HUFFMAN CODING

Compression & Decompression

OVERVIEW

Huffman Coding is a lossless data compression algorithm where each character/byte in the data is assigned a variable length prefix code. The least frequent character gets the largest code and the most frequent one gets the smallest code.

BUILDING THE TREE

The input is a set of unique characters/bytes, along with their frequency of occurrence in the file. The output is the Huffman Tree.

1. Create a leaf node for each unique character/byte, and build a min heap containing all leaf nodes.

   Min Heap is used as a priority queue. The frequency is used to compare two nodes in the min heap.

2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two extracted nodes frequencies. Make the first extracted node its left child, and the other extracted node its right child. Add this new node to the min heap.
4. Keep repeating step 2 and step 3 until the heap contains only one node. The remaining node is the root node of the Huffman tree.

ENCODING THE TREE

For optimization, we traverse the Huffman tree one time only, generating the codes and encoding the tree at the same time. The following function is called starting at the root of the tree and with an empty string:

TraverseTree(Node node, String code**)**:

if (node.IsLeafNode):

- HuffmanCodes.add(node.data, code)
- Write '1'
- Write node.data (1 byte)

else:

- Write '0'
- **TraverseTree (**node.Left, code + '0'**)**;
- **TraverseTree (**node.Right, code + '1'**)**;

DECODING THE TREE

DecodeNode(String code**)**:

if (Read_Bit == 1):

- HuffmanCodes.add(Read_Byte, code)

else:

- **DecodeNode(**code + '0'**)** //left
- **DecodeNode(**code + '1'**)** //right

ENCODING THE FILE

Each byte from the original file is replaced by its generated Huffman code. This is the body section of the compressed file.

DECODING THE FILE

We read the body section of the compressed file bit-by-bit, adding bits to a buffer. With each bit added, we search our set of Huffman codes for a match of the buffer. When a match is found, we output its corresponding byte and clear the buffer. We keep doing this until we reach the end of the body section.

TIME COMPLEXITY

Building The Tree: O(n log n)

The time complexity of building the Huffman tree is O(n log n), where n is the number of unique characters/bytes in the file. If there are n leaf nodes, extractMin() is called 2*(n–1) times. extractMin() takes O(log n) time as it calls minHeapify(). So, the overall complexity is O(n log n).

Encoding The Tree: O(n)

The time complexity of encoding the Huffman tree is the time of traversing a binary tree a depth-first traversal, which is O(n), where n is the number of nodes in the tree.

Decoding The Tree: O(n)

The time complexity of decoding the Huffman tree is O(n), where n is the number of nodes in the tree.

COMPRESSED FILE FORMAT

Header:

- 3 bits: number of zeros inserted as a padding at the end of the file.
- The encoded tree.

Body:

- Each byte from the original file is replaced by its generated Huffman code.

COMPRESSED FOLDER FORMAT

- 3 bits: number of zeros inserted as a padding at the end of the file.
- For each file in the folder we write a section containing:
    - 3 bits: number of zeros inserted as a padding at the end of this section.
    - 5 bytes: the length of this section (number of bytes).
      (5 bytes gives a very reasonable size limit of 1 TB for the compressed section, it can be increased).
    - The file path inside the folder.
    - The encoded tree.
    - The body: each byte from the original file is replaced by its generated Huffman code.