AIM

To implement Huffman, Lempel-Ziv, and Hamming encoders and decoders and to study the effect of introducing bit-errors into the Hamming encoded bitstream with varying probability values on error correction capabilities.

OBJECTIVES

1. To understand the theory and principles of source and channel coding.
2. To evaluate the performance of each algorithm in terms of compression ratio, computational complexity, and error correction capability.
3. To implement the functions in MATLAB.

THEORY

HUFFMAN CODE

Huffman Coding is a widely-used data compression technique that is commonly used in digital communication and storage systems. The purpose of Huffman Coding is to reduce the size of data without losing any important information. The algorithm is named after its inventor, David Huffman, who developed it in 1952.

The key concept behind Huffman Coding is to use shorter binary codes to represent frequently occurring symbols, while reserving longer codes for symbols that occur less frequently. This approach exploits the statistical redundancy in the data, as most natural languages and signals tend to exhibit a degree of repetition and regularity. By encoding the most common symbols with the shortest binary codes, Huffman Coding can achieve a significant reduction in the number of bits required to represent the original data stream.

Huffman Encoding

There are mainly two major steps involved in obtaining the Huffman Code for each unique character

1. Build a Huffman Tree from the given stream of data (taking only unique characters).
2. Traverse the Huffman Tree which is formed and assign codes to characters and encode the given string using these Huffman codes.

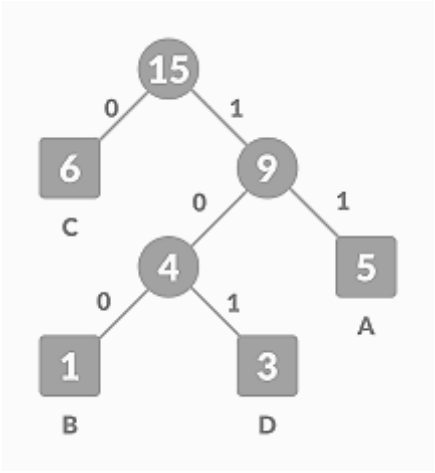

Fig 1. Sample Huffman tree

Huffman Coding is a Greedy Algorithm, which means that it uses a local optimization strategy to achieve the best possible compression result at each step. The algorithm builds a binary tree from the input data, with each leaf node representing a unique symbol in the data stream. The weight of each node is proportional to the frequency of the symbol it represents. The algorithm then assigns a binary code to each symbol by traversing the tree from the root to the leaf node corresponding to that symbol, with a "0" bit indicating a branch and a "1" bit indicating the other branch. The binary code for each symbol is then determined by concatenating the bits in the path from the root to the corresponding leaf node.

Huffman Decoding

The Huffman encoder outputs the encoded message along with the coding scheme. This encoding scheme is sent before transmission so that the receiver can understand how to decode the received bits. Once the encoded message is received, the receiver needs to use the same coding scheme to decode the message and retrieve the original data.

Huffman codes are prefix codes, which means that no code word is a prefix of any other code word. This property makes it easier to decode the encoded message, as there is no need to look ahead to determine the end of a code word. The decoder can simply read in the bits one by one until a code word is found, at which point it can be decoded to its corresponding symbol. This is possible because the prefix property ensures that no other code word could start with the same sequence of bits.

Advantages of Huffman coding

1. Huffman Coding is used to reduce the size and to improve the speed of transmission.
2. it is fast and efficient, as it can process large amounts of data with minimal computational overhead.
3. It is used to compress files. Many multimedia storages like JPEG, PNG, and MP3 use Huffman encoding for compressing the files.

Disadvantages of Huffman Coding

1. It is not considered to be optimal unless all probabilities/frequencies of the characters are the negative powers of 2.
2. It is not always optimal, meaning that it may not always achieve the smallest possible code length for a given set of symbols.
3. The encoding scheme used by Huffman Coding must be sent along with the encoded data, which can increase the overall size of the message.

LEMPEL ZIV WELCH CODE

Lempel-Ziv Welch Code, also known as LZW algorithm, is a widely used lossless compression technique developed by Abraham Lempel and Jacob Ziv, and later published by Terry Welch in 1984. LZW algorithm compresses data by identifying repeating patterns of data and encoding them with shorter codes. This results in a compressed output which can be easily decompressed to recover the original data.

Lempel Ziv Encoding

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string W in the dictionary that matches the current input.
3. Emit the dictionary index for W along with the next bit to output and remove W with next bit from the input.
4. Add W followed by the next symbol in the input to the dictionary.

By building a translation table that maps input strings to integer codes, the LZW algorithm achieves compression by replacing longer input strings with shorter integer codes. This approach allows for efficient compression of data with repeating patterns or strings, as longer strings can be represented by shorter codes.

The encoder also sends the code size (size for each encoded word) so that the decoder can initialize its dictionary with the same set of code words and to use the correct code size for decoding each subsequent code word.

Lempel Ziv Decoding

1. Using the code size, the input is split into code words.
2. Initialize the dictionary to contain all strings of length one.
3. Code is split to suffix and prefix, with prefix containing the last bit only.
4. Suffix of the code is looked up in dictionary to find the correct bits and prefix is attached to its end to form the output bits
5. Decoded message word is added as a new entry

The LZW algorithm works by dynamically creating the decoding dictionary during the decoding process, rather than relying on a pre-existing dictionary to be transmitted along with the encoded message. The dictionary is initialized with a set of standard codes, and new codes are added to it as the algorithm processes the input data.

Advantages of Lempel Ziv coding

1. The LZW algorithm works more efficiently for files containing lots of repetitive data.
2. Another important characteristic of the LZW compression technique is that it builds the encoding and decoding table on the go and does not require any prior information about the input.

Disadvantages of Lempel Ziv Coding

1. Files that do not contain any repetitive data at all cannot be compressed much.
2. LZW is a compression algorithm that is good at text files but not as good at other types of files.

HAMMING CODES

Hamming Code is the set of error-correction codes that are used to detect the error in the transmitted data and to correct the error. It can be applied to data units of any length. Hamming code is generally used for detecting as well as correcting single-bit errors. Generally, the source encodes the message by adding a few redundant bits during

the transmission. These redundant bits are generated and inserted at certain positions in the message. They are mostly used for error detection and correction process.

If enough parity data is added, it enables forward error correction (FEC), where errors can be automatically fixed when read back. FEC can increase the data transmission rate for noisy channels by reducing the amount of necessary retransmits.

The amount of parity data added to Hamming code is given by the formula

$\mathbf{2^p \geq d + p + 1}$, where p is the number of parity bits and d is the number of data bits.

For example, if you wanted to transmit 4 data bits, the formula would be $2^3 \geq 4 + 3 + 1$, so 3 parity bits are required.

The minimum Hamming distance determines the error detection/correction capabilities of a code. Hamming (7,4) have a minimum Hamming distance of 3 ($d_{min} = 3$), so they can detect 2-bit errors ($d_{min} - 1 = 2$) or correct 1-bit errors ($(d_{min} - 1) / 2 = 1$)

Hamming encoding

So, each 4 bits of data is sent as 7-bit code words. For 4-bit data, the parity bits are given by

$$\mathbf{P_1 = d_1 + d_2 + d_4}$$

$$\mathbf{P_2 = d_1 + d_3 + d_4}$$

$$\mathbf{P_3 = d_2 + d_3 + d_4}$$

The code words can also be found out by multiplying the Generator matrix G with message, M.

$$\text{Generator Matrix, } \mathbf{G} = \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

$$\text{Code word, } \mathbf{C = M \times G}$$

Position	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
Data/Parity	Parity 1	Parity 2	Data 1	Parity 3	Data 2	Data 3	Data 4

Fig 2. Format of an encoded code word

Hamming decoding

The Hamming code can be decoded easily using a syndrome decoder. Syndromes are obtained by multiplying received code word with transpose of parity matrix.

$$\text{Parity matrix, } \mathbf{H} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{bmatrix}$$

$$\text{Syndrome, } \mathbf{S} = \mathbf{C} \times \mathbf{H}^{\mathbf{T}}$$

If the syndrome is 000, it means that there are no errors. And the bits from location 3,5,6,7 are the data bits. For any 1-bit error, the location of bit error is found using the syndrome and that bit is flipped to get the correct code.

Hamming code can thereby detect and correct any single-bit error. If two data bits were flipped, it could detect it but not correct the error, either resulting in incorrect decoding or could request the sender to retransmit the word again. Higher bit errors could even go undetected.

Advantages of Hamming coding

1. Hamming codes can detect and correct single-bit errors, which ensures that data transmission is accurate and reliable.
2. Hamming codes are more efficient than other error-correcting codes, such as the Reed-Solomon codes, in terms of both storage and computation requirements.
3. Hamming codes are relatively simple to implement and can be easily understood and used by engineers and technicians without extensive knowledge of coding theory.

Disadvantages of Hamming coding

1. Hamming codes (7,4) can only detect and correct single-bit errors. If multiple bits are flipped, the code can detect the error but cannot correct it, requiring retransmission.
2. Hamming codes require additional bits to be added to the data being transmitted, which increases the amount of data that must be transmitted, reducing the efficiency of the data transmission.
3. Hamming codes are best suited for applications that require low levels of error correction. For applications that require higher levels of error correction, more advanced error-ccorrecting codes such as Reed-Solomon codes or Turbo codes are more appropriate.

BIT ERRORS

Bit errors can occur during the transmission of digital signals due to various reasons, such as noise, interference, attenuation, and distortion. A bit error refers to the change or corruption of a bit in the transmitted signal, resulting in an incorrect signal being received at the receiving end.

If these errors are not corrected, they can accumulate and cause the receiver to misinterpret the data. Therefore, reducing bit errors is necessary to ensure the integrity and accuracy of transmitted data. This is achieved through error detection and correction techniques such as parity checking, checksums, and forward error correction.

EXPERIMENTAL SETUP

The Dataset used here is a part of '**world192.txt**' which is a subset of The Canterbury Corpus, which is used as a benchmark to evaluate lossless compression methods. The input size for each run was varied.

SOURCE CODE

Huffman Encoder

This function implements a Huffman encoder, which takes an input message as a string, and encodes it using variable-length codes based on the frequency of occurrence of each character in the input message and returns an encoded string along with the encoding table.

The code first converts the input string to its ASCII representation, and then to binary. It then creates a frequency map for each character in the input message, and constructs a Huffman tree based on the frequency of occurrence of each character. The tree is used to generate the variable-length codes for each character, with more frequent characters having shorter codes.

The encoded message is generated by concatenating the variable-length codes for each character in the input message. The encoding table, which maps each character to its corresponding variable-length code, is also generated and returned by the function.

```
function [encodingtable, encoded_message] = huffman_encoder(input)

ascii = uint8(input);

binary = dec2bin(ascii,8);

freq_map = containers.Map('KeyType','double','ValueType','double');
for i = 1:length(ascii)
   char = ascii(i);
   if isKey(freq_map, char)
      freq_map(char) = freq_map(char) + 1;
   else
      freq_map(char) = 1;
   end
end

keys = freq_map.keys;
values = freq_map.values;
```

```
%disp('Letter-frequency mappings:');
% for i = 1:length(keys)
%     fprintf('%d -> %d\n', keys{i}, values{i});
% end


letters = cell2mat(freq_map.keys);
frequencies = cell2mat(freq_map.values);
[sorted_frequencies, sorted_indices] = sort(frequencies);
sorted_letters = letters(sorted_indices);
% disp('Letters in terms of freq:');
% disp(sorted_letters);


num_nodes = length(sorted_frequencies);
nodes = struct('letter', {}, 'freq', {}, 'left', {}, 'right', {});
for i = 1:num_nodes
    nodes(i).letter = sorted_letters(i);
    nodes(i).freq = sorted_frequencies(i);
end


while num_nodes > 1
    left_node = nodes(1);
    right_node = nodes(2);
    new_node = struct('letter', [], 'freq', left_node.freq + right_node.freq, 'left', left_node,
'right', right_node);

    nodes(1:2) = [];
    nodes = [nodes new_node];

    frequencies = [nodes.freq];
    [sorted_frequencies, sorted_indices] = sort(frequencies);
    nodes = nodes(sorted_indices);

    num_nodes = length(sorted_frequencies);
end
```

```
codes = containers.Map('KeyType','double','ValueType','char');
traverse_tree(nodes, '', codes);
keys = codes.keys;
values = codes.values;
% disp('Huffman codes:');
% for i = 1:length(keys)
%     fprintf('%d -> %s\n', keys{i}, values{i});
% end


encoded_message = '';
for i = 1:length(input)
   letter = input(i);
   encoded_message = [encoded_message codes(double(letter))];
end

encodingtable = '';
for i = 1:length(keys)
   ascii_val = dec2bin(keys{i}, 8);
   code = values{i};
   encodingtable = [encodingtable sprintf('%s %s\n', ascii_val, code)];
end

end
```

Huffman Decoder

This function implements Huffman decoding by taking in the encoded message and the encoding table as input. The encoding table is used to create a decoding dictionary that maps the binary code to the corresponding ASCII value. The function then uses the decoding dictionary to decode the encoded message bit by bit. It reads the bits one by one and appends them to a string variable 'curr' until it finds a match in the decoding dictionary. Once a match is found, the corresponding ASCII character is appended to the decoded message, and 'curr' is reset. The decoded message is written to a file named 'decoded.txt,' and the file name is returned.

```
function [decoded_message,file] = huffman_decoder(encoded_message ,encodingtable)

lines = strsplit(encodingtable, '\n');

decoding_dict = containers.Map;
for i = 1:length(lines)
    if isempty(lines{i})
        continue;
    end

    line_parts = strsplit(lines{i}, ' ');
    ascii_val = line_parts{1};
    code = line_parts{2};
    code_decimal = bin2dec(code);
    decoding_dict(code) = char(bin2dec(ascii_val));
end

decoded_message = '';
curr = '';
for i = 1:length(encoded_message)
    curr = [curr encoded_message(i)];
    if isKey(decoding_dict, curr)
        decoded_message = [decoded_message decoding_dict(curr)];
        curr = '';
    end
end

file = fopen('decoded.txt','w+');
fprintf(file,'%s',decoded_message);
fclose(file);
file = 'decoded.txt';
end
```

Lempel Ziv Encoder

This function implements Lempel Ziv encoder by taking the message string as input. the input string is converted to its ASCII representation and then to a binary string. The function then creates a dictionary to hold unique strings and their positions from the binary string.

Next, the function iterates through the binary string, until a string not in the dictionary is reached. When such a match is found, the function splits the match into suffix and prefix, with prefix containing only last bit. The suffix is replaced by the value from the dictionary corresponding to suffix. The code word now contains the unique position code along with the prefix. The new code gets added to the output and also to dictionary along with its position.

The output of the function is the encoded message and the number of bits required to represent each code in the dictionary.

```
function [encoded_message, numbits]  = lempelziv_encoder(input)

ascii = uint8(input);
input = dec2bin(ascii, 8)';
input = input(:)';

dictionary = containers.Map;
i = 1;
c = 1;
code_words = {};
while i <= length(input)
   match = input(i);
   while isKey(dictionary, match) && i<length(input)
      i=i+1;
      match = [match input(i)];
   end

   if ~isKey(dictionary, match)
      dictionary(match) = c;
      c = c + 1;
   end
```

```
    code_words{end+1} = match;
    match='';
    i=i+1;

end

keys = dictionary.keys;
values = dictionary.values;
length(dictionary);
numbits = ceil(log2(length(dictionary)+1)) + 1;
dictionary('')=0;

encoded_message = '';

for i=1:length(code_words)
    match = code_words{i};
    code = match(1:end-1);
    encoded_message = strcat(encoded_message, dec2bin(dictionary(code), numbits-1));
    encoded_message = strcat(encoded_message, match(end));
end

end
```

Lempel Ziv Decoder

This function implements the Lempel-Ziv decoding algorithm. It takes in an encoded message and the number of bits used to encode each code word.

It first converts the encoded message into individual code words and initializes a dictionary with the first code word. It takes the code word splits into suffix and prefix based on last bit. The suffix is then replaced by the value from the dictionary corresponding to suffix. And the original message is formed which is then appended to output stream and also added to the dictionary along with its position.

Finally, it converts the binary decoded message to ASCII characters, writes them to a file, and returns the file name.

```
function [decoded_message,file]  = lempelziv_decoder(encoded_message,numbits)

j = 1;
code_words = {};
code_words{j} = '';
for i = 1:length(encoded_message)
   code_words{j} = [code_words{j} encoded_message(i)];
   if mod(i, numbits) == 0 && i<length(encoded_message)
      j = j + 1;
      code_words{j} = '';
   end
end

dictionary = containers.Map;
dictionary(dec2bin(1,numbits-1)) = code_words{1}(end);
decoded_message = code_words{1}(end);

for i = 2:length(code_words)
   code = code_words{i};
   if code(1:end-1) == dec2bin(0,numbits-1)
      dictionary(dec2bin(i,numbits-1)) = code(end);
      decoded_message = [decoded_message code(end)];

   else
      msg = [dictionary(char(code(1:end-1))) code(end)];
      dictionary(dec2bin(i,numbits-1)) =char(msg);
      decoded_message = [decoded_message msg];
   end
end
ascii = bin2dec(reshape(decoded_message, 8, [])');
decoded_message = char(ascii);

file = fopen('decoded.txt','w+');
fprintf(file,'%s',decoded_message);
fclose(file);
file = 'decoded.txt';
end
```

Hamming Encoder

This function implements a Hamming code encoder. It takes an input message in the form of a binary string and pads zeros to ensure its length is a multiple of 4. The function then performs matrix multiplication between the message and a generator matrix G, resulting in a new binary string of length 7 for each block of 4 bits in the original message. The resulting bit stream is the concatenation of all of these 7-bit code words.

```
function encoded_bitstream = hamming_encoder(encoded_message)

    pad = mod(4 - mod(numel(encoded_message), 4), 4);
    encoded_message = [encoded_message, repmat('0', 1, pad)];
    G = [1 1 1 0 0 0 0;
         1 0 0 1 1 0 0;
         0 1 0 1 0 1 0;
         1 1 0 1 0 0 1];

    codes = {};
    for i = 1:length(encoded_message)/4
      codes{i} = encoded_message((i-1)*4+1:i*4);
    end

    encoded_bitstream = [];
    for i = 1:length(codes)
        word = [];
        for j = 1:4
            word = [word str2num(codes{i}(j))];
        end
        encoded_bitstream = [encoded_bitstream mod(word * G, 2)];
    end
end
```

Hamming Decoder

This function implements a Hamming decoder that takes an encoded bitstream and returns the decoded bitstream. The function first creates a matrix 'H' that represents the parity-check matrix for the Hamming code. Then, the function splits the encoded bitstream into 7-bit code words and calculates the syndromes for each code word using

the parity-check matrix. If the syndrome is zero, the code word is error-free, and the function returns the decoded word. If the syndrome is not zero, the function uses the syndrome to determine the position of the error and corrects it. Finally, the function returns the decoded bitstream.

```
function decoded_bitstream = hamming_decoder(encoded_bitstream)

    H = [1 0 1 0 1 0 1;
         0 1 1 0 0 1 1;
         0 0 0 1 1 1 1];

    syndromes = {};
    for i = 1:length(encoded_bitstream)/7
        word = encoded_bitstream((i-1)*7+1:i*7);
        syndrome = mod(word * H', 2);
        syndromes{i} = syndrome;
    end

    decoded_bitstream = [];
    for i = 1:length(syndromes)
        if sum(syndromes{i}) == 0
            decoded_word = encoded_bitstream((i-1)*7+1:i*7);
            decoded_bitstream = [decoded_bitstream, decoded_word([3, 5, 6, 7])];
        else
            error_pos = bin2dec(num2str(fliplr(syndromes{i})));
            corrupted_word = encoded_bitstream((i-1)*7+1:i*7);
            if corrupted_word(error_pos) == '0'
                corrupted_word(error_pos) = '1';
            else
                corrupted_word(error_pos) = '0';
            end
            decoded_word = corrupted_word([3, 5, 6, 7]);
            decoded_bitstream = [decoded_bitstream decoded_word];
        end
    end
    if isnumeric(decoded_bitstream)
```

```
        decoded_bitstream = char(decoded_bitstream + '0');
    end
end
```

Bit Error

The function biterror takes in an input binary stream and a probability p. It generates a new bitstream by flipping each bit with probability p using a random number generator. This simulates error in transmission. The new bitstream with errors is returned as a string of '0's and '1's.

```
function output_bitstream = biterror(input,p)
    input = char(input + '0');
    output_bitstream = '';

    for i = 1:length(input)
        error_prob = rand();

        if error_prob < p
            if input(i) == '0'
                output_bitstream =[output_bitstream '1'];
            else
                output_bitstream = [output_bitstream '0'];
            end
        else
            output_bitstream =[output_bitstream input(i)];
        end
    end
end
```

One Bit Error

The onebiterror function takes in a bit stream as input and outputs another bit stream after inverting one bit in each 7-bit code with probability p, thus creating a bit error in each code word.

For each 7-bit code, the function generates a random number between 0 and 1 to determine whether an error will be introduced or not. If the error probability is less than p, the function randomly selects a bit position in the current 7-bit code and flips the bit at that position. Otherwise, it simply appends the current 7-bit code to the output bitstream as is.

```
function output_bitstream = onebiterror(input,p)
   input = char(input + '0');
   output_bitstream = '';

   for i = 1:length(input)/7
      current_code = input((i-1)*7+1:i*7);
      error_prob = rand();

      if error_prob < p
         pos = randi(7);
         if current_code(pos) == '0'
            current_code(pos) = '1';
         else
            current_code(pos) = '0';
         end
      end
      output_bitstream = [output_bitstream current_code];

   end

end
```

RESULT

The results of the encoding and decoding process were verified through several iterations, confirming the successful implementation of the algorithms. A comparison between the original input files and the decoded output files showed that they were identical, indicating that the decoding process was accurate and reliable. These findings demonstrate the effectiveness and efficiency of the Huffman and Lempel-Ziv algorithms in source coding and their potential for further development and application in various fields.

In accordance with the methodology employed in this study, a series of iterations were conducted and the resulting outcomes were recorded in the following tabular format:

NO OF INPUT BITS (bits)	ENCODING/ DECODING TYPE	ENCODED OUTPUT SIZE (bits)	COMPR-ESSION RATIO	CORRECTLY DECODED
989360 (123.67 KB)	Huffman	618867 (77.35 KB)	1.5987	YES
	Lempel-Ziv	878939 (109.86 KB)	1.1256	YES
1978720 (247.34 KB)	Huffman	1245996 (155.74 KB)	1.5881	YES
	Lempel-Ziv	1676201 (209.52 KB)	1.1805	YES
3957440 (494.68 KB)	Huffman	2495496 (209.52 KB)	1.5858	YES
	Lempel-Ziv	3184310 (209.52 KB)	1.2428	YES
9893600 (1.23 MB)	Huffman	6239724 (0.77 MB)	1.5856	YES
	Lempel-Ziv	7337725 (0.91 MB)	1.3483	YES

The results indicate that Huffman encoding achieved almost consistent compression ratio across different file sizes. However, Lempel-Ziv compression ratio exhibited an increasing trend with increasing file size and could potentially surpass Huffman's compression ratio at a certain point.

Sample output from an iteration.

```
sample Input:
****The Project Gutenberg Edition of THE WORLD FACTBOOK 1992****

******This file should be named world92.zip or world92

Huffman Coding

The input and decoded files are identical.
Number of bits for non-encoded string: 791488
Number of bits for Huffman encoded string: 495926
Compression ratio: 1.596

Lempel Ziv Coding

The input and decoded files are identical.
Number of bits for non-encoded string: 791488
Number of bits for Lempel Ziv encoded string: 729407
Compression ratio: 1.0851

Hamming coding
The input and decoded files are identical.

Introducing random bit error
The input and decoded files are different.

Introducing one bit error in each code word
The input and decoded files are identical.
```

Fig 3. MATLAB output

The encoded bits from Huffman encoding were passed through a Hamming encoder and errors were intentionally introduced to evaluate the Hamming decoder's ability to correct them. The Hamming decoder was successful in correcting single bit errors in each code word, even when the error was introduced with a probability of 1. However, the decoder was unable to correct two or more-bit errors, even with a lower probability of 0.1. This led to an incorrect decoding of the text file at the end. These findings suggest that while the hamming code is effective in correcting single bit errors, it is not suitable for correcting multiple bit errors. This highlights the importance of carefully choosing the appropriate error-correcting code for a particular application to ensure reliable data transmission and decoding.

CONCLUSION

In conclusion, the implementation and analysis of Huffman, Hamming, and Lempel-Ziv encoding and decoding in MATLAB have revealed valuable insights into the performance and efficiency of these algorithms for channel and source coding and also for compression and storage.

Each algorithm has its strengths and limitations, and their effectiveness varies depending on the type of data and application. Huffman was computationally faster than Lempel Ziv but the results show that at higher file sizes, Lempel Ziv could surpass Huffman compression ratio. The Hamming code could only correct one-bit errors and wrongly decoded code words with more errors. The potential implications and significance of these findings lie in the importance of efficient data compression for the storage and transmission of information, and the need for further research and development in this field.

Limitations of the study include the MATLAB implementation and the need for more extensive testing and evaluation with larger and more diverse datasets. Future works can explore the potential for other programming languages or platforms, as well as further improvements and modifications to optimize the performance and efficiency of these algorithms. Overall, this study provides a foundation for continued exploration and innovation into the field of channel and source coding.

REFERENCES

[1] Sayood, Khalid, 2012, Introduction to data compression, 4 th ed. Morgan Kaufmann, Elsevier.

[2] Cipta, Subhan Panji and Rizki Nur Ikhsan. “Implementation Of Huffman Bigram Compression Algorithm In .Txt Extension Files.” (2017).

[3] H. C. Kotze and G. J. Kuhn, "An evaluation of the Lempel-Ziv-Welch data compression algorithm," COMSIG 1989 Proceedings: Southern African Conference on Communications and Signal Processing, Stellenbosch, South Africa, 1989, pp. 65-69, doi: 10.1109/COMSIG.1989.129018.

[4] W. Rurik and A. Mazumdar, "Hamming codes as error-reducing codes," 2016 IEEE Information Theory Workshop (ITW), Cambridge, UK, 2016, pp. 404-408, doi: 10.1109/ITW.2016.7606865.