Byte Pair Transformation using Zero-Frequency Bytes with Varying Number of Passes

Jyotika Doshi
GLS Inst.of Computer Technology
Opp. Law Garden, Ellisbridge
Ahmedabad-380006, India

Savita Gandhi
Dept. of Computer Science;Guj. Uni.
Navrangpura
Ahmedabad-380009, India

ABSTRACT

Byte pair encoding (BPE) algorithm was suggested by P. Gage is to achieve data compression. It encodes all instances of most frequent byte-pair using zero-frequency byte in the source data. This process is repeated for maximum m possible number of passes until no further compression is possible, either because there are no more frequently occurring byte pairs or there are no more unused zero-frequency bytes to represent pairs. It writes out substitution information before the encoded data in each pass. This algorithm is very time consuming as it requires to determine most frequent byte-pair in each pass before starting substitution. We have proposed k-pass byte-pair transformation algorithm where k may be very very small as compared to maximum possible passes m. Our aim is to minimize the compression time and achieve equvivalent compression rate. Proposed algorithm transforms half of the possible most-frequent byte pairs in each pass except the last. In the last pass, it transforms all remaining possible byte-pairs. This reduced number of passes save the time taken in computing frequency of byte-pairs in maximum m passes. Experimental results have shown that proposed algorithm had taken 3.213, 9.794, 13.324, 16.323, 22.388 seconds with 1, 2, 3, 4 and 6 passes respectively as compared to 295.642 seconds of m-passes. Compression rate achieved due to transformation is 14.72%, 20.12%, 21.89%, 22.67% and 22.96% with 1, 2, 3, 4 and 6 passes respectively as compared to 25.55% using maximum m-passes. As the number of passes increases, compression is better with increased execution time. Our aim of achieving speed is achieved with little loss in compression rate.

General Terms

Data Compression, Algorithms

Keywords

Data Compression, better compression rate, byte-pair data transformation, substitution using zero-frequency byte symbols

1. INTRODUCTION

Data transformation transforms data from one form to another in order to reduce data size or to encrypt the data. This paper describes a vaariation of data transformation applied on byte pairs; a pair of adjacent bytes. Frequent byte pairs are encoded using zero-frequency bytes (bytes not used the data source).

Byte Pair Encoding (BPE) algorithm proposed by P. Gage [1] compresses data by finding the most frequent pair of adjacent bytes in the data and replacing all instances of this pair with a byte that was not in the original data. The algorithm repeats this process until no further compression is possible, either because there are no more frequently occurring pairs or there are no more unused bytes to represent pairs [9]. Let us consider m as the number of passes or number of times the process is repeated. A variation of BPE is block-wise transformation that is applied on small data blocks in stead of entire file. Here, it is referred to as m-pass BPT-Z (Byte-Pair Transformation using Zero-frequency bytes).

In this paper, the proposed variation of BPE is k-pass BPT-Z where it encodes more than one most-frequent byte-pairs with zero-frequency bytes in a data block in each pass. Here, block-wise byte pair transformation is performed in specified small number of passes k << maximum m. This reduced number of passes is suppoised to reduce exceution time.

In the first k-1 passes, half of the remaining unused bytes are used to replace most frequent byte pairs in each pass. In the last pass, all remaining unused bytes are used for replacing frequent integers. For k=3, half of the total unused bytes are used in 1st pass, half of the remaining (1/4th of the total unused) are used in the 2nd pass and then remaining (1/4th of the total) are used in 3rd pass.

The intention of proposed k-pass BPT-Z is to reduce the time of execution in addition to achieve compression.

2. LITERATURE REVIEW

In most of the source data, adjacent bytes are observed to be repeated many times [7]. Therefore, it is a better source for achieving data compression where 2-bytes are replaced with single byte. Byte-Pair Encoding (BPE) [1,5], digram encoding [3,4] and Iterative Semi-Static Digram Coding (ISSDC) [2] are such algorithms. These algorithms are intended for text files. However, they can be applied to any type of source.

Digram encoding and its variation ISSDC are dictionary based algorithms. Dictionary is created from the source data before transformation and it is used by decoder also.

Digram coding first builds dictionary and then does encoding in the second pass. All the individual 1-byte characters used in the source are added to the first part of the dictionary and the most frequently used digrams are added to the second part of the dictionary. If the source contains n individual characters, and the dictionary size is d, then the number of digrams that can be added to the dictionary is d − n. The n and d values and the dictionary are written in the destination file along with encoded data for decoder. Digrams are encoded using index position in the dictionary.

Altan Mesut and Aydin Carus [2] in their Iterative Semi-Static Digram Coding (ISSDC) use repeated digram coding. It requires the entire source to be in memory due to its two-pass multi-iterations.

Digram encoding and ISSDC are beneficial only when small-size alphabet is used in the source. If all 256 symbols are used in the source, the dictionary size needs to be longer than 256 words and each digram in the dictionary will be encoded using more than 8 bits.

Authors of this paper had proposed QBT-I [6] and BPT-I [7] dictionary based transformation techniques. They are intended to introduce redundancy in the data for second stage conventional data compression techniques. Algorithm forms logical groups of most frequent quad-byte or byte-pair data in dictionary and encodes data using variable-length group number and index position of data within group. Thus a quad-byte or byte-pair is encoded with less than 16-bits.

Byte pair encoding (BPE) is a text compression algorithm proposed by P. Gage [1]. Authors of paper [8] have used BPE to accelarete pattern matching.

BPE compresses data by finding the most frequent pair of adjacent bytes in the data and replacing all instances of this pair with a byte that was not in the original data. The algorithm repeats this process until no further compression is possible, either because there are no more frequently occurring pairs or there are no more unused bytes to represent pairs [9]. Let us consider m as the number of passes or number of times the process is repeated. In each pass, the algorithm writes out the pair substitution information before the packed data. This data is used for encoding in the next pass.

Byte-pair encoding (BPE) is beneficial only when the source is having small alphabet size, because it needs unused zero-frequency byte symbols for encoding most frequent byte-pairs.

Another drawback of this BPE algorithm is that it is using multiple passes, replacing only single most frequent byte pair with single unused byte at a time in each pass in the entire file. To avoid frequent file i/o, entire data is stored in memory.

BPE has two potential problems: 1. Some files may be too large to fit in memory; 2. Large binary files may not contain unused byte to encode byte pair and therefore compression may not be achieved.

These problems are solved by applying BPE to blocks of data instead of entire file at a time. This block-wise BPE variation [1,5] in the algorithm requires buffering small blocks of data and compressing each block separately. Here additional cost incurs due to storing substitution information and the output block size information for each data block.

The advantages of using block-wise BPE are:

- Increased chances of finding unused (zero-frequency) bytes in a small block as compared to finding unused bytes in entire file
- It provides local adaptation to varying data and improves overall compression.
- It can be implemented using parallel programming.

The source code for block-wise BPE is available at mattmahoney.net/dc/bpe2v2.cpp [5]. It is m-pass encoding where m is the maximum possible number of passes. At each pass, it replaces the single most frequent byte-pair using one unused byte in a bock. Here, we have denoted it as m-pass BPT-Z (maximum pass Byte Pair Transformation using Zero-frequency bytes).

3. RESEARCH SCOPE

Digram encoding, ISSDC and BPT-I are all dictionary based techniques using index in encoding byte pair. Digram encoding and ISSDC are better only when applied to small size alpahbet source. QBT-I and BPT-I are intended for introducing data redundancy for second stage data compression.

The problem with BPE and m-pass BPT-Z is the large execution time of encoding due to maximum possible passes.

A reserach scope is seen in reducing this large execution time taken by BPE and m-pass BPT-Z.

An assumption is that the reduced number of passes will reduce the transformation time.

4. INTRODUCTION TO BPT-Z

In all BPE, m-pass BPT-Z and k-pass BPT-Z, each pass is of two stages:

1. Determine unused bytes and frequency of byte-pair data
2. Perform transformations on most frequent byte-pairs by substitution with unused zero-frequency bytes

The main difference in BPE and m-pass BPT-Z is the data size used in encoding. BPE transforms only one most frequent byte-pair at a time and it is applied on entire file; whereas m-pass BPT-Z also transforms only one most frequent byte-pair at a time but is applied on small chunks of data blocks of a file.

Like m-pass BPT-Z, k-pass BPT-Z also performs block-wise encoding. The difference is that k-pass BPT-Z transorms more than one most-frequent byte-pairs at a time in stead of only one byte pair at a time. Additionally user can experiment with varying number of passes.

4.1 m-Pass BPT-Z

Let us first understand m-pass BPT-Z with example. With m-pass byte-pair transformation, only one most frequent byte-pair is transformed in each pass. Thus header containing substitution information is of 3 bytes; 2 bytes for byte-pair to be replaced and 1 byte for byte used to replace byte-pair. If most-frequent byte pair occurs less than 4 times, there is no saving. Thus the process stops when most frequent byte pair occurs less than 4 times or there is no more zero-frequency byte.

Consider “abcdababdcdabcdcdabcdab” as the source data to be encoded.

Pass-1

- Input: abcdababdcdabcdcdabcdab, length=23
- Most frequent Byte-pair: ab with frequency 6
- Encoding: substitute ab by unused symbol e
- Output (header, transformed data): abeecdeedcdecdcdecde, length:20

header	Transformed data	Total length
abe	ecdeedcdecdcdecde	20

Thus abcdababdcdabcdcdabcdab is transformed to ecdeedcdecdcdecde with header information abe.

Pass-2 (input is output of pass 1)

- Input: abeecdeedcdecdcdecde, length=20
- Most frequent Byte-pair: cd with frequency 5
- Encoding: substitute cd by unused symbol f
- Output (header, transformed data): cdfabeefeedfeffefe, length:18

Header	Transformed string	Total length
cdf	Abeefeedfeffefe	18

Thus abeecdeedcdecdcdecde is transformed to abeefeedfeffefe with header information cdf.

Pass-3 (input is output of pass 2)

- Input: cdfabeefeedfeffefe, length:18
- Most frequent Byte-pair: fe with frequency 4
- Encoding: substitute fe by unused symbol g
- Output (header, transformed data): fegcdfabeegedgfgg, length:17

Header	Transformed string	Total length
feg	Cdfabeegedgfgg	17

Thus cdfabeefeedfeffefe transformed to cdfabeegedgfgg, header information feg.

Pass-4 (input is output of pass 3)

- Input: fegcdfabeegedgfgg, length:17
- Most frequent Byte-pair: eg with frequency 2

The algorithm stops here, as there is no byte pair with frequency > 3.

If most frequent byte pair is repeated 3 times, saving is only of 3 bytes in transformation. This saving nullifies due to 3-byte in header. Thus, there is no benefit.

Note that it requires finding most frequent byte pair in each pass. It involves scanning data buffer, finding most frequent byte pair. This adds to the cost of execution time.

4.2 k-Pass BPT-Z

With k-pass BPT-Z, it transforms one or more byte pair in each pass. The header contains the information: n=number of byte pairs used in substitution and n times substitution information. Thus header information requires (1+n*3) bytes. Here also, if frequency of byte pair is <4, it is not reducing the data size due to overhead of this header information. So, the process stops either due to no more byte pairs to transform or no more zero-frequency bytes for substitution or number of passes are over.

k-pass BPT-Z with k=1

Here, all possible byte pairs are transformed at once in single pass only. Consider Consider “abcdababdcdabcdcdabcdab” as the source data to be encoded.

Pass-1

- Input: abcdababdcdabcdcdabcdab, length=23
- Most frequent Byte-pairs: ab with frequency 6, cd with frequency 5 and da with frequency 4.
- Encoding: substitute ab by e, cd by f and da by g; where e, f and g are unused symbols.
- Output (header, transformed data): 3abecdfdagefeedfeffefe, length:22

Header	Transformed data	Total length
3abecdfdag	Efeedfeffefe	22

Thus abcdababdcdabcdcdabcdab is transformed to efeedfeffefe with header information 3abecdfdag.

Note that byte pair da is repeated 4 times but is a part of cda where cd is repeated 5 times. Thus in the result, byte pair da is not encoded at all, but its information is stored in header. This is the overhead cost.

k-pass BPT-Z with k=2

Half of the possible byte-pairs are encoded in the first pass.

Pass-1: encode half of the possible byte pairs.

- Input: abcdababdcdabcdcdabcdab, length=23
- Most frequent Byte-pairs: ab with frequency 6, cd with frequency 5 and da with frequency 4.
- Encoding: substitute half of the 3, say 2 most frequent byte pairs ab and cd by e and f.
- Output (header, transformed data): 2abecdfefeedfeffefe, length:19

Header	Transformed data	Total length
2abecdf	Efeedfeffefe	19

Pass-2: encode all possible byte pairs.

- Input: 2abecdfefeedfeffefe, length=19
- Most frequent Byte-pairs: fe with frequency 5
- Encoding: substitute fe by g.
- Output (header, transformed data): 2abecdfefeedfeffefe, length:19

Header	Transformed data	Total length
1feg	2abecdggedgfgg	18

5. k-PASS BPT-Z ALGORITHM

Here, the file is processed reading a data block with specified number of bytes, say BlockSize.

5.1 k-pass Encoder

With k-pass BPT-Z, for each block, substitution takes place by using half of the remaining unused bytes in every k-1 passes. In the last pass, all possible substitutions takes place.

At each pass, the encoded buffer contains the header information (number of substitution pairs, values in substitution pairs (byte-pair, unused byte)) and then encoded data. After k passes, for each block, it writes encoded buffer size and the encoded buffer itself.

- Repeat till end of file
    - Read a block of size block_size in array, say buf
    - MaxToReplace=0
    - Repeat for k-1 times or until MaxToReplace=0
        - Determine number of unused (zero-frequency) bytes, say m
        - Store unused bytes in an array, say unused
        - Determine frequency of byte pairs. Here binary search tree (BST) data structure is used.
        - Sort the byte pairs in the order of their frequency
        - Find number of byte pairs, say n, to be considered for replacement. It is considered for replacement if its occurrence is more than 4 times.
        - Let maxToReplace = min(m, n)/2; maximum number of byte pairs chosen to be replaced by unused bytes
        - Write the value of maxToReplace and those many substitution pairs (byte pair, unused byte) for maxToReplace most frequent byte pairs to output buffer
        - Repeat till end of buf
            - Read two bytes number from buf
            - If it is from list of byte pairs to be replaced, write corresponding unused byte in output buffer and move ahead for next byte pair in buf; else copy first byte of byte pair in output buffer and move ahead by 1 byte in buf
    - Repeat above process for last pass with maxToReplace=min(m,n)
    - Write the length of output buffer to output file
    - Write the output buffer to output file

5.2 k-PASS Decoder

- Repeat till end of decoded file
    - Read block_len = size of encoded block from decoded file
    - Read buffer of block_len bytes from file to buf
    - Read m (number of byte pairs chosen for replacement) from buf
    - Read m substitution pairs (byte pair, unused byte) in array from buf
    - Repeat till end of buf
        - Read 1 byte, say ch, from buf
        - If ch is same as any of the unused byte in array of substitution pairs, write corresponding byte pair in output file; else write ch in output file

6. EXPERIMENTAL RESULTS AND ANALYSIS

Programs for BPT-Z are written in C language and compiled using Visual C++ 2008 compiler.

Programs are executed on a personal computer with Intel(R) Core(TM)2 Duo T6600 2.20 GHz processor and 4GB RAM.

(BPT-Z is performed with 1 and k (2 or more) passes in addition to maximum m-pass (implementation similar to code available at mattmahoney.net [5].

Experimental results are recorded using average of five runs on each test files. Most of the test files are selected from Calgary corpus, Canterbury corpus, ACT web site. Test files are selected to include all different file types and various file sizes as shown in Table 1.

Table 1 gives the experimental results on 18 test files with total size of nearly 40 MB, overall transformation rate (% of saving in transformed file size), total execution time of transformation and reverse transformation of all test files.

Transformation rate and BPS given in Table 1 shows that m-pass BPT-Z gives better compression as compared to k-pass BPT-Z. Compression rate achieved due to transformation is 14.72%, 20.12%, 21.89%, 22.67% and 22.96% with 1, 2, 3, 4 and 6 passes respectively as compared to 25.55% using maximum m-passes. As seen in figure 1, in k-pass BPT-Z, as value of k increases, compression improves.

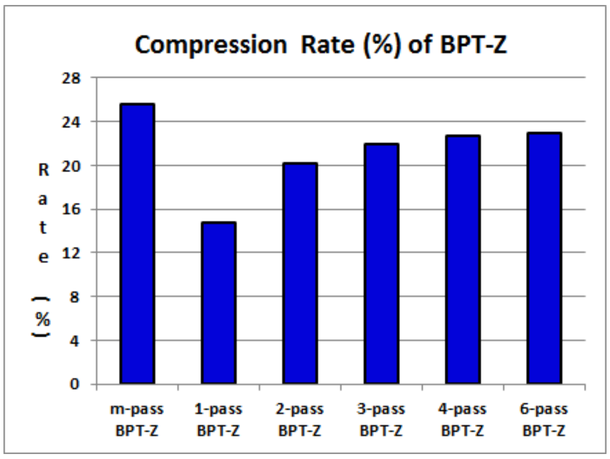

Fig 1: Compression Rate

As shown in Table 1, BPT-Z transformation process have taken 3.213, 9.794, 13.324, 16.323, 22.388 and 295.642 seconds to transform data using 1, 2, 3, 4, 6 and m-passes respectively. As seen in figure 2, time taken by m-pass transformation is significantly large as compared to k-pass BPT-Z. As per figure 2, transformation time increases linearly with varying value of number of passes.

Reverse transformation is comparatively very fast as expected. Time taken to get original data back is only 6.294 seconds with m-pass BPT-Z and 1.569, 3.213, 2.61, 3.623, 3.742 seconds with k-pass BPT-Z for number of passes 1, 2, 3, 4, 6 respectively. Again, execution time increases with increased number of passes but it is too small to get back data of size 40 MB. Figure 3 represents this decompression time.

It is observed that transformation and inverse transformation time is very high with m-pass BPT-Z as compared to k-pass BPT-Z. However, with m-pass BPT-Z, transformation rate is nearly 3.5% higher as compared to 6-pass BPT-Z showing better data compression. Thus, one needs to have trade-off between improvement in compression and execution time.

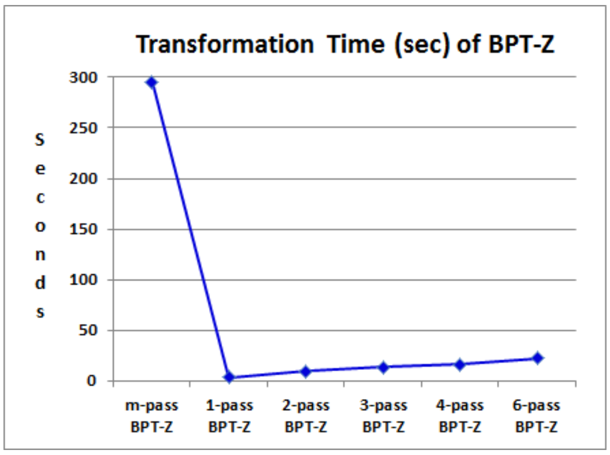

Fig 2: Transformation Time

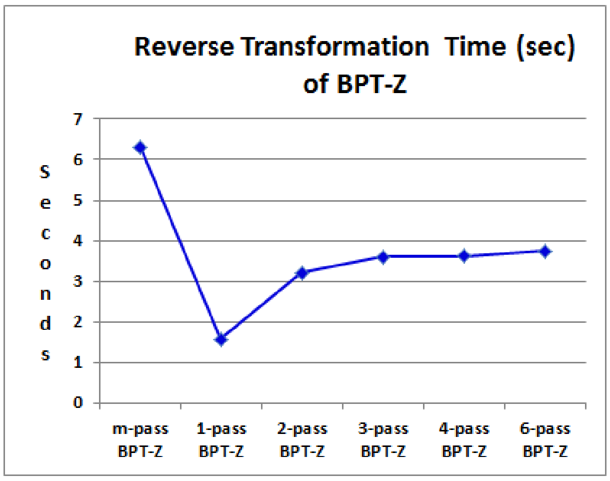

Fig 3: Reverse Transformation Time

Table 1. Experimental Results and Analysis of BPT-Z with varying number of passes

No.	Source File Name	Source size (Bytes)	Transformed File Size (Bytes) after m-pass and k-pass BPT-Z					
			m pass	**1 pass**	**2 pass**	**3 pass**	**4 pass**	**6 pass**
1	act2may2.xls	1348036	579249	907329	750499	676313	625412	623398
2	calbook2.txt	610856	333395	404795	367722	361070	359220	358650
3	cal-obj2	246814	151999	177315	168221	165241	164392	164050
4	cal-pic	513216	67297	279818	171054	121302	99031	86056
5	cycle.doc	1483264	564010	1001966	784175	685409	639858	611538
6	every.wav	6994092	6995149	6995121	6995992	6996846	6997700	6999408
7	family1.jpg	198372	198267	198376	198317	198337	198359	198408
8	frymire.tif	3706306	1025735	2247839	1679718	1469078	1389311	1348727
9	kennedy.xls	1029744	220447	608919	388373	337542	302425	294626
10	lena3.tif	786568	779995	783809	780443	780509	780584	780771
11	linux.pdf	8091180	5924171	6893760	6428656	6272289	6215611	6195157
12	linuxfil.ppt	246272	177750	209089	192714	186006	182977	180931
13	monarch.tif	1179784	1086942	1134918	1103577	1101105	1100468	1100193
14	pine.bin	1566200	1084151	1258230	1158763	1145406	1141295	1139168
15	profile.pdf	2498785	2493138	2496010	2494867	2494487	2494566	2495103
16	sadvchar.pps	1797632	1728078	1755441	1739964	1736218	1735210	1735154
17	shriji.jpg	4493896	4483977	4489213	4488723	4488754	4489223	4490235
18	world95.txt	3005020	1733330	2096542	1896547	1867720	1859872	1857181
Total Size (Bytes)		**39796037**	**29627080**	**33938490**	**31788325**	**31083632**	**30775514**	**30658754**
Overall Transformation Rate (%)			**25.553**	**14.719**	**20.122**	**21.893**	**22.667**	**22.960**
Overall BPS (Bits Per Symbol)			**5.956**	**6.822**	**6.390**	**6.249**	**6.187**	**6.163**
Total Transformation Time (Sec)			**295.642**	**3.213**	**9.794**	**13.324**	**16.323**	**22.388**
Total Inverse Transformation Time (Sec)			**6.294**	**1.569**	**3.213**	**3.610**	**3.623**	**3.742**

7. OTHER FACTORS

Other than number of passes in BPT-Z, factors like block-size and data structures used also affect the performance.

Larger block-size will reduce the possibility of finding zero-frequency bytes in block, especially in binary files. So, the compression may be poor. In our experiment, we have used 8KB size data block.

The data structure used in our experiment is Binary Search Tree. The node of the tree contains byte-pair and its frequency. BST is used to speedup search and sort.

Use of direct access with array data structure may improve the speed.

8. CONCLUSION

As the number of passes in BPT-Z increases, transformed file size is reduced but transformation time is increased. With 6-pass BPT-Z, compression achieved is nearly 2.5% less as compared to m-pass BPT-Z, but transformation time taken by m-pass BPT-Z is extremely high as compared to k-pass BPT-Z. As a trade-off, BPT-Z with 4 or 6 passes can be taken as optimal considering both compression and execution time.

As expected, k-pass BPT-Z is considerably faster to execute as compared to conventional m-pass BPT-Z.

9. REFERENCES

[1] Philip Gage, "A New Algorithm For Data Compression", The C Users Journal, vol. 12(2)2, pp. 23–38, February 1994

[2] Altan Mesut, Aydin Carus, “ISSDC: Digram Coding Based Lossless Data Compression Algorithm”, Computing and Informatics, Vol. 29, pp.741–754, 2010

[3] Sayood Khalid, "Introduction to Data Compression",2nd edition, Morgan Kaufmann, 2000

[4] Ian H. Witten, Alistair Moffat, Timothy C. Bell, “Managing Gigabytes-Compressing and Indexing Documents and Images”, 2nd edition, Morgan Kaufmann Publishers, 1999

[5] mattmahoney.net/dc/bpe2v2.cpp

[6] Jyotika Doshi, Savita Gandhi, “Quad-Byte Transformation as a Pre-processing to Arithmetic Coding”, International Journal of Engineering Research & Technology (IJERT), Vol.2 Issue 12, December 2013, e-ISSN: 2278-0181

[7] Jyotika Doshi, Savita Gandhi, “Article: Achieving Better Compression Applying Index-based Byte-Pair Transformation before Arithmetic Coding”, International Journal of Computer Applications 90(13):42-47, March 2014.

[8] Y. Shibata, T. Kida, S. Fukamachi, T. Takeda, A. Shinohara, S. Shinohara and S. Arikawa, "Byte-pair encoding: An text compression scheme that accelerates pattern matching", Technical Report, Department of Informatics, Kyushu University, Japan, 1999.

[9] Aydin Carus, Altan Mesut, “Comparison of the Performance of Compression Methods Depending on Natural Languages”, International Scientific Conference 23-24 Nov 2007, GABROVO