A BEGINNER-FRIENDLY APPROACH TO HUFFMAN ENCODING

POGĂCEAN PAUL-ANDREI

Abstract. This paper documents the design and implementation of a beginner-friendly Huffman encoder and decoder in C++. The focus lies on practical compression of text files using binary codes based on symbol frequencies. The method uses histograms, binary trees, code tables, bitwise operations, and file I/O techniques for efficient data compression and decompression. We provide a step-by-step description of the encoding and decoding process, discuss implementation details, and present results on small text examples.

1. Introduction

Huffman encoding is a popular lossless compression technique where symbols are encoded into binary sequences of variable length depending on their frequency. Frequent symbols are assigned shorter codes, while rare symbols receive longer ones, resulting in overall reduced file size.

In this work, we define and use several important terms:

- **Histogram**: An array of size 256 where each index represents a symbol, and the value at that index represents the frequency of that symbol in the input file.
- **Huffman Tree**: A binary tree built from the histogram, where paths from the root determine the binary codes assigned to each symbol.
- **Code Table**: A table mapping each symbol to its corresponding binary code derived from the Huffman tree.

We describe a complete implementation of both encoding and decoding using C++, with emphasis on clarity, modularity, and practical file handling.

2. Background

In 1951, David A. Huffman, a graduate student at MIT, devised Huffman encoding while attempting to find the most efficient way to represent symbols using binary codes. His discovery provided an optimal solution for minimizing the average length of codes based on symbol frequency.

Huffman encoding is widely used today in various areas, such as image compression (JPEG), video compression, and data storage formats. The

Date: April 23, 2025.

essential idea is simple: symbols that occur more often should require fewer bits to encode.

This project implements a Huffman encoder and decoder based on the following basic ideas:

- Building a histogram of character frequencies.
- Constructing a Huffman tree using a priority queue.
- Generating a code table from the tree.
- Writing and reading compressed binary files.
- Using bitwise operations for compact bit-level data management.

In the following sections, we detail the methods and structures used in the implementation.

3. Methods

In this section, we describe the design and implementation choices made for the Huffman encoder and decoder. Each major component corresponds to a C++ function or class method, and we include pseudocode for clarity.

3.1. **Building the Histogram.**

The first step in the encoding process is to compute a histogram of symbol frequencies in the input file.

- We read the file byte by byte.
- For each byte (character), we increment its corresponding count in a 256-entry array.
- We ensure that at least two distinct symbols exist by incrementing the counts for symbols 0 and 255.

Pseudocode:

```
function build_histogram(filename):
    initialize histogram[256] to all zeros
    open file for binary reading
    for each byte in file:
        increment histogram[byte]
    increment histogram[0] and histogram[255]
```

3.2. **Building the Huffman Tree.**

Using the histogram, we construct a Huffman tree:

- Each symbol with non-zero frequency becomes a leaf node.
- Nodes are inserted into a priority queue ordered by frequency.
- The two nodes with lowest frequency are repeatedly dequeued, merged into a parent node, and reinserted.
- The process continues until only one node remains—the root.

Pseudocode:

```
function build_huffman_tree(histogram):
    create a priority queue
    for each symbol in histogram:
        if frequency > 0:
            create a new node
            insert node into priority queue
    while queue has more than one node:
        left = dequeue()
        right = dequeue()
        parent = join(left, right)
        enqueue(parent)
    return the last node (root)
```

3.3. **Building the Code Table.** Next, we traverse the Huffman tree to build binary codes for each symbol:

- A left traversal adds a `0` to the current code.
- A right traversal adds a `1` to the current code.
- When a leaf is reached, the current code is saved for that symbol.

Pseudocode:

```
function build_code_table(node, current_code):
    if node is leaf:
        code_table[node.symbol] = current_code
    else:
        append 0 to current_code
        build_code_table(node.left, current_code)
        remove last bit from current_code

        append 1 to current_code
        build_code_table(node.right, current_code)
        remove last bit from current_code
```

3.4. **Writing and Dumping the Tree.** When compressing a file, the Huffman tree structure must be saved to the compressed output:

- We perform a post-order traversal.
- For each leaf, we write 'L' followed by the symbol.
- For each internal node, we write 'I' (with no symbol).

Pseudocode:

```
function dump_tree(node):
    if node is not null:
        dump_tree(node.left)
        dump_tree(node.right)
        if node is leaf:
            write 'L'
            write node.symbol
        else:
            write 'I'
```

3.5. **Encoding the File.** The actual encoding process involves:

- Writing a header (magic number, original file size, tree size).
- Writing the serialized Huffman tree.
- For each input byte, writing its corresponding code bit-by-bit to the output.
- Managing a bit buffer manually to ensure efficient writing.

Pseudocode:

```
function encode_file(input_filename, output_filename):
    build histogram
    build Huffman tree
    build code table
    write header to output
    dump tree to output
    open input file
    for each byte:
        get its code
        write code bits to output
    flush any remaining bits
```

3.6. **Decoding the File.** Decompression works by:

- Reading and validating the header.
- Reconstructing the Huffman tree from the dumped structure.
- Reading the encoded data bit-by-bit and traversing the tree.
- Outputting a symbol whenever a leaf node is reached.

Pseudocode:

```
function decode_file(input_filename, output_filename):
    read header
    rebuild tree from dumped structure
    read compressed bits
    for each bit:
        traverse tree (0 = left, 1 = right)
        if at leaf node:
            output symbol
            return to root
```

3.7. **Bitwise Operations.** Because binary data is handled at the bit level, several bitwise operations are used:

- **Setting a Bit**: `buffer |= (1 << position)`
- **Clearing a Bit**: `buffer &= ~(1 << position)`
- **Checking a Bit**: `(buffer >> position) & 1`

These operations allow us to pack multiple bits into a byte efficiently before writing them to or reading them from the files.

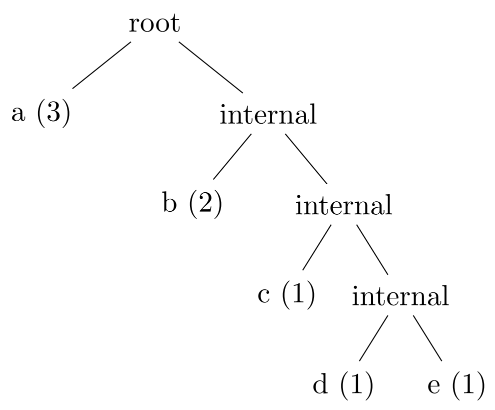

FIGURE 1. More complex Huffman Tree for "aaabbcde"

3.8. **Example Huffman Tree Construction.** Figure 1 shows a more complex example Huffman tree for the string `"aaabbcde"`.

The frequencies are:

- a: 3
- b: 2
- c: 1
- d: 1
- e: 1

From the tree, we derive the following codes:

- a: 0
- b: 10
- c: 110
- d: 1110
- e: 1111

Notice that the most frequent character `a` receives the shortest code, while the less frequent characters `d` and `e` receive longer codes, which aligns with Huffman's principle of minimizing the overall weighted code length.

4. Conclusion

We presented a beginner-friendly design and implementation of Huffman encoding and decoding in C++. Our approach focused on clarity, practical file handling, and efficient bit-level operations. Through step-by-step descriptions and illustrative examples, we hope to make Huffman compression accessible to new learners. Future work could explore optimizations such as adaptive Huffman coding or parallel compression strategies.

References

[1] D. A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the IRE, 1952.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press, 2009.

5. Acknowledgments

Chat-GPT was used in the process of structuring this documentation to rephrase the ideas in a clear and direct manner.