Computational Complexity of Multiple Dictionary Lempel Ziv Welch (MDLZW) Algorithm and its Data Structure Implementations

Nishad, P.M. and Dr. R. Manicka Chezian

LZW is a universal lossless data compression algorithm which takes linear time in encoding and decoding. In order to reduce the computational cost, a simplified LZW architecture is designed which is called as MDLZW (Multiple dictionaries LZW) architecture. LZW dictionary constructions with any data structure take huge time. This architecture reduces the computational cost of LZW algorithm. This paper propose and experiments various data structures in use with MDLZW such as linear array with Linear Search, Binary Search Tree (BST), and Chained Hash table algorithms in MDLZW architecture. The mathematical as well as practical nature of the proposed MDLZW is examined in this paper. The enhanced architecture of LZW shows less computational complexity than the traditional LZW with any existing data structures. Linear array with linear search on MDLZW achieves 23% improvement, BST MDLZW achieves 38.24% improvement when comparing with traditional LZW BST implementation, and Chained hash table implementation of MDLZW achieves 27.36% improvements

Conclusion: the experimental result shows that the MDLZW give better performance when comparing with traditional LZW implementation, and the performance Data structure employed with the MDLZW is also optimized.

© 2014 AENSI Publisher All rights reserved


INTRODUCTION

The major feature of conventional implementations of the LZW data compression algorithms is that they usually use only single dictionary, a quite lot of compression time is wasted in searching the large-address-space dictionary instead of using a unique single dictionary a multiple dictionary set containing several small address space dictionaries with increasing the pattern widths is used for the compression algorithm. The results show that the new architecture not only can be easily implemented in, also has reduced computation cost since it no longer needs to search the dictionary recursively as the conventional implementations do. In order to reduce the computational cost, a simplified LZW architecture is designed is called MDLZW (Multiple dictionary LZW) architecture. This architecture improves and modifies the features of both LZW algorithms in the following ways. First, calculate the width of each pattern to be lookup from the dictionary and select the dictionary based on the length of the pattern. For example if the width of the pattern is two then the second dictionary is opted for the search, if the width is three then the third dictionary. In case if the dictionary does not have such pattern then the pattern is updated in the same dictionary. Finally each single dictionary has uniform size of patterns. For example the dictionary used in MDLZW compression algorithm is one that consists of m small variable-pattern width dictionaries, numbered from 1 to m, with each of which increases its pattern width by one byte. That is to say, dictionary one has one only single width pattern, dictionary two has only dual width pattern, and so on. These dictionaries: constitute a dictionary set. In general, different address space distributions of the dictionary set will present significantly distinct performance of the MDLZW compression algorithm. However, the optimal distribution is strongly dependent on the actual input data files. This paper is mainly focus on how the important data structures and algorithm like Linear array, Binary search tree, Hash table and Binary insertion sort can play important role in the implementation of MDLZW approach, and how MD implementation of LZW can give optimal reduction of computational cost using the above data structure or algorithms. All four types of MDLZW implementation is done and both theoretically and practical evaluation is done.

Corresponding Author: Nishad, P.M., Ph.D scholar, Dr. Mahalingam for research and development, Department of Computer Science, NGM College Bharathiar University, Pollachi, India-642001. Ph: (+91 9495262832) E-mail: nishadpalakka@yahoo.co.in
Preface:
In order to improve the performance of dictionary based algorithm, two methods are suggested (S. Kwong 2011). 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 (Nan Zhang 2004) 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 (Perapong Vichitkraivin 2009) 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) (Ming-Bo 2006) and an approximated adaptive algorithm. Another two stage compression algorithm combines the features of PDLZW and Arithmetic Coding (Nirali Thakkar 2012) (NiraliThakkar 2012). 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) (Liu Feng 2009). 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 (CUI Wei 2008) 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 (Saud NAQVI 2011) 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) (Pagiamtzis 2004) cells to accelerate the implementation of LZW (Rupak Samanta 2007).

Definitions:
Let us assume that $X = \{x_1, x_2, ... x_N\}$ the input sequence. The sequence length is denoted $|X|$. $D$ is the dictionary; number of dictionaries is denoted by $m$, i.e. 1 to $m$. The length of each dictionary $|D_i| = n_i$ where $i = 1, 2, ..., M$ and $\sum_{i=1}^{m} n_i = N$ or $\sum_{i=1}^{m} |D_i| = N$. $\phi$ Indicate empty or null dictionary initially each $|D_i| = \phi$, and $m$ is the number of non empty dictionary

```
STRING = get input character
WHILE there are still input characters
{
  CHAR = get input character
  L = Length (STRING + CHARACTER)
  Act = SEARCH (STRING + CHARACTER, L); // using appropriate data structure and search in the $L^{th}$ dictionary
  IF Act equal to true then
  {
    STRING = STRING + CHAR
  }
  ELSE
  {
    Output the code for STRING
    Add STRING + CHAR to the string to $L^{th}$ dictionary// use appropriate data structure
    STRING = CHAR
  }
}
```

Fig. 1: Multiple Dictionaries LZW (MDLZW) Algorithm.
The MDLZW algorithm has two phases i) Switching and coding phase and ii) searching and updating phase. For encoding initially the initial dictionary or dictionary –zero $D_0$ contains all the possible patterns in ascending order (0 to 255), this dictionary is a virtual directory so no insertion is made in the initial dictionary while encoding (compression) this dictionary is shown in the figure-2, figure-3 and figure-4 using dashed lines, this dictionary is called virtual dictionary (no data structure operation is made in such dictionaries or in other words no data structure operation are made in the virtual directory or its imaginary). The first phase of the encoding algorithm uses three variables STRING, CHAR and L and also single dictionary is replaced with multiple dictionaries. The L has an integer value to identify the dictionary to be searched based on the pattern “STRING+CHAR”, CHAR variable holds only a single character. The variable STRING is a variable length string (i.e.) it may have a group of one or more character with each character being a single byte. Initially the encoding algorithm initializes the variable L is 2 and then 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. Each time after storing byte to CHAR the switching is processed based on the L to determine if the concatenation of the two variables, STRING+CHAR, has already been assigned a code for example initially L is 2 and STRING is “A” and CHAR is “B” so the search for the pattern STRING+CHAR is made only within the dictionary -1 $D_1$ and the comparison required is zero (shown in figure-2) because initially the length of all dictionary is zero or $D_i = \phi$ where $i = 1 \text{ to } M$ and $\phi$ indicates the dictionary is empty. If the pattern STRING+CHAR is “ABBABB” number of comparison required is zero, but the algorithm use single dictionary the comparison required using single dictionary is five. So using the multiple dictionaries and the linear array with search the computation cost is reduced so this directly leads to reduce the time complexity. If the pattern STRING+CHAR search result is true then the STRING+CHAR is assign to STRING and the L variable is incremented by one and CHAR is the next character in the input sequence of X the Length of the pattern increases then based on the L the search process again continues in the very next dictionary, this process continues the search fails or until the longest match found if the longest mach found then the index is given as the output string. After the unsuccessful search CHAR is stored in to the STRING and the L is reset to two and the looping is continued. For the searching and updating the dictionary is the second phase of the algorithm is used each time after switching the dictionary is selected for the search base on the L. Linear search is used for searching if the pattern STRING+CHAR is reported in the corresponding dictionary then the function return true else the pattern is updated as the last element in which the search failed. The algorithm is shown in the figure-1.

**MDLZW Linear Array Implementation:**

The length of the input stream X is $|X|$, initially the number of dictionaries is M. where M is rebuild based on the length of the pattern and in the beginning all dictionary’s are empty, i.e., $D_i = \phi$ where $i = 1 \text{ to } M$, except virtual dictionary, but during the computational cost evaluation this dictionary is not considered because of no search or comparison is made in the virtual dictionary $D_0$. After each unsuccessful search each dictionary is updated with STRING+CHAR, the dictionary is represented using the linear array shown in the Figure -2 and for the lookup operations the linear search is used, so by performing the update in the dictionary that corresponding dictionary length is incremented by one, after the last update in the dictionary the length of the dictionary is represented by $|D_i| = n_i$ where $i = 1 \text{ to } m$, $m$ indicates non empty dictionaries in $D_1 \text{ to } D_m$, during the complexity study only up to $m$ is considered. $N = \sum_{i=1}^{m} n_i$, i.e., the total number of elements in all dictionaries $D_1 \text{ to } D_m$. For switching search between the dictionaries the STRING+CHAR length L is used.

---

**Fig. 2**: MDLZW Linear array with linear search implementation for the input sequence “ABCABBABBBCABABBBD”
Theorem 1:
The Linear array with linear search implementation of MDLZW compression algorithm takes.

\[ |X| \left( \frac{|D| + m \cdot 3}{m \cdot |D| + 4} \right) \text{time} \]

Proof: The computational cost per dictionaries is calculated by \( \frac{1}{m} \sum_{i=1}^{n} \left( \frac{i+1}{2} \right) \), so the average number of comparison required for the input sequence is calculated as follows.

\[
|X| \left( \frac{1}{|D|} \left( \frac{1}{m} \left( \sum_{i=1}^{n_1} \frac{i+1}{2} \right) + \left( \frac{1}{n_2} \sum_{i=1}^{n_2} \frac{i+1}{2} \right) + \left( \frac{1}{n_3} \sum_{i=1}^{n_3} \frac{i+1}{2} \right) + \ldots \right) \right)
\]

Where

\[
\frac{1}{n_i} \sum_{i=1}^{n_i} \frac{i+1}{2} = \frac{n_i + 3}{4}
\]

Then equation (1) is simplified as

\[
= |X| \left( \frac{1}{|D|} \left( \frac{1}{m} \sum_{i=1}^{n} \frac{n_i + 3}{4} \right) \right) \ldots \ldots (2)
\]

Where

\[
\sum_{i=1}^{m} n_i = |D|
\]

Then the equation (2) is simplified as

\[
= |X| \left( \frac{1}{|D|} \left( \frac{1}{m} \left( \frac{|D| + (m \cdot 3)}{4} \right) \right) \right)
\]

\[
= |X| \left( \frac{1}{(m \cdot |D|)} \left( \frac{|D| + (m \cdot 3)}{4} \right) \right) \ldots \ldots (3)
\]

By simplifying the above equation,

\[
= |X| \left( \frac{1 \cdot \left( \frac{|D| + (m \cdot 3)}{m \cdot |D| + 4} \right)}{m \cdot |D| + 4} \right)
\]

\[
= |X| \left( \frac{|D| + (m \cdot 3)}{m \cdot |D| + 4} \right)
\]

Mdlzw with Binary Search Tree Implementation:
The length of the input stream \( X \) is \( |X| \), initially the number of dictionaries is \( M \), where \( M \) is rebuilt based on the length of the pattern and in the beginning all dictionary’s are empty, i.e., \( D_i = \Phi \) where \( i = 1 \) to \( M \), except virtual dictionary, but during the computational cost evaluation this dictionary is not considered because of no search or comparison is made in the virtual BST (dictionary \( D_0 \)). After each unsuccessful search in the dictionary constructed using separate BST shown in the figure -5 , the BST insertion operation is made with a new element or STRING+CHAR, so by performing the insertion, that corresponding dictionary the number of nodes is incremented by one (each BST represents the unique dictionary), after the last insertion in the BST the total number of elements in each BST represented by \( |D_i| = n_i \), where \( i = 1 \) to \( M \), \( m \) indicates number of non empty trees in \( D_1 - D_M \), during the complexity study only up to \( m \) is considered. \( N = \sum_{i=1}^{m} n_i \), i.e., the total number of elements in all dictionaries \( D_1 - D_m \). For each STRING+CHAR insertion or a search or both will take place in the corresponding BST. For switching search between the dictionaries the STRING+CHAR length \( L \) is used.
Fig. 4: Binary Search Tree MDLZW implementation for the input sequence “ABCABBABBBCABABBBD”

**Theorem 2:**

The Binary search tree implementation of MDLZW compression algorithm takes

\[ T = O(|X|) \sum_{i=1}^{m} \log \left( n_i \frac{1}{|D|} \right) \] time

**Proof:** The computational cost per dictionaries is calculated by \( \log \prod_{i=1}^{n} \frac{1}{|D|} \). so the average number of comparison required for the input sequence is calculated as follows.

\[ = O |X| \left( \frac{1}{|D|} \right) \left( \frac{1}{m} \right) \left( \log \prod_{i=1}^{n_1} i \right) + \left( \log \prod_{i=1}^{n_2} i \right) + \left( \log \prod_{i=1}^{n_3} i \right) + \ldots \]

\[ = O |X| \left( \frac{1}{|D|} \right) \left( \frac{1}{m} \right) \left( \log \prod_{i=1}^{n_1} i \right) + \left( \log \prod_{i=1}^{n_2} i \right) + \left( \log \prod_{i=1}^{n_3} i \right) + \ldots \]

\[ = O |X| \left( \frac{1}{|D|} \right) \left( \frac{1}{m} \right) \sum_{i=1}^{m} \left( \log \prod_{j=1}^{n_i} j \right) \]
\[ = O \left| X \right| \left( \sum_{i=1}^{m} \left( \log \prod_{j=1}^{n_{i}} j \right) \right) \frac{1}{(|D|+m)} \]  

Where

\[ \prod_{j=1}^{n_{i}} j = n_{i}! \]

Then the equation (4) is simplified as

\[ = O \left| X \right| \left( \sum_{i=1}^{m} \log \left( \frac{n_{i}!}{n_{i}^{1}} \right) \right) \frac{1}{(|D|+m)} \]

**MdLzw with Chained Hash Table Implementation:**

The length of the input stream X is |X|, initially the number of dictionaries is M, where M is rebuild based on the length of the pattern and in the beginning all dictionary’s are empty, i.e., \( D_{i} = \Phi \) where \( i \) is 1 to M, except virtual dictionary, but during the computational cost evaluation this dictionary is not considered because of no search or comparison is made in the virtual Chained hash table (dictionary \( D_{0} \)). After each unsuccessful search in the dictionary constructed using chained hash table shown in the figure -5, the chained hash table is updated with a new element or STRING+CHAR, so by performing the insertion, that corresponding dictionary the number of nodes is incremented by one (each Hash table represents the unique dictionary), after the last insertion in the hash table the total number of elements in each hash table represented by \( |D_{i}| = n_{i} \) where \( i \) is 1 to m, \( m \) indicates number of non empty trees in \( D_{1} - D_{M} \), during the complexity study only up to m is considered. \( N = \sum_{i=1}^{m} n_{i} \), i.e., the total number of elements in all dictionaries \( D_{1} - D_{M} \). For switching search between the dictionaries the STRING+CHAR length \( L \) is used.

**Fig. 4:** Chained Hash Table MDLZW implementation for the input sequence “ABCABBABBCABABBBD”

**Theorem 3:**

The Chained hash table implementation of MDLZW compression algorithm takes

\[ O \left( \sum_{j=1}^{M} \left( 1 + AM(a_{j}) \right) \right) \frac{1}{(N+M)} \]  

**Proof:** The computational cost per dictionaries is calculated by \( \frac{n+\sum_{i=1}^{n} n_{i}}{m} \) so the average number of comparison required for the input sequence is calculated as follows.
\[
1 \left( \frac{1}{M} \left( \left| n_1 \right| + \sum_{i=1}^{\left| n_1 \right|} a_i \right) + \left| n_2 \right| + \sum_{i=1}^{\left| n_2 \right|} a_i \right) + \cdots + \left| n_m \right| + \sum_{i=1}^{\left| n_m \right|} a_i \right) \right)
\]

By simplifying the above equation we get

\[
\frac{1}{N} \left( \frac{1}{M} \left( \sum_{j=1}^{M} \left| n_j \right| + \sum_{i=1}^{\left| n_j \right|} a_i \right) \right) \right)
\]

\[
= \frac{1}{N} \left( \sum_{j=1}^{M} \left( \left| n_j \right| + \sum_{i=1}^{\left| n_j \right|} a_i \right) \right) \right)
\]

\[
= \left( \sum_{j=1}^{M} \left( \frac{\left| n_j \right| + \sum_{i=1}^{\left| n_j \right|} a_i}{\left| n_j \right|} \right) \right)^{\frac{1}{1+AM}} \cdots \cdots \cdots (5)
\]

Where

\[
\frac{\left| n_1 \right| + \sum_{i=1}^{\left| n_1 \right|} a_i}{\left| n_1 \right|} = 1 + AM \{a_j\} \text{ then}
\]

Where AM is arithmetic mean

Then the equitation (5) is simplified as

\[
O \left( \sum_{j=1}^{M} \left( 1 + AM \{a_j\} \right) \right)^{\frac{1}{1+AM}}
\]

**Graph 1:** Comparative analysis of LZW and MDLZW with various data structures.
Graph 2: Comparative analysis of various Data structure implementation of MBLZW using famous bench mark text.

Experimental Result:

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 was Windows XP SP3 (32 bit). All programs were compiled using java version jdk1.6.0_13. The times were recorded in nanoseconds. The time taken of each algorithm is calculated using the tool compression time complexity evaluator; graphs are plotted with MS Excel. All data structures reside in main memory during computation.

By evaluating the experimental result and the above graphs the MDLZW is reduced computational cost when comparing to the traditional LZW and its data structure implementation. Among the MDLZW data structure experimentation Chained hash table implementation shows best result. The time taken per byte is calculated using the formula given below.

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

Based on the above formula the graph-2 is plotted. When comprising the performance of data structures employed in MDLZW and traditional LZW data structures the MDLZW achieves best result, linear array with linear search on MDLZW achieves 23%, linear array with BST MDLZW achieves 38.24% and Chained hash table implementation of MDLZW achieves 27.36% ( the comparative analysis with various data chart-1) improvements when comparing to the traditional LZW, so the MDLZW approach is best when comparing with the original LZW data compression algorithm, for the experimentation various bench mark text files are used. Theorem -1, Theorem -2 and Theorem -3 shows the computational cost of MDLZW with linear array with linear search, BST and Chained hash table respectively.

Conclusion and Future Enhancement:

The major feature of conventional implementations of the LZW data compression algorithms is that they usually use only single dictionary. This takes linear time in encoding. The MDLZW architecture improves and reduces the computational cost of LZW algorithm. This paper experiments various data structures and algorithms used with the MDLZW architecture, namely the data structures used for the experimentation is Linear array with Linear Search, Binary Search Tree (BST), and Chained Hash table. The experimental results show the MDLZW architecture is best with any data structure and algorithm when comparing to the simple LZW architecture. Linear array with linear search on MDLZW achieves 23%, linear array with BST MDLZW achieves 38.24% and Chained hash table implementation of MDLZW achieves 27.36% improvements. This work can be further enhanced and expanded for the authentication of compression techniques to obtain optimum accuracy in time.
REFERENCE


