Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2016) 24: 4465 – 4480

doi:10.3906/elk-1410-124

A new dictionary-based preprocessor that uses radix-190 numbering

Mete Eray ŞENERGİN, Erhan Aliriza İNCE*
Eastern Mediterranean University, Department of Electrical and Electronic Engineering, Famagusta, North Cyprus, via Mersin, Turkey

Received: 21.10.2014 • **Accepted/Published Online:** 10.08.2015 • **Final Version:** 20.06.2016

Abstract: Various scholarly works in the literature have pointed out that placing a preprocessor in front of a standard postcompressor would help achieve higher gains while compressing natural-language text files. Ever since, there has been much research on preprocessors to improve the gain attained by concatenated systems. With the same goal in mind our paper proposes a new word-based preprocessor named METEHAN190 (M190) and contrasts its performance with four other state-of-the-art preprocessors. Throughout the experiments source files from the Wall Street Journal (WSJ) archive, and the Calgary, Canterbury, Gutenberg, and Pizza and Chili corpora were used. Postcompressors adapted were Prediction by Partial Matching compressor using method-D (PPMD) and Monstrous PPM II compressor (PPMonstr). It was observed that in all three experiments WRT and M190 would achieve the two highest compression gains. For small text and transcription files from the Calgary corpus, M190 would outperform all preprocessors including WRT. On the other hand, a look at average encoding and decoding times shows that the semistatic byte-oriented methods are much faster in comparison to the static dictionary-based methods that encode words with characters.

Key words: Lossless text compression, preprocessing, postcompressor, dictionary, semistatic byte-oriented preprocessors, METEHAN 190

1. Introduction

To cope with the ever-growing need of multimedia and textual information sharing, researchers have proposed various source coding algorithms. The primary aim of these algorithms is to represent a source signal with minimum number of bits or symbols. Source coding algorithms can be classified in two major groups: (i) lossless and (ii) lossy compressors. While lossless coding guarantees the exact reconstruction of the original source data, for lossy coding algorithms only an approximate copy of the original data can be obtained. The aim of this paper is to propose a new preprocessor that can be used prior to standard postprocessing algorithms (PPAs) to achieve higher compression gains.

In the literature we can find many lossless compression algorithms and these generally process bits, bytes, or words. Classical compressors like Huffman are known to use characters as the symbols to compress. Word-based algorithms on the other hand elect words as their source alphabet. Character and word-based algorithms can be categorized as: (i) statistical data compression methods and (ii) dictionary-based compression methods. Statistical compression methods employ variable-length codes and are based on a model. The model is used to map input data to bit sequences in such a way that probable (frequently encountered) data will produce

*Correspondence: erhan.ince@emu.edu.tr

shorter outputs in comparison to improbable data. The quality of compression is based on the model adapted. Static, semistatic, and adaptive models are among the well-known models. A static model is fixed and is known by both the compressor and the decompressor and does not depend on the data that are being compressed. A semistatic model, which is also fixed, is constructed from the data to be compressed and must be included as part of the compressed data. An adaptive model changes during the compression and is a function of the previously compressed part of the data. Since that part is available at the decoder storing the model is not necessary.

Entropy encoders such as Huffman coder [1], adaptive Huffman coder [2], arithmetic coder (AC) [3], prediction by partial matching (PPM) [4], and PAQ [5] are examples of statistical source coding algorithms. Statistical two-pass techniques (also known as semistatic) have good compression ratios and also permit faster searching of compressed texts. Plain Huffman (PH)[6], tagged Huffman (TH)[6], end-tagged dense codes (ETDC)[7], (s, c)-dense coding [8], and restricted prefix byte coding [9, 10] are examples of the two-pass techniques. In these methods the first pass is used to gather statistics about the list of source symbols (vocabulary) and based on the gathered information a model of the text is constructed. During the second pass each symbol is substituted by a codeword.

Examples of dictionary-based methods include LZ77, LZ78, LZW ([11, 12]), and DEFLATE [13]. Star transform [14], length index preserving transformation (LIPT) [15], star new transform (StarNT) [16], and word replacement transformation (WRT) [17] are preprocessing methods that use a static dictionary. Other algorithms that cannot be classified in either of the two groups are run-length encoders (RLE) [18], Burrows–Wheeler transform (BWT) ([19, 20]), and BZIP2 [21], which uses block sorting BWT together with Huffman coding.

Prediction by Partial Matching (PPM) ([4, 22]) is an adaptive statistical data compression technique that uses context modeling and prediction. PPMC [22] is a hybrid combination of Methods A and B described in [4]. Performance of these compression methods is based on the escape probabilities (the probability of new symbols occurring in the context). PPMD+ [23], PPMd [24], PPM* [25], and Monstrous PPMII.J (PPMonstr) [26] are some other variants of the prediction by partial matching algorithm. Compressors like Durilca and Durilca Light [27] are based on Shkarins' PPMd [24] and PPMonstr [26]. mPPM, described in [28], is a two-stage compressor. The first stage maps words into two-byte codewords using a limited length dictionary, and in the second stage conventional PPM is used to encode codewords or new words.

Electronic book (eBook) use around the world has been constantly on the rise due to the huge success of tablet computers and eReaders such as Kindle and Elonex. A small survey carried out with a total of 40,337 people from 13 different countries showed that 75 % of the respondents are expected to start reading eBooks by 2015. Anticipating that these numbers would be even higher if the scope of the survey is widened, text compression would clearly play an important role in reducing the storage space required. In addition, compression would reduce the transmission and/or download times and would lead to reduced bandwidth requirement. Motivated by the fact that placing a preprocessor in front of a postcompressor would help achieve higher compression gain(s) while compressing text file(s), in the paper we have proposed a new dictionary-based preprocessor named METEHAN 190 (M190). Our preprocessor is unique in the following ways: i) it employs a larger alphabet than most static dictionary-based methods so that it can achieve shorter average code length (alphabet sizes for LIPT, WRT, and M190 are respectively 52, 128, and 190), ii) its larger alphabet provides the flexibility for a larger dictionary (words from multiple languages can be included in the dictionary), iii) it does not use capital conversion so that the burden of using extra flags is removed (reduces computational

complexity), iv) it does not code space and punctuation characters. This is an advantage since PPAs can get a better compression gain by exploiting the redundancy.

Average bits per character (BPC) compression results obtained from experiments have showed that M190 can outperform all other preprocessors in compressing the Calgary corpus, and when Canterbury and Gutenberg corpora were used BPC values for M190 were slightly higher than those of WRT but it would still outperform the remaining algorithms. Time complexity of the proposed M190 preprocessor is also acceptable in comparison to word-based preprocessors and some PPAs. It has smaller encoding times than WRT, RPBC, PPMD-o4, and PPMonstr but is approximately four times slower than the semistatic byte-oriented preprocessors ETDC and SCDC.

The rest of the paper is organized as follows: Section 2 provides a brief summary of some state-of-the-art preprocessing techniques, namely LIPT, StarNT, WRT, ETDC, SCDC, and RPBC. Section 3 introduces the encoder and decoder blocks for M190, and provides an example to show its application. Section 4 summarizes experimental results given four different corpora [Calgary [29], Gutenberg [30], Canterbury [31], and Pizza and Chili [32]] and the Wall Street Journal (WSJ) archive obtained from TREC project [33]. First, preprocessors compared are placed prior to PPMD and PPMonstr and BPC values for the concatenated systems are obtained using Calgary, Canterbury and Gutenberg corpora. Secondly, Section 4 provides a comparison between static dictionary-based methods [M190 and WRT] and semistatic byte-oriented preprocessors [ETDC and SCDC]. For comparison six medium-to-large size text files from the Pizza and Chili corpus and a 100 MB text file from the Wall Street Journal archive were used. Section 5 provides comparative bar graphs for time complexity of the M190 encoder/decoder pair, other preprocessors, and PPAs. Finally, a short summary is provided and the paper is concluded.

2. Preprocessing techniques

Preprocessors using a static dictionary replace words in a given text by a character encoding that represents a pointer to the encoded word in the dictionary. Semistatic techniques on the other hand do not assume any data distribution and learn it during a first pass in which the model is built. After the creation of the model, text can be encoded by replacing each symbol with a fixed codeword based on the model. Subsections below summarize details of some well-known preprocessing algorithms, namely LIPT, StarNt, WRT, ETDC, SCDC, and RPBC.

2.1. Preprocessors derived from the star transform

The star transform [14] was proposed by Nelson in 2002. The main idea behind this transformation is to define a unique signature for each word by replacing the letters of the word by a special character, $(*)$, and to use a minimum number of characters to identify each specified word. Subsections 2.1.1.–2.1.3. below are examples of algorithms that have been derived from the basic star transform.

2.1.1. Length index preserving transformation (LIPT)

The LIPT algorithm [15], uses a static English language dictionary with 59,951 words (0.5 MB) and a transform dictionary of size 0.3 MB. Given a compiled dictionary, the LIPT algorithm would create many disjoint dictionaries based on word lengths. All words of length i would then be placed in dictionary D_i and then sorted according to the frequency of the word in the corpus being compressed. The algorithm then carries out mapping to encode words in each disjoint dictionary D_i.

A word in position k of the disjoint dictionary D_i, denoted $D_i[k]$, is encoded using the alphabet [a–z, A–Z] and Eq. (1):

$$\begin{array}{cc} 1 < k < 52 & *c_{\text{len}}\ c \\ 53 < k < 2756 & *c_{\text{len}}\ c\ c \\ 2756 < k < 140,608 & *c_{\text{len}}\ c\ c\ c. \end{array} \tag{1}$$

Since LIPT encoded words have '*' in front of them, the LIPT decoder locates encoded words with a star and then finds the length block indicator, which comes after the '*' symbol. Afterwards, characters that come after the length block indicator are used to compute an offset from the beginning of the chosen length block. The word at this location in the original dictionary would be the decoded word. Words without '*' in front of them are nontransformed words and hence are written to decoded file as they are, without any decoding.

2.1.2. Star new transform (StarNT)

StarNT transformation [16], which was proposed by Sun et al. has a variable length code structure similar to that of LIPT. Both algorithms use a radix 52 alphabet (a–z, A–Z) and words of higher occurrence are coded using shorter codewords. As for the usage of a star (*), in the earlier transformations it would denote the beginning of a codeword but in starNT it implies that the following word does not exist in the dictionary. This change was adapted in order to reduce the size of the transformed intermediate file and will help lower the encoding/decoding time of the backend compressor. To encode, starNT uses a dictionary, where the first 312 words (the most frequently occurring words in English) appear at the top in decreasing order of their frequencies and the remaining words are sorted according to their lengths. The alphabet adapted is [a ... z , A ... Z].

2.1.3. Word replacement transformation (WRT)

The word replacement transform (WRT) [17] has been proposed by Skibinski et al. and is a variation of the starNT with some improvements like capital conversion, word ordering, q-gram replacement, and end-of-line (EOL) coding. Details of capital conversion, dictionary ordering, q-gram replacement, and EOL coding techniques have been well explained in [17] and [34]; hence interested readers are referred to these sources for further details.

2.2. Semistatic word-based byte-oriented preprocessors

In semistatic preprocessors, using bytes instead of bits may slightly worsen the compression ratio; however, both the encoding and decoding processes will speed up. Byte-oriented preprocessors also provide the flexibility to carry out a direct pattern search on the compressed text since they are self-synchronized codes. Subsections 2.2.1.–2.2.3. below provide details on the end-tagged dense coding, (s, c)-dense coding, and restricted prefix byte coding (RPBC) techniques.

2.2.1. End-tagged dense codes

End-tagged dense coding (ETDC) [7] is a word-based byte-oriented compression method. To compute the codeword of each source word, ETDC uses a semistatic model that is simply the vocabulary ordered by frequency. The first 128 words in the vocabulary are given one byte codewords. Words in positions 128 to 128 + 128^2 - 1 are assigned two-byte codewords and the three-byte codewords are given to the remaining words. ETDC has

been inspired from the tagged Huffman code [35], and has been obtained through a very simple change. Rather than marking the beginning of each codeword the most important bit of every byte has been used to mark their end. Hence whenever a given byte is the last byte of a codeword the highest bit is set to 1; otherwise it must be set to 0. In ETDC the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits. Therefore, there is no need to use Huffman coding over the remaining 7 bits.

2.2.2. (s,c)-Dense codes (SCDC)

(s, c)-Dense coding [8] is a more sophisticated variant of word-based byte-oriented text compressors. End-tagged dense codes use 128 target symbols for the bytes that do not end a codeword (continuers), and the remaining 128 target symbols for the last byte of the codeword (stoppers). An (s, c)-dense code on the other hand adapts the number of stoppers and continuers to the word frequency distribution of the text, so that s values are used as stoppers and c = 256 - s values as continuers. SCDC assigns one-byte codewords from 0 to s-1 to the first s words of the vocabulary. Words in positions s to $s + sc$ - 1 are sequentially given two-byte codewords. Three-byte codewords are for words from $s+sc$ to $s + sc + sc^2$ -1 The encoding and decoding algorithms are the same as those of ETDC. One only needs to change the 128 value of stoppers and continuers by s and c, respectively.

2.2.3. Restricted prefix byte coding (RPBC)

Restricted prefix byte coding (RPBC) was first proposed in [9]. Unlike the (s, c)-dense codes, which use an infinite tuple of numbers, RPBC uses a length-N finite tuple, where the numbers in the tuple refer to the initial digit ranges in the radix-R code. We say that the code is restricted since $v_1 + v_2 + \cdots + v_N \leq R$. Under RPBC, the first byte of each codeword is used to describe the length of the codeword and additional bytes use the remaining code space. For example, while encoding with N=4, (v_1, v_2, v_3, v_4), the code has v_1 one-byte codewords, Rv_2 two-byte codewords, R^2v_3 three-byte codewords, and R^3v_4 four-byte codes. It is required that $v_1 + Rv_2 + R^2v_3 + R^3v_4 \leq n$, where n represents the cardinality of the source alphabet.

3. METEHAN 190 (M190)

The proposed preprocessor, M190, uses 1–3-byte-long codewords while encoding text documents. Each codeword is composed of characters that are drawn from an extended, 190-character-long alphabet. The value 190 was obtained as follows. Due to their high frequencies, space and punctuation marks deserve to be coded using the shortest codewords (1-byte). However, since this does not result in any compression, M190 administers encoding to only words, and space and punctuation characters are left as they are (uncoded). From the 256-character extended-ASCII set, this would leave only 191 classified otherwise. In addition, anticipating that some words that we need to encode may not be in the dictionary, the 127th ASCII character was reserved for encoding of such words. Hence, a total of 190 characters would remain. With three bytes and the extended character set, it is possible to represent up to 6,858,999 different words.

The M190 dictionary (M190DICT) contains 178,343 words and is 1.56 MB in size. M190DICT has been compiled using Webster's Unabridged dictionary, some text files from the Project Gutenberg, a name dictionary, various computer transcriptions, and various Internet sources. The sources in compilation sum up to 46.24 MB. The dictionary has been created as follows. The compiled text file is scanned sequentially and words are ordered based on their occurrence frequencies. The dictionary is then created by sorting the frequencies in descending

order and writing one copy of each word in a text file at the position dictated by this ordering. Besides the regular words, M190DICT also contains some characters or short abbreviations. These come about due to the use of various computer transcription files while compiling the dictionary. M190DICT also includes all upper and first letter uppercase versions of certain words since M190 does not use modified capital conversion.

The basic order of processing for the M190 preprocessor can be summarized as follows: (i) encoding represents punctuation marks and separators as they are, (ii) unknown words are escaped with the 1-byte 127th ASCII character, and represented in plain form; similarly, the end of an unknown word is also marked with the 127th ASCII character, (iii) words in the dictionary are given a codeword depending on their rank using a radix-190 number; the alphabet used by the M190 preprocessor is provided in Figure 1. Note that it is possible to redesign M190DICT to include non-English words from other languages; however, this expansion in the dictionary would lead to slower encoding.

Figure 1. Ordered Character Set

3.1. Encoder for M190

The M190 encoder does not replace the space and punctuation characters and would only administer word encoding. A word for M190 starts when an alphanumeric character is read and ends when a space character or a punctuation character is encountered in the sequential scanning. Words that are not in the dictionary are coded by placing a flag before and after writing the words into the encoded file. The flag used by M190 is the 127th ASCII character. Encoding for M190 starts by scanning a text file and every time a word is encountered its position in the dictionary is determined. If this value is in the range 1–189, one character (1-byte) that corresponds to this word's position is selected from the ordered extended alphabet as the codeword (please refer to Figure 1). If the word's position is in the range 190–36,099, two characters from the alphabet are used. For words with position values in the range 36,099–6,858,999, the encoder uses three characters.

3.2. Decoder for M190

Given a codeword, the task of the decoder is to generate a position in the dictionary based on the characters that make up the codeword and output the actual word corresponding to this position. For example, if the codeword to decode is ABE, then the position could be computed as shown in Figure 2.

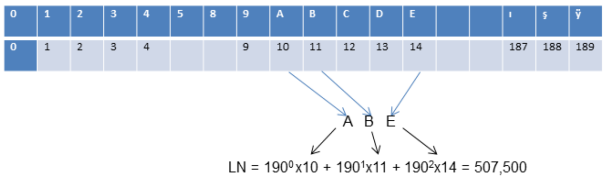


Figure 2. Calculating the position in dictionary.

3.3. Encoder/decoder example

This section demonstrates M190's encoding/decoding processes using the proverb "he who hesitates is lost". If we assume that the proverb is in a text file and the encoder is scanning it for words, every time a word is encountered its position in the dictionary (line number) is determined. As shown in Table 1, the word "he" is at position 36 and "who" is at position 129. Position values for spaces and punctuations (the full stop) are not needed because the encoder does not map these characters to codewords. When the position of the word to encode is less than 190, only one character from the extended alphabet will be used as the codeword. For positions in the range 190–36,099 the codeword would contain two characters. For example, the word "he", which is at position 36, will be encoded using the 'a' symbol and the word "lost", which is at position 1019, will be coded as ‡5. In the decoder block, the codeword '7', which is one-byte long, must have a position value that equals $190^0\times$ (position of character in extended alphabet). This value is 7 and it corresponds to the word "is" in the dictionary.

Table 1. Codeword assignment for words in a proverb.

		Space		Space		Space		Space		Punctuation
proverb	he		who		hesitates		is		lost	.
line number	36	-	129	-	102,860	-	7	-	1019	-
codeword	a		Ã		ˆ ã 2		7		‡5	.

4. Performance evaluation

In this section we compare the compression effectiveness of some well-established preprocessors and M190. Bit-, byte-, or word-based preprocessing algorithms have been evaluated in concatenation with PPAs such as PPMD and PPMonstr. During experiments PPMD with order-4 and PPMonstr with order-8 and memory limit of 256 MB was assumed. The compression effectiveness is given in terms of BPC and is computed using (2):

$$\text{BPC} = \frac{\text{Compressed file size (bytes)}}{\text{Original file size (bytes)}} \times 8 \tag{2}$$

Table 2 provides a list of the corpora, preprocessing algorithms, and PPAs that have been used in this study. For each entry in the table there is a web address from which the source codes can be downloaded. Executable files for encoder and decoder blocks of M190 can be downloaded from the web address also provided in Table 2. We would like to point out that M190 does not use a separate dictionary file but instead it embeds the dictionary into the code in order to speed up the encoding/decoding processes. Note that when dynamic memory allocation functions are used and the dictionary is embedded into the code using a preclassified form there would be no need for gigantic nested loops and temporary variables and this would help lower the encoding and decoding times.

The first two experiments analyze the frequency of punctuation characters in the aggregate text file called *CALGARY.TXT* [36] and also provide distribution of character types (punctuations/space/other) for text files and computer transcriptions selected from the Calgary, Gutenberg, and Canterbury corpora. Results from the first experiment show that, among the 31 punctuation characters depicted, only 14 have a percentage of occurrence above 0.1% and 2 are 1% or above. Results of the second experiment are given in Table 3. For the 13 text files and 3 computer transcriptions, the percentage of punctuation characters plus the space characters would fluctuate from 21% to 43% depending on the file. Note that, among the tested files, the percentage of punctuation plus space characters is highest for computer transcriptions, namely *progc*, *progl*, and *progp*.

Table 2. Corpora and source codes used in experiments.

Corpora	
Calgary corpus	http://corpus.canterbury.ac.nz/descriptions/
Gutenberg	http://www.promo.net/pg/
Canterbury	http://corpus.canterbury.ac.nz/descriptions/
Pizza and Chili	http://pizzachili.dcc.uchile.d/texts/nlang/
Preprocessing algorithms	
StarNT	https://code.google.com/p/starnt/source/
M190	http://faraday.ee.emu.edu.tr/eaince/downloads.html
WRT4.6	http://www.ii.uni.wroc.pl/ inikep/research/WRT/WRT46.zip
ETDC	http://vios.dc.fi.udc.es/codes/files/etdc.tar.gz
SCDC	http://vios.dc.fi.udc.es/codes/files/scdc.tar.gz
Standard postprocessing algorithms	
BZIP2	bzip2 under 7zip
PPMD	http://compression.ru/ds/
PPMonstr	http://compression.ru/ds/
Other	
mPPM	http://www.infor.uva.es/ jadiego/download.php

Table 3. Distribution of character types in each file.

FILE	Punctuations (%)	Spaces (%)	Remaining (%)
bib	9.70	17.99	72.31
book1	4.51	18.49	76.99
book2	6.11	16.93	76.97
news	9.83	17.61	72.56
paper1	8.82	16.65	74.53
paper2	4.31	16.92	78.77
progc	16.11	24.37	59.51
progl	20.28	23.59	56.13
progp	14.73	28.31	56.96
1musk10	4.67	18.68	76.65
alice29	5.59	21.89	72.51
anne11	4.00	19.39	76.61
asyoulik	4.01	21.07	74.92
bible	3.02	19.68	77.30
dickens	4.26	18.84	76.90
lcet10	4.28	17.83	77.89

Tables 4 and 5 provide compression effectiveness of LIPT, StarNT, WRT, M190, ETDC, SCDC, and RPBC when they are used prior to PPMD and PPMonstr. Experiments consider only a subset of the Calgary corpus [29]. Note that column five of Table 4 provides results for the *Universal Processor* of Abel and Teahan [37] concatenated with (PPMD+)[23]. The Universal preprocessor does not require an external dictionary and is known to work for all languages that are Latin based. The last column of Table 4 has been reserved for compression results with mPPM [28]. Since mPPM first maps words into two-byte codewords and then encodes the codewords using conventional PPM (two-stage compressor), the authors chose to compare it in this table with results obtained from other preprocessors concatenated with PPMD. A quick look at Table 4 indicates that M190+PPMD, WRT+PPMD, and StarNT+PPMD are the three best performing methods. Their respective average BPCs are 1.87, 1.92, and 1.93. Table 5 provides BPC values for M190+PPMonstr and

other preprocessors such as LIPT, StarNT, WRT, ETDC, and SCDC in concatenation with PPMonstr. Again M190+PPMonstr, WRT+PPMonstr and StarNT+PPMonstr provide better compression in comparison with the other methods. Their respective BPCs are 1.61, 1.65, and 1.80. This implies that M190+PPMonstr has respective gains of 2.42% and 10.6% over WRT+PPMonstr and StarNT+PPMonstr.

Figures 3 and 4 provide grouped bar graphs for the data presented in Tables 4 and 5. They have been provided to show the degree to which each source file can be compressed by the competing preprocessors when each preprocessor is used prior to PPMD or PPMonstr. It can be seen from Figure 3 that when the postprocessor is PPMD, M190 attains lower BPC values while compressing 'bib', 'news', 'paper1', 'paper2', 'progc', 'progl', and 'progp' and WRT is good for 'book1' and 'book2'.

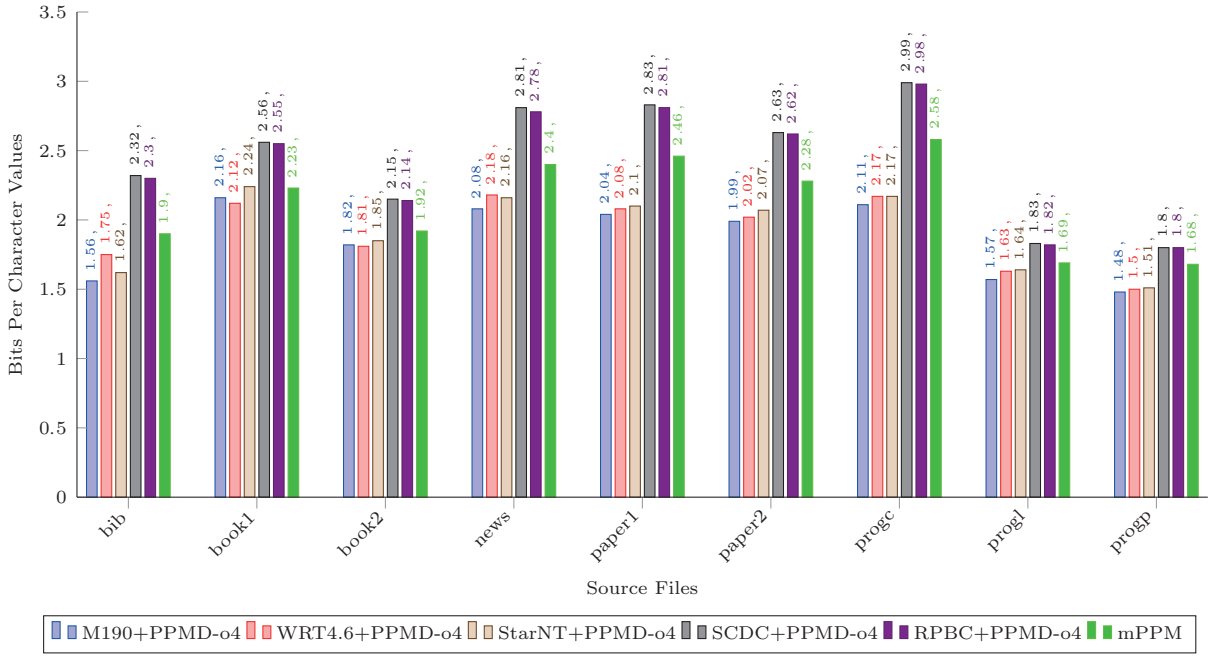


Figure 3. Comparison of preprocessors plus PPMD and stand-alone mPPM using files from the Calgary corpus.

Similarly with PPMonstr as the PPA, M190 gets lower BPCs for 'bib', 'news', 'paper1', 'paper2', 'progc', and 'progl'. On the other hand, WRT excels in compressing 'book1' and 'book2'. For 'progp' BPC values for M190 and WRT are identical. Note that results for ETDC and LIPT have been excluded from Figures 3 and 4 since ETDC has the highest average BPC among the semistatic methods and LIPT among the dictionary-based methods.

Tables 6 and 7 provide BPC results for text files from the Gutenberg [30] and Canterbury [31] corpora when preprocessors are concatenated with PPMD and PPMonstr. During experiments WRT and M190 would provide the two highest compression gains. When the postprocessor is PPMD, average BPC values for WRT, M190, and StarNT are respectively 1.87, 1.88, and 1.89. This shows that WRT+PPMD has 1.07% shorter compressed-file size than StarNT+PPMD and 0.53% shorter compressed-file size than M190+PPMD. Similarly, when the postprocessor is PPMonstr, the respective average BPC values are 1.66, 1.68, and 1.75. This implies that StarNT+PPMonstr has 3.55% longer compressed-file size than M190+PPMonstr, and M190+PPMonstr has 1.81% longer compressed-file size than WRT+PPMonstr. It has been stated in [17] that, while compiling the dictionary for WRT, a training corpus of 3 GB has been taken from Project Gutenberg. This explains the

Table 4. Comparison of preprocessors when they are concatenated with PPMD.

FILE	SIZE	LIPT	StarNT	Universal	WRT4.6	WRT4.6	M190	ETDC	SCDC	RPBC	mPPM
		+	+	+	+	+	+	+	+	+	
		PPMD-o5	PPMD-o5	(PPMD+)	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	
	(bytes)	BPC [16]	BPC [16]	BPC [23]	BPC	BPC	BPC	BPC	BPC	BPC	BPC
					(Aspell dict)[39]	(M190 dict)					
bib	111,261	1.83	1.62	1.85	1.69	1.75	1.56	2.33	2.32	2.30	1.90
book1	768,771	2.23	2.24	2.20	2.10	2.12	2.16	2.57	2.56	2.55	2.23
book2	610,856	1.91	1.85	1.91	1.81	1.81	1.82	2.16	2.15	2.14	1.92
news	377,109	2.31	2.16	2.34	2.23	2.18	2.08	2.82	2.81	2.78	2.40
paper1	53,161	2.21	2.10	2.28	2.03	2.08	2.04	2.86	2.83	2.81	2.46
paper2	82,199	2.17	2.07	2.23	2.03	2.02	1.99	2.66	2.63	2.62	2.28
progc	39,611	2.30	2.17	2.32	2.25	2.17	2.11	3.04	2.99	2.98	2.58
progl	71,646	1.61	1.51	1.62	1.55	1.50	1.48	1.83	1.80	1.80	1.68
progp	49,379	1.68	1.64	1.66	1.67	1.63	1.57	1.86	1.83	1.82	1.69
Av BPC		2.03	1.93	2.05	1.93	1.92	1.87	2.46	2.44	2.42	2.13

Table 5. Comparison of preprocessors when they are concatenated with PPMonstr.

FILE	SIZE	LIPT	StarNT	WRT4.6	WRT4.6	M190	ETDC	SCDC	RPBC
		+	+	+	+	+	+	+	+
		PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr
	(bytes)	BPC	BPC	BPC	BPC	BPC	BPC	BPC	BPC
				(Aspell dict)	(M190 dict)				
bib	111,261	1.81	1.63	1.46	1.51	1.32	2.04	2.04	2.03
book1	768,771	2.19	2.07	1.90	1.90	1.95	2.34	2.34	2.33
book2	610,856	1.91	1.72	1.58	1.59	1.62	1.94	1.94	1.93
news	377,109	2.14	2.05	1.91	1.86	1.76	2.48	2.47	2.46
paper1	53,161	2.08	2.00	1.80	1.85	1.82	2.59	2.57	2.57
paper2	82,199	2.28	1.97	1.82	1.81	1.80	2.44	2.42	2.42
progc	39,611	2.24	2.04	1.94	1.88	1.84	2.72	2.69	2.68
progl	71,646	1.59	1.33	1.23	1.19	1.19	1.61	1.59	1.59
progp	49,379	1.64	1.41	1.27	1.25	1.22	1.59	1.56	1.56
Av BPC		1.99	1.80	1.66	1.65	1.61	2.20	2.18	2.17

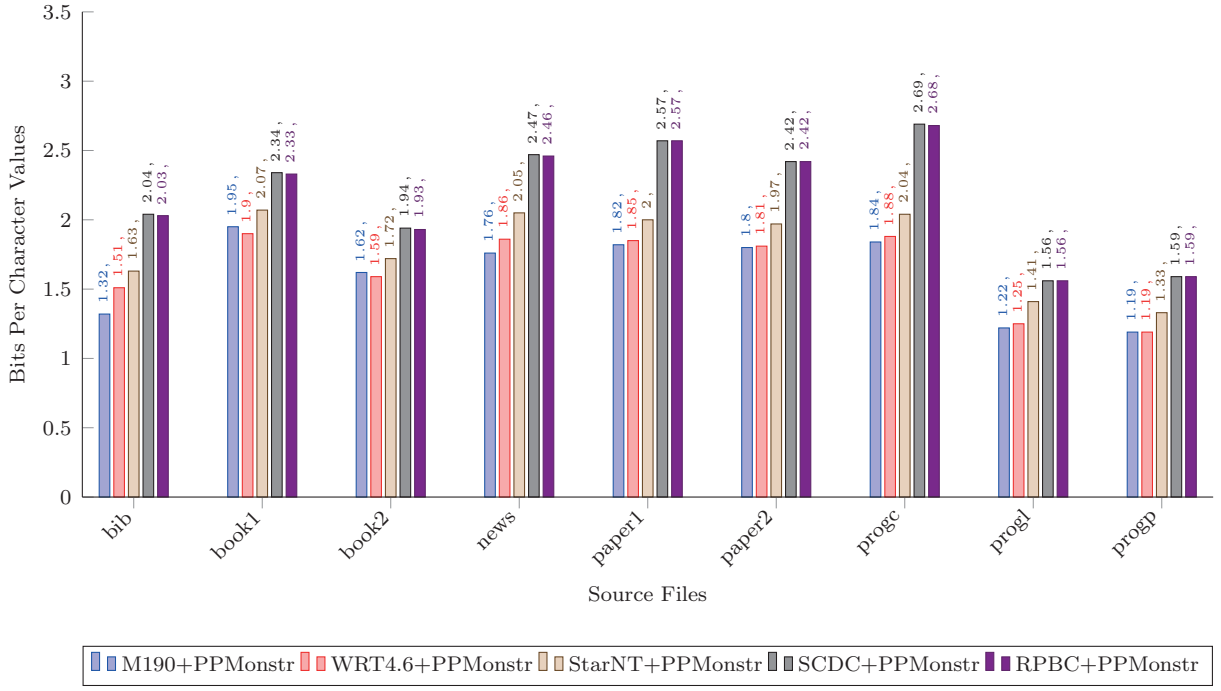


Figure 4. Comparison of preprocessors plus PPMonstr using source files from the Calgary corpus.

lower BPC values when WRT is using the Aspell dictionaries. It is expected that better training would lead to enhanced compression gains. Note that the data provided in Table 7 have been plotted in Figure 5 and show how each source file is compressed by the preprocessor plus PPMonstr pair. The data in Table 6 were not plotted since BPC values with PPMD as postcompressor were lower.

The paper also compares dictionary-based preprocessors, WRT and M190, against the byte-oriented semistatic methods (SCDC and RPBC) using seven medium-to-large size text files. The first four files, which had names *wealthnations*, *warpeace*, *big* and *dickens*, were from Project Gutenberg and had sizes of 2.12, 4.23, 6.3, and 30 MB. The files named *english50* and *english200* were taken from the Pizza and Chili corpus. *WSJ100* text file, which is 100 MB in size, was taken from the TREC Project archives and this is the only text file not related to Project Gutenberg. Figure 6 provides a comparative bar graph that shows the BPCs attained by

Table 6. Compression effectiveness of preprocessor + PPMD while using files from the Gutenberg and Canterbury corpora.

Source	FILE	SIZE	LIPT	StarNT	WRT4.6	WRT4.6	M190	ETDC	SCDC	RPBC
			+	+	+	+	+	+	+	+
			PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4	PPMD-o4
		(bytes)	BPC [15]	BPC	BPC	BPC	BPC	BPC	BPC	BPC
					(Aspell dict)	(M190 dict)				
Gutenberg	1musk10	1,349,139	1.85	1.82	1.72	1.80	1.84	2.03	2.03	2.02
Gutenberg	anne11	587,051	2.04	2.01	1.91	2.00	2.03	2.27	2.26	2.25
Canterbury	alice29	152,089	2.06	2.00	1.90	1.96	1.96	2.37	2.35	2.34
Canterbury	asyoulik	125,179	2.35	2.24	2.24	2.20	2.21	2.77	2.75	2.74
Canterbury	lect10	426,754	1.86	1.78	1.72	1.76	1.75	2.07	2.06	2.05
Large	bible	4,047,392	1.57	1.47	1.46	1.48	1.50	1.52	1.52	1.52
Av BPC			1.96	1.89	1.83	1.87	1.88	2.17	2.16	2.15

Table 7. Compression effectiveness of preprocessor + PPMonstr while using files from the Gutenberg and Canterbury corpora.

Source	FILE	Size	LIPT	StarNT	WRT4.6	WRT4.6	M190	ETDC	SCDC	RPBC
			+	+	+	+	+	+	+	+
			PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr	PPMonstr
		(bytes)	BPC	BPC	BPC	BPC	BPC	BPC	BPC	BPC
					(Aspell dict)	(M190 dict)				
Gutenberg	1musk10	1,349,139	1.83	1.70	1.56	1.64	1.68	1.84	1.83	1.83
Gutenberg	anne11	587,051	1.98	1.88	1.71	1.81	1.85	2.09	2.08	2.08
Canterbury	alice29	152,089	1.99	1.87	1.70	1.76	1.80	2.19	2.17	2.17
Canterbury	asyoulik	125,179	2.21	2.12	1.98	1.94	1.98	2.53	2.51	2.51
Canterbury	lect10	426,754	1.77	1.68	1.52	1.55	1.55	1.88	1.88	1.87
Large	bible	4,047,392	1.58	1.32	1.25	1.26	1.28	1.34	1.33	1.33
Av BPC			1.89	1.75	1.62	1.66	1.68	1.98	1.97	1.96

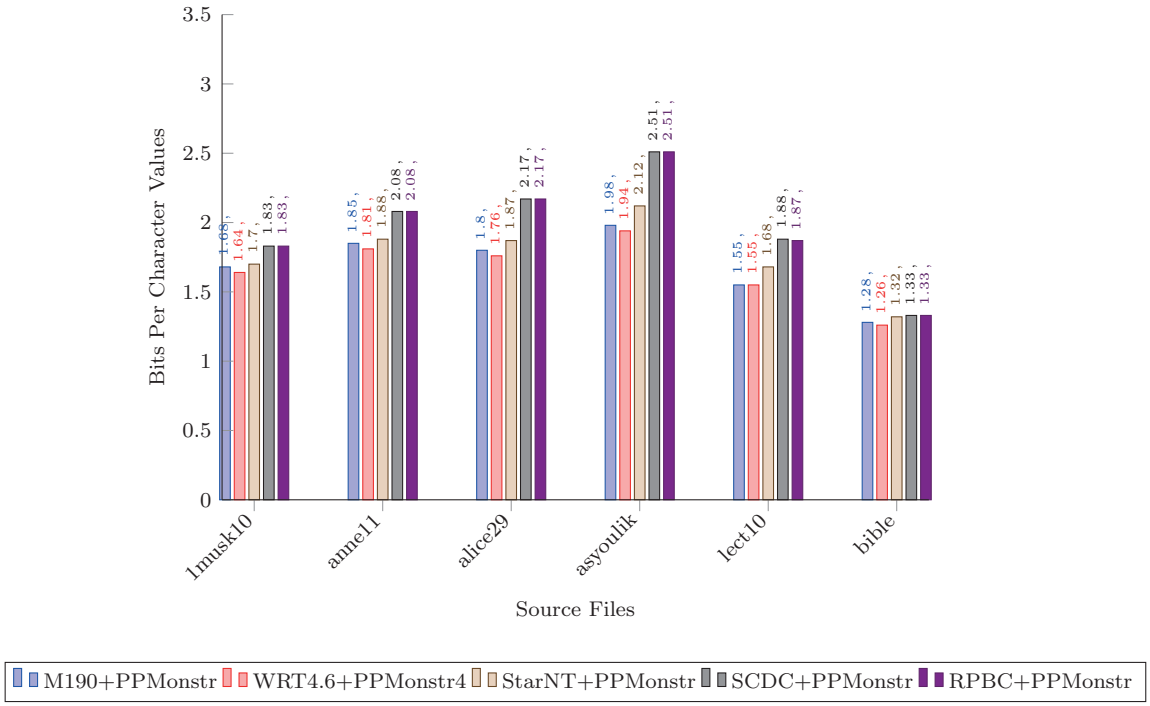


Figure 5. Comparison of preprocessors plus PPMonstr using source files from the Gutenberg and Canterbury corpora.

the algorithms considered when they are concatenated with PPMonstr. The average BPCs for SCDC, RPBC, M190, and WRT are respectively 1.50, 1.50, 1.45, and 1.42. Clearly, both WRT and M190 achieve higher average gains than the semistatic methods. For the 100 MB WSJ100 file, which is not from the Gutenberg Project Library, the BPC difference between WRT and M190 is 0.01. This corresponds to a difference of 105 KB in compressed file sizes. Also note that as the file sizes grew the difference between dictionary-based and semistatic methods would become less significant. However, since most of the time the files one would like to

exchange are smaller than 200 MB, it is fair to say that for small to moderately large files the dictionary-based methods would outperform the semistatic byte-oriented methods.

Finally, Figure 7 provides the encoding and decoding times and speeds for various preprocessors and some well known PPAs. Time measurements are in seconds and are obtained as ensemble average values for 10 runs using a 2.5 GHz Intel core i5 CPU supported by 3 GB of RAM. Speed measurements are in megabytes per second. The source files used in the experiments are the same as in Table 1 of [38]. Encoding/decoding times depicted in subplot 7-(a) show that M190 has smaller encoding time than WRT, RPBC, PPMD-o4, and PPMonstr but is approximately 6.7% slower than BZIP2. When static dictionary-based preprocessors are compared with semistatic byte-oriented preprocessors, we see that both WRT and M190 need four times as much encoding time. As for decoding, M190 is faster than PPMonstr, requires the same time as PPMD-o4, and is 0.01 seconds slower than both BZIP2 and WRT. ETDC and SCDC have the least decoding time in comparison with the other algorithms. It is clear from subplot 7-(b) that techniques requiring less time have the highest speed values.

5. Summary and conclusions

The paper has proposed a new preprocessor called M190 that can help improve overall compression when it is used prior to well-known PPAs. Experimental results have shown that while compressing source files from the Calgary, Canterbury, and Gutenberg corpora, M190 and WRT (two static word-based preprocessors) can always attain higher compression in comparison to the semistatic byte-oriented methods: ETDC, SCDC, and RPBC. With the Calgary corpus, M190 would outperform all preprocessors including WRT regardless of the postcompressor it is coupled with. For six medium-to-large-size files from Project Gutenberg, the Pizza and Chili corpus, and WSJ archive, both M190 and WRT would again provide lower average BPCs in comparison to semistatic byte-oriented methods. Among themselves, WRT would outperform M190 for Project Gutenberg

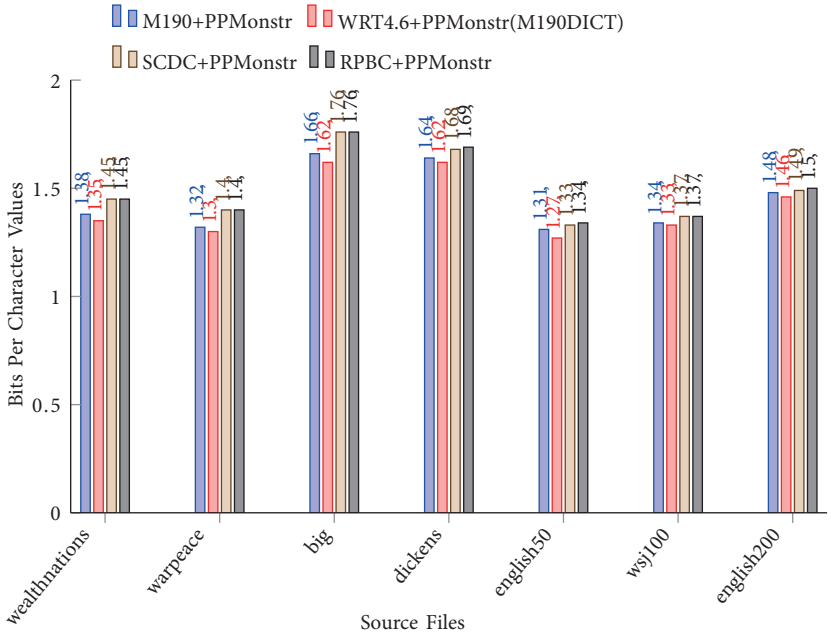
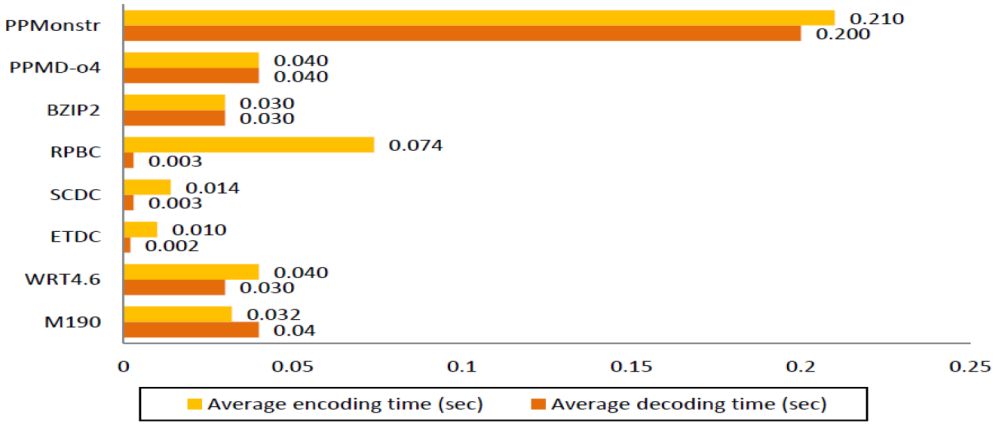
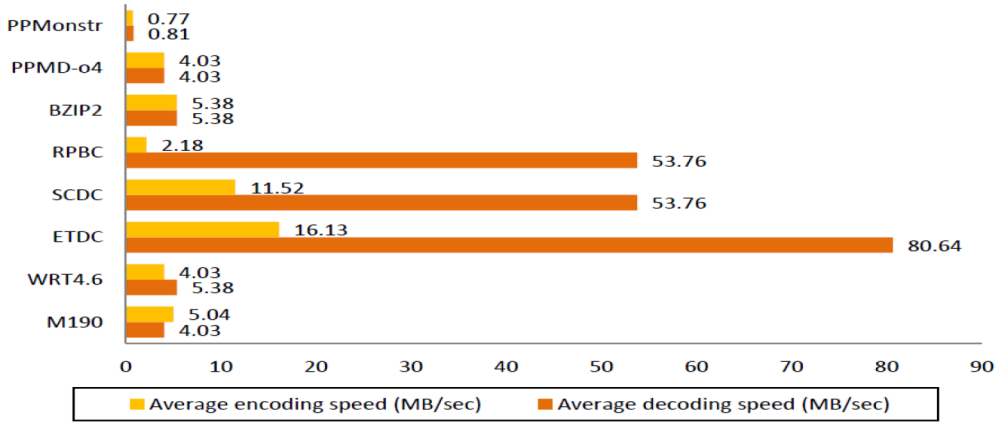


Figure 6. M190 and WRT4.6 compared with word-based byte-oriented preprocessors.



(a)



(b)

Figure 7. Comparison of encoding/decoding (a) times, (b) speeds.

related files and for WSJ100, which is not Project Gutenberg related, the difference between M190 and WRT is marginal. Finally, in all three experiments ETDC and SCDC have the lowest encoding and decoding times. To conclude, the proposed M190 preprocessor has simple logic, it can provide good compression gains in comparison to other state-of-the-art preprocessors, and it also has an acceptable time complexity.

Acknowledgments

Authors would like to thank the anonymous reviewers for their constructive comments and suggestions. In addition, they thank Dr Antonio Farina, Dr Miguel A Martinez-Prieto, Dr Gonzalo Navarro, and Dr Susana Ladra Gonzales for their kind cooperation.

References

[1] Huffman DA. A method for the construction of minimum-redundancy codes. P IRE 1952; 40: 1098-1101.

[2] Gallager RG. Variations on a theme by Huffman. IEEE T Inf Theory 1978; 24: 668-674.

[3] Rissanen J, Langdon GG. Arithmetic coding. IBM J Res Devel 1979; 23: 149-162.

[4] Cleary JG, Witten IH. Data compression using adaptive coding and partial string matching. IEEE T Commun 1984; 32: 396-402.

[5] Mahoney MV. (2004-) The PAQ6 data compression program, 2004-present. Available online at http://mattmahoney.net/dc/paq.html.

[6] Moura E, Navarro G, Ziviani N, Baeza-Yates R. Fast and flexible word searching on compressed text. ACM T Inf Syst 2000; 18: 113-139.

[7] Brisaboa N, Iglesias E, Navarro G, Parama J. An efficient compression code for text databases. In: IEEE 2003 Advances in Information Retrieval Conference; 14–16 April 2003; Pisa, Italy: IEEE. pp. 468-481.

[8] Brisaboa N, Farina A, Navarro G, Parama J. Leight-weight natural language text compression. Inf Ret 2007; 10: 1-33.

[9] Culpepper JS, Moffat A. Enhanced byte codes with restricted prefix properties. In: Springer-Verlag 2005 String Processing and Information Retrieval Conference; LNCS 3772: Springer-Verlag. pp. 1-12.

[10] Brisaboa N, Faria A, Ladra S, Navarro G. Implicit indexing of natural language text by reorganizing bytecodes. J Inf Ret 2012; 15: 527-557.

[11] Ziv J, Lempel A. A universal algorithm for sequential data compression. IEEE T Inf Theory 1977; 23: 337-343.

[12] Welch TA. A technique for high-performance data compression. Comput J 1984; 17: 8-19.

[13] Deutsch P. (1996-) Deflate compressed data format specification, version 1.3, 1951-present. Available online at https://www.ietf.org/rfc/rfc1951.txt.

[14] Nelson MR. (2002-) Star encoding, Dr. Dobb's J., 2002-present. Available online at http://marknelson.us/page/11/.

[15] Awan F, Mukherjee A. LIPT: A lossless text transform to improve compression. In: IEEE 2001 Information Technology: Coding and Computing Conference; April 2001: IEEE. pp. 452-460.

[16] Sun W, Mukherjee A, Zhang N. A dictionary-based multi corpora text compression system. In: IEEE 2003 Data Compression Conference; March 2003: IEEE. pp. 1-11.

[17] Skibinski P, Grabowski S, Deorowicz S. Revisiting dictionary based compression. Softw J : Pract and Exper 2005; 35: 1455-1476.

[18] Golomb SW. Run-length encoding. IEEE T Inf Theory 1966; 12: 337-343.

[19] Burrows M, Wheeler DJ. A Block-sorting lossless data compression algorithm. Digital Systems Research Center, Research Report 124, 1994.

[20] Effros M, Visweswariah K, Kulkarni SR, Verdu S. Universal lossless source coding with the Burrows Wheeler transform. IEEE n Inf Theory 2002; 48: 1061-1081.

[21] Seward J. On the performance of BWT sorting algorithms. In: IEEE 2000 Data Compression Conference; March 2000: IEEE. pp. 173-182.

[22] Moffat A. Implementing the PPM data compression scheme. IEEE T Commun 1990; 38: 1917-1921.

[23] Teahan W. Probability estimation for PPM. In: 1995 New Zealand Computer Science Research Students Conference; University of Waikato, New Zealand, 1995.

[24] Shkarin D. (2001-) PPMd: fast PPM compressor for textual data, 2001-present. Available online at ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdh.rar.

[25] Teahan EJ, Witten IH. Unbounded length contexts for PPM.In: IEEE 1995 Data Compression Conference; March 1995: IEEE. pp. 52-61.

[26] Shkarin D. (2002-) Monstrous PPM II compressor based on PPMd var.I, 2002-present. Available online at http:/www.compression.ru/ds/ppmdi1.rar.

[27] Shkarin D. (2004-) The Durilca and Durilca Light 0.4a programs, 2006-present, Available online at http://www.compression.ru/ds/durilca.rar.

[28] Adiego J, Martinez-Prieto MA, Fuente P. High performance word-codeword mapping algorithm on PPM. In: IEEE 2009 Data Compression Conference; 1 January 2009: IEEE. pp. 23-32.

[29] Witten I, Bell T, Cleary J. (1987-) Calgary corpus. University of Calgary, Canada, 1987-present. Available online at ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus.

[30] Hart MS. (2012-) Project Gutenberg corpus, University of Illinois, 1971-present. Available online at https://www.gutenberg.org/.

[31] Arnold R, Bell T. (1997-) The Canterbury text compression corpora, University of Canterbury, New Zealand. Available online at http://corpus.canterbury.ac.nz/.

[32] Ferragina P, Navarro G. (2005-) Pizza-and-Chili corpus: Compressed Indexes and their Testbeds, University of Chili, 2005-present. Available online at http://pizzachili.dcc.uchile.cl/texts.html.

[33] Harman DK. Text research collection. In: NIST 1994 Second Text Retrieval Conference (TREC-2); March 1994: NIST. pp. 233-242.

[34] Batista L, Alexandre LA. Text pre-processing for lossless compression. In: IEEE 2008 Data Compression Conference; March 2008: IEEE. pp. 506-516.

[35] Silva de Mura E, Navarro G, Ziviani N, Baeza-Yates R. Fast and flexible word searching on compressed text. ACM T Inf Syst 2000; 18: 113-139.

[36] Brisaboa NR, Farina A, Navarro G, Parama JR. Improving semistatic compression via phrase-based modelling. Inf Process Manag 2011; 47: 545-559.

[37] Abel J, Teahan W. Universal text preprocessing for data compression. IEEE T Comput 2005; 54: 497-507.

[38] Sun W, Zhang N, Mukherjee A. Dictionary-based fast transform for text compression. In: IEEE 2003 Information Technology: Computers and Communications Conference; April 2003: IEEE. pp. 176-182.

[39] Atkinson K. (2004-) Spell checking oriented word lists (SCOWL), revision 5, 2004-present. Available online at http://wordlist.sourceforge.net.