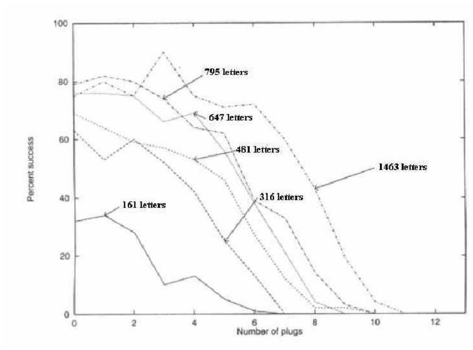
Code Breakers Journal

SECURITY ANALYSIS

SEPTEMBER 17, 2005 HTTP://WWW.CODEBREAKERS-JOURNAL.COM VOL. 2, NO. 3 2005

This paper describes cryptographic methods for concealing information during data compression processes. These include novel approaches of adding pseudo random shuffles into the processes of dictionary coding (Lampel-Ziv compression), arithmetic coding, and Huffman coding. An immediate application of using these methods to provide multimedia security is proposed.



Cryptography in Data Compression

Chung-E Wang

1. Introduction

Data compression is known for reducing storage and communication costs. It involves transforming data of a given format, called source message, to data of a smaller sized format, called codeword.

Data encryption is known for protecting information from eavesdropping. It transforms data of a given format, called plaintext, to another format, called cipher text, using an encryption key.

The major problem existing with the current compression and encryption methods is the large amount of processing time required by the computer to perform the tasks. To lessen the problem, I combine the two processes into one.

To combine the two processes, I introduce the idea of adding a pseudo random shuffle into a data compression process. The method of using a pseudo random number generator to create a pseudo random shuffle is well known. A simple algorithm as below can do the trick. Assume that we have a list (x_1, x_2, … x_n) and that we want to shuffle it randomly.

```
for i = n downto 2 {
   k = random(1,i);
   swap $x_i$ and $x_k$;
}
```

Since we are adding pseudo random shuffles into data compression processes, understanding all three compression algorithms used is critical in the understanding of our algorithms.

Even though our algorithms are based on random shuffles, our algorithms don’t merely re-ordering data. Unlike substitution ciphers, our algorithms don’t encrypt plaintext by simply replacing a piece of data with another equal sized data. Unlike transposition cipher, our algorithms don’t encrypt plaintext by just playing anagrams.

As I will explain in detail, simultaneous data compression and encryption offers an effective remedy for the execution time problem in data security. These methods can easily be utilized in multimedia applications, which are lacking in security and speed.

2. Adding a Pseudo Random Shuffle to a Dictionary Coding (Lampel-Ziv compression)

During a Lampel-Ziv (LZ) compression [10, 12], a group of consecutive characters is replaced with an index into a dictionary that is built during the compression. There are many implementations of the LZ compression. Naturally, different implementations of the LZ compression have different means of implementing the dictionary.

FIG. 1 illustrates the steps of combining a random shuffle with a LZ compression to achieve simultaneous data compression and encryption. In step 110, the encryption key is used to initialize a pseudo random number generator. In step 120, the pseudo random number generator is used to shuffle the initial values of the dictionary.

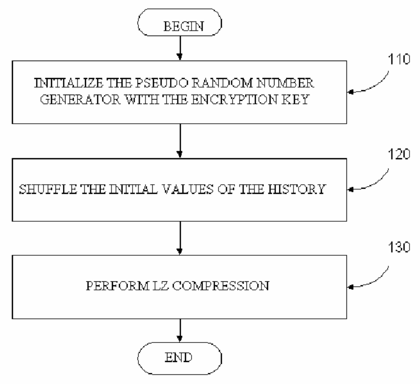

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
110
SHUFFLE THE INITIAL VALUES OF THE HISTORY
120
PERFORM LZ COMPRESSION
130
END

Fig. 1 Combining a random shuffle with a LZ compression

In a codebook type of implementation such as the LZW compression [10], the dictionary consists of strings of characters that have been processed. Initially, it contains all strings of length 1 in alphabetical order. In this case, step 120 shuffles all strings of length 1 plus a small number of null strings which is determined by the first number generated by the pseudo random number generator. The purpose of including null strings in the shuffling is to make chosen plaintext attacks more difficult. Moreover, it cripples any unauthorized decompression. FIG.2 shows an example in which the size of the alphabet, i.e. the set of permissible characters, is 5 and the random number is 3.

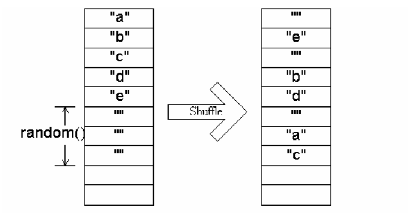


Fig. 2 Shuffling an initial codebook

In a sliding window type of implementation, e.g. LZ77 [12], the dictionary is a window that consists of the last n characters processed, where n could be as big as the size of the window. Initially, the window is empty. In this case, step 120 first uses the pseudo random number generator to generate a number, say m, between s and 2s, where s is the size of the alphabet. Then the step fills the first m entries of the window by cycling through all characters of the alphabet in alphabetical order and then shuffling the window. Note that the purpose of choosing m randomly between s and 2s is to make chosen plaintext attack more difficult. Again, it also cripples an unauthorized decompression. FIG. 3 shows an example in which s=5 and m = 8.

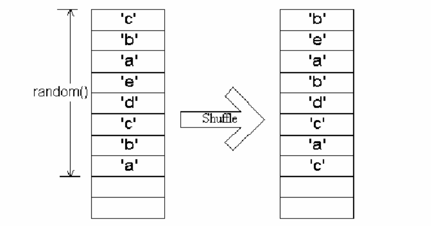


Fig. 3 Shuffling an initial window

In step 130, the compression process is performed on the input string in its usual fashion.

FIG. 4 illustrates the steps of simultaneous decompression and decryption. In step 410, the encryption key is used to initialize the pseudo random number generator. Note that the pseudo random number generator used in step 410 should be identical to the one used in step 110. In step 420, the pseudo random number generator is used to shuffle the initial values of the dictionary as in step 120. In step 430, the decompression is performed in its usual fashion. The output of step 430 is the final decompressed and decrypted string.

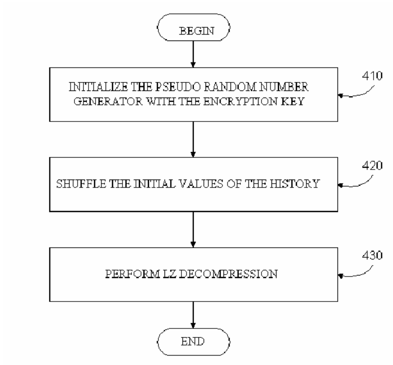

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
410
SHUFFLE THE INITIAL VALUES OF THE HISTORY
420
PERFORM LZ DECOMPRESSION
430
END

Fig. 4 Simultaneous decompression and decryption

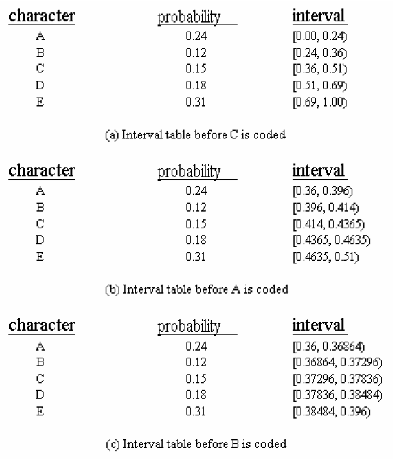
3. Adding a Pseudo Random Shuffle to an Arithmetic Coding

In an arithmetic coding [6, 8, 11], a message of any length is coded as a real number between 0 and 1. The length of the message determines the precision used in the coding. A longer message is encoded with more precision. This is done as follows:

1. Initialize the current interval as [0,1), i.e. the set of real numbers from 0 to 1, including 0 and excluding 1.
2. Divide the current interval into smaller intervals according to the probability distribution of permissible characters.
3. From these new intervals, choose the one corresponding to the next character in the source message as the new current interval.
4. Repeat steps b) and c) until the entire message is coded.
5. Represent the last current interval's value using a binary fraction.

FIG. 5 shows an example. The message to be coded is “CAB”. Probabilities of permissible characters are shown in all three tables. Part (a) shows the intervals before the coding of the first character ‘C’. Part (b) shows the intervals before the coding of the second character ‘A’. Part (c) shows the intervals before the coding of the third character ‘B’. The number 0.36864 is the final result of the arithmetic coding.

An arithmetic coding could be static or adaptive. In a static arithmetic coding, the probabilities of characters stay the same in the entire coding process. In an adaptive arithmetic coding, probabilities of characters are updated according to the data processed.



character	probability	interval
A	0.24	[0.00, 0.24)
B	0.12	[0.24, 0.36)
C	0.15	[0.36, 0.51)
D	0.18	[0.51, 0.69)
E	0.31	[0.69, 1.00)

(a) Interval table before C is coded

character	probability	interval
A	0.24	[0.36, 0.396)
B	0.12	[0.396, 0.414)
C	0.15	[0.414, 0.4365)
D	0.18	[0.4365, 0.4635)
E	0.31	[0.4635, 0.51)

(b) Interval table before A is coded

character	probability	interval
A	0.24	[0.36, 0.36864)
B	0.12	[0.36864, 0.37296)
C	0.15	[0.37296, 0.37836)
D	0.18	[0.37836, 0.38484)
E	0.31	[0.38484, 0.396)

(c) Interval table before B is coded

Fig. 5 Example Interval tables

3.1 The algorithm

The basic approach to concealing information within the process of an arithmetic coding entails the use of an encryption key to shuffle the probability table before the coding begins. Without the encryption key, the probability table cannot be shuffled in the same manner. Thus, the division of an interval into smaller intervals will not be the same. Because of these incongruous divisions, decompression cannot be done properly. Consequently, the original information cannot be retrieved.

FIG. 6 illustrates the steps of combining a random shuffle with the arithmetic coding. In step 610, the encryption key is used to initialize a pseudo random number generator. In step 620, the pseudo random number generator is used to shuffle the probability table. In step 630, the arithmetic coding process is performed on the input message in its usual fashion.

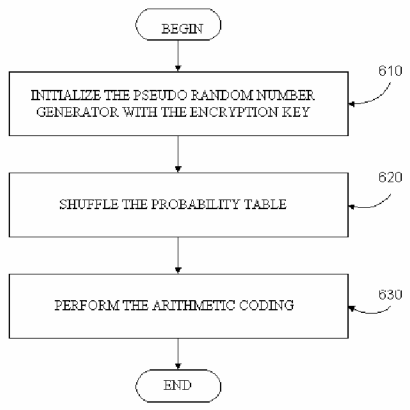
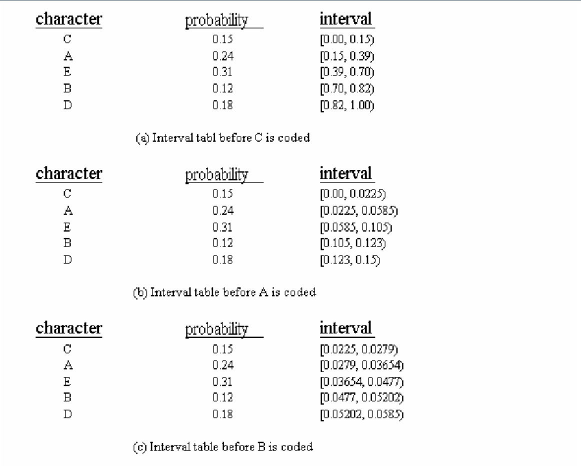

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
610
SHUFFLE THE PROBABILITY TABLE
620
PERFORM THE ARITHMETIC CODING
630
END

Fig. 6 Combining a random shuffle with the arithmetic coding

FIG. 7 shows the effect of a pseudo random shuffle. As in FIG. 5, Parts (a), (b), and (c) show the intervals before the coding of characters ‘C’, ‘A’, and ‘B’ respectively. The number 0.0477 is the final result of the arithmetic coding.



character	probability	interval
C	0.15	[0.00, 0.15)
A	0.24	[0.15, 0.39)
E	0.31	[0.39, 0.70)
B	0.12	[0.70, 0.82)
D	0.18	[0.82, 1.00)

(a) Interval tabl before C is coded

character	probability	interval
C	0.15	[0.00, 0.0225)
A	0.24	[0.0225, 0.0585)
E	0.31	[0.0585, 0.105)
B	0.12	[0.105, 0.123)
D	0.18	[0.123, 0.15)

(b) Interval table before A is coded

character	probability	interval
C	0.15	[0.0225, 0.0279)
A	0.24	[0.0279, 0.03654)
E	0.31	[0.03654, 0.0477)
B	0.12	[0.0477, 0.05202)
D	0.18	[0.05202, 0.0585)

(c) Interval table before B is coded

Fig. 7 Shuffled Interval tables

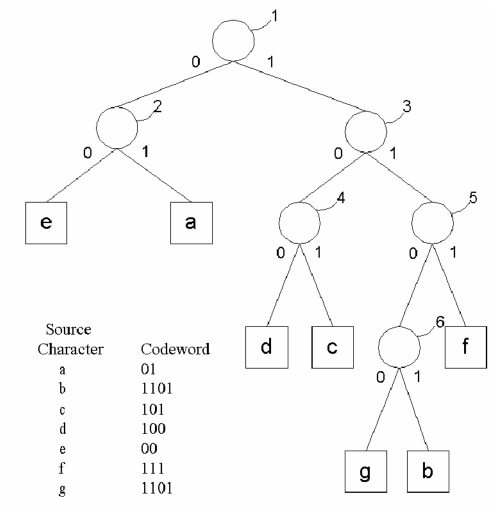
3.2 Guarding against chosen plaintext attacks

Similar to Jones' algorithm [6], this algorithm is vulnerable to chosen plaintext attacks [1]. To guard against chosen plaintext attacks, two different randomly shuffled probability tables are created from the same probability distribution of permissible characters. Then bits of the encryption key are used to determine which table should be used to divide the current interval into smaller intervals. When the 1st source character is encoded, the 1st bit of the encryption key is checked. A value of 0 indicates that the 1st table is to be used to divide the current interval. A value of 1 indicates the other table is to be used. Similarly, when the 2nd source character is encoded, the 2nd bit of the encryption key will be used, etc. When bits of the encryption are exhausted, the 1st bit will be used again.

4. Adding a Pseudo Random Shuffle to a Huffman Coding

Huffman coding is a compression algorithm introduced by David Huffman in 1952. The basic idea behind Huffman coding is to construct a tree, called a Huffman tree, in which each character has it's own branch determining its code.

There are two types of Huffman coding: static and adaptive. In a static Huffman coding, the Huffman tree stays the same in the entire coding process. In an adaptive Huffman coding, the Huffman tree changes according to the data processed. For further discussion about static and adaptive Huffman coding, refer to [2, 3, 4, 5, 7, 9].



Source Character	Codeword
a	01
b	1101
c	101
d	100
e	00
f	111
g	1101

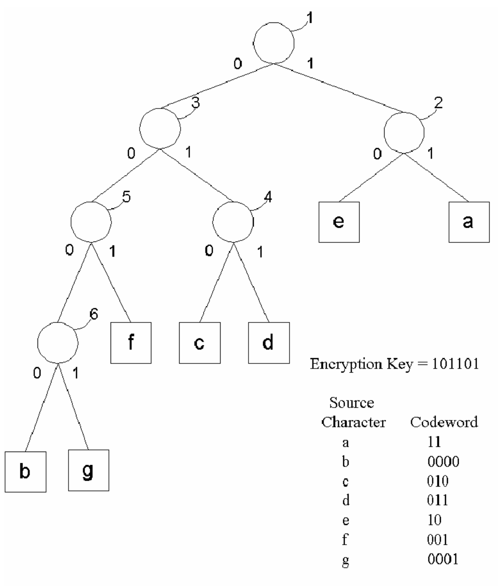
Fig. 8 An example Huffman tree

Once the Huffman tree is built, regardless of its type, the encoding process is identical. The codeword for each source character is the sequence of labels along the path from the root to the leave node representing that character. For example, in FIG. 8, the codeword for ‘a’ is ‘01’, ‘b’ is ‘1101’, etc.

4.1 The algorithm

The essential idea of concealing information in the process of a Huffman coding is to use an encryption key to shuffle the Huffman tree before the encoding process. Without the encryption key, the Huffman tree cannot be shuffled in the same way and thus the decompression cannot be done properly. Consequently, the original information cannot be revealed.

To shuffle a Huffman tree, the interior nodes i.e. nodes with 2 children, are first numbered. There are many ways of numbering these interior nodes. For example, by performing a queue traversal on the Huffman tree, the interior nodes can be numbered in the top-down, left-right fashion. In FIG. 8, the labeling of the interior nodes shows the top-down, left-right numbering of the interior nodes.



Encryption Key = 101101

Source Character	Codeword
a	11
b	0000
c	010
d	011
e	10
f	001
g	0001

Fig. 9 Shuffled Huffman tree

Afterward bits of the encryption key are associated with the interior nodes according to the numbering; interior node 1 is associated with the first bit of the encryption key, interior node 2 is associated with the second bit of the encryption key, etc. Finally, of each interior node that has a corresponding encryption bit of 1, the left child is swapped with the right child. In FIG. 9, the encryption key used is “101101”. Thus, the two children of interior nodes 1, 3, 4, and 6 are swapped. After the shuffling, the codeword of permissible characters are changed dramatically and cannot be decoded without an identically shuffled Huffman tree.

4.2 Guarding against chosen plaintext attacks

In order to guard against chosen plaintext attacks, two extra steps are added to the algorithm.

First, the pseudo random number generator is used to generate a sequence of m random bits, where m is an integer bigger than the height of the Huffman tree, i.e. the length of the longest codeword. Then, the mathematical bit-wise exclusive or operation will be performed on the random sequence and the first m output bits of the Huffman coding.

Second, a node swapping scheme is added to the algorithm. In this scheme, interior nodes are associated with bits of the encryption key according to the order of the interior nodes being visited during the encoding process. After the encoding of a character, the two children of a visited interior node with an associated bit of 1 are swapped. Assume that we want to use the Huffman tree of FIG. 9 to encode the message “ab”. When ‘a’ is encoded, interior nodes 1 and 2 are visited. Interior node 1 is associated with the 1st bit of the encryption key and interior node 2 is associated with the 2nd bit of the encryption key. Since the 1st bit of the encryption key is 1 and the 2nd bit of the encryption key is 0, only the two children of interior node 1 are swapped. FIG. 10 shows the resulting Huffman tree.

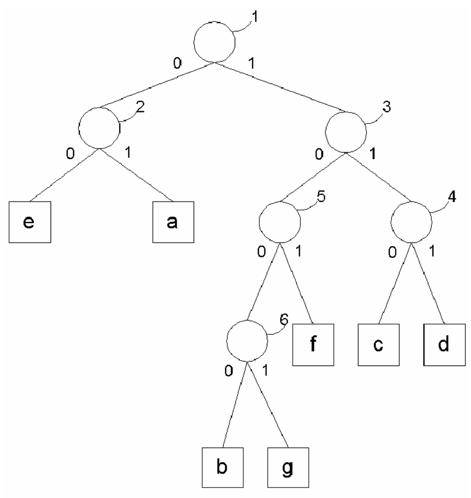


Fig. 10 Shuffled Huffman tree after 'a' is encoded.

When ‘b’ is encoded, interior nodes 1, 3, 5, and 6 are visited and thus are associated with bits 3, 4, 5, and 6 of the encryption key. Since bits 3, 4, and 6 are 1, children of interior nodes 1, 3, and 6 are swapped. FIG. 11 shows the final Huffman tree.

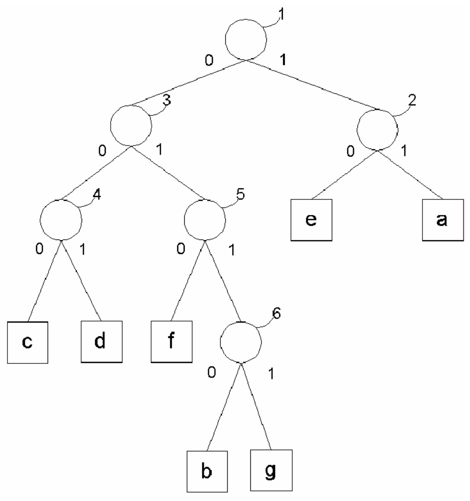


Fig. 11 Shuffled Huffman tree after 'b' is encoded

5. An Immediate Application - Scrambling Multimedia Data

These methods of simultaneous encryption and compression may serve to remedy the security and speed issues that currently concern the multimedia world.

A multimedia encoder such as JPEG, MPEG, H.263, or H.264 usually consists of two parts: a lossy compression part and a lossless compression part. The lossy compression part eliminates redundant or unnecessary information. The lossless compression part uses a traditional compression algorithm such as Huffman coding or arithmetic coding to reduce the size of the data.

A simple and efficient way of scrambling multimedia data is to use the methods described in this paper to convert the lossless compression part of a multimedia encoder into a simultaneous encryption and compression process.

Since chosen plaintext attacks rarely happen in multimedia applications such as video conferencing, video on-demand, video broadcasting, and pay-TV, the extra steps mentioned for guarding against chosen plaintext attacks can be omitted in order to achieve better performance.

6. Conclusions

In this paper, three different methods for converting three different types of compression algorithms into encryption algorithms have been described. An immediate multimedia application of our methods is proposed. Extra steps for guarding against chosen plaintext attacks are also suggested.

In reality, chosen plaintext attacks are mostly applicable to public key encryption algorithms. For secret key encryption algorithms, chosen plaintext attacks rarely happen. Moreover, in most applications, a session key is used to do the encrypting rather than the secret key. Since the session key is changed constantly, plaintext attacks are essentially impossible. Thus, in reality, algorithms described in this paper are the most simple and yet effective simultaneous encryption and compression algorithms.

6.1 Compression efficacy

The three methods presented in this paper maintain the effectiveness of their respective antecedents. The dictionary built in the first method is almost identical to the dictionary built in the original compression algorithm. The second method creates interval tables identical to those created in the original compression algorithm, with the exception of the order in which they occur. In the third method, the length of the codeword of a character is the same as that of the original compression algorithm. These imply that the efficacy of the compressions is not compromised by our methods.

6.2 Encryption strength

As stated in Section 1, our algorithms are much more than substitution ciphers or transposition ciphers. Our algorithms can be classified as stream ciphers. Without the identical initial shuffle, the decompression process, i.e. the decryption process, will be crippled and thus the cryptanalysis can't even be completed. The following two paragraphs discuss the numerical measures of encryption strengths of our algorithms.

In the first two methods, the encryption key is mainly used to initialize the pseudo random number generator. Thus, the strength of the resulting encryption doesn't depend on the length of the encryption key. Instead, the strength depends on the size of an internal variable, called the seed of the pseudo random number generator. Since there are 256! (factorial of 256) different permutations for 8-bit characters, the maximum size of the seed of the pseudo random number generator could be as big as $\log_2 256!$ This is equivalent to a 2048-bit encryption key.

The encryption strength of the method of combining a random shuffle with a Huffman coding depends on the length of the encryption key. Since a Huffman tree can have at most 255 interior nodes, the maximum effective key length of the third method is 255.

6.3 Time efficiency

Excluding those extra steps for guarding against chosen plaintext attacks, the only overhead of methods described in this paper is the CPU time needed for shuffling a dictionary, a probability table, or a Huffman tree before a compression process. This overhead is a fixed cost, independent of the size of the data to be compressed and encrypted.

The CPU time required by the extra steps for guarding against chosen plaintext attacks is also cheap comparing with traditional encryption algorithms such as DES and RCA. In arithmetic coding, the cost of the extra step is proportional to the size of the data to be compressed. In the Huffman coding, the cost of the extra step is proportional to the number of interior nodes processed, which is the same as the number of compressed output bits.

Furthermore, the encryption algorithms resulted from methods of this paper are classified as stream ciphers. Therefore, in the case that the compression is adaptive, there is no need to re-shuffle the interval table or the Huffman tree after the re-computation of probabilities of characters.

Thus, methods described in this paper provide prudent and time saving approaches to simultaneous data compression and encryption.

7. References

[1] H.A. Bergen, J.M. Hogan, A chosen plaintext attack on an adaptive arithmetic coding compression algorithm, in: Computers and Security, vol. 12, 1993, pp. 157-167.

[2] G.V. Cormack, R.N. Horspool, Algorithms for Adaptive Huffman Codes, in: Inform. Process. Lett. 18, 3 (Mar.), 1984, pp. 159-165.

[3] N. Faller, An adaptive system for data compression, in: Record of the 7th Asilomar Conference on Circuits, Systems and Computers. Naval Postgraduate School, Monterey, Calif., Nov. 1973, pp. 593-597.

[4] R.G. Gallager, Variations on a theme by Huffman, in: IEEE Trans. Inf. Theory, 24, 6 (Nov.), 1978, 668-674.

[5] D.A. Huffman, A Method for the Construction of Minimum-Redundancy Codes, in: Proc. IRE, 40, 9 (Sept.), 1952, pp. 1098-1101.

[6] C.B. Jones, An efficient Coding system for long source sequences, in IEEE Trans. Inf. Theory, vol IT-27, 1984, pp. 280-291.

[7] D.E. Knuth, Dynamic Huffman Coding, in: J. Algorithms, 6, 2 (June), 1985, pp. 163-180.

[8] A. Moffat, R.M. Neal, I.H. Witten, Arithmetic coding revisited, in: ACM Transactions on Information Systems, vol. 16, 1995, 256-294.

[9] J.S. Vitter, Design and analysis of dynamic Huffman codes, in: J. ACM, 34, 4 (Oct.), 1987, 825-845.

[10] T.A. Welch, A technique for high-performance data compression, in: Computer 17, 6 (June) 1984, 8-19.

[11] I.H. Witten, R.M. Neal, R.J. Cleary, Arithmetic coding for data compression, in: Communications of the ACM, vol. 30, 1987, pp. 520-540.

[12] J. Ziv, A. Lampel, A universal algorithm for sequential data compression, in: IEEE Trans. Inf. Theory 23, 3 (May) 1977, pp. 337-343.