Improvements to the Burrows-Wheeler Compression Algorithm: After BWT Stages

JUERGEN ABEL
University Duisburg-Essen

The lossless Burrows-Wheeler Compression Algorithm has received considerable attention over recent years for both its simplicity and effectiveness. It is based on a permutation of the input sequence – the Burrows-Wheeler Transform – which groups symbols with a similar context close together. In the original version, this permutation was followed by a Move-To-Front transformation and a final entropy coding stage. Later versions used different algorithms which come after the Burrows-Wheeler Transform, since the stages after the Burrows-Wheeler Transform have a significant influence on the compression rate. This article describes improved algorithms for the run length encoding, inversion frequencies and weighted frequency count stages that follow the Burrows-Wheeler Transform. Results for compression rates are presented for different variations of the algorithm together with compression and decompression times. Finally, an implementation with a compression rate of 2.238 bps on the Calgary Corpus is introduced, which is the best result published in this field to date.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**] - *Data compaction and compression*; H1.1 [**Models and Principles**]: Systems and Information Theory - *Information theory*; *Value of information*

General Terms: algorithms, data compression

Additional Key Words and Phrases: Burrows-Wheeler Transform, text compression, block sorting, compression, move to front coding, weighted frequency count, inversion frequencies, run length encoding, entropy coding, BWT, BWCA, MTF, IF, WFC, RLE, EC, GST, SIF, AWFC

---

1. INTRODUCTION

The family of the block sorting algorithms based on the Burrows-Wheeler Transform (BWT) has grown over the past few years starting with the first implementation described by Burrows and Wheeler [1994]. Several authors have presented improvements to the original algorithm. Andersson and Nilsson have published several papers about Radix Sort, which can be used as a first sorting step during the BWT [1994, 1996, 1998]. In his final BWT research report, Fenwick described some BWT sort improvements including sorting long words instead of single bytes [1995]. Kurtz presented several papers about BWT sorting stages with suffix trees, which needed less space than other suffix tree implementations and are linear in time [1998, 1999].

Author's address: Juergen Abel, University Duisburg-Essen, Department "Communications Systems", Faculty of Engineering Sciences, Bismarckstrasse 81, D-47057 Duisburg, Germany; Email juergen.abel@acm.org

Sadakane described a fast suffix array sorting scheme in 1997 and 2000. In 1999, Larsson presented an extended suffix array sorting scheme. Based on already sorted suffices, Seward developed in 2000 two fast suffix sorting algorithms called "copy" and "cache". Itoh and Tanaka presented a fast sorting algorithm called the two stage suffix sort [1999]. Kao improved the two stage suffix sort by some new techniques which are very fast for sequences of repeat symbols [1999]. Manzini and Ferragina published in 2002 some improved suffix array sorting techniques based on the results of Seward and of Itoh and Tanaka.

Several techniques for the post BWT stages have been also published. Besides the MTF improvements from Schindler [1997], and from Balkenhol and Shtarkov [1999], an MTF replacement, called Inversion Frequencies, was introduced by Arnavut and Magliveras in 1997, and in 2000 Deorowicz presented another MTF replacement, named Weighted Frequency Count. Both MTF replacements will be described later.

Various modeling techniques for the entropy coding at the end of the compression process were presented by Fenwick [1995, 1996], Balkenhol and Shtarkov [1999] and Deorowicz [2000].

This paper concentrates on improvements of the basic stages subsequent to the BWT, with no special preprocessing for different kinds of data like text preprocessing [Grabowski, 1999; Kruse and Mukherjee, 1999; Franceschini et al., 2000; Awan et al., 2001; Isal and Moffat, 2001; Isal et al., 2002] or binary preprocessing before the BWT. Several improved variants will be presented and compared using compression rate, compression time and decompression time with other compression algorithms. Finally, a complete compression algorithm will be introduced, which uses a hybrid scheme and which achieves a high compression rate.

2. THE BURROWS-WHEELER COMPRESSION ALGORITHM

2.1 Typical scheme

A typical scheme of the Burrows-Wheeler Compression Algorithm (BWCA) is presented in Figure 1 and consists of four stages. Each stage is a transformation of the input data and reaches the output data to the next stage. The stages are processed sequentially from left to right. The first stage is the BWT. This stage sorts the data in a way that symbols with a similar context are grouped closely together. The BWT stage keeps the number of symbols during the transformation constant. The second stage is called in this article Global Structure Transformation (GST), which transforms the local context of the symbols to a global context [Balkenhol and Kurtz, 1998; Deorowicz, 2000]. A typical

representative of a GST stage is the Move-To-Front transformation (MTF), which was used by Burrows and Wheeler in their original publication and which was the first algorithm used as a GST stage in a BWCA. The MTF stage is a List Update Algorithm (LUA), which replaces the input symbols with corresponding ranking values (Bentley et al., 1986). Just like the BWT stage, a LUA stage does not alter the number of symbols. The third stage typically shrinks the number of symbols by applying a Run Length Encoding scheme (RLE). Different algorithms have been presented for this purpose, with the Zero Run Transformation (RLE0) from Wheeler found to be an efficient one [Fenwick, 1996]. The last stage is an Entropy Coding stage (EC), which compresses the symbols by using an adapted model.

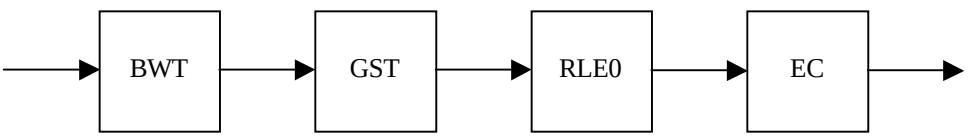

Fig. 1. Typical scheme of the Burrows-Wheeler Compression Algorithm

In order to elucidate the operation modes of the different stages, Figure 2(a) - 2(e) displays the transformed data of the input string "abracadabraabracadabra" in hexadecimal. The input data of the BWT stage is shown in Figure 2(a). As can be seen in Figure 2(b) the output data of the BWT stage contains many sequences of repeating symbols and has a local structure, i.e. symbols with a similar context form small fragments. The GST stage – in this example an MTF scheme is used – transforms the local structure of the BWT output to a global structure by using a ranking scheme according to the last recently used symbols and produces sequences of continuous zeros which are displayed in Figure 2(c). The RLE0 stage removes the zero runs in Figure 2(d) and the final EC stage produces a bit output by using an arithmetic coding scheme in Figure 2(e).

```
(a) BWT input  : 61 62 72 61 63 61 64 61 62 72 61 61 62 72 61 63 61 64 61 62 72 61
(b) BWT output : 61 72 72 64 64 61 72 72 63 63 61 61 61 61 61 61 61 61 62 62 62 62
(c) GST output : 61 72 00 65 00 02 02 00 65 00 02 00 00 00 00 00 00 00 65 00 00 00
(d) RLE0 output: 63 74 00 67 00 04 04 00 67 00 04 00 00 00 67 00 00
(e) EC output  : 00 0D 01 8D B3 FF 81 00 72 A8 E8 2B
```

Fig. 2. Transformed data of the input string "abracadabraabracadabra" by the different stages

In the following, the focus is on developing improvements, and on exploring the reordering and possible replacement of stages, which follow the BWT stage.

2.2 Definitions

For the description of the algorithms, the following notation will be used. Let A be an ordered set, called alphabet, with size $|A|$. Let $X = x_0x_1x_2\ldots x_{n-1}$ denote a sequence with length n and $x_i \in A$. The first index of a sequence is 0. Each stage has an input sequence X_{in} and an output sequence X_{out} as well as a corresponding input alphabet A_{in} and an output alphabet A_{out}. The stage processes the symbols of X_{in} and calculates the corresponding symbols of X_{out}. After finishing one stage, X_{out} of this stage will be used as X_{in} of the following stage. The maximal size for X_{in} is called the blocksize b_n. Up to the GST stage, A_{in} and A_{out} will have a bit width of 8 bits resulting in $|A_{in}| = |A_{out}| = 256$. Since some GST stages have output symbols with values greater than 255, the bit width of A_{in} and A_{out} after the GST stage will be assumed as 32 bits in order to be able to handle values greater than 255. Furthermore, the binary representation sequence of a symbol a is called B_a, for example B_4 = "100" and B_7 = "111". The compression rate and speed is measured on the Calgary Corpus [Bell et al., 1989, 1990], a standard set of files used for benchmarking compression algorithms.

2.2 Run Length Encoding

In the past, different RLE schemes were presented [Fenwick, 1996; Maniscalco, 2000, 2001]. The main function of the RLE is to support the probability estimation of the next stage. Long runs of a symbol a tend to overestimate the global symbol probability of a for fragments, where a occurs only occasionally. The result is that within these disjointed fragments, the probability value for a is too high which leads to lower compression. Balkenhol and Shtarkov name this phenomenon “the pressure of runs” [1999]. The RLE stage helps to decrease this pressure. In order to improve the probability estimation of the EC stage, the common BWCA schemes position the RLE stage directly in front of the EC stage.

One common RLE stage for BWT based compressors is the Zero Run Transformation (RLE0) from Wheeler [Fenwick, 1996]. Wheeler suggested to code only the runs of the 0 symbols and no runs of other symbols, since 0 is the symbol with the most runs. Hereto an offset of 1 is added to symbols greater than 0. The run length is incremented by one

and all bits of its binary representation except the most significant bit – which is always 1 – are stored with the symbols 0 and 1.

Some authors suggested an RLE stage before the BWT stage for speed optimization, but such a stage deteriorates the compression rate in general [Deorowicz, 2000]. Since there are sorting algorithms now known which sort the runs of symbols practically in linear time [Itoh and Tanaka, 1999; Kao, 1999; Seward, 2000; Manzini and Ferragina, 2002], there is no reason to use such a stage before the BWT stage anymore.

2.3 Inversion Frequencies

Several GST stages have been unveiled since the birth of the BWCA in 1994. Their purpose is to produce an output sequence which is more compressible by the entropy coder than the output sequence of the original MTF stage. One of these MTF replacements is the algorithm from Arnavut and Magliveras [1997], which they named Inversion Frequencies (IF). The IF algorithm is not a LUA. It produces for each symbol $a \in A_{in}$ a part sequence S_a. For each alphabet symbol a the input sequence X_{in} is scanned and if the current element of X_{in} is equal to a, the number of symbols greater than a between the current position and the last position of a is output. In order to reproduce X_{in} from the set of S_a either the frequencies of the alphabet symbols or a terminator symbol behind each S_a is needed in addition. One advantage of the IF algorithm is the fact that the part sequence of the last symbol z of the alphabet, called S_z, consists only of the symbol 0. Therefore S_z is not needed in order to reproduce the original sequence and the length of X_{out} gets smaller than the length of X_{in}. X_{out} of IF is different from X_{out} of MTF in more aspects. X_{out} of the MTF stage contains many zero runs, which represent runs of equal symbols, and these runs are equally distributed over the whole sequence. X_{out} of the IF stage consists of several part sequences S_a, one for each $a \in A_{in}$ except the last symbol z. S_a of higher symbols have typically smaller values than S_a of lower symbols, since the number of alphabet symbols, which are greater than the scanned symbol, are smaller. S_a for the last symbols of A_{in} have usually many long runs of zeros. In order to represent this behavior, Figure 3 compares the share of the zeros of the file `book1` over the file position for both the MTF stage output and for the IF stage output. For a better comparison, S_z is included in X_{out} of IF. As can be seen, the average share of zeros in the output of IF is rising towards the end of the file until it reaches 100% at the end. In the output of MTF, the average share of zeros fluctuates around 60%.

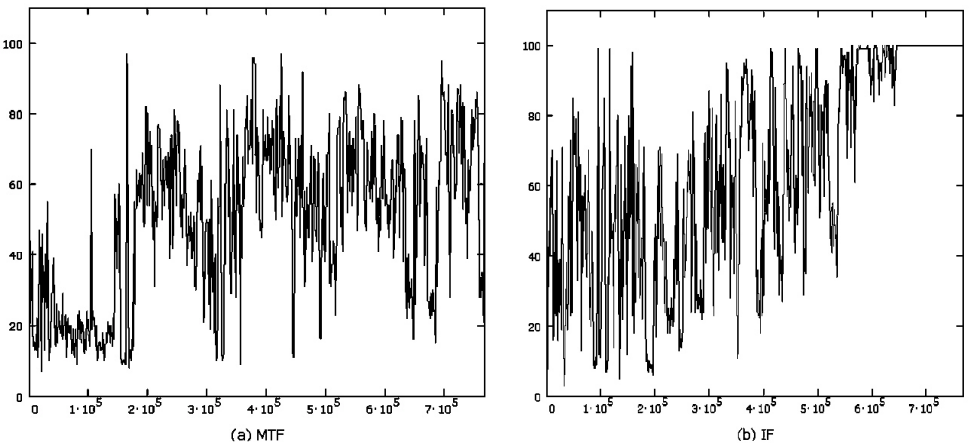

Fig. 3. Share of the zeros in percent of the (a) MTF output position and (b) IF output position of `book1`

2.4 Weighted Frequency Count

Another GST stage is the Weighted Frequency Count algorithm (WFC) presented by Deorowicz in 2002. The WFC is a representative of a LUA and is closer to MTF than to IF. It replaces the input symbol x with a corresponding ranking value r_x. The difference between WFC and MTF is the function, which calculates r_x. Inside the MTF algorithm, r_x is the index of the current input symbol x within the list L of alphabet symbols. Upon each request of x, the current index r_x of the symbol within L is output and the symbol is moved to the front of L, i.e. to index 0. Since a symbol is moved straight to the front of L without taking the former frequency distribution of the symbol into account, the MTF stage pushes many times more frequent symbols aside by less frequently used symbols. This leads to higher ranking values for frequently used symbols and hampers the compression at the EC stage, since lower values are cheaper to compress with the commonly used EC models. The WFC stage calculates r_x by a function, which takes into account the symbol frequencies and the distance of the last occurrences of x inside a sliding window of size t_{max} [Deorowicz, 2002]. More frequently used symbols get a lower r_x than less frequently used symbols, which supports the following EC probability estimation. Table I presents the average ranking values r_x of the MTF and WFC stage together with the corresponding execution times in seconds for all files of the Calgary Corpus. The average ranking values are the averages of the output sequences of the corresponding stages. The MTF and WFC stages are both performed with an RLE stage processed beforehand.

Table I. Average ranking values r_x and execution times in seconds for the MTF and WFC stage

File	MTF Average r_x	WFC Average r_x	MTF Time in Secs	WFC Time in Secs
bib	4.45	4.29	0.01	0.12
book1	3.25	2.97	0.05	0.98
book2	3.52	3.28	0.03	0.68
geo	48.74	44.01	0.03	0.91
news	5.57	5.12	0.03	0.58
obj1	34.59	32.64	0.01	0.13
obj2	18.59	17.81	0.02	0.71
paper1	4.65	4.48	0.01	0.08
paper2	3.95	3.73	0.01	0.11
pic	6.12	5.20	0.01	0.23
progc	5.66	5.48	0.01	0.06
progl	3.62	3.47	0.01	0.08
progp	4.00	3.93	0.01	0.05
trans	4.14	4.00	0.01	0.08
Average	**10.78**	**10.03**		
Sum			**0.25**	**4.80**

In all cases, the average ranking values for the WFC stage are smaller or equal than for the MTF stage, therefore the WFC output sequence is higher compressible. The drawback of the WFC stage is the high time consumption, about 20 times as high as for the MTF stage, since the function recalculates the rankings of all alphabet symbols for each symbol x of X_{in}.

2.5 Other MTF Replacements

Beside MTF, WFC and IF, there have been more GST stages published, like the MTF-1 algorithm from Balkenhol, Kurtz and Shtarkov [1999], the MTF-2 algorithm from Balkenhol and Shtarkov [1999] and the Distance Coding algorithm from Binder [Binder, 2000; Deorowicz, 2002]. The MTF-1 and MTF-2 algorithms are close to the MTF algorithm. MTF-1 moves only the symbol from the second position to the front of the list, symbols with higher positions are moved to the second position. MTF-2 differs from MTF-1, by the fact, that symbols from the second position are moved to the front of the

list only if the last ranking value was not zero, i.e. if the same symbol occurred again. The DC algorithm is more related to the IF algorithm and is based on the Interval Encoding scheme from Elias [1987]. For each symbol of the input sequence, the DC algorithm outputs the distance to the next occurrence of the same symbol. If the symbol does not occur again, a zero is output. Binder [2000] proposed three improvements to the basic algorithm. If the length of the input sequence is transmitted too, the last sequence of ending zeros is redundant. Furthermore, for calculating the distance to the next occurrence of the same symbol, only unknown symbols have to be counted. The last improvement means that if the last symbol is equal to the current symbol, nothing has to be output and DC proceeds to the next symbol. The main difference to the Interval Encoding of Elias is, that DC does not count known symbols and skips repeated symbols. Comparisons of published results [Deorowicz, 2002] with the experiences of the author of this paper indicate that the IF stage and the WFC stage tend to produce the best compression rates.

3. IMPROVEMENTS TO RUN LENGTH ENCODING

3.1 General operation

RLE is a simple and popular data compression algorithm. The sequence of length l of a repeated symbol s is replaced by a shorter sequence, usually containing one or more symbols of s, length information and sometimes an escape symbol c. RLE algorithms differ from each other mainly in three points: the threshold t, the marking of the start of a run and the coding of the length information. If l is smaller than t, the run keeps unchanged, and if l is greater or equal to t, the run is replaced. The start of a run can be indicated by a threshold run or an escape symbol c. If a threshold run is used, the start is characterized by a small sequence of s, which has a length greater or equal to t. If an escape symbol c indicates the start of a run, s is normally put behind c in order to characterize the run symbol. The escape symbol c must not be an element of A_{in} or occurrences of c have to be coded in such a manner that they are not mixed up with the start of a run. The length information l can be coded in different ways. Usually l is put directly behind the threshold run or behind s.

Maniscalco [2001] describes an algorithm which uses a variable length code and divides the length information into two parts: an exponent part e and a binary representation part B_a. The exponent part e, called the *size* of the variable length code in Maniscalco's paper, reflects the logarithm of l. The binary representation part B_a, called the *value* of the

variable length code by Maniscalco, contains the bits of the binary representation of *l*. Such a structure leads to an elegant and efficient RLE algorithm for the BWCA.

Two RLE algorithms are introduced and discussed in the next section. One is based on the variable length code of Maniscalco and is called RLE-EXP. The other algorithm is a new algorithm named RLE-BIT and is based on the idea of using two escape symbols rather than one.

3.2 A new position for the RLE stage

Gringeler had the idea to position the RLE stage directly after the BWT stage instead of in front of the EC stage [2002]. There are two reasons for the new order. Since the length of X_{out} of the RLE stage is usually smaller than the one of X_{in}, the GST stage has to process less symbols with an RLE stage in front. In addition, an RLE stage is usually faster than a GST stage, so the whole compression process becomes faster. The second reason is that the coding of the runs lowers the pressure of runs already at the GST stage and that leads usually to a better compressible GST output sequence. The compression rates for a BWCA with an RLE stage before (RLE-BIT, see Section 3.4) and an RLE after a WFC stage (RLE0, [Fenwick, 1996]) are compared in Table II. Positioning the RLE stage in front of the WFC stage achieves a compression gain of 1.3%.

Table II. Compression rates with RLE stage before and after the WFC stage in bps

File	RLE after WFC	RLE before WFC
bib	1.923	1.888
book1	2.302	2.265
book2	1.982	1.942
geo	4.180	4.190
news	2.423	2.386
obj1	3.765	3.732
obj2	2.441	2.365
paper1	2.423	2.382
paper2	2.367	2.332
pic	0.703	0.709
progc	2.454	2.419
progl	1.681	1.659
progp	1.690	1.659
trans	1.459	1.440
Average	**2.271**	**2.241**

3.3 The RLE-EXP algorithm

Since the escape symbol *c* is different from the run symbol *s*, *c* disrupts the local symbol context. Therefore, the usage of a threshold run instead of an escape symbol normally leads in BWCA implementations to better results [Maniscalco, 2001]. The following RLE-EXP algorithm is based on a threshold run. The threshold run consists of two parts: a fixed length run of size *t* and a variable length run of size *e*. Both runs consists of sequences of *s*. Size *e* is defined by the following equation:

$$e = \log_2(L-1).$$

The variable length run contains the information of the logarithm of the length of *L*, which is identical to the length of the binary representation B_a. B_a would disrupt the local symbol context if placed behind the threshold run. Therefore, B_a is placed as a bit sequence in a separate data stream called the RLE Mantissa Buffer (RMB). During the decoding, the algorithm decodes *e* first and then reads B_a from RMB. The data of RMB will not be processed by the GST stage but is coded directly within the EC stage. Hence the RLE-EXP algorithm replaces each run of length *l* by a run of length $t + e$ and a bit sequence B_a in RMB, which is processed separately. Especially long runs are encoded very efficiently because of the logarithmic structure. Since the pressure of runs is taken out before the GST stage already, there is no need anymore for a further RLE0 stage in front of the EC stage. The entire algorithm is presented in Figure 4.

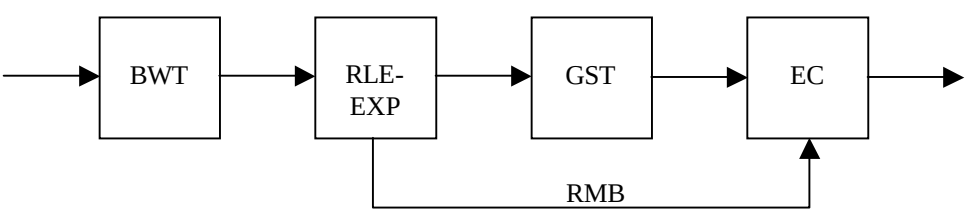

Fig. 4. The RLE-EXP algorithm

Table III presents some examples of threshold runs with $t = 2$. The RLE-EXP algorithm works especially well with the IF algorithm.

Table III. Threshold runs with $t = 2$

Original Run	***L***	***e***	***B_a* (binary)**	**Threshold Run**
aa	2	0	–	aa
aaa	3	1	0	aaa
aaaa	4	1	1	aaa
aaaaa	5	2	00	aaaa
aaaaaa	6	2	01	aaaa
aaaaaaa	7	2	10	aaaa
aaaaaaaa	8	2	11	aaaa
aaaaaaaaa	9	3	000	aaaaa
aaaaaaaaaa	10	3	001	aaaaa

3.4 The RLE-BIT algorithm

Apart from the use of a threshold run as in the RLE-EXP algorithm, the start of a run can be encoded by escape symbols. Since escape symbols usually disturb the symbol context of the GST stage, a new technique is introduced in this paper, which does not hamper the GST context. Hereto the RLE-BIT algorithm is split into two parts, with both processes being very fast and which bypasses the run information around the GST stage. The first part is called RLE-BIT-0 and is located before the GST stage. The second part is called RLE-BIT-1 and is located after the GST stage. RLE-BIT-0 stores the position and the length of each run in a separate temporary buffer TB and removes all symbols of this run except the very first one. Therefore the output sequence of RLE-BIT-0 contains no runs of symbols anymore and the length of the output sequence of RLE-BIT-0 is shorter than the corresponding length of an output sequence of RLE-EXP. After the GST stage, RLE-BIT-1 inserts a sequence of escape symbols at the former position of the run in order to encode the run length. Figure 5 represents the function of the RLE-BIT algorithm.

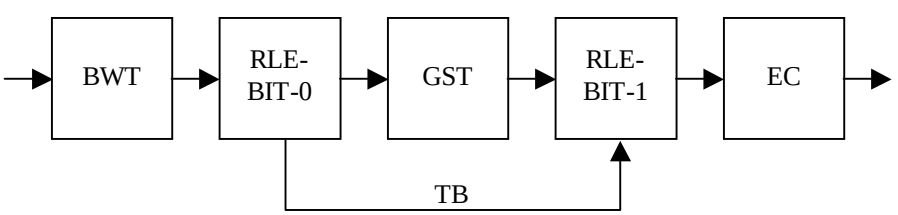

Fig. 5. The RLE-BIT algorithm

The run length is encoded by the escape symbols 0 and 1. All bits of the binary representation of the run length except the most significant bit are saved with the symbols 0 and 1, similar to the second part of the Elias code and the RLE0 coding. Table IV displays some examples of encoded run lengths for different length *L*.

Table IV. RLE-BIT encodings of run lengths

Original Run	***L***	**Encoding**
aa	2	0
aaa	3	1
aaaa	4	00
aaaaa	5	01
aaaaaa	6	10
aaaaaaa	7	11
aaaaaaaa	8	000
aaaaaaaaa	9	001
aaaaaaaaaa	10	010

All symbols from the output sequence of the GST stage get incremented by 2 in order to be able to decode the escape symbols unambiguously. Since RLE-BIT-1 inserts the escape symbols at the former position of the run, the length of X_{out} of the GST stage must be the same as the length of X_{in}. Because the length of X_{out} at the IF stage is smaller than X_{in}, the RLE-BIT algorithm does not work with the IF stage.

4. IMPROVEMENTS TO THE INVERSION FREQUENCIES

4.1 Symbol Sorting by Frequency Distribution

The IF stage produces for each symbol a of A_{in} a part sequence S_a. During this process, only symbols which are greater than a are counted. Hence, if symbols with a high frequency distribution are processed first, the part sequences of the following symbols, with a lower frequency distribution get smaller values. On the other hand, the part sequences for symbols with a high frequency distribution are longer than the part sequences for symbols with a lower frequency distribution. In order to point out the influence of the frequency distribution, A_{in} of the IF stage is permuted either in ascending frequency order or in decreasing frequency order. Table V denotes the compression rates for an original alphabet and for both permuted alphabets. In most cases, the ascending alphabet permutation produces better compression rates than the original alphabet. Only

the files `geo` and `obj1`, which are both binary files, have a worse compression rate. If the alphabet is sorted by descending frequencies, both binary files get a better compression rate than the one from the original alphabet. The file `pic` gets a better compression rate for both sorting orders. Therefore, the alphabet permutation of the IF stage has a strong influence on the compression rate and the remaining question is which direction to select for the respective file.

Table V. Compression rates in bps for IF stages with an original alphabet, a permuted alphabet sorted by ascending frequencies and a permuted alphabet sorted by descending frequencies. Compression rates better than the rates with the original alphabet are printed bold

File	Original Alphabet	Ascending Alphabet	Descending Alphabet
`bib`	1.922	**1.916**	1.934
`book1`	2.227	**2.226**	2.238
`book2`	1.939	**1.929**	1.945
`geo`	4.383	4.434	**4.214**
`news`	2.417	**2.399**	2.429
`obj1`	3.849	3.849	**3.824**
`obj2`	2.430	**2.418**	2.436
`paper1`	2.433	**2.411**	2.448
`paper2`	2.346	**2.336**	2.358
`pic`	0.707	**0.706**	**0.704**
`progc`	2.480	**2.464**	2.499
`progl`	1.713	**1.696**	1.721
`progp`	1.726	**1.713**	1.737
`trans`	1.516	**1.500**	1.541
Average	**2.292**	**2.286**	**2.288**

4.2 Sorting Order

In order to find the optimal sorting direction, some characteristics of the frequency distribution of A_{in} can be used. For each symbol a of A_{in}, let f_a be the number of occurrences of a within X_{in}, i.e. the symbol distribution. Let F_{avg} denote the average frequency count of X_{in} with length n by

$$F_{avg} = \frac{n}{|A_{in}|}.$$

Further, G is defined as the set of symbols, for which f_a is greater or equal to 2 F_{avg} by

$$G = \{a \mid f_a > 2F_{avg}\}.$$

Then S describes the percentage share of symbols a of the alphabet A_{in}, for which f_a is greater or equal to 2 F_{avg} by

$$S = 100\frac{|G|}{|A_{in}|}.$$

Table VI reveals S for each file of the Calgary Corpus.

Table VI. S as the percentage share of symbols a of the alphabet, for which $f_a \geq 2\ F_{avg}$

File	**Share**
bib	13.41
book1	15.85
book2	15.46
geo	5.86
news	15.15
obj1	9.38
obj2	10.94
paper1	14.58
paper2	16.30
pic	11.32
progc	13.98
progl	18.18
progp	14.44
trans	17.17
Average	**13.72**

The values of S for the files `geo` and `obj1`, which achieve the best compression rate with a descending sort order, are the lowest ones in the table and are below 10. Therefore, the IF stage is improved by a sorting of the alphabet where the sorting direction is dependent on the symbol distribution. Hereto, the IF stage calculates in the first place the frequency distribution of the symbols and the corresponding value of S, and performs afterwards a permutation of A_{in} either in ascending order or in decreasing order depending on S. As a

threshold for S, the value of 10 is used: if for one file, S is greater or equal to 10, the alphabet permutation is performed in ascending frequency order, in the other case in decreasing frequency order.
The improved IF stage in this paper is called Sorted Inversion Frequencies (SIF) because of the sorting permutation.

4.3 EC Stage

The choice of the model at the EC stage is very important in order to achieve a good compression rate. Since $|A_{out}|$ of the SIF stage is greater than 256, a binary coder is used for X_{out} of the SIF stage. Each symbol a of X_{out} is divided in two parts: the exponential part e and the binary representation B_a:

$$e = \left|B_a\right| = \log_2(a).$$

Part e is coded with a hierarchical model [Fenwick, 1996] with 2 levels, the first level handles values from 0 until 4 and the second level handles all values greater or equal to 5. A typical BWCA blocksize b_n is 1 MB, resulting in a maximum for e of 20.
The bit sequences B_a are processed more extensively. They get sorted by their length e. All B_a which have a length of 1, get coded by a model M_1 with values 0 and 1. All bit sequences B_a with a length of 2 are treated as a single value between 0 and 3 and get coded by a model M_2. Accordingly, bit sequences B_a with a length of 3 are treated as a single value between 0 and 7 and get coded by a model M_3. All B_a with a length of 4 or greater are divided in two parts. The first part consists of the first 3 bits of B_a, which are processed by M_3. The rest of the bits of B_a are coded sequentially by a separate model M_0 with values 0 and 1. The reason for using separated models for the first 3 bits of B_a is because the first bits have a stronger context relation than the rest of the bits, i.e. the distribution of the first bits is not as random as the distribution of the rest of the bits.
Since the sequences of e are stored alongside the bit sequences B_a, it is possible to decode X_{out} later from e and B_a unambiguously.
Apart from X_{out} of the SIF stage, the B_a data from the RMB of the RLE-EXP stage must be encoded too as pictured in Figure 4. Hereto the bit sequences B_a from the RMB get coded in the same way as the bit sequences B_a from X_{out}.

5. IMPROVEMENTS TO THE WEIGHTED FREQUENCY COUNT

5.1 Finer Graduation

The WFC implementation of Deorowicz gets the best compression rate, 2.249 bps for the Calgary Corpus by the weight function w_{6q}, which uses 5 logarithmic quantized levels

[2002]. For the implementation of this algorithm, a wide set of different weighting functions was examined. The weighting functions were based on logarithmic levels because of efficient computation as described in the paper of Deorowicz. Since the compression rate depends on several parameters beside the weight function and the number of logarithmic levels, like the kind of RLE algorithm and the model of the EC stage, it is not easy to predict which weight function and number of levels will lead generally to the best compression results. The approach of this paper uses a finer graduation by using more levels than Deorowicz, and usually leads to improved results. Of course, more levels need more time to calculate since the number of counters, where the values are changing, rises proportional to the number of levels [Deorowicz, 2002]. When using the RLE-BIT stage and the model of the EC stage, the best compression rate was achieved at 12 logarithmic levels instead of only 5. For the size of the sliding window t_{max}, the same value as by Deorowicz is used:

$$t_{max} = 2048 .$$

5.2 Calculated Weights

The individual weights of the weight function are of central significance for the compression rate. Since the structure and symbol distribution varies from file to file, a weighting function with fixed weights independent from the file structure will not lead to optimal compression rates for all files. For some files, a function with stronger weights is best suited for symbols of the immediate past than for older symbols. For other files, a weighting function, which weights older symbols almost the same as more recent symbols, gives better results. Therefore, the present implementation does not use fixed weights, but calculates the weights depending on the symbol distribution.

Hereto the parameter S is defined as in the SIF stage, which describes the percentage share of symbols a of the alphabet A_{in}, for which the frequency count f_a of a inside X_{in} is greater or equal to 2 F_{avg} by

$$F_{avg} = \frac{n}{|A_{in}|},$$

$$G = \{a \mid f_a > 2F_{avg}\},$$

$$S = 100\frac{|G|}{|A_{in}|}.$$

Further, $f(l)$ is defined as an integer function with parameters p_0, p_1 and S:

$$f_{p_0,p_1,S}(l) = \begin{cases} 2^{17} & l = 0 \\ 2^{14} & l = 1 \\ \dfrac{f_{p_0,p_1,S}(l-1)\,p_0}{p_1 + (lS^2)} & l \geq 2 \end{cases} .$$

Then the weight function $w_{p_0,p_1,S}(t)$ for the present implementation, starting from 0, is defined as:

$$w_{p_0,p_1,S}(t) = \begin{cases} f_{p_0,p_1,S}(0) & t = 0 \\ f_{p_0,p_1,S}(1) & 2^0 \leq t \leq 2^1 - 1 \\ f_{p_0,p_1,S}(2) & 2^1 \leq t \leq 2^2 - 1 \\ f_{p_0,p_1,S}(3) & 2^2 \leq t \leq 2^3 - 1 \\ \ldots & \\ f_{p_0,p_1,S}(11) & 2^{10} \leq t \leq 2^{11} - 1 \\ 0 & t \geq 2048 \end{cases} .$$

Table VII displays the compression rates for different values of p_0 and p_1. The parameters p_0 and p_1 were chosen empirically, whereas the value of S is determined by the symbol distribution of the respective file.

Table VII. Compression rates in bps for $w_{p_0,p_1,S}(t)$.

Best compression rates in a row are printed bold

File	p_0=2400 p_1=4000	p_0=2600 p_1=4200	p_0=2800 p_1=4400	p_0=2400 p_1=4400	p_0=2800 p_1=4000	p_0=2600 p_1=4185
bib	**1.887**	1.888	1.892	1.887	1.904	1.888
book1	2.270	2.266	**2.264**	2.279	**2.264**	2.265
book2	1.943	**1.942**	**1.942**	1.949	1.948	**1.942**
geo	4.198	4.190	**4.185**	4.225	4.219	4.190
news	2.390	2.387	**2.385**	2.399	2.386	2.386
obj1	**3.732**	**3.732**	3.739	3.737	3.798	3.733
obj2	**2.362**	2.365	2.370	**2.362**	2.440	2.365
paper1	**2.380**	2.382	2.386	2.382	2.399	2.382
paper2	2.336	2.332	**2.331**	2.341	2.333	2.332
pic	0.711	**0.709**	0.718	0.715	0.758	**0.709**
progc	**2.418**	2.419	2.424	2.421	2.435	2.420
progl	1.660	**1.659**	**1.659**	1.664	1.660	**1.659**
progp	1.656	1.659	1.664	**1.654**	1.675	1.659
trans	1.442	1.440	**1.439**	1.444	**1.439**	1.440
Average	2.242	**2.241**	2.243	2.247	2.261	**2.241**

The best overall compression rate was achieved by the following parameter values:

$p_0 = 2600$,

$p_1 = 4185$.

The improved algorithm with the calculated weights $w_{p_0,p_1,S}(t)$ is called Advanced Weighted Frequency Count (AWFC).

5.3 EC Stage

Since $|A_{out}|$ of the RLE-BIT-1 stage is equal to 258 and therefore much smaller than $|A_{out}|$ of the SIF stage, the composition of the model for the AWFC implementation is simpler than the model for the SIF implementation.

The model is divided into three parts, whose first part handles the symbols 0, 1 and 2, which are the most frequent symbols. The symbols 0 and 1 are the escape symbols of the RLE-BIT stage. All symbols greater than 2 are handled by the second and third part of

the model. Since the symbols 0 and 1 represent runs of zeros, this structure is similar to the ternary structure which Balkenhol and Shtarkov have suggested [1999].

Part two and three build a structured coding model with two levels. The first level as a selector divides the symbols in eight disjoint subsets: {3}, {4, ..., 5}, {6, ..., 9}, {10, ..., 15}, {16, ..., 31}, {32, ..., 69}, {70, ..., 149} and {150, ..., 257}. The second level handles the offset of the current symbol in each group [Fenwick, 1996]. For example the symbol 11 would be coded in the first level as 3, since it is in the fourth subset, and in the second level as 1, since it is the second symbol of the subset.

6. AN IMPROVED BURROWS-WHEELER COMPRESSION ALGORITHM

6.1 Selecting the GST Stage

Since the structure of the output sequence of the AWFC and SIF stage is very different, it is not easy to predict which GST stage for a respective file is better suitable in order to achieve the best compression rate. There is one property of the SIF stage which can be used in this context. The number of additional information beside the part sequences which the SIF stage needs for decoding, like symbol frequencies and terminator symbols, is proportional to $|A_{in}|$ of the input sequence. Since $|A_{in}|$ is limited to 256, the quotient r with

$$r = \frac{|A_{in}|}{n}$$

is getting smaller for larger input sizes n, which means that larger files need a smaller proportion of additional information than smaller files. As a result, the SIF scheme gives usually better results on larger files.

The results of the SIF scheme and of the AWFC scheme are displayed in Table VIII.

Table VIII. Compression rates in bps for the SIF and the AWFC scheme.
Best compression rates for each row are printed in bold font

File	Size	SIF	AWFC
bib	111,261	1.916	**1.888**
book1	768,771	**2.226**	2.265
book2	610,856	**1.929**	1.942
geo	102,400	4.214	**4.190**
news	377,109	2.399	**2.386**
obj1	21,504	3.824	**3.733**
obj2	246,814	2.418	**2.365**
paper1	53,161	2.411	**2.382**
paper2	82,199	2.336	**2.332**
pic	513,216	**0.706**	0.709
progc	39,611	2.464	**2.420**
progl	71,646	1.696	**1.659**
progp	49,379	1.713	**1.659**
trans	93,695	1.500	**1.440**
Average		2.268	**2.241**

The files book1, book2 and pic, which are the biggest files, get a better compression rate when processed by the SIF scheme, whereas all other files compress better with the AWFC scheme. Another reason why the SIF scheme is more suitable for larger files is its higher speed compared to the AWFC scheme. In order to achieve the best possible compression rate, the final implementation of this article uses a hybrid GST algorithm. For smaller files, the GST stage consists of an AWFC stage and is combined with the RLE-BIT-0/1 stages and a matching EC stage as described above. Larger files use an SIF stage as the GST stage bundled with an RLE-EXP stage and a EC stage, adapted on the SIF output as described above. Despite the fact that the file news, which is greater than 256 KB, gets a better compression with AWFC than with SIF, experiences on other corpora show that a threshold T_{SIF} of

$T_{SIF} = 256\ KB$

achieves generally the best results. All files smaller than T_{SIF} get processed with the AWFC scheme, and files greater or equal than T_{SIF} get processed with the SIF scheme. Figure 6 illustrates the final BWCA with the hybrid GST stage.

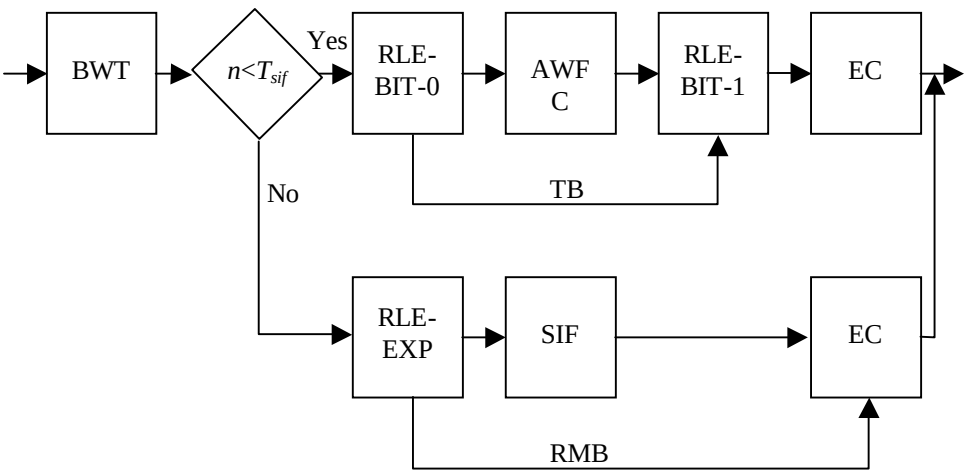

Fig. 6. The final BWCA with the hybrid GST stage

6.2 Comparisons of Compression Rates and Times

In Table IX, the results of the compression rates for the Calgary Corpus are compared between known algorithms from the literature and the hybrid BWCA presented in this paper. Table X shows the compression and decompression times from GZIP, a widespread compression program which is freely available, and from the hybrid approach presented in this paper. Hence the speed can be compared indirectly to compression programs on other operating systems, since GZIP is available on most operating systems. The following algorithms are itemized:

- GZIP93-V1.2.4 with option -9 – from Jean-loup Gailly and Mark Adler, based on LZ77 [1993],
- PPM*95 – from William Teahan [1995],
- cPPMII64-02 – from Dmitry Shkarin [2002],
- CTW95 – from Frans Willems, Yuri Shtarkov and Tjalling Tjalkens, based on CTW [1995],
- VW98 – from Paul Volf and Frans Willems, based on PPM* and CTW [1998],
- BW94 – from Michael Burrows and David Wheeler, based on BWT [1994],
- F96 – from Peter Fenwick, based on BWT [1996],
- BS99 – from Bernhard Balkenhol and Yuri Shtarkov, based on BWT [1999],
- D02 – from Sebastian Deorowicz, based on BWT [2002],
- A03 – the BWT approach presented in this paper.

The results show that A03 produces the best average compression rate of all BWT based algorithms. A comparison must also take into account that most PPM and CTW based approaches need higher computational resources. A03 needs about double as much time as GZIP93 for compression and about five times as much as GZIP93 for decompression, which is significantly less than most PPM and CTW approaches.

The execution time of the BWT, RLE, SIF, AWFC and EC stages are very different from file to file. During compression, the BWT and EC stage of A03 need about the same time on average. The IF stage of A03 needs about 50% of the time of the BWT/EC stage and the WFC stage takes about 100% more time than BWT/EC in average during compression. During decompression the BWT stage of A03 is about ten times faster than the EC stage on average.

All tests were performed on a WINDOWS 2000 PC with a 700 MHz Pentium III processor.

Table IX. Compression rates for the Calgary Corpus in bps

File	**GZIP 93**	**PPM* 95**	**cPPMII 64-02**	**CTW 95**	**VW98**	**BW94**	**F96**	**BS99**	**D02**	**A03**
bib	2.516	1.86	1.676	1.79	1.71	2.02	1.95	1.91	1.896	1.888
book1	3.256	2.41	2.135	2.19	2.15	2.48	2.39	2.27	2.274	2.226
book2	2.702	2.00	1.782	1.87	1.82	2.10	2.04	1.96	1.958	1.929
geo	5.355	4.78	4.158	4.46	4.53	4.73	4.50	4.16	4.152	4.190
news	3.072	2.37	2.137	2.29	2.21	2.56	2.50	2.42	2.409	2.399
obj1	3.839	3.83	3.498	3.68	3.61	3.88	3.87	3.73	3.695	3.733
obj2	2.628	2.31	2.110	2.31	2.25	2.53	2.46	2.45	2.414	2.365
paper1	2.792	2.33	2.142	2.25	2.15	2.52	2.46	2.41	2.403	2.382
paper2	2.880	2.34	2.124	2.21	2.14	2.50	2.41	2.36	2.347	2.332
pic	0.816	0.84	0.704	0.79	0.76	0.79	0.77	0.72	0.717	0.706
progc	2.679	2.34	2.161	2.29	2.20	2.54	2.49	2.45	2.431	2.420
progl	1.807	1.61	1.390	1.56	1.48	1.75	1.72	1.68	1.670	1.659
progp	1.812	1.55	1.391	1.60	1.46	1.74	1.70	1.68	1.672	1.659
trans	1.611	1.39	1.172	1.34	1.26	1.52	1.50	1.46	1.452	1.440
Avg.	**2.697**	**2.28**	**2.041**	**2.19**	**2.12**	**2.40**	**2.34**	**2.26**	**2.249**	**2.238**

Table X. Compression and decompression times
for the Calgary Corpus in seconds

File	Comp. Time GZIP	Decomp. Time GZIP	Comp. Time A03	Decomp. Time A03
bib	0.09	0.05	0.23	0.20
book1	0.62	0.25	1.23	1.10
book2	0.41	0.18	0.86	0.79
geo	0.28	0.07	1.11	1.11
news	0.25	0.13	0.56	0.55
obj1	0.04	0.04	0.18	0.18
obj2	0.23	0.09	0.96	0.91
paper1	0.05	0.04	0.14	0.13
paper2	0.07	0.04	0.20	0.18
pic	0.64	0.09	0.31	0.31
progc	0.04	0.04	0.11	0.10
progl	0.07	0.04	0.14	0.12
progp	0.05	0.03	0.10	0.08
trans	0.07	0.04	0.17	0.14
Sum	**2.91**	**1.13**	**6.30**	**5.90**

7. CONCLUSIONS

The Burrows-Wheeler Compression Algorithm achieves good compression rates combined with high speed. Within this field, the post BWT stages play a central role in order to realize the best possible results. Implementations of post BWT stages typically consist of three stages: a Global Structure Transformation (GST), a Run Length Encoding (RLE) stage and an Entropy Coder (EC) stage.

This paper presents a new position for the RLE stage together with adapted RLE techniques. The first technique, called RLE-EXP and based on the variable length code of Maniscalco, uses a threshold run with a logarithmic length and compresses the mantissa information of the run length outside the normal symbol buffer directly with an adapted entropy coder. The second technique, called RLE-BIT, is a new RLE algorithm and removes all runs from the symbol buffer before the GST stage and codes the run length information back into the buffer after the GST stage by using two escape symbols.

An improved version of the Inversion Frequencies algorithm, called Sorted Inversion Frequencies (SIF), was introduced, which performs a sorting permutation of the input alphabet either in ascending or descending order depending on the symbol distribution. This permutation together with an adapted EC model helps to upgrade the compression rate and to accelerate the execution.

A variation of the Weighted Frequency Count algorithm, called Advanced Weighted Frequency Count (AWFC), improves compression by using more logarithmic levels than the original implementation and by using a calculating scheme for the weights depending on the symbol distribution. Together with a special EC model, a better compression rate to the original implementation is achieved.

SIF and AWFC stages were combined in a hybrid scheme, the SIF stage with the RLE-EXP stage and the AWFC stage with the RLE-BIT stages. Files smaller than 256 KB are treated by the AWFC scheme and files larger or equal to 256 KB are processed by the SIF scheme.

This hybrid scheme achieves a compression rate for the Calgary Corpus of 2.238 bps. The compression speed achieves around 45% of the speed of GZIP and the decompression speed achieves about 20% of the speed of GZIP.

At this point of time, it is difficult to predict if further improved techniques of already known GST algorithms will give better compression rates or whether the development of a totally new approach for the GST algorithm could lead to better compression rates and higher speeds. In any case, the distance between the compression rates of the BWCA and the compression rates of the strongest compression algorithms known, the PPM and CTW methods, is getting smaller.

ACKNOWLEDGMENTS

Special thanks go to William Teahan, who provided extensive comments on this work. The RLE discussions with Yaakov Gringeler and several remarks on this work by Sebastian Deorowicz, Szymon Grabowski, Uwe Herklotz, Michael Maniscalco and Vadim Yoockin are appreciated very much.

REFERENCES

ANDERSSON, A. AND NILSSON, S. 1994. A New Efficient Radix Sort. In *35th Symposium on Foundations of Computer Science*, 714–721.

ANDERSSON, A. AND NILSSON, S. 1998. Implementing Radixsort. *The ACM Journal of Experimental Algorithmics*. Volume 3, Article 7.

ARNAVUT, Z. AND MAGLIVERAS, S.S. 1997. Block Sorting and Compression. In *Proceedings of the IEEE Data Compression Conference 1997*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 181–190.

AWAN, F.S., ZHANG, N., MOTGI, N., IQBAL, R.T. AND MUKHERJEE, A. 2001. LIPT: A reversible lossless text transform to improve compression performance. In *Proceedings of the IEEE Data Compression Conference 2001*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 481.

BALKENHOL, B. AND KURTZ, S. 1998. Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice. *IEEE Transactions on Computers*, 49(10), 1043–1053.

BALKENHOL, B., KURTZ, S. AND SHTARKOV, Y.M. 1999. Modifications of the Burrows and Wheeler Data Compression Algorithm. In *Proceedings of the IEEE Data Compression Conference 1999*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 188–197.

BALKENHOL, B. AND SHTARKOV, Y.M. 1999. One attempt of a compression algorithm using the BWT. *SFB343: Discrete Structures in Mathematics*, Falculty of Mathematics, University of Bielefeld, Preprint, 99–133.

BELL, T.C., CLEARY, J.G. AND WITTEN, I.H. 1990. *Text Compression*. Prentice-Hall, Englewood Cliffs, NJ.

BELL, T.C., WITTEN, I.H. AND CLEARY, J.G. 1989. Modelling for Text Compression. *ACM Computing Surveys*, 21(4), 557.

BENTLEY, J., SLEATOR, D., TARJAN, R. AND WEI, V. 1986. A locally adaptive data compression scheme. *Communications of the ACM*, 29, 320–330.

BINDER, E. 2000. Distance Coder. *Usenet group: comp.compression*. http://groups.google.com/groups?selm=390B6254.D5113AD2%40T-Online.de.

BURROWS, M. AND WHEELER, D.J. 1994. A Block-Sorting Lossless Data Compression Algorithm. *Technical report*, Digital Equipment Corporation, Palo Alto, California.

DEOROWICZ, S. 2000. Improvements to Burrows-Wheeler Compression Algorithm. Software – *Practice and Experience 2000*, 30(13), 1465–1483.

DEOROWICZ, S. 2002. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software – Practice and Experience 2002*, 32(2), 99–111.

ELIAS, P. 1987. Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length Schemes. *IEEE Transactions on Information Theory*, Vol. 21 (2), 194–203.

FENWICK, P. 1995. Improvements to the Block Sorting Text Compression Algorithm. *Technical Report 120*, University of Auckland, New Zealand, Department of Computer Science.

FENWICK, P. 1996. Block Sorting Text Compression - Final Report. *Technical Report 130*, University of Auckland, New Zealand, Department of Computer Science.

FRANCESCHINI, R., KRUSE, H., ZHANG, N., IQBAL, R. AND MUKHERJEE, A. 2000. Lossless, Reversible Transformations that Improve Text Compression Ratios. *Project paper*, University of Central Florida, USA.

GAILLY, J.L. 1993. *GZIP - The data compression program - Edition 1.2.4*. http://miaif.lip6.fr/docs/gnudocs/gzip.pdf.

GRABOWSKI, S. 1999. Text Preprocessing for Burrows-Wheeler Block-Sorting Compression. In *VII Konferencja Sieci i Systemy Informatyczne - Teoria, Projekty, Wdrozenia*, Lodz, Poland.

GRINGELER, Y. 2002. Private correspondence.

ISAL, R.Y.K. AND MOFFAT, A. 2001. Parsing Strategies for BWT Compression. In *Proceedings of the IEEE Data Compression Conference 2001*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 429–438.

ISAL, R.Y.K., MOFFAT, A. AND NGAI, A.C.H. 2002. Enhanced Word-Based Block-Sorting Text Compression. In *Proceedings of the twenty-fifth Australasian conference on Computer science*, Volume 4, January 2002, 129–138.

ITOH, H. AND TANAKA, H. 1999. An Efficient Method for Construction of Suffix Arrays. *IPSJ Transactions on Databases*, Abstract Vol.41, No. SIG01 – 004.

KAO, T.-H. 2001. *Improving Suffix-Array Construction Algorithms with Applications*. Master's thesis, Department of Computer Science, Gunma University, Japan.

KRUSE, H. AND MUKHERJEE, A. 1999. Improving Text Compression Ratios with the Burrows-Wheeler Transform. In *Proceedings of the IEEE Data Compression Conference 1999*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 536.

KURTZ, S. 1998. Reducing the Space Requirement of Suffix Trees. *Report 98–03*, Technische Fakultat, Universitat Bielefeld.

KURTZ, S. AND BALKENHOL, B. 1999. Space Efficient Linear Time Computation of the Burrows and Wheeler-Transformation. ALTHÖFER, I. ET AL. Eds. *Numbers, Information and complexity*, Festschrift in honour of Rudolf Ahlswede's 60th Birthday, 375–384.

LARSSON, N.J. 1999. *Structures of String Matching and Data Compression*. PhD thesis, Department of Computer Science, Lund University, Sweden.

MANISCALCO, M.A. 2000. A Run Length Encoding Scheme For Block Sort Transformed Data. *Technical paper*, http://www.geocities.com/m99datacompression/papers/rle/rle.html.

MANISCALCO, M.A. 2001. A Second Modified Run Length Encoding Scheme For Block Sort Transformed Data. *Technical paper*, http://www.geocities.com/m99datacompression/papers/rle2.html.

MANZINI, G. AND FERRAGINA, P. 2002. Engineering a Lightweight Suffix Array Construction Algorithm. *Lecture Notes in Computer Science*, Springer Verlag, Volume 2461, 698–710.

NILSSON, S. 1996. *Radix Sorting & Searching*. PhD thesis, Department of Computer Science, Lund University, Lund, Sweden.

SADAKANE, K. 1997. *Improvements of Speed and Performance of Data Compression Based on Dictionary and Context Similarity*. Master's thesis, Department of Information Science, Faculty of Science, University of Tokyo, Japan.

SADAKANE, K. 2000. *Unifying Text Search And Compression -Suffix Sorting, Block Sorting and Suffix Arrays*. PhD thesis, University of Tokyo, Japan.

SCHINDLER, M. 1997, A Fast Block-sorting Algorithm for lossless Data Compression. In *Proceedings of the IEEE Data Compression Conference 1997*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 469.

SEWARD, J. 2000. On the performance of BWT sorting algorithms. In *Proceedings of the IEEE Data Compression Conference 2000*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 173–182.

SHKARIN, D. 2002. PPM: One step to practicality. In *Proceedings of the IEEE Data Compression Conference 2002*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 202–211.

TEAHAN, W. 1995. Probability estimation for PPM. In *Proceedings of the New Zealand Computer Science Research Students' Conference*, University of Waikato, New Zealand.

VOLF, P. AND WILLEMS, F. 1998. Switching Between Two Universal Source Coding Algorithms. In *Proceedings of the IEEE Data Compression Conference 1998*, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 491–500.

WILLEMS, F.M.J., SHTARKOV, Y.M. AND TJALKENS, T.J. 1995. The Context Tree Weighting Method: Basic Properties. *IEEE Transactions on Information Theory*, Vol. IT–41 (3), 653–664.