to achieve so-called 'sub-linear' complexity; that is, for most files, it solves the pattern matching problem in fewer than n comparisons.

If the file to be searched is already compressed, the above algorithms can be used in a 'decompress-then-search' approach, where the compressed file is first decompressed. This

[1]Department of Computer Science, University of Canterbury, New Zealand; {tim,mjp86}@cosc.canterbury.ac.nz

[2]School of Electrical Engineering and Computer Science, University of Central Florida, USA; amar@cs.ucf.edu (Partially supported by a grant from the National Science Foundation: IIS-9977336)

[3]Lane Department of Computer Science and Electrical Engineering, West Virginia University, USA; adjeroh@csee.wvu.edu

	F		L
1	i	mississip	p
2	i	ppimissis	s
3	i	ssippimis	s
4	i	ssissippi	m
5	m	ississipp	i
6	p	imississi	p
7	p	pimississ	i
8	s	ippimissi	s
9	s	issippimi	s
10	s	sippimiss	i
11	s	sissippim	i

(a) Sorted matrix

	L	F	
1	p	i	
2	s	i	ppi
3	s	i	ssippi
4	m	i	ssissippi
5	i	m	ississippi
6	p	p	i
7	i	p	pi
8	s	s	ippi
9	s	s	issippi
10	i	s	sippi
11	i	s	sissippi

(b) Sorted list of substrings

Figure 1: Burrows-Wheeler transform of the string `mississippi`

has significant overheads, in terms of both computation time and storage requirements. *Compressed-domain* pattern matching has the advantage of avoiding the time required to decompress, and is likely to be dealing with a smaller file. A recent survey by Bell, Adjeroh & Mukherjee (2001) outlines several techniques for online compressed-domain pattern matching in both text and images. Many of these techniques are based on the LZ family of compression systems (Ziv & Lempel 1977, Ziv & Lempel 1978), but others include methods for Huffman-coded text and run-length encoding. Little work has been done with the Burrows-Wheeler transform, although some research has been undertaken in the area of *offline* pattern matching (Ferragina & Manzini 2000, Ferragina & Manzini 2001, Sadakane & Imai 1999, Sadakane 2000).

Throughout this paper we will refer to the pattern matching problem in terms of searching for a pattern P of length m in a text T of length n. The input alphabet will be referred to as Σ; similarly, $|\Sigma|$ will denote the size of the alphabet. All arrays are 1-based unless otherwise noted.

2 The Burrows-Wheeler transform

The Burrows-Wheeler transform, also called 'block-sorting' (Burrows & Wheeler 1994) sorts the characters in a block of text according to a lexical ordering of their following context. This process can be thought of in terms of sorting a matrix containing all cyclic rotations of the text. Figure 1(a) shows such a matrix, constructed for the input string `mississippi`. Each row is one of the eleven rotations of the input, and the rows have been sorted lexically. The first column (F) of this matrix contains the first characters of the contexts, and the last column (L) contains the permuted characters that form the output of the BWT. It is also necessary to transmit the position of the original string in the sorted matrix (row 5 in Figure 1(a)); therefore, the complete BWT output for the string `mississippi` is the pair $\{$`pssmipissii`, 5$\}$. Because only a few characters are likely to appear in any given context, the permuted text is ideal for coding with a move-to-front transform (Bentley, Sleator, Tarjan & Wei 1986), which assigns shorter codes to more recently-seen symbols.

Given the seemingly random nature of the permuted text, it may seem almost impossible to recover the original text with no other information, but in fact the reverse transformation can be performed relatively easily given the permuted string L, the index of the first character, and the sorted matrix F, which can be recovered from the permuted text in $\mathcal{O}(n \log |\Sigma|)$ time. This is due to two key observations: firstly, that the sorted matrix is constructed using cyclic rotations, so each character in L is immediately followed by the corresponding character in F, and secondly, that the instances of each distinct character appear in the same order in both arrays—for example, the third occurrence of the letter `i` in L corresponds to the third `i` in F, and so on. In the example, we know that the first character is the `m` at position 5 of F, and that this must correspond to the `m` at position 4 of L, since there is only one `m` in the text. The next character is the i at position 4 of F, since characters in F immediately follow the characters at the same index in L. This is the fourth `i` in F, and so it corresponds to the fourth `i` in L (at position 11), which in turn must be followed by an `s`. Continuing in this way, it is possible to decode the whole string in $\mathcal{O}(n \log |\Sigma|)$ time.

If it is not necessary to decode the entire string at once, a transform array W can be computed in linear time with a single pass through the L and F arrays, such that

$$\forall i : 1 \leq n, T[i] = L[W^i[id]]$$

where $W^0[x] = x$, $W^{i+1}[x] = W[W^i[x]]$, and id is the index of the original text in the sorted matrix. By traversing L with W, we have a means of decoding arbitrary substrings of the text.

3 A Boyer-Moore-based approach

The Boyer-Moore algorithm (Boyer & Moore 1977) is considered one of the most efficient algorithms for general pattern matching applications. It is able to recognise and skip certain areas in the text where no match would be possible.

The pattern is shifted from left to right across the text, as in brute-force pattern matching, but comparison is performed from right to left on the pattern. As soon as a mismatch is detected, the pattern is shifted to the right according to one of two key heuristics: the **extended bad character rule** and the **good suffix rule**.

To illustrate the operation of these heuristics, suppose that the pattern, P, is aligned at position k of T, and that a mismatch has been detected between the character at position i of the pattern—that is, $P[i] \neq T[k+i-1]$. Then let $c = T[k+i-1]$, the mismatched character of the text, and $t = P[i+1 \ldots m]$, the suffix of the pattern which matches the corresponding portion of the text. The **extended bad character rule** proposes that if there is an occurrence of c in P to the left of i, that the pattern be shifted so that the two occurrences of c are aligned. If no such shift is possible, the pattern is shifted completely past the c in the text. The **good suffix rule** attempts to align the matched suffix, t, with a previous occurrence of t in the pattern (for example, in the pattern `reduced`, the suffix `ed` occurs twice). If there are no other occurrences of t in the pattern, then the pattern is either shifted so that the a prefix of the pattern matches a suffix of t in the text, or, if this is not possible, shifted completely past t.

The Boyer-Moore algorithm checks both of these heuristics at each stage of the matching process; if both shifts are possible, then the maximum is chosen. In this way, Boyer-Moore achieves so-called 'sub-linear' performance for most texts.

To apply the Boyer-Moore algorithm to the permuted output of the Burrows-Wheeler transform, we define an array Hr to relate the characters in T with their position in F, such that

$$\forall i : 1 \leq i \leq n, T[i] = F[Hr[i]]$$

The Hr array can be computed efficiently with the following procedure:

```
COMPUTE-HR-ARRAY($W$, $id$)
1  $i \leftarrow id$
2  for $j \leftarrow 1$ to $n$ do
3      $i \leftarrow W[i]$
4      $Hr[j] \leftarrow i$
5  end for
```

where id is the index of the first character in L, obtained from the BWT output.

The Hr array introduces the possibility of accessing random characters in the permuted string. Using this technique, we have implemented a Boyer-Moore algorithm for BWT by adapting a standard Boyer-Moore routine to access character i at $F[Hr[i]]$ instead of at $T[i]$. The asymptotic complexity is the same for this approach as for Boyer-Moore on uncompressed text, although in practice it is a little slower, due to the time taken to construct Hr, and an extra dereference each time a character is needed. Details of the performance of this method are given in Section 5.

4 Binary searching

A useful side-effect of the Burrows-Wheeler transform is that it produces, as an artifact of the inverse transform, a list of all the substrings of the text in sorted order, making it possible to perform a binary search on any file encoded with the Burrows-Wheeler transform—an $\mathcal{O}(m \log n)$ pattern matching algorithm!

Figure 1(b) shows the sorted list of substrings for the text `mississippi`, generated by the BWT process. (The substrings start with the characters in the F column; the L array is provided for reference only.) Notice that all the occurrences of the substring `is` are together in the list (positions 3–4). This is an important observation in the application of binary search to BWT-compressed files.

In fact, it is possible to improve even on the above $\mathcal{O}(m \log n)$ figure, and obtain a pattern matching algorithm that runs in $\mathcal{O}(m \log \frac{n}{|\Sigma|})$ time, if the F array of the Burrows-Wheeler transform is not stored explicitly, but the starting position and length of each run of characters in F are stored in two arrays of size $\mathcal{O}(|\Sigma|)$, as is often the case in efficient implementations of the BWT. Therefore, if the first character of the pattern is c, then the lower and upper bounds for a binary search are available with two array lookups.

The enhanced binary search algorithm obtains all the matches with a fraction of the number of comparisons needed by linear-based algorithms such as Boyer-Moore. In fact, in situations where there are multiple matches present in the text, some matches will be found without any comparisons at all. This is possible because once two matches have been found, it is known that everything between them must also be a match, due to the sorted nature of the F array.

This extremely fast performance makes the binary search approach ideal for situations where speed is of the essence. One example is the 'incremental search' found in multimedia encyclopædias and online application help. As the user enters a word character by character, a list of possible results is displayed. This list becomes shorter as more of the word is typed and the query made more specific. For example, a search for the word *compression* in a dictionary would start with all the results matching *c*, then *co*, then *com*, and so on. By the time the prefix *compr* has been typed, the number of possible results is probably small enough to fit on one screen.

Saving the start and end indices of the previous searches allows an improvement in the performance of the binary search, since these indices can be used as the bounds for the next search.

5 Experimental results

For the purposes of experimentation, a simple Burrows-Wheeler-based compression program, bsmp, was developed. bsmp uses a four-stage compression system:

1. a Burrows-Wheeler transform, with the block size set to the size of the entire file,
2. a move-to-front coder (Bentley et al. 1986), which takes advantage of the high level of local repetition in the BWT output,
3. a run-length coder, to remove the long sequences of zeroes in the MTF output, and
4. an order-0 arithmetic coder.

Naturally, bsmp does not compare particularly favourably in terms of speed, since no real effort was spent in optimising this simple system, but the compression ratios achieved by bsmp are at least comparable to bzip2's. For the remainder of the experiments in this chapter, bsmp will be used as the compression system, since this allows us to work with partially decompressed files, and to treat the entire file as one block. It is intended to examine in more detail the relationship between the BWT and the compression process, and eventually to adapt the techniques in this report for use with bzip2-compressed files.

There are a number of interesting choices to be made when implementing a block-sorting compressor such as bsmp. Fenwick (1996) examines these choices in some detail, but his compression system does not use run-length coding (step 3 above). In implementing bsmp, we have chosen to use two models for the arithmetic coder. The first models the lengths of runs in the MTF output, and the second the MTF values themselves. By always outputting a run length, we effectively compose a flag denoting whether this is a run or not with the actual run length, since there will be approximately a 90% chance that the run length will be one[4]. In a system based on Huffman coding, this would be wasteful, but an arithmetic coder allows a representation that is arbitrarily close to the entropy (Witten, Moffatt & Bell 1999), and so a probablility of 90% can be coded in $-\log_2 0.9 = 0.152$ bits.

There is some room for fine-tuning, but we are mainly concerned here with relative speed, not compression performance.

The main result we are interested in is, of course, whether or not searching is actually faster with the Burrows-Wheeler transform. Assuming that we have a file that has been compressed as described above, we could either completely decompress the file and then use a tool like grep to do the pattern matching, or only partially decompress the file (to the BWT-permuted output) and use one of the BWT search algorithms described in this report.

In considering which method to use, we must take into account the overheads associated with each stage of the decompression process. For example, if the inverse BWT phase is particularly costly, it may be better to search on the BWT-encoded file. However, if the time taken to decompress the arithmetic coding, run-length coding and move-to-front coding together far outweighs the cost of the inverse BWT and the search on the decoded text, or the search on the BWT file, then the choice of whether or not to perform the inverse BWT in full is largely arbitrary.

To investigate the individual parts of the decompression process, the bsmp decompressor was modified to produce as output a BWT file, which could then be fed into a BWT decoder if required. This allowed us to compare the relative performance of the BWT search algorithms with a decode-and-search approach.

Figure 2 shows a timeline view of the three search algorithms on the 3.86 megabyte `bible.txt` file from the Canterbury Corpus. The '`uncompress`' phase represents the time taken to decode the arithmetic coding, run-length coding and move-to-front coding, producing a BWT file. This file is then used for each of the three search methods:

- `unbwt` and bm apply the inverse BWT and uses a standard Boyer-Moore algorithm to search for the pattern. Rather than use a pre-built utility like grep for the pattern-matching stage, it was decided to build a pattern-matching program from scratch, for

[4]Based on empirical data.

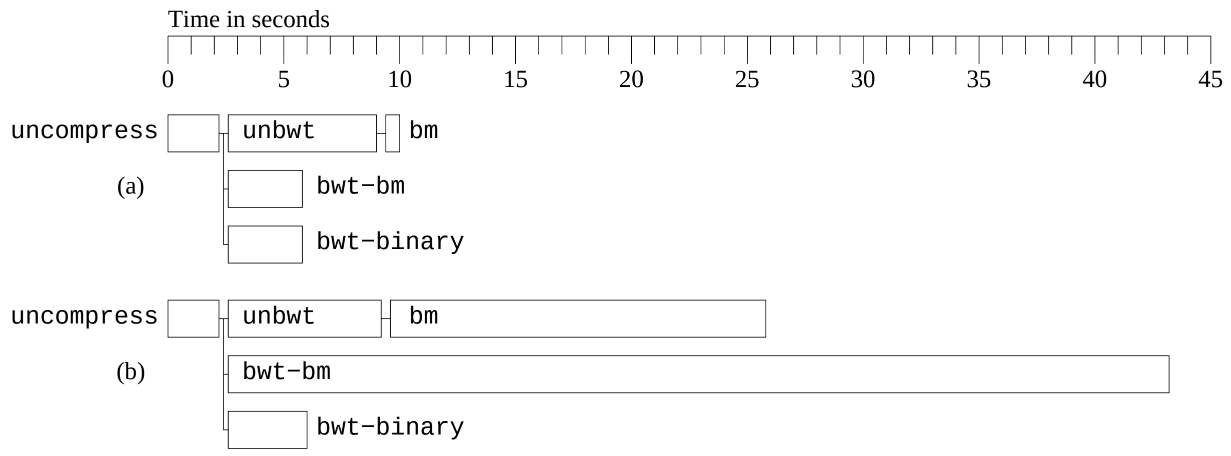

Figure 2: Search times for (a) a single pattern, and (b) a hundred randomly-selected words from the file `bible.txt`

consistency of implementation (`grep` is heavily optimised, and this may have produced biased results).

- `bwt-bm` is the compressed-domain Boyer-Moore algorithm, as described in Section 3.
- `bwt-binary` is the binary search algorithm defined in Chapter 4.

Figure 2(a) shows the time taken to search for a single pattern. In this case, the `unbwt` phase takes approximately twice as long as the `bwt-bm` and `bwt-binary` methods, which take approximately equal time. In this case, either of the two compressed-domain algorithms would be appropriate. The main overhead for these algorithms is the time taken to construct the various transform arrays from the BWT output, as the search time for a single pattern is almost trivial.

Figure 2(b) illustrates the behaviour of the different search algorithms when searching for more than one pattern. In this case, a hundred words were randomly selected from the file as candidate patterns, and the pattern matching algorithms searched for each one in turn. The compressed-domain Boyer Moore algorithm was dramatically slower than even the plain-text search. One possible reason for this is that although the two programs share almost identical code, the permuted nature of the BWT output means that a compressed-domain Boyer-Moore algorithm must perform several array lookups per comparison, whereas in the plain-text algorithm, these can be replaced with pointer arithmetic. It is interesting to note that there is almost no difference between searching for one pattern with the binary algorithm and searching for a hundred. This underscores the fact that most of the time taken for the binary search is in loading and processing the BWT file.

Above about twenty patterns it becomes more efficient to decode the BWT file and search on the plain text than to use the compressed-domain Boyer-Moore algorithm. However, both these methods are dramatically slower than the compressed-domain binary search, which is almost constant at around 5.9 seconds for any number of patterns.

		Comparisons	
m	**Linear**	**Boyer-Moore**	**Binary**
1	4047392	4047392	0
2	4241070	2182361	31
3	4224631	1490789	47
4	4214350	1150890	54
5	4217160	944326	57
6	4209334	814137	59
7	4207811	719478	64
8	4213156	643483	68
9	4213079	584208	72
10	4206945	538273	75
12	4198407	464460	76
15	4236323	381292	82

Table 1: Mean number of comparisons by pattern length for `bible.txt` (4047392 bytes)

The main disadvantage of the binary search approach is its memory requirements. All the search methods use a number of arrays whose size is proportional to the length of the input file. Commonly, characters are stored in a single byte and integers take four bytes. Since integers in this context are used as indices into the file, this representation allows for files up to $2^{32} = 4$ GB in size.

Assuming single-byte characters and four-byte integers, an array of n characters will take n bytes, and an array of integers will take $4n$ bytes. Both the decompress-then-search and compressed-domain Boyer-Moore methods use one array of characters and one of integers, for a total of $5n$ bytes of memory (disregarding other, smaller arrays whose size is $\mathcal{O}(|\Sigma|)$), while the binary search method uses one character array and two integer arrays, for a total of $9n$ bytes. For a typical file one megabyte in size, this is an additional memory requirement of four megabytes. However, we believe that this cost is justified by the performance benefits of binary search, especially in light of the increasing amount of memory available on personal computers.

The other figure of interest is the number of comparisons performed during the search phase, since this represents the amount of work done by the algorithm. Table 1 shows the mean number of comparisons required to find all occurrences of a pattern in `bible.txt` by pattern length. Somewhat incredibly, the binary search algorithm is able to find every instance of a particular pattern in under a hundred comparisons on average, compared to Boyer-Moore (hundreds of thousands of comparisons) and linear search (millions of comparisons).

In the above experiments we have used patterns that are known to appear in the input file. However, in practice, there is no noticeable difference when searching for patterns that do *not* appear in the file. Binary search may have to perform one or two extra iterations, because it has direct access to patterns starting with a given character, but terminates almost as quickly as if an occurrence of the pattern had been found, while Boyer-Moore still traverses the entire file.

6 Conclusion

In this paper we have demonstrated how to take advantage of the structure placed in files compressed with the Burrows-Wheeler transform, and in particular how to use this structure for pattern-matching applications, introducing two new methods for solving the pattern matching problem in the context of BWT-compressed files.

As noted in Section 5, there is almost no difference in speed between the two compressed-domain algorithms for straightforward, single-pattern searches. In this case, the Boyer-Moore algorithm has the advantage of lower memory requirements, and should prove useful for applications where memory is at a premium. On the other hand, the slightly higher memory footprint of the binary search algorithm is offset by dramatic performance gains for multiple pattern searches, and by a considerable increase in the power of possible searches, including boolean and 'pseudo-boolean' queries (AND, OR, NEAR) and approximate matching.

In fact, the Burrows-Wheeler transform has many more uses than simple pattern-matching. There are many other uses of a sorted list of substrings, such as finding the longest repeated sequence, detecting plagiarism in collections of text or source code, constructing a keyword-in-context (KWIC) index, and simple data mining, since, for example, all substrings containing the letters '`www.`' will be adjacent.

There is still considerable scope for future investigation and research.

In Section 5, we showed that it is possible to search directly on the permuted text, and in this way to achieve significant reductions in search time. However, if a file has been compressed with the 'BWT $\rightarrow$ MTF $\rightarrow$ RLE $\rightarrow$ VLC' method, it is still necessary to perform three stages of decoding to obtain the permuted text. This represents a significant amount of work—up to a third of the total search time in some cases.

It is possible to obtain the F array from the move-to-front output in $\mathcal{O}(n \log |\Sigma|)$ time. It may be possible to extend this to other arrays required for pattern-matching, to gain a performance increase by skipping the reverse move-to-front step altogether. It may even be possible to skip the run-length decoding process in a similar way, leaving only the arithmetic decoding stage.

A disadvantage of the compressed-domain binary search algorithm is that it requires that the entire input file be treated as a single block. This requires enough memory to load the entire file at once, which may be infeasible when decoding files that are many megabytes or even gigabytes in size, although when encoding it may be possible to perform the sorting in several phases, similar to mergesort.

Most block-sorting compressors use much smaller blocks for the sake of increased performance. It is therefore necessary to investigate the effect of 'blocked' files on the performance of the binary search algorithm. This will involve developing a method to deal with the case when an instance of a pattern overlaps a block boundary. It is expected that the modified routine will be approximately $\mathcal{O}(\frac{mn}{b} \log \frac{b}{|\Sigma|})$, where b is the block size.

So far we have only discussed exact pattern matching with the Burrows-Wheeler transform. It is possible to adapt the binary search techniques outlined here to the task of wildcard pattern matching. For example, one could rotate the pattern so that the longest substring is at the beginning and can then be used to filter the text for possible matches, which can then be verified (Witten et al. 1999). This is similar to the approach of Bratley & Choueka (1982), but without the significant memory overhead of storing the rotated lexicon, since this is captured by the BWT.

References

Bell, T., Adjeroh, D. & Mukherjee, A. (2001), Pattern matching in compressed texts and images. Draft of May 23, 2001 (Submitted to ACM Computing Surveys).

Bentley, J. L., Sleator, D. D., Tarjan, R. E. & Wei, V. K. (1986), 'A locally adaptive data compression scheme', *Communications of the ACM* **29**(4), 320–330.

Boyer, R. S. & Moore, J. S. (1977), 'A fast string searching algorithm', *Communications of the ACM* **20**(10), 762–772.

Bratley, P. & Choueka, Y. (1982), 'Processing truncated terms in document retrieval systems', *Information Processing and Management* **18**(5), 257–266.

Burrows, M. & Wheeler, D. (1994), A block-sorting lossless data compression algorithm, Technical report, Digital Equipment Corporation.

Fenwick, P. M. (1996), Block sorting text compression—final report, Technical Report 130, Department of Computer Science, The University of Auckland.

Ferragina, P. & Manzini, G. (2000), Opportunistic data structures with applications, *in* 'IEEE Symposium on Foundations of Computer Science'.

Ferragina, P. & Manzini, G. (2001), An experimental study of a compressed index. Part of this work appeared in ACM-SIAM Symposium on Discrete Algorithms.

Knuth, D. E., Morris, Jr, J. H. & Pratt, V. R. (1977), 'Fast pattern matching in strings', *SIAM Journal of Computing* **6**(1), 323–350.

Sadakane, K. (2000), Unifying Text Search and Compression—Suffix Sorting, Block Sorting and Suffix Arrays, PhD thesis, University of Tokyo.

Sadakane, K. & Imai, H. (1999), A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler transformation, *in* 'Advances in Database Technologies', number 1552 *in* 'Lecture Notes in Computer Science', pp. 434–445.

Witten, I. H., Moffatt, A. & Bell, T. C. (1999), *Managing Gigabytes*, second edn, Morgan Kaufmann.

Ziv, J. & Lempel, A. (1977), 'A universal algorithm for sequential data compression', *IEE Transactions on Information Theory* **IT-23**(3), 337–343.

Ziv, J. & Lempel, A. (1978), 'Compression of individual sequences via variable rate coding', *IEE Transactions on Information Theory* **IT-24**(5), 530–536.