International Journal of Computer Applications (0975 – 8887)
Volume 41– No.15, March 2012

Implementing a Novel Data Structure for Maintaining Cumulative Frequency of Symbols

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

A new data structure, namely “cumulative frequency matrix (CFM)”, is proposed here for maintaining cumulative frequencies. For an order-0 model having 256 symbols, CFM is a 2-D array of 16 rows and 16 columns. Two nibbles, say L for left and R for right, of a byte symbol represents row and column dimensions respectively. Matrix element (L, R) represents cumulative frequency of symbol with right nibble as R among symbols with left nibble as L. Within row, it stores cumulative frequency of symbols with right nibble varying from 0 to 15. Adaptive arithmetic coding is a lossless data compression method. It needs to update cumulative frequencies at runtime. Various algorithms for maintaining cumulative frequencies, computing cumulative frequency interval etc. are discussed here. Practical implementation shows that proposed data structure is simpler as well as efficient as compared to other data structures in use.

General Terms

Data Compression, Data Structures, Algorithms, Adaptive Arithmetic Coding, Cumulative Frequencies

Keywords

Adaptive arithmetic coding, data compression, cumulative frequency maintenance, data structure, algorithm

1. INTRODUCTION

Arithmetic coding is a lossless coding method that depends on the probabilities of the symbols. In arithmetic coding, cumulative probabilities of symbols are used to compute subintervals in the range [0, 1). In implementations [1,6,8,12-15] with integer arithmetic, cumulative frequencies are used instead of cumulative probabilities. Algorithms for adaptive arithmetic coding need building statistical data model of cumulative frequencies on fly. Thus data structure for maintaining cumulative frequencies has a very potential role to play for algorithms to be efficient.

Shannon theorem [5] guarantees that compression below the entropy of the source is impossible. One can remove as much redundancy from one’s data as one likes, but entropy is proven to be a hard limit. However one can optimize one’s algorithms in at least two dimensions: memory usage and speed [6]. Here an attempt is made to improve the speed of algorithms by suggesting a novel data structure; 2-D matrix.

Adaptive data compression model of arithmetic coding is a single pass model. It needs an ability to adjust the data model “on the fly” to the spatially varying statistical nature of data contents [10]. It uses estimation of probabilities of source symbols [17] and adapts statistical cumulative frequency data model dynamically as the symbol is read from file and encoded. During decoding, decoder creates the same model as the encoder. Initially both encoder and decoder assume uniform distribution of all possible symbols in an alphabet.

It has been proven [11] that the compression efficiency of arithmetic coding with an adaptive model is never significantly inferior to arithmetic coding with the exact data model. As compared to static model, an additional cost in adaptive arithmetic data compression is a task of maintaining cumulative frequencies dynamically while encoding and decoding symbol read from file. Other tasks like computing subintervals while encoding-decoding and searching for interval segment when decoding remains same.

Speed of adaptive arithmetic coding is strongly affected by how quickly the cumulative counts can be calculated, so it is important to use a data structure that makes it easy to compute cumulative count for a symbol and easy to search the cumulative counts for a target [10].

2. RELATED DATA STRUCTURES

It is found that various data structures are used to organize cumulative frequencies for efficient storage and retrieval.

Alexey [7] says that in the practical implementations seen so far, the cumulative symbol frequency table is represented by an array of integer numbers containing the sum of frequencies of all the symbols having indices less than given symbol. The array is organized in sorted order by value of symbols or by (non-cumulative) frequencies of symbols. With adaptive arithmetic coding, encoding a regular symbol needs corresponding frequency interval to be determined; decoding needs determining a frequency interval containing a given point. Both actions are followed by updating the adaptive model, what's commonly done by updating frequency or cumulative frequency of symbols.

Various data structures used to organize cumulative frequency information are discussed here.

2.1 Linear 1-D array

The simplest data structure is an array of cumulative frequencies arranged in the order of symbols in an alphabet. Computing cumulative frequency interval is very fast, O(1). Determining a frequency interval containing a given point can be done fast enough by binary search in the array. But updating adaptive model requires a considerable number of increments to perform. For symbol s, it needs to increment cumulative frequency of all symbols from s onwards. Thus it is having linear time complexity of O(|A|) per symbol processing. Here |A| represents the size of an alphabet. In order-0 model, |A| is 256.

2.2 MTF array, Moffat's HEAP, Splay tree

Witten, Neal and Cleary [1] have suggested move-to-front (MTF) method. List, stored in 1-D array, is reorganized by moving an element to the head of the list every time a symbol is processed. List may not remain in sorted order of symbols, so it requires storing index of symbols in an array in addition to an array used to store frequency. It needs to use 2|A| words to store index of symbols and frequency of symbols. It also has linear space and time complexity of O(|A|). Sorting the array by frequencies gives effect only for small arrays and extremely non-uniform distribution of symbols [7].

Moffat [2] described a heap-like binary tree structure. It requires reorganizing tree to satisfy HEAP properties. Here it maintains index of symbols in the nodes of the tree. Each leaf node contains frequency of individual symbol and intermediate nodes contain weight (total frequency of nodes in its subtree). Here time complexity is $O(\log_2|A|)$ and space requirement is 2A words.

Jones [3] uses splay trees for handling cumulative frequencies. A splay tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. Accessing a node requires a special rotating step called splaying that moves recently accessed node to the root and make the tree "more balanced". It performs basic operations such as insertion, look-up and removal in $O(\log_2|A|)$ amortized time. Like MTF, splay-tree data structure is more useful when the symbols are repeating in sequence.

In all these MTF, HEAP and Splay-tree techniques, the attempt is made to keep most frequent symbols in quickly-referenced positions. It requires extensive data reorganization in the data structure to move most frequent symbol in head position. These methods work well for highly skewed distribution of symbols but are less efficient for more uniform distributions.

2.3 Heap as described by Solomon

Solomon [9] describes a heap-like binary tree structure. Solomon's HEAP data structure uses nodes with three elements (symbol, frequency, total frequency in left subtree). Heap property is to be fulfilled using frequency element. Obviously HEAP is housed in 1-D array of these nodes. When a symbol is read, it needs to reorganize Heap to keep the array in sorted order of frequency of symbol. It also requires computing total frequency of symbols in left subtree for each symbol being processed. Worst case number of iterations is $O(\log_2|A|)$ but each iteration is too heavy in computation.

2.4 Binary Indexed Tree (BIT)

Fenwick [4] presented a binary-indexed-tree (BIT) data structure housed in 1-D array that stores partial cumulative frequencies of symbols. BIT is an elegant data structure for less skewed distribution. Here each node contains the cumulative sum of specific range of counts. BIT preserves symbol ordering; i.e. information for symbol s is stored at index s+1. It does not need to reorganize tree.

Elements of an array in BIT are stored as follows: Considering symbols starting with value 0 and index from1, it stores actual frequencies of symbols at odd index positions (2i-1). At even index, it stores partial sum of frequencies of some specific number of previous symbols. At index j = 4i-2 (i.e. at 2,6,10,…), it stores sum of the frequency of two immediately preceding symbols; at j = 8i-4 (i.e. at 4,12,20,…), the value stored is the sum of the frequency of immediately preceding four symbols; at j = 16i–8 (i.e. at 8,24,40,…), the value stored is the sum of the frequency of immediately preceding eight symbols; and so on [4, 16]. Table 1(a) gives a sample data considering 16 symbols in an alphabet for easy understanding of tree organization. SumN denotes sum of frequency of N preceding symbols. Tree is an array having partial cumulative frequency of symbols. HghCF is upper bound (high count) of cumulative frequency interval of symbol. Note that only tree array is a data structure used in algorithm. Other rows are given for better understanding only.

Table 1(a)
Frequency, Cumulative frequency (CF), partial cumulative frequency (Tree) information

index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
freq	1	3	5	2	8	3	2	5	7	0	0	2	3	0	0	2
Sum1	1		5		8		2		7		0		3		0	
Sum2		4				11				7				3		
Sum4				11								9				
Sum8								29								
Sum16																43
Tree	1	4	5	11	8	11	2	29	7	7	0	9	3	3	0	43
HghCF	1	4	9	11	19	22	24	29	36	36	36	38	41	41	41	43

Table 1(b)
Symbols involved in partial CF at even index

index	2	4	6	8	10	12	14	16
symbols involved	1..2	1..4	5..6	1..8	9..10	9..12	13..14	1..16

Using BIT, cumulative frequency counts can be updated in logarithmic time by using the following algorithm. Suppose that s is some symbol number in the range $1 \leq s \leq n$. For example, updating frequency at position 3 needs to update frequencies at index 4, 8, 16 and so on. Similarly computing cumulative frequency of symbol needs to add some partial cumulative frequencies. For example, to compute cumulative frequency of symbol at position 13, it needs to add elements at index 13, 12 and 8. For these operations, to get next index to operate upon, Fenwick has defined backward(s) and forward(s) to be used while computing cumulative frequencies and updating array elements respectively. Function backward(s) computes next index in backward direction, it is an integer obtained from s by subtracting the binary number corresponding to the rightmost one-bit in s; i.e. by converting right-most occurrence of bit 1 to 0. For example, the binary representation of 13 is 1101, so backward(13) = 12 (1100 in binary); backward(12) = 8 (1000 in binary); and backward(8) = 0. Similarly, forward(s) computes next index in forward direction, it is an integer to be at $s + 2^i$ where i is again the position of the rightmost one-bit in s. For example, forward(13) = 14, forward(14) = 16, and forward(16) = 32. Both backward and forward can be readily implemented using bitwise operations if integers are represented in two's-complement form: backward(i) as either "i - (i AND - i)" or "i AND (i-1)"; and forward(i) as "i + (i AND - i)", where AND is a bitwise logical "and" operator. If one does not want to depend on 2's complement arithmetic, one can compute next index based on explanation given in previous paragraph. It is also available at codeguru [7].

The advantage of using BIT over above methods is that it is not required to reorganize the data structure and it provides logarithmic access with time complexity O($\log_2|A|$) per symbol. It uses only |A| = 256 words to store cumulative frequencies. It is shown by Fenwick [4] that BIT is more efficient than MTF, HEAP and Splay tree.

3. PROPOSED DATA STRUCTURE

In this paper, a new data structure is proposed for efficient management of cumulative frequency information. It is Cumulative Frequency Matrix (CFM) that stores partial cumulative frequency of 256 symbols of an order-0 alphabet. In order-0 model, a single byte symbol takes values from 0 to 255. Left nibble and right nibble, each of 4 bits, are used as two dimensions for row and column respectively. Both nibbles take values from 0 to 15. Thus CFM data structure uses 16 rows and 16 columns in a matrix.

CFM stores partial cumulative frequencies, i.e. cumulative frequency of symbols within a specific row only. Matrix element (L, R) represents the partial cumulative frequency of symbol within all symbols in row L; here L is the left nibble and R is the right nibble of a symbol. Obviously last element in each row is the total number of symbols occurred in that row; i.e. total symbols with left nibble L in row L. All elements in a row are in sorted order as they represent cumulative frequencies within row. This property helps to make search faster using bisection method.

4. CONCEPT USING 16 SYMBOLS

To understand the concept, consider 16 symbols in an alphabet. Each symbol is of 4 bits. Let most significant two bits be represented as L and least significant two bits as R. For example, in binary number $(1011)_2$, L is $(10)_2$ and R is $(11)_2$.

Here algorithms are considered for various operations on cumulative frequencies as needed in adaptive arithmetic data compression. These operations include: (1) initialize data structure (2) update statistical data model, i.e. adapting cumulative frequencies (3) compute cumulative frequency interval for a given symbol; to be used when encoding (4) find cumulative frequency interval for a given target; to be used when decoding.

4.1 Initializing CFM Data Structure:

Initially all symbols are assumed to have uniform distribution with frequency 1. Initial CFM is as shown in table 2(b) with all rows {1,2,3,4}. Note that frequency matrix in table 2(a) is given just for better understanding only. It is not required by any algorithm. Algorithm for initializing CFM is given in section 5.1.

Table 2(a) Initial Frequency

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	1	1	1
$(01)_2$	1	1	1	1
$(10)_2$	1	1	1	1
$(11)_2$	1	1	1	1

Table 2(b)Initial Cumu.Freq.

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	2	3	4
$(01)_2$	1	2	3	4
$(10)_2$	1	2	3	4
$(11)_2$	1	2	3	4

4.2 Adapting Statistical Model

When a symbol is read, CFM needs to be updated. As CFM stores cumulative frequency of symbols within row, it simply requires to increment elements from column R = rightNibble onwards in row L = leftNibble. Note that it has no effect on elements in other rows. For example, consider occurrences of sequence of binary symbols 0010, 0101, 0000, 1111. See an effect on cumulative frequency matrix in Table 3 to 6. Algorithm for updating CFM is given in section 5.2.

Table 3(a) Frequency after reading 0010_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	1	**2**	1
$(01)_2$	1	1	1	1
$(10)_2$	1	1	1	1
$(11)_2$	1	1	1	1

Table 3(b) Cumu. Freq. after reading 0010_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	2	**4**	**5**
$(01)_2$	1	2	3	4
$(10)_2$	1	2	3	4
$(11)_2$	1	2	3	4

Table 4(a) Frequency after reading 0101_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	1	2	1
$(01)_2$	1	**2**	1	1
$(10)_2$	1	1	1	1
$(11)_2$	1	1	1	1

Table 4(b) Cumu. Freq. after reading 0101_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	1	2	4	5
$(01)_2$	1	**3**	**4**	**5**
$(10)_2$	1	2	3	4
$(11)_2$	1	2	3	4

Table 5(a) Frequency after reading 0000_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	**2**	1	2	1
$(01)_2$	1	2	1	1
$(10)_2$	1	1	1	1
$(11)_2$	1	1	1	1

Table 5(b) Cumu. Freq. after reading 0000_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	**2**	**3**	**5**	**6**
$(01)_2$	1	3	4	5
$(10)_2$	1	2	3	4
$(11)_2$	1	2	3	4

Table 6(a) Frequency after reading 1111_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	2	1	2	1
$(01)_2$	1	2	1	1
$(10)_2$	1	1	1	1
$(11)_2$	1	1	1	**2**

Table 6(b) Cumu. Freq. after reading 1111_2

L\R	**$(00)_2$**	**$(01)_2$**	**$(10)_2$**	**$(11)_2$**
$(00)_2$	2	3	5	6
$(01)_2$	1	3	4	5
$(10)_2$	1	2	3	4
$(11)_2$	1	2	3	**5**

4.3 Computing lower and upper bound of cumulative frequency interval

When encoding a symbol, arithmetic coding algorithm needs to compute cumulative frequency interval corresponding to the symbol. To understand how it can be computed, consider CFM representing the symbols read at some point of time as shown in table 7(b). Table 8 represents symbols, its frequency and cumulative frequency according to data in CFM. Note that table 7(a) and table 8 are given only for better understanding and they are not required by any algorithm.

For computing interval, consider an example with symbol 9. To have lower bound, i.e. lowCount, use previous symbol 8; compute its nibbles L and R. For symbol 8, L=10 and R=00. Remember that the last column in each row contains cumulative frequency of symbols in the corresponding row. So adding elements in last column of previous rows will give cumulative frequency of all symbols whose left nibble is less than L. Here adding elements from last column in previous rows, i.e. row 00 and row 01, results in 20+25=45. Refer table 7(b). Now add cumulative frequency of symbol in current row, i.e. add element at (L,R) to get lowCount for symbol s. Here 45+4=49 is the lowCount for symbol 9. To get highCount of symbol, if R=last column then add element at (L+1, 0); otherwise add element at (L, R+1). Here it is 45+10=55. Thus a cumulative frequency interval (lowCount, HighCount] can be computed for a symbol. Refer an algorithm given in section 5.3.

Table 7(a)
Frequencies

L\R	$(00)_2$	$(01)_2$	$(10)_2$	$(11)_2$
$(00)_2$	7	3	6	4
$(01)_2$	5	4	7	9
$(10)_2$	4	6	2	3
$(11)_2$	3	4	8	5

Table 7(b)
Cumulative Frequencies

L\R	$(00)_2$	$(01)_2$	$(10)_2$	$(11)_2$
$(00)_2$	7	10	16	20
$(01)_2$	5	9	16	25
$(10)_2$	4	10	12	15
$(11)_2$	3	7	15	20

Table 8
Symbols (S), Frequencies (F), Cumu. Frequencies (CF)

S	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F	7	3	6	4	5	4	7	9	4	6	2	3	3	4	8	5
CF	7	10	16	20	25	29	36	45	49	55	57	60	63	67	75	80

4.4 Finding cumulative interval in which a given target value falls

During decoding, arithmetic coding algorithm needs to find a cumulative frequency interval within which a given target code value lies. For example, consider target value 50. For target 50, symbol to be decoded should be 9 and interval should be (49 , 55]; i.e. lowCount 49 and highCount 55.

To find cumulative frequency interval for given target, keep on adding values in last column from first row onwards till sum exceeds target. Here 20+25 = 45 < target 50, but 20+25+15 = 60 > target 50. It means that target interval is somewhere in 3rd row. Now search in this row for element greater than remaining target (here 50-45=5). In given example, it is in second column. Thus interval is between (45+4, 45+10] and corresponding symbol to be decoded is 9. An algorithm for this operation is given in section 5.4.

5. ALGORITHMS

Here algorithms are considered for performing four operations as discussed in section 4, but here total symbols to be considered are 256; size of an alphabet of order-0 model. Each symbol is of single byte taking values from 0 to 255. Two nibbles of the symbol are of 4 bits each. Values of nibble range from 0 to 15. Thus required CFM data structure is a 2-D array of 16x16=256 words that stores partial cumulative frequencies of various symbols.

5.1 Initializing Data Structure

Assuming uniform distribution of symbols initially, each row in cumulative frequency matrix is initialized with elements {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}. In table 9, first column and first row represents row (LeftNibble) and column (RightNibble) index respectively in 2-D array of CFM.

Table 9
Initial Cumulative Frequencies

L\R	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
14	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
15	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
Algorithm: Initialize CFM data structure

dimSize = 16
row = 0
While (row < dimSize)
Begin
        cf = 1
        col = 0
        While (col < dimSize)
        Begin
          CFM(row,col) = cf
          col = col + 1
          cf = cf + 1
        End
        row = row + 1
End
```

5.2 Adapting Statistical model

In adaptive arithmetic coding, after encoding symbol, cumulative frequencies are updated. Here CFM needs to be updated for encoded symbol, say s. It requires incrementing cumulative frequency elements in single row only corresponding to left nibble of symbol s. Note that use of pointer arithmetic can improve speed while accessing contiguous elements.

```
Algorithm: adapt model, i.e. update CFM

dimSize = 16
row = s/dimSize          // left nibble
col = s mod dimSize      // right nibble, remainder

While (col < dimSize)
Begin
  CFM(row,col) = CFM (row,col) + 1
End
```

5.3 Computing lower and upper bound of cumulative frequency interval

While encoding a symbol, arithmetic coding method needs computation of lowCount (lower bound) and highCount (upper bound) of cumulative frequency interval for a symbol.

```
Algorithm: compute cumulative frequency interval for a
           given symbol

dimSize = 16
LowCount = 0, lastCol = dimSize - 1
If symbol=0 then
Begin
    highCount=CFM(0,0)
    Stop
End
s = symbol-1
row = s/dimSize          // left nibble
col = s mod dimSize      // right nibble

// sum of last column from 0 to row-1
sum=0, i=0
While ( i < row)    Begin
      sum = sum + CFM(i,lastCol)
End

// add element in current row
LowCount = sum + CFM(row,col)
If (col = lastCol) then
    HighCount = sum + CFM(row+1,0)
Else HighCount = sum + CFM(row, col+1)
```

5.4 Finding cumulative interval in which a given target value falls

While decoding in arithmetic coding method, a coded value is read and it needs to find a cumulative frequency interval where this target value lies. A symbol corresponding to this interval is then considered to be a symbol to be decoded.

```
Algorithm: Find cumulative frequency interval in which
           target value lies

dimSize = 16
If target > CFM(0,0) then
Begin
  lowCount=0, highCount=CFM(0,0)
  Stop
End

//add last column of rows if sum < target
lastCol = dimSize -1, row=0, sum = CFM(0,lastCol)
While (target > sum)
Begin
   row = row + 1
   sum = sum + CFM(row,lastCol)
End
sum = sum – CFM(row,lastCol)

// search within row for first element > remaining target
// note: can use bisection search
col=0, target = target - sum
While (target > CFM(row,col))
Begin
   col = col + 1
End
highCount = sum + CFM(row,col)
If col = 0 then
  lowCount = sum
Else
  lowCount = sum + CFM(row, col-1)
```

Here, using last value of row and col, a symbol to be decoded can be computed as row*16 + col.

Note that in the second while loop, bisection search can be used to search within a row because row elements are in increasing order. Also note that search in first loop can be enhanced using an additional 1-D array of 16 elements to contain cumulative frequency of symbols with left nibble as index. But doing so will increase an overhead of updating this array also with CFM while adapting the model at the time of encoding/decoding symbols. Thus overall time of all operations combined together may not be affected.

6. PERFORMANCE ANALYSIS

Algorithms using various data structures are analyzed for time and space complexity as shown in tables 10 and 11. In an analysis, |A| is used for size of alphabet and n is used for the number of bytes in source file.

Fenwick [4] has shown that, as compared to MTF, splay and Moffat's HEAP data structure, BIT is using less space and is very efficient for all probability distributions, whether skewed or not. So these three structures are not considered while analyzing.

Data structures considered here for comparison are:
(1) Linear 1-D array (inefficient, but a basic data structure)
(2) HEAP as given by Solomon [9]
(3) Fenwick's [4] Binary Indexed Tree (BIT)
(4) Proposed Cumulative Frequency Matrix (CFM)

As seen in table 10, linear 1-D array is very inefficient having linear space and time complexity.

HEAP as suggested by Solomon [9] performs its operations in logarithmic time O(n*log2|A|), but still it may not be as efficient as BIT because of its too heavy iterations in terms of computation.

CFM performs in O(n*√|A|) time whereas BIT performs in O(n*log2|A|). For order-0 model, CFM requires 16 iterations in the worst case; whereas BIT requires 9. For example, in BIT, for first symbol, adapting cumulative frequencies require to update the values at index 1, 2, 4, 8, 16, 32, 64, 128 and 256. Worst case in CFM is a symbol with right nibble 0 and it requires 16 consecutive values to be updated in single row. But with BIT, computing next index in iterations is very complex and time consuming if they are not implemented using 1's or 2's compliment and bitwise operation. Thus practically CFM may perform with better or equal efficiency.

Space complexity analysis is given in table 11. CFM and BIT data structure have same space complexity O(|A|). It can be observed that HEAP data structure requires almost three times the space as compared to CFM and BIT.

7. EXPERIMENTAL RESULTS

C Program is developed to compare performance of algorithms for various operations considering order-0 model. Algorithms are implemented using four data structures: linear 1-D array, Solomon's HEAP, Fenwick's BIT and proposed CFM. Results of practical implementation using 1-D linear array are obviously very inefficient, so not shown here.

Table 10.Time Complexity Analysis

Operation	**Linear 1-D**	**HEAP [Solomon]**	**BIT [Fenwick]**	**CFM [Proposed]**
Initialize data structure	O(|A|)	O(|A|) for symbol, O(|A|) for frequency, O(|A|) for total frequency in left subtree	O(|A|)	O(|A|)
Adapting model when a symbol is read	O(n*|A|)	Searching farthest node: $O(n*\log_2\|A\|)$, Updating total frequency in left subtree: $O(n*\log_2\|A\|)$	$O(n*\log_2\|A\|)$	O(n*√|A|)
Compute cumulative frequency interval for a given symbol	O(n)	$O(n*\log_2\|A\|)$	$O(n*\log_2\|A\|)$	O(n*√|A|)
Find cumulative frequency interval in which target value falls	$O(n*\log_2\|A\|)$ using bisection search	$O(n*\log_2\|A\|)$	$O(n*\log_2\|A\|)$	within last column: O(n*√|A|), within row $O(n*\log_2 \sqrt{\|A\|})$ using bisection search

Table 11.Space Complexity Analysis

Linear 1-D	**HEAP [Solomon]**	**BIT [Fenwick]**	**CFM [Proposed]**
O(|A|)	O(|A|) for symbol, O(|A|) for frequency, O(|A|) for total freq. in left subtree	O(|A|)	O(|A|)
256 words	256 bytes for symbols, total 512 words for freq. and cumu. freq. in left subtree	256 words	256 words

Practical implementation using Fenwick's BIT data structure is done using two methods of computing next index in iteration. We have named these two implementations as BIT1 and BIT2.

In BIT1, index for next iteration is computed using backward and forward functions as described in section 2.4. It is dependent on whether machine is using 1's complement or 2's complement integer arithmetic.

In BIT2, index to be used in next iterations is computed using starting value and step value for loop variables as given by Alexey [7]. Note that this implementation does not require the knowledge of machine architecture implementing 1's or 2's complement as it is not using forward or backward functions.

Thus Experimental results are shown in table 12 for practical implementation of algorithms HEAP (using Heap data structure), BIT1 (using BIT data structure and forward-backward functions), BIT2 (using BIT data structure but not using forward-backward functions) and CFM (using CFM data structure).

In implementations, total execution time of three operations as required during arithmetic encoding is considered here. These operations are: initializing data structure, adapting data model, computing cumulative frequency interval of a given symbol. Anyways, initialization is independent of the size of source.

Several files are taken into consideration for testing. Some of the test files are selected from Calgary and Canterbury corpus, a widely used benchmark and also from web site compression.ca/act/act_files.html.

8. ANALYSIS OF EXPERIMENTAL RESULTS

It is seen that HEAP is not that efficient even when its time complexity is O(n*log$_2$|A|) which is same as that of BIT. It is all due to the cost of maintaining HEAP property (array sorted by frequency of symbols) that requires searching farthest node in array according to frequency of symbols, computing weight of left subtree and rearranging nodes as per frequency of symbols. It has the worst performance among these four algorithms.

Runtime of BIT2 is comparatively higher than BIT1. Remember that it is independent of 1's or 2's complement.

Now comparing BIT1 and CFM, it is seen that there is no significant difference in their performance. This can also be observed from figure 1 where the lines of both CFM and BIT1 are overlapping. To highlight the non-significant difference between these two, figure 2 is included with bars showing performance of only CFM and BIT1.

Table 12.Execution Time (seconds)

No.	**File name**	**Size Bytes**	**HEAP**	**BIT1**	**BIT2**	**CFM**
1	act2may2.xls	1348036	0.6044	0.2747	0.3846	0.2747
2	calbook2.txt	610856	0.4396	0.1648	0.2198	0.1099
3	cal-obj2	246814	0.1648	0.0549	0.1099	0.0549
4	cal-pic	513216	0.2198	0.1648	0.1099	0.1099
5	cycle.doc	1483264	1.1538	0.3297	0.4396	0.3297
6	every.wav	6994092	4.0110	1.4835	2.1429	1.5385
7	family1.jpg	198372	0.2198	0.0549	0.1099	0.0549
8	frymire.tif	3706306	2.6374	0.7143	1.0989	0.7692
9	kennedy.xls	1029744	0.4945	0.2198	0.2747	0.2198
10	lena3.tif	786568	0.8791	0.1648	0.2747	0.1648
11	linux.pdf	8091180	4.5604	1.6484	2.4725	1.7033
12	linuxfil.ppt	246272	0.2198	0.0549	0.0549	0.0549
13	monarch.tif	1179784	1.2637	0.2747	0.3846	0.2198
14	pine.bin	1566200	1.2637	0.3297	0.4396	0.3297
15	sadvchar.pps	1797632	1.4286	0.3846	0.5495	0.3846
16	shriji.jpg	4493896	3.0220	0.9341	1.4286	0.9341
17	world95.txt	3005020	2.0330	0.6044	0.9341	0.6593

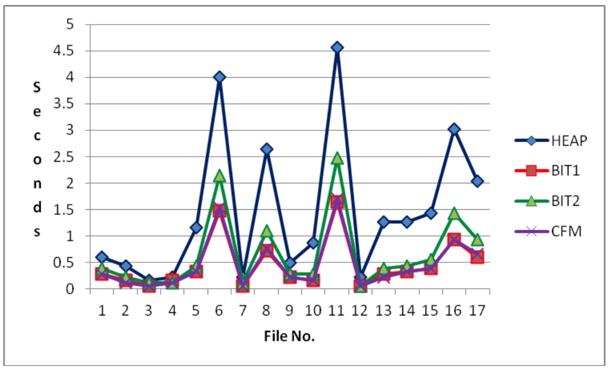

Figure 1. Execution Time (sec)

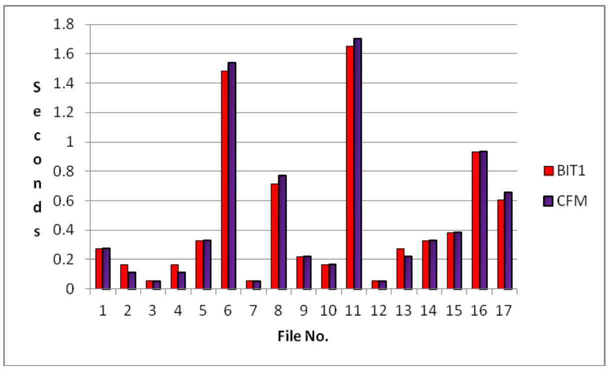

Figure 2.Execution Time (sec) of BIT1 and CFM

9. COMPARISON OF BIT AND CFM

As seen in section 8, there is no significant difference in the runtime of CFM and BIT implemented using BIT1. So they are compared here with respect to other factors like simplicity, ease of implementation, dependence on 1's or 2's complement arithmetic, ability to exploit pointer arithmetic and possibility of overflow in cumulative frequency values.

Following are the advantages of using CFM over BIT1.

As compared to BIT, CFM data structure is much simpler to understand and easier to implement.

In BIT, operations are not performed on consecutive elements; whereas in CFM, operations are performed on consecutive elements of an array. Thus in BIT, computing the index of next element to operate upon is very complex. In CFM, computing next index is simply to increment it.

Algorithm BIT1 requires negation operation while computing next index using backward and forward functions. Negation operation is dependent on implementation of 1's or 2's complement arithmetic on machines. CFM is totally independent of such implementations on machines.

Another advantage of using CFM over BIT is an ability to use pointer arithmetic to access next consecutive element in an array and achieve better performance.

One more advantage of CFM over BIT is the reduced chances of overflow in cumulative frequency values. As a result, it hardly requires re-adjusting frequencies. In CFM, partial cumulative frequencies are sum of at the most 16 symbols within a row; whereas in BIT, it contains sum of 2, 4, 8, 16, 32, 64, 128 and 256 symbols. Thus sum in BIT may exceed the word capacity earlier as compared to sum of maximum 16 elements in CFM.

10. CONCLUSION

Main advantages of using CFM data structure is its simplicity, ease of practical implementation, independence of machine architecture, possibility of exploiting pointer arithmetic to increase execution speed and reduced chances of word overflow in cumulative frequency.
CFM is extremely easy to understand and very convenient in implementation without compromising in performance.

11. ACKNOWLEDGMENT

I thank Ms. Lata Gohil, Assistant Professor at GLS Institute of Computer Technology, for providing her valuable inputs while coding and testing algorithms using various data structures.

12. REFERENCES

[1] I.H.Witten, R.Neal and J.G.Cleary, "Arithmetic compression for data compression", CACM, 30, (6), 520-540 (1987)

[2] A.Moffat, "Linear time adaptive coding", IEEE Trans Info. Theory, 36, (2), 401-406 (1990)

[3] D.W.Jones, Application of splay trees to data compression", Comm ACM, 31, (8), 996-1007 (1988)

[4] Peter M. Fenwick, "A New Data Structure for Cumulative Frequency Tables", Software – Practice and Experience, Vol 24(3), 327-336 (March 1994)

[5] W. Weaver and C.E. Shannon. The Mathematical Theory of Communication. University of Illinois Press, Urbana, Illinois, 1949. Republished in paperback 1963.

[6] Eric Bodden, Malte Clasen, Joachim Kneis, "Arithmetic Coding revealed-A guided tour from theory to praxis", Sable Technical Report No. 2007-5, May 2007, available at http://www.bodden.de/legacy/arithmetic-coding/

[7] Alexey V. Shanin, "Optimizing Tip on Adaptive Arithmetic Coding", January 2002 available at http://www.codeguru.com/cpp/cpp/algorithms/compression/article.php/c5089/Optimizing-Tip-on-Adaptive-Arithmetic-Coding.htm

[8] Paul G. Howard, Jeffrey S. Vitter, "Analysis of Arithmetic Coding for Data Compression", Information Processing and Management, Vol. 28, No. 6, pp. 749-764 (1992)

[9] David Solomon, "Data Compression – The Complete Reference", 3rd edition, Springer, 2004

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

[11] Bin Zhu, En-hui Yang, Ahmed H. Tewfik, "Arithmetic Coding with Dual Symbol Sets and Its Performance Analysis", IEEE transactions on Image Processing, Vol. 8, No. 12, 1999, pp 1667

[12] Ida Mengyi Pu, Fundamental Data Compression, Butterworth-Heinemann, 2006

[13] P. G. Howard and J. S. Vitter, "Practical implementation of arithmetic coding," in Image and Text Compression, J. A. Storer, Ed. Norwell, MA: Kluwer Academic, 1992, pp. 85–112.

[14] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," ACM Trans. Inf. Syst., vol. 16, no. 3, pp. 256–294, 1998.

[15] Ranjan Bose, Saumitr Pathak, "A Novel Compression and Encryption Scheme Using Variable Model Arithmetic Coding and Coupled Chaotic System", IEEE Transaction on Circuits and Systems – I: Regular Papers, Vol. 53, no. 4, 2006, pp 848-857

[16] Algorithm Tutorials available at http://community.topcoder.com

[17] Amir Said, "Introduction to Arithmetic Coding - Theory and Practice", Hewlett-Packard Laboratories Report, HPL-2004-76, Palo Alto, CA, April 2004 available at http://www.hpl.hp.com/techreports