EE69210: Machine Learning for Signal Processing Laboratory
Department of Electrical Engineering, Indian Institute of Technology, Kharagpur

Lossless Compression

Anirvan Krishna | Roll no. 21EE38002

Keywords: Huffman Coding, Lossless Compression

Grading Rubric

	Tick the best applicable per row			Points
	Below Expectations	Lacking in Some	Meets all Expectations	
Completeness of the report				
Organization of the report (5 pts)				
Quality of figures (5 pts)				
Building the min-heap (20 pts)				
Generating the code book and com- pressed bit stream (25 pts)				
Ability to decode the Huffman tree from its in-order traversal represen- tation (20 pts)				
Ability to recover the dataset (25 pts)				
			TOTAL (100 pts)	

1. Theory: Huffman coding

Huffman coding is a lossless data compression algorithm that assigns variable-length binary codes to input symbols based on their frequencies. It is an optimal prefix coding technique that ensures that no code is a prefix of another, allowing for efficient and unambiguous decoding.

Steps of Huffman coding

(1) **Frequency Analysis:** Compute the frequency of each symbol in the input data.

(2) **Building a Min-Heap:** Create a priority queue (min-heap) where each node represents a symbol with its frequency.

(3) **Constructing the huffman tree:**
- Extract two nodes with the lowest frequencies.
- Merge them into a new node with their combined frequency.
- Repeat until only one node remains, forming a binary tree.

(4) **Generating Huffman Codes:** Assign '0' to the left branch and '1' to the right branch, traversing the tree to generate unique binary codes for each symbol.

(5) **Encoding and Decoding:** The input data is replaced with its Huffman codes for compression. The decoding follows the tree structure to reconstruct the original data.

1.1 Frequency analysis

The process starts with identifying the number of unique symbols in $\mathbf{X} \in \mathbb{R}^{D\times N}$, say, represented as $\mathbf{U} = [\mathbf{u}_0, \mathbf{u}_1, \ldots, \mathbf{u}_j, \ldots, \mathbf{u}_{k-1}] \in \mathbb{R}^{D\times k}$. For a text corpus consisting of ASCII characters, say $\mathbf{X} = \{'h','e','l','l','o'\} \in \mathbb{R}^{8\times 5}$. Here, $\mathbf{U} = \{'h','e','l','o'\}$. The frequency of any symbol is determined by its probability:

$$p(\mathbf{u}_k;\mathbf{X}) = \frac{\texttt{count}(\mathbf{u}_k;\mathbf{X})}{N} \tag{1}$$

1.2 Data structure for a huffman node

For the construction of the Huffman tree, we use a priority queue (min heap). The class `HuffmanNode` is defined with the given structure. A given member of the class `HuffmanNode` stores information about the *symbol* it stores, the frequency (probability) of the occurrence of that symbol. `left` and `right` store the left and right children of the given node, where the probability of occurrence of the left node is lower than the right. The method `__lt__()` is the *less-than* operator that allows a priority queue to compare two nodes based on their frequency.

```python
class HuffmanNode:
    def __init__(self, symbol, freq):
        self.symbol = symbol
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq
```

1.3 Construction of Huffman tree

After extracting the symbol probabilities, the construction of the Huffman tree is done by following the given algorithm.

Algorithm 1 Build Huffman Tree

```
[Require:]A frequency dictionary $\mathscr{F}$ where keys are symbols and values are their occurrence probabilities.
[Ensure:]A Huffman tree with a root node.
Initialize a min-heap $\mathscr{H}$ with nodes created from $\mathscr{F}$
Convert $\mathscr{H}$ into a priority queue using heapify
while $|\mathscr{H}| > 1$ do
    Extract two nodes with least frequencies: $L \leftarrow \texttt{heappop}(\mathscr{H})$, $R \leftarrow \texttt{heappop}(\mathscr{H})$
    Create a new internal node $N$ with frequency $\texttt{freq}(N) = \texttt{freq}(L) + \texttt{freq}(R)$
    Set $L$ as the left child of $N$ and $R$ as the right child of $N$
    Insert $N$ back into $\mathscr{H}$ using heappush
end while
return The last remaining node in $\mathscr{H}$ (root of the Huffman Tree)
```

1.4 Retrieving the Huffman codes for each symbol

Given the Huffman tree, the bit codes corresponding to each symbol are assigned using the following algorithm. From a given node, if we traverse left, a 0 gets appended to the bit code; if we traverse right, a 1 gets appended. Once, we have the bit-code for each symbol, the original symbol can be replaced by its corresponding *huffman code* to return the *encoded dataset*.

Algorithm 2 Get Huffman Codes

```
[Require:]A Huffman tree node $N$, an initial code string $C$, and an empty dictionary $\mathscr{D}$ to store symbol-code mappings.
[Ensure:]A dictionary $\mathscr{D}$ containing Huffman codes for each symbol.
if $N =$ None then
    return
end if
if $N$ is a leaf node (i.e., contains a symbol) then
    Store the code: $\mathscr{D}[N.\text{symbol}] \leftarrow C$
    return
end if
Recursively traverse the left subtree with $C +' 0'$
BUILD HUFFMAN CODES($N$.left,$C +' 0'$,$\mathscr{D}$)
Recursively traverse the right subtree with $C +' 1'$
BUILD HUFFMAN CODES($N$.right,$C +' 1'$,$\mathscr{D}$)
```

1.5 Encoding the Huffman tree and generation of code-book

After converting the dataset into encoded format using huffman code, the compressed file can be shared with different users. For extracting the original file from the compressed version, the receiver needs a *code-book*, a look-up table where the codes corresponding to each symbol can be found. This code-book also needs to be transmitted along with the encoded dataset. The same tree can be decoded by inverting this procedure

Algorithm 3 Encode Huffman Tree

```
1: [Require:]A Huffman tree node $N$
2: [Ensure:]A bit sequence representing the Huffman tree structure
3: function ENCODETREE($N$)
4:     if $N$ = None then
5:         return an empty bit sequence
6:     end if
7:     Initialize an empty bit sequence $B$
8:     if $N$ is a leaf node (i.e., contains a symbol) then
9:         Append bit 1 to $B$ (indicating a leaf node)
10:        Append the symbol $N$.symbol as an $N$-bit encoded character to $B$
11:    else
12:        Append bit 0 to $B$ (indicating an internal node)
13:        Append ENCODETREE($N$.left) to $B$
14:        Append ENCODETREE($N$.right) to $B$
15:    end if
16:    return $B$
17: end function
```

Algorithm 4 Decode Huffman Tree

```
1: [Require:]A bit sequence $B$ and an index $i$ (starting position)
2: [Ensure:]A reconstructed Huffman tree and the updated index
3: function DECODETREE($B, i$)
4:     if $B[i] = 1$ then                                        ▷ Leaf node
5:         $i \leftarrow i + 1$
6:         Extract the next 8 bits from $B[i : i+8]$ and decode it as a symbol $S$
7:         $i \leftarrow i + 8$
8:         return a new leaf node $N(S, 0)$ and updated index $i$
9:     end if
10:    $i \leftarrow i + 1$                                      ▷ Internal node
11:    $(L, i) \leftarrow$ DECODETREE($B, i$)                     ▷ Decode left child
12:    $(R, i) \leftarrow$ DECODETREE($B, i$)                     ▷ Decode right child
13:    Create a new internal node $N(\text{None}, 0)$ with left child $L$ and right child $R$
14:    return $N, i$
15: end function
```

1.6 Creating encoder and decoder

The original dataset is compressed into a .huf file with the following structure.

- First 2 bytes: The first 2 bytes (say N) of `.huf` file contain the length of the bitstream corresponding to the Huffman tree.
- Next N bits: The next N bits contain the Huffman tree encoded in bit format using the `EncodeTree` function mentioned earlier.
- Remaining bits: The remaining bits of the `.huf` file is the bit-encoded format of the dataset.

Using the bits corresponding to the Huffman tree and the `DecodeTree` function, we can retrieve the code corresponding to each of the symbols and use it to reconstruct the dataset from its bitstream. The algorithm for constructing the `Decoder` are as follows:

Algorithm 5 Huffman Decoding Algorithm

```
[Require:]A compressed file $F$ containing Huffman tree structure and encoded bitstream
[Ensure:]Decoded text $X$
function DECODER($F$)
    Open file $F$ in binary read mode
    Read $traversal\_length$ (tree traversal length) as a 2-byte integer
    Read $traversal$ (encoded Huffman tree structure)
    Trim $traversal$ to its actual length
    Read $padding\_bits$ (padding length) as a 1-byte integer
    Read the encoded bitstream $X\_bitstream$
    Remove $padding\_bits$ from $X\_bitstream$
    Reconstruct Huffman tree $T = \text{DECODETREE}(traversal, 0)$
    Generate Huffman codes $\mathscr{C} = \text{GETHUFFMANCODES}(T)$
    Construct reverse mapping $D = \{c : s \mid s, c \in \mathscr{C}\}$
    Initialize empty string $current\_code$
    Initialize empty list $decoded\_text$
    for each bit $b$ in $X\_bitstream$ do
        Append $b$ to $current\_code$
        if $current\_code \in D$ then
            Append $D[current\_code]$ to $decoded\_text$
            Reset $current\_code$ to empty string
        end if
    end for
    Convert $decoded\_text$ list to string
    return $decoded\_text$
end function
```

1.7 Compression Factor

We define compression factor as follows $\text{CF} = \frac{\texttt{SizeOf}\ (\text{Original file})}{\texttt{SizeOf}\ (\texttt{.huf}\text{file})}$

2. Experiments:

In this section, we conduct experiments on two different datasets and try to encode and decode the datasets using Huffman coding. We observe the compression factor for different datasets and verify that the mean squared error between the original and reconstructed datasets is zero.

2.1 Dataset-1: Text corpus

In this case we generate a lowercase alphabet and numeric string of a given length N consisting of random characters without blank spaces or line breaks or special characters and we repeat this for multiple values of N. An example of such a corpus for $N = 50$ is:

$$\mathbf{X} = \texttt{'843u6xcvm8sskavg10gq1di3540yzmvxdfbcpgnkmllbvpuhad'}. \tag{2}$$

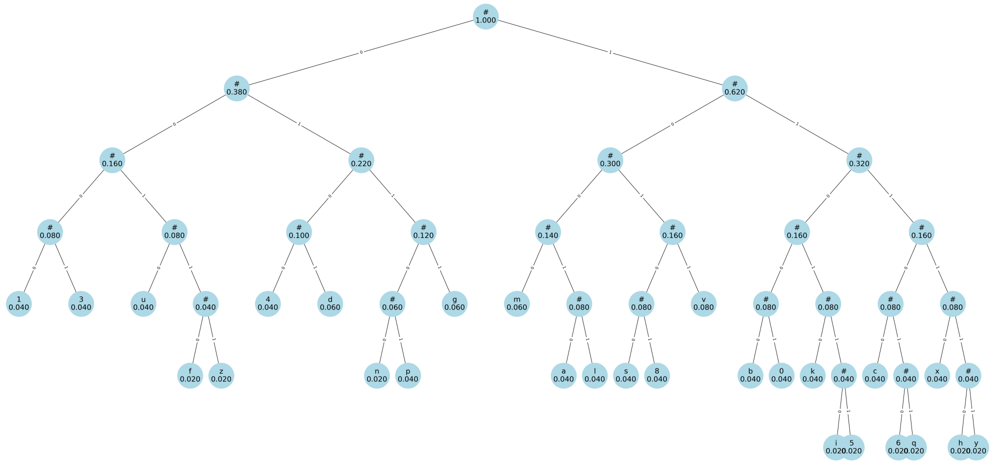

Fig. 1. Huffman tree corresponding to the above text corpus

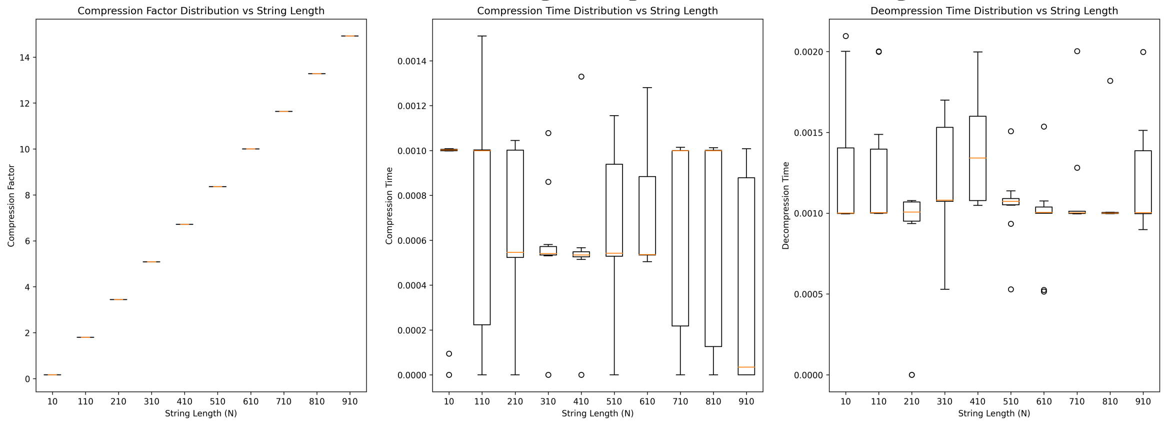

Fig. 2. Distribution of compression factor, compression time and decompression time for text corpus of different size

For all these cases shown in the plot above, we successfully reconstruct the corpus using the `Decoder` function from the `.huf` file with a mean squared error (MSE) = 0 for reconstruction.

2.2 Dataset-2: Olivetti faces

The olivetti faces dataset consists of 400 images of shape 64×64 i.e., $\mathbf{X} \in \mathbb{R}^{400\times64\times64}$. Each image $\mathbf{x}_i$ can be flattened to get $\mathbf{x}_i \in \mathbb{R}^{4096}$. Therefore, we have a 4096 length vector of integers in range $[0,255]$ if we encode the image using `uint8` (unisigned 8-bit integer). Therefore, $\mathbf{x}_i \in \mathbb{B}^{8\times4096}$. Where $\mathbb{B} = \{0,1\}$. All the images are converted first into a `.bmp` file i.e., `uint8`. Therefore, each pixel value is an unsigned integer in range $[0,255]$. Since, ASCII values also range from 0-255, each pixel can be replaced by its corresponding ASCII character and the previously designed `Encoder` and `Decoder` functions can be applied.

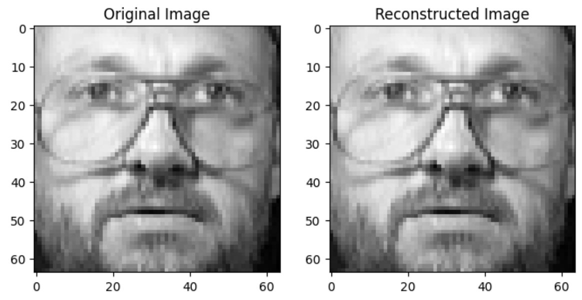

Fig. 3. Comparison of an image from the face dataset and its reconstructed version after conversion. We verify a reconstruction MSE = 0.

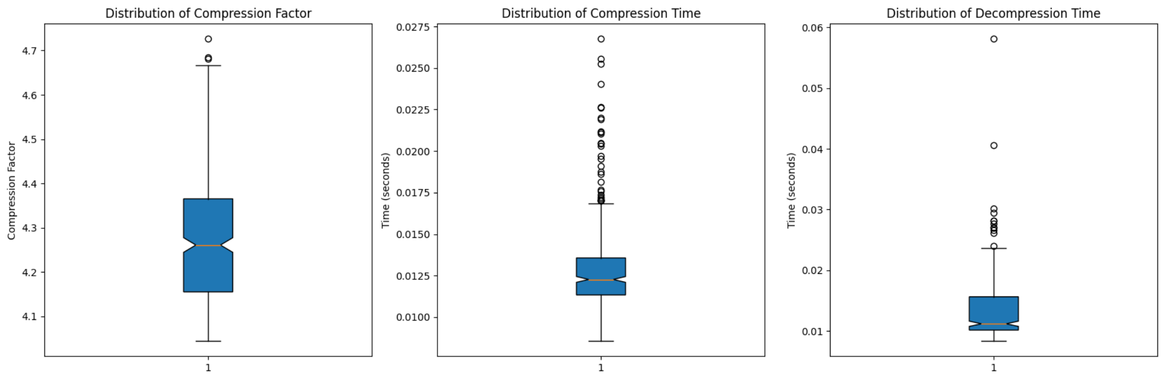

Fig. 4. Distribution of compression factor, compression time and decompression time for different faces from the dataset.

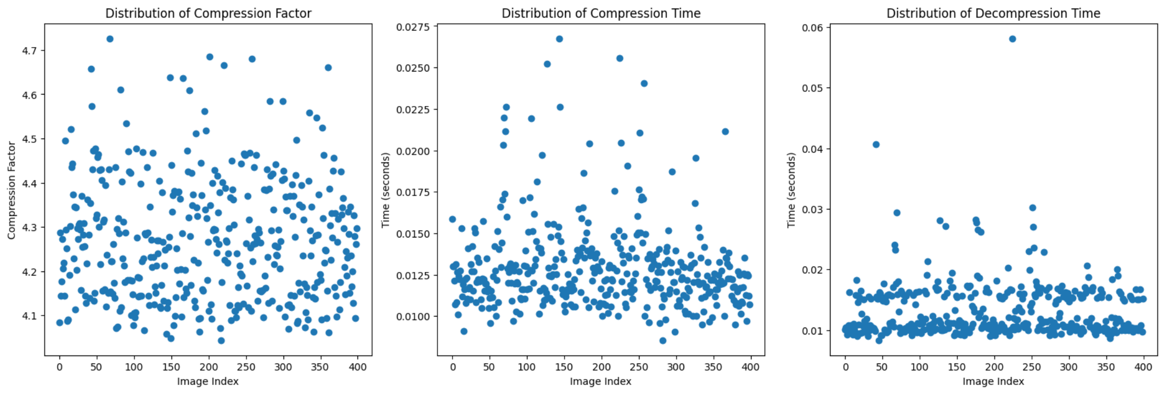

Fig. 5. Compression factor, compression time and decompression time corresponding to different faces from the dataset.