Arabian Journal for Science and Engineering (2018) 43:8209–8221
https://doi.org/10.1007/s13369-018-3378-9

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Multi-Stream Word-Based Compression Algorithm for Compressed Text Search

Emir Öztürk[1] · Altan Mesut[1] · Banu Diri[2]

Received: 12 May 2017 / Accepted: 31 May 2018 / Published online: 12 June 2018

Abstract
In this article, we present a novel word-based lossless compression algorithm for text files using a semi-static model. We named this method the 'Multi-stream word-based compression algorithm (MWCA)' because it stores the compressed forms of the words in three individual streams depending on their frequencies in the text and stores two dictionaries and a bit vector as side information. In our experiments, MWCA produces a compression ratio of 3.23 bpc on average and 2.88 bpc for files greater than 50 MB; if a variable length encoder such as Huffman coding is used after MWCA, the given ratios are reduced to 2.65 and 2.44 bpc, respectively. MWCA supports exact word matching without decompression, and its multi-stream approach reduces the search time with respect to single-stream algorithms. Additionally, the MWCA multi-stream structure supplies the reduction in network load by requesting only the necessary streams from the database. With the advantage of its fast compressed search feature and multi-stream structure, we believe that MWCA is a good solution, especially for storing and searching big text data.

Keywords Data compression · Text compression · Dictionary-based compression · Compressed matching · MWCA

1 Introduction

Everyday, many GBs of text data are submitted to the internet, and widespread use of social media has particularly increased this amount. The size of text data on the web is measured in terabytes. With the aid of text compression, this type of data could be stored using less disk space. Text compression is about exploiting redundancies in the text to represent it in less space [1].

With advances in technology, the capacity of storage devices are increasing and the cost is simultaneously decreasing. However, this increase in capacity is not sufficient to meet demand due to the increase in the amount of data produced. At the same time, similar to storage media, the processing power of CPUs is continuously increasing. With the allocation of a certain amount of CPU power, compression could be used to reduce the required storage space and input/output load. Another advantage of compression is a reduction in the search space in the case of compressed search support. Reducing the search space ensures a gain of both CPU time and input/output operations.

It is necessary to be able to access these rapidly increasing data, and at the same time, it is necessary to be able to obtain information from these data when requested. For this reason, large text databases are created and used. Compression algorithms suitable for text databases should meet two requirements. The first requirement is compressed search support, and the second is that the file can be decompressed from a desired location [2]. Text compression algorithms that use a semi-static model can supply these two features. Although general-purpose adaptive compression algorithms are successful in terms of compression ratio, they are generally not suitable for text databases because they do not supply compressed search support and decompression from a desired location [3]. Adaptive compression algorithms with compressed search support are ineffective due to their need for decompression to conduct a search operation [4].

✉ Emir Öztürk
emirozturk@trakya.edu.tr

Altan Mesut
altanmesut@trakya.edu.tr

Banu Diri
banu@ce.yildiz.edu.tr

1 Computer Engineering Department, Trakya University, Edirne, Turkey

2 Computer Engineering Department, Yildiz Technical University, Istanbul, Turkey

Although classic Huffman is a known character-based technique, it is not suitable for text databases due to its poor compression ratio [2]. Moffat suggests the use of words as symbols to solve this problem [5]. Additionally, the use of words instead of characters is more suitable for text-retrieval systems [2]. Because methods such as ETDC and SCDC support compressed text search and decompression from the desired point of the stream, they can be used effectively in text databases by addressing the limitations of general-purpose algorithms.

This work introduces a word-based compression algorithm that uses a semi-static model known as MWCA. MWCA aims to increase the search efficiency with a multi-stream structure in addition to the compressed search support. This algorithm is designed to reduce the search space by offering compressed search support (such as ETDC) and using only the required streams. The search space, which is previously minimized with the aid of compression, is further reduced by only requesting the required streams. Thus, this approach aims to reduce the network and input/output loads. The CPU time required for the search operation is also reduced because decompression is not used.

2 Related Work

This section describes general-purpose and word-based compression methods. In addition, general string-matching algorithms are addressed. Text files are commonly compressed with lossless compression methods to avoid loss of information. General-purpose compression algorithms such as LZ77, LZ78, LZW, LZMA, Deflate, and the PPM family (PPMa, PPMb, PPMd) and word-based compression algorithms such as ETDC and SCDC are widely used in lossless text compression.

The LZ77 [6] and LZ78 [7] algorithms were developed by Abraham Lempel and Jacob Ziv. The LZ77 algorithm works by maintaining a history window of the most recently viewed data and comparing the current data under encoding with the data in the history window. The information that is actually placed into the compressed stream is referenced to the position in the history window and the length of the match. The LZ78 algorithm searches the longest match of a substring (phrase) in the dictionary that is constructed on the fly. When the longest match is found, the algorithm encodes the dictionary index, forms a new phrase by adding the next character in the source file to the end of the matching substring and adds this phrase to the dictionary. The LZW (Lempel Ziv Welch) algorithm, which was developed by Terry Welch in 1984, is a variant of LZ78 [8]. LZMA (Lempel Ziv Markov algorithm), which is the native compression mode of the 7-zip archiver, is a variant of LZ77. Deflate, which is used in ZIP archiving tools (pkzip, gzip), is an algorithm that combines LZ77 and Huffman coding [9]. PPM (prediction by partial matching) is an adaptive statistical data compression method proposed by Cleary and Witten in 1984 [10]. PPM attempts to predict the probability that a character is in a specific location from the previously occurring n symbols.

Word-based text compression methods encode the words in the text as symbols, and high compression ratios can be gained based on the characteristics of frequency distribution in natural language data [5,11]. Based on this foundation, algorithms such as word-based Huffman code [12], plain Huffman code [13], tagged Huffman code [13], End-tagged dense code (ETDC) and (s,c)-dense code (SCDC) [14] are proposed.

In ETDC, the words that are acquired using a spaceless word model [5] are sorted according to their frequencies in the first pass. The first 128 are represented with one byte. Two-byte codewords are applied for the words from 129 to $128^2 + 128$. If the source file has additional individual words, codewords of three or more bytes can be used. The first bit of each byte is a flag. If the flag is 1, it indicates that this byte is the last byte of the codeword. In other words, one-byte codewords are between 10000000 and 11111111, and two-byte codewords are between 000000000:10000000 and 01111111:11111111. Although ETDC uses 128 symbols for the bytes that do not end a codeword (continuers) and the other 128 symbols for the last byte of the codeword (stoppers), SCDC adapts the number of stoppers and continuers to the word frequency distribution of the text. In SCDC, s values are used as stoppers, and $c = 256 - s$ values are used as continuers. For example, the first 230 words could be encoded with 1-byte codewords, and the remaining words could be encoded with 2-byte codewords ($s = 230$, $c = 26$). Using this scheme, 230+26*230 = 6210 words could be encoded using only 1- and 2-byte codewords.

Dynamic versions of ETDC and SCDC (DETDC and DSCDC) are better than semi-static ETDC and SCDC in terms of compression time [15], but they are worse in terms of decompression time and slightly worse in compression ratio. The subsequently developed dynamic lightweight dense codes (DLETDC and DLSCDC) are able to perform decompression as quickly as semi-static dense codes [16]. However, the compression speeds and compression ratios of dynamic lightweight dense codes are worse than those of the dynamic dense codes. In summary, semi-static methods are best in terms of compression ratio and decompression time, and dynamic methods are best in terms of compression time. Carus and Mesut proposed another compression algorithm that uses a semi-static model (CAFTS: compression algorithm for fast text search) [17]. CAFTS produces a main dictionary that contains several sub-dictionaries in the first pass. Each sub-dictionary contains the most frequently used tri-grams after a particular di-gram. The algorithm selects the appropriate sub-dictionary by looking at the last encoded di-gram, and the corresponding index is encoded in the second pass. Carus and Mesut also show that word-based compression methods such as dense codes are not efficient for agglutinative languages such as Finnish and Turkish because the numbers of different words in these languages are greater than those of other languages.

Word-based text compression algorithms such as ETDC and SCDC could be used as a preprocessor for general lossless compression algorithms to gain more efficient compression ratios [18,19]. However, transformation algorithms such as BWT [20], LIPT [21], StarNT [22] and WRT [23] could be more effective in the pre-processing stage.

To conduct the search operation on compressed text files, the file can be decompressed prior to searching, or the search operation can be performed directly over the compressed file if the compressed form of the text is suitable for pattern matching. Direct pattern matching on compressed texts is generally known as compressed pattern matching, and it was first defined in the work of Amir and Benson [24]. Searching on the compressed file could be up to 8 times faster than searching on the original text file [13,25]. General string-matching algorithms such as Boyer Moore [26], Berry Ravindran [27], Horspool [28] and KMP [29] can be used with compressed texts instead of brute force search.

Although the LZMA and PPM compression algorithms give better results for the compression ratio than LZ78/LZW and word-based text compression algorithms, they are not suitable for fast compressed pattern matching. LZGrep, which was proposed by Navarro and Tarhio in 2005 [30], could be used in compressed string matching over texts that are compressed with LZW-based Unix Compress. LZgrep can also search files compressed with Unix Gzip using a fast decompress-then-search technique. Although decompress-then-search techniques are useful when decompression is fast, direct searching on a compressed file is more effective [31].

3 Proposed Method

In this section, we propose a word-based text compression method using semi-static dictionaries (MWCA). The descriptions of our compression and decompression algorithms are given in the following subsections.

3.1 MWCA Encoding

MWCA creates two dictionaries. The first dictionary consists of 255 words, and thus words in this dictionary can be represented with one byte. We reserve the index ‘0’ in the dictionary for the escape situation. The second dictionary contains 65536 words that are represented with two bytes.

The dictionaries are created by obtaining the words and their frequencies from the given file. In the first pass, the words and their frequencies are acquired, and the words are ordered by their frequencies. In the encoding stage, MWCA creates the first dictionary with the most frequent 255 words and the second dictionary with the next 65536 most frequent words. To obtain the words, MWCA searches for a sequence of punctuation or alphanumeric characters. MWCA uses ‘spaceless word model’ and assumes all non-alphanumeric characters as punctuation. A sequence can be described as a group of successive characters with the same type. The encoder reads a sequence to obtain a word until it encounters a character with a different attribute. If given word is present in the dictionary, its frequency is increased by 1. A space that comes after an alphanumeric sequence is not encoded. In the decoding stage, the decoder looks for the attribute that can be ‘alphanumeric’ or ‘punctuation’. If two alphanumeric sequences are read from the encoded file, the decoder places a space character in the output file. To obtain a distinct word alphabet, MWCA uses a hash implementation used in ETDC. We determine the hash size according to the file size using Heaps’ law [32].

After the dictionaries are created, the encoding phase begins. The MWCA stores the encoded words in three different streams. The first stream contains one-byte codewords of the words present in *D1*, and the second one contains two-byte codewords of the words present in *D2*. The third stream contains the words that are not present in either dictionary. We refer to these streams as *S1*, *S2* and *S3*, respectively. The encoder also generates a bit vector *BV* to store the information related to which dictionary is used to encode the words (we use 0 for *D1* and 1 for *D2*). The function C describes the code of the word, meaning that C(*word*) is a one- or two-byte code of the word. The file to be compressed is read in pieces using a buffer of given size to obtain the words ($\sum_{i=0}^{n} W_i$).

When the encoder encounters a word W_i, we look up the word in *D1*. If $W_i \in D1$, we insert a ‘0’ bit into *BV* and C(W_i) into *S1*. If $Wi \notin D1$, we look up if $W_i \in D2$. If *D2* contains the word W_i, a ‘1’ bit is inserted into *BV*, and the two-byte code C(W_i) of the word is sent to *S2*. If the word W_i is not present in both dictionaries, a ‘0’ bit is inserted into *BV*, and a null character (NUL) is inserted into *S1*. This NUL indicates the escape character, and the word is inserted into the *S3* stream without encoding. The words in *S3* are separated with an NUL. The created streams could be saved as separate files or as a single file after the encoding process. Huffman coding [33] could be applied to the dictionaries, streams and the bit vector *BV* to increase the compression ratio. The pseudocode of MWCA encoding is given in Fig. 1.

As an example, we explain the compression of the sentence The book is composed of text, footnotes, and appendices’ as presented in one of our test files English50MB.txt. The dictionaries are generated according to the word fre-

```
1.  Find all sequences and acquire their frequencies
2.  Sort all sequences by their frequencies
3.  Take 255 for first dictionary, skip 255 and take 65536 for second dictionary
4.  Read one alphanumeric or punctuation sequence from input file
5.  Search the code of the word C(W_i) in dictionaries
6.  If W_i is present in one of the dictionaries
7.          Write '0' or '1' to BV related to dictionary number
8.          Write C(W_i) to S1 or S2 related to dictionary number
9.  IF W_i is not present in the dictionaries
10.         Write '0' bit to BV
11.         Write '0' byte to S1
12.         Write W_i to S3 with space (W_i_)
13. Repeat steps from 5 to 12 While not End of Stream
```

Fig. 1 MWCA encoding algorithm

Table 1 Words and assigned codes

First dictionary (*D1*)	
,_	5 (00000101)
and	6 (00000110)
of	7 (00000111)
is	25 (00011001)
The	33 (00100001)
Second dictionary (*D2*)	
book	664 (0000001010011000)
text	1338 (0000010100111010)
composed	2235 (0000100010111011)
footnotes	41,884 (1010001110011100)

quencies. The related words and their codes taken from the dictionaries of English50MB.txt are given in Table 1.

In the beginning, the encoder reads the first word ‘The’ and searches for it in the first dictionary *D1*. This word is found in *D1* at position 32, and therefore, we insert ‘0’ into *BV* and ‘00100001’ into *S1*. Subsequently, we read ‘book’ and obtain the code C(‘book’) as ‘0000001010011000’. The word is in *D2*, and therefore, we insert ‘1’ into *BV* and ‘0000001010011000’ into *S2*. The encoder next encounters the word ‘is’. For this step, the encoder finds the word ‘is’ in the dictionary and the codeword C(‘is’) as ‘00011001’ in the first dictionary, and writes ‘0’ to *BV* and C(‘is’) into *S1* due to the occurrence in *D1*. The encoder finds the C(‘composed’), which is ‘0000100010111011’ and writes ‘1’ to *BV* and C(‘composed’) to *S2*. The process is the same until the last word ‘appendices’ is encountered. When the encoder finds ‘appendices’, which is not present in *D1* or *D2*, it inserts ‘0’ into *BV* and ‘00000000’ into *S1* as an escape character. Finally, the encoder writes the word ‘appendices’ with one null character into *S3*. The streams after the full process are given in Table 2.

After the process is finished, the encoder writes the dictionaries, streams and bit vector into separate files or into a single file with the side information of their sizes.

Table 2 Contents of streams and bit vector after compression

	Content	Content (decimal)
BV	0 1 0 1 0 1 0 1 0 0 0	
S1	00100001 00011001 00000111 00000101	
	00000101 00000110 00000000	(33, 25, 7, 5, 5, 6)
S2	0000001010011000 0000100010111011	
	0000010100111010 1010001110011100	(664, 2235, 1338, 41,884)
S3	appendicesNUL	

```
1.  Define PW and CW as 'Previous Word' and 'Current Word'
2.  Set PW = 0, CW = 0
3.  Read one bit from BV
4.  Set PW = CW
5.  If BV is '0'
6.          Read one byte from S1
7.          If the byte is '0'
8.                  Read one word (W) from S3
9.          If the byte is not '0'
10.                 Find corresponding word (W) from D1
11. If BV is '1'
12.         Read two bytes from S2
13.         Find corresponding word (W) D2
14. If (W) is alphanumeric set CW = 0
15. Else if (W) is punctuation set CW = 1
16. CW = 0 and PW = 0 write space character to output
17. Write W to output
18. Repeat steps 3 to 17 until the end of BV
```

Fig. 2 MWCA decoding algorithm

3.2 MWCA Decoding

If Huffman coding is applied to the streams or dictionaries at the encoding stage, these items are decoded first at the decoding stage. The MWCA reads symbols (BV_i) from BV, where BV_i can be ‘0’ or ‘1’. If BV_i is ‘0’, one-byte code is read from *S1*. If $S1_i \neq 0$, the word corresponding to the code is read from the dictionary and inserted into the output file. If $S1_i = 0$, it indicates an escape, and the decoder reads a word from *S3* and writes it to the output file. If BV_i is ‘1’, a two-byte code is read from *S2*, and the corresponding element from *D2* is written into the output file. The decoding stage is repeated until it reaches the end of BV. The pseudo-code of MWCA decoding is given in Fig. 2.

For the decoding example, we use the same sentence ‘The book is composed of text, footnotes, and appendices’ that is given in the encoding stage. The decoder first reads ‘0’ from BV, and therefore, it reads a byte from *S1*. The value of this byte is ‘33’, and the process finds the corresponding element ‘The’ from *D1* and writes it into the output file. In the second step, the decoder encounters a ‘1’ bit and reads two bytes from *S2*. The first word in *D2* is ‘book’, and therefore, the decoder must write the ‘book’ word into the output

Table 3 Steps of decoding algorithm

BV	*S1*	*S2*	*S3*	Prev./Pres.	Add.	Output
0	33 (The)			-A		The
1		664 (book)		AA	'_'	The_book
0	25 (is)			AA	'_'	The_book_is
1		2235 (composed)		AA	'_'	The_book_is_composed
0	7 (of)			AA	'_'	The_book_is_composed_of
1		1338 (text)		AA	'_'	The_book_is_composed_of_text
0	5 (,_)			AP		The_book_is_composed_of_text,_
1		41884 (footnotes)		PA		The_book_is_composed_of_text,_footnotes
0	5 (,_)			AP		The_book_is_composed_of_text,_footnotes,_
0	6 (and)			PA		The_book_is_composed_of_text,_footnotes,_and
0	0 Escape		appendices	AA	'_'	The_book_is_composed_of_text,_footnotes,_and_appendices

file. However, before writing this word, the decoder checks whether the previous word is alphanumeric. If the current word and previous word are both alphanumeric, a space char is written into the output file before the second word. Therefore, the output is similar to '_book'. If one of the words is punctuation, a space char is included at the end of this word. Therefore, there is no need to write a space character into the output file. The steps of the decoding algorithm are given in Table 3. Prev./Pres. shows the attribute of the previous and present word as alphanumeric (A) or punctuation (P). If the previous and present words are both alphanumeric (AA), a space character is added before the second word. The Add. column shows when a space character is written into the output file before the second word. The output column shows the content of the decoded file.

4 Searching Words Directly on MWCA Compressed Texts

Our method allows for us to search for a word directly in a compressed file. Our statistical analysis of text files shows that the most frequent words in the files are prepositions or pronouns. The most frequent words in the Calgary, Canterbury and Pizza&Chili corpuses are given in Table 4. As shown in this table, the most frequent words are stop words, and they are less likely to be searched.

It is obvious that the least frequent words are searched more than the most frequent words because searching by prepositions or pronouns is not meaningful. This means that most search operations are performed in our *S2* stream. Using different streams, we can shrink the search space such that the search process is shorter.

Our search method can find the occurrence count of a given word in a MWCA compressed text file. To return the occurrence count of a given word, search algorithm will first search for word W in dictionaries. If the algorithm finds the word in one of the dictionaries, it will obtain the corresponding codeword of W, C(W). If the given word is found in *D1*, the algorithm will search C(W) in *S1* and if the word is found in *D2*, search operation will be carried out on *S2*. If the given word could not be found in either *D1* or *D2*, the algorithm will search W in *S3* with a selected string-matching algorithm. At the end of the procedure, the count of given word will be returned. Steps of search algorithm are given in Fig. 3.

For example, if we search for the word 'and', the algorithm first searches the word in *D1*. After finding the word, codeword C('and') = 6 is obtained from the dictionary. The algorithm searches for the '6' byte in *S1*, and for each occur-

Table 4 The most frequent 40 words in Calgary, Canterbury and Pizza&Chili corpuses

1	the	9	was	17	her	25	at	33	my
2	and	10	he	18	had	26	by	34	me
3	of	11	his	19	him	27	which	35	they
4	to	12	it	20	not	28	this	36	all
5	a	13	with	21	be	29	have	37	were
6	in	14	is	22	on	30	but	38	said
7	I	15	for	23	you	31	from	39	so
8	that	16	as	24	at	32	she	40	one

```
1. Take the word (W) to be searched.
2. SMA <= Preferred string matching algorithm
3. Search W in D1 with SMA
4. If W is found in D1
5.          Find the codeword C(W) corresponding to word W in D1
6.          Search C(W) in S1
7.          Increase count by one for each occurrence of C(W) in S1
8.          Return count
9. Else
10.         Search W in D2
11.         If W is found in D2
12.                 Find the codeword C(W) corresponding to word W in D2
13.                 Search C(W) in S2
14.                 Increase count by one for each occurrence of C(W) in S2
15.                 Return count
16.         Else
17.                 Search W in S3 with SMA
18.                 Increase count by one for each occurrence of W in S3
19.                 Return count.
```

Fig. 3 Pseudo-code of SM2 algorithm

rence, it increases the count by one. If we search for the word ‘book’, the algorithm first looks at *D1*. After it cannot find ‘book’ in *D1*, it searches that word in *D2*. The algorithm obtains a two-byte code with value ‘664’ and searches it in *S2*. In the worst case (e.g., searching for the word ‘appendices’), the algorithm searches *S3* with a preferred string-matching algorithm and counts the occurrences of the word.

5 Experimental Results

We obtained the source codes of ETDC and SCDC for comparison with our method. We also obtained the general-purpose lossless compression algorithm results for LZMA, PPMd, Gzip, Bzip2 and DEFLATE. To obtain these results, we compiled 7z 15.09 x64 from its source, and we used mx1, mx5 and mx9 as the three different compression parameters, where mx1 means ‘the fastest compression’ and mx9 means ‘the highest compression ratio’.

In our experiments, we used a machine with an Intel Core i7 2.5 GHz, 256 KB L2 cache, 6 MB L3 cache processor and 16 GB of RAM. The source codes are compiled with gcc 5.2.1 at the maximum optimization level.

We used a test corpus that contains selected natural language text files from the Calgary, Canterbury, Silesia and Pizza&Chili corpuses and from Project Gutenberg. Our test corpus is given in Table 5 with grouping of the text files according to their sizes.

In our corpus, the first group consists of English text files smaller than 10 MB, the second group consists of 8 files with 30 MB size (each in different languages), and the last group consists of files equal to or larger than 50 MB. The last group contains one multilingual ‘GutenbergCorpus’ file derived from Project Gutenberg and 5 other files in English from the Pizza&Chili corpus.

The compression results for MWCA and the other selected algorithms are given in Tables 6, 7 and 8. We give 4 different MWCA results in these tables. The first result does not use Huffman coding, the second uses Huffman coding only on the dictionaries, the third uses Huffman coding on the dictionaries and *S3* stream, and the last uses Huffman coding on all streams (*D1*, *D2*, *S1*, *S2*, *S3* and *BV*). The best results for each column are given in bold to increase readability.

As shown in Table 6, in most situations, the general-purpose algorithms produce better results than the word-based text compression algorithms. The most successful algorithm in this group is PPMd. Although MWCA is slightly worse than ETDC, if we use Huffman encoding only on *D1* and *D2*, we achieve better results than ETDC and SCDC (SCDC is better only on the Dickens file). Due to the small size of the files, *D2* does not contain 65536 words, and there is no *S3* stream. Therefore, use of the MWCA + Huffman (*D1*, *D2*, *S3*) method does result in any change in the compression ratio.

The size ratios of the dictionaries, streams and bit vector over the compressed files are given in Fig. 4. It is clear that as the file size increases, the weight of the dictionaries over the total size of the files decreases. It is difficult to observe the length of *D1* because it contains only 255 words, and its size is a slightly greater than 1 KB in all test files.

As shown in Table 7, the compression results change with the language of the files. Word-based compression methods ETDC, SCDC and MWCA produce their worst results in Finnish and Turkish because of the agglutinating structure of these languages. PPMd is again the best compression algorithm, and the English file is the most compressible one. This time, using MWCA + Huffman (*D1*, *D2*, *S3*) gives better

Table 5 Test corpus

Group	File	Taken from	Size (byte)
(<10 MB)	Book1	Calgary Corpus	768,771
	Book2		610,856
	News		377,109
	Bible	Canterbury Corpus	4,047,392
	World192		2,473,400
	Dickens	Silesia Corpus	10,192,446
(= 30 MB)	Dutch	Derived from Project Gutenberg	30,792,532
	English		30,792,111
	Finnish		30,792,616
	French		30,792,398
	German		30,792,171
	Italian		30,792,700
	Spanish		30,792,223
	Turkish		30,792,241
(>50 MB)	GutenbergCorpus	Derived from Project Gutenberg	635,293,766
	English 50 MB	Pizza&Chili Corpus	52,428,800
	English 100 MB		104,857,600
	English 200 MB		209,715,200
	English 1024 MB		1,073,741,802
	English 2108 MB		2,210,395,521

Table 6 Compression ratios of Group 1 (bpc)

Compression algorithms	News	Book2	Book1	World192	Bible	Dickens
ETDC	4.87	3.6	3.71	3	2.54	2.75
ETDC + Huffman(vocabulary)	4.23	3.25	3.24	2.80	2.44	2.66
SCDC	4.77	3.47	3.63	2.93	2.43	2.67
MWCA	4.96	3.66	3.79	3.04	2.61	2.8
MWCA + Huffman($D1$, $D2$)	4.31	3.3	3.32	2.84	2.52	2.71
MWCA + Huffman($D1$, $D2$, $S3$)	4.31	3.3	3.32	2.84	2.52	2.71
MWCA + Huffman($D1$, $D2$, $S1$, $S2$, $S3$, BV)	3.96	2.92	2.93	2.47	2.08	2.34
Bzip2 mx1	2.85	2.4	2.82	2.17	1.98	2.59
Bzip2 mx5	2.52	2.06	2.42	1.58	1.67	2.2
Bzip2 mx9	2.52	2.06	2.42	1.58	1.67	2.2
DEFLATE mx1	2.9	2.59	3.18	2.14	2.32	2.98
DEFLATE mx5	2.53	2.23	2.72	1.62	1.76	2.22
DEFLATE mx9	2.52	2.22	2.72	1.58	1.75	2.22
Gzip mx1	3.18	2.83	3.41	2.44	2.48	3.17
Gzip mx5	2.98	2.59	3.13	2.26	2.2	2.89
Gzip mx9	2.97	2.59	3.12	2.25	2.19	2.89
LZMA mx1	2.9	2.59	3.18	2.12	2.31	2.97
LZMA mx5	2.53	2.23	2.72	1.62	1.76	2.22
LZMA mx9	2.52	2.22	2.72	1.58	1.75	2.22
PPMd mx1	2.49	2.02	2.38	1.83	1.65	2.12
PPMd mx5	**2.22**	**1.85**	**2.18**	1.36	1.48	1.84
PPMd mx9	2.36	1.88	2.22	**1.26**	**1.45**	**1.82**

Table 7 Compression ratios of Group 2 (bpc)

Compression algorithms	Dutch	English	Finnish	French	German	Italian	Spanish	Turkish
ETDC	2.76	2.59	3.8	3.04	3.07	3.03	3.13	3.95
ETDC + Huffman(vocabulary)	2.60	2.53	3.35	2.95	2.81	2.85	2.99	3.78
SCDC	2.72	2.54	3.76	2.98	3.04	2.99	3.09	3.88
MWCA	2.76	2.64	3.94	3.09	3.11	3.04	3.15	4.02
MWCA + Huffman(*D1*, *D2*)	2.69	2.58	3.86	3.03	3.04	2.97	3.09	3.96
MWCA + Huffman(*D1*, *D2*, *S3*)	2.59	2.58	3.33	3.01	2.78	2.82	2.99	3.79
MWCA + Huffman(*D1*, *D2*, *S1*, *S2*, *S3*, *BV*)	2.3	2.23	2.91	2.62	2.51	2.53	2.62	3.32
Bzip2 mx1	2.5	2.48	2.66	2.52	2.56	2.68	2.58	2.66
Bzip2 mx5	2.16	2.08	2.35	2.1	2.2	2.37	2.23	2.27
Bzip2 mx9	2.16	2.08	2.35	2.1	2.2	2.36	2.23	2.27
DEFLATE mx1	2.85	2.85	2.99	2.84	2.9	3.03	2.92	2.92
DEFLATE mx5	2.11	1.78	2.28	2.05	2.05	2.24	2.17	1.9
DEFLATE mx9	2.11	1.78	2.27	2.04	2.04	2.12	2.16	1.89
Gzip mx1	3.05	3.05	3.19	3.06	3.12	3.24	3.14	3.14
Gzip mx5	2.73	2.76	2.9	2.75	2.83	2.94	2.82	2.88
Gzip mx9	2.73	2.76	2.9	2.75	2.83	2.94	2.82	2.88
LZMA mx1	2.84	2.83	2.97	2.82	2.88	3.01	2.9	2.91
LZMA mx5	2.11	1.78	2.28	2.05	2.05	2.24	2.17	1.9
LZMA mx9	2.11	1.78	2.27	2.04	2.04	2.12	2.16	1.89
PPMd mx1	2.07	2.02	2.24	2.03	2.16	2.26	2.15	2.24
PPMd mx5	1.85	1.73	2.03	1.77	1.86	2.03	1.92	1.91
PPMd mx9	**1.82**	**1.73**	**2**	**1.73**	**1.83**	**1.99**	**1.88**	**1.86**

results than MWCA + Huffman(*D1*, *D2*) because all of these files (except English) produce an *S3* stream (see Fig. 4).

It can be observed in Table 8 that MWCA generally gives better results than the mx1 versions of DEFLATE, Gzip and LZMA, and Gzip-m9 compresses more than MWCA and is faster in decompression. However, MWCA gives worse results than ETDC if we use Huffman encoding in last stage, and it could obtain better results than ETDC and SCDC with the exchange of compression time. Use of Huffman encoding in the last stage also causes it to lose the ability to conduct search operations on compressed files.

Use of MWCA + Huffman (*D1*, *D2*) does not have much effect on the compression ratio for text files larger than 200 MB, because the size of *D1*+*D2* becomes quite small compared with the total size (see Fig. 4).

Word-based compression algorithms give worse results on Gutenberg, which is multilingual. In multilingual files, the unique word count is higher than that of files that consist of one language. Therefore, the dictionaries for these files are larger than monolingual files. The assigned codes for the words become larger in ETDC and SCDC, and in our algorithm, words after 255+65536 are written in *S3*, which is not compressed if Huffman coding is not used.

The average compression and decompression speeds of MWCA and the other algorithms are given in terms of Mbps in Tables 9 and 10. In terms of compression speed, MWCA gives the best results on Group 1 and Group 3 but is slightly worse than ETDC in Group 2, which contains 7 non-English files. This result probably occurs because in non-English files, the unique word count is larger than in English files. This scenario causes our algorithm to use the *S3* stream more frequently than in English texts. Use of the *S3* stream means writing words into *S3* without encoding, and thus the time complexity of writing strings is higher than writing one byte into *S1* or two bytes into *S2*.

Our algorithm gives worse compression ratios than other general-purpose compression algorithms used with the mx5 and mx9 flags, but it is faster even if the mx1 flag is chosen for the given algorithms. Our test results show that using mx9 instead of mx5 decreases the compression speed dramatically, especially in Gzip and Bzip2, and does not have sufficient impact on the compression ratio. If we encode our files with Huffman coding, we lose speed in exchange for compression ratio.

Although all of these compression algorithms increase their decompression speed as the file size grows, MWCA is more successful than the others. Even though our decoding algorithm is slower than ETDC and SCDC, it catches up with these algorithms at Group 3. The reason for this obser-

Table 8 Compression ratios of Group 3 (bpc)

Compression algorithms	English 50 MB	English 100 MB	English 200 MB	Gutenberg	English 1024 MB	English 2108 MB
ETDC	2.82	2.79	2.73	2.82	2.68	2.67
ETDC + Huffman(vocabulary)	2.76	2.75	2.70	2.78	2.67	2.66
SCDC	2.76	2.74	2.68	2.77	2.63	2.62
MWCA	2.86	2.85	2.82	3.06	2.83	2.84
MWCA + Huffman(*D1*, *D2*)	2.82	2.84	2.81	3.06	2.83	2.84
MWCA + Huffman(*D1*, *D2*, *S3*)	2.79	2.78	2.75	2.9	2.76	2.76
MWCA + Huffman(*D1*, *D2*, *S1*, *S2*, *S3*, *BV*)	2.43	2.43	2.4	2.57	2.4	2.4
Bzip2 mx1	2.63	2.63	2.63	2.64	2.63	2.63
Bzip2 mx5	2.27	2.25	2.25	2.25	2.27	2.27
Bzip2 mx9	2.27	2.24	2.24	2.25	2.27	2.27
DEFLATE mx1	2.94	2.95	2.96	2.96	2.97	2.97
DEFLATE mx5	1.61	1.77	1.93	1.87	2.04	2.09
DEFLATE mx9	**1.59**	1.73	**1.71**	1.8	1.94	2
Gzip mx1	3.17	3.17	3.18	3.16	3.19	3.18
Gzip mx5	2.88	2.89	2.89	2.89	2.9	2.89
Gzip mx9	2.88	2.88	2.88	2.88	2.89	2.89
LZMA mx1	2.92	2.93	2.94	2.94	2.95	2.95
LZMA mx5	1.61	1.76	1.91	1.85	2.02	2.07
LZMA mx9	**1.59**	**1.73**	**1.71**	1.78	1.93	1.98
PPMd mx1	2.23	2.21	2.2	2.24	2.21	2.21
PPMd mx5	1.9	1.9	1.92	1.94	1.93	1.94
PPMd mx9	1.69	1.75	1.81	**1.77**	**1.88**	**1.89**

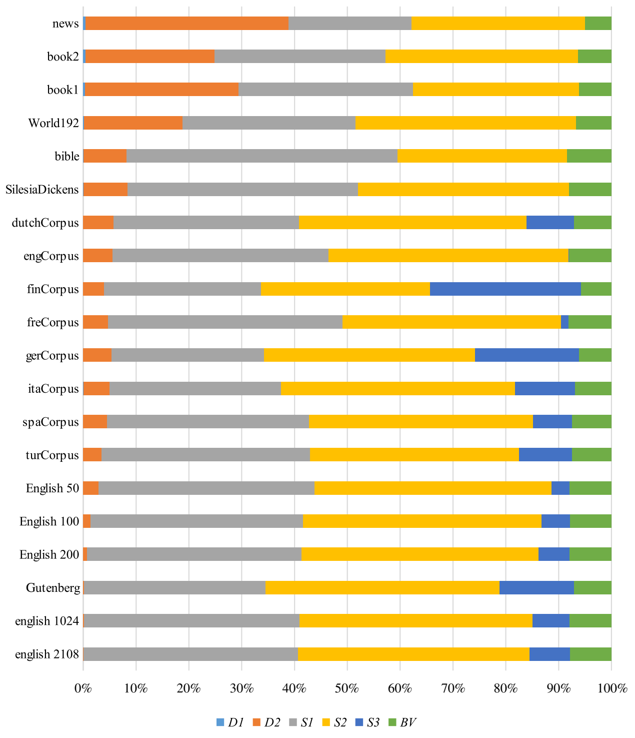

Fig. 4 Size ratios of dictionaries, streams and bit vector over compressed file (%)

vation is that the ratio of time spent in controlling flows is reduced with increasing file size.

The memory usage of our encoding and decoding algorithms for each file is given in Table 11.

Our encoding algorithm uses more than 20 MB of memory even if the file size is too small because MWCA uses a hash table to store the vocabulary of file. This means that we must predict the unique word count to open a hash table before reading and obtaining the words from the file. To predict the unique word count, we could use Heaps' law, but if the language of the text is changed, the prediction parameters should also be changed. Therefore, we define three different hash table sizes for three different file sizes. With increasing file size, the allocated memory is also increased because of the change in unique word count and word length in different languages. However, the (required memory/file size) ratio decreases as the file size increases. For example, the difference in required memory for encoding between English 100 MB and English 1024 MB is only 32.5 MB, although the file size difference is 924 MB. When the source file is too large, to reduce the memory requirement, MWCA uses buffers with a size of 100 MB in the encoding and decoding stage.

Table 9 Average compression speeds (Mbps)

Compression algorithms	Group 1 (< 10 MB)	Group 2 (= 30 MB)	Group 3 (> 50 MB)
ETDC	305.04	**336.64**	356.77
ETDC + Huffman(vocabulary)	144.47	219.19	320.34
SCDC	244.19	316.07	336.49
MWCA	**310.70**	327.97	**376.86**
MWCA + Huffman(*D1*, *D2*)	268.00	321.63	375.21
MWCA + Huffman(*D1*, *D2*, *S3*)	266.87	312.20	367.18
MWCA + Huffman(*D1*, *D2*, *S1*, *S2*, *S3*, *BV*)	190.54	242.32	281.09
Bzip2 mx1	228.66	290.94	289.35
Bzip2 mx5	105.01	185.08	196.03
Bzip2 mx9	13.28	19.01	22.45
DEFLATE mx1	230.30	456.56	470.49
DEFLATE mx5	30.01	13.49	34.41
DEFLATE mx9	27.61	12.80	20.91
Gzip mx1	214.85	213.13	219.22
Gzip mx5	49.57	47.75	48.10
Gzip mx9	8.18	7.54	7.66
LZMA mx1	105.17	107.94	105.77
LZMA mx5	30.06	13.45	20.18
LZMA mx9	29.25	12.76	10.43
PPMd mx1	113.16	130.84	115.62
PPMd mx5	82.73	77.56	72.67
PPMd mx9	38.78	27.49	27.61

Table 10 Average decompression speeds (Mbps)

Compression algorithms	Group 1 (< 10 MB)	Group 2 (= 30 MB)	Group 3 (> 50 MB)
ETDC	**1092.97**	949.91	1077.98
ETDC + Huffman(vocabulary)	862.95	**1062.82**	1158.82
SCDC	875.47	941.14	**1130.07**
MWCA	275.87	495.76	776.51
MWCA + Huffman(*D1*, *D2*)	189.38	454.78	761.75
MWCA + Huffman(*D1*, *D2*, *S3*)	189.38	416.55	705.22
MWCA + Huffman(*D1*, *D2*, *S1*, *S2*, *S3*, *BV*)	96.88	223.10	304.75
Bzip2 mx1	423.14	549.44	577.18
Bzip2 mx5	227.36	296.81	285.83
Bzip2 mx9	243.50	260.62	283.10
DEFLATE mx1	330.95	379.87	390.35
DEFLATE mx5	393.83	509.07	584.91
DEFLATE mx9	408.35	506.63	588.14
Gzip mx1	595.71	774.00	809.77
Gzip mx5	627.81	836.50	864.06
Gzip mx9	645.87	844.60	868.90
LZMA mx1	337.11	385.38	389.08
LZMA mx5	400.75	495.94	589.63
LZMA mx9	413.89	467.77	590.87
PPMd mx1	97.38	96.85	98.36
PPMd mx5	74.49	56.21	63.69
PPMd mx9	37.28	23.92	26.42

Table 11 Memory usage of encoding and decoding algorithms (MB)

Files	MWCA encoding	MWCA decoding
News	23.24	10.22
Book2	20.54	10.27
Book1	20.62	10.35
World192	25.45	10.92
Bible	26.85	11.26
Dickens	33.21	13.44
Dutch	63.02	20.18
English	61.86	19.7
Finnish	66.15	24.46
French	62.28	21.39
German	64.32	21.45
Italian	63.35	21.19
Spanish	63.12	21.6
Turkish	63.91	24.79
English 50 MB	101.31	27.86
English 100 MB	153.92	35.25
English 200 MB	157.84	40.21
GutenbergCorpus	177.85	79.54
English 1024 MB	186.4	84.6
English 2108 MB	262.84	146.4

In our experiments, we used the brute force, KMP, Horspool, and Berry Ravindran algorithms to obtain the search time results. We selected 5 random words each from the *D1*, *D2* and *S3* files to use in the search time comparison. The average search time results of these 15 words are given in Table 12 for each file.

In the search algorithm, the string-matching algorithms are used to find an occurrence of a word in the dictionaries or the *S3* stream. Use of string-matching algorithms to search the occurrence of a word in the *D1* and *D2* dictionaries does not have a high impact on search time because *D1* and *D2* are small in size (nearly 1 KB for *D1* and 500 KB for *D2*). However, if we use a string-matching algorithm on *S3*, a gain is obtained in average search time (this gain could be greater than 35% for large files).

The search algorithm implementation that can be obtained from the Dense Codes site that is used on the ETDC and SCDC compressed files returns the occurrence number of a given word in a given file. Therefore, it is appropriate to compare our algorithm with the search algorithm of ETDC and SCDC. Due to the advantage of using multiple streams and an untagged file structure, our search algorithm with BF is 6 times faster than ETDC and 6.2 times faster than SCDC on average. Use of Horspool instead of BF is 8.8 times faster than ETDC and 9.1 times faster than SCDC.

Table 12 Search times of search algorithm using Brute Force (BF), Knuth Morris Pratt (KMP), Horspool (HP) and Berry Ravindran (BR) algorithms (ms)

Files	MWCA				Dense codes	
	BF	KMP	HP	BR	ETDC	SCDC
News	**2**	4	**2**	2.1	2.3	2.8
Book2	**2**	2.2	**2**	**2**	2.1	2.6
Book1	2.1	2.1	2.4	**2**	3.1	3.3
World192	2.3	**2.1**	2.7	2.6	5.5	6.2
Bible	2.7	2.8	**2.6**	2.8	5	5.7
Dickens	3.7	4.8	**3.6**	3.7	13.4	15.3
Dutch	6.3	5.8	**5.1**	5.3	39.7	45.7
English	6.8	5.8	5.2	**5.1**	27.1	35.8
Finnish	7	7.5	6.7	**6.5**	78.2	91.2
French	7.5	6.8	**6**	6.1	31.9	34.5
German	6.2	6	**5.1**	5.3	52.3	62
Italian	6.5	6.3	**5.4**	**5.4**	44.7	54
Spanish	7	7.2	**5.4**	5.8	41.2	46.4
Turkish	8	7.2	**6.6**	6.7	56.1	65
English 50 MB	10.3	8.3	**7.6**	**7.6**	50.1	61.2
English 100 MB	19.6	13.2	12.2	**12.1**	95.9	123.5
English 200 MB	36.7	**23.9**	24.1	25.2	183.6	207
GutenbergCorpus	105.1	68.6	**67.7**	68.4	565.6	616.1
English 1024 MB	155.1	**100.7**	101	104.5	949.8	1077
English 2108 MB	268.2	**179.6**	179.9	189.1	1559.9	1590.2

6 Conclusions

Although the presented word-based text compression technique in this article achieves worse compression results than the general-purpose compression algorithms, it supplies a considerable advantage over these algorithms by supporting compressed search and minimizing the search space with a multi-stream structure. Such as ETDC and SCDC, MWCA is also suitable for use in text databases because of its word-based model. Because MWCA compresses data using block buffers, the size of the data to be compressed is not important. The block buffer size can be adjusted according to the amount of memory to be used and could be tailored to the desired configuration.

MWCA differs from ETDC with its multi-stream file structure. Although this structure reduces the speed of decompression, with the aid of compressed matching, the decompression is often not necessary. The limitation of our compressed search algorithm is that it supports only exact word matching, such as in ETDC. Although the compression ratio of MWCA is worse than that of ETDC, its multi-stream file structure allows it to find words faster without decompression.

Retrieving big text data stored in databases for search operations could result in a notably load on the network. Although compression algorithms such as ETDC reduce the size of the data, the need exists for retrieval of the whole file from the database. With the advantage of a multi-stream structure, MWCA can obtain only the necessary streams from the database, and thus the load on the network could be reduced. Additionally, the server load is decreased, and better performance could be obtained with reduced data exchange. We believe that our proposed compression algorithm is suitable for addressing the need to store big text data that are expected to be frequently searched.

Our test results show that we can achieve high compression ratios using Huffman coding after MWCA. However, in that case, direct search on the compressed files could not be used. It was shown in [18] that semi-static word-based byte-oriented compressors such as ETDC and SCDC can be used as a text transformation to boosts classical compression or indexing techniques. We believe that MWCA can also be used as a transform method because of its similar structure.

Acknowledgements The first author would like to thank to TÜBİTAK, for providing support with 2211-Graduate Scholarship Program.

References

1. Bell, T.C.; Cleary, J.G.; Witten, I.H.: Text compression. vol. 348. Englewood Cliffs: Prentice Hall, Inc., Upper Saddle River, NJ, USA (1990)
2. Brisaboa, N.R.; Farina, A.; Navarro, G.; Param, J.R.: Improving semistatic compression via pair-based coding. In: International Andrei Ershov Memorial Conference on Perspectives of System Informatics (pp. 124–134) (2006)
3. Brisaboa, N.R.; Farina, A.; Navarro, G.; Esteller, M.F.: (S, C)-dense coding: an optimized compression code for natural language text databases. In: International Symposium on String Processing and Information Retrieval (pp. 122–136) (2003)
4. Navarro, G.; Tarhio, J.: Boyer–Moore string matching over Ziv–Lempel compressed text. In: Annual Symposium on Combinatorial Pattern Matching (pp. 166–180) (2000)
5. Moffat, A.: Word-based text compression. Softw. Pract. Exp. **19**(2), 185–198 (1989)
6. Ziv, J.; Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theory **23**(3), 337–343 (1977)
7. Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)
8. Welch, T.A.: A technique for high-performance data compression. IEEE Comput. **17**(6), 8–19 (1984)
9. Deutsch, L.P.: DEFLATE compressed data format specification version 1.3 (1996)
10. Cleary, J.; Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. **32**(4), 396–402 (1984)
11. Bentley, J.L.; Sleator, D.D.; Tarjan, R.E.; Wei, V.K.: A locally adaptive data compression scheme. Commun. ACM **29**(4), 320–330 (1986)
12. Turpin, A.; Moffat, A.: Fast file search using text compression. Australian Comput. Sci. Commun. **19**, 18 (1997)
13. de Moura, E.; Navarro, G.; Ziviani, N.; Baeza-Yates, R.: Fast and flexible word searching on compressed text. ACM Trans. Inf. Syst. (TOIS) **18**(2), 113–139 (2000)
14. Brisaboa, N.R.; Faria, A.; Navarro, G.; Param, J.R.: Lightweight natural language text compression. Inf. Retr. **10**(1), 133 (2007)
15. Brisaboa, N.R.; Farina, A.; Navarro, G.; Parama, J.R.: New adaptive compressors for natural language text. Softw. Pract. Exp. **38**(13), 1429–1450 (2008)
16. Brisaboa, N.; Farina, A.; Navarro, G.; Param, J.: Dynamic lightweight text compression. ACM Trans. Inf. Syst. (TOIS) **28**(3), 10 (2010)
17. Carus, A.; Mesut, A.: A new compression algorithm for fast text search. Turk. J. Electr. Eng. Comput. Sci. **24**(5), 4355–4367 (2016)
18. Farina, A.; Navarro, G.; Param, J.R.: Word-based statistical compressors as natural language compression boosters. In: Data Compression Conference, 2008. DCC 2008 (pp. 162–171) (2008)
19. Farina, A.; Navarro, G.; Param, J.R.: Boosting text compression with word-based statistical encoding. Comput. J. bxr096 (2011)
20. Burrows, M.; Wheeler, D.J.: A Block-Sorting Lossless Data Compression Algorithm. Digital Systems Research Center Research Reports, Palo Alto, California, USA (1994)
21. Awan, F.S.; Mukherjee, A.: LIPT: a lossless text transform to improve compression. In: Information Technology: Coding and Computing, 2001. Proceedings. International Conference on (pp. 452–460) (2001)
22. Sun, W.; Zhang, N.; Mukherjee, A.: Dictionary-based fast transform for text compression. In: Information Technology: Coding and Computing [Computers and Communications], 2003. Proceedings. ITCC 2003. International Conference on (pp. 176–182) (2003)
23. Skibiski, P.; Grabowski, S.; Deorowicz, S.: Revisiting dictionary-based compression. Softw. Pract. Exp. **35**(15), 1455–1476 (2005)
24. Amir, A.; Benson, C.: Efficient two-dimensional compressed matching. In: Data Compression Conference, 1992. DCC'92. (pp. 279–288) (1992)
25. Wu, S.; Manber, U.: Fast Text Searching with Errors. University of Arizona, Department of Computer Science, Tucson (1991)

26. Boyer, R.S.; Moore, J.S.: A fast string searching algorithm. Commun. ACM **20**(10), 762–772 (1977)
27. Berry, T.; Ravindran, S.: A fast string matching algorithm and experimental results. In: Stringology (pp. 16–28) (1999)
28. Horspool, R.N.: Practical fast searching in strings. Softw. Pract. Exp. **10**(6), 501–506 (1980)
29. Knuth, D.E.; Morris Jr., J.H.; Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
30. Navarro, G.; Tarhio, J.: LZgrep: a Boyer–Moore string matching tool for Ziv–Lempel compressed text. Softw. Pract. Exp. **35**(12), 1107–1130 (2005)
31. Ziviani, N.; De Moura, E.S.; Navarro, G.; Baeza-Yates, R.: Compression: a key for next-generation text retrieval systems. Computer **33**(11), 37–44 (2000)
32. Heaps, H.S.: Information retrieval: Computational and theoretical aspects. Academic Press, Inc., Orlando, FL, USA (1978)
33. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. IRE **40**(9), 1098–1101 (1952)