Data Compression and Decompression

Simon GIRARD

May 27, 2025

Abstract

This project implements Huffman coding for file compression and decompression in C++. It supports both text and binary files. The program builds a Huffman tree based on byte frequency, generates prefix codes, compresses the data, and can accurately reconstruct the original file. Deterministic sorting and tie-breaking ensure consistent results. The tool is simple, efficient, and meets the project's core requirements.

1 Design

1.1 Overview

The program performs lossless compression and decompression using Huffman coding. It processes arbitrary binary files by analyzing byte frequencies to build a Huffman tree, which generates prefix codes for efficient encoding. The same tree structure is used during decompression to restore the original data.

1.2 Frequency Analysis and Tree Construction

The program reads the input file byte-by-byte and counts the frequency of each unique byte. Using these frequencies, it constructs a Huffman tree via a priority queue, merging nodes with the lowest frequencies iteratively. Tie-breaking rules in the priority queue comparator ensure deterministic tree construction.

1.3 Code Generation and Encoding

Once the Huffman tree is built, the program traverses it to generate prefix codes for each byte. These codes replace the original bytes in the compressed file, resulting in a smaller output size. The program handles bit-level writing to store codes efficiently. [Zel18]

1.4 Huffman Header Format

The compressed file begins with a custom header containing metadata needed for decompression. This header includes:

- The number of unique bytes.
- Each byte and its frequency, serialized in a fixed format.

This frequency table allows the decompressor to reconstruct the exact same Huffman tree used during compression, ensuring lossless recovery of the original file.

Note that storing both the byte and its frequency could be improved to just storing all frequencies in a predefined order. This would result in better compression for somes files. The current solution is best for files using less than half of the character set, like text.

1.5 Decompression

During decompression, the program reads the header to rebuild the frequency table and Huffman tree. It then decodes the bitstream by traversing the tree according to the encoded bits, outputting the original bytes sequentially.

2 Usage

2.1 Compilation

Compile the program using the following command:

```
g++ -std=c++17 -o HuffmanCompressor main.cpp huffman.cpp logs.cpp
```

2.2 Running the Program

The program supports two modes: compression and decompression.

- **Compression:** `./HuffmanCompressor c inputFile outputFile [verbose]` Compresses `inputFile` into `outputFile`. The optional `verbose` flag enables detailed logging.
- **Decompression:** `./HuffmanCompressor d inputFile outputFile [verbose]` Decompresses `inputFile` (a Huffman compressed file) into `outputFile`. The optional `verbose` flag enables detailed logging.

2.3 Examples

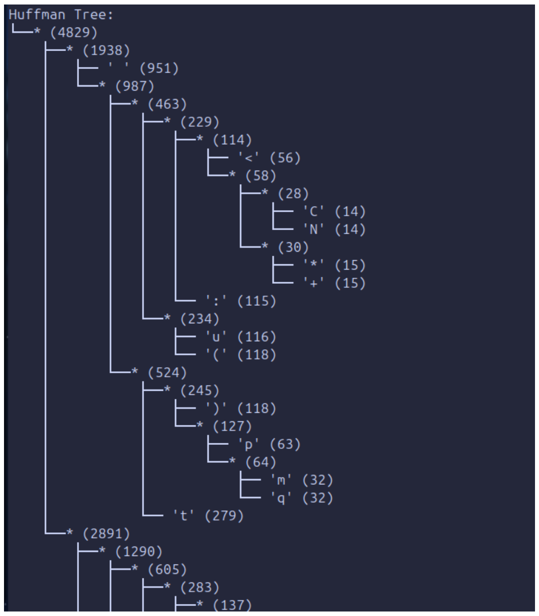
```
./HuffmanCompressor c example.txt example.huf
./HuffmanCompressor d example.huf example_out.txt
./HuffmanCompressor c example.txt example.huf verbose
./HuffmanCompressor c image.png inter.huf && ./HuffmanCompressor d inter.huf image2.png
```

If incorrect arguments are provided, the program outputs usage instructions.

3 Debugging

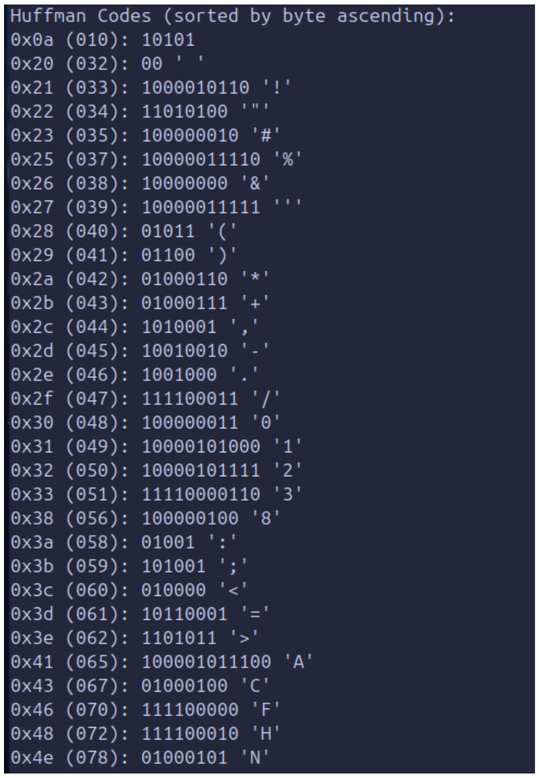
Although the program is designed to work reliably without user debugging, detailed logging has been implemented to facilitate issue detection at every stage of execution. When the `verbose` flag is enabled during compression or decompression, the program outputs its internal state and key information to `stdout` in a fully deterministic manner.

This includes frequency tables, Huffman codes, and tree structures. Users can save these logs and compare the compression and decompression states using text comparison tools such as https://www.diffchecker.com/text-compare/. This approach allows easy identification of inconsistencies or errors by highlighting differences, streamlining troubleshooting without the need to directly debug the program's source code.



```
Huffman Tree:
└──* (4829)
    ├──* (1938)
    │   ├── ' ' (951)
    │   └──* (987)
    │       ├──* (463)
    │       │   ├──* (229)
    │       │   │   ├──* (114)
    │       │   │   │   ├── '<' (56)
    │       │   │   │   └──* (58)
    │       │   │   │       ├──* (28)
    │       │   │   │       │   ├── 'C' (14)
    │       │   │   │       │   └── 'N' (14)
    │       │   │   │       └──* (30)
    │       │   │   │           ├── '*' (15)
    │       │   │   │           └── '+' (15)
    │       │   │   └── ':' (115)
    │       │   └──* (234)
    │       │       ├── 'u' (116)
    │       │       └── '(' (118)
    │       └──* (524)
    │           ├──* (245)
    │           │   ├── ')' (118)
    │           │   └──* (127)
    │           │       ├── 'p' (63)
    │           │       └──* (64)
    │           │           ├── 'm' (32)
    │           │           └── 'q' (32)
    │           └── 't' (279)
    └──* (2891)
        ├──* (1290)
        │   ├──* (605)
        │   │   ├──* (283)
        │   │   │   ├──* (137)
```

Figure 1: Dump of a Huffman Tree



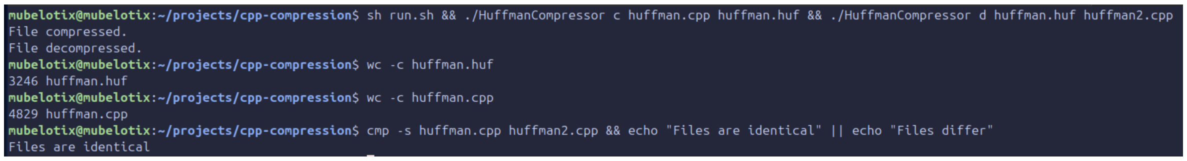
```
Huffman Codes (sorted by byte ascending):
0x0a (010): 10101
0x20 (032): 00 ' '
0x21 (033): 1000010110 '!'
0x22 (034): 11010100 '"'
0x23 (035): 100000010 '#'
0x25 (037): 10000011110 '%'
0x26 (038): 10000000 '&'
0x27 (039): 10000011111 '''
0x28 (040): 01011 '('
0x29 (041): 01100 ')'
0x2a (042): 01000110 '*'
0x2b (043): 01000111 '+'
0x2c (044): 1010001 ','
0x2d (045): 10010010 '-'
0x2e (046): 1001000 '.'
0x2f (047): 111100011 '/'
0x30 (048): 100000011 '0'
0x31 (049): 10000101000 '1'
0x32 (050): 10000101111 '2'
0x33 (051): 11110000110 '3'
0x38 (056): 100000100 '8'
0x3a (058): 01001 ':'
0x3b (059): 101001 ';'
0x3c (060): 010000 '<'
0x3d (061): 10110001 '='
0x3e (062): 1101011 '>'
0x41 (065): 100001011100 'A'
0x43 (067): 01000100 'C'
0x46 (070): 111100000 'F'
0x48 (072): 111100010 'H'
0x4e (078): 01000101 'N'
```

Figure 2: Dump of the code map

4 Results

The Huffman compressor performs reliably on various file types and sizes. As expected, the algorithm achieves better compression on plain text files compared to already compressed or binary files such as images.

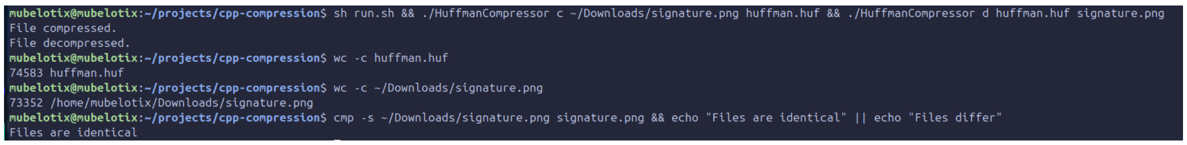
For instance, compressing a text source file (**`huffman.cpp`**) reduced its size from 4829 bytes to 3246 bytes, a reduction of approximately 33%. The decompressed output was verified to be byte-for-byte identical using comparison tools.



```
mubelotix@mubelotix:~/projects/cpp-compression$ sh run.sh && ./HuffmanCompressor c huffman.cpp huffman.huf && ./HuffmanCompressor d huffman.huf huffman2.cpp
File compressed.
File decompressed.
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c huffman.huf
3246 huffman.huf
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c huffman.cpp
4829 huffman.cpp
mubelotix@mubelotix:~/projects/cpp-compression$ cmp -s huffman.cpp huffman2.cpp && echo "Files are identical" || echo "Files differ"
Files are identical
```

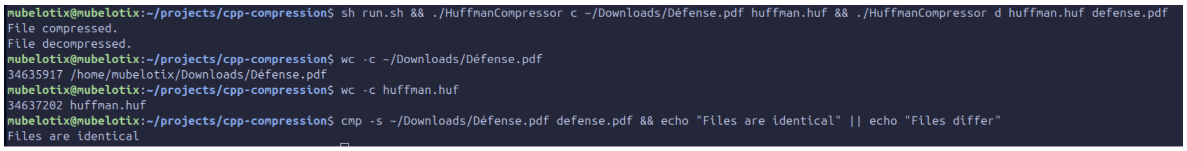
Figure 3: Results on the code of the project itself

When applied to a PNG image (**`signature.png`**), the file grew slightly from 73352 bytes to 74583 bytes. This is expected, as such formats are already compressed using sophisticated algorithms. Nonetheless, the program still handled the file correctly, and decompression produced an identical copy.



```
mubelotix@mubelotix:~/projects/cpp-compression$ sh run.sh && ./HuffmanCompressor c ~/Downloads/signature.png huffman.huf && ./HuffmanCompressor d huffman.huf signature.png
File compressed.
File decompressed.
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c huffman.huf
74583 huffman.huf
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c ~/Downloads/signature.png
73352 /home/mubelotix/Downloads/signature.png
mubelotix@mubelotix:~/projects/cpp-compression$ cmp -s ~/Downloads/signature.png signature.png && echo "Files are identical" || echo "Files differ"
Files are identical
```

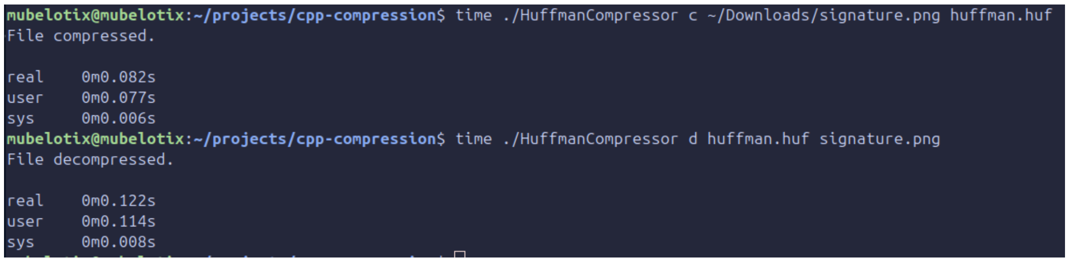
Figure 4: Results on a random image



```
mubelotix@mubelotix:~/projects/cpp-compression$ sh run.sh && ./HuffmanCompressor c ~/Downloads/Défense.pdf huffman.huf && ./HuffmanCompressor d huffman.huf defense.pdf
File compressed.
File decompressed.
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c ~/Downloads/Défense.pdf
34635917 /home/mubelotix/Downloads/Défense.pdf
mubelotix@mubelotix:~/projects/cpp-compression$ wc -c huffman.huf
34637202 huffman.huf
mubelotix@mubelotix:~/projects/cpp-compression$ cmp -s ~/Downloads/Défense.pdf defense.pdf && echo "Files are identical" || echo "Files differ"
Files are identical
```

Figure 5: Results on a 35MB file

Performance-wise, the program is fast and responsive even on larger files. Compressing and decompressing the image file each completed in under 130 milliseconds on a standard machine, even though the program was not compiled with optimizations.



```
mubelotix@mubelotix:~/projects/cpp-compression$ time ./HuffmanCompressor c ~/Downloads/signature.png huffman.huf
File compressed.

real    0m0.082s
user    0m0.077s
sys     0m0.006s
mubelotix@mubelotix:~/projects/cpp-compression$ time ./HuffmanCompressor d huffman.huf signature.png
File decompressed.

real    0m0.122s
user    0m0.114s
sys     0m0.008s
```

Figure 6: Time measurements for compression and decompression

Overall, the implementation is robust, fast, and accurate across file types.

5 Limitations

While the program performs reliably and efficiently, it has a few limitations I am well-aware of:

- **Limited Efficiency on Already-Compressed Files:** Huffman coding is most effective on data with redundant patterns, such as plain text. Binary files like PNG or ZIP are often already compressed using advanced algorithms, so applying Huffman coding can result in negligible gains or even slight size increases.
- **No Streaming Support:** The program currently reads the entire input file into memory. This approach works well for small to moderately large files, but may be inefficient or impractical for very large files (e.g., multi-gigabyte datasets).
- **Single-threaded Implementation:** All operations are performed sequentially. While compression and decompression are fast, performance could potentially be improved using parallelism, especially during frequency analysis and tree traversal.
- **Fixed Header Format:** The header in the compressed file stores the frequency table in a fixed format. While simple and effective, this format is not space-optimized and could be refined for better compression ratios in some cases.
- **No Error Detection or Recovery:** The program assumes the input and output files are valid and accessible. It does not currently include mechanisms to detect or recover from corrupted or malformed compressed files.

References

[Zel18] Julie Zelenski. Huffman encoding supplement. https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1186/assnFiles/assign6/huffman-encoding-supplement.pdf, 2018. CS106B Assignment 6 Supplement, Stanford University. Edited by Keith Schwarz.