9-30-1991

Cascading space efficient variation of the LZW algorithm and arithmetic coding

Shobha Raj
New Jersey Institute of Technology

ABSTRACT

Title of Thesis: Cascading space efficient variation of the LZW Algorithm and the Arithmetic Coding.

Shobha Raj

Master of Science in Computer and Information Science
October 1991

Thesis Directed By: Dr. Yehoshua Perl
Professor of Computer and Information Science.

Two known compression algorithms appropriate for communication are LZW and the Arithmetic Coding method. Both are adaptive and require no extra communication from the encoder to the decoder. Here we have considered a space efficient variation of the LZW algorithm which achieves better compression.

We present a scheme to cascade the space efficient variation of the LZW algorithm with Arithmetic Coding into a space efficient data compression algorithm which achieves a higher compression ratio and is appropriate for communication.

CASCADING SPACE EFFICIENT VARIATION OF THE LZW ALGORITHM AND ARITHMETIC CODING

by

Shobha Raj

Thesis submitted to the faculty of the Graduate School of
New Jersey Institute of Technology in partial fulfilment of the
requirement for degree of Master of Science in
Computer and Information Science.
1991

APPROVAL SHEET

Title of the Thesis :	Cascading Space efficient variation of the LZW algorithm and the Arithmetic Coding.
Name of the Candidate :	Shobha Raj
	Master of Science in Computer and Information Science 1991
Thesis and Abstract Approved :	------------------------------ Dr. Yehoshua Perl Professor Department of Computer and Information Science
	-------------- Date

VITA

Name :	Shobha Raj	
Permanent Address :		
Degree to be conferred:	Master of Science, 1991 in Computer and Information Science	
Date of Birth:		
Place of Birth:		
Collegiate Institutes Attended	**Dates**	**Degree**
New Jersey Institute of Technology Newark, NJ 07102	09-89 to 09-91	MSCIS
B.M.S. College of Engineering Bangalore, India	07-82 to 05-87	BSEE
Positions Held :	Systems Assistant Computing Services Department, NJIT	
	Teaching Assistant Physics Department, NJIT	
	Software Engineer Motor Industries Pvt Ltd. Bangalore, India	

Dedicated to my parents and family

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to Dr. Yehoshua Perl without whom this research would not have been possible.

I would also like to thank my colleagues and friends for their co-operation at all times.

My special thanks to my family and relatives for the encouragement and moral support.

TABLE OF CONTENTS

LIST OF TABLES

I Introduction

Two of the most known compression algorithms are the LZW algorithm and Arithmetic Coding [R1,R2]. Both the algorithms are adaptive hence appropriate for communication. They do not require a prior knowledge on the text to be compressed. Furthermore, during the compression process they do not require any extra information to be transferred from the encoder to the decoder apart from the compressed text. The LZW algorithm is adaptive in nature. The space efficient LZW algorithm is a variation of the LZW algorithm that uses post order traversal of the tree when scanning the leaves to delete.

The two algorithms differ in principle they use to achieve compression. The space efficient variation of the LZW algorithm like the LZW algorithm constructs a dictionary of variable length strings from the text and replaces each occurrence of a string by the (usually shorter) index to the appropriate entry of the dictionary. The Arithmetic Coding encodes each character separately. It accumulates the frequency of the characters in the text and uses a variable number of bits to transfer each character according to its frequency.

The problem of constructing a compression dictionary of variable length character strings from the text has received considerable attention in the literature. Heuristic experiments are reported by Lynch [L]. The problem of finding such an optimum dictionary maximizing the compression is considered in Storer and Szymanski[SS1] and [SS2]. They model the dictionary as one long string, where each substring constitutes an entry in the dictionary, and show that finding an optimum dictionary is a NP-Hard, i.e., probably not solvable in polynomial time. A dynamic version of this model called the Sliding dictionary is used in Smith and Storer [SS3] for parallel data compression.

Choueka, Fraenkel and Perl [CFP] show that if the dictionary is limited to contain only prefixes or suffixes of words in the text, then it is possible to construct an optimum dictionary in polynomial time by dynamic programming algorithm. On the other hand if both suffixes and prefixes are permitted in the dictionary then the problem is shown by Fraenkal, Mor and Perl to be NP-Hard. They describe experiments with hueristics for constructing dictionaries with only prefixes and both prefix and suffixes. The LZW algorithm differs from all these algorithms except the Sliding Dictionary of [SS3] in that it is adaptive, while all others are based on preprocessing of text.

The fact that the two algorithms differ in the principle raises the question as to whether there is a way to combine the two algorithms to yield a higher compression and still be appropriate for the communication process. We now present an algorithm for compression obtained by cascading the space efficient variation of the LZW algorithm and the arithmetic coding. The technique involves the accumulation of the frequencies of the entries of the space efficient LZW algorithm and utilizing these frequencies by the Arithmetic coding which will relate to the entries of the spaces efficient variation of the LZW dictionary as an alphabet. Since these entries differ in their frequencies an extra secondary compression is achieved, in addition to the primary compression due to LZW algorithm. Different variation of refining the cascading are tested to optimize the secondary compression. For communication purposes the compression time does not matter as the communication speed is much slower than the computation process.

In the next two sections the space efficient variation algorithm and the arithmetic coding techniques are described. Their cascading is discussed in section 4 and the experiments are reported in Section 5. The secondary compression achieved is higher for smaller dictionary sizes. The experiments show that fine tuning of the cascading process yields a significant increase in the compression ratio.

II. The Space Efficient Variation of the LZW Algorithm

One of the known compression algorithms is the algorithm of Lempel and Ziv [ZL1][ZL2]. Welch [W] described the implementation of this algorithm known as the LZW algorithm. This algorithm has the advantage of being adaptive. That is, the algorithm does not assume any advance knowledge of the properties of the input and builds the dictionary used for the compression only on the basis of the input itself, as the input is read. This property is especially important in compression for communication. This method is in contrast to compression algorithms which are based on advance knowledge of the properties of the input, e.g Huffman Algorithm.

The LZW algorithm starts with a dictionary containing entries for each character in the alphabet. The algorithm scans the input matching it with entries in the dictionary. When the matching is finished, i.e, we read from the input a string Y, not in the dictionary, such that Y = X.a, where X is a string already in the dictionary, "a" is a character and "." is the concatenation operator. The encoder then sends the code for X (index in the dictionary table) and inserts Y into the dictionary. The string Y is called as character extension of X. The encoding of the input continues from the character "a" that follows X. The decoder builds an identical dictionary to the one built by the encoder. The process works as follows. When the decoder accepts the code for X it already has code in its dictionary, so it identifies X and sends it to the output. The next code accepted by the decoder will also lie in its dictionary so it knows the corresponding string starts with a character "a". Thus the decoder concludes the encoder has inserted the string Y = X.a into its dictionary. The decoder then includes Y as the next entry in his dictionary. This way both the encoder and the decoder construct the same dictionary without transmitting the dictionary. Detailed description appears in [W] and [PCM].

We present an example of an application of the LZW algorithm for the Binary text 0100101011110101101.

The dictionary obtained is as follows

1	2	3	4	5	6	7	8	9	10	11	12
0	1	01	10	00	010	0101	11	111	101	1010	100

and the coded output is : 1, 2, 1, 3, 6, 2, 8, 4, 10, 4, 3.

The entries for the LZW dictionary satisfy the property that "If a string X is in the dictionary then every prefix of X is also in the dictionary. Another property is that for every code sent by the encoder a new entry is added to the dictionary. Since the dictionary is finite and may be limited for practical reasons, the dictionary fills up fast. The LZW algorithm then continues encoding according to the existing dictionary without adding new entries. After a while a significant decline in the compression ratio is observed. This decline is typically due to change in the properties of the text , so the dictionary is no longer appropriate.

Totally ignoring the old dictionary and building a new one from scratch is wasteful. Even though the decline in the compression ratio is no longer adequate, there are probably parts of it that will appear in the new dictionary. The various methods of incorporating updates into the LZW algorithm resulting in continuously adaptive versions are reported by Perl, Chatterjee and Mahapatra [PCM].

It is helpful to view the dictionary as an ordered labeled rooted tree. Each edge emanating from a vertex is labeled by a character of the alphabet. A vertex represents the string obtained by the concatenation of all the characters along the path from the root to the vertex. Thus all vertices on the path from the root to a vertex representing a string X of the dictionary represent all the prefixes of X. Using this tree representation if the string of a vertex is deleted then all its descendents are deleted. The approach used for deleting keys in the dictionary are reported in [PCM].

Since the encoder and decoder have identical dictionaries, they get full at the same time. Also the update process is identical for the encoder and the decoder. Therefore after deletions encoder and the decoder will still have identical dictionaries. At this point we consider adding identical new strings to both the dictionaries. Hence at each point of the process the encoder and the decoder work with identical dictionaries.

Continuous adaptive approach of the LZW algorithm is based on deleting from the the dictionary the keys of low frequency which are likely to appear in the future text and use their space to add new entries to the dictionary. For this purpose computation of frequency of each entry in the dictionary along with pointers to the parent ,child and sibling is added to the LZW algorithm [PCM]. The space efficient variation of the LZW algorithm is a variation of the adaptive version designed to reduce storage requirements by not having a parent pointer. The order in which the leaves are deleted are arbitrary with respect to their position in the tree, but we traverse the tree in post-order traversal and delete the leaves in left to right order. The implementation of this has been reported in [PCM]

III The Arithmetic Coding Method

In the Arithmetic Coding method (e.g.,[CNW]) strings of characters are represented by a numeric interval of real numbers between 0 and 1. As the string becomes longer, the interval needed to represent it becomes smaller and the number of bits needed to specify that interval grows. Successive symbols of the strings reduce the size of the interval in accordance with their symbol probabilities in the input. The more likely symbols reduce the range by less than the unlikely symbols and hence add fewer bits to the interval representation of the encoded message. This method is adaptable and does not need the probabilities of symbols in the input in advance. These probabilities could be dynamically updated as the input is read, and mapped into the interval.

The first references to the Arithmetic Coding method appear in [A]. Practical techniques were first introduced by Rissanen [R1][R2]. Detailed implementation is presented in [CNW]. A broader class of Arithmetic Codes are presented in [RL]. In [PG] Perl and Gabriel used the Arithmetic Coding Method to modify Interpolation search for Alphabetic tables.

The Arithmetic Coding method has advantages over the Huffman Coding Method [H]. In addition to the requirement of symbol probabilities in advance , in the Huffaman coding method each symbol has to be translated into integral number of bits in the encoding. Huffman coding indeed achieves minimum redundancy, in other words, it performs optimally, according to these assumptions. But the requirement of the integral number of bits of Huffman coding can take upto one extra bit per symbol. The worst case is realized by a source in which one has probability approaching unity. The Arithmetic Coding method dispenses with the restriction that each symbol must translate into an integral number of bits, thereby coding more efficiently since there is a better relationship between the symbol probability in the input and the corresponding number of bits it contributes to the interval. The Arithmetic Coding actually achieves theoretical bound to compression efficiency for any source [CNW]. In the implementation of Arithmetic Coding method it is required to maintain and update the cumulative frequencies of the symbols in the input. The cumulative frequency of a symbol is the sum of the frequencies of all the symbols preceding and including this symbol in the frequency table. We use the frequencies in calculating the new interval to represent the occurrence of the symbol whenever a symbol is being accepted by the encoder while encoding and recognized by the decoder while decoding.

Initially we assume a frequency of 1 for every symbol and initialize the cumulative frequency table accordingly. Low and High represent the interval at any instance of the compression process. Before anything is transmitted Low and High are initialized with 0 and Top_value respectively. Integer arithmetic is adapted to overcome the overflow and underflow problems of the floating point arithmetic. Incremental transmission and incremental reception [CNW] are used to get around the restriction on the length of the message by the narrowing interval. The values of High and Low are compared for the same higher order bits and equal bits in both High and Low are buffered for transmitting to the decoder, as these bits are not affected by future narrowing. Packets made of 8 such bits are sent to the decoder. The interval is readjusted and the next symbol is accepted and processed in the same way. This process is continued until a special symbol (EOF) is encountered. This procedure takes care of very long strings. The decoder receives these packets incrementally using a variable value. In the beginning the value consists of bits received. the initial values of Low and High and the initialization and updating of the dictionary follows that of the encoder. As the symbols are identified, the processed bits flow out at the high significance end and the newly received fresh ones flow in at the low significant end. this process is continued until the special symbol (EOF) is recognized.

The cumulative frequency table is sorted in the frequency order to minimize the number of updates to it after every symbol is processed. Translation tables of character to index (char_to_index) and Index to char (ind_to_char) are used to simplify the process of sorting the cumulative frequency table is updated. To overcome the overflow and underflow problems of the integer arithmetic, the frequencies are scaled down by a normalization factor at regular steps.

Consider an alphabet of symbols {a,e,i,o,u,!}. Initially let us assume a frequency of 1 for every symbol. For the simplicity of presentation we use real arithmetic. As each symbol is processed, the range is narrowed to that portion of it is allocated to the symbol. Consider transmitting the message eaii!. Initially, both encoder and decoder know that the interval is [0,1] and the probability table is:

Symbol	Cum. Freq	Probability	Range
a	1	0.167	[0.000 0.167]
e	2	0.167	[0.167 0.333]
i	3	0.167	[0.333 0.500]
o	4	0.167	[0.500 0.667]
u	5	0.167	[0.667 0.833]
!	6	0.167	[0.833 1.000]

After seeing the first symbol, "e", the encoder narrows the range to [0.167,0.333] as "e" was allocated the range [0.167, 0.333] in the initial range distribution. The probability table is updated as follows:

Symbol	Cum. Freq	Probability	Range
a	1	0.143	[0.000 0.143]
e	3	0.286	[0.143 0.429]
i	4	0.143	[0.429 0.572]
o	5	0.143	[0.572 0.715]
u	6	0.143	[0.715 0.858]
!	7	0.143	[0.858 1.000]

The second symbol "a" narrows the interval to [0.167,0.191] which is the 1/7 of the previous interval [0.167, 0.333], since "a" was of the range [0.000, 0.167] or [0.0, 1/7] in the previous probability table. The probability table after seeing "a" is:

Symbol	Cum. Freq	Probability	Range
a	2	0.250	[0.000 0.250]
e	4	0.250	[0.250 0.500]
i	5	0.125	[0.500 0.625]
o	6	0.125	[0.625 0.750]
u	7	0.125	[0.750 0.875]
!	8	0.125	[0.875 1.000]

Similarly after seeing the third symbol "i" the interval is narrowed to [0.179, 0.182] and the probability table is updated to:

Symbol	Cum. Freq	Probability	Range
a	2	0.222	[0.000 0.222]
e	4	0.222	[0.222 0.444]
i	6	0.222	[0.444 0.666]
o	7	0.111	[0.666 0.777]
u	8	0.111	[0.777 0.888]
!	9	0.111	[0.888 1.000]

Similarly after seeing the fourth symbol "i", the interval is narrowed to [0.1803, 0.1810) and the probability table is changed to:

Symbol	Cum. Freq	Probability	Range
a	2	0.200	[0.000 0.200]
e	4	0.200	[0.200 0.400]
i	7	0.300	[0.400 0.700]
o	8	0.100	[0.700 0.800]
u	9	0.100	[0.800 0.900]
!	10	0.100	[0.900 1.000]

Similarly after seeing the message terminating symbol "i", the final interval is [0.1809, 0.1810). This range is transmitted to the decoder.

When the decoder receives the final range [0.1809, 0.1810) it follows the same procedure of the encoder and successfully decodes the message eaii!

IV. Cascading

The space efficient variation of the LZW algorithm builds up a dictionary of 2^k entries from the text. In the encoding of the input text, we match a maximum length prefix which appears in the dictionary and replace it by the index of k bits of the appropriate entry in the dictionary. The encoding then continues with the rest of the input, similarly. This way each string in the input text is replaced by an index of the dictionary.

We can count the frequencies of the entries of the space efficient variation of the LZW dictionary, i.e., the number of times each entry of the dictionary is sent in the compression process. We then notice that not all entries of the dictionary have the same frequency, some may be used more often than the others and some may never be used. This skewed distribution of the frequencies of the space efficient variation of the LZW dictionary could be well utilized in the Arithmetic coding. We apply the Arithmetic Coding method algorithm by treating the entries of space efficient variation of the LZW dictionary as alphabet to obtain further compression. We call the compression obtained by applying the LZW algorithm to the input text, "primary compression" and the extra compression achieved by applying the Arithmetic Coding method to the compressed data from LZW encoder, "Secondary compression".

In this cascading process, the alphabet for Arithmetic coding method is the set of indices of the space efficient variation of the LZW dictionary, i.e., integers from 0 to 2^k. We maintain a cumulative frequency table, Low and High values to represent the interval at any instance of compression process. The initial upper and lower bounds of the range are Top_value and 0 respectively. To start with, we assign High = Top_value and Low = 0. We assume a frequency of 1 for every entry of the frequency table representing the fact that such an entry was inserted into the dictionary although it was not sent at the first occurrence.

A corresponding cumulative frequency table is constructed. The output from the encoder of the LZW algorithm is directed to the encoder of the Arithmetic coding method. As the encoder of Arithmetic coding method receives the indices of the LZW dictionary, it updates the cumulative frequencies of all the symbols from that index to the end of the table. We use different methods in updating the cumulative frequency table during the compression process using different weights added to the frequencies. The algorithm then modifies low and high values to represent the occurrence of the symbol just received.

Initially we assume a frequency of 1 for every symbol. We have adopted different strategies in updating the cumulative frequency table. Instead of incrementing the cumulative frequencies of the symbol occurred (both at the decoder and encoder end) by a value of 1, we have used different weights for different phases of the compression process. To describe the weights for updates in the different phases, we used a triple weight method (X, Y, Z) where X is the initial weight for an entry, when it is inserted into the dictionary, Y is the weight of the update frequency for each occurrence of the existing entry until the dictionary gets full.

We now explain the reason for having different weights for these three phases of the compression process. The initial weight X is assigned as the frequency of an entry which was inserted into the dictionary but was never sent. A large portion of these entries are never sent. Thus we want to assign this initial occurrence a lower weight than further occurrences. in cascading the space efficient variation of the LZW algorithm with the Huffman coding [PM] found experimentally that assigning these occurrences half the weight of the other occurrences optimizes the secondary compression.

We also distinguish between the occurrences before and after the dictionary gets full. In the description we refer to the tree representation of the dictionary described in section

2. Consider an entry with "d" children which are leaves in the tree representation of the dictionary, when it got full. This means that this entry was sent each time one of its children was inserted into the dictionary. However once the dictionary gets full no entries are added to the dictionary, we expect the children of this entry to be sent rather than the entry itself due to the maximum length matching rule used by the space efficient variation of the LZW algorithm. To reflect this expectation we choose a weight Z (higher than Y) for occurrences after the dictionary gets full.

Cascading the space efficient variation of LZW Algorithm with Arithmetic coding eases many of the problems encountered in cascading LZW algorithm with the Huffman coding method [PM]. The Huffman coding method requires actual frequencies of the symbols in the input text known at the beginning of the compression process. The Arithmetic Coding method is not constrained by this requirement as we adopt a dynamic update of the cumulative frequency table. Cascading of the space efficient variation of LZW algorithm with Arithmetic Coding achieves better results than the results obtained by cascading LZW algorithm with Arithmetic Coding [PMK][PMMK]. These results are comparable with the results obtained by cascading the ever adaptive variation of the LZW compression algorithm and the Arithmetic coding method [PT].

In the experiments reported in the next section we test for appropriate values for X, Y and Z.

V. Experimental Results

The Experiments were conducted using different data files. The files used were a UNIX manual of 202792 bytes, a text book of Engineering, a binary data file of 77824 bytes and a file of 58284 bytes made up of the first three chapters of the text book of Engineering. Different dictionary sizes were used for all of these files. Different strategies were used to update the cumulative frequency table. The results are presented in tables 1 through 8.

From the tables 1 through 8 we observe that the secondary compression is higher in the case of smaller dictionary sizes and larger data sets. The reason for higher compression for smaller dictionary is that for those LZW is less effective and there is more room for further improvements. The reason for higher compression for larger texts is that they enable better approximation.

To test independently the effect of the two reasons mentioned above for varying weights we perform two sets of experiments. One version, described by the triple (1,W,W) for W = 1, 2, 3 checks the effect of higher weight for each occurrence of an entry versus the initial occurrence when the entry was inserted into the dictionary but not sent. The results for W = 2 is marginally better than for W = 1. For W = 3 there is only a slight improvement over W = 2 which does not justify the extra computation time involved (The extra time taken is by the Arithmetic Coding process and is not explained here).

The second version described by the triple (1, 1, W) for W= 1,2, 3 checks the effect of higher weight for each occurrence after the dictionary gets full. The results for W = 2 is higher than W= 1 and marginally lower than W = 3. These results are slightly higher than the version (1, W, W) except for the largest dictionary sizes for which it is slightly lower.

Checking a combined effect of both the versions is a version described by the triple (1, 2, 3) shows only a slight improvement over the previous version. Thus our conclusion is that either of the versions (1,2,2) or (1, 1, 2,) should be used where the difference is negligible.

In general our conclusion is that cascading the space efficient variation of the LZW algorithm with Arithmetic Coding yields a significant increase in the compression ratio. This increase is specially high for small dictionary sizes and binary files. However as we show the cascading should not be done directly, but a fine tuning is required.

Table 1

File ch123 size 58284 bytes
Updating method: (1,W,W)
Compression ratio using Arithmetic Coding = 36.11

Dictionary size # of bits	Comp ratio Traversal	Comp Ratio Cascade W=1	Comp Ratio Cascade W=2	Comp Ratio Cascade W=3
9	49.25	49.52	50.41	50.8
10	56.93	55.5	55.95	56.22
11	60.71	59.26	59.26	59.43
12	62.42	61.64	61.48	61.47
13	62.48	61.32	61.29	61.28

Table 2

File Allbin size 77824 bytes
Updating method: (1,W,W)
Compression ratio using Arithmetic Coding = 23.61

Dictionary size # of bits	Comp Ratio Traversal	Comp Ratio Cascade W=1	Comp Ratio Cascade W=2	Comp Ratio Cascade W=3
9	40.43	40.3	40.77	40.88
10	39.90	40.97	41.18	41.32
11	39.45	41.98	42.19	42.31
12	40.81	44.33	44.65	44.55
13	45.19	46.75	46.81	46.77

Table 3

File Allch size 151305 bytes
Updating method: (1,W,W)
Compression ratio using Arithmeic Coding = 38.26

Dictionary size # of bits	Comp Ratio Traversal	Comp Ratio Cascade W=1	Comp Ratio Cascade W=2	Comp Ratio Cascade W=3
9	47.53	50.09	50.49	50.63
10	55.61	55.97	56.09	56.14
11	59.87	59.88	59.85	59.84
12	62.45	62.62	62.6	62.55
13	63.62	63.27	63.25	63.16

Table 4

File Allman size 202792 bytes
Updating method: (1,W,W)
Compression ratio using Arithmetic Coding = 37.242

Dictionary size # of bits	Comp Ratio Traversal	Comp Ratio Cascade W=1	Comp Ratio Cascade W=2	Comp Ratio Cascade W=3
9	47.44	50.21	50.49	50.62
10	54.33	55.00	55.09	55.09
11	58.74	59.03	59.02	58.98
12	61.34	61.81	61.83	61.74
13	62.85	62.87	62.82	62.68

Table 5

File ch123 size 58284 bytes
Updating method (1,w1,w2)
Compression ratio using only Arithmetic Coding = 36.11

# of bits	LZW Trav	(1,1,1)	(1,1,2)	(1,1,3)	(1,2,3)	(1,3,2)
9	49.25	49.52	50.97	51.33	51.33	50.99
10	56.93	55.50	56.14	56.31	56.32	56.25
11	60.71	59.26	59.33	59.59	59.57	59.55
12	62.42	61.64	61.54	61.51	61.25	61.30
13	62.48	61.32	61.30	61.26	61.41	61.57

Table 6

File allbin Size 77824 bytes
Updating method (1,w1,w2)
Compression ratio using only Arithmetic Coding = 23.61

# of bits	LZW Trav	(1,1,1)	(1,1,2)	(1,1,3)	(1,2,3)	(1,3,2)
9	40.43	40.39	40.85	40.95	41.02	40.90
10	39.39	40.97	41.22	41.31	41.38	41.32
11	39.45	41.98	42.25	42.29	42.35	42.38
12	40.81	44.33	44.56	44.51	44.59	44.68
13	45.19	46.75	46.79	46.65	46.69	46.63

Table 7

File allch Size 151305 bytes
Updating method (1,w1,w2)
Compression ratio using only Arithmetic Coding = 38.26

# of bits	LZW Trav	(1,1,1)	(1,1,2)	(1,1,3)	(1,2,3)	(1,3,2)
9	47.53	50.09	50.70	50.84	50.84	50.71
10	55.61	55.97	56.17	56.18	56.18	56.21
11	59.87	59.88	59.96	59.90	59.91	59.98
12	62.45	62.62	62.61	62.55	62.52	62.66
13	63.72	63.27	63.26	63.13	63.13	56.67

Table 8

File allman Size 202792 bytes
Updating method (1,w1,w2)
Compression ratio using only Arithmetic Coding =37.242

# of bits	LZW Trav	(1,1,1)	(1,1,2)	(1,1,3)	(1,2,3)	(1,3,2)
9	47.44	50.21	50.59	50.70	50.71	50.63
10	54.33	55.00	55.17	55.13	55.15	55.18
11	58.74	59.03	59.07	58.98	59.01	59.11
12	61.34	61.81	61.87	60.02	60.05	61.86
13	62.85	62.87	62.80	62.65	62.66	62.84

REFERENCES

[A] N.Abramson:Information Theory and Coding, McGraw-Hill, New York,1963.

[CFP] Y.Choueka, A.S.Fraenkel, Y.Perl: Polynomial Construction of optimal prefix tables for text compression. Proc. of 19th Annual Allerton Conferences on Communication, control and computing,1981,762-768.

[CNW] J.G.Clearny, R.M. Neal, I.H.Witten: Arithmetic coding for data Compression. Communications of the ACM, 30, 1987,520-540. [FMP] A.S.Fraenkel, M.Mor, Y.Perl: Is Text compression by prefixes and suffixes practical? Acta Informatica 20, 1983, 371-389.

[H] D.A.Huffman: A method for construction of minimum redundany codes.Proc. Inst. Electr. Radio Engg. 40, 1952 1098-1101.

[L] M.F. Lynch: Compression of Biblographic files using adoption of run-length coding. Inform Stor. Retr. 9, 1973, 207-214.

[PCM] Y.Perl, S.Chaterjee, T.Mahapatra: Incorporating Updates into the LZW algorithm. Presented at the 21st Southeastern International Conference on Combinatrics, Graph Theory and Computing, Boca Rotan 1990.

[PG] Y.Perl, L.Gabriel: Arthmetic interpolation search for Alphabetic Tables, Research Report, CIS-89-18, New Jersey Institute of Technology, Newark, NJ, to appear in IEEE Trans. on Computers.

[PGS] Y.Perl, S.Gupta, A.Srivastva: Towards a bound for the compression of the LZW algorithm. Presented at the 21st Southeastern International Conference on Combinatrics, Graph Theory and Computing, Boca Rotan 1990.

[PM] Y.Perl, A.Mehta: Cascading LZW algorithm with Huffman Coding method: a variable to variable length compression algorithm. Presented at the First Great lakes Computer Science Conference, Kalamazoo, 1990.

[PMK] Y.perl, V.Maram, N.Kadakuntla: The Cascading of Two Compression Algorithms: LZW and Arithmetic Coding, in the proceedings of Data Compression Conference,1991, pp. 277-286.

[PMMK] Y.Perl, A.Mehta, V.Maram, N. Kadakuntla: The variable to variable length Cascaded Compression Algorithms,a Journal Papaer in preparation.

[PT] Y.Perl, K.Tejani: Cascading the ever adaptive variation of the LZW compression algorithm and the arithmetic Coding method, to be submitted.

[R1] Rissanen J.J: Generalized Kraft inequality and Arithmetic Coding. IBM J. Res. Dev 20, 1976, 198-203.

[R2] Rissanen J.J: Arithmetic Coding as number representations. Acta Polytech. Scand. Math 31, 1979, 44-51.

[RL] Rissanen J.J and Langdon, G.G. : Arithmetic Coding. IBM J. Res. Dev 23.2, 1979, 149-162.

[S] Storer J.A.: Data Compression Methods and Theory. Computer Science Press, 1988, Rockville, MD.

[SS1] J.A. Storer, T.G. Szymanski : The macro model for data compression. Proc. Tenth Annual ACM symposium on Theory of Computing, San Diego, CA, 1978, 30-39.

[SS2] J.A. Storer, T.G. Szymanski : Data Compression via textual substitution. JACM 29, 1982, 928-951.

[SS3] M.E.G Smith, J.A. Storer: Parallel algorithms for Data Compression, JACM, 32, 1985.

[W] T.A. Welch: A technique for high performance Data Compression, IEEE Comp. Journal, 1984, 8-19.

[ZL1] J. Ziv, A. Lempel: Compression of individual sequences via variable rate coding. IEEE Trans. on Info. Theory. Vol.IT-24, 1978, 530-536.