BEHAVIORAL STUDY OF DATA STRUCTURES ON LEMPEL ZIV WELCH (LZW) DATA COMPRESSION ALGORITHM AND ITS COMPUTATIONAL COMPLEXITY

Nishad PM
Ph.D Scholar, Dr. Mahalingam Center for research,
Department of Computer Science
NGM College, Coimbatore Pollachi, India-642001
Email: nishadpalakka@yahoo.co.in

Dr. R. Manicka Chezian
Associate professor,
Dr. Mahalingam Center for research, Department of
computer science, NGM College
Coimbatore, Pollachi, India-642001
Email: chezian_r@yahoo.co.in

***Abstract*: LZW (Lempel Ziv Welch) compression is the first widely used universal data compression method on computers. After the invention of LZW there are lots of improvements and enhancement done on LZW in order to improve the compression ratio and reduce the computational complexity. LZW is very effective on the files containing lots of repetitive data especially for text and monochrome images. The LZW algorithm uses dictionary while decoding and encoding. The computational and space complexity of LZW data compression algorithm is purely depends on the effective implementation of data structure. A space efficient implementation of dictionary with data structure is most important. The data structure implementation must give better performance on the following operation: insertion of new pattern to the dictionary, searching of a given pattern and returning the matching code word if it is present in the dictionary as a phrase. This paper analyses a number of data structures commonly employed in the LZW dictionary based compression and decompression.**

Keywords - LZW, compression, decompression, computational, space complexity, monochrome, data structure.

I. INTRODUCTION

LZW compression became the first widely used universal data compression method on computers. After the invention of LZW there are lots of improvements and enhancement done in LZW for data compression that is discussed in this section. LZW compression works best for files containing lots of repetitive data especially for text and monochrome images. The LZW algorithm uses dictionary **D** while decoding and encoding. LZW compression uses a code table common choice is to provide 4096 entries in the table. In this case, the LZW encoded data consists of 12 bit codes, each referring to one of the entries in the code table. Decompression is achieved by taking each code from the compressed file, and translating it through the code table to find what character or characters it represents. Codes 0-255 in the code table are always assigned to represent single byte from the input file. When the LZW program starts to encode a file, the code table contains only the first 256 entries, with the remainder of the table being blank. This means that the first code in the compressed file is of single byte from the input file being converted to 12 bits. As the encoding continues, the LZW algorithm identifies repeated sequences in the data, and adds them to the code table. Compression starts the second time a sequence is encountered. The key point is that a sequence from the input file is not added to the code table until it has already been placed in the compressed file as individual characters (codes 0 to 255). This is important because it allows the decompression program to reconstruct the code table directly from the compressed data, without having to transmit the code table separately.

The compression algorithm uses two variables: CHAR and STRING. The variable, CHAR, holds a single character, (i.e.), a single byte value between 0 and 255. The variable, STRING, is a variable length string, (i.e.), a group of one or more characters, with each character being a single byte. The algorithm starts by taking the first byte from the input file, and placing it in the variable, STRING. This is followed by the algorithm looping for each additional byte in the input file. Each time a byte is read from the input file it is stored in the variable, CHAR. The data table is then searched to determine if the concatenation of the two variables, STRING+CHAR, has already been assigned a code. If a match in the code table is not found, three actions are taken, (i), output the code for STRING, When a match in the code table is found, (ii), the concatenation of STRING+CHAR is stored in the variable, STRING, without any other action taking place. That is, if a matching sequence is found in the table, no action should be taken before determining whether there is a longer matching sequence is present in the table or not. Since this longer sequence is not in the table, the algorithm adds it to the table, outputs the code for the shorter sequence that is in the table (code 256), and starts over searching for sequences beginning with the character, 'B'. This flow of events is continued until there are no more characters in the input file. The program is wrapped up with the code corresponding to the current value of STRING being written to the compressed file. LZW compression algorithm is illustrated in figure-1. The Decompression algorithm uses four variables NCODE, OCODE, STRING, and CHAR. The decompression algorithm starts by taking the first byte from the input file and placing it in the variable, OCODE and output the OCODE. This is followed by the algorithm looping for each additional byte in the input file; each time a byte is read from the input file it is stored in the variable, NCODE. The data table is then searched to find the variable NCODE. If a match in the code table is not found STRING = OCODE

978-1-4799-3966-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ICICA.2014.64

+CHAR else if the NCODE is found then STRING = NCODE, then output the STRING. First Character of STRING is assigned to CHAR, then adds entry (OCODE+ CHAR) in table for and assigns NCODE to OCODE. This process will continue up to the last input. The decoding algorithm is shown in figure-2.

```
STRING = get input character
  WHILE there are still input characters
{
        CHAR = get input character
        IF STRING+CHARACTER is in the string D the
        {
                STRING = STRING + CHAR
        }
        ELSE
        {
                Output the code for STRING
                Add STRING+CHAR to the string to D
                STRING = CHAR
        }
}
Output the code for STRING
```

Figure 1 LZW Encoding Algorithm

```
Read OLD_CODE
Output OLD_CODE
WHILE there are still input characters
{
Read NEW_CODE
STRING = get translation of NEW_CODE from D
Output STRING
CHARACTER = first character in STRING
Add OLD_CODE + CHARACTER to the translation D
OLD_CODE = NEW_CODE
}
```

Figure 2 LZW Encoding Algorithm

DEFINITION 1 Let us assume that $X = \{x_1, x_2 \dots x_n\}$ is the input sequence. The sequence length is denoted by n or $|X|$, i.e., a total number of elements in X. x_i denotes the i^{th} element of X. the element x_i belongs to the finite ordered set $A = \{a_0, a_1 \dots a_{k-1}\}$ that is called an alphabet with the corresponding probabilities $P = \{p_0, p_1 \dots p_{k-1}\}$. The number of elements in A is the size of the alphabet denoted by k, the elements of the alphabet are called symbols or characters. Let p_i be the weight, or probability of a_i. The sequence after compression is $C = \{C_1, C_2 \dots C_m\}$ where m or $|C|$ is the length of input sequence. $\bar{X} = \{\bar{x}_1, \bar{x}_2 \dots \bar{x}_n\}$ is the sequence after decompression, the size of the sequence is $|\bar{X}|$ and the time taken is denoted by t. D is the dictionary and size of the dictionary is |D| or N.

II. EMPIRICAL REVIEW

In order to improve the performance of dictionary based algorithm, two methods are suggested [1]. i). studding how to choose the minimum dictionary size to minimize the time taken and maximize compression. ii). Modifying the dictionary building process in order to reduce the ultimate dictionary size. The performance of LZW improved using three modifications. The first two enhancements eliminates the frequent flushing of the dictionary, this removes the computational complexity of LZW. The third enhancement improves the compression ratio. Two pass LZW [2] offline algorithm uses separate static dictionary is used for each file. Tire is constructed in the first phase. Actual Encoding carried in the second phase. The tire is attached to the compressed file. The tire like implementation leads to reduce the computational cost. A unique dictionary is partitioned into hierarchical variable word-width dictionaries. This allows us to search through dictionaries in parallel. Moreover, the barrel shifter [3] is adopted for loading a new input string into the shift register in order to achieve a faster speed. However, the original PDLZW uses a simple FIFO update strategy, which is replaced with a new window based updating technique is implemented to better classify the difference in how often each particular address in the window is referred. The freezing policy is applied to the address most often referred, which would not be updated until all the other addresses in the window have the same priority. This guarantees that the more often referred addresses would not be updated until their time comes; this leads to an improvement on the compression efficiency and low complexity. A new Two-Stage architecture that combines the features of both Parallel-dictionary –LZW (PDLZW) [4] and an approximated adaptive algorithm. Another two stage compression algorithm combines the features of PDLZW and Arithmetic Coding [5] [6]. In this architecture order list instead of tree based structure is used. The approach replace the hardware cost in addition the compression ratio and time cost is reduced. A multi processor based algorithm is called Bi Directory LZW (BDLZW) [7]. The implementation is approximately like PDLZW. The algorithm runs on multi processor system such as CELL, so this can minimize the computation time. The algorithm reads string from both the ends and loading two threads fairly and easily. Comparing the running time is ten percentages is improved then the conventional PDLZW when there are two threads running on CELL architecture. The parallel dictionary of LZW is implemented with the parallel VLSI architecture [8] can improve the throughput; the compression ratio and it reduce the time complexity because of the parallel search using the VLSI architecture. The algorithm combines the features of both parallel dictionary LZW (PDLZW) and an approximated Adaptive Huffman (AH) algorithm. The algorithm achieves compression ratio but cost computation little high than PDLZW. High speed Low-complexity register transfer logic (RTL) design and implementation [9] of LZW algorithm on Xilinx vertex in device family for high bandwidth applications. This offers high throughput and lower power requirements. Another hardware implementation using the enhanced content-Addressable-Memory (CAM) [10] cells to accelerate the implementation of LZW [11].

III. LZW COMPRESSION LINEAR ARRAY IMPLEMENTATION WITH LINEAR SEARCH

In this section analyze the linear array with linear search implementation on LZW dictionary. For this test and implementation used the linear array and for the lookup linear search is experimented. Each time STRING+CHARACTER is passed to the dictionary D for search ,the search is initialized form the lower bound of the dictionary to the upper bound of the dictionary in a linear manner and returned a Boolean value true or false based on the search result. After returning the return value the dictionary is updated by the STRING+CHARACTER as the top most element of the dictionary D if the search fails to find the STRING+CHARACTER, each time before the updating the dictionary the size of dictionary is incremented by one to accommodate the new element. Initially the size of the dictionary is zero, and the algorithm continued accordingly. The compression algorithm is given below. And the figure-3 illustrates the linear array implementation on LZW data compression.

0	AB
1	BC
2	CA
3	ABB
4	BA
5	ABBB
6	BCA
7	ABA
8	ABBBD

Figure -3 linear array dictionary implementation of LZW

Theorem 1 The Linear array with linear search implementation of LZW compression algorithm takes $|X| * \left(\frac{N+3}{4}\right)\ time$

Proof: After each unsuccessful search the dictionary size is incremented, the process is continued until the last element is inserted to the dictionary D. So the average case analysis is calculated as follows

$$= |X| * \left(\frac{1}{N}\left(\frac{1}{1} + \frac{1+2}{2} + \frac{1+2+3}{3} + \cdots + \frac{1+2+3+\cdots+N}{N}\right)\right)$$

$$= |X| * \left(\frac{1}{N}\sum_{i=1}^{N}\left(\frac{i+1}{2}\right)\right)$$

$$= |X| * \frac{1}{2N}\left(\sum_{i=1}^{N} i + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N(N+1)}{2} + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N+1}{2} + 1\right)$$

$$= |X| * \frac{1}{2}\left(\frac{N+3}{2}\right)$$

$$= |X| * \left(\frac{N+3}{4}\right)$$

IV. LZW DECOMPRESSION LINEAR ARRAY IMPLEMENTATION WITH LINEAR SEARCH

LZW decompression is the process of reversing the LZW compression. This section discussed about the LZW decompression with the linear array implementation. During the decoding operation no search operation will taken place. Instead of any kind of search operation the New_Code is used to check the availability of translation of New_Code in the dictionary, if it is available return the translation. If the translation is not available in the dictionary D then the New_Code is updated as the top most elements in the linear dictionary D after incrementing the Max_index. The dictionary is in a unsorted manner. This type of LZW decompression gives optimal reduction in the computation cost because only one comparison required in the dictionary needed to find the translation or to check the availability of new_code in the dictionary D. So the decompression approach may not need any improvements in order to reduce the computational cost.

Theorem 2 The Linear array implementation of LZW decompression algorithm takes O ($|X|$)$time$

Proof: Only single lookup required fetching the translation of New_Code then the time taken for decompression is

$$|C| * \left(\frac{1}{n}\sum_{1=1}^{n} 1\right)$$

$$= |C| * \left(\frac{1}{n}n\right)$$

$$= |C| * \left(\frac{n}{n}\right)$$

$$= |C| * (1)$$

$$= O(|C|)$$

V. BINARY SEARCH TREE IMPLEMENTATION OF LZW

In the previous section discussed and evaluated the implementation of LZW with Linear array. The BST (Binary search tree) implementation is also very famous in implementing the dictionary D. instead of Linear array this approach uses the binary Tree T for searching and updating. Like hashing the BST is also powerful approach.

In the each iteration of compression phase has two BST operations

i) Search for the pattern *STRING+CHAR:* - while compressing the sequence X the algorithm checks the availability of *STRING+ CHAR* in the dictionary D. for example initially the *STRING+CHAR* is 'AB'. The tree T has no nodes so the so the tree is updated with the *STRING+CHAR* shown in the figure- 4 (a). After the insertion the value of *STRING* and *CHAR* is re assigned to' B' and 'C' respectively then

again. Then the *STRING+CHAR* are 'BC', again the search operation take place. This time the search operation failed to find the STRING+*CHAR* in the tree T so again the insertion of tree is taken place shown in the figure 4(b). This process will continue until the last iteration.

ii) Insertion of the pattern *STRING+CHAR*:- Obviously the second operation is insertion *STRING+CHAR* on the tree if the search operation fails.

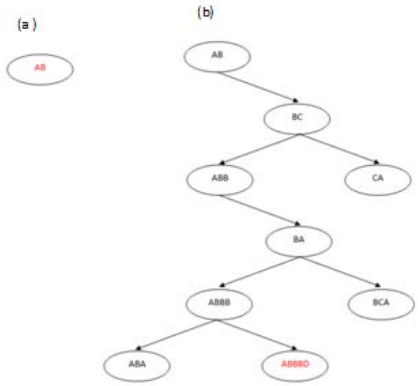


Figure -4 BST implementation of LZW Dictionary

In order to reduce the time complexity two operation is companied, i.e., if the search fails then the insertion operation carried out at the appropriate node. So the total time took for the compression algorithm is purely based on the number of iteration and the search required finding the element in T.

Theorem-3 The BST implemented LZW compression algorithm Takes

$$O\left(X * \left(log\left((N!)^{\frac{1}{N}}\right)\right)\right) time$$

Proof: While building D using BST, there are two possibilities of search successful and unsuccessful. Then consider the average of the both that is

$$= \frac{O\log(N) + O\log(N)}{2}$$

N is total number of nodes in the tree, considering the average. $O\log(N)$

The number of nodes is gradually increased after each unsuccessful search in the T. that is from1 $to\ n_N$. N is the number of nodes after last insertion. So in general we represent n_i, where i represent i[th] insertion in the T so n_i means the number of element after i[th] insertion. so the average comparison require after i[th] insertion is $O\log(n_i)$. Then the overall average comparison is calculated by.

$$= \frac{1}{N}(O\log(n_1) + O\log(n_2) + \cdots + O\log(n_{N-1}) + O\log(n_N))$$

$$= O\frac{1}{N}\left((\log(n_1) + \log(n_2) + \cdots + \log(n_{N-1}) + \log(n_N))\right)$$

$$= O\left((\log(n_1 * n_2 * \ldots * n_{N-1} * n_N))^{\frac{1}{N}}\right)$$

$$= O\left(\left(\log\prod_{i=1}^{N} n_i\right)^{\frac{1}{N}}\right)$$

$$= O\left(log\left((N!)^{\frac{1}{N}}\right)\right)$$

The number of iteration is base on the length of X

$N = |X|$

Where to BST operation takes place so

N

So the comparison required to compress the sequence X is

$$= O\left(X * \left(log\left((N!)^{\frac{1}{N}}\right)\right)\right)$$

VI. LZW COMPRESSION CHAINED HASH TABLE IMPLEMENTATION

In this section analyze the hash table implementation on LZW dictionary. For this test and implementation used the Chained hash table. The Hash function is used. Each time STRING+CHARACTER is passed to the hash table for search first the hash table find the Key value from the STRING+CHARACTER based on the length L and then the f (K) is evaluation and performed a mod operation to find the index i.e., $i = f(K)\ mod\ n$ and search performed base on the value of i or in the i[th] linked list and returned a Boolean value true or false based on the search result. Before returning the return value the hash table is updated by the STRING+CHARACTER as the last node of i[th] linked list if the search fails to find the STRING+CHARACTER, and the algorithm continued accordingly. The compression algorithm is given below. And the figure-5 illustrates the chained hash table implementation on LZW data compression. The compression algorithm is shown in the figure-1.

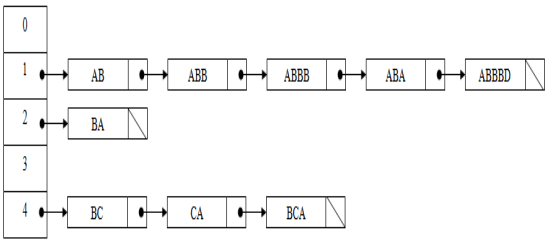


Figure-5 LZW Dictionary with Chained Hash table

Theorem 4: The average expected time for LZW compression with Chaining Hash table is

$O\left(|X| * (1 + AM\{\alpha_i\})\right)$

Proof: While performing the search in the LZW dictionary D the algorithm meets both successful and unsuccessful search so consider the average for the both that is

$$\frac{(1+\alpha) + (1+\alpha)}{2}$$

$$= \frac{(1+\alpha) + (1+\alpha)}{2}$$

$$= \frac{2 + (2 * \alpha)}{2}$$

$$= (1 + \alpha)$$

After each insertion to the dictionary D the load factor changes. For example after the first insertion the load factor is α_1. So the time required is $(1 + \alpha_1)$ and after the second insertion the load factor is α_2. So the time require for the search after the second insertion is $1 + \alpha_2$ and so on. So the average time require is.

$$\frac{(1 + \alpha_1) + (1 + \alpha_1) + (1 + \alpha_3) + \cdots + (1 + \alpha_{n-1}) + (1 + \alpha_n)}{n}$$

$$= \frac{n + (\alpha_1 + \alpha_1 + \alpha_1 + \cdots + \alpha_{n-1} + \alpha_n)}{n}$$

$$= \frac{n + \sum_{i=1}^{n} \alpha_i}{n}$$

$$= O\left(\frac{\sum_{i=1}^{n} \alpha_i}{n}\right)$$

$$= 1 + \frac{1}{n}\sum_{i=1}^{n} \alpha_i$$

$$= 1 + AM\{\alpha_i\}$$

Where $i = 1,2, \ldots, n$

The length of the X is n =|X| then

$O\ |X| * (\ 1 + AM\{\alpha_i\})$

VII. Experimental result

A. Experimental Setup

All experiments done on a 2.20 GHz Intel (R) Celeron (R) 900 CPU equipped with 3072KB L2 cache and 2GB of main memory. The machine had no other significant CPU tasks running and only a single thread of execution was used. The OS is Windows XP SP3 (32 bit). All programs are compiled using java version jdk1.6.0_13. The times are recorded in nanoseconds. The time taken of each algorithm is calculated using the tool compression time complexity evaluator and the graphs are plotted with MS Excel. All data structures reside in main memory during computation.

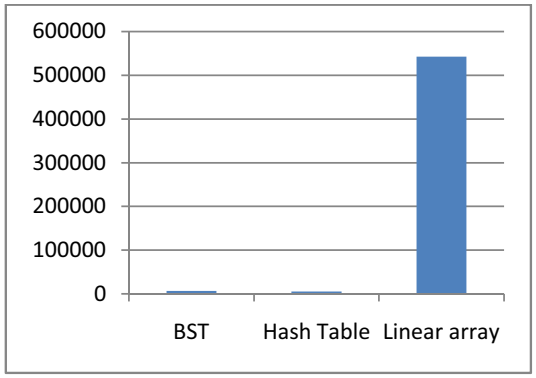
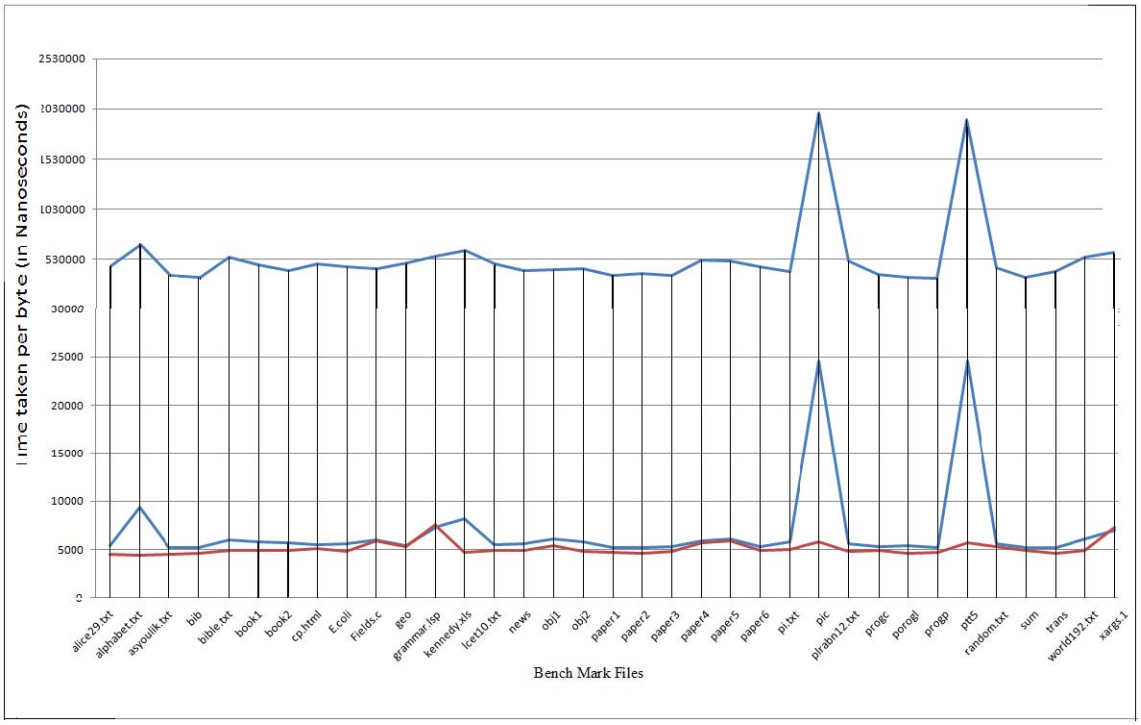


Figure -6 Comparative analysis of LZW and LZW with various data structures

By evaluating the experimental result and the above graphs of LZW data structure implementation, the LZW hash table implementation give better performance than BST and Linear array implementation. The time taken per byte is calculated using the formula given below

$$Time\ taken\ per\ byte = \frac{total\ time\ took\ to\ compress\ or\ de\ compress}{the\ size\ of\ the\ file}$$

Based on the above formula the graph-in figure-6 is plotted. When comprising the performance of data structures employed in LZW Chained hash table implementation of MDLZW achieves 98.72433% and 99.05715% improvement when comparing with linear array and BST respectively.(The comparative analysis with various data chart-1) for the experimentation various bench mark text files are used. Theorem 1, Theorem 3 and Theorem 4 shows the computational cost of LZW with linear array with linear search, BST and Chained hash table respectively. The Theorem 2 shows the computational complexity of LZW decoding algorithm the theorem 4 proves that the LZW decompression algorithm with Linear array is almost give minimal computational cost or it doesn't need any further optimization in order to reduce the computational cost.



Graph -2 comparative analysis of various Data structure implementation of LZW using famous bench mark text files

VIII. Conclusion and Future Enhancement

LZW (Lempel Ziv Welch) compression is the first widely used universal data compression method on computers. After the invention of LZW there are lots of improvements and enhancement done on LZW in order to improve the compression ratio and reduce the computational complexity. This paper experiments various data structure implementation of LZW in both theoretical and Empirical manner. For the implementation and testing various data structures are used, namely linear array, BST and Chained hash table. When comparing the performance of the above data structure, the LZW chained hash table implementation (for Encoding) gives better result than Linear array and BST data structure implementation, but when comparing the LZW decoding algorithm the Linear array is better than Linear array and BST implementation and this Linear array implementation of Linear array LZW decoding algorithm almost gives minimal computational cost or it doesn't need any further optimization in order to reduce the computational cost. This work can be further enhanced and expanded for the authentication of compression techniques to obtain optimum accuracy in time.

Reference

[1] S. Kwong and Y. F. Ho "A statistical Lempel-Ziv compression algorithm for personal digital assistant(PDA)" IEEE Transactions on Consumer Electronics, Vol. 47, No. 1, FEBRUARY 2001 pp 154 161

[2] Nan Zhang, Tao Tao, Ravi Vijaya Satya, and Amar Mukherjee "Modified LZW Algorithm for Efficient Compressed Text Retrieval" Proceedings of the International Conference on Information Technology: Coding and Computing -2004

[3] Perapong Vichitkraivin and Orachat Chitsobhuk An Improvement of PDLZW implementation with a Modified WSC Updating Technique on FPGA World Academy of Science, Engineering and Technology 36 2009 pp 611 -615

[4] Ming-Bo Lin Jang-Feng Lee, and Gene Eu Jan "A Lossless Data Compression and Decompression Algorithm and Its Hardware Architecture" – IEEE transactions on very large scale integration (vlsi) systems, vol. 14, no. 9, September 2006 pp 925-936

[5] Nirali Thakkar and Malay Bhatt Two-Stage Algorithm for Data Compression Proc. of the Intl. Conf. on Advances in Computer, Electronics and Electrical Engineering 2012 ISBN: 978-981-07-1847-3 pp 350-354

[6] NiraliThakkar and Malay Bhatt Cascading of the PDLZW Compression Algorithm with Arithmetic Coding International Journal of Computer Applications (0975 – 8887)Volume 46– No.16, May 2012 pp 21-24

[7] Liu Feng and Gao Dong-Mei DNA Algorithm of Verifiable Secret Sharing ETP International Conference on Future Computer and Communication -2009 pp 244-246

[8] CUI Wei and WU Silling An improved LZW data Compression Algorithm and its VLSI implementation Chinese Journal of Electronics Vol. 17, No 2 Apr 2008 pp 320- 324

[12] Saud NAQVI, Rameez NAQVI, Raja Ali RIAZ1and Faisal SIDDIQUI1 Optimized RTL design and implementation of LZW algorithm for high bandwidth applications PRZEGLĄD ELEKTROTECHNICZNY (Electrical Review), ISSN 0033-2097, R. 87 NR 4/2011 pp 279-285

[9] Pagiamtzis, K. Sheikholeslami, A. A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme Solid-State Circuits, IEEE Journal of (Volume:39 , Issue: 9) Page(s): 1512 - 1519 Sept. 2004 Page(s): 1512 – 1519

[10] Rupak Samanta and Rabi. N. Mahapatra "An Enhanced CAM Architecture to Accelerate LZW Compression Algorithm" 20th International Conference on VLSI Design (VLSID'07) – 2007

[11] Nishad PM and R.Manicka Chezian "Enhanced LZW (Lempel-Ziv-Welch) Algorithm by Binary Search with Multiple Dictionary to Reduce Time Complexity for Dictionary Creation in Encoding and Decoding" - International Journal of Advanced Research in Computer Science and Software Engineering - Volume 2, Issue 3, March 2012 pp192-198