Advanced engineering informatics. 2008, vol. 22, issue 3, p. 410-417. http://dx.doi.org/10.1016/j.aei.2008.05.001

COMPRESSION OF SMALL TEXT FILES

Jan Platoš, Václav Snášel
Department of Computer Science
VŠB – Technical University of Ostrava, Czech Republic
jan.platos.fei@vsb.cz, vaclav.snasel@vsb.cz

Eyas El-Qawasmeh
Computer Science Dept.
Jordan University of Science and Technology, Jordan
eyas@just.edu.jo

ABSTRACT
This paper suggests a novel compression scheme for small text files. The proposed scheme depends on Boolean minimization of binary data accompanied with the adoption of Burrows-Wheeler transformation (BWT) algorithm. Compression of small text files must fulfil special requirements since they have small context. The use of Boolean minimization and Burrows-Wheeler transformation generate better context information for compression with standard algorithms. We tested the suggested scheme on collections of small and medium-sized files. The testing results showed that proposed scheme improve the compression ratio over other existing methods.

KEY WORDS
Compression, Decompression, Small files, Quine-McCluskey.

1. Introduction

Data compression aims to remove redundant data in order to reduce the size of a data file [1], [2]. For example, an ASCII file is compressed into a new file, which contains the same information, but with smaller size. The compression of a file into half of its original size increases the free memory that is available for use [3]. The same idea applies to transmission of messages through the network with limited bandwidth channels.

Currently, the volume of all data types is increasing continuously. This holds for all data types including textual data. Many lossless compression techniques and algorithms for normal and large texts exist. These techniques can be classified into three main categories. They are: substitution, statistical, and dictionary based data compression techniques [4]. The substitution category replaces a certain repetition of characters by a smaller one. Run Length Encoding (RLE) is one of these techniques that take advantage of repetitive characters. The second category involves generation of the shortest average code length based on an estimated probability of characters [5]. An example of this category is Huffman coding [[]. In Huffman coding, the most common symbols in the file assigned the shortest binary codes, and the least common assigned the longest codes. The last category is dictionary based data compression schemes such as Lempel-Ziv-Welch (LZW). This category involves the substitution of sub-string of text by indices or pointer code, relative to a dictionary of the sub-strings [14], 3, [20].

Most previous mentioned compression algorithms need sufficient context information. Context information in large text files (larger than 5MB) is very big and therefore it is sufficient even we take a word as a base unit of compression. This approach of compression is called word-based [12, [9, 19] (see Figure 1a). Text files of medium size (200KB–5MB) has context information smaller than large text files, therefore, it is insufficient to take the word as a base unit, however, we can take the smaller grammar part – syllables [11] (see Figure 1b). Context information in small files (100KB – 200KB) is difficult to obtain. In small files, the context information is sufficient context information only when we process them by characters (see Figure 1c). In [23], authors have compared the single file parsing methods used on input text files of size 1KB–5MB by means of Burrows-Wheeler Transform for different languages (English, Czech, and German). They considered these element types: letters, syllables, words, 3-grams and 5-grams. Comparing the letter-based, syllable-based and word-based compression, they found out that the character-based compression is the most suitable for small files (up to 200KB) and the syllable-based compression is the best-fit for files of size 200KB–5MB. The compression which uses natural text units like words or syllables is 10–30 % better than the compression with 5-grams and 3-grams.

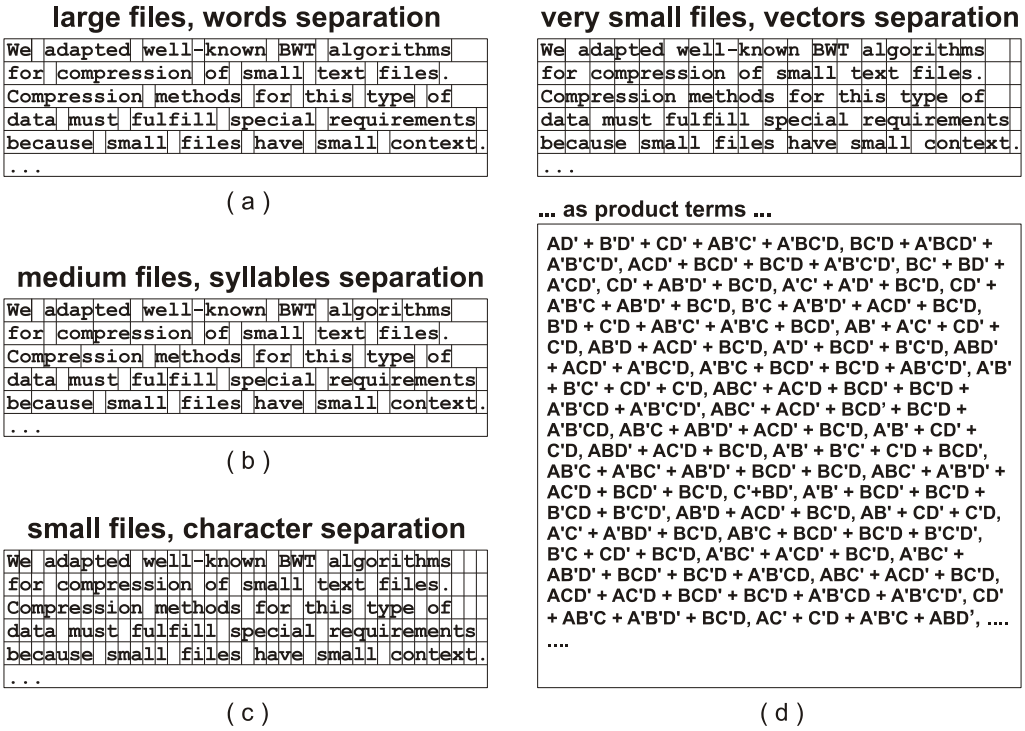

Figure 1: Context information in text files of various sizes

Very small text files (up to 160KB), like SMS, chat messages, messages in instant-messaging networks and emails, have different characters. Size of such messages is very small and context information is minimal. However, for different variations of these messages, we can use any of the previous mentioned algorithms. Compression of large number small files is difficult task for any program; therefore, a so-called solid mode is used. The solid mode means that all files are processed as one big file and a model for statistical-based compression or a dictionary for dictionary-based compression is common for all these files. In this case, a large context for compression is used. If we do not have all data, we must compress files one by one. Therefore, we need a good compression algorithm, which only requires a small context or we need an algorithm that transforms input data into another form (see Figure 1d). An alternative approach is to use Boolean minimization for data compression, which was presented by El-Qawasmeh and Kattan in 2006 [16]. In this technique, the compression process splits input data stream into 16-bit. After that, Quine-McCluskey approach is used to handle this input in order to find the minimized expression. The minimized expressions of the input data stream are stored in a file. After minimization, El-Qawasmeh uses static Huffman encoding for obtaining redundancy-loss data. The obtained Huffman code is used to convert the original file into a compressed one. On the decompression side, the Huffman tree is used to retrieve the original file.

This paper suggests a novel compression scheme for small text files. The proposed scheme is designed for very small text files and it uses Burrows-Wheeler transformation [25] besides Boolean minimization at the same time.

The organization of this paper will be as follows: Section 2 presents a review of the Quine-McCluskey algorithm. Section 3 explains the improvements and enhancements suggested. Section 4 describes the compression algorithm. Section 5 describes the implementation. Section 6 shows the performance results. Section 7 is a discussion, and Section 8 contains the conclusion of this paper.

2. Quine-McCluskey algorithm

The Quine-McCluskey algorithm is a technique for presenting minimization of Boolean functions [21]. It seeks to collect “product terms” by looking for entries that differ only in one bit. The Quine-McCluskey algorithm is capable of minimizing logical relationships for any number of inputs. It starts with the truth table and extracts all the “product terms” (all input combinations that produce a true output). After that, it groups all the “product terms” by the number of "ones" they contain. Then it combines the “product terms” from adjacent groups [17].

The proposed technique uses the Quine-McCluskey algorithm. It considers every 16-bit block from the input file, as one chunk called a vector, and finds the minimized Boolean function for it. Each 16-bit vector corresponds to 4-input variables truth table. The number of different functions that can be generated from 4 variables is 2^(2^4) = 65536 where the first 2 represents either 0 or 1 and 2^4 is equal to the 16-bit vector. The total number of all distinct "product terms" from these combinations is 82 (see Table 1). The minimized "product term" might contain 1, 2, 3 or 4 variables. For example, the number of different "product terms" that contain exactly one variable is 10 out of 82 "product terms". For the input variables (A, B, C, D), the set S_1 that contains exactly one "product term" will be S_1= {A, A`, B, B`, C, C`, D, D`, 0, 1}. The distribution of these 82 "product terms" and the number of variables in each "product term" is as follows:

Number of "product terms"	1-variable	2-variables	3-variables	4-variables	Total
	10	24	32	16	82

Table 1 contains all the possibilities of the minimized "product terms" with the corresponding distinct variables in each "product term". Table 1 contains all possible "product terms" from a 16-bit vector after applying the Quine-McCluskey algorithm. (Note that a set of "product terms" will constitute a "sum of products"). The proposed compression algorithm creates a frequency lookup table of size 83 elements where 82 are distinct elements that are listed in Table 1 plus one extra element for the comma (serves as a separator between adjacent vectors). Each entry in table 1 has a counter that is initialized to zero at the beginning of the compression phase. Later, any detected "product term" during the compression process updates the value of its corresponding counter in the frequency lookup table.

1-variable	2- variables			3- variables				4- variables	
A	AB	A`D	CD	ABC	AB`D	A`CD	B`C`D	ABCD	A`BC`D
B	AB`	A`D`	CD`	ABC`	AB`D`	A`CD`	B`C`D`	ABCD`	A`BC`D`
C	A`B	BC	C`D	AB`C	A`BD	A`C`D		ABC`D	A`B`CD
D	A`B`	BC`	C`D`	AB`C`	A`BD`	A`C`D`		ABC`D`	A`B`CD`
A`	AC	B`C		A`BC	A`B`D	BCD		AB`CD	A`B`C`D
B`	AC`	B`C`		A`BC`	A`B`D`	BCD`		AB`CD`	A`B`C`D`
C`	A`C	BD		A`B`C	ACD	BC`D		AB`C`D	
D`	A`C`	BD`		A`B`C`	ACD`	BC`D`		AB`C`D`	
0	AD	B`D		ABD	AC`D	B`CD		A`BCD	
1	AD`	B`D`		ABD`	AC`D`	B`CD`		A`BCD`	

Table 1: All possible "product terms" of 1, 2, 3, and 4 variables.

3. Suggested improvements and enhancements

In this section, we will describe the basic ideas behind our suggested scheme. This includes the improvements and enhancements that we will suggest. Our experiments were started by testing the compression algorithm compressed which uses Quine-McCluskey that was suggested by El-Qawasmeh and others [16]. We have tested the mentioned algorithm on a randomly generated data, and the test results showed that the best compression ratio was achieved on files with a very small or very high probability of ones. Therefore, if we transform vectors from compressed file into more suitable vectors and their "product-terms", then we can achieve a very good compression ratio. This is possible for text files since they have very low, distinct vectors, typically less than 5000 distinct vectors for all text files. For example, the *bible.txt* file from Canterbury Compression Corpus [[]3] has 1121 distinct vectors and *latimes.txt* from TREC [6, 7, 8] has 4984 distinct vectors. There are several ways to do this.

The first improvement is based on an idea which state that vectors with less count of "ones" have shorter length of Boolean minimization. This idea is based on the principle that when there is less number of ones, then we will get less grouping size and hence shortest Boolean minimization. Realization of this improvement is simple; all possible vectors are sorted based on the number of "ones" in the first step. A list of used vectors is created in the second step. The mapping table is a junction of a list of sorted vectors and a list of used vectors.

The second improvement is based on similar idea of sorting as previous one. However, the parameter of sorting will be different. In this case we will sort all vectors by length (num of "product terms") of their Boolean minimization. This sorting and mapping lead to shortest possible representation of original data after Boolean minimization. The advantage of sorting on

both cases is the preservation of context information from original data and their improvement by sequences of "product terms". Realization of this improvement is similar to the previous one, but vectors are sorted by the length of their Boolean minimization instead of frequency of "ones". Disadvantage of this improvement is the necessity of calculation of Boolean minimization for all possible product terms, but this can be done only once, because we can store that minimizations into a look-up table and for compression we can use only this table. In addition, count of all possible vectors is quite small.

In the compression algorithm that were suggested by El-Qawasmeh and others, they use Quine-McCluskey algorithm combined with the static Huffman encoding in order to achieve the compression for the product terms. In fact, the use of static Huffman has many disadvantages. The most obvious disadvantage is the requirement of two passes over the file. Another disadvantage is that, in most cases, it has worse compression ratio than others, e. g. dictionary algorithms.

Text files that contain ASCII characters can consists of 256 distinct characters. By using Quine-McCluskey, we reduce the alphabet of the file from 256 symbols (more precisely from 65,536, because we use 16-bit vectors) into 82 product terms and one extra symbol for separating between vectors after Boolean minimization. In this case, we can get better context information in transformed data than we have in original, non-minimized data. Currently, there are many compression methods that require good context in order to achieve a good compression ratio. One of them is the BWT (Burrows-Wheeler) compression algorithm [2[2].

The Burrows-Wheeler algorithm is based on the frequent occurrence of symbol pairs in similar context and it uses this feature for obtaining long strings of the same characters. These strings can be transformed into another form with MTF (Move To Front) transformation, which is dominated by zeros. With BWT and MTF, algorithm forms like RLE compression, Huffman encoding and others, can be applied.

BWT can achieve good compression ration provided that there is a sufficient context which is formed by frequent occurrences of symbols with same or similar prefixes. Maximizing the context leads to the better compression ratio. This can be achieved with sorting product terms in some order, e.g. by their length and alphabetical order. The term "0" is then assigned 0, the term "1" is assigned 1, the term "A" is assigned 2, and so on until the term " A'B'C'D' " which assigned 82 and a comma which assigned 83.

Another improvement can be achieved with separator elimination. Since product terms are sorted by their number, i.e. lowest to highest, the passing between product terms of adjacent vectors can be detected by decreasing the term number. This situation is displayed in Figure 2. This principle works in most cases but not in all. The case in which we can use separator elimination is depicted in Figure 2 between vector 3 and 4. In this case all "product terms" in Boolean minimization of vector 4 has larger number than any "product term" in Boolean minimization of vector 3. And therefore we must store separator into compressed file.

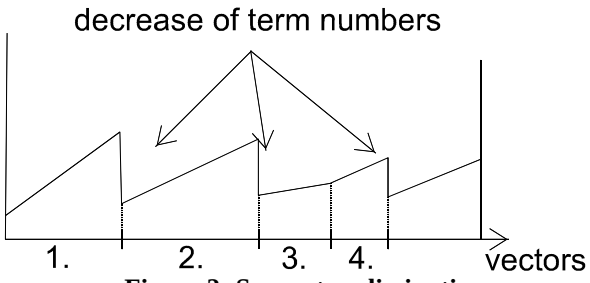

Figure 2: Separator elimination

The most recent tested improvement was not compression by product terms, but compression by smaller parts. Compression by logical variables (A, A', B … D') is one. In this case, the separator between product terms, '+', must be compressed too. A second possibility is compression by separate characters A, B, C, D, +, ' and comma. The volume of original data significantly increases in both cases, because the we increase the number of symbols,, but the alphabet size decreases, also significantly.

4. Compression algorithm

The whole compression algorithm is depicted in Figure 3 and can be described as follows :

1. Create output file/init network communication
2. Process each input file as follows
   2.1. Initialize local mapping function – this function map input vector to the vectors with better characteristic for compression using Boolean minimization.

2.2. Update mapping function – this is performed only if mapping need to be updated by vectors occurrence
  2.2.1. Use all vectors in source file to update mapping function
  2.2.2. Make model of mapping function
  2.2.3. Store used mapping model to the output file
2.3. For each vector in input file process:
  2.3.1. Substitute source vector by global mapping function
  2.3.2. Substitute globally mapped vector by local mapping function
  2.3.3. Encode mapped vector into output file
3. Close output file

Following is Figure 3 which shows the whole process (The compression and decompression at the same time).

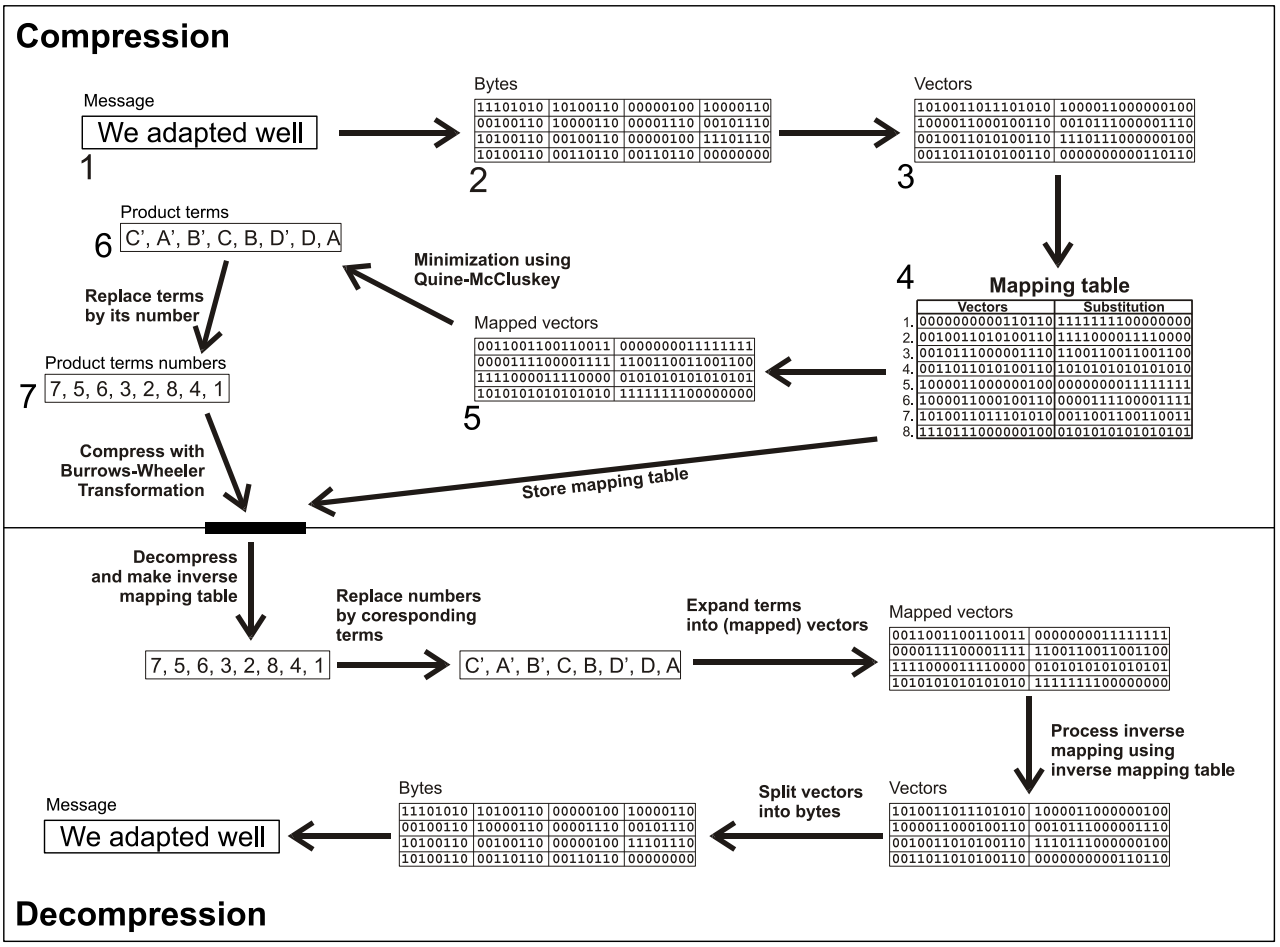

Figure 3: Compression and decompression algorithm

Figure 3 depicts compression and decompression algorithm suggested for very small text file compression. Input message is read by bytes and each two bytes are concatenated to form a 16-bit vector. Sequence of vectors are transformed into another vectors by using the mapping table, where its creation is described in Section 3 and then stored into a file with compressed message. After transformation, Quine-McCluskey algorithm is used to find minimization for each vector. Resulting "product terms" for each vector are converted into numbers and sorted. These numbers are then processed by Burrows-Wheeler compressor.

Decompression process is the inversion of compression process. Numbers of term are decompressed by Burrows-Wheeler compressor, number are converted into "product terms", "product terms" are used to create 16-bit vectors which are mapped using inverted mapping table based on mapping table stored in file. Resulting vectors are divided into bytes which are stored into decompressed file.

5. Implementation

We have implemented all the previous improvements and integrate them with the compression algorithm that uses the Boolean minimization. Implementation of Boolean minimization using the Quine-McCluskey method was realized by one class which transforms the 16-bit vector from input into a set of product terms, a number and a character format. The Quine-McCluskey method itself is described well and the improvements were only focused on the last step of this algorithm, in which a set of terms for representing original functions are selected. This modification select terms so that the number of terms was as small as possible and its order was this term by their number. A demand on small term numbers ensures the use of small alphabet and larger frequency for smaller terms.

For mapping function, the transformation of vectors from text files into vectors with less number of one's or few product terms after minimization was realized with a class called Mapper. This class transforms original vectors into other vectors which have larger context. Two versions of this class were used. First, one transforms vectors into others which are ordered by a certain amount of ones. The second one transforms vectors into others which are ordered by theirs amount of product terms. A mapping table for describing which vectors were mapped from original files into new "ones" must be stored in both versions. Many methods were tested; those which store numbers used in the first vector and with some distance from the proceeding vector, were most effective. These numbers are stored by Fibonacci encoding for achieving maximum effectiveness.

6. Performance Results

The suggested method was compared with four different methods. The first compression method is Semi-static Huffman encoding [[] which was used in [16]. This algorithm is a well known compression algorithm and can be found in many references. The second compression method is Adaptive Huffman encoding (AHuff) [15], which is also well known, and does not need to store a Huffman tree into its file. This compression method was used for the purpose of comparing as a character compressor and as a part of a compressor based on Burrows-Wheeler transformation. The third method that we used is LZW, which is described in [18]. This method was used as a normal compressing algorithm with a character-based approach. The fourth algorithm which we compared with is based on a BWT compression algorithm. In this algorithm, Burrows-Wheeler transformation is applied to the data block. Then MTF transformation is used. Data from MTF is compressed using the RLE algorithm which uses Fibonacci encoding for counts and Adaptive Huffman encoding for symbols.

Experimental results showed that the Maximal compression efficiency was achieved by applying another transformation which was used as the first step in any used algorithm. This transformation maps one vector into another, but this is not the same as a transformation, which is described in Section 4. In this case, vectors were mapped by their frequency. This transformation is realized by a priority queue. The range of used vectors is decreased by this transformation, which simplifies mapping for each file. In contrast to the previous transformation, it does not require storing of the mapping table because it is used for all files in a stream and is adaptively updated during compression.

The Compression efficiency was tested in many tests over several files. In the first step, the algorithm described in [16] was compared to the suggested algorithm which is based on Boolean minimization and BWT. In the second step, the described algorithm was compared to another compression algorithm.

The first group of the files that we used for testing is *sms.txt*, which is a collection of SMS which is available at *http://www.mla.iitkgp.ernet.in/~monojit/sms.html*. This collection was collected from a public SMS stored server and was prepared for processing by Monojit Choudhury and his team. The collection contains 854 messages in original and human translated form. The original form was used for compression. The second file is *bible.txt* from Canterbury compression corpus [[]3]. This file was processed by lines. Each line contains one verse form the English bible. These files will be designed as "non-enron files".

The second group of files that we used for testing was parts of Enron corpus. Enron corpus consists of 3 parts:

- *enron_nh.txt* contains the 20,000 emails without headers.
- *enron_sh.txt txt* contains the 20,000 emails smaller than 1KB without headers and
- *enron.ssh.txt* contains 20,000 emails smaller than 512B.

These files will be designed as "enron files". The following part of this section describes comparison results for the algorithm described in [16] and our proposed algorithm based on BWT and Boolean minimization at the same time. The base algorithm is designed as *MinHuff*, algorithms with Boolean minimization and BWT over product terms is designed as *MinBwt*, same algorithm with separator elimination as *MinBwtS*, algorithms with minimization and BWT over logical variables as *MinBwtL*

and algorithms with minimization and BWT over character representation as *MibBwtC*. All variants of MinBwt algorithms had block sizes set at 1MB.

Results for non-enron files are shown in Table 2 (CS means compression size and CR means compression ratio). Both of them contain very short messages. The compression ratio of the *MinHuff* method is greater than 100%, which is caused by storing of Huffman trees into file. All variants of *MinBwt* algorithms are better than *MinHuff* algorithms. Base *MinBwt* versions achieved the best compression ratio; results achieved by *MinBwtS* variant are a little worse. Variants with compression on logical variables or characters are still better than *MinHuff*, but quite worse than *MinBwt*.

	sms.txt [93 288 B]		**bible.txt [4 017 008 B]**	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**
MinHuff	108 696	116,52	4 447 367	110,71
MinBwt	73 941	79,26	2 946 782	73,36
MinBwtL	96 786	103,75	3 891 008	96,86
MinBwtC	88 934	95,33	3 590 441	89,38
MinBwtS	81 881	87,77	3 260 340	81,16

Table 2: Comparison between base algorithm and algorithms with BWT

Results for Enron files are depicted in Table 3. Order by achieved compression ratio is same as in Table 2.

	enron_nh.txt [36 891 613 B]		**enron_sh.txt [8 095 564 B]**		**enron_ssh.txt [4 533 827 B]**	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**	**CS [bytes]**	**CR [%]**
MinHuff	29 548 821	80,10	7 909 173	97,70	5 018 207	110,68
MinBwt	21 055 382	57,07	6 345 648	78,38	3 706 019	81,74
MinBwtL	26 657 872	72,26	7 851 240	96,98	4 667 562	102,95
MinBwtC	25 353 586	68,72	7 470 631	92,28	4 416 945	97,42
MinBwtS	21 391 560	57,98	6 511 943	80,44	3 912 122	86,29

Table 3: Comparison between base algorithm and algorithm based on BWT -Enron files

7. Discussion

Comparison with standard compression algorithm is described in this section. *MinBwt* variant was chosen as the best compression method based on Boolean minimization and BWT. This algorithm was compared with standard, character based, algorithms. Three algorithms were tested. *BWT* is an algorithm based on BWT with MTF transformation, RLE compression and Fibonacci and Huffman encoding. Block size was set on 1MB for testing. *LZW* is a traditional compression algorithm described previously. The size of the dictionary was set at 1MB again. The most recently used algorithm was Adaptive Huffman encoding, which is designed as AHuff. *MinHuff* algorithm was only included in the results for the purpose of comparing.

Results for non Enron files are depicted in Table 4. The MinBwt algorithm was the best for both files. Other algorithms are significantly worse. This is probably caused by very short messages which do not have sufficient context. Please note the CS in table 4 means the size of the compressed file and CR is the compression ratio.

	sms.txt [93 288 B]		**bible.txt [4 017 008 B]**	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**
BWT	92 502	99,16	3 701 212	92,14
LZW	83 624	89,64	3 405 503	84,78
AHuff	78 110	83,73	3 291 873	81,95
MinHuff	108 696	116,52	4 447 367	110,71
MinBwt	73 941	79,26	2 946 782	73,36

Table 4: Comparison with standard algorithms

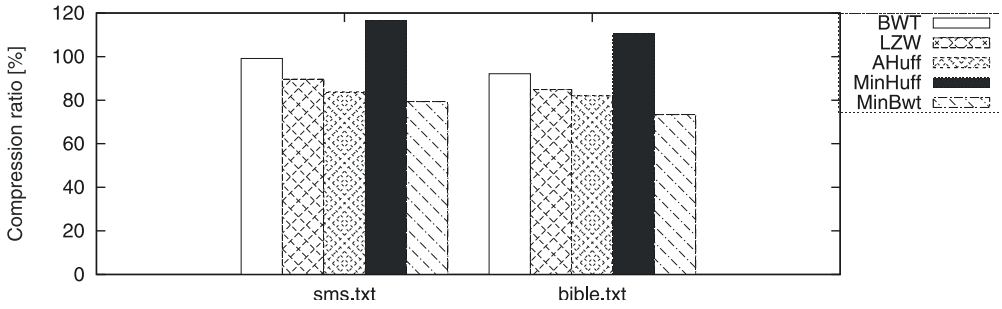

Figure 4: Comparison with standard algorithms

Results for Enron files are depicted in Table 5 and Figure 5. These files contain messages with variable lengths. The average length of messages in *enron_nh.txt* is 1845 B, in enron_sh.txt is 405 B and in enron_ssh.txt is 227 B. The previous two files have message lengths mostly shorter than 400 B. Messages in Enron files are so long that context based algorithms are effective on these files.

The best result for e*nron_nh.txt* file was achieved with a *BWT* algorithm. *MinBwt* is in second place, just before *LZW* algorithm. *AHuff* and *MinHuff* algorithms have the worst results.
BWT algorithm was the best also for *enron_sh.txt* file. *AHuff* algorithm was a little worse. *MinBwt* and *LZW* algorithms were worse by 3 % of compression ratio. *MinHuff* was the worst again.
Enron_ssh.txt file contains shorter messages than other Enron files. Compression algorithms that need large context are worse than MinBwt, because context in messages from these files is insufficient. The best result, however, has been achieved by AHuff algorithm. MinBwt was worse by 1 %. LZW algorithm came in third place with BWT coming in fourth.

	enron_nh.txt [36 891 613 B]		**enron_sh.txt [8 095 564 B]**		**enron_ssh.txt [4 533 827 B]**	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**	**CS [bytes]**	**CR [%]**
BWT	17 469 403	47,35	6 050 234	74,74	3 876 253	85,50
LZW	21 321 651	57,80	6 298 213	77,80	3 840 240	84,70
AHuff	24 570 316	66,60	6 099 495	75,34	3 647 439	80,45
MinHuff	29 548 821	80,10	7 909 173	97,70	5 018 207	110,68
MinBwt	21 055 382	57,07	6 345 648	78,38	3 706 019	81,74

Table 5: Comparison with standard algorithms - Enron files

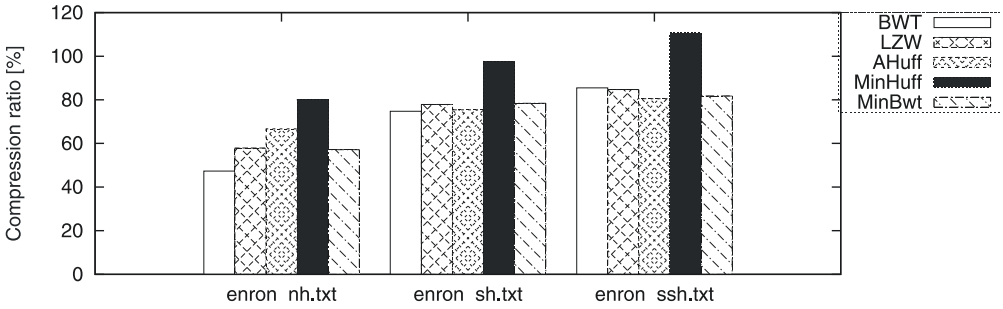

Figure 5: Comparison with standard algorithms – Enron files

As can be seen from results above, our algorithm is for very small messages better than other algorithms. Therefore, we compare supposed algorithm with programs which are industry standards in field of data compression and with programs which are on highest places in compression benchmark[1].
Very well known programs GZIP and BZIP were selected as industry standard representatives. The GZIP program is based on LZ77 [24] and use same method as very well known program WinZIP. The BZIP2 program is based on Burrows-Wheeler transformation [22]. The data compression benchmark mentioned above contain more than 150 compression programs and three of them which were on highest places in English text compression section were selected. The Paq8o9 program was

[1] http://www.maximumcompression.com/index.html

developed by Matt Mahoney and uses Context mixing [25]. The *Durilca*, by Dmitry Shkarin, and *Slim*, by Serge Voskoboynikov, programs uses modified PPM method with pre-processing. For comparison purpose was compared sums of compressed files sizes. If program store the filename in compressed data then the length of filename was subtracted from total length of compressed file.

The results for non-enron files are depicted in Table 6 and Figure 6. As can be seen, the best results for very short messages were achieved by *MinBwt* algorithm. The second was *Durilca* and third was *paq8o9*. The worst results were achieved by program Slim, because it store very large overhead. The programs GZIP and BZIP2 are not so effective for so small files as they are for larger files.

	sms.txt [93 288 B]		bible.txt [4 017 008 B]	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**
MinBwt	73 941	79,26	2 946 782	73,36
GZIP	93 564	100,30	3 799 574	94,59
BZIP2	95 312	102,17	3 867 578	96,28
paq8o9	92 599	99,26	3 651 313	90,90
Durilca 0.5	89 826	96,29	3 411 830	84,93
Slim 0.23	179 218	192,11	6 717 498	167,23

Table 6: Comparison with standard program

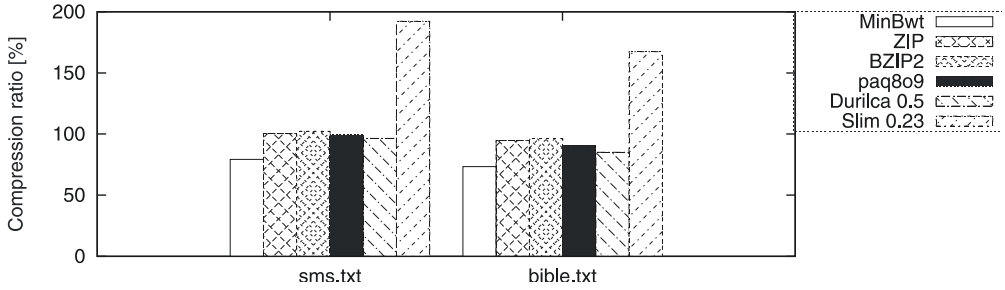

Figure 6: Comparison with standard program

The results for enron files are depicted in Table 7 and Figure 7. Enron files contain longer files than *sms.txt* or *bible.txt* and therefore results achieved by standard programs are much better. The best result was achieved by *Durilca* for all three files. The second was *Slim* and third was *paq8o9* for first two files with larger files. For file *enron_ssh.txt* with shortest files was on second place program *paq8o9* and on third place was *MinBwt* algorithm.

	enron_nh.txt [36 891 613 B]		enron_sh.txt [8 095 564 B]		enron_ssh.txt [4 533 827 B]	
	CS [bytes]	**CR [%]**	**CS [bytes]**	**CR [%]**	**CS [bytes]**	**CR [%]**
MinBwt	21 055 382	57,07	6 345 648	78,38	3 706 019	81,74
GZIP	16 539 617	44,83	5 637 415	69,64	3 796 825	83,74
BZIP2	16 769 534	45,46	6 063 197	74,90	4 072 790	89,83
paq8o9	14 476 007	39,24	5 373 854	66,38	3 636 618	80,21
Durilca 0.5	11 663 530	31,62	5 020 348	62,01	3 489 462	76,97
Slim 0.23	16 105 745	43,66	7 316 159	90,37	5 629 814	124,17

Table 7: Comparison win standard programs – enron files

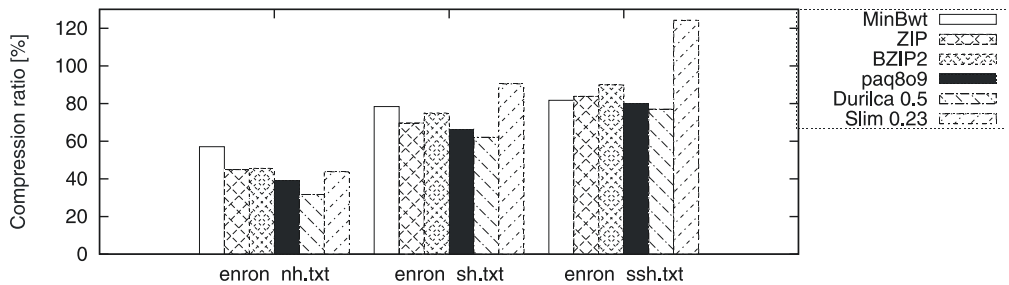

Figure 7: Comparison win standard programs – enron files

8. Conclusion

This paper describes a novel compression scheme for very small text files. This scheme may be useful for many applications which need to store or transfer many small files, e.g. instant messaging, SMS, chat communication. The described scheme is more effective for small text files than standard compression algorithms and commonly used compression programs. The advantage of this algorithm is usability in any other system, because it works independently to other systems. Therefore, it can be used at the lowest level of any application, which needs to store or transfer data of this type.

Future work may focus on modification of minimizing algorithms for 32-bit length vectors accompanied with applying other mapping transformation and using other compression algorithms in the second phase of compression.

References

[1] G. J. Mathews. Selecting a general-purpose data compression algorithm. In *Proceedings of the Science Information Management and Data Compression Workshop*, NASA Publication 3315, pp. 55–64, 1995.

[2] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37– 44, 2000.

[3] J. Mandal. *An Approach Towards Development of Efficient Data Compression Algorithms and Correction Techniques*. PhD thesis, Jadavpur University, 2000.

[4] M. K. Pandya. Data compression: Efficiency of varied compression techniques. Technical report, University of Brunel, UK, 2000.

[5] J. S. Vitter. Dynamic Huffman encoding. *Journal of the ACM*, 15 (2):158–167, 1989.

[6] D.K. Harman, editor. *The First Retrieval Conference (TREC-1)*. National Inst. of Standards and Technology, Gaithersburg, USA, 1993.

[7] D.K. Harman, editor. *The Second Retrieval Conference (TREC-2)*. National Inst. of Standards and Technology, Gaithersburg, USA, 1994.

[8] D.K. Harman, editor. *The Fourth Retrieval Conference (TREC-4)*. National Inst. of Standards and Technology, Gaithersburg, USA, 1997.

[9] R. N. Horspool and G. V. Cormack. Construction word-based text compression algorithms. In *Proc. 2nd IEEE Data Compression Conference*. Snowbird, 1992.

[10] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of IRE*, 40(9):1098–1101, 1952.

[11] J. Lansky and M. Zemlicka. Compression of small text files using syllables. In *Data Compression Conference*, volume DCC 2006, page 458, 2006.

[12] J. Dvorský, J. Pokorný, and V. Snášel. Word-based compression methods and indexing for text retrieval systems. In *ADBIS'99*, pages 75–84. Springer Verlag, 1999.

[13] R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In J. A. Storer and M. Cohn, editors, *Proc. 1997 IEEE Data Compression Conference*, pp. 201–210. IEEE Computer Society Press, Los Alamitos, California, March 1997.

[14] M. Nelson. Lzw data compression. *Dr Dobb's Journal*, 14(10):62–75, 1989.

[15] M. Nelson and J-L. Gailly. *The Data Compression Book*. M&T Books, 1995.

[16] E. El-Qawasmeh and A. Kattan. Development and Investigation of a Novel Compression Technique Using Boolean Minimization. *Neural Network World, Czech Republic*, Vol. 4, No.6, pp. 313-326, 2006.

[17] M. D. Ercegovac, T. Lang, and J. Moreno. *Introduction to Digital Systems*. John Wiley and Sons Inc, 1999.

[18] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[19] I.H. Witten, Alistair Moffat, and T.C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994.

[20] E.-H. Yang and J. C. Kieffer. Simple universal lossy data compression schemes derived from the lempel-ziv algorithm. *IEEE Trans. Inform. Theory*, IT-42(1):239–245, 1996.

[21] D. Zissos. *Logic Design Algorithms*. Harwell/Oxford University Press, 1972.

[22] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital Systems Research Center Research Report 124, 1994.

[23] J. Lansky; K. Chernik, Z. Vlickova. Comparison of Text Models for BWT, Data Compression Conference, 2007. DCC'07, Date: March 2007, Pages: 389 – 389

[24] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, Volume 23, Number 3, 337-343, 1977

[25] M. Mahoney, Adaptive Weighing of Context Models for Lossless Data Compression, Florida Tech. Technical Report CS-2005-16, 2005.