Huffman Coding: File Compression using Greedy Algorithm

Northeastern University | CS5800 Algorithms | Prof. Aida Sharif Rohani

By
Bhalchandra Shinde
Shantanu Wankhare

Introduction

Key Question.

How can we design a lossless compression system that significantly reduces text file size while enabling fast, real-time decompression for reading?

CONTEXT

Redundancy
Traditional text formats uses uncompression. It has most words/char repeated

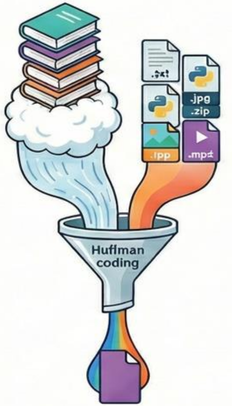
Compression
Efficient compression is needed for text files as well.

Size
Digital text is produced and consumed at massive scale - ebooks, log, documents, research papers websites

Memory & Time
Storing and transferring large files is expensive

Huffman tool

Reduce storage size without losing information.



Reader Application

Decode compressed data efficiently enough to display pages of a book on demand.

Encodings

File Type	Extensions	Existing Compression / Encoding Method	Is File Already Compressed?	Expected Huffman Compression Ratio	Explanation
Plain Text	.txt, .log, .csv, .json, .xml	None	❌ No	40% – 80% smaller	Text has skewed character frequency; Huffman is ideal.
Documents (Office XML)	.docx, .xlsx, .pptx	ZIP (DEFLATE: LZ77 + Huffman)	✓ Already compressed	File becomes 20–200% larger	Office files are ZIP containers; content already compressed using Huffman + LZ.
PDF	.pdf	Flate/DEFLATE, LZW, JPEG, JP2	✓ Already compressed	+20% to +300% expansion	PDF streams already use entropy coding; random-like distribution.
Images (raw)	.bmp, .ppm, .tiff (uncompressed)	None (sometimes RLE for TIFF)	❌ No	30% – 70% smaller	Raw pixel data compresses fairly well.
Images (compressed)	.jpg, .jpeg	DCT + Quantization + Huffman	✓ Yes	Huge expansion: 300% – 800%	JPEG already uses Huffman coding inside. Compressing again makes it worse.
Images (compressed)	.png	DEFLATE (LZ77 + Huffman)	✓ Yes	Very large expansion: 200% – 600%	PNG uses entropy coding and filters; nearly incompressible.
Audio (raw)	.wav, .pcm, .aiff	None	❌ No	10% – 40% smaller	Raw amplitude distributions slightly skewed; small gains.
Audio (compressed)	.mp3, .aac, .flac	MP3: MDCT + Huffman / AAC: Huffman / FLAC: Rice/Huffman	✓ Yes	Massive expansion: 300% – 1000%	Audio codecs already use Huffman coding internally.
Video (raw)	.yuv	None	❌ No	10% – 30% smaller	Pixel values partly skewed; limited improvement.
Video (compressed)	.mp4, .mov, .mkv, .avi	H.264/HEVC/AV1 (CABAC, CAVLC, entropy coding)	✓ Yes	Very large expansion: 200% – 800%	These codecs use advanced entropy coding more efficient than Huffman.
Python/Source Code	.py, .java, .cpp, .html, .css, .js	None	❌ No	30% – 70% smaller	High redundancy and repeated keywords; good for Huffman.
Binary Executables	.exe, .dll, .bin	Often packed or randomized	⚠️ Sometimes	Likely expansion: 20% – 500%	Binaries include many random bytes or pre-packed segments.
Archives	.zip, .7z, .rar, .gz, .whl	DEFLATE, LZMA, PPMD	✓ Fully compressed	Always expands	These formats already use Huffman, arithmetic coding, or LZ — cannot compress again.

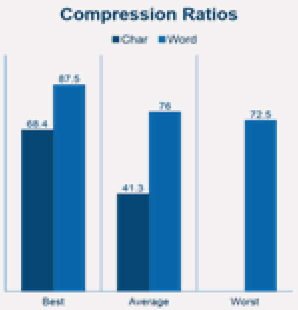
Rationale

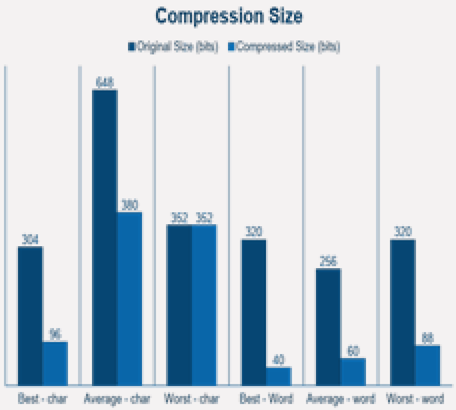
- Huffman Coding is a **classic Greedy Algorithm** that produces an **optimal prefix-free encoding** based on symbol frequencies.
- It is widely used in real systems (ZIP, JPEG, MP3), but classroom examples rarely show practical applications.
- This project extends the Huffman algorithm beyond theory, building a **complete working system**:
  - A compression tool
  - A decompression module
  - A chunk-based paging system
  - A Python Tkinter based book reader

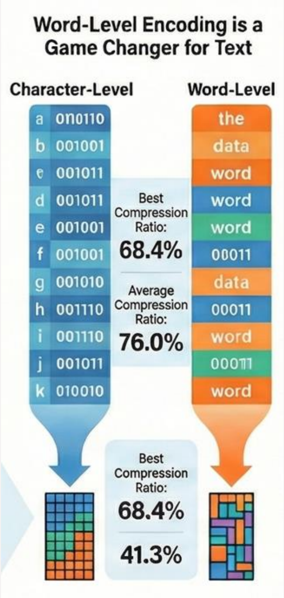
Why this matters:

- Demonstrates how algorithmic theory can be used to build a real application.
- Highlights trade-offs in designing usable compression systems (speed, memory, chunking, file formats).

Analysis - Word level?







- Character-level Huffman coding compresses but leaves redundancy:
  "the", "and", "to", "of" appear thousands of times.
- Word-level encoding:
  - Dramatically reduces redundancy
  - Produces shorter average codewords
  - Improves readability after decoding
  - Works perfectly for book-style text
- Compression improvement observed:
- Word-level coding achieved up to 70% space savings vs raw .txt.

Huffman.py

```
compress_file(file_path, mode)
Steps:
├── Read input file
├── Instantiate HuffmanCoding() object
├── Call → encode(text, code_output_path, mode)
├── write encoded data → compress_<file>.huff
├── write freq table → compress_<file>.huff.freq.json
├── visualize_tree(root)  ← optional
│       ├── Graphviz Digraph OR Custom Matplotlib
│       └── save PNG image
└── return success
```

```
encode()
├── build_frequency_table()
├── build_huffman_tree()
│       ├── create Node for each symbol
│       ├── heapq.heappush()
│       ├── heapq.heappop()
│       ├── merge two smallest nodes
│       └── return root
├── generate_codes(root)
│       ├── DFS traversal
│       ├── prefix + '0' for left
│       └── prefix + '1' for right
├── calculate compressed bits
├── write codebook summary file
└── return (encoded_bitstring, freq_table)
```

Huffman tool

Char level encoding

```
!python code/huffman_tool.py sample_char.txt compress char
```

```
[INFO] Huffman tree saved as /Users/bhalchandra/SEM1_NEU/huffman_tool/data/../output/sample_char.txt_tree.png.png
[INFO] Compression complete. Compressed file saved to: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/../output/sam
har.txt.huff
```

```
from collections import deque
print("".join(deque(open("/Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_char.txt_codes.txt"), 2)))
```

```
Actual compression ratio (file sizes): 44.65%
Original size: 422448 bytes, Compressed size: 233833 bytes
```

Decoding Char level encoded .huff file

```
!python ../code/huffman_tool.py sample_char.txt.huff decompress char
```

```
[INFO] Decompression complete using char-level Huffman decoding.
[INFO] Output: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/uncompressed_sample_char.txt.txt
```

sample_char.txt	422 KB	Plain Text
sample_char.txt_codes.txt	3 KB	Plain Text

sample_char.txt.huff	234 KB	Document
sample_char.txt.huff.freq.json	18 bytes	Plain Text
uncompressed_sample_char.txt.txt	422 KB	Plain Text

Word level encoding

```
!python code/huffman_tool.py sample_word.txt compress word
```

```
[INFO] Huffman tree saved as /Users/bhalchandra/SEM1_NEU/huffman_tool/data/../output/sample_word.txt_tree.png.
[INFO] Compression complete. Compressed file saved to: /Users/bhalchandra/SEM1_NEU/huffman_tool/data/../outpu
ord.txt.huff
```

```
from collections import deque
print("".join(deque(open("/Users/bhalchandra/SEM1_NEU/huffman_tool/data/sample_word.txt_codes.txt"), 2)))
```

```
Actual compression ratio (file sizes): 84.67%
Original size: 422448 bytes, Compressed size: 64753 bytes
```

Decoding word level encoded .huff file

```
!python ../code/huffman_tool.py sample_word.txt.huff decompress word
```

```
[INFO] Decompression complete using word-level Huffman decoding.
[INFO] Output: /Users/bhalchandra/SEM1_NEU/huffman_tool/output/uncompressed_sample_word.txt.txt
```

sample_word.txt	422 KB	Plain Text
sample_word.txt_codes.txt	14 KB	Plain Text

sample_word.txt.huff	65 KB	Document
sample_word.txt.huff.freq.json	5 KB	Plain Text
uncompressed_sample_word.txt.txt	421 KB	Plain Text

Reader - pseudocode

```
CHUNKED-HUFFMAN-COMPRESS(Text, PAGE_SIZE)
    words = split Text into tokens
    chunks = group words into PAGE_SIZE blocks
    outputBinaries = empty
    metadata = empty list

    for each chunk:
        freq = frequency count(chunk)
        tree = build Huffman tree(freq)
        codes = generate codes(tree)

        encodedBits = concat(codes[word] for word in chunk)
        packedBytes, padding = pack_into_bytes(encodedBits)

        append packedBytes to output file
        store in metadata: offset, length, padding, freq

    write metadata.json
```

```
CHUNKED-HUFFMAN-DECODE(file.huff, metadata)
    for each chunkInfo in metadata:
        bytes = read chunk from file
        bitstring = unpack_bytes(bytes, chunkInfo.padding)


        tree = rebuild Huffman tree(chunkInfo.freq)
        decodedWords += decode_bits_using_tree(bitstring, tree)


    return decodedWords
```

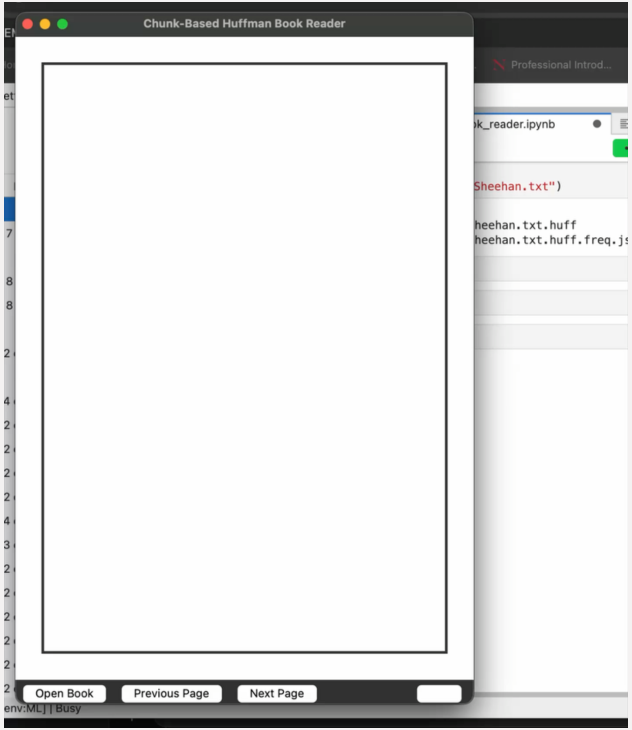
Inputs

- book.txt — raw book
- book.txt.huff — compressed binary
- book.txt.huff.freq.json — chunk metadata
- Chunk sizes, padding values, frequency tables

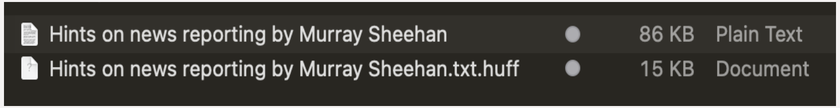
Outputs

- Decoded individual pages
- Page count
- Compression ratio
- Memory-efficient reading of entire text

Reader application



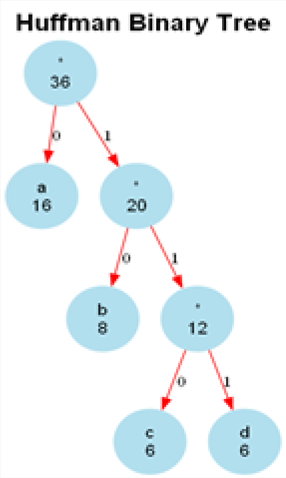
- Python Tkinter GUI
- Paper like reading experience
- Chunks of 250 words decoded per page
- Page Navigation controls
- Load any book with .huff and .json metadata file



Observations

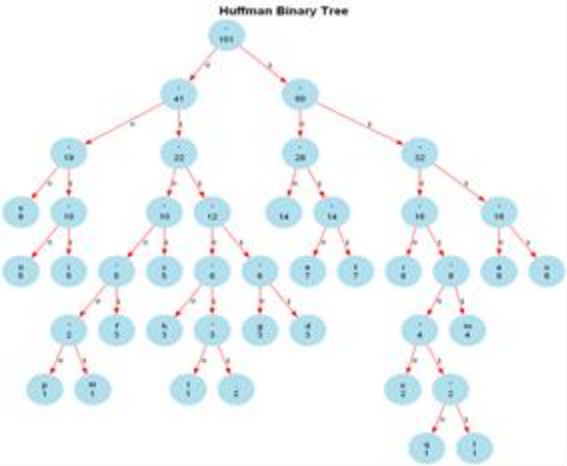
- More repetition → better compression
- Huffman coding efficiency depends on frequency distribution
- Worst-case for char: no gain, overhead may slightly increase size
- Visual tree helps understand code assignment
- Huffman Coding is not just theoretical, it powers real systems.
- This project demonstrates complete integration from **algorithm → compression → metadata → decoding → UI**.
- Shows how algorithm design, data structures and UX can combine to produce a working, practical application.

Best case Example:
"aaaaaaaaaaaaaaaabbbbbbbbccccccdddddd"



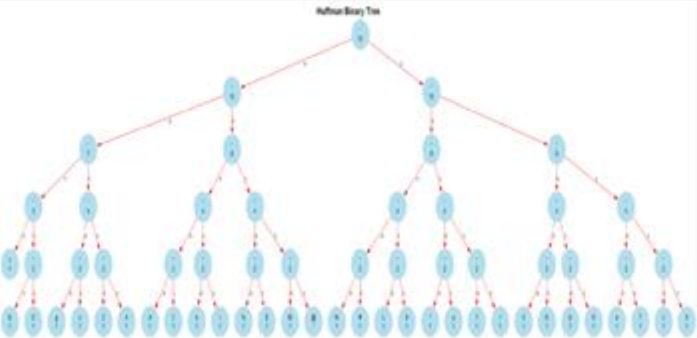

```
Huffman Encoding Report (Character-Level)
Original Text Length: 36 tokens
Symbol Table:
Symbol      | Frequency  | Huffman Code | Bits Used
================================================
a           | 16         | 0            | 16
b           | 8          | 10           | 16
c           | 6          | 110          | 18
d           | 6          | 111          | 18
================================================
Original size (bits): 288
Compressed size (bits): 68
Compression ratio: 76.39%
Unique symbols: 4
Most frequent symbol: 'a' (16 times)
Least frequent symbol: 'c' (6 times)
```

Average case Example: "Huffman coding is a data compression algorithm. It assigns shorter codes to more frequent characters."



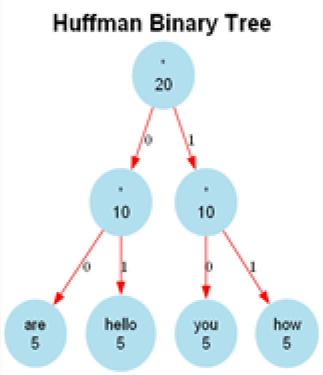

```
Huffman Encoding Report (Character-Level)
Original Text Length: 101 tokens
======================================
Original size (bits): 808
Compressed size (bits): 417
Compression ratio: 48.39%
Unique symbols: 22
Most frequent symbol: ' ' (14 times)
Least frequent symbol: 'H' (1 times)
```

Worst case Example:
"abcdefghiJKLMNO123456789!@#$%^&"



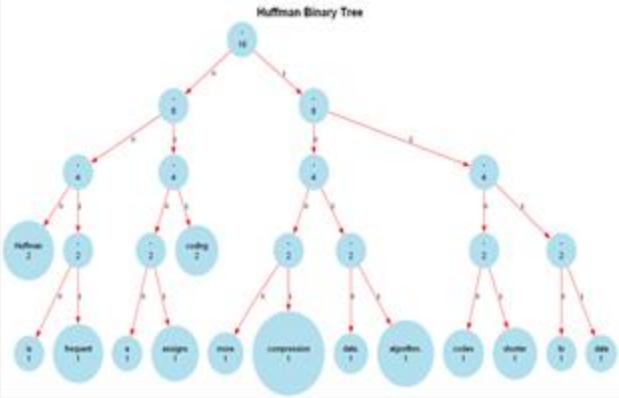
```
Huffman Encoding Report (Character-Level)
Original Text Length: 31 tokens
============================================
Original size (bits): 248
Compressed size (bits): 154
Compression ratio: 37.90%
Unique symbols: 31
Most frequent symbol: 'a' (1 times)
Least frequent symbol: 'a' (1 times)
```

Best case Example: "hello how are you hello how are you hello how are you hello how are you hello how are you"



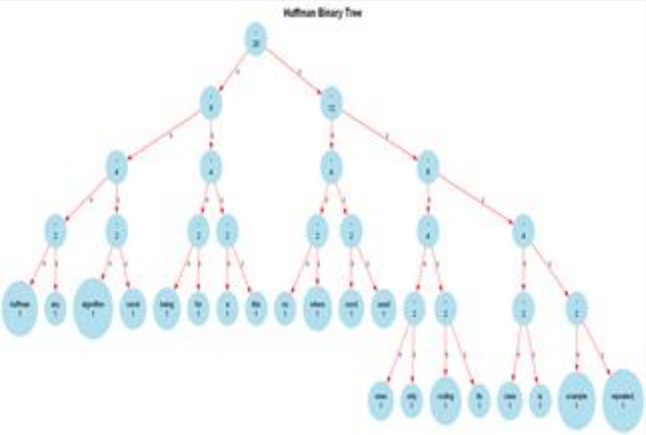

```
Huffman Encoding Report (Word-Level)
Original Text Length: 20 tokens
Symbol Table:
Symbol   | Frequency | Huffman Code | Bits Used
===============================================
are      | 5         | 00           | 10
hello    | 5         | 01           | 10
you      | 5         | 10           | 10
how      | 5         | 11           | 10
===============================================
Original size (bits): 320
Compressed size (bits): 40
Compression ratio: 87.50%
Unique symbols: 4
Most frequent symbol: 'hello' (5 times)
Least frequent symbol: 'hello' (5 times)
```

Best case Example: "Huffman coding is a data compression algorithm.
Huffman coding assigns shorter codes to more frequent data."




```
Huffman Encoding Report (Word-Level)
Original Text Length: 16 tokens
============================================
Original size (bits): 256
Compressed size (bits): 60
Compression ratio: 76.56%
Unique symbols: 14
Most frequent symbol: 'Huffman' (2 times)
Least frequent symbol: 'is' (1 times)
```

Worst case Example: "this is a worst case example for huffman coding algorithm where no any word being repeated, its used only ones"



```
Huffman Encoding Report (Word-Level)
Original Text Length: 20 tokens
===============================================
Original size (bits): 320
Compressed size (bits): 88
Compression ratio: 72.50%
Unique symbols: 20
Most frequent symbol: 'this' (1 times)
Least frequent symbol: 'this' (1 times)
```

Conclusion

Implemented system

- Reduces storage usage substantially
- Correctly reconstructs text
- Ensures fast page loading
- Works well across multiple books

Huffman Coding is not just theoretical,

it powers real systems

This project demonstrates complete integration from **algorithm** → **compression** → **metadata** → **decoding** → **UI**.

Shows how algorithm design, data structures and UX can combine to produce a working, practical application.

THANK YOU