Universal Text Preprocessing for Data Compression

Jürgen Abel, *Member, IEEE* and William Teahan, *Member, IEEE*

***Abstract*— Several preprocessing algorithms for text files are presented which complement each other and which are performed prior to the compression scheme. The algorithms need no external dictionary and are language independent. The compression gain is compared along with the costs of speed for the BWT, PPM and LZ compression schemes. The average overall compression gain is in the range of 3 to 5 percent for the text files of the Calgary Corpus and between 2 to 9 percent for the text files of the large Canterbury Corpus.**

***Index Terms* — data compression, BWT, LZ, PPM, preprocessing, text compression.**

I. Introduction

TODAY the most popular schemes for lossless data compression are the Burrows-Wheeler Compression Algorithm (BWCA) [1], Prediction by Partial Matching (PPM) [2] and Lempel-Ziv (LZ) [3] schemes. The first two schemes are context related, whereas the LZ scheme is dictionary based. Even though each of these schemes can be used to compress text files, they do not consider the special properties of textual data, and the compression rate can be enhanced by using preprocessing algorithms specialized for textual data. Text preprocessing algorithms are reversible transformations which are performed before the actual compression scheme prior to encoding and behind the decompression scheme after decoding. Since textual data make up a substantial part of the Internet and other information systems, efficient compression of textual data is of significant practical interest.

This paper presents a number of text preprocessing algorithms, including a text recognition scheme and five separate algorithms: capital letter conversion, EOL coding, word replacement, phrase replacement and alphabet reordering. The different techniques are processed sequentially in order to complement and boost each other. The approach presented here is a universal one, which in contrast to many others needs no fixed external information like dictionaries. It is language independent – as long as the text is based on Latin letters. Nevertheless, it achieves a significant compression gain. The first four algorithms work with most universal compression schemes; the last algorithm is especially designed for sort based schemes like the BWCA.

The impact of the text preprocessing algorithms is illustrated using the example of the 10 text files of the Calgary Corpus – `bib`, `book1`, `book2`, `news`, `paper1`, `paper2`, `progc`, `progl`, `progp` and `trans` – for the BWCA, PPM and LZ compression schemes. The BWCA is represented by the program ABC 2.4 from Abel [4], the PPM algorithm is represented by the program PPMD+ from Teahan [5] and for the LZ scheme the program GZIP from Gailly is used [6].

· Jürgen Abel, University Duisburg-Essen, Department "Communications Systems", Faculty of Engineering Sciences, Bismarckstrasse 81, D-47057 Duisburg, Germany. E-mail: juergen.abel@data-compression.info.

· William Teahan, School of Informatics, University of Wales, Bangor, Gwynedd LL57 1UT, United Kingdom. E-mail: wjt@informatics.bangor.ac.uk.

II. Previous Work

The preprocessing of textual data for data compression is a subject of many publications. In some articles, the treatment of textual data is embedded within the compression scheme itself but could easily be separated into two independent parts: a preprocessing algorithm and a standard compression algorithm, which are processed sequentially one after the other.

Bentley et al. describe a word based compression scheme [7], where words are replaced by an index into an MTF (Move To Front) list. The dictionary of the words is transmitted implicitly by transmitting the word during its first occurrence. This scheme can be divided into a parsing part that performs the preprocessing and a standard MTF ranking scheme. A word based variation of the PPM scheme is presented by Moffat [8]. He uses order-0, order-1 and order-2 word models to achieve better compression than the MTF scheme from Bentley et al. Similar schemes, which differentiate between alphanumeric strings and punctuation strings, and which also use an implicit dictionary, are presented by Horspool and Cormack [9]. Again, these schemes can be divided into a parsing part and a coding part using Huffman codes.

Teahan and Cleary describe several methods for enlarging the alphabet of the textual data [10]. Besides the replacement of common bigrams by a one symbol token, they propose methods for encoding special forms of bigrams called digrams (two letters representing a single sound as *ea* in "bread" or *ng* in "sing"). The replacements are processed using a fixed set of frequently used bigrams in the English language, which makes

this attempt language dependent. Teahan and Cleary [11] describe a word based compression scheme where the word dictionary is adaptively built from the already processed input data. This can also be achieved by a preprocessing stage if the words are replaced by corresponding tokens. Teahan presents a further comparison between two different word based compression schemes in his Ph.D. thesis [12]. The first scheme uses function words, which include articles, prepositions, pronouns, numbers, conjunctions, auxiliary verbs and certain irregular forms. The second scheme uses the most frequently used words in the English language. Both schemes require external dictionaries and are language dependent.

A special case of word encoding is the star encoding method from Franceschini and Mukherjee [13] and later from Kruse and Mukherjee [14]. This method replaces words by a symbols sequence that mostly consist of repetitions of the symbol '*'. This requires the use of an external dictionary that must be known by the sender as well as the receiver. Inside the dictionary, the words are first sorted by their length and second by their frequency in the English language using information obtained from Horspool and Cormack [9]. In [13], the words are encoded by sequences of the same length, where letters where replaced by '*', e.g. "b*D**". In [14], all words of the same length are encoded by sequences "*…*", "A*…*", … , "Z*…*", "a*…*", …, "z*…*", "*A*…*", … where the length of the encoded sequence is equal to the length of the word being encoded. The requirement of an external dictionary makes these methods again language dependent. Later, Sun, Zhang and Mukherjee present an improved dictionary based scheme called StarNT, which works with a ternary search tree [15] and is about twice as fast in encoding and about four times as fast in decoding than the former approach. The first 312 words of the dictionary are the most frequently used words of the English language. The rest of the dictionary is filled by words sorted by their length first and then by their frequency. This language dependent approach reaches an average compression gain between 10% for PPM, 13% for BWCA and 19% for LZ based compression schemes.

Preprocessing methods, specialized for a specific compression scheme, are presented by Chapin and Tate [16] and later by Chapin [17]. They describe several methods for alphabet reordering prior to using the BWCA in order to place letters with similar contexts close to one another. Since the Burrows-Wheeler transformation (BWT) is a permutation of the input symbols based on a lexicographic sorting of the suffices, this reordering places areas of similar contexts at the BWT output stage closer together, and these can be exploited by the latter stages of the BWCA. The paper compares several heuristic and computed reorderings where the heuristic approaches achieve a better result on text files than the computed approaches. The average gain for BWCA using heuristic reorderings over the normal alphabetic order was 0.4% on the text files of the Calgary Corpus. Balkenhol and Shtarkov use a very similar heuristic alphabet reordering for preprocessing with BWCA [18]. A different alphabet reordering for BWCA is used in the paper from Kruse and Mukherjee [19]. It also describes a bigram encoding method and a word encoding method which is based on their star encoding.

Grabowski proposes several text preprocessing methods in his publication [20], which focuses on improvements for BWCA but some techniques can also be used for other compression schemes. Besides the already mentioned techniques like alphabet reordering, bigram-, trigram- and quadgram replacement, Grabowski suggests three new algorithms. The first one is capital conversion. An escape symbol and the corresponding lower letter replace capital letters at the beginning of a word. If the second letter of the word is capitalized too, the replacement is omitted. This technique increases context dependencies and similarities between words, which can be exploited by standard compression schemes. The second algorithm is space stuffing, where a space symbol is placed at the beginning of each line in order to change the context that follows the end of line symbol (EOL) to one space instead of various symbols. The last algorithm is EOL coding, which replaces EOL symbols by space symbols and separately encodes the former EOL positions, which are represented by the number of blanks since the previous EOL symbol. These numbers are encoded either within the symbol stream itself or in a separate data stream. Grabowski, who attributes the EOL coding idea to Taylor, suggests using either space stuffing or EOL coding for preprocessing text files, but because of unstable side-effects, EOL coding is omitted. His preprocessing algorithms without EOL coding achieve an average gain for BWCA of 2.64% on the 10 text files of the Calgary Corpus. Since he uses a set of fixed bigrams, trigrams and quadgrams, his proposal requires an external dictionary and is language dependent.

Franceschini et al. extend the star encoding method by using different schemes for the indices into the dictionary [21], called Length-Preserving Transform (LPT), Reverse Length-Preserving Transform (RLPT) and Shortened-Context Length-Preserving Transform (SCLPT). All of these require an external dictionary and are language dependent. Franceschini reported for SCLPT, which achieves the best results, a gain for BWCA of 7.1% and for PPMD+ a gain of 3.8% for the files of the Calgary Corpus (including the files `paper3`, `paper4`, `paper5` and `paper6`). A further improvement of the star encoding method, presented by Awan et al. [22], is called Length Index Preserving Transform (LIPT). LIPT encodes a word as a string that can be interpreted as an index into a dictionary. The string consists of three parts: a single symbol '*', a symbol between 'a' and 'z', and a sequence of symbol from the set 'a'…'z', 'A'…'Z'. The second part of the string, the single symbol, represents the length *l* of the word, where 'a' stands for length 1 and 'z' for length 26. The third part is the encoded index inside the set of words with length *l*. They are encoded as a number representation of base 52 decremented by 1, where 'a' represents 0, …, 'z' represents 25, 'A' represents 26, …, and 'Z' represents 51. An empty substring represents the

number 0. Therefore, a word of length 3 with index 0 is encoded as "*c", a word of length 3 with index 1 as "*ca", a word of length 3 with index 27 as "*cA" and so on. LIPT achieves a gain for BWCA on the Calgary Corpus of 4.1% and of 5.6% for GZIP.

Isal and Moffat present different text preprocessing schemes for bigrams and words [23] using internal and external dictionaries. In their paper, tokens are used with values above 255, so they can be used together with normal symbols, as the compression scheme needs to handle alphabets with more than 8 bits. For text files, the word based schemes with internal dictionaries give the highest compression gain. Later Isal et al. combine the word preprocessing scheme with different global structure transformations and entropy coding schemes [24]. Because of the use of an internal dictionary, where each word is spelt out the first time it occurred, the schemes of Isal and Moffat are all language independent.

Teahan and Harper propose a switching algorithm for combining both dynamic and static PPM models that also involves an initial text preprocessing step [25]. In this step that occurs prior to the encoding step, the text is essentially marked up by additional switch symbols to indicate when the compression algorithm should switch to another model. A greedy search algorithm which minimizes the overall code length of the encoded stream (of both the original symbols and additional switch symbols) is used to determine the positions of the markup symbols. This scheme is only relevant to context based schemes such as PPM, and it requires a modification of the subsequent PPM compression scheme.

III. Recognizing Text Files

Text files have statistical properties which differ from properties of other types of files like images, measurement or executable files. Text preprocessing algorithms are specifically designed for the properties of textual data. Using a text preprocessing algorithm on non textual data leads to a misinterpretation of the context and to worse compression rates. In order to get optimal compression results, it is important to distinguish between text files and other type of files and to use the text preprocessing algorithms only on text files.

It is assumed that text files are based on Latin letters represented by the normal byte oriented ASCII character set, supplemented if needed by some language dependent characters above ASCII 127, like the Umlaute 'ä', 'ö' 'ü' in German for example.

Table I displays some symbol frequencies of the files from the Calgary Corpus and large Canterbury Corpus including non-text files. Besides the file size in column two, the frequency of the 63 alphanumerical symbols, consisting of letters (A...Z, a...z), digits (0...9) and the space symbol, is listed in column three and the frequency of the single space symbol in column four. The fifth column indicates that the frequency percentage share of the alphanumerical symbols for text files like `book1` is much higher than for non-text files like

TABLE I
Symbol Frequencies

File	Size	Alpha-num [a]	Space [b]	Alphanum % [c]	Space/ Alphanum % [d]
bib	111,261	94,190	13,739	84.66	14.59
book1	768,771	717,438	125,551	93.32	17.50
book2	610,856	556,047	85,885	91.03	15.45
geo	102,400	25,314	590	24.72	2.33
news	377,109	327,891	54,269	86.95	16.55
obj1	21,504	5,202	377	24.19	7.25
obj2	246,814	70,574	3,417	28.59	4.84
paper1	53,161	46,920	7,301	88.26	15.56
paper2	82,199	76,858	12,112	93.50	15.76
pic	513,216	1,830	43	0.36	2.35
progc	39,611	30,500	6,925	77.00	22.70
progl	71,646	52,457	12,238	73.22	23.33
progp	49,379	39,599	11,474	80.19	28.98
trans	93,695	66,973	9,901	71.48	14.78
bible.txt	4,047,392	3,894,894	766,111	96.23	19.67
E.coli	4,638,690	4,638,690	0	100.00	0.00
world192.txt	2,473,400	2,188,279	428,662	88.47	19.59
Avg.				**70.72**	**14.19**

[a] Number of alphanumerical symbols (A...Z, a...z, 0...9) and space.
[b] Number of spaces.
[c] Percentage share of alphanumerical symbols to the size.
[d] Percentage share of space to alphanumerical symbols.

e.g. obj1. All text files have values above 70% and all non-text files have much lower values with one exception. The file E.coli is a protein file and belongs to the non-text files, since its symbols represent codes for amino acids. Nevertheless the symbols consist only of letters and 100% of the file's symbols are alphanumerical. Therefore, the share of alphanumerical symbols alone is not a reliable indicator for text files and some more indicators should be considered.

In text files, words are mostly separated by space symbols and therefore space symbols are frequently used symbols inside text files. Column six shows that the space symbol compared to the alphanumerical symbols has a frequency percentage share of more than 10% for text files even though it is only one of 63 alphanumerical symbols. For non-text files, the value is much lower including the protein file E.coli, which has no space symbols at all.

Based on these two observations, two requirements are defined in order for a file to be categorized as a text file by the text recognition scheme:

1. The percentage frequency share of alphanumerical symbols (A...Z, a...z, 0...9, ' ') compared to all symbols should be greater than 66%.
2. The percentage frequency share of the space symbol compared to alphanumerical symbols should be greater than 10%.

These conditions categorize the following 10 files of the Calgary Corpus as text files: bib, book1, book2, news, paper1, paper2, progc, progl, progp and trans. The files geo, obj1, obj2 and pic are classified as non-text files. From the large Canterbury Corpus, the files bible.txt and world192.txt are also classified as text files and E.coli is classified as a non-text file.

A text recognition module at the beginning of the preprocessing chain displayed in Figure 2 classifies files either as text files or as non-text files respectively based on these conditions and cancels further preprocessing for non-text files.

Clearly, the assumptions on which these conditions are taken are a great simplification because of the lack of an official definition of a text file. This simple approach can lead in some cases to false classifications; furthermore, languages not based on Latin letters, like Russian and Chinese, or 16-bit character sets have not been taken into account. However, the advantage of the approach is its simplicity and speed, and experiments show that the method is extremely effective across a wide range of text files.

IV. Universal Preprocessing Algorithms

A. Capital Letter and Upper Word Conversion

Within text files, words often appear in two forms: starting with a lower letter and starting with a capital letter. In many cases, the current form can be easily determined by a linguistic rule such as the first word of a new sentence starts with a capital letter.

While it is quite easy for a human reader to recognize that both forms mean the same word, for a computer scheme this is problematical. The basic idea of letter conversion is to help the compression scheme to recognize the similarities of these two forms by replacing the capitalized form by the lower case form and to add a corresponding escape symbol in front of the word. This way, the text gets more equally written words. On the other hand, the escape symbols tend to appear in similar contexts, for example behind a dot, carriage return or linefeed symbol as they are usually in front of a new sentence. These properties can be well exploited by context based and repetition based compression schemes.

In contrast to other schemes which use a one phase approach, this paper presents a two phase algorithm which leads to better and more predictable results.

In the first phase, a ternary search tree (a tree for which every node represents a symbol and has up to three edges for: less than, equal, or greater than [26]) is constructed containing all words of the text with length of size 2 or more. Hereto a word is defined as a sequence of lower or upper case letters surrounded by non-letter symbols. In the second phase, only words that start with a capital letter and occur with a lowercase letter elsewhere are considered for capital letter conversion. This phase ensures that words that always start with a capital letter are unconverted, since this conversion would decrease context similarities. Additionally, it is checked if the second letter is a lowercase letter. Only if both conditions are fulfilled, then the capital letter is converted into a capital-letter escape symbol, followed by a space symbol and the corresponding lowercase letter. Even though the original symbol is now replaced by three symbols, there is a gain on average for the compressed text files.

A further enhancement is achieved if words, consisting only of uppercase letters, are converted into a capital-word escape symbol followed by the word in lowercase letters. This word conversion is performed only if the number of lowercase letters inside the text is larger than the number of uppercase letters. This test ensures that files, consisting mainly of upper letters, are unchanged.

The symbols for the capital-letter escape and capital-word-escape are not fixed escape symbols but are calculated as follows. In principle, any symbol, which has a frequency count of zero, could be used as an escape symbol. In order to save symbols with a zero frequency count for phrase and word tokens, a different approach is used here instead. Inside normal text, uppercase letters rarely occur as a separate single letter (except the article 'A' and the personal pronoun 'I' in the English language), therefore this approach uses two uppercase letters as escape symbols. The two uppercase letters are determined that occur most rarely after a non-letter symbol. All occurrences inside the original text, where these two letters occur after a non-letter symbol, are replaced by two of the respective letters in order to unambiguously decode the escape symbols inside the decoder later.

The two uppercase letters are transmitted as the first two symbols of the transformed text. Besides saving possible token symbols, using uppercase letters as escape symbols has the

advantage that the escape symbols are sorted closely to the other upper letters, which plays a positive role for sort based compression algorithms such as BWCA.

Table II shows the impact of the capital letter and upper word conversion. All compression schemes achieve a gain on average. The gain is greater for BWCA and PPM than for LZ, since BWCA and PPM are able to exploit context similarities whereas LZ schemes are based on simple repetitions.

TABLE II
PERCENTAGE GAIN FOR CAPITAL LETTER AND UPPER WORD ALGORITHM

FILE	BWCA % GAIN	PPM % GAIN	LZ % GAIN
bib	0.27	-0.57	-0.79
book1	0.49	0.26	0.27
book2	0.53	0.25	0.33
news	0.44	-0.66	0.01
paper1	0.66	0.75	0.37
paper2	0.70	0.79	0.50
progc	0.79	0.70	0.78
progl	0.78	0.42	0.78
progp	-0.07	-0.29	-0.35
trans	0.06	-0.60	-1.51
Avg.	**0.47**	**0.11**	**0.04**

B. EOL Coding

Within a text file, words are most of the time separated by space symbols. There are some less frequent symbols like "Carriage Return" (CR) and "Linefeed" (LF), dot, comma or similar symbols, which separate words from each other. The CR and LF symbols separate lines from each other and are called end of line (EOL) symbols. If the EOL symbols are simply replaced by space symbols, text files like book1 and book2 can be compressed much better because more words become separated by space symbols and context similarities increase. Columns two to four of Table III show the percentage share of gain if all EOL symbols are replaced by space symbols. All three compression schemes achieve an average compression gain of about 2%. In order to reconstruct the original file, the old EOL symbol positions need to be encoded additionally.

If the code space needed to encode the former EOL symbol positions is smaller than the code space saved by the EOL replacement, the files can be reconstructed and an overall gain would be achieved. Since EOL symbols stand at the end of a line, the distance between two EOL symbols is in many text files quite stable. If the EOL symbols are replaced by spaces, the former position can be encoded by the number of spaces since the occurrence of the last EOL symbol. Column six to eight of Table III shows the impact of a simple EOL scheme, where the positions of the EOL symbols are encoded in a separate stream using the number of space symbols that occurred since the last EOL. The stream is encoded with variable length integer codes similar to the Elias codes [27]. The binary lengths of the integers are encoded first with an arithmetic coder. The bits of the integer are encoded with a binary arithmetic coder, with the most significant bit first. The size of the respective stream is listed in the fifth column of the table and added to the size of the compressed symbol stream for comparison. Except for file book1, which gets a compression gain between 1% and 2%, all files get a worse compression rate. Obviously, this scheme gives in some cases a noticeable gain, but in many cases the encoding costs are too expensive.

Four connected problems can be identified and need to be solved:

1. Shall every EOL be encoded?
2. Shall every text file be EOL encoded?
3. What symbol shall be encoded as the EOL symbol?
4. How shall the EOL positions be encoded?

TABLE III
PERCENTAGE GAIN FOR THE SIMPLE EOL ALGORITHM

FILE	CR/LF BWCA % gain [a]	CR/LF PPM % gain [a]	CR/LF LZ % gain [a]	EOL STREAM SIZE [b]	EOL BWCA % gain [c]	EOL PPM % gain [c]	EOL LZ % gain [c]
bib	0.05	0.11	0.32	1,940	-7.28	-7.40	-5.16
book1	4.81	5.33	3.53	6,786	1.64	1.76	1.39
book2	3.89	4.01	3.03	6,806	-0.70	-0.86	-0.24
news	2.08	2.24	1.92	4,689	-2.07	-2.17	-1.31
paper1	2.94	3.21	2.47	610	-0.94	-0.86	-0.85
paper2	3.22	3.54	2.67	864	-0.41	-0.30	-0.26
progc	0.17	0.34	0.60	712	-5.88	-5.92	-4.83
progl	0.06	-0.12	0.72	1,053	-7.04	-7.19	-5.86
progp	0.97	1.29	1.43	937	-8.19	-7.27	-6.92
trans	0.42	0.34	0.99	907	-4.79	-4.31	-3.73
Avg.	**1.86**	**2.03**	**1.77**		**-3.57**	**-3.45**	**-2.78**

[a] Percentage share of gain if all CR/LF are replaced by spaces without encoding former positions (non reversible).
[b] Size of the EOL positions stream in bytes.
[c] Percentage share of gain if CR/LF are replaced by spaces with encoding the former positions.

The answers to these problems are quite subtle and represent the key to a more successful and reliable EOL coding.

The coding of a position of an EOL symbol requires code space. Therefore, only EOL symbols, which lead to a more predictable context, should be replaced by space symbols and their positions encoded. In this approach, only EOL symbols that are surrounded by lowercase letters are replaced. In other words, only a part of all EOL symbols inside the file are considered for replacement and the others are ignored. Table IV displays the percentage share of valid EOL symbols in column two. All files, which have a percentage share above 10%, receive a compression boost. Therefore, if the percentage share of the valid EOL symbols is less than 10%, the whole file will not be EOL encoded.

CR and LF are treated as EOL symbols together with the combination CR/LF, which is treated as one EOL symbol. Some files not only have a unique EOL symbol, like CR, LF or CR/LF, but some files like `trans` have a set of different EOL symbols. Therefore, the EOL symbol, which has the highest frequency count inside the file, is regarded as the EOL symbol that is being replaced. The combination CR/LF is treated in a particular manner. For files, where the combination CR/LF is most frequent, all CR/LF combinations are replaced foremost by a single EOL symbol, which is then treated as the EOL symbol to replace.

The implementation of the last question – the coding of the positions – has a very strong influence on the result. Since in many text files the number of symbols per line is on average quite similar, this property can be exploited with a sophisticated encoding scheme. The length of the current line is compared with the average length of the last 16 lines. If the current length is greater or equal to the average line length, then the numbers of spaces between the average line length and the current EOL positions are counted and encoded. If the current length is smaller, an escape symbol is encoded as a minus sign and the numbers of spaces between the current EOL position and the average line length is encoded. If the current EOL is an invalid EOL – in the sense that it has not been considered for replacement – two escape symbols are encoded and the EOL is left unchanged.

The effect of the enhanced EOL scheme can be seen at the last three columns of Table IV. The share of replaced EOL symbols for the files `bib`, `progc`, `progl`, `progp` and `trans` is smaller than the threshold, and therefore these files are omitted for EOL coding (the EOL stream size of 1 byte represents a flag, which is transmitted in front of the actual data, giving information if the following data is EOL encoded or not). All other files achieve a compression boost with the three compression schemes.

C. Token based Replacement

Token based replacement is a technique which replaces substrings inside a text file by tokens. The tokens are shorter than the replaced substrings and therefore, the text file is transformed into a shorter representation. The correlation between the tokens and the corresponding substrings can be achieved by an external dictionary or by including an implicit dictionary. In order to stay language independent, this approach uses an implicit dictionary. The dictionary can be constructed if the substrings at their first occurrence are written out together with the corresponding tokens. Since the number of tokens is limited in the approach presented here, algorithms are devised in order to calculate the most valuable substrings for replacement.

More precisely, given an alphabet A, a set of letter symbols $L = \{a,...,z,A,...,Z,\_\}$ and a set of non-letter symbols $N = A \setminus L$, all occurrences except the first one of a substring X inside the input text are replaced by a token t in the output text. Two kinds of substrings are considered for replacement. The first kind is a word. A word is a sequence of letter symbols from L, which are surrounded in the input text by non-letter symbols from N. The second kind of substrings are either bigrams or trigrams, which consist of two or three letter

TABLE IV
PERCENTAGE GAIN FOR THE ENHANCED EOL ALGORITHM

FILE	VALID EOLS %	EOL STREAM SIZE [a]	BWCA % gain	PPM % gain	LZ % gain
bib	0.0	1	0.00	0.00	0.00
book1	56.3	3,882	1.82	1.81	1.16
book2	33.6	3,776	0.46	0.25	0.37
news	10.7	856	0.26	0.31	0.09
paper1	20.2	194	0.58	0.78	0.52
paper2	30.5	334	1.23	1.38	0.97
progc	0.4	1	-0.01	-0.01	-0.01
progl	0.0	1	-0.01	-0.01	-0.01
progp	0.5	1	-0.01	-0.01	-0.01
trans	5.4	1	-0.01	-0.01	-0.01
Avg.			**0.43**	**0.45**	**0.31**

[a] Size of the EOL positions stream in bytes.

symbols from L in the input text respectively. Bigrams and trigrams are normally part of a word. For a token t, a symbol is used which has an initial frequency count of zero inside the text.

1) Determination of available Tokens

Before the actual replacement can start, the set of available tokens needs to be determined. All symbols with a frequency count of zero and ordinal values between 0 and 31 and between 127 and 255 are identified. This token set is encoded as the sequence T:

$$T = nq_0n_0q_1n_1...$$

where n is the total number of tokens, q_i is a token, and n_i is the number of continuous tokens that follow. The q_i are ordered in ascending order. For example, if the tokens with ordinal value 2, 3, 4, 5, 127, 128, 129, 150, 151, 152, 200, 201, 202, 203 have a frequency count of zero, T would be:

$$T = 14,2,4,127,3,150,3,200,4\ .$$

The sequence T is transmitted at the beginning of the text. The first two tokens t_0 and t_1 (in the former example 2 and 3) are used for trigram replacement, the next two tokens t_2 and t_3 (in the former example 4 and 5) for bigram replacement and the rest of the tokens are used for word replacement. If there are less than 6 tokens, the word replacement is omitted; if there are less than 4 tokens, the phrase replacement is omitted too.

2) Word Replacement

Word replacement is the preprocessing algorithm that gives the most compression boost on average. The basic idea is to replace frequently used words by tokens, which are indices into a word dictionary. The size s_D of the dictionary is given by the number of tokens available for word replacement as described in the last section. The token t_4 is used as a first-occurrence escape symbol. In order to keep the approach language independent, no fixed dictionary is used. Instead, the dictionary is built adaptively and transmitted together with the data stream. In the first phase, the frequencies of all words of the file are calculated using a ternary search tree for words with length of size 2 or more. In many languages, the stem of a word is extended with a separate letter at the end for declination like the nominative plural in English and the genitive singular in German or for conjugations like the 's' for the third person singular in English. In order not to waist a valuable token besides the token for the shorter stem S, only words W are replaced which occur at least four times as often as the shorter stem S. Furthermore, a value for each word W is calculated, which represents the value for replacing W by a token inside the whole file. The value v_W is calculated by:

$$v_W = (f_W - 1) \cdot (l_W - 1)\ ,$$

where f_W is the frequency of W and l_W is the length of W. The set of all words is then sorted descending by their respective values and the first – most valuable – s_D words inside the file are replaced by tokens in the third phase. When a word occurs for the first time, it is transmitted as is. After the word, t_4 is output, indicating that the prior word should be added into the dictionary. During further occurrences, the word is replaced by the token t_{i+5}, where i is the number of words already encoded in the adaptively growing dictionary. Since the decoder is able to rebuild the dictionary from the transmitted data, no external dictionary is needed and the approach stays independent from a specific language. Not all replaced words boost the compression, e.g. a word with $f_W = 1$ hampers the compression, since its first and only occurrence is written out and there are no more occurrences to replace. Also, the longer the word, the greater is the number of symbols saved by each replacement. Therefore, in the present approach only words which have a value for f_W of at least 16 are considered for replacement.

3) Phrase Replacement

Besides word replacement, bigrams and trigrams are also used for token based replacement. Using larger n-grams than bigrams and trigrams for replacement does not lead to a better compression rate usually. One reason is that the larger n-grams are much less frequent than bigrams and trigrams, another reason is that the tokens are withdrawn from the available tokens for word encoding, which achieves better results than encoding of larger n-grams.

Therefore, the two most frequent trigrams are determined and are encoded with the tokens t_0 and t_1. During the fist occurrence of a trigram, the trigram is transmitted as is and the respective token is transmitted afterwards. The most frequent bigrams are treated the same way, encoded with the tokens t_2 and t_3. There are cases for which encoding a bigram or trigram hampers the compression rate, e.g. if the bigram or trigram occurs only once in the file. It is hard to predict for each individual file the particular frequency count at which a bigram or trigram should be encoded, therefore a general threshold of 64 is used in this approach. Bigrams and trigrams are only encoded if their frequency count is greater than this value.

Table V shows the result of the token based replacement for words and phrases. Almost all files have a compression boost between 1 and 4 percent. A strange effect is presented by the file `trans`. The file `trans` is a transcript of a terminal session that contains many control characters and consists of several sections with quite different contexts. It is difficult to explain why BWCA and LZ based schemes are hampered in

TABLE V
PERCENTAGE GAIN FOR THE TOKEN BASED REPLACEMENT ALGORITHM

FILE	BWCA % GAIN	PPM % GAIN	LZ % GAIN
bib	0.97	2.65	3.59
book1	0.98	1.46	5.53
book2	-0.07	3.57	6.72
news	0.53	2.05	4.02
paper1	0.70	0.43	4.60
paper2	1.22	1.06	6.31
progc	2.17	1.28	1.27
progl	2.36	5.96	2.84
progp	2.17	3.76	4.96
trans	-0.43	2.53	-1.47
Avg.	**1.06**	**2.48**	**3.84**

this case while PPM achieves a compression gain. One possible reason might be the fact that PPM can gain especially from the word and phrase replacement because it reduces the number of possible escapes to lower order contexts. If words are replaced by tokens, the number of symbols to encode decrease and, on the other hand, the alphabet size increases. But with each symbol that is saved from being encoded, not only the code spaces for the other symbols of the replaced word or phrase are saved, but also the code spaces for possible escapes are saved (in case a new symbol appears). If several symbols are replaced by one token, only the code space for one escape symbol is needed. Also the fact that all files achieve a compression boost for PPM supports this argument. Sequences that are shortened by token replacements also result in many of the contexts being longer, and this may help with the prediction.

The gain of word and phrase replacement can be enhanced if the capital letter conversion is processed beforehand as depicted in Table VI. The decomposition of capital words into an escape symbol and a lower cased word supports the word replacement by producing more words with exactly the same spelling, which can be efficiently replaced by tokens afterwards. It is interesting to note that the overall gain of the compound scheme is in all cases higher than the sum of the separate gains of the two algorithms, i.e. capital letter conversion really boosts word and phrase replacement.

TABLE VI
PERCENTAGE GAIN FOR THE TOKEN BASED REPLACEMENT ALGORITHM IN CONJUNCTION WITH CAPITAL LETTER AND UPPER WORD ALGORITHM

FILE	BWCA % GAIN	PPM % GAIN	LZ % GAIN
bib	1.81	2.92	3.77
book1	1.76	2.20	6.20
book2	1.81	4.38	7.51
news	1.26	2.48	4.28
paper1	1.89	1.66	5.51
paper2	2.30	2.25	7.17
progc	2.99	2.49	2.50
progl	3.24	6.82	3.73
progp	3.16	3.75	4.84
trans	0.01	2.78	-1.20
Avg.	**2.02**	**3.17**	**4.43**

V. SPECIAL TEXT PREPROCESSING ALGORITHMS

For compression schemes that are based on sorting stages like BWCA, for example, not only the sorting algorithm has an impact on the sorted output but also the lexicographic order of the alphabet during the sorting process [16]. During BWCA, the sorted data is usually processed by a Global Structure Transformation (GST), which is usually a ranking scheme that transforms the local context of the different output segments to a global context [4, 19, 28]. This global context data is finally compressed by an Entropy Coding (EC) stage. If the lexicographic order of the alphabet is changed in a way that symbols with a similar context are grouped closer together, segments with similar context properties will also be grouped closer together. This can be exploited by the GST stage since context changes will be smoother and less abrupt, resulting in lower ranking values in the output of the GST stage. A typical example of a GST stage is the MTF stage. Table VII displays the influence of the different alphabet orders on the average ranking value of the MTF output after a BWT stage [4]. Smaller values lead to a better compression ratio since they are much more frequent in the highly skewed distribution of the MTF output and have a higher probability at the EC stage than larger ranking values [29].

The second column shows the average ranking value for the original alphabet order. Column three displays the result for the ordering "AEIOUBCDGFHRLSMNPQJKTWVXYZ" plus the punctuation groupings "?!" and "+-,." from Chapin and Tate [16]. The main difference to the original order is the grouping of the vocals at the beginning of the letters. The average ranking value is about 0.7% lower than the one from the original order. The results for the alphabet order of this approach are revealed in column four. The average result is about 3% lower than the result from the original order. The presented alphabet order is different to former approaches of grouping all vowels together [16, 17, 18, 19]. This approach groups the vowels "aoui" in the middle of the consonants together and the 'e' at the end of the consonants. Furthermore, other characters like digits and punctuation symbols are also reordered. Figure 1 contains the complete alphabet order.

The first attempt of the authors to obtain a better alphabet order was to calculate an order based on statistical properties of the current data. The probability distributions of bigrams and trigrams have been analyzed as well as solutions based on

```
"´_", tab, "@", space, 0x1B, lf, cr,
".:?,];)|~&<=>{*+}[/!-"'\0123456789%$",
"SNLMGQZBPCFWRHAOUIYXVDKTJE",
"^#(",
"snlmgqzbpcfwrhaouiyxvdktje",
0x7F … 0xFF,
0x00 … 0x08, 0x0B, 0x0C, 0x0E … 0x1A, 0x1C … 0x1F
```

Fig. 1. New alphabet order

TABLE VII
AVERAGE SYMBOL VALUE OF THE MTF OUTPUT FOR DIFFERENT BWT ALPHABET ORDERS

FILE	AVG. RANKING ORIGINAL ORDER %	AVG. RANKING CHAPIN/TATE ORDER %	AVG. RANKING PRESENTED ORDER %
bib	2.50	2.48	2.46
book1	2.45	2.44	2.43
book2	2.25	2.24	2.22
news	3.80	3.78	3.76
paper1	3.27	3.22	3.13
paper2	2.82	2.79	2.74
progc	3.91	3.85	3.72
progl	1.99	1.96	1.91
progp	2.18	2.15	2.04
trans	1.97	1.94	1.92
Avg.	**2.71**	**2.69**	**2.63**

TABLE VIII
PERCENTAGE GAIN FOR THE ALPHABET REORDERING ALGORITHM

FILE	BWCA % GAIN	PPM % GAIN	LZ % GAIN
bib	0.52	0.00	-0.01
book1	0.61	0.00	-0.01
book2	0.82	0.00	0.00
news	0.67	0.00	0.01
paper1	1.37	0.00	0.02
paper2	1.26	0.00	0.01
progc	1.65	0.00	0.01
progl	1.66	0.00	0.01
progp	2.06	0.00	0.02
trans	0.39	0.00	0.02
Avg.	**1.10**	**0.00**	**0.01**

different Traveling Salesman Problems (TSP) defined and calculated. Just as Chapin and Tate reported in [16] and Kruse and Mukherjee in [19], the results were quite unstable and usually worse than a simply grouping of vowels at the beginning of the letters. Therefore, a heuristic approach based on a simple trial-and-error method was used instead. The process starts with the original alphabet order, uses several runs and permutes the symbols from the string s containing all symbols from 0 to 127 in ascending order at the beginning. Each run consists of two nested loops. The outer loop runs from c=0 to 127 and uses the symbol at index c as a moving symbol. The inner loop moves the symbol away from the original position c and inserts it at all positions p_i between 0 up to 127 (including the former position). For each position p_i, the average compression ratio $r[p_i]$ for a set of text files is calculated. After completing the inner loop, the position p_i with the best $r[p_i]$ is used for the new position of the symbol and s is updated accordingly. It is possible that the same symbol is first moved to the back of s and later moved again to the front of s, since in the meantime symbols in between have been permuted too, which lead to different compression rates. The whole process is repeated many times using the last permutation of s from the former process as the new starting configuration until the compression rate stagnates. The final result depends on the GST stage and EC stage of the BWCA, since different parameters of these stages will lead for the same input data to different compression rates [4, 28] and therefore to a different alphabet order. Furthermore, as the alphabet order represents a local minimum of compression rates, a different starting configuration could lead to a different local minimum. This approach makes no assumptions about the kind of alphabet, e.g. the place of vowels, before it starts, therefore, it can also be used for non-ASCII alphabets like EBCDIC from IBM with a corresponding result.

The compression results of the alphabet reordering are listed in Table VIII. The gain of 1.1% using the new alphabet order for the BWCA is more than double as much as that reported by Chapin and Tate [16], Kruse and Mukherjee [19], and Chapin [17], who achieved between 0.2% and 0.5%. Since PPM and LZ do not depend on a sorting stage, the complete preprocessing scheme can be used as a black box in front of all standard compression schemes, without hampering PPM and LZ.

VI. RESULTS

The compression rates of the complete approach between the raw files and the text preprocessed files of the Calgary Corpus for BWCA, PPM and LZ schemes are listed in Table IX. The compression gain on average is between 3% to 5%.

Table X displays the respective results for the text files of the large Canterbury Corpus. The compression gain here is about 2% for the BWCA and between 7% to 9% for the LZ and PPM scheme. One reason for the smaller boost for the BWCA may be the fact that the BWCA has already a very strong compression rate (1.379 bps), which is more difficult to improve than the 1.689 bps from PPM and 2.334 bps from the LZ scheme. Furthermore, the program ABC uses a big block size of 5MB and gains especially from large files like `bible.txt` and `world192.txt`, as with larger files longer runs are generated inside the BWT output.

The costs for the preprocessing prior to compression and afterwards for decompression in terms of the amount of supplementary time needed to process the text files of the Calgary Corpus are revealed in Table XI for compression and in Table XII for decompression. The entries include the total times for preprocessing and compression/decompression. All times are in seconds and have been averaged over several runs of each file. The computer system was a WINDOWS 2000 PC with 768 MB RAM and a Pentium III running at 700 MHz.

The cost for preprocessing during compression is quite high, even though the total length of the textual data is shorter after preprocessing because of the token replacement (on the other

TABLE IX
COMPRESSION RATES IN BPS AND PERCENTAGE GAIN FOR THE TEXT FILES OF THE CALGARY CORPUS

File	BWCA raw [a]	BWCA prepr. [b]	BWCA %gain [c]	PPM raw [a]	PPM prepr. [b]	PPM %gain [c]	LZ raw [a]	LZ prepr. [b]	LZ %gain [c]
bib	1.888	1.840	**2.51**	1.901	1.845	**2.92**	2.516	2.421	**3.76**
book1	2.226	2.136	**4.05**	2.302	2.198	**4.52**	3.256	3.015	**7.42**
book2	1.929	1.869	**3.10**	2.017	1.913	**5.13**	2.702	2.491	**7.82**
news	2.400	2.347	**2.20**	2.408	2.338	**2.88**	3.072	2.938	**4.37**
paper1	2.381	2.286	**4.00**	2.341	2.280	**2.59**	2.791	2.630	**5.76**
paper2	2.331	2.230	**4.34**	2.317	2.226	**3.91**	2.880	2.649	**8.02**
progc	2.418	2.309	**4.49**	2.380	2.321	**2.48**	2.678	2.612	**2.48**
progl	1.658	1.585	**4.41**	1.739	1.621	**6.82**	1.806	1.739	**3.75**
progp	1.658	1.582	**4.61**	1.726	1.661	**3.74**	1.812	1.724	**4.85**
trans	1.439	1.433	**0.42**	1.530	1.488	**2.78**	1.611	1.630	**-1.20**
Avg.	**2.033**	**1.962**	**3.41**	**2.066**	**1.989**	**3.78**	**2.512**	**2.385**	**4.70**

[a] "raw" means file without preprocessing, compression rates in bps (bits per symbol).
[b] "prepr." means file with preprocessing, compression rates in bps (bits per symbol).
[c] Percentage gain between raw file and preprocessed file.

hand, the use of the new tokens enlarge the alphabet size, which causes new symbols and contexts to be encoded). PPM needs 62% more time, BWCA needs 129% more time and LZ needs 300% more time than without preprocessing. While for BWCA and LZ the time overhead compared to the compression gain is indeed high, for PPM it is moderate compared to other PPM improvements, which have far greater costs in execution speed.

For decompression, the costs are much lower: 33% more time for BWCA, 36% for PPM and 63% for LZ. The reason why is because each algorithm can be performed in one phase in contrast to compression, where several algorithms need two or more phases in order to calculate the thresholds, trees and available tokens.

VII. CONCLUSIONS

The compression rates of the popular compression techniques, which are based on BWT, PPM or LZ schemes, can be improved by using a text preprocessing algorithm beforehand.

This paper puts forward a compound text preprocessing scheme that consists of a text/non-text recognition scheme and five different text preprocessing algorithms. The first algorithm is capital letter and capitalized word conversion, which converts the capital letter of a word to a lowercase letter and converts words that consist of only of uppercase letters into words with lowercase letters. This transformation helps to increase the similarities of the contexts of the words. The second algorithm, EOL coding, replaces EOL symbols like "carriage return" and "linefeed" in the text with space characters and encodes the former positions into a separate data stream by a special encoding method that uses the average line length in terms of the number of spaces. Replacing the EOL by space symbols helps to improve the predictability of the context, since words are usually separated by spaces. The third algorithm replaces words by tokens. The most valuable words, in relation to their frequency and length, are calculated and replaced by byte tokens, except for the first occurrence. The fourth algorithm replaces the most frequent two trigrams and bigrams. Both replacement algorithms lead to a shorter

TABLE X
COMPRESSION RATES IN BPS AND PERCENTAGE GAIN FOR THE LARGE CANTERBURY CORPUS

File	BWCA raw [a]	BWCA prepr. [b]	BWCA %gain [c]	PPM raw [a]	PPM prepr. [b]	PPM %gain [c]	LZ raw [a]	LZ prepr. [b]	LZ %gain [c]
bible.txt	1.452	1.421	**2.14**	1.699	1.524	**10.29**	2.330	2.113	**9.35**
world192.txt	1.306	1.286	**1.51**	1.679	1.545	**7.99**	2.337	2.217	**5.15**
Avg.	**1.379**	**1.354**	**1.83**	**1.689**	**1.535**	**9.14**	**2.334**	**2.165**	**7.25**

[a] "raw" means file without preprocessing, compression rates in bps (bits per symbol).
[b] "prepr." means file with preprocessing, compression rates in bps (bits per symbol).
[c] Percentage gain between raw file and preprocessed file.

TABLE XI
EXTRA TIME CONSUMED BY THE PREPROCESSING DURING COMPRESSION FOR THE CALGARY CORPUS

File	BWCA raw [a]	BWCA prepr. [b]	BWCA % dif.[c]	PPM raw [a]	PPM prepr. [b]	PPM % dif.[c]	LZ raw [a]	LZ prepr. [b]	LZ % dif.[c]
bib	0.23	0.53	**132**	0.79	1.36	**73**	0.08	0.35	**342**
book1	1.25	3.08	**146**	8.15	14.06	**73**	0.61	2.61	**328**
book2	0.88	2.26	**156**	5.98	9.97	**67**	0.40	1.91	**376**
news	0.58	1.49	**157**	4.32	6.69	**55**	0.25	1.23	**393**
paper1	0.15	0.33	**118**	0.40	0.66	**64**	0.05	0.19	**273**
paper2	0.20	0.48	**140**	0.62	1.06	**71**	0.07	0.29	**314**
progc	0.12	0.23	**95**	0.31	0.48	**56**	0.04	0.12	**209**
progl	0.14	0.31	**124**	0.47	0.73	**56**	0.06	0.20	**239**
progp	0.11	0.22	**97**	0.35	0.54	**53**	0.05	0.14	**173**
trans	0.17	0.38	**126**	0.60	0.93	**56**	0.06	0.27	**357**
Avg.	**0.38**	**0.93**	**129**	**2.20**	**3.65**	**62**	**0.17**	**0.73**	**300**

[a] "raw" means times without preprocessing, times in seconds.
[b] "prepr." means total times with preprocessing, times in seconds.
[c] Percentage share of difference between raw and preprocessed times.

text length directly. The last algorithm is alphabet reordering, which supports sort based compression schemes like BWCA. A heuristic permutation technique is presented which helps to place segments with similar contexts closer together after the sorting.

All five algorithms are processed sequentially one after the other as illustrated in Figure 2 after the initial text recognition scheme. They complement each other by reaching a higher overall compression rate than the sum of the compression rates of the separate algorithms. Besides the pure compression gain, the independence from a specific language and from an external dictionary makes this universal approach very attractive for all three kinds of compression schemes.

ACKNOWLEDGMENT

The discussions with Brenton Chapin, Sebastian Deorowicz, Szymon Grabowski, Uwe Herklotz, Maxim Smirnov and Vadim Yoockin have been very much appreciated.

REFERENCES

[1] M. Burrows and D. Wheeler, "A Block-Sorting Lossless Data Compression Algorithm," Technical report 124, Digital Equipment Corporation, Palo Alto, California, 1994.
[2] J. Cleary and I. Witten, "Data Compression Using Adaptive Coding and Partial String Matching", *IEEE Transactions on Communications*, pp. 396–402, 1984.
[3] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data

TABLE XII
EXTRA TIME CONSUMED BY PREPROCESSING DURING DECOMPRESSION FOR THE CALGARY CORPUS

File	BWCA raw [a]	BWCA prepr. [b]	BWCA % dif.[c]	PPM raw [a]	PPM prepr. [b]	PPM % dif.[c]	LZ raw [a]	LZ prepr. [b]	LZ % dif.[c]
bib	0.21	0.27	**29**	0.83	1.18	**42**	0.05	0.07	**40**
book1	1.11	1.41	**27**	8.42	12.53	**49**	0.25	0.45	**81**
book2	0.81	1.04	**28**	6.19	8.84	**43**	0.18	0.35	**93**
news	0.56	0.76	**36**	4.49	6.07	**35**	0.12	0.24	**100**
paper1	0.13	0.19	**44**	0.42	0.57	**35**	0.03	0.05	**56**
paper2	0.19	0.26	**37**	0.65	0.90	**38**	0.04	0.06	**50**
progc	0.11	0.14	**27**	0.32	0.42	**31**	0.03	0.04	**33**
progl	0.12	0.17	**42**	0.49	0.63	**29**	0.03	0.05	**68**
progp	0.09	0.12	**37**	0.37	0.48	**31**	0.03	0.04	**44**
trans	0.14	0.17	**21**	0.63	0.77	**22**	0.03	0.05	**67**
Avg.	**0.35**	**0.45**	**33**	**2.28**	**3.24**	**36**	**0.08**	**0.14**	**63**

[a] "raw" means times without preprocessing, times in seconds.
[b] "prepr." means total times with preprocessing, times in seconds.
[c] Percentage share of difference between raw and preprocessed times.

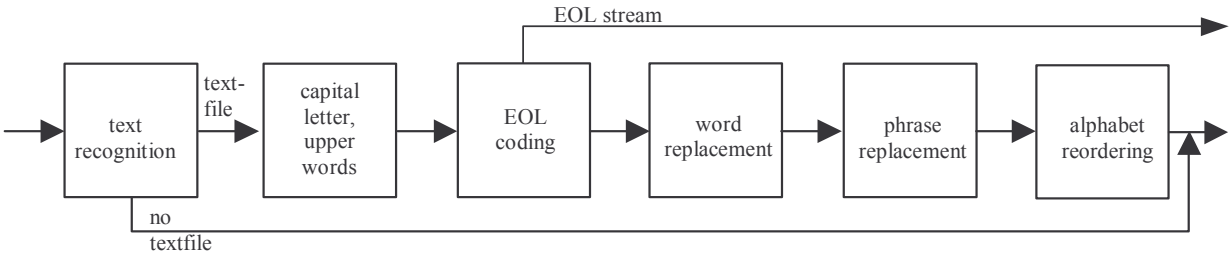

Fig. 2. The complete text preprocessing scheme.

Compression," *IEEE Transactions on Information Theory*, pp. 337–342, 1977.

[4] J. Abel, "Improvements to the Burrows–Wheeler Compression Algorithm: After BWT Stages," submitted for publication in *ACM Transactions on Computer Systems*, 2003.

[5] W. Teahan, " Probability estimation for PPM," *Proceedings of the New Zealand Computer Science Research Students' Conference*, University of Waikato, New Zealand, 1995.

[6] J. Gailly. (1993). GZIP – The data compression program – Edition 1.2.4. [Online]. Available: http://miaif.lip6.fr/docs/gnudocs/gzip.pdf

[7] J. Bentley, D. Sleator, R. Tarjan and V. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, 29, pp. 320–330, 1986.

[8] A. Moffat, "Word–based Text Compression," *Software – Practice and Experience*, pp. 185–198, 1989.

[9] N. Horspool and G. Cormack, "Constructing Word–Based Text Compression Algorithms," *Proceedings of the IEEE Data Compression Conference 1992*, Snowbird, pp. 62–71.

[10] W. Teahan and J. Cleary, "The Entropy of English using PPM–Based Models," *Proceedings of the IEEE Data Compression Conference 1996*, Snowbird, pp. 53–62.

[11] W. Teahan and J. Cleary, "Models of English Text," *Proceedings of the IEEE Data Compression Conference 1997*, Snowbird, pp. 12–21.

[12] W. Teahan, "Modelling English text," Ph.D. dissertation, Department of Computer Science, University of Waikato, New Zealand, 1998.

[13] R. Franceschini and A. Mukherjee, "Data Compression Using Encrypted Text," *Proceedings of the IEEE Data Compression Conference 1996*, Snowbird, p. 437.

[14] H. Kruse and A. Mukherjee, "Preprocessing Text to Improve Compression Ratios," *Proceedings of the IEEE Data Compression Conference 1998*, Snowbird, p. 556.

[15] W. Sun, N. Zhang and A. Mukherjee, "Dictionary-Based Fast Transform for Text Compression," *Proceedings of the IEEE International Conference on Information Technology: Coding and Computing*, Las Vegas 2003

[16] B. Chapin and S. Tate, "Higher Compression from the Burrows-Wheeler Transform by Modified Sorting," *Proceedings of the IEEE Data Compression Conference 1998*, Snowbird, p. 532.

[17] B. Chapin, "Higher Compression from the Burrows–Wheeler Transform with new Algorithms for the List Update Problem," Ph.D. dissertation, Department of Computer Science, University of North Texas, 2001.

[18] B. Balkenhol and Y. Shtarkov, "One attempt of a compression algorithm using the BWT," *SFB343: Discrete Structures in Mathematics*, Falculty of Mathematics, University of Bielefeld, Germany, 1999.

[19] H. Kruse and A. Mukherjee, "Improving Text Compression Ratios with the Burrows–Wheeler Transform," *Proceedings of the IEEE Data Compression Conference 1999*, Snowbird, p. 536.

[20] S. Grabowski, "Text Preprocessing for Burrows–Wheeler Block–Sorting Compression," *VII Konferencja Sieci i Systemy Informatyczne – Teoria, Projekty, Wdrozenia*, Lodz, Poland, 1999.

[21] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal and A. Mukherjee, "Lossless, Reversible Transformations that Improve Text Compression Ratios," Preprint of the M5 Lab, University of Central Florida, 2000.

[22] F. Awan. N. Zhang, N. Motgi, R. Iqbal and A. Mukherjee, "LIPT: A Reversible Lossless Text Transform to Improve Compression Performance," *Proceedings of the IEEE Data Compression Conference 2001*, Snowbird, pp. 481–210.

[23] R. Isal and A. Moffat, "Parsing Strategies for BWT Compression," *Proceedings of the IEEE Data Compression Conference 2001*, Snowbird, pp. 429–438.

[24] R. Isal, A. Moffat and A. Ngai, "Enhanced Word–Based Block–Sorting Text Compression," *Proceedings of the twenty–fifth Australasian conference on Computer science*, pp. 129–138, 2002.

[25] W. Teahan and D. Harper, "Combining PPM Models Using a Text Mining Approach," *Proceedings of the IEEE Data Compression Conference 2001*, Snowbird, pp. 153–162.

[26] J. Bentley and R. Sedgewick, "Fast Algorithms for Sorting and Searching Strings," *Eighth Annual ACM–SIAM Symposium on Discrete Algorithms*, New Orleans, 1997.

[27] P. Elias, "Universal Codeword Sets and Representations of the Integers," *IEEE Transactions on Information Theory*, pp. 194–203, 1975.

[28] S. Deorowicz, "Improvements to Burrows–Wheeler Compression Algorithm," *Software – Practice and Experience*, pp. 1465–1483, 2000.

[29] P. Fenwick, "Block Sorting Text Compression - Final Report", *The University of Auckland, Department of Computer Science Report No 130*, April 1996.