Experiment No.: 01

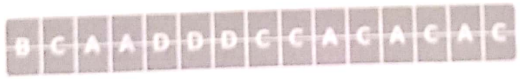
Experiment Name: Explain and implementation of Huffman Codes

Huffman Coding

Huffman coding is a lossless data encoding algorithm. The process behind its scheme includes sorting numerical values from a set in order of their frequency The least frequent numbers are gradually eliminated via the Huffman tree, which adds the two lowest frequencies from the sorted list in every new "branch." The sum is then positioned above the two eliminated lower frequency values, and replaces them in the new sorted list. Each time a new branch is created, it moves the general direction of the tree either to the right (for higher values) or the left (for lower values). When the sorted list is exhausted and the tree is complete, the final value is zero if the tree ended on a left number, or it is one if it ended on the right. This is a method of reducing complex code into simpler sequences and is common in video encoding.

How Huffman Coding works?

Suppose the string below is to be sent over a network.



Initial string

Each character occupies 8 bits. There are a total of 15 characters in the above string.

Thus, a total of 8 * 15 = 120 bits are required to send this string.

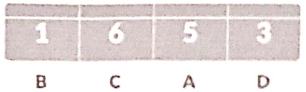
Using the Huffman Coding technique, we can compress the string to a smaller size.

Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.

Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

Huffman Coding prevents any ambiguity in the decoding process using the concept of **prefix code** ie. a code associated with a character should not be present in the prefix of any other code. The tree created above helps in maintaining the property. Huffman coding is done with the help of the following steps.

1. Calculate the frequency of each character in the string.

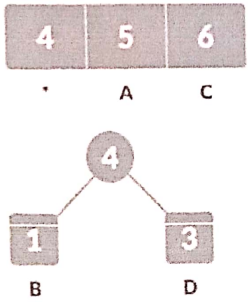


Frequency of string

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q.

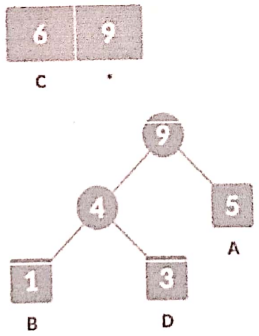
Characters sorted according to the frequency

3. Make each unique character as a leaf node.
4. Create an empty node z. Assign the minimum frequency to the left child of z and assign the second minimum frequency to the right child of z. Set the value of the z as the sum of the above two minimum frequencies.

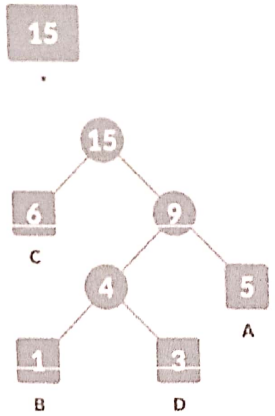


Getting the sum of the least numbers

5. Remove these two minimum frequencies from Q and add the sum into the list of frequencies (* denote the internal nodes in the figure above).
6. Insert node z into the tree.
7. Repeat steps 3 to 5 for all the characters.

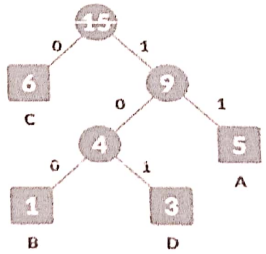


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Assign 0 to the left edge and 1 to the right edge

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

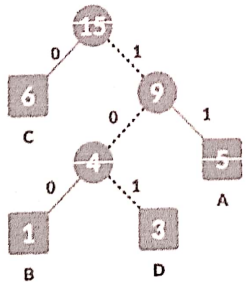
Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits

Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to 32 + 15 + 28 = 75.

Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



Decoding

Experiment No.: 02

Experiment Name: Explain and implementation of Convolutional Coding

Errors and Error Correcting Codes

Errors occurs when bits get corrupted while transmission over the computer network, due to interference and network problems.

Error-correcting codes (ECC) are a sequence of numbers generated by specific algorithms for detecting and removing errors in data that has been transmitted over noisy channels. Error correcting codes ascertain the exact number of bits that has been corrupted and the location of the corrupted bits, within the limitations in algorithm.

ECCs can be broadly categorized into two types, block codes and convolution codes.

Binary Convolutional Coding

In convolutional codes, the message comprises of data streams of arbitrary length and a sequence of output bits are generated by the sliding application of Boolean functions to the data stream.

In block codes, the data comprises of a block of data of a definite length. However, in convolutional codes, the input data bits are not divided into block but are instead fed as streams of data bits, which convolve to output bits based upon the logic function of the encoder. Also, unlike block codes, where the output codeword is dependent only on the present inputs, in convolutional codes, output stream depends not only the present input bits but also only previous input bits stored in memory.

Convolutional codes were first introduced in 1955, by Elias. After that, there were many interim researches by many mathematicians. In 1973, Viterbi developed an algorithm for maximum likelihood decoding scheme, called Viterbi scheme that lead to modern convolutional codes.

Encoding by Convolutional Codes

For generating a convolutional code, the information is passed sequentially through a linear finite-state shift register. The shift register comprises of (-bit) stages and Boolean function generators.

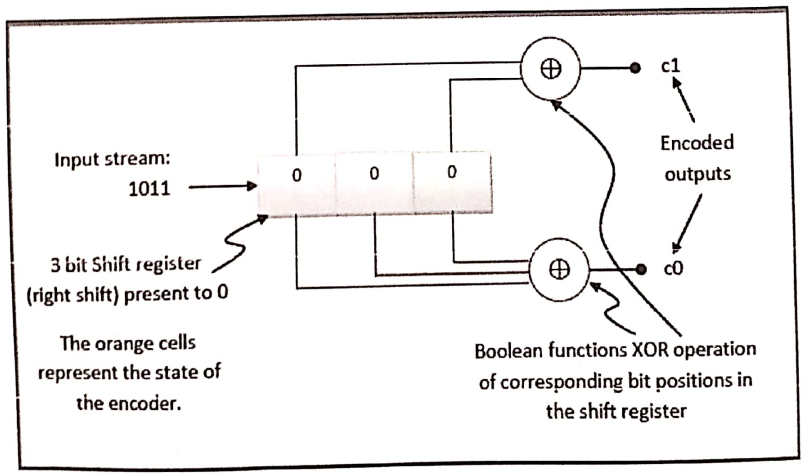
A convolutional code can be represented as (n,k, K) where

- *k* is the number of bits shifted into the encoder at one time. Generally, *k = 1.*
- n is the number of encoder output bits corresponding to k information bits.
- The code-rate, $R_c = k/n$.
- The encoder memory, a shift register of size k, is the constraint length.
- *n* is a function of the present input bits and the contents of *K*.
- The state of the encoder is given by the value of *(K - 1)* bits.

Example of Generating a Convolutional Code

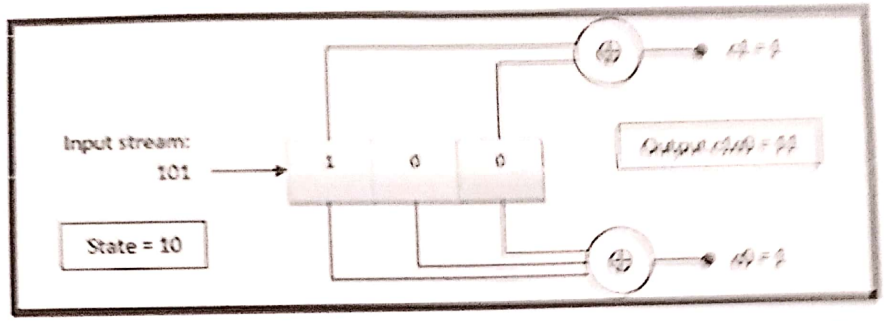
Let us consider a convolutional encoder with *k = 1, n = 2* and K = 3.

The code-rate, $R_c = k/n = 1/2$.

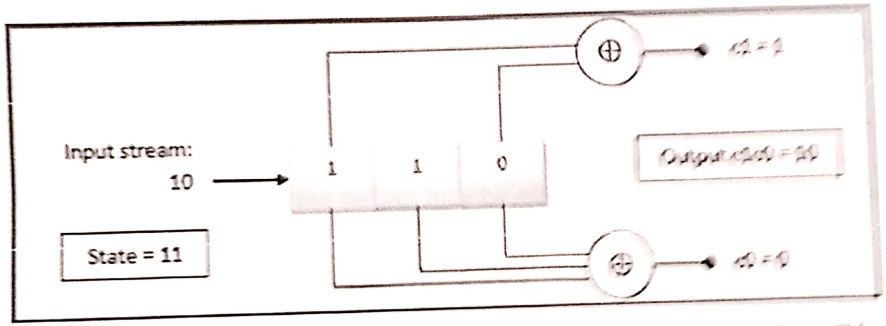


The input string is streamed from right to left into the encoder.

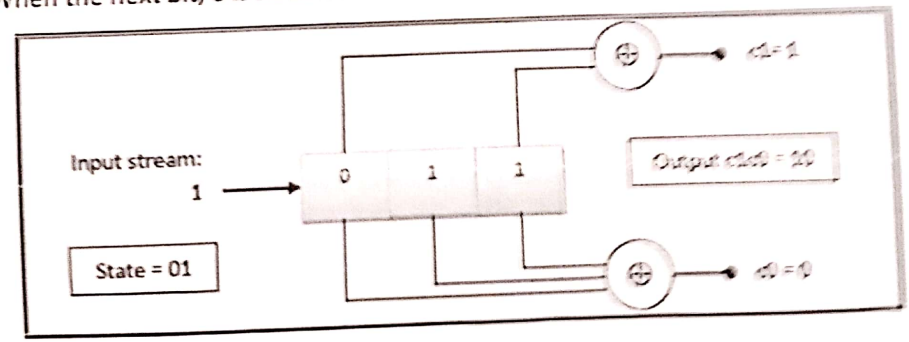
When the first bit, 1, is streamed in the encoder, the contents of encoder will be –



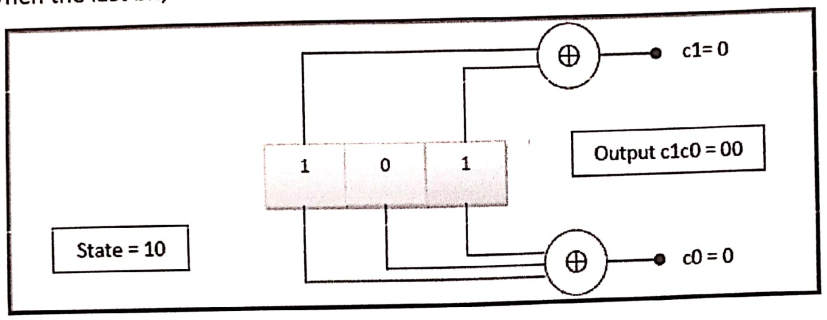
When the next bit, 1 is streamed in the encoder, the contents of encoder will be –



When the next bit, 0 is streamed in the encoder, the contents of encoder will be –



When the last bit, 1 is streamed in the encoder, the contents of encoder will be –



Representing Convolution Encoder with State Transition Diagram and State Table

From the above example, we can see that any particular binary convolutional encoder is associated with a set of binary inputs, a set of binary outputs and a set of states. The transitions and the output may be effectively represented by a state transition diagram and a state table.

For the binary convolution encoder given in the example –

The set of inputs = {0, 1}

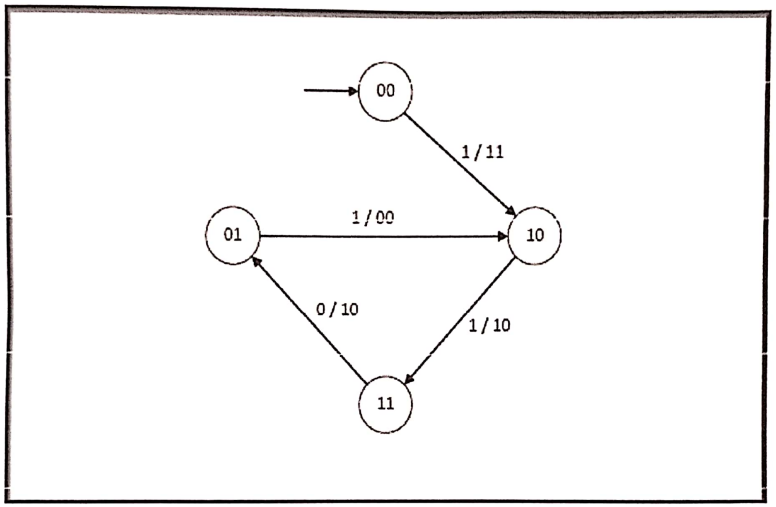
The set of outputs = { 00, 10, 11}

The set of states = { 00, 01, 10, 11}

We can see that in the initial state, 00, when the input 1 was given, the next state became 10 and the corresponding output was 11. In this state 10, when the input 1 was given, the next state was 11 and the encoder outputs were 10. In the same manner we get the other transitions. When this is tabulated, we get the state transition table as follows –

Present State	Next State		Outputs	
	Input = 0	Input = 1	Input = 0	Input = 1
00	-	10	-	11
01	-	10	-	00
10	-	11	-	10
11	01	-	10	-

The corresponding state transition diagram will be –



Modulo-2 Arithmetic

In modulo 2 arithmetics, 1+1 ≡ 0 mod 2, 1+0 ≡ 1 mod 2 and 0+0 ≡ 0 mod 2, which coincide with bit-XOR, i.e. 1⊕1=0, 1⊕0=1 0⊕0=0. Therefore for binary polynomials, addition is simply bit-by-bit XOR. Also, in modulo 2 arithmetics, -1 ≡ 1 mod 2, so the result of subtraction of elements is the same as addition. For example:

- $(x^2+x+1) + (x+1) = x^2+2x+2$, since 2 ≡ 0 mod 2 the final result is x^2. It can also be computed as 111⊕011=100. 100 is the bit string representation of x^2.
- $(x^2+x+1) - (x+1) = x^2$

Multiplication of binary polynomials can be implemented as simple bit-shift and XOR. For example:

- $(x^2+x+1)*(x^2+1) = x^4+x^3+2x^2+x+1$. The final result is x^4+x^3+x+1 after reduction modulo 2. It can also be computed as $111*101=11100 \oplus 111=11011$, which is exactly the bit string representation of x^4+x^3+x+1.

$$
\begin{array}{r}
111 \\
\times\ 101 \\
\hline
111 \\
0000 \\
11100 \\
\hline
11011
\end{array}
$$

Experiment No.: 03

Experiment Name: Explain and implementation of Lempel-Ziv Code

Lempel-Ziv Codes

The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and optionally in PDF and TIFF. Unix's 'compress' command, among other uses. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress and is used in the GIF image format.

The Idea relies on reoccurring patterns to save data space. LZW is the foremost technique for general-purpose data compression due to its simplicity and versatility. It is the basis of many PC utilities that claim to "double the capacity of your hard drive".

Lempel-Ziv Encoding (PSEUDOCODE)

```
1   Initialize table with single character strings
2   P = first input character
3   WHILE not end of input stream
4       C = next input character
5       IF P + C is in the string table
6         P = P + C
7       ELSE
8         output the code for P
9       add P + C to the string table
10        P = C
11      END WHILE
12  output code for P
```

Experiment No.: 04

Experiment Name: Explain and implementation of Hamming Code

Hamming Codes

Hamming code is used to detect and correct the error in the transmitted data. So, it is an error detection and correction code. It was originally invented by *Richard W. Hamming* in the year 1950. Hamming codes detect 1-bit and 2-bit errors.

While transmitting the message, it is encoded with the redundant bits. The redundant bits are the extra bits that are placed at certain locations of the data bits to detect the error. At the receiver end, the code is decoded to detect errors and the original message is received.

So before transmitting, the sender has to encode the message with the redundant bits. It involves three steps, as described below.

Encoding the message with hamming code

Selecting the number of redundant bits

The hamming code uses the number of redundant bits depending on the number of information bits in the message.

Let n be the number of information or data bits, then the number of redundant bits P is determined from the following formula,

For example, if 4-bit information is to be transmitted, then n=4. The number of redundant bits is determined by the trial and error method.

Let P=2, we get,

The above equation implies 4 not greater than or equal to 7. So let's choose another value of P=3.

Now, the equation satisfies the condition. So number of redundant bits, P=3.
In this way, the number of redundant bits is selected for the number of information bits to be transmitted.

Choosing the location of redundant bits

For the above example, the number of data bits n=4, and the number of redundant bits P=3. So the message consists of 7 bits in total that are to be coded. Let the rightmost bit be designated as bit 1, the next successive bit as bit 2 and so on.

The seven bits are bit 7, bit 6, bit 5, bit 4, bit 3, bit 2, bit 1.

In this, the redundant bits are placed at the positions that are numbered corresponding to the power of 2, i.e., 1, 2, 4, 8,... Thus the locations of data bit and redundant bit are *D4, D3, D2, P3, D1, P2, P1.*

Assigning the values to redundant bits

Now it is time to assign bit value to the redundant bits in the formed hamming code group. The assigned bits are called a parity bit.

Each parity bit will check certain other bits in the total code group. It is one with the bit location table, as shown below.

Bit Location	7	6	5	4	3	2	1
Bit designation	D4	D3	D2	**P3**	D1	**P2**	**P1**
Binary representation	111	110	101	100	011	010	001
Information / Data bits	D4	D3	D2		D1		
Parity bits				**P3**		**P2**	**P1**

Parity bit P1 covers all data bits in positions whose binary representation has 1 in the least significant position(001, 011, 101, 111, etc.). Thus P1 checks the bit in locations 1, 3, 5, 7, 9, 11, etc..

Parity bit P2 covers all data bits in positions whose binary representation has 1 in the second least significant position(010, 011, 110, 111, etc.). Thus P2 checks the bit in locations 2, 3, 6, 7, etc.

Parity bit P3 covers all data bits in positions whose binary representation has 1 in the third least significant position(100, 101, 110, 111, etc.). Thus P3 checks the bit in locations 4, 5, 6, 7, etc.

Each parity bit checks the corresponding bit locations and assign the bit value as 1 or 0, so as to make the number of 1s as even for even parity and odd for odd parity.

Example problem 1

Encode a binary word 11001 into the even parity hamming code.

Given, number of data bits, n =5.

To find the number of redundant bits,

Let us try P=4.

The equation is satisfied and so 4 redundant bits are selected.

So, total code bit = n+P = 9

The redundant bits are placed at bit positions 1, 2, 4 and 8.

Construct the bit location table.

Bit Location	9	8	7	6	5	4	3	2	1
Bit designation	D5	**P4**	D4	D3	D2	**P3**	D1	**P2**	**P1**
Binary representation	1001	1000	0111	0110	0101	0100	0011	0010	0001
Information bits	1		1	0	0		1		
Parity bits		1				1		0	1

To determine the parity bits

For P1: Bit locations 3, 5, 7 and 9 have three 1s. To have even parity, P1 must be 1.

For P2: Bit locations 3, 6, 7 have two 1s. To have even parity, P2 must be 0.

For P3: Bit locations 5, 6, 7 have one 1s. To have even parity, P3 must be 1.

For P4: Bit locations 8, 9 have one 1s. To have even parity, P2 must be 1.

Thus the encoded 9-bit hamming code is 111001101.

How to detect and correct the error in the hamming code?

After receiving the encoded message, each parity bit along with its corresponding group of bits are checked for proper parity. While checking, the correct result of individual parity is marked as 0 and the wrong result is marked as 1.

After checking all the parity bits, a binary word is formed taking the result bits for P1 as LSB. So formed binary word gives the bit location, where there is an error.

If the formed binary word has 0 bits, then there is no error in the message.

Example problem 2

Let us assume the even parity hamming code from the above example (111001101) is transmitted and the received code is (110001101). Now from the received code, let us detect and correct the error.

To detect the error, let us construct the bit location table.

Bit Location	9	8	7	6	5	4	3	2	1
Bit designation	D5	**P4**	D4	D3	D2	**P3**	D1	**P2**	**P1**
Binary representation	1001	1000	0111	0110	0101	0100	0011	0010	0001
Received code	1	1	0	0	0	1	1	0	1

Checking the parity bits

For P1 : Check the locations 1, 3, 5, 7, 9. There is three 1s in this group, which is wrong for even parity. Hence the bit value for P1 is 1.

For P2 : Check the locations 2, 3, 6, 7. There is one 1 in this group, which is wrong for even parity. Hence the bit value for P2 is 1.

For P3 : Check the locations 3, 5, 6, 7. There is one 1 in this group, which is wrong for even parity. Hence the bit value for P3 is 1.

For P4 : Check the locations 8, 9. There are two 1s in this group, which is correct for even parity. Hence the bit value for P4 is 0.

The resultant binary word is 0111. It corresponds to the bit location 7 in the above table. The error is detected in the data bit D4. The error is 0 and it should be changed to 1. **Thus the corrected code is 111001101.**