International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 3 Issue 2, February - 2014

Improved Compression Rate using Quad-Byte Index Based Transformation as a Pre-Processing to Arithmetic Coding

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

Savita Gandhi
Dept. of Computer Science, Gujarat University
Navrangpura
Ahmedabad-380009, INDIA

***Abstract*—Transformation algorithms are used to increase redundancy in data sets and achieve better compression when conventional compression techniques applied later. Arithmetic coding is the most widely preferred entropy encoder used in most of the compression methods. It is nearly optimal and compression rate cannot be further improved without changing the data model. In this paper, we have used QBT-I (Quad-Byte Transformation using Indexes) technique to change the data model and introduce more redundancy in the data. We have experimented QBT-I at a pre-processing stage before applying arithmetic coding compression method. QBT-I transforms most frequent 4-byte (quad-byte) integers. Most frequent quad-bytes are arranged in sorted order of their frequency and then divided in a group of 256 quad-bytes. Each quad-byte in a group is encoded using two tokens: group number and the location in a group. Group number is denoted using variable length codeword; whereas location within a group is denoted using 8-bit index. QBT-I can be applied on any source; not necessarily text or image or audio. Minimum of 2.5% compression gain is observed using QBT-I at a pre-processing as compared to compression using only arithmetic coding. Increasing number of groups gives better compression.**

Keywords—data compression, data transformation, quad-byte transformation, arithmetic coding

I. INTRODUCTION

Data transformation transforms data from one format to another. When data transformation is applied before applying conventional compression, the main purpose of a data transformation is to re-structure the data such that the transformed file is more compressible by a second-stage conventional compression algorithm. The intention here is to improve the overall compression rate as compared to what could have been achieved by using only arithmetic coding compression algorithm.

Majority of the data compression methods transforms data first and then apply entropy coding in the last step. Some of such methods are: LZ algorithms [21, 25, 28, 29]; DMC (Dynamic Markov Compression) [2, 4]; PPM [15] and their variants, context-tree weighting method [26], Grammar—based codes [10] and JPEG-MPEG methods used for image and video compression. Earlier-generation image and video coding standards such as JPEG, H.263, and MPEG-2, MPEG-4 were using Huffman coding in the entropy coding step; whereas recent generation standards including JPEG2000 [7, 23] and H.264 [13, 24] utilize arithmetic coding.

Arithmetic coding [9, 12, 27] is the most widely preferred efficient entropy coding technique providing optimal entropy. Here, the problem is that further improvement in compression is not possible due to its entropy limitations. To achieve better compression, the only possibility is to change the data model and have it more skewed. One way to change the data model is applying data transformation.

Authors of this paper have proposed Quad-Byte Transformation using Index (QBT-I) method with an intention to introduce more redundancy in the data and make it more compressible using arithmetic coding at the second stage [6]. Implementation of this proposal has resulted in minimum of nearly 2.5% improvement in compression as compared to compression using only arithmetic coding.

QBT-I transforms most frequent quad-bytes (4-byte integers) forming various groups of 256 quad-bytes and then encoding quad-byte using two tokens: group number and location (8-bit index) of quad-byte within a group.

Due to two-stage process of transformation and then compression, it is obviously going to be somewhat slower. This slowness is acceptable since the transform truly skews the data source to fulfil our purpose of achieving more compression.

Another advantage of QBT-I is that it can be applied to any type of source; may be text, binary file, image, video or any other format.

II. LITERATURE REVIEW

Most of the research work in data transformation is intended to compress specific type of files. Transformation techniques like DCT and wavelet are used for image files.

Burrows Wheeler Transform (BWT) [3, 16] performs block encoding. Even though it is intended for text source only, it can be used for any source. For each block, BWT requires rotation-sorting-indexing. It is very time consuming and requires better data structures for efficient pattern matching. It gives better compression only when it is combined ad-hoc compression techniques Run Length Encoding (RLE) and Move-To-Front (MTF) encoding and then entropy coding.

Star family transformation techniques are intended to compress text files. Star Transform [11], Length Index Preserving Transform (LIPT) [1, 17], and StarNT [22] are some such techniques shown in Table 1.

TABLE I. STAR FAMILY TRANSFORMATION TECHNIQUES

	Star encoding	LIPT	StarNT
Source Type	Text	Text	Text
Dictionary	22 sub-dict	22 sub-dict	single
Size of token to be encoded	word upto 22 letters	word upto 22 letters	Word
comparison time per token	O(Sub-Dict-size)	O(Sub-Dict-size)	O(Dict- size)
Code length	variable length: word-size	variable length: <*, word length, index>	variable length: index with max. 3-letters
Redundancy using	*	index, length	index, length
Compression methods that can be applied later	RLE, LZW, Huffman, Arithmetic coding	Huffman or Arithmetic Coding	
Drawback	• Applicable to text source only • Requires pattern matching		

Dictionary Based Encoding (IDBE) [20], Enhance Intelligent Dictionary Based Encoding (EIDBE) [18] and Improved Intelligent Dictionary Based Encoding (IIDBE) [19] are the transformation methods used for text files as shown in Table 2. They transform words using their index position in the dictionary.

TABLE II. DICTIONARY BASED ENCODING TECHNIQUES

	IDBE	EIDBE	IIDBE
Source Type	Text	Text	Text
Dictionary	single	22 sub-dict	22 sub-dict
Size of token to be encoded	Word	word	word
comparison time per token	O(Dict-size)	O(sub-dict-size)	O(sub-dict-size)
Code length	variable length: <1-byte codeword length, codeword>	variable length: <1-byte word length, codeword>	variable length: <1-byte codeword length, codeword>
Redundancy using	(index, length) index using ASCII characters 33-250, length 1-4 using ASCII characters 251-254	(index, length) index using ASCII characters 33-231, length 1-22 using ASCII characters 232-253	(index, length) index using characters (A-Z, a-z) as in StarNT, length 1-22 using ASCII characters 232-253
Compression methods that can be applied later	Pre-processing to BWT, Later MTF and RLE and entropy encoding		
Drawback	• Applicable to text source only • Requires pattern matching • Compression-time more as used as a pre-processing to BWT		

Methods BPE (Byte Pair Encoding) [8], digram encoding and ISSDC (Iterative Semi-Static Digram Coding) [14] are also intended for text files, but can be applied to any type of source. They will benefit more only when applied to small-alphabet source like text files.

TABLE III. DIGRAM BASED ENCODING TECHNIQUES

	Digram encoding	ISSDC	BPE
Source Type	Any	Any	Any
Dictionary	semi-static	semi-static	---
Size of token to be encoded	Digram (2 bytes)		
Matching	string or integer comparision		
comparison time per token	O(Dict-size)	O(Dict-size)	O(1): 2-byte comparison
Code length	fixed, depends on dictionary size		1 byte
Redundancy using	index	index	substitution
Compression methods that can be applied later	Huffman or Arithmetic coding		
Drawback	benefits only with small-alphabet source	repetitive, benefits only with small sized source file and small alphabet source	repetitive, benefits only when source have some unused symbols, i.e. for small alphabet source

Many of the present day transformation techniques, along with transforming data, may introduce some compression also. Additionally they retain enough context and redundancy for later applied compression algorithms to be beneficial.

III. RESEARCH SCOPE

Star family and dictionary based methods are applicable to text source only and string matching is time consuming.

BWT can be applied to any source even though it is designed for text files. It is very slow due to the need of rotations, sorting and mapping. It gives good compression only when later applied sequence of MTF, RTF and entropy encoding.

All these methods require better data structures and pattern matching algorithms for efficiency.

Digram based encoding can be applied to any type of source, but they will be beneficial only for small-alphabet source files like text. Here the advantage is of integer comparison leading to speed in transformation.

We saw a research scope in transforming quad-bytes instead of digrams. Our assumption is that it will result in a reduced file size and take less transformation time as compared to digram based transformation.

As arithmetic coding is the most widely used entropy encoding method used with almost all compression methods, we intend to apply quad-byte transformation that can be applied to any type source data and introduce redundancy to skew the distribution for getting better compression using arithmetic coding later.

We have already proposed Quad-Byte Transformation using Index (QBT-I) in paper [6]. In this paper, we have shown experimental results that proved our assumption true.

IV. Brief Introduction to QBT-I

QBT-I transforms most frequent quad-bytes.. It first prepares the dictionary of quad-bytes sorted in decreasing order of their occurrence. The dictionary is then logically divided into groups of 256 quad-bytes. Number of groups may be specified by a user. If number of groups is nGrp, then it can encode (256 x nGrp) quad-bytes.

Each quad-byte found in the dictionary is encoded using two tokens; group number and the location of quad-byte within a group. Group number is denoted using variable length prefix codeword and location is denoted using 8-bit index. Redundancy is introduced with 8-bit index. More the number of groups; more is the redundancy and better is the compression assumed to be achieved using arithmetic coding later.

For decoder, it requires to know whether it is reading transformed quad-byte or not. Encoder uses group codeword 0 to denote untransformed quad-byte. Minimum-length codeword 0 is chosen considering that most of the quad-bytes will not be available in the dictionary. Quad-bytes found in dictionary are encoded using variable length prefix code starting with bit 1 to denote group number and its index position within a group.

Thus, a quad-byte integer is transformed using two components <variable-length prefix code for group number, 8-bit index code>.

Here, prefix codes are 0, 10, 110, 1110, 11110,....,all 1s. Prefix codes are 0 and 1 for nGrp=1; 0, 10, 11 for nGrp=2; 0, 10, 110, 111 for nGrp=3 and so n. Thus, prefix codes denotes the group: code starting with 0 implies no transformation, with as many 1s as the number of groups implies the last group and otherwise it implies group number 1 to nGrp-1.

8-bit index codeword introduces redundancy in the dataset. To exploit redundancy at the time of arithmetic coding, we have kept group code and index code in separate files.

Use of variable length code leads to more reduction the size of transformed file. Most frequent codes reside in the initial groups and are assigned shorter prefix code.

Shortest prefix code 0 is used for untransformed integers assuming smaller dictionary size. Smaller dictionary sizes helps to speed up the search process.

V. Algorithm

Algorithm uses two output files: **transformed data file** and **code file**.

The **transformed data file** contains the index codewords (for transformed quad-bytes only). Purpose of storing index codewords separately is to introduce redundancy in the data for better compression.

Prefix codes denoting group codeword are copied in the **code file**.

The number of bytes in a source file may not be in multiple of size 4, so initial nExtrabytes (= filesize modulo 4) bytes are not processed and output as they are. Transformation is applied to remaining bytes.

The structure of **code file** is as follows:

- Byte 1: nExtrabytes (2 bits) and nGrp (6 bits, maximum 64 groups)
- Byte 2 to nExtrabytes+1: unprocessed initial extra bytes from source file
- Next 2 bytes: Dictionary size d = number of most frequent integers to be stored
- Next 4*d bytes: quad-bytes in descending order of frequency
- Remaining bytes: prefix codes of transformed and untransformed integers

A. *QBT-I Encoder:*

1. Setup:

a. Find source file size, Accept nGrp
b. nExtrabytes = filesize module 4
c. Combine nExtrabytes (2 bits) and nGrp (6 bits) in a byte and write in the code file
d. Read nExtrabytes bytes from source file and write to code file

2. Pass I (Dictionary building)
   a. Scan source file and compute frequency of quad-bytes
   b. Sort integers in descending order of the frequency.
   c. Output dictionary information in code file
      - Dictionary size = minimum (256 x nGrp, number of integers with frequency > 0)
      - Write dictionary size (using 2-bytes) and those many most frequent quad-bytes in the code file. Keep the dictionary stored in memory for later use in pass II. (One may use data structure like array or binary search tree (BST). BST is more efficient while searching.)
3. Pass II (Transformation: Rescanning the source from the beginning after extra bytes)
   a. Let prefix array contain binary numbers 10, 110, 1110,… for nGrp groups.
   b. Read integer.
   c. Search in dictionary.
   d. If found at location k in dictionary,
   - Output index = (k modulo 256) in transformed data file
   - Determine prefix code:

- Grp = k/256
- If Grp is the last group, i.e. value of Grp is same as nGrp-1, then write last prefix (i.e. nGrp times 1) to prefix code file
- If Grp is not the last group, write bits of prefix[Grp] to prefix code file

e. If not found in dictionary, output integer data in the transformed data file as it is and write prefix bit 0 in prefix code file.

f. Repeat steps from b onwards till all integers are scanned.

B. QBT-I Decoder:

1. Setup
   a. Read nExtrabytes and nGrp from code file
   b. Read nExtrabytes bytes from code file and write in output file
2. Dictionary building
   a. Read Dictionary size and corresponding number of integers from code file.
   b. Store most frequent integers in dictionary (in memory) in the order of their arrival. For dictionary, one may use data structures like array or Binary Search Tree.
3. Inverse Transformation:
   a. Fetch prefix code from code file (bits are extracted till either 0 is found or nGrp bits are extracted)
   b. If prefix code is 0 (i.e. untransformed data), read 4-bytes integer from transformed data file and write in the output file.
   c. If prefix is not 0, it means transformed data file contains 8-bit index for actual data.
      - Determine the group where the actual data belongs:
        - If prefix is all 1s (i.e. lastPrefix), Grp = nGrp-1 (i.e. last group)
        - Otherwise, search for prefix in prefix array. If it is found at location k, then Grp = k. (To avoid searching array, count number of leading 1s and then subtract 1 to determine Grp)
      - Determine location of the data in dictionary:
        - Read 1 byte index from transformed data file
        - Location of data in dictionary = Grp*256 + index
      - Write quad-byte from location in dictionary to output file.
4. Repeat step 3 till end of code file.

EXPERIMENTAL RESULTS AND ANALYSIS

Programs for QBT-I and arithmetic coding are written in C language and compiled using Visual C++ 2008 compiler.

Experiment is performed on computer with Intel(R) Core(TM)2 Duo T6600 2.20 GHz processor with 4GB RAM.

QBT-I is experimented with number of groups varying from 1 to 4. 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 4.

We have used AC-nShft implementation of arithmetic coding with multi-bit processing [5]. It is faster than conventional implementation of arithmetic coding.

Our prime motive is to improve compression using data transformation techniques as a pre-processing stage for applying arithmetic coding. So, execution time is not considered that important.

Table 4 presents transformed file size (bytes) after applying QBT-I with number of groups varying from 1 to 4. With QBT-I, it is observed that as number of groups increases, resulting transformed file size decreases. Larger number of groups may increase the size of code file due to the use of longer prefix codes; but at the same time, it also increases the number of transformed integers which results in smaller file size. Additionally it introduces more redundancy in data set.

Table 5 shows the size of compressed files as a result of compression (i) using only arithmetic coding (AC) and (ii) using AC after applying data transformation with QBT-I at pre-processing stage.

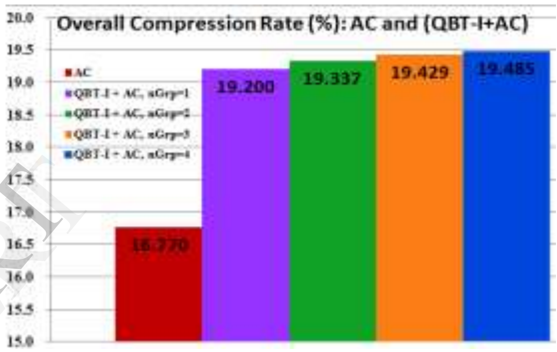

Fig. 1. Overall Compression Rate (%) using only AC, using AC after QBT-I with varying nGrp

As seen in Table 5 and Figure 1, increasing number of groups in QBT-I gives better compression; from 19.20% to 19.49% for nGrp=1 to 4. Minimum of 2.5% compression gain is observed using QBT-I over using only arithmetic coding.

Figure 2 presents the compressed file size of 18 individual test files using AC only and applying QBT-I transformation as preprocessing to AC. Here QBT-I is applied with only 256 most frequent quad-bytes in the dictionary; i.e. nGrp=1.

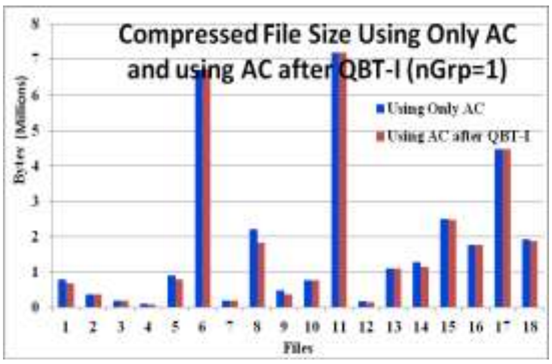

Fig. 2. Compressed File Size using only AC, using AC after QBT-I with nGrp=1

VI. FINDING MOST FREQUENT QUAD-BYTES

Possible values of quad-byte are from 0 to 4GB. To store the frequencies of all possible quad-bytes, use of array data structure needs memory of 4GB integers. Here, we have used binary search tree to accommodate initial 4096 distinct quad-bytes and used nGrp*256 most frequent quad-bytes.

VII. CONCLUSION

With QBT-I data transformation applied before arithmetic coding, our purpose of achieving better data compression is achieved. Using QBT-I at a pre-processing stage of arithmetic coding, more than 2.5% overall data compression is achieved over compression using only arithmetic coding.

TABLE IV. TRANSFORMED FILE SIZE (BYTES) AFTER APPLYING QBT-I

No.	File name	Corpus and Description	Source Size (Bytes)	File Size (Bytes) After Applying QBT-I Data Transformation			
				nGrp=1	nGrp=2	nGrp=3	nGrp=4
1	act2may2.xls	ACT: excel file	1348036	1019293	1020582	687486	685917
2	calbook2.txt	Calgary: troff format	610856	528536	496279	353580	351240
3	cal-obj2	Calgary: object file, Mac executable	246814	232428	231727	185236	184686
4	cal-pic	Calgary: CCITT fax file, bitmap image	513216	188699	200260	95375	95151
5	cycle.doc	Own: word doc with images, text,drawing	1483264	1013909	1031846	799990	798697
6	every.wav	ACT: sound file	6994092	7211412	7210525	6858272	6858198
7	family1.jpg	Own: photograph	198372	204187	204000	200671	200568
8	frymire.tif	ACT: graphics file	3706306	2401825	2369254	1852155	1846827
9	kennedy.xls	Canterbury: excel	1029744	605851	573195	390123	390488
10	lena3.tif	ACT: graphics file	786568	807800	805860	777784	777719
11	linux.pdf	Own: pdf file, large	8091180	8277780	8280338	7342506	7342367
12	linuxfil.ppt	Own: power-point with text, drawing	246272	180380	182313	154276	154101
13	monarch.tif	ACT: graphics file	1179784	1199573	1193682	1115984	1114711
14	pine.bin	ACT: executable	1566200	1361793	1343690	1141274	1138803
15	profile.pdf	Own: pdf file with text, photos	2498785	2558049	2558631	2525449	2525331
16	sadvchar.pps	Own: ppt show	1797632	1825283	1826270	1782983	1782868
17	shriji.jpg	Own: image file	4493896	4616222	4615417	4560971	4560877
18	world95.txt	ACT: text file	3005020	2677540	2576469	1865242	1856176
		Total Size	**39796037**	**36910560**	**36720338**	**32689357**	**32664725**

TABLE V. COMPRESSED FILE SIZE (BYTES) USING ONLY AC AND USING QBT-I BEFORE AC

No.	File name	Source	Compressed File Size (Bytes)				
		Size (Bytes)	using only AC	Applying AC (Arithmetic Coding) after data transformation using QBT-I			
				nGrp=1	nGrp=2	nGrp=3	nGrp=4
1	act2may2.xls	1348036	789951	670903	669755	667583	666612
2	calbook2.txt	610856	367017	357514	351368	347377	344817
3	cal-obj2	246814	194255	184946	184534	184083	183521
4	cal-pic	513216	108508	81292	84761	84657	84705
5	cycle.doc	1483264	891974	776520	782263	780583	779452
6	every.wav	6994092	6716811	6735354	6739310	6741340	6741060
7	family1.jpg	198372	197239	197905	197934	197877	197837
8	frymire.tif	3706306	2200585	1833394	1806508	1794159	1788738
9	kennedy.xls	1029744	478038	372619	371831	369205	369167
10	lena3.tif	786568	762416	761667	761177	761338	761442
11	linux.pdf	8091180	7200113	7198297	7202927	7203486	7203460
12	linuxfil.ppt	246272	175407	151576	152064	151819	151758
13	monarch.tif	1179784	1105900	1099243	1095444	1093428	1092356
14	pine.bin	1566200	1265047	1146782	1137193	1132226	1130004
15	profile.pdf	2498785	2490848	2483069	2480761	2484303	2484867
16	sadvchar.pps	1797632	1771055	1760557	1761645	1761713	1761710
17	shriji.jpg	4493896	4481092	4477193	4478571	4479594	4479663
18	world95.txt	3005020	1925940	1866426	1842754	1829326	1820744
Total Size		**39796037**	**33122196**	**32155257**	**32100800**	**32064097**	**32041913**
Overall Compression Rate			**16.77**	**19.2**	**19.337**	**19.429**	**19.485**
Overall Bits Per Symbol			**6.658**	**6.464**	**6.453**	**6.446**	**6.441**

REFERENCES

[1] F. D. Awan, N. Zhang, N. Motgi, R. T. Iqbal, A. Mukherjee. "LIPT: A reversible lossless text transform to improve compression performance", Proceedings of the IEEE Data Compression Conference (DCC'2001), pp. 481, March 27–29, 2001
[2] T.C. Bell, A. Moffat, "A Note on the DMC Data Compression Scheme", Computer Journal, vol. 32(1), pp.16-20, 1989
[3] M. Burrows,D. J. Wheeler. "A block-sorting lossless data compression algorithm", Digital Systems Research Center, Research Report 124, Digital Equipment Corporation, Palo Alto, California, May 10, 1994
[4] G.V. Cormack, R.N. Horspool, "Data Compressing Using Dynamic Markov Modeling", Computer Journal, vol. 30(6), pp.541-550, 1987
[5] Jyotika Doshi and Savita Gandhi, "Computing Number of Bits to be processed using Shift and Log in Arithmetic Coding", International Journal of Computer Applications 62(15):14-20, January 2013, Published by Foundation of Computer Science, New York, USA. BibTeX
[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] M. Dyer,D. Taubman, S. Nooshabadi, "Improved throughput arithmetic coder for JPEG2000", Proc. Int. Conf. Image Process., Singapore, pp. 2817–2820, Oct. 2004
[8] Philip Gage, "A New Algorithm For Data Compression", The C Users Journal, vol. 12(2)2, pp. 23–38, February 1994
[9] P. G. Howard, J. S. Vitter, "Arithmetic coding for data compression", Proc. IEEE. , vol.82: pp.857-865, 1994
[10] J. C. Kieffer, E. H. Yang, "Grammar-based codes: A new class of universal lossless source codes", IEEE Trans. Inform. Theory, vol. 46, pp. 737–754, 2000
[11] H. Kruse, A. Mukherjee. "Preprocessing Text to Improve Compression Ratios",Proc. Data Compression Conference, pp. 556, 1998
[12] G. Langdon, "An introduction to arithmetic coding", IBM Journal Research and Development, vol. 28, pp. 135-149, 1984
[13] Detlev Marpe, Heiko Schwarz, Thomas Wiegand, "Context-Based Adaptive Binary Arithmetic Coding in the H.264/AVC Video Compression Standard", IEEE Trans. On Circuits and Systems for Video Technology, vol. 13(7), pp. 620-636, July 2003
[14] Altan Mesut, Aydin Carus, "ISSDC: Digram Coding Based Lossless Dtaa Compression Algorithm", Computing and Informatics, Vol. 29, pp.741–754, 2010
[15] Moffat, "Implementing the PPM Data Compression Scheme", IEEE Transactions on Communications, vol.38, pp.1917-1921, 1990
[16] M. Nelson, "Data Compressin with the Burrows-Wheeler Transform", Dr. Dobb's Journal, pp. 46-50, Sept 1996 available at http://marknelson.us/1996/09/01/bwt/
[17] Radescu R., "Lossless Text Compression Using the LIPT Transform", Proceedings of the 7th International Conference Communications 2008 (COMM2008), ISBN 978-606-521-008-0., pp. 59-62, Bucharest, Romania, 5-7 June 2008
[18] Senthil S, Robert L, "Text Preprocessing using Enhanced Intelligent Dictionary Based Encoding (EIDBE)", Proceedings of Third International Conference on Electronics Computer Technology, pp.451-455, Apr 2011
[19] Senthil S, Robert L, "IIDBE: A Lossless Text Transform for Better Compression", International Journal of Wisdom Based Computing, vol. 1(2), August 2011
[20] Shajeemohan B.S, Govindan V.K, "Compression scheme for faster and secure data transmission over networks", IEEE Proceedings of the International conference on Mobile business, 2005
[21] Storer J. A., Szymanski T. G., "Data Compression via Textual Substitution", Journal of ACM Vol. 29(4), pp. 928-951, Oct 1982
[22] W. Sun, A. Mukherjee, N. Zhang, "A Dictionary-based Multi-Corpora Text compression System", Proceedings of the 2003 IEEE Data Compression Conference, March 2003
[23] S. Taubman and M. W. Marcellin, "JPEG2000: Image Compression Fundamentals", Standards and Practice. Norwell, MA: Kluwer Academic, 2002
[24] T. Wiegand, G. Sullivan, G. Bjontegaard, A. Luthra, "Overview of the H.264/AVC video coding standard", IEEE Trans. Circuits Syst.Video Technol., vol. 13(7), pp. 560–576, Jul 2003
[25] T. Welch, "A Technique for High-Performance Data Compression", IEEE Computer, vol. 17(6), pp. 8-19, June 1984
[26] M. J. Willems, Y. M. Shtarkov, T. J. Tjalkens, "The context-tree weighting method: Basic properties", IEEE Trans. Inform. Theory, vol.41, pp. 653–664, May 1995
[27] H. Witten, R. M. Neal, J. G. Cleary, "Arithmetic coding for data compression", Commun. ACM, vol. 30(6), pp. 520–540, 1987
[28] J. Ziv, A. Lempel, "Compression of individual sequences via variable rate coding", IEEE Transactions on Information Theory, IT-24(5), pp.530-536, 1978
[29] J. Ziv, A. Lempel. "A Universal Algorithm for Sequential Data Compression", IEEE Trans. Information Theory, IT-23, pp.337-343, 1977