# Modifications of the Burrows and Wheeler Data Compression Algorithm*

*Bernhard Balkenhol*†
Fakultät für Mathematik
Universität Bielefeld
Postfach 100 131
33501 Bielefeld
Germany

*Stefan Kurtz*‡
Technische Fakultät
Universität Bielefeld
Postfach 100 131
33501 Bielefeld
Germany

*Yuri M. Shtarkov*§
Institute for Problems on
Information Transmission
100 447, Moscow, GSP-4
B. Karetnyi per. 19
Russia

## 1 Introduction

In 1994 Burrows and Wheeler [3] described a universal data compression algorithm (BW-algorithm, for short) which achieved compression rates that were close to the best known compression rates. Due to it's simplicity, the algorithm can be implemented with relatively low complexity. Fenwick [5] described ideas to improve the efficiency (i.e. the compression rate) and complexity of the BW-algorithm. He also discusses relationships of the algorithm with other compression methods. Schindler [12] proposed a Burrows and Wheeler Transformation (BWT, for short) that is based on a limited ordering. This speeds up the algorithm for compression, but slows it down for decompression and slightly decreases the efficiency. Larsson [8] describes relationship of the BWT with suffix trees and with context trees. Sadakane [11] suggests a method to compute the BWT faster, and compares it to other methods. Recently Balkenhol and Kurtz [1] gave a thorough analysis of the BWT from an information theoretic point of view. They described implementation techniques for data compression algorithms based on the BWT, and developed a program with a better compression rate.

In this paper we improve upon these previous results on the BW-algorithm. Based on the context tree model, we consider the specific statistical properties of the data at the output of the BWT. We describe six important properties, three of which have not been described elsewhere. These considerations lead to modifications of the coding method, which in turn improve the coding efficiency. We shortly describe how to compute the BWT with low complexity in time and space, using suffix trees in two different representations. Finally, we present experimental results about the compression rate and running time of our method, and compare these results to previous achievements. More references on the methods described in this paper can be found in [1, 5].

*To appear in Proceedings of the IEEE Data Compression Conference 1999

†Email: bernhard@mathematik.uni-bielefeld.de

‡Email: kurtz@techfak.uni-bielefeld.de, work partially supported by DFG-grant Ku 1257/1-1

§Email: shtarkov@iitp.ru, Work done while visiting the Fakultät für Mathematik, Universität Bielefeld, partially supported by DFG-Sonderforschungsbereich 343: Diskrete Strukturen in der Mathematik

# 2 Context Tree Models of Sources

Let $A$ be the discrete alphabet of $\alpha$ symbols, $\alpha \geq 2$; $x^k$, $x_i \in A$, be the first $k$ symbols of the message $x^n$; $p(x^k|\omega)$ be the probability of the occurrence of $x^k$ at the output of the source $\omega$ and $\varphi = \{\varphi(x^n),\, x^n \in A^n\}$ be a uniquely decodable (arithmetic) code for sequences of arbitrary (in particular, unknown beforehand) length $n$. The codeword lengths satisfy the inequality $|\varphi(x^n)| = -\log q(x^n|\varphi) \leq -\sum_{k=1}^{n} \log \vartheta(x_k|x^{k-1}, \omega) + 1$, where $|z|$ is the length of the sequence $z$ or the cardinality of the set $z$, $\log(.) = \log_2(.)$, $\{q(x^n)\}$ is a coding probability distribution, which is described by conditional probabilities $\{\vartheta(x_k|x^{k-1}, \omega),\ x^k \in A^k, k = 1, 2, \ldots\}$. The choice $q(x^n) = p(x^n|\omega)$ is the optimal one. But if the statistics of the source is unknown, then the universal coding approach for a given source model or for some set of source models can be used. Below we shall refer to the important set of *Context Tree* (CT) models.

A finite memory CT-source $\omega$ is defined by a proper and complete set $S$ of contexts (sequences over the alphabet $A$) of length $d = |s| \leq D$, by the set of conditional probability distributions $\{\Theta_s, s \in S\} = \{\{\theta(a|s), a \in A\}, s \in S\}$, and by the probability distribution for the first $D$ symbols of the message. Completeness and properness of a set $S$ mean that exactly one context exists for any $x^k \in A^k, k \geq D$, i.e. the equality $x_k \ldots x_{k-d+1} = s_k \in S$ holds for exactly one $d \leq D$. Then the probability $p(x^n|\omega)$, divided by the probability of the first $D$ symbols, is equal to the product of the conditional probabilities $\theta(x_{k+1}|s_k)$, $k \geq D$, or (in other words) to the product of the probabilities $p(x^k(s)|\Theta_s)$ over all $s \in S$, where $x^k(s)$ is the subsequence of *independent* symbols from $x^k$ occurring after context $s$. Thus $S$ is a model of the CT-source which can be represented as a set of leaves (contexts) of a complete and proper $\alpha$-ary tree $T_S$ (and vice versa).

Any CT-source can be described by a Markov chain of order $D' \geq D$ with a *larger number* $K$ of "free" parameters (values of conditional probabilities). Since the cumulative (per message) redundancy of universal (relative to the values of free parameters) coding is proportional to $K \log n$, the decreasing of $K$ is important for decreasing the coding redundancy under the condition that *it does not decrease* the accuracy of the source description. CT-models satisfy the last condition: their structure allows to exploit the fact that the "actual" lengths of the contexts are *different*.

Usually the number of different symbols that occur after a context, decreases with the length of the context. Consider for example english text: if $x_k$ is a blank and $x_{k-1}$ is a period, then with high probability the next symbol will be *any* capital letter independent of $x_{k-2}$, $x_{k-3}$, etc. (the actual length of the context is two). Of course, exceptions exist: for example, if $x_k = q$, then almost surely $x_{k+1} = u$ (although the actual length of the context $q$ is one).

Thus some paradoxical situations occur: the shorter the contexts, the smaller the number $K$ of free parameters and the coding redundancy (see above), and the larger the uncertainty relative to the next symbol and the coding rate. In fact, only rather short contexts influence the coding rate, and increasing $D$ above some threshold does not improve the coding rate. Therefore we shall use conventional terms "bad" and "good" for short and long contexts, respectively, although contexts of almost all lengths exist, as well as exceptions of the kind mentioned above.

## 3 The BW-algorithm and CT-Models

The BW-algorithm [3] consists of three phases:

1. The Burrows and Wheeler transformation (BWT) $x^n \to y^n$ is the reordering (permutation) of the symbols of $x^n$ according to the following rule. All $n$ cyclic shifts of $x^n$ (including $x^n$) are lexicographically ordered, and the sequence of the last symbols of the ordered shifts is $y^n$. It is easy to show that this is a one-to-one mapping if the position of $x^n$ in the sequence of shifts is described. For more details on the transformation, we refer the reader to [1, 3].

2. Move-to-Front transformation (MTF) $y^n \to z^n$ of the sequence $y^n$ into the sequence of numbers $z^n = z_1 \ldots z_n$, $0 \leq z_i < \alpha$ in the following way: If $y_{k+1} = a \in A$ and $z_k(a)$ is the number for $a$ after $k$ steps, then $z_{k+1}(a) = 0$ and $z_{k+1}(b) = z_k(b) + 1$ for all symbols $b$ with $z_k(b) < z_k(a)$. For any given initial numbering of symbols (known to coder and decoder) MTF is a one-to-one mapping.

3. Noiseless coding of $z^n$ (including run length coding of runs of zeroes).

If we apply the BWT to the reverse $x_n \ldots x_1$ of $x^n$, then the first $k$ symbols of it's periodical shift $x_k \ldots x_1 x_n \ldots x_{k+1}$ form the *context of maximal length* for $x_{k+1}$, i.e. for the last element of this shift (ignoring symbols $x_n \ldots x_{k+2}$). Therefore the BWT of any message corresponds to the lexicographically ordered (not necessarily complete) $\alpha$-ary tree $T^*$, describing the ordered set of contexts, and $y_i$ is the only symbol generated in the $ith$ leaf of $T^*$. Then the connection of the BW-algorithm and PPM*, as mentioned in [5], is clear. And the limited ordering, proposed in [12], corresponds to the context tree $T(D)$ for a Markov chain of order $D$, $y^n = x^n(s_1), x^n(s_2), \ldots$ (see Section 2) and such a modified BW-algorithm corresponds to PPM [4] (see also [5]).

As any one-to-one mapping, the BWT does not change the entropy of the source or the probability of the message, but the (relatively) low complexity of the BWT makes it very attractive for data compression programs. Therefore it is important to look for an efficient coding of $y^n$, taking into account the difference of the probability distributions for $x^n$ and $y^n$ over $A^n$. But for studying the properties of $y^n$, we have to know at least some properties of $x^n$. The following properties of $y^n$ correspond to the CT-model of $x^n$ which is consistent with real input sequences (see Section 2).

**Property 1:** $y^n$ is the sequence of *independent* symbols over $A$ with *variable* probabilities of occurrence. It corresponds to an *infinite memory* of the source generating $y^n$, although the original CT-source has a restricted depth of memory.

**Property 2:** $y^n$ consists of "good" and "bad" fragments corresponding to good and bad contexts, respectively. The probabilities of occurrence of symbols do *almost* not change inside the same fragment (in fact, it can be an undetectable concatenation of "close" fragments), but it can change essentially between two fragments.

**Property 3:** The statistics of the fragments (i.e. the sets of different symbols in the fragments) are different.

**Property 4:** The number of different symbols in a fragment usually decreases with the actual length of the corresponding context (in particular, most of the good fragments consist of repetitions of one symbol and this is one of the reasons for using MTF and run length coding as it was originally proposed in [3]). Therefore the method of *multialphabet* coding, which allows to adapt to an unknown subset of symbols in the fragments, should be used; MTF and grouping of numbers, as e.g. proposed in [5], define such a coding method.

**Property 5:** The longer the common prefix of two contexts (the "closer" they are), the smaller the difference of the sets of symbols generated at any of these contexts. This is one more reason for applying MTF (and, in fact, this is the basis of PPM).

**Property 6:** With an increasing message length at the output of any given CT-source, the number of fragments slightly increases (because for a short message length some subsequences $x^n(s)$ (see Section 2) are empty). Thus the (average) length of the fragments grows almost linearly with the growing message length.

Note that properties 1, 3, and 5 were already discussed in [1]. The formulation of Properties 2, 4, and 6 is an important contribution of this section.

Usually during sequential universal coding, the statistics of only the previous part of the message is used (it is known that the pre-coding description of the statistics of the message does not give us an advantage). Therefore the continuous jumping from one fragment to another one (typical for context-based coding) does not decrease the efficiency of coding. And knowledge of the "current" context can be used very efficiently (see, for example, PPM). Thus the BWT does not give us some additional advantages (for increasing efficiency) but results in the loss of knowledge about the context of the coded symbol. So we may suppose that *the efficiency of the BW-algorithm can not reach the efficiency of the best context-based algorithms* (such supposition was formulated in [5] as well). Nevertheless because the best context based algorithms require considerably more space and time than the BW-algorithm, it is desirable to increase the efficiency of the latter as much as possible.

# 4 Increasing the Efficiency of the BW-algorithm

In this section we show how to exploit the properties from Section 3 to increase the efficiency of the BW-algorithm. The general scheme is related to the approaches described in [1, 3, 5], but there are some important differences, mentioned below.

**Alphabet Encoding:** In contrast to [3, 5], we encode the set $A' \subseteq A$ of symbols that *really* occur in the input sequence. This is done very efficiently, by the method described in [1]. In the sequel, let $M$, $M \leq \alpha$, be the size of $A'$. Note that encoding $A'$ reduces the number of free parameters.

**Modification of MTF:** MTF is defined by the following modified rules: if the next symbol has the current number $z$, then after its coding we change the number as follows: if $z > 1$ then shift $z$ to position 1 else shift $z$ to position 0. In [5, 12] different modifications of MTF are discussed.

**Grouping of Symbols:** The grouping of "close" numbers $z$ (after MTF) improves the multialphabet properties of the BW-algorithm. This is important for coding of different fragments: it improves the properties of the universal coding algorithms in practice. Grouping was originally proposed (with uniform coding of all elements inside the group) for picture compression [6]. See also the theoretical study of [15]. In [5] and [1] grouping was also used, but in a different way.

The grouping is done as follows: we transform the sequence $z^n$ of numbers at the output of MTF into a ternary sequence $z_1^n$ over the alphabet $\{0,1,2\}$ and into a sequence $z_2^{n'}$ over the alphabet $\{2,\ldots,M-1\}$. $z_1^n$ is obtained from $z$ by substituting all $z_k \geq 2$ by 2, i.e. we group all the numbers $\geq 2$. $z_2^{n'}$ is the subsequence of all $z_k \geq 2$ in $z^n$. Two different methods were used for coding $z_1^n$ and $z_2^{n'}$:

**Coding of $\mathbf{z_1^n}$:** We use the ternary sequence as a Markov chain of order 3 as a well-known universal coding scheme. This means that we implement the arithmetic coding with conditional probabilities $\vartheta(z|s(z_1^k)) = (\tau(z|s(z_1^k)) + \frac{1}{2})/(k+\frac{3}{2})$ where $z \in \{0,1,2\}$, $s(z_1^k) = z_k^{(1)} z_{k-1}^{(1)} z_{k-2}^{(1)}$ are the last three numbers of $z_1^k$, and $\tau(z|s)$ is the number of occurrences of $z$ after the state $s$ in $z_1^k$. We use this Markov chain approach to obtain information about where we are in the fragments, i.e. whether we are inside a good fragment, on the boundary of two fragments, or inside a bad fragment. The higher the order $d$ of the Markov chain, the more information we have about the situation. But simultaneously, the statistics $\tau(z|s)$ become more and more "poor" and the redundancy of universal coding grows proportionally with the number $3^d$ of different states $s$. Therefore $d = 3$ is a reasonable choice.

**Coding of $\mathbf{z_2^{n'}}$:** We group $\{2,\ldots,M-1\}$ into disjoint subsets $\{2\}$, $\{3,\ldots,4\}$, $\{5,\ldots,8\},\ldots,\{65,\ldots,128\},\{129,\ldots,255\}$ except, if $129 \leq M < 255$, then we form the group $\{65,\ldots,M-1\}$, and if $65 \leq M < 128$ (the typical case), then we form the group $\{33,\ldots,M-1\}$, etc. Thus usually the number $\nu$ of groups is $\leq 7$. A similar grouping scheme was used in [5] and in [1]. Any number $a = z \in z_2^{n'}$ is encoded with arithmetic coding and the conditional probabilities

$$\vartheta(a|z^k) = \frac{T_i(z^k)+1/2}{T(z^k)+\nu/2}\,\frac{t(a|z^k)+1/2}{T_i(z^k)+m_i/2}, \tag{1}$$

where $t(a|z^k)$ is the number of occurrences of $a$ in $z_2^k$, $T_i(z^k)$ is the sum of $t(a|z^k)$ over all the numbers $a$ of the $ith$ group, $1 \leq i \leq \nu$, $T(z^k) = \sum_i T_i(z^k)$ is the length of $z_2^k$, and $m_i$ is the number of elements in the $ith$ group. The two factors on the right hand side of (1) describe the number of the group and the element in this group.

**Updating Scheme:** A slight improvement of the efficiency is achieved by updating the frequencies $\tau(z|s)$ and $t(a|z^k)$. All the frequencies $\tau(z|s)$ are replaced by the integer parts of $\tau(z|s)/2$ whenever one of these frequencies exceeds 50. A similar updating scheme was used for the values of $t(a|z^k)$ but with threshold 150 (the values of $T_i(z^k)$ and $T(z^k)$ are recalculated correspondingly). Updating allows to adapt (in some degree) to the change of the statistics of the encoded fragments. For a more detailed description of this updating scheme, see [1].

## 5 Complexity of the Implementation

Since the computation of the BWT dominates the space and time requirement of the BW-algorithm, we now consider how to compute the transformation efficiently.

As already observed in [3], the BWT can be computed in linear time and space. The idea is to construct the suffix tree $ST$ for the input sequence $x^n\$$ [10], where \$ is a symbol not occurring in $x^n$. The construction takes $O(n)$ time and space [10]. There is a one-to-one correspondence between the non-empty suffixes of $x^n\$$ and the leaves of $ST$. If for each node of $ST$, the edges outgoing from that node are stored appropriately, then a simple depth first traversal of $ST$ (in linear time and space) computes the lexicographic order of the non-empty suffixes of $x^n\$$. Given this order, it is trivial to compute the BWT for $x^n$ in $O(n)$ time. For more details on this method, we refer the reader to [1].

In our data compression program the suffix tree based method is used to compute the BWT of $x^n$. Suffix trees are constructed using the algorithm of McCreight [10]. Two different representations of the suffix tree were implemented:

The first representation stores the edges of the suffix tree in a linked list. Since it takes $O(M)$ time to select a certain edge outgoing from a node, this representation is computed in $O(Mn)$ time. Using the space reduction techniques of [7], about $10n$ bytes of space are required in the average case. This is a considerable improvement over previous implementation techniques for suffix trees (see [10]), which require about $19n$ bytes on average.[1]

The second representation stores the edges of the suffix tree in a hash table. This table implements a function mapping each pair $(v, a)$, consisting of a node $v$ of $ST$ and a symbol $a$, to the node $w$, whenever there is an edge from $v$ to $w$ whose edge label starts with symbol $a$. The hash table is implemented using an open addressing technique with double hashing to resolve collisions. Using the space reduction techniques of [7], the hash table representation requires about $15n$ bytes of space on average.

As shown in [7], the hash table representation of the suffix tree is computed faster than the linked list representation for long input sequences or large alphabets. Unfortunately, the hash table representation does not directly allow the depth first traversal of the suffix tree to run in linear time. As remarked in [8], the hash table can be sorted in $O(n)$, such that the edges outgoing from some node are stored in consecutive positions. However, as shown in [7], the additional sorting of the hash table is slow in practice and requires considerable amount of extra space. So the hash table representation only gives a speed advantage for large alphabets.

Based on these observations, our data compression program selects one of the two suffix tree representations, based on the size of the input alphabet: If the alphabet contains at least 200 symbols, then the hash table representation is used. In all other cases, the linked list representation is used. Note that the program described in [1] only uses the linked list representation.

[1]The assumption is that the symbols of the input sequence can be represented by one byte, and that integers occupy 4 bytes.

There are other algorithms to compute the BWT, which do not run in linear time:

- The algorithm of Manber and Myers [9] runs in $O(n \log n)$ worst case time and it requires $8n$ bytes of space. In practice it was shown to be considerably slower than a suffix tree based method, see [9, 11].

- The algorithm of Bentley and Sedgewick [2] is based on quicksort. It is space efficient ($5n$ bytes plus stack space for quicksort) and fast in the average case, but the worst case running time is $O(n^2)$. This occurs, if the input sequence contains long repeated substrings. For example, for the file *pic* of the Calgary Corpus the algorithm requires about 190 seconds.[2] This behavior rules out the Bentley-Sedgewick Algorithm to be used in our program. Note that this algorithm is used in the *bzip2*-program [13], which is also based on the BWT. However, in contrast to our program, *bzip2*, applies a run length encoding to the input sequence, before computing the BWT. So the worst case behavior of the Bentley-Sedgewick algorithm is less likely to occur in *bzip2*.

- Recently, Sadakane [11] has shown how to combine the Manber-Myers Algorithm with the Bentley-Sedgewick Algorithm, to achieve a method running in $O(n \log n)$ worst case time and using $8n$ bytes of space. Experiments in [11] show that Sadakane's algorithm is on average slightly slower than a suffix tree based sorting method implemented by Larsson.

The main advantage of these non-linear algorithms is that they use less space than a suffix tree based method. However, with the space reduction techniques of [7], this advantage has decreased considerably. So using a suffix tree based method to compute the BWT, is a reasonable choice.

# 6 Experimental Results

In a first experiment we determined the compression rate of our program in bits/byte for the files of the Calgary Corpus, and compared it to other programs. Table 1 shows the compression rate of the switching method VW98 of Volf and Willems [14], of CTW (Context Tree Weighting with PPMDE, $D = 8$), of PPMDE ($D = 5$), of *gzip* with option –9, of the program *BW94* developed by Burrows and Wheeler [3], of the program *F96* by Fenwick [5], of the program *BK98* developed by Balkenhol and Kurtz [1], and of the program *BKS98* described in this paper. The last row of the table shows the total length of the files and for each program the average compression rate. The three programs *VW98*, *CTW*, and *PPMDE* have a better compression rate than our program: While we achieve an average compression rate of 2.30, they achieve 2.12, 2.19, and 2.27, respectively. However, they require much more computational resources. If we restrict to the programs which have similar requirements in space and time (the last five columns of Table 1) then our program

[2]For *pic* our suffix tree based method requires 2.3 seconds, while the Manber Myers Algorithm takes 20.3 seconds. These numbers are for the computer mentioned in Section 6.

shows the best compression rates for most files (see the grey boxes). In particular, we have improvements over *BK98* (a predecessor of *BKS98*), except for *book1*, *book2*, *paper2*, and *pic*. The improvement for *geo* is mainly due to the fact that we reverse an input sequence, if $M = 256$. This heuristic makes sense, because if $M = 256$, then the input sequence is usually encoded in some way (e.g. object code), so that it is of advantage to look at the preceeding context of the symbols. This is achieved by reversing the input sequence, a technique which is also applied in *szip* [12].

In a second experiment we applied our program to the Canterbury Corpus (including the large files *e.coli*, *bible.txt*, and *world192.txt*). Since this corpus is rather new, compression rates were not available for *VW98*, *CTW*, *PPMDE*, *BW98*, and *F96*. Therefore, Table 2 shows the compression rates of *gzip*, of *PPM* with option *-o3* and escape method D, of *bred* (a program developed by Wheeler), of *bzip2* with option –9 (see [13]), of *szip* with block size 1.7MB (see [12]), of *BK98*, and finally of *BKS98*. Note that all these programs, except for *gzip* and *PPM*, are based on the BWT. For most files *BKS98* achieves the best compression rate. The average compression rate is 2.04. *BKS98* improves over *BK98*, except for *ptt5*, *kennedy*, and *xargs.1*. Note that for the DNA sequence *e.coli* (alphabet size 4), we exactly hit the base line figure of 2 bits per symbol. Some people prefer to split the Canterbury Corpus into two groups: the group of small files (alice29, . . . ,asyoulik) and the group of large files (the remaining). For the former group we achieve an average compression rate of 2.13 bits/byte and for the latter it is 1.72 bits/byte. For each of the large files of the Canterbury Corpus we could achieve even better compression rates by choosing a larger block size. (The results presented are for the block size of maximal 900,000 symbols.)

To demonstrate the practical relevance of our program, we measured its running time and compared it to *gzip*. Since *gzip* is available on most computers, these results allow an indirect comparison to other programs. Table 3 shows compression time (*ctime*) and decompression time (*dtime*) for *gzip* and for *BKS98* when applied to the files of the Calgary and the Canterbury Corpus. The last row gives the sums of the corresponding columns. The results were obtained on a Toshiba Notebook *460 CDT* (Pentium MMX-processor, 166 MHz, 32 MB RAM) under the operating system Linux. We used the *gcc* compiler, version 2.7.2.3 with the optimizing option –O3. Times are user times in seconds (averaged over ten runs) as reported by the *gnu time* utility. For the Calgary Corpus *gzip* achieves about 2.6 times the speed of *BKS98* for compression. *BKS98* is slightly slower than *BK98* (by factor 1.1) for the Calgary Corpus (we refer to the results of [1]). This is due to the fact that *BKS98* does not apply run length encoding and that the encoding after the BWT is more complicated in *BKS98*. For the Canterbury Corpus *BKS98* is about 1.6 times faster than *gzip*. This is because of the good performance for *e.coli* and *kennedy* (for the latter file the hash table representation of the suffix tree is used). We confirmed this speed advantage of *BKS98* over *gzip* on a different computer architecture: on a Sun-UltraSparc (143 MHz, 64 MB RAM) our program is 1.43 times faster than *gzip*, when compressing the Canterbury Corpus. For both corpora, *gzip* decompresses much faster than our program does. However, this is a general disadvantage of the BW-algorithm.

| *file* | *length* | *M* | *VW98* | *CTW* | *PPMDE* | *gzip* | *BW94* | *F96* | *BK98* | *BKS98* |
|---|---|---|---|---|---|---|---|---|---|---|
| bib | 111261 | 81 | 1.71 | 1.79 | 1.84 | 2.51 | 2.02 | 1.95 | 1.94 | 1.93 |
| book1 | 768771 | 81 | 2.15 | 2.19 | 2.30 | 3.25 | 2.48 | 2.39 | 2.31 | 2.33 |
| book2 | 610856 | 96 | 1.82 | 1.87 | 1.96 | 2.70 | 2.10 | 2.04 | 2.00 | 2.00 |
| geo | 102400 | 256 | 4.53 | 4.46 | 4.73 | 5.34 | 4.73 | 4.50 | 4.49 | 4.27 |
| news | 377109 | 98 | 2.21 | 2.29 | 2.35 | 3.06 | 2.56 | 2.50 | 2.49 | 2.47 |
| obj1 | 21504 | 256 | 3.61 | 3.68 | 3.72 | 3.84 | 3.88 | 3.87 | 3.87 | 3.79 |
| obj2 | 246814 | 256 | 2.25 | 2.31 | 2.39 | 2.63 | 2.53 | 2.46 | 2.46 | 2.47 |
| paper1 | 53161 | 95 | 2.15 | 2.25 | 2.31 | 2.79 | 2.52 | 2.46 | 2.45 | 2.44 |
| paper2 | 82199 | 91 | 2.14 | 2.21 | 2.30 | 2.89 | 2.50 | 2.41 | 2.38 | 2.39 |
| pic | 513216 | 159 | 0.76 | 0.79 | 0.81 | 0.82 | 0.79 | 0.77 | 0.74 | 0.75 |
| progc | 39611 | 92 | 2.20 | 2.29 | 2.35 | 2.68 | 2.54 | 2.49 | 2.50 | 2.47 |
| progl | 71646 | 87 | 1.48 | 1.56 | 1.66 | 1.80 | 1.75 | 1.72 | 1.71 | 1.70 |
| progp | 49379 | 89 | 1.46 | 1.60 | 1.67 | 1.81 | 1.74 | 1.70 | 1.70 | 1.69 |
| trans | 93695 | 99 | 1.26 | 1.34 | 1.44 | 1.61 | 1.52 | 1.50 | 1.48 | 1.47 |
| | 3141622 | | 2.12 | 2.19 | 2.27 | 2.70 | 2.40 | 2.34 | 2.32 | 2.30 |

Table 1: Compression rates (in bits/byte) for the Calgary Corpus

| *file* | *length* | *M* | *gzip* | *ppm* | *bred* | *bzip2* | *szip* | *BK98* | *BKS98* |
|---|---|---|---|---|---|---|---|---|---|
| alice29 | 152089 | 74 | 2.85 | 2.31 | 2.55 | 2.27 | 2.25 | 2.23 | 2.21 |
| ptt5 | 513216 | 159 | 0.82 | 0.99 | 0.82 | 0.78 | 0.82 | 0.74 | 0.75 |
| fields | 11150 | 90 | 2.24 | 2.11 | 2.17 | 2.18 | 2.19 | 2.11 | 2.09 |
| kennedy | 1029744 | 256 | 1.63 | 1.08 | 1.21 | 1.01 | 0.84 | 0.90 | 0.92 |
| sum | 38240 | 255 | 2.67 | 2.68 | 2.77 | 2.70 | 2.70 | 2.62 | 2.57 |
| lcet10 | 426754 | 84 | 2.71 | 2.19 | 2.47 | 2.02 | 2.00 | 1.97 | 1.96 |
| plrabn12 | 481861 | 81 | 3.23 | 2.48 | 2.89 | 2.42 | 2.38 | 2.36 | 2.35 |
| cp | 24603 | 86 | 2.59 | 2.38 | 2.50 | 2.48 | 2.44 | 2.43 | 2.42 |
| grammar | 3721 | 76 | 2.65 | 2.43 | 2.69 | 2.79 | 2.60 | 2.55 | 2.54 |
| xargs.1 | 4227 | 74 | 3.31 | 3.00 | 3.26 | 3.33 | 3.25 | 3.11 | 3.12 |
| asyoulik | 125179 | 68 | 3.12 | 2.53 | 2.84 | 2.53 | 2.51 | 2.49 | 2.48 |
| e.coli | 4638690 | 4 | 2.24 | 2.03 | 2.16 | 2.16 | 2.07 | 2.04 | 2.00 |
| bible | 4047392 | 63 | 2.33 | 1.66 | 2.09 | 1.67 | 1.62 | 1.63 | 1.62 |
| world192 | 2473400 | 94 | 2.33 | 1.66 | 2.24 | 1.58 | 1.60 | 1.56 | 1.54 |
| | 13970266 | | 2.48 | 2.11 | 2.33 | 2.14 | 2.09 | 2.05 | 2.04 |

Table 2: Compression rates (in bits/byte) for the Canterbury Corpus

| | | *gzip* | | *BKS98* | |
|---|---|---|---|---|---|
| *file* | *length* | *ctime* | *dtime* | *ctime* | *dtime* |
| bib | 111261 | 0.35 | 0.04 | 0.96 | 0.41 |
| book1 | 768770 | 3.64 | 0.25 | 10.29 | 3.22 |
| book2 | 610856 | 2.14 | 0.18 | 7.01 | 2.38 |
| geo | 102400 | 1.00 | 0.06 | 1.66 | 0.72 |
| news | 377109 | 0.99 | 0.12 | 5.27 | 1.56 |
| obj1 | 21504 | 0.06 | 0.02 | 0.29 | 0.14 |
| obj2 | 246814 | 1.08 | 0.08 | 3.36 | 1.02 |
| paper1 | 53161 | 0.14 | 0.03 | 0.43 | 0.20 |
| paper2 | 82199 | 0.31 | 0.04 | 0.72 | 0.31 |
| pic | 513216 | 2.95 | 0.08 | 3.42 | 1.61 |
| progc | 39611 | 0.10 | 0.03 | 0.33 | 0.15 |
| progl | 71646 | 0.26 | 0.02 | 0.50 | 0.24 |
| progp | 49379 | 0.19 | 0.03 | 0.33 | 0.17 |
| trans | 93695 | 0.20 | 0.03 | 0.67 | 0.31 |
| | 3141621 | 13.40 | 1.01 | 35.23 | 12.45 |

| | | *gzip* | | *BKS98* | |
|---|---|---|---|---|---|
| *file* | *length* | *ctime* | *dtime* | *ctime* | *dtime* |
| alice29 | 152089 | 0.69 | 0.06 | 1.52 | 0.59 |
| ptt5 | 513216 | 2.94 | 0.08 | 3.41 | 1.62 |
| fieldsc | 11150 | 0.03 | 0.02 | 0.09 | 0.06 |
| kennedy | 1029744 | 36.36 | 0.23 | 13.21 | 5.48 |
| sum | 38240 | 0.42 | 0.03 | 0.36 | 0.17 |
| lcet10 | 426754 | 1.66 | 0.13 | 4.74 | 1.66 |
| plrabn12 | 481861 | 3.30 | 0.17 | 6.19 | 2.02 |
| coptr | 24603 | 0.05 | 0.03 | 0.18 | 0.10 |
| grammar | 3721 | 0.02 | 0.01 | 0.04 | 0.03 |
| xargsman | 4227 | 0.03 | 0.01 | 0.04 | 0.03 |
| asyoulik | 125179 | 0.50 | 0.05 | 1.28 | 0.51 |
| ecoli | 4638690 | 166.49 | 1.20 | 52.13 | 19.53 |
| bible | 4047392 | 25.65 | 1.00 | 43.31 | 15.23 |
| world192 | 2473400 | 7.84 | 0.60 | 26.77 | 9.18 |
| | 13970266 | 245.97 | 3.62 | 153.28 | 56.22 |

Table 3: Running times (in seconds) for Calgary Corpus and Canterbury Corpus

**Acknowledgements:** We wish to thank Jan Åberg for providing the compression rates of the programs *CTW* and *PPMDE*.

## References


[1] B. Balkenhol and S. Kurtz. Universal Data Compression Based on the Burrows and Wheeler Transformation: Theory and Practice. Technical Report, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, Universität Bielefeld, 98-069, 1998. `http://www.mathematik.uni-bielefeld.de/sfb343/preprints/`.

[2] J. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997. `http://www.cs.princeton.edu/~rs/strings/`.

[3] M. Burrows and D.J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Systems Research Center, 1994. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html`.

[4] J.G. Cleary, W. Teahan, and I.H. Witten. Unbounded Length Contexts for PPM. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah*, pages 52–61. IEEE Computer Society Press, 1995.

[5] P. Fenwick. Block Sorting Text Compression. In *Proceedings of the 19th Australian Computer Science Conf., Melbourne, Australia, Jan. 31 - Feb. 2, 1996*, 1996.

[6] O. Franceschi, Y.M. Shtarkov, and R. Forchheimer. An Adaptive Coding Method for Still Images. In *Picture Coding Symp, PCS-88, Torini, Italy*, pages 6.5–1 – 6.5–2, 1988.

[7] S. Kurtz. Reducing the Space Requirement of Suffix Trees. Manuscript, 1998.

[8] N.J. Larsson. The Context Trees of Block Sorting Compression. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 189–198. IEEE Computer Society Press, 1998.

[9] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5):935–948, 1993.

[10] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

[11] K. Sadakane. A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation. In *Proceedings of the IEEE Data Compression Conference, Snowbird, Utah, March 30 - April 1*, pages 129–138. IEEE Computer Society Press, 1998.

[12] M. Schindler. A Fast Block-Sorting Algorithm for Lossless Data Compression. Technical report, 1996. `http://www.compressconsult.com/szip/`.

[13] J. Seward. The bzip2 program, version 0.1pl2, 1997. `http://www.muraroa.demon.co.uk`.

[14] P.A.J. Volf and F.M.J. Willems. The Switching Method: Elaborations. In *Proc. 19-th Symp. Inform. Theory in the Benelux, Veldhoven, The Netherlands, May 28-29*, pages 13–20, 1998.

[15] F.M.J. Willems. Universal Data Compression and Repetition Times. *IEEE Trans. on Inform. Theory*, **35**(1):54–58, 1989.