# Improvements to Burrows-Wheeler Compression Algorithm

Sebastian Deorowicz

Silesian University of Technology,
Institute of Computer Science,
44-100 Gliwice, Akademicka 16, Poland
(e-mail: sdeor@star.iinf.polsl.gliwice.pl)

November 17, 2000

This is a preprint of an article published in
Software—Practice and Experience, 2000; 30(13):1465–1483

http://www.interscience.wiley.com

**Abstract**

In 1994 Burrows and Wheeler presented a new algorithm for lossless data compression. The compression ratio that can be achieved using their algorithm is comparable with the best known other algorithms, whilst its complexity is relatively small. In this paper we explain the internals of this algorithm and discuss its various modifications that have been presented so far. Then we propose new improvements for its effectiveness. They allow us for obtaining the compression ratio equal to 2.271 bpc for the Calgary Corpus files, which is the best result in the class of Burrows-Wheeler Transform based algorithms.

KEY WORDS: data compression; lossless compression; Burrows-Wheeler Transform; block-sorting; Calgary Corpus; Canterbury Corpus

## 1 Introduction

In 1994 Burrows and Wheeler [1] presented a data compression algorithm based on the Burrows-Wheeler Transform (BWT). Its compression ratios were comparable with the ones obtained using the best statistical methods. Many papers describing various modifications of the original algorithm have appeared since then. Their authors analyzed the algorithm from the point of view of information theory and offered methods to improve the compression ratio as well as to decrease its run time and memory complexity.

In the first part of this paper we define the compression algorithm and the BWT, which is a base for it. We further discuss the modifications that have been presented so far. In the end we propose several new modifications that improve the efficiency and compare our compression algorithm with other known methods.


Figure 1: Burrows-Wheeler Compression Algorithm.

# 2 Burrows-Wheeler Compression Algorithm

## 2.1 Definitions

In this section we introduce some definitions and notations, which are necessary to precisely describe the Burrows-Wheeler Transform (BWT).

Let us assume that $x = x_1 x_2 \ldots x_n$ is a *sequence*. The sequence *length* denoted by $n$ is a number of elements in $x$, and $x_i$ denotes the $i$-th element of $x$. We define the *reverse sequence*, $x^{-1}$, as $x_n x_{n-1} \ldots x_1$. Let $x = uvw$ for some, possibly empty, sequences $u$, $v$, $w$. Then $u$ is called a *prefix* of $x$, $v$ a *component* of $x$ and $w$ a *suffix* of $x$. The element $x_i$ belongs to a finite ordered set $\mathcal{A} = \{a_0, a_1, \ldots, a_{k-1}\}$ that is called an *alphabet*. The number of elements in $\mathcal{A}$ is the *size* of the alphabet and is denoted by $k$. The elements of the alphabet are called *symbols* or *characters*. A *run* is a non-empty component of $x$ that consists of identical symbols.

## 2.2 Compression algorithm

In this section we provide an insight description of the steps performed by the Burrows-Wheeler compression algorithm (BWCA) (figure 1). Let the input data of the BWCA be a sequence $x$ of length $n$. First we have to compute the BWT. To achieve this we create $n$ sequences by rotating $x$ by one symbol. The $n$ sequences created in this way are put into $n \times n$ matrix $M(x)$:

$$M(x) = \begin{bmatrix} x_1 & x_2 & \cdots & x_{n-1} & x_n \\ x_2 & x_3 & \cdots & x_n & x_1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{n-1} & x_n & \cdots & x_{n-3} & x_{n-2} \\ x_n & x_1 & \cdots & x_{n-2} & x_{n-1} \end{bmatrix}$$

Then matrix $M(x)$ is transformed into $\widetilde{M}(x)$ by sorting its rows in lexicographic order. Let $R(x)$ denote the row number of sequence $x$ in $\widetilde{M}(x)$. The results of the BWT are: $R(x)$ and the last column of matrix $\widetilde{M}(x)$ that we denote by $x^{bwt}$.

Once the BWT is completed, $x^{bwt}$ is encoded using the Move To Front (MTF) transform [2]. The coding proceeds as follows. First the list $L = (a_0, a_1, \cdots, a_{k-1})$ consisting of the symbols of the alphabet $\mathcal{A}$ is created. Then to each symbol of $x_i^{bwt}$, $i = 1, 2, \ldots, n$, a number $p_i$ is assigned such that $x_i^{bwt}$ is equal to the $p_i$-th element of $L$, and the $p_i$-th element is moved to the beginning of list $L$. As the result we obtain a sequence $x^{mtf}$ over the alphabet $\mathcal{A}_{mtf}$ consisting of integer numbers from range $[0, k-1]$.

In the last step, sequence $x^{mtf}$ is compressed using a universal entropy coder, which could be the Huffman or arithmetic coder. The number $R(x)$ is also coded as a binary number of $\lceil \log_2 n \rceil$ bits.

### 2.3 Decompression algorithm

The reverse Burrows-Wheeler Transform is based on the observation that sequence $x^{bwt}$ is a permutation of sequence $x$ and its sorting gives the first column of matrix $\widetilde{M}(x)$ that is a first character of a context by which the sequence $x^{bwt}$ is sorted. Therefore given a symbol $x_i^{bwt}$, a symbol $s$ located in the first column and $i$-th row of matrix $\widetilde{M}(x)$ can be found. Knowing that this is the $j$-th occurrence of $s$ in the first column of matrix $\widetilde{M}(x)$ we find its $j$-th occurrence in the last column. Moreover the symbol $s$ precedes $x_i^{bwt}$ in sequence $x$. Thus if we know $R(x)$ we also know the last character of the sequence $x$, i.e. $x_{R(x)}^{bwt}$. Starting from this character we can iterate in a similar manner to restore the original sequence $x$ in time $O(n)$.

### 2.4 Example

Let us present the execution of the compression algorithm on sequence $x = abracadabra$ and $\mathcal{A} = \{a, b, c, d, r\}$. In the first step we compute:

$$M(x) = \begin{bmatrix} a & b & r & a & c & a & d & a & b & r & a \\ b & r & a & c & a & d & a & b & r & a & a \\ r & a & c & a & d & a & b & r & a & a & b \\ a & c & a & d & a & b & r & a & a & b & r \\ c & a & d & a & b & r & a & a & b & r & a \\ a & d & a & b & r & a & a & b & r & a & c \\ d & a & b & r & a & a & b & r & a & c & a \\ a & b & r & a & a & b & r & a & c & a & d \\ b & r & a & a & b & r & a & c & a & d & a \\ r & a & a & b & r & a & c & a & d & a & b \\ a & a & b & r & a & c & a & d & a & b & r \end{bmatrix},$$

$$\widetilde{M}(x) = \begin{bmatrix} a & a & b & r & a & c & a & d & a & b & \mathbf{r} \\ a & b & r & a & a & b & r & a & c & a & \mathbf{d} \\ \underline{a} & \underline{b} & \underline{r} & \underline{a} & \underline{c} & \underline{a} & \underline{d} & \underline{a} & \underline{b} & \underline{r} & \underline{\mathbf{a}} \\ a & c & a & d & a & b & r & a & a & b & \mathbf{r} \\ a & d & a & b & r & a & a & b & r & a & \mathbf{c} \\ b & r & a & a & b & r & a & c & a & d & \mathbf{a} \\ b & r & a & c & a & d & a & b & r & a & \mathbf{a} \\ c & a & d & a & b & r & a & a & b & r & \mathbf{a} \\ d & a & b & r & a & a & b & r & a & c & \mathbf{a} \\ r & a & a & b & r & a & c & a & d & a & \mathbf{b} \\ r & a & c & a & d & a & b & r & a & a & \mathbf{b} \end{bmatrix},$$

$$x^{bwt} = rdarcaaaabb,$$

$$R(x) = 3.$$

In the second step the MTF transform is carried out. We assume that the elements of list $L = (a, b, c, d, r)$ appear in the alphabetical order. The result of coding is $x^{mtf} = 44224200040$. Below the contents of list $L$ during coding of symbols from $x^{bwt}$ are shown:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $L$ | a<br>b<br>c<br>d<br>r | r<br>a<br>b<br>c<br>d | d<br>r<br>a<br>b<br>c | a<br>d<br>r<br>b<br>c | r<br>a<br>d<br>b<br>c | c<br>r<br>a<br>d<br>b | a<br>c<br>r<br>d<br>b | a<br>c<br>r<br>d<br>b | a<br>c<br>r<br>d<br>b | a<br>c<br>r<br>d<br>b | b<br>a<br>c<br>r<br>d |
| $x^{bwt}$ | r | d | a | r | c | a | a | a | a | b | b |
| $x^{mtf}$ | **4** | **4** | **2** | **2** | **4** | **2** | **0** | **0** | **0** | **4** | **0** |

Now we present the reverse Burrows-Wheeler Transform. At the beginning the first and last columns of $\widetilde{M}(x)$ are found:

$$\widetilde{M}(x) = \begin{bmatrix} a & \cdots & \mathbf{r} \\ a & \cdots & \mathbf{d} \\ \underline{a} & \cdots & \underline{\mathbf{a}} \\ a & \cdots & \mathbf{r} \\ a & \cdots & \mathbf{c} \\ b & \cdots & \mathbf{a} \\ b & \cdots & \mathbf{a} \\ c & \cdots & \mathbf{a} \\ d & \cdots & \mathbf{a} \\ r & \cdots & \mathbf{b} \\ r & \cdots & \mathbf{b} \end{bmatrix}.$$

The reverse Burrows-Wheeler Transform starts from the row number $R(x) = 3$, whose last character is $a$. We check that this is the first occurrence of $a$ in the last column, and we find the first occurrence of $a$ in the first column, which happens to be in the first row. Then we find the last symbol in this row, $r$, and check that this is the first occurrence of that symbol in the last column. Thus we search for the first occurrence of $r$ in the first column finding it in the tenth row. This procedure is repeated until the whole sequence $x^{-1}$ is retrieved:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| row number | 3 | 1 | 10 | 6 | 2 | 9 | 5 | 8 | 4 | 11 | 7 |
| last character in row | a | r | b | a | d | a | c | a | r | b | a |
| number of occurrence of character | 1 | 1 | 1 | 2 | 1 | 5 | 1 | 4 | 2 | 2 | 3 |
| new row number | 1 | 10 | 6 | 2 | 9 | 5 | 8 | 4 | 11 | 7 | 3 |

$$x^{-1} = arbadacarba.$$

Using $x^{-1}$ we obtain:

$$x = abracadabra.$$

## 2.5 Theoretical results

A theoretical analysis for the BWT algorithm was given by Balkenhol and Kurtz [3]. They presented the effective implementation techniques for data compression programs using the BWT. Arimura and Yamamoto [4] presented a theoretical analysis of the efficiency of the BWCA. They proved it to be asymptotically optimal for a stationary ergodic finite order Markov source. Other theoretical analyzes of the BWT and BWT-based compression algorithm are also presented by Arnavut and Magliveras [5] and Effros [6].

# 3 Modifications of the algorithm

In this section we describe various modifications introduced to the BWCA after the formulation of the Burrows-Wheeler Transform. Some of them were introduced to improve the compression ratio, other to speed up the compression process.

## 3.1 Run Length Encoding

Since sequences $x$ may contain many long runs the time of computation of the BWT using sorting (see sec. 3.2) can be quite long. Various mechanisms were suggested to overcome this problem. Burrows and Wheeler introduced special sorting of such sequences. Fenwick used the Run Length Encoding (RLE) that eliminates long runs and decreases the length of a sequence on which the transform is performed. As the result the sorting time is reduced. In most cases the RLE makes the compression ratio slightly, by about 0.1%, worse because the RLE causes some of the information of the context in which symbols occur to be lost. If sequence $x$ contains many long runs, then it turns out that sequence $x^{mtf}$ is composed in 90% of 0's. In such a case the RLE allows for substantial reduction of the number of zeroes in sequence $x^{mtf}$ and the better estimation of non-zero symbols probabilities, thus improving the overall compression ratio. Taking this observation into account Balkenhol and Kurtz [3] suggested to use the RLE only when the length of sequence $x^{rle}$ is less than $0.7n$.

Note that if the algorithms for calculating the BWT based on suffix tree construction are used (see sec. 3.2), then the occurrences of runs do not significantly influence their execution time. If the RLE-0 transform is used (see sec. 3.4) then the problem of exact probability estimation in the entropy coder also becomes insignificant. Therefore it seems that the RLE should not be used.

## 3.2 Burrows-Wheeler Transform

Burrows and Wheeler [1] presented an algorithm for the BWT calculation based on sorting. Even though in most practical cases the algorithm is effective enough, its worst case time complexity is $O(n^2 \log n)$. Burrows and Wheeler suggested also the suffix tree application for faster transform calculation. To this goal a special symbol '\$' is appended to sequence $x$. The symbol '\$' has to be lexicographically greater than any symbol in $x$. For sequence $x\$$ the problem of sorting the cyclic shifts is reduced to the problem of suffix sorting that can be solved by building a suffix tree and searching it, in linear time, in the lexicographic order.

There are many effective algorithms for building a suffix tree. In References [7, 8] the linear time complexity algorithms were shown. Their serious deficiency is a high memory consumption. Balkenhol and Kurtz [9] used a version of the McCreight algorithm, which was modified by Kurtz [10], whose memory complexity was reduced to $20n$ in the worst case and to $10n$ in the average case. Except the already mentioned, there are other algorithms that work in higher than linear time. e.g.:

- Manbers-Myers algorithm [11] with the worst case time complexity $O(n \log n)$ and the memory complexity $8n$,
- Bentley-Sedgewick algorithm [12] based on quicksort with the worst case time complexity $O(n^2)$ and the average complexity $O(n \log n)$; it is characterized by a very low memory consumption equal to $5n$ plus the memory needed to store the quicksort's stack,
- Larsson-Sadakane algorithm [13] with the worst case time complexity $O(n \log n)$ and the memory complexity $8n$.

Larsson and Sadakane [13] showed that for most sequences in practical applications, algorithms of higher complexity than linear could work faster than linear complexity algorithms.

The authors of most works concerning the BWT concentrate on accelerating the transform itself. Recently Chapin and Tate [14] noted that in contrast to other compression methods, such as PPM or LZ77, the compression ratio of the BWCA largely depends on the lexicographic order of the alphabet. Based on this observation, they proposed several methods for a slight improvement of the compression ratio.

Balkenhol, Kurtz and Shtarkov [9] showed that applying the compression algorithm to reverse sequence $x^{-1}$ for an alphabet of size $k = 256$ results sometimes in significant improvement of the compression ratio.

### 3.3 Move To Front Transform

The BWT groups together similar contexts occurring in sequence $x$. Hence the short components of sequence $x^{bwt}$ often contain only a few frequently occurring characters. The sequence $x^{bwt}$ is a permutation of $x$. Therefore the symbols grouped in different fragments of $x^{bwt}$ differ from each other. The application of the Move To Front (MTF) transform is meant to replace this local structure by a global one that should allow for a better probability estimation in the entropy coder and a better compression ratio.

In their original paper Burrows and Wheeler [1] proposed usage of the MTF transform, but they suggested also that refraining from moving symbols from the higher positions, i.e. such that have not occurred for a long time, to the beginning of list $L$, may result in improving the compression ratio. Fenwick [15] explored such possibilities but failed to obtain better compression ratios. Other modifications of the MTF were described by Schindler [17], though he has not improved its performance either. Lately another modification was suggested by Balkenhol, Kurtz and Shtarkov [9]. Their idea is to move to the beginning of list $L$ (position 0) only the symbols appearing at position 1. The characters located at higher positions on list $L$ are moved to position 1 (such a version is denoted by MTF-1). It turns out that such a modification improves the compression ratio.

Fenwick [15] investigated also an option to abandon the MTF transform while greatly improving the adaptiveness of the entropy coder. His work, however, shows a significant decline in compression ratio, by $5 - 10\%$, when the entropy coder is applied directly to sequence $x^{bwt}$.

The problem solved by the MTF algorithm is known as the List Update Problem. The specific structure of sequence $x^{bwt}$ causes other algorithms solving this problem, e.g. Translate and Frequency Count, not to produce good results. Lately Albers [18] suggested a new algorithm, called Time Stamp, for solving the List Update Problem. The results obtained by Albers and Mitzenmacher [19] show that replacing the MTF with it in the BWCA results in a small improvement of the compression ratio only for some sequences, whilst for most cases the results are worse.

### 3.4 Zero Run Transform

Zero is a dominant symbol of sequence $x^{mtf}$. For some sequences $x$, taken from the Calgary Corpus [20], the percentage of zeroes in sequence $x^{mtf}$ may reach 90%. On the average they contain 60% zeroes. Hence, there are many long runs in sequence $x^{mtf}$ consisting of zeros, so called 0-runs. As already mentioned this may lead to some difficulties in an effective probability estimation. Therefore a Zero Run Transform was suggested [3, 16] (denoted here by RLE-0) that treats 0-runs in a special way. Table 1 shows an example of 0-runs coding, and a more detailed description of the RLE-0 can be found in Reference [16]. Applying the RLE-0 transform results in sequence $x^{rle0}$ over the alphabet $\mathcal{A}_{rle0} = (\mathcal{A}_{mtf} \setminus \{0\}) \cup \{0_a, 0_b\}$. Experimental results indicate [3] that the application of the RLE-0 transform indeed improves the compression ratio.

Table 1: Example of the RLE-0 transform.

| 0-run length | RLE-0 code |
|---|---|
| 1 | $0_a$ |
| 2 | $0_b$ |
| 3 | $0_a 0_a$ |
| 4 | $0_a 0_b$ |
| 5 | $0_b 0_a$ |
| 6 | $0_b 0_b$ |
| 7 | $0_a 0_a 0_a$ |
| 8 | $0_a 0_a 0_b$ |
| 9 | $0_a 0_b 0_a$ |
| ... | ... |

### 3.5 Entropy Coding

Sequence $x^{rle0}$ or $x^{mtf}$, in case when the RLE-0 is not used, is compressed with an entropy coder. Burrows and Wheeler [1] used the Huffman coder in their work. Other authors chose the arithmetic coders for that goal. The usage of arithmetic coding is motivated by the fact, that for the Huffman coding in the worst case the length of the code may be longer by $n$ bits than the entropy of the coded sequence (a detailed analysis may be found in Reference [3]). On the other hand, the application of the arithmetic coding using the Krichevsky-Trofimov probability model [21] allows to reduce the distance of the code length in the worst case from entropy to $\frac{k-1}{2}\log_2 n + c$ bits. Unfortunately this estimation is true only if the arithmetic coder works on infinite precision numbers. An analysis of practical applications of arithmetic coding was recently shown by Moffat, Neal and Witten [22].

The analysis of the probability distribution of symbols in sequence $x^{mtf}$ shows that zero is a dominant symbol and the probabilities of higher symbols usually decrease monotonically. The second important observation is that the probabilities of lower and higher symbols occurrence may differ by $4-5$ orders of magnitude. Furthermore the probabilities of symbol occurrence significantly change depending on a current position of a symbol in sequence $x^{mtf}$. For these reasons applying a universal entropy coder directly to sequence $x^{mtf}$ does not lead to satisfying results. The very high probability of occurrence of 0's causes difficulties in estimation of probabilities of occurrence of higher symbols (this problem is partially solved by using the RLE-0). Significant changes in probability distribution require a highly adaptive coder to allow the coder to quickly adapt to local probability distribution. However occurrences of higher symbols severely disturb the probability distribution (a precise analysis of this phenomenon is done by Fenwick [23]). Thus several authors [3, 9, 15, 23] use a hierarchical model in which the characters are divided into some classes in such a way that within a class the differences in probabilities of occurrence of symbols are relatively small. Arithmetic coders are based on probability estimation in such models. Since the structure of sequence $x^{mtf}$ is not fully known, it seems that the proper choice of the model may greatly improve the compression ratio.

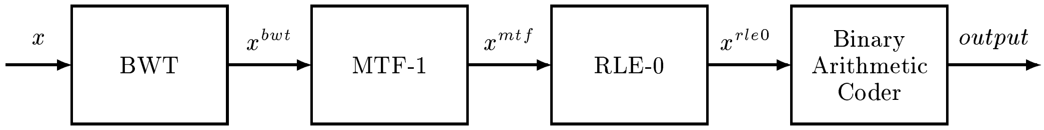


Figure 2: An improved version of the BWT-based compression algorithm.

Table 2: Encoding of alphabet $\mathcal{A}_{rle0}$.

| symbol | code |
|---|---|
| $0_a$ | 0 0 |
| $0_b$ | 0 1 |
| 1 | 1 0 |
| $2, \ldots, 7$ | 1 1 0 $b_2$ $b_1$ $b_0$ |
| $8, \ldots, 15$ | 1 1 1 0 $b_2$ $b_1$ $b_0$ |
| $16, \ldots, 31$ | 1 1 1 1 0 $b_3$ $b_2$ $b_1$ $b_0$ |
| $32, \ldots, 63$ | 1 1 1 1 1 0 $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ |
| $64, \ldots, 127$ | 1 1 1 1 1 1 0 $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ |
| $128, \ldots, 255$ | 1 1 1 1 1 1 1 $b_6$ $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ |

# 4 Improvements to Burrows-Wheeler compression algorithm

In this section we present our modifications to the BWCA. A general scheme of the compression algorithm is shown in figure 2. Our suggestions of changes to the specific steps of compression and achieved improvements in the compression ratio are presented below.

The structure of sequence $x^{rle0}$ is not well known, therefore it is difficult to decide which modeling method should be used to achieve the best compression results. Our proposition, presented below, allows us to achieve higher compression ratios than in other approaches.

We have chosen to use arithmetic coding in the entropy coder. However in contrast to other approaches we have chosen a binary coder. It requires a slightly more complicated modeling. In the first step we group symbols from alphabet $\mathcal{A}_{rle0}$ into a few subsets: $\{0_a\}$, $\{0_b\}$, $\{1\}$, $\{2, \ldots, 7\}$, $\{8, \ldots, 15\}$, ..., $\{128, \ldots, 255\}$. Then we encode all the symbols using the binary prefix code, which is presented in table 2. Each code consists of two parts: a unique prefix, which distinguishes the subsets from each other, and a suffix, which is a binary representation of the symbol ($b_i$ denotes the $i$-th bit of the binary representation of the symbol). The second part of the code, which is not always present, contains only these bits that are indispensable to differentiate all characters in a given subset. (Thus, for example, for subsets $\{8, \ldots, 15\}$, ..., $\{128, \ldots, 255\}$ it is not necessary to encode the most significant bit of the symbol.)

Another problem is to encode the string of bits which is obtained after coding symbols with the above-described binary code. In order to improve the probability estimation, the separate statistics for different contexts in which consecutive bits occur are calculated. Probabilities of bits in different contexts may be completely different, thus the correct choice of contexts is essential. The way in which the context $c$ is calculated for successive bits of code is shown in figure 3.

| code bit | number of possible contexts | possible contexts |
|---|---|---|
| first bit | 6 | last character is 0[1], and the length of the 0-run is not longer than 2 |
| | | last character is 0, and the length of the 0-run is longer than 2 |
| | | last character is 1, and the last but one character is 0 |
| | | last character is 1, and the last but one character is not 0 |
| | | last character is greater then 1, and the last but one character is 0 |
| | | last character is greater then 1, and the last but one character is not 0 |
| second bit (first bit = 0) | $\log_2 n$ | length of 0-run forms a context |
| second bit (first bit = 1) | 2 | last character is 0 or 1 |
| | | otherwise |
| successive code bits | 254 | all previous code bits form a context |

Figure 3: The way in which context is calculated.

In our algorithm we treat the sequence of bits occurring in context $c$ as a Markov chain of order $d$, and we use the Krichevsky-Trofimov estimator [21] to estimate the probability of bit occurrence. This means that we encode the subsequent bits with conditional probability:

$$P_c^d(a|s_c^d) = \frac{t_c^d(a|s_c^d) + \frac{1}{2}}{t_c^d(0|s_c^d) + t_c^d(1|s_c^d) + 1}, \tag{1}$$

where $s_c^d$ are the $d$ last bits coded in context $c$, $t_c^d(a|s_c^d)$ is a number of occurrences of bit $a$ in context $c$, provided the last $d$ coded bits were $s_c^d$. (The KT-estimator has two advantages: it gives very good results and the quality of its estimation is well known.) The higher the order $d$ of the Markov chain, the better the probability estimation. Unfortunately as $d$ grows, the number of states $s$ for which the probability is estimated grows exponentially (as $2^d$). Thus it adds redundancy caused by estimating probability in many different states. The empirical results (see the first five rows in table 3) show that the reasonable choice is $d = 2$.

In further experiments we checked the results of the probability estimation defined as a weighted probability for different orders, i.e. calculated as follows:

$$P_c^{d_1,d_2}(a|s_c^{\max(d_1,d_2)}) = \frac{1}{2}P_c^{d_1}(a|s_c^{d_1}) + \frac{1}{2}P_c^{d_2}(a|s_c^{d_2}), \tag{2}$$

for different orders $d_1$ and $d_2$. It follows from formula (2) that we treat the sequence of bits both as a Markov chain of order $d_1$ and of order $d_2$, and estimate the probabilities in both models. In the next step of coding the average of these two probabilities is used. Due to the fact, that we do not know which model describes the encoded sequence of bits in a particular context, and the fact that a model may depend on context $c$ and a current position of a symbol in the sequence, using such an estimation may give good results. As the experiments show (see the last ten rows in table 3) this method of calculating the probability indeed leads to the better compression ratios. The best results are obtained setting $d_1 = 0$ and $d_2 = 2$, i.e. estimating the probability as follows:

$$P_c^{0,2}(a|s_c^{\max(d_1,d_2)}) = \frac{1}{2}\left(\frac{t_c^0(a|s_c^0) + \frac{1}{2}}{t_c^0(0|s_c^0) + t_c^0(1|s_c^0) + 1} + \frac{t_c^2(a|s_c^2) + \frac{1}{2}}{t_c^2(0|s_c^2) + t_c^2(1|s_c^2) + 1}\right) \tag{3}$$

We want to point out that formula (3) follows from our practical experiments in which we achieved very good compression ratios. Unfortunately we cannot support it by a precise theoretical analysis due to still incomplete knowledge about the structure of sequence of bits occurring in context $c$, which would prove the concept of using this model (and the choice of $d_1 = 0$ and $d_2 = 2$ as well).

The analysis of sequence $x^{mtf}$ indicates (figure 4) that for most data files from the Calgary Corpus and Canterbury Corpus the average value of symbols does not exceed 15. However for some binary files (`geo`, `kennedy`, `obj1`) there are fragments for which it grows significantly, even exceeding 100. It can be seen in figure 4 that the fragments with a large average value are preceded by fragments with a very small average. In such a case to achieve a good adjustment of the arithmetic coder to the local properties of a coded sequence, its adaptive capabilities are crucial. The second observation is that for the fragments with a high symbol average the model we have used is less effective because the relatively long codes correspond to each symbol. Thus a running average is calculated during arithmetic coding:

$$avg(t) = avg(t-1) \cdot (1-\epsilon) + x_t^{rle0} \cdot \epsilon. \tag{4}$$

If it exceeds a certain threshold, the symbols from alphabet $\mathcal{A}_{rle0}$ are encoded with the code presented in table 4 (we set $\epsilon = 0.15$).

[1]By 0 any of symbols $0_a$ and $0_b$ is meant.

Table 3: The average compression ratio for various methods of probability estimation for the Calgary Corpus.

| method | compression ratio |
|---|---|
| 0-order | 2.298 |
| 1-order | 2.289 |
| 2-order | 2.287 |
| 3-order | 2.289 |
| 4-order | 2.293 |
| 0-1-weighted | 2.273 |
| **0-2-weighted** | **2.271** |
| 0-3-weighted | 2.272 |
| 0-4-weighted | 2.273 |
| 1-2-weighted | 2.285 |
| 1-3-weighted | 2.282 |
| 1-4-weighted | 2.281 |
| 2-3-weighted | 2.286 |
| 2-4-weighted | 2.284 |
| 3-4-weighted | 2.288 |

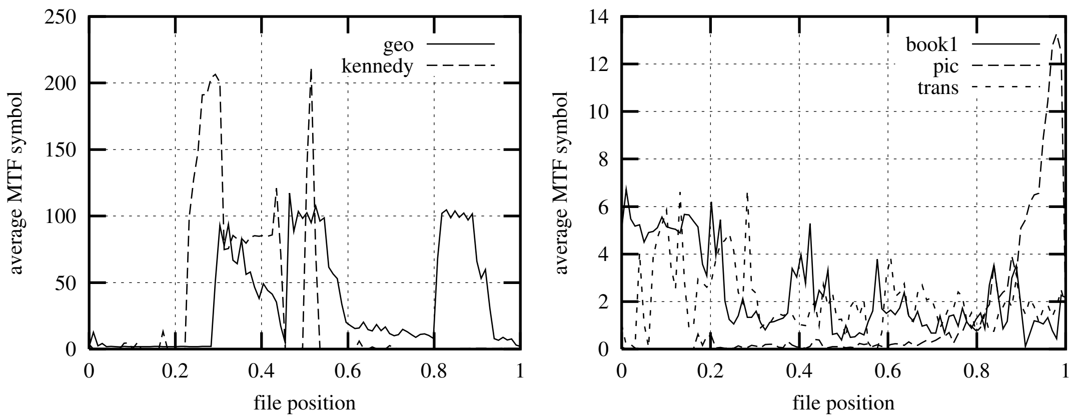


Figure 4: Average value of symbols in consecutive fragments of sequence $x^{mtf}$ for different data files.

Table 4: Encoding of alphabet $\mathcal{A}_{rle0}$ for a high average symbol value.

| symbol | code |
|---|---|
| $0_a$ | 0 0 |
| $0_b$ | 0 1 |
| $1, \ldots, 255$ | 1 $b_7$ $b_6$ $b_5$ $b_4$ $b_3$ $b_2$ $b_1$ $b_0$ |

Table 5: Dependence of compression ratio on modeling chosen during compression.

| file | without threshold | threshold = 64 |
|---|---|---|
| geo | 4.233 | **4.221** |
| kennedy | 0.698 | **0.659** |
| obj1 | 3.738 | **3.737** |

Table 5 shows the comparison of the compression ratios when the above-described coding is applied with the average threshold value equal 64.

Balkenhol and Kurtz [3] showed that if we assume that sequence $x$ is generated by a tree source, then sequence $x^{bwt}$ consists of components, each of them corresponding to one of the leaves in the tree source model. (A similar observation was made by Cleary, Teahan and Witten [24].) Unfortunately, during a compression process the model of the source describing sequence $x$ is not known. Furthermore, it is not known where the change of the context takes place in sequence $x^{bwt}$. The probability distribution, as well as a number of symbols, varies significantly from one context to another, thus the probability distribution in subsequent fragments of sequence $x^{rle0}$ may also vary a lot. Therefore to adjust the coder to the local symbol distribution in the best possible manner, the counters $t_c^0(a|s_c^0)$ and $t_c^2(a|s_c^2)$ are halved following the formulas below:

$$t_c^0(a|s_c^0) = \left\lfloor \frac{t_c^0(a|s_c^0)}{2} \right\rfloor, \qquad t_c^2(a|s_c^2) = \left\lfloor \frac{t_c^2(a|s_c^2)}{2} \right\rfloor.$$

This halving occurs when the sum of the counters in a given context for 0 and 1 exceeds thresholds $t_{max}^0$ and $t_{max}^2$. Due to the different speed at which the probability distributions change in different contexts, the threshold values $t_{max}^0$ and $t_{max}^2$ differ from context to context (see table 6).

A small improvement in compression ratio can be also achieved by the initialization of counters $t_c^0$ and $t_c^2$ with values $\frac{t_{max}^0}{32}$ and $\frac{t_{max}^2}{32}$ respectively, and by incrementing these counters by 2 (in this case $t_{max}^0$ and $t_{max}^2$ are multiplied by 2).

Table 6: Thresholds for counter halving in different contexts.

| contexts | $t^0_{max}$ | $t^2_{max}$ |
|---|---|---|
| first bit<br>second bit (first bit = 1)<br>remaining bits of code prefix (except $b_i$) | 20 | 150 |
| first bit $b_i$, when the average $\leq 64$<br>first four bits $b_i$, when the average $> 64$ | 30 | 300 |
| second bit (first bit = 0)<br>all except the first bit $b_i$, when the average $\leq 64$<br>last four bits $b_i$, when the average $> 64$ | 300 | 700 |

# 5 Complexity analysis

It is easily seen that the time complexity of the MTF, RLE-0 and arithmetic coding steps of our algorithm are $O(n)$, whilst, under the assumption that the size of the alphabet is neglected, their memory complexity is $O(1)$. The same applies to the complexity of the invert algorithms.

Now we focus on the BWT complexity analysis. As already mentioned (see sec. 3.2) there are many different algorithms for its computation. If it is important to have the best worst case time complexity, then one of the linear time algorithms should be used. In such a case we would have an algorithm with the time and memory complexities $O(n)$. However in our work we used the Larsson-Sadakane algorithm [13], which has the worst case time complexity $O(n \log n)$. Theoretically it has worse parameters than the linear algorithms, but in practice it runs faster. Such a behavior is observed in the empirical results for the test data files from the Calgary Corpus and Canterbury Corpus. Its memory complexity, equal $8n$, is also lower than for the linear algorithms. For the sake of completeness the parameters of the invert BWT were also calculated: the memory complexity — $4n$, and the time complexity — $O(n)$.

# 6 Experimental results

A series of experiments has been carried out to compare the proposed compression algorithm with other methods in terms of their time complexity and compression ratio.

A standard set of data files — the Calgary Corpus [20] — has been used in the first experiment. The results of the comparison of our algorithm with other algorithms known from the literature are shown in table 7. Each column contains the results expressed in output bits per input character (bpc) for the following methods:

- gzip — standard *gzip* program with option –9,
- PPMD+ — algorithm PPMD+ (the results come from Reference [15]),
- B97 – version of PPM algorithm (PPMD*XSM) proposed by Bunton [25],

Table 7: Comparison of compression algorithms for Calgary Corpus.

| file | size | gzip | PPMD+ | B97 | CTW | VW98 | BW94 | F96 | BKS98 | D99 |
|---|---|---|---|---|---|---|---|---|---|---|
| bib | 111261 | 2.509 | 1.862 | 1.786 | 1.79 | 1.714 | 2.07 | 1.95 | 1.93 | 1.904 |
| book1 | 768771 | 3.250 | 2.303 | 2.184 | 2.19 | 2.150 | 2.49 | 2.39 | 2.33 | 2.317 |
| book2 | 610856 | 2.700 | 1.963 | 1.862 | 1.87 | 1.820 | 2.13 | 2.04 | 2.00 | 1.983 |
| geo | 102400 | 5.345 | 4.733 | 4.458 | 4.46 | 4.526 | 4.45 | 4.50 | 4.27 | 4.221 |
| news | 377109 | 3.063 | 2.355 | 2.285 | 2.29 | 2.210 | 2.59 | 2.50 | 2.47 | 2.450 |
| obj1 | 21504 | 3.839 | 3.728 | 3.678 | 3.68 | 3.607 | 3.98 | 3.87 | 3.79 | 3.737 |
| obj2 | 246814 | 2.628 | 2.378 | 2.283 | 2.31 | 2.245 | 2.64 | 2.46 | 2.47 | 2.427 |
| paper1 | 53161 | 2.791 | 2.330 | 2.250 | 2.25 | 2.152 | 2.55 | 2.46 | 2.44 | 2.411 |
| paper2 | 82199 | 2.887 | 2.315 | 2.213 | 2.21 | 2.136 | 2.51 | 2.41 | 2.39 | 2.369 |
| pic | 513216 | 0.817 | 0.795 | 0.781 | 0.79 | 0.764 | 0.83 | 0.77 | 0.75 | 0.741 |
| progc | 39611 | 2.678 | 2.363 | 2.291 | 2.29 | 2.195 | 2.58 | 2.49 | 2.47 | 2.446 |
| progl | 71646 | 1.805 | 1.677 | 1.545 | 1.56 | 1.482 | 1.80 | 1.72 | 1.70 | 1.678 |
| progp | 49379 | 1.812 | 1.696 | 1.531 | 1.60 | 1.460 | 1.79 | 1.70 | 1.69 | 1.665 |
| trans | 93695 | 1.611 | 1.467 | 1.325 | 1.34 | 1.256 | 1.57 | 1.50 | 1.47 | 1.448 |
| Avg | 3141622 | 2.695 | 2.283 | 2.177 | 2.19 | 2.123 | 2.43 | 2.34 | 2.30 | 2.271 |

- CTW — Context Tree Weighting method proposed by Willems, Shtarkov and Tjalkens [26] (the results come from Reference [9]),
- VW98 — switching method, algorithm presented by Volf and Willems [27, 28],
- BW94 — original Burrows-Wheeler algorithm [1],
- F96 — BWT-based algorithm proposed by Fenwick [23],
- BKS98 — BWT-based algorithm proposed by Balkenhol, Kurtz and Shtarkov [9],
- D99 — BWT-based algorithm proposed in this paper.

The average compression ratio obtained with our algorithm (2.271) is worse than the results achieved for the following algorithms: PPMD*XSM, CTW and VW98, for which it equals 2.177, 2.190 and 2.123, respectively. Note however, that these compression algorithms have much larger time and memory complexity. If the comparison is restricted to the methods of a similar complexity, *gzip* and all BWT-based algorithms, then our algorithm achieves the best results for all files from the Calgary Corpus. The average compression ratio is better by more than 1% as compared to the best BWT-based algorithm (BKS98).

The second experiment was carried out using the files from the Canterbury Corpus [29]. This corpus is fairly new, therefore not all the methods have been evaluated for it. So we had to choose a different set of algorithms for comparison. The results are collected in table 8. Each column contains the compression ratios for the following methods:

- gzip — standard *gzip* program with option –9,
- PPMD+ — PPMD+ algorithm (the results were obtained with Teahan's program [30]),

Table 8: Comparison of compression algorithms for Canterbury Corpus.

| file | size | gzip | PPMD+ | bred3 | bzip 0.21 | BKS98 | D99 |
|---|---|---|---|---|---|---|---|
| alice29 | 152089 | 2.850 | 2.183 | 2.402 | 2.253 | 2.21 | 2.181 |
| asyoulik | 125179 | 3.120 | 2.467 | 2.717 | 2.513 | 2.48 | 2.452 |
| cp | 24603 | 2.594 | 2.261 | 2.483 | 2.444 | 2.42 | 2.389 |
| fields | 11150 | 2.249 | 2.036 | 2.135 | 2.089 | 2.09 | 2.049 |
| grammar | 3721 | 2.670 | 2.361 | 2.610 | 2.546 | 2.54 | 2.470 |
| kennedy | 1029744 | 1.629 | 1.344 | 1.399 | 0.915 | 0.92 | 0.659 |
| lcet10 | 426754 | 2.707 | 1.927 | 2.156 | 1.999 | 1.96 | 1.938 |
| plrabn12 | 481861 | 3.225 | 2.324 | 2.612 | 2.394 | 2.35 | 2.333 |
| ptt5 | 513216 | 0.817 | 0.795 | 0.843 | 0.770 | 0.75 | 0.741 |
| sum | 38240 | 2.672 | 2.624 | 2.764 | 2.609 | 2.57 | 2.506 |
| xargs | 4227 | 3.320 | 2.945 | 3.193 | 3.132 | 3.12 | 3.060 |
| Small files | 2810784 | 2.532 | 2.115 | 2.301 | 2.151 | 2.13 | 2.071 |
| bible | 4047392 | 2.326 | 1.596 | 1.705 | 1.652 | 1.62 | 1.509 |
| e.coli | 4638690 | 2.240 | 1.945 | 2.123 | 2.132 | 2.00 | 2.007 |
| world192 | 2473400 | 2.333 | 1.491 | 1.600 | 1.573 | 1.54 | 1.334 |
| Large files | 11159482 | 2.300 | 1.677 | 1.809 | 1.786 | 1.72 | 1.617 |
| All files | 13970266 | 2.482 | 2.021 | 2.196 | 2.073 | 2.04 | 1.973 |

- bred3 — program by Wheeler [31],
- bzip 0.21 — program by Seward [32]; it has the same compression performance as Fenwick method (the results were obtained from Reference [15]),
- BKS98 — BWT-based algorithm proposed by Balkenhol, Kurtz and Shtarkov [9],
- D99 — BWT-based algorithm presented in this paper.

The compression ratios obtained with our algorithm are the best among Burrows-Wheeler transform based algorithm, what was also the case for the Calgary Corpus. If the results are compared with PPMD+ algorithm, it can be seen that the compression ratio is worse for seven files from the corpus, and better for the remaining seven files. Also the average compression ratio of our algorithm is much better (partly because of a very good compression results obtained for the `kennedy` file). The large difference between the compression ratios for the files of large sizes (> 900kB) may be caused by the fact that we have applied the BWT to the whole files, whereas bzip 0.21 and BKS98 were divided into the 900kB-sized pieces which were compressed separately.

It was very hard to directly compare the speed of the algorithms, because of various hardware used by the authors of the algorithms. Thus it has been decided to compare the speed[2] of compression and

[2]Speed tests were carried out on a computer equipped with an AMD K6-2 350MHz processor, 64MB RAM, running under

Table 9: Comparison of compression times (in seconds) for Calgary Corpus.

| | | gzip | | D99 | |
|---|---|---|---|---|---|
| file | size | ctime | dtime | ctime | dtime |
| bib | 111261 | 0.20 | 0.02 | 0.42 | 0.20 |
| book1 | 768771 | 2.20 | 0.09 | 3.93 | 1.77 |
| book2 | 610856 | 1.18 | 0.07 | 2.90 | 1.22 |
| geo | 102400 | 0.57 | 0.02 | 0.63 | 0.42 |
| news | 377109 | 0.49 | 0.04 | 1.79 | 0.84 |
| obj1 | 21504 | 0.03 | <0.01 | 0.11 | 0.07 |
| obj2 | 246814 | 0.57 | 0.03 | 1.13 | 0.55 |
| paper1 | 53161 | 0.06 | 0.01 | 0.19 | 0.11 |
| paper2 | 82199 | 0.16 | 0.01 | 0.31 | 0.18 |
| pic | 513216 | 1.22 | 0.03 | 2.24 | 0.55 |
| progc | 39611 | 0.05 | 0.01 | 0.14 | 0.08 |
| progl | 71646 | 0.14 | 0.01 | 0.26 | 0.11 |
| progp | 49379 | 0.09 | 0.01 | 0.17 | 0.08 |
| trans | 93695 | 0.09 | 0.01 | 0.34 | 0.14 |
| Total | 3141622 | 7.05 | 0.36 | 14.56 | 6.32 |

decompression with the standard *gzip* program (see table 9, 10)[3]. For the Calgary Corpus files *gzip* was 2.1 times faster than our algorithm. It proved the opposite for the Canterbury Corpus: our compression was twice as fast as the standard program. The slow compression of *gzip* for the files `kennedy` and `e.coli` gave such result. For both corpora the decompression speed of D99 algorithm is $2.3 - 2.5$ times faster than the speed of compression. The algorithm benefits from the application of the inverted BWT performed during the decompression, which is done in $O(n)$ time. The speed of decompression of the proposed algorithm is much worse than the speed of decompression of the *gzip* program. This is because of the relatively slow process of arithmetic decoding, which is carried out during decompression.

# 7 Conclusions

The works known from the literature regarding the compression algorithms based on the Burrows-Wheeler Transform gave very promising results. The achieved compression ratios were only slightly worse than those obtained for the best known methods. The time and memory complexities of the BWT-based algorithms are significantly lower than for other algorithms.

In this paper several modifications to the BWCA were proposed. They allowed us to obtain the best compression ratios among the BWT-based methods. It seems that applying even more sophisticated encoding in place of the MTF-1 may result in further improvements of the compression ratio. The structure of $x^{mtf}$

control of Linux. Program was written in C++ and compiled with gcc 2.95 and option –O3. The times were obtained using the standard command *time* and are meant as the *user time*.

[3]Similarly to the previous case, *gzip* was executed with option –9.

Table 10: Comparison of compression times (in seconds) for Canterbury Corpus.

| | | gzip | | D99 | |
|---|---|---|---|---|---|
| file | size | ctime | dtime | ctime | dtime |
| alice29 | 152089 | 0.38 | 0.02 | 0.62 | 0.31 |
| asyoulik | 125179 | 0.28 | 0.02 | 0.51 | 0.26 |
| cp | 24603 | 0.02 | <0.01 | 0.09 | 0.05 |
| fields | 11150 | 0.01 | <0.01 | 0.04 | 0.02 |
| grammar | 3721 | <0.01 | <0.01 | 0.02 | 0.01 |
| kennedy | 1029744 | 21.17 | 0.08 | 5.38 | 2.16 |
| lcet10 | 426754 | 0.94 | 0.05 | 1.94 | 0.84 |
| plrabn12 | 481861 | 1.79 | 0.05 | 2.39 | 1.09 |
| ptt5 | 513216 | 1.22 | 0.03 | 2.24 | 0.55 |
| sum | 38240 | 0.19 | 0.01 | 0.17 | 0.08 |
| xargs | 4227 | <0.01 | <0.01 | 0.02 | 0.01 |
| Small files | 2810784 | 26.00 | 0.26 | 13.42 | 5.38 |
| bible | 4047392 | 15.82 | 0.36 | 21.39 | 7.61 |
| e.coli | 4638690 | 103.17 | 0.44 | 25.28 | 11.98 |
| world192 | 2473400 | 4.36 | 0.22 | 12.69 | 4.20 |
| Large files | 11159482 | 123.35 | 1.02 | 59.36 | 23.79 |
| Total | 13970266 | 149.35 | 1.28 | 72.78 | 29.17 |

sequence is not completely known, hence better models for this sequence can be most likely found. A certain improvement can be also achieved by adaptive choices of parameters $t^0_{max}$ and $t^2_{max}$ for different contexts during compression. Unfortunately it is not clear how to change them. In our paper we have not used a method suggested in Reference [14]. Chapin and Tate improved the compression ratio but the benefit seemed unsignificant. Furthermore they obtained best results for the Calgary Corpus using several different methods, and it is not obvious which one to choose in a particular case without checking every possible option.

The speed of the compression with the proposed algorithm is also quite good. For most files it was half as fast as *gzip*, but for some it was much faster. Improvements in overall speed may be achieved when faster algorithms for construction of suffix trees are found. The speed of the compression and decompression can be also significantly improved using the Huffman coder rather than the arithmetic coder. However this modification would decrease the compression ratio.

It was already mentioned that the BWT relies on grouping characters which occur in similar contexts. In this sense the BWCA behaves in a similar way as the PPM algorithms. Unfortunately in contrast to the PPM algorithms, during the BWCA execution we do not know where the change in the context takes place. As the result it is hard to estimate the probabilities with a comparable quality to the PPM algorithms. Therefore the compression ratios achieved by the best PPM algorithms should always be slightly better than even the best versions of the BWCA.

## Acknowledgements

The research was carried out under the Silesian University of Technology Research Project No. BK-245/RAu2/99. The author wishes to thank N. Jesper Larsson for the source code of the suffix tree creation algorithm and John Carpinelli, Alistair Moffat, Radford Neal, Wayne Salamonsen, Lang Stuiver and Ian Witten for the source code of the arithmetic coding algorithm which were both used in the experiments. The author also wishes to thank Zbigniew J. Czech, Ryszard Maceluch, Marcin Ciura and Jacek Stolarczyk for reviewing the manuscript and suggesting improvements.

## References

[1] Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. *Digital Equipment Corporation (SRC Research Report 124)* 1994. ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip.

[2] Bentley JL, Sleator DD, Tarjan RE, Wei VK. A locally adaptive data compression scheme. *Communications of ACM* 1986; **29**(4):320–330.

[3] Balkenhol B, Kurtz S. Universal data compression based on the Burrows and Wheeler Transformation theory and practice. *Sonderforschungsbereich: Diskrete Strukturen in der Mathematic, Universitat Bielefeld (Technical Report 98-069)* 1998. http://www.mathematik.uni-bielefeld.de/sfb343/preprints.

[4] Arimura M, Yamamoto H. Asymptotic optimality of the block sorting data compression algorithm. *IEICE Trans. Fundamentals* 1998; **E81-A**(10):2117–2122.

[5] Arnavut A, Magliveras SS. Block sorting and compression. In *Proceedings of the IEEE Data Compression Conference* 1997; 181–190.
http://helios.unl.edu/~spyros/dcc_v5.ps.

[6] Effros M. Universal lossless source coding with the Burrows Wheeler Transform. In *Proceedings of the Data Compression Conference* 1999; 178–187.

[7] McCreight EM. A space-economical suffix tree construction algorithm. *Journal of the ACM* 1976; **23**(2):262–272.

[8] Ukkonen E. On-line construction of suffix tress. *Algorithmica* 1995; **14**(3):249–260.

[9] Balkenhol B, Kurtz S, Shtarkov YM. Modifications of the Burrows and Wheeler data compression algorithm. In *Proceedings of the IEEE Data Compression Conference* 1999; 188–197.
http://www.mathematik.uni-bielefeld.de/ahlswede/pub/balkenhol/dcc99.ps.gz.

[10] Kurtz S. Reducing the space requirement of suffix trees. *Software–Practice and Experience* 1999; **29**(13):1149–1171.

[11] Manber U, Myers EW. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing* 1993; **22**(5):935–948.

[12] Bentley JL, Sedgewick R. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms* 1997; 360–369.

[13] Larsson NJ, Sadakane K. Faster suffix sorting. *Dept of Computer Science, Lund University, Sweden (Technical Report LU-CS-TR:99-214)* 1999.
http://www.cs.lth.se/~jesper/ssrev-tr.pdf.

[14] Chapin B, Tate SR. Higher compression from the Burrows-Wheeler Transform by modified sorting. In *Proceedings of the IEEE Data Compression Conference* 1998; 532.
http://www.cs.unt.edu/~srt/papers/bwtsort.ps.

[15] Fenwick P. Block sorting text compression. In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia* 1996; 193–202.
ftp://ftp.cs.auckland.ac.nz/out/peter-f/ACSC96paper.ps.

[16] Fenwick P. Block sorting text compression — final report. *The University of Auckland, Department of Computer Science (Technical Report 130)* 1996.
ftp://ftp.cs.auckland.ac.nz/pub/peter-f/TechRep130.ps.

[17] Schindler M. A fast block-sorting algorithm for lossless data compression. In *Proceedings of the IEEE Data Compression Conference* 1997; 469.
http://www.apm.tuwien.ac.at/~michael/papers/dcc97prc.ps.

[18] Albers S. Improved randomized on-line algorithms for the list update problem. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms* 1995; 412–419.

[19] Albers S, Mitzenmacher M. Average case analysis of list update algorithms, with applications to data compression. *Algorithmica* 1998; **21**(3):312–329.

[20] Bell TC, Cleary JG, Witten IH. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[21] Krichevsky RE, Trofimov VK. The performance of universal encoding. *IEEE Transactions on Information Theory* 1981; **27**:199–207.

[22] Moffat A, Neal RM, Witten IH. Arithmetic coding revisited. *ACM Transactions on Information Systems* 1998; **16**(3):256–294.

[23] Fenwick P. The Burrows-Wheeler Transform for block sorting text compression: principles and improvements. *The Computer Journal* 1996; **39**(9):731–740.

[24] Cleary JG, Teahan WJ, Witten IH. Unbounded length context for PPM. In *Proceedings of the IEEE Data Compression Conference* 1995; 52–61.
http://www.cs.waikato.ac.nz/~wjt/papers/DCC95a.ps.gz.

[25] Bunton S. Semantically motivated improvements for PPM variants. *The Computer Journal* 1997; **40**(2/3):76-93.

[26] Willems FMJ, Shtarkov YM, Tjalkens TJ. The context tree weighting method: basic properties. *IEEE Transactions on Information Theory* 1995; **41**:653–664.
http://ei1.ei.ele.tue.nl/~frans/ctw1.ps.

[27] Volf PAJ, Willems FMJ. Switching between two universal source coding algorithms. In *Proceedings of the IEEE Data Compression Conference* 1998; 491–500.
http://ei1.ei.ele.tue.nl/~paul/p_and_w/dcc98.zip.

[28] Volf PAJ, Willems FMJ. The switching method: elaborations. In *Proceedings of the 19th Symposium on Information Theory in the Benelux* 1998; 13–20.
http://ei1.ei.ele.tue.nl/~paul/p_and_w/ben98.ps.zip.

[29] Arnold R, Bell T. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of the IEEE Data Compression Conference* 1997; 201–210.
http://corpus.canterbury.ac.nz//report.ps.gz.

[30] Teahan B. The PPMD+ program.
http://www.cs.waikato.ac.nz/~wjt/software/ppm.tar.gz [1996].

[31] Wheeler D. The bred3 program.
ftp://ftp.cl.cam.ac.uk/users/djw3 [1997].

[32] Seward J. The bzip0.21 program.
http://www.muraroa.demon.co.uk [1997].