Project 2

Sameer Shaik

1 Problem Statement

Huffman coding is a popular method for lossless data compression. It was developed by David A. Huffman in 1952. The main idea behind Huffman coding is to assign variable-length codes to input characters, with shorter codes assigned to more frequent characters. This way, the more common characters are represented by shorter bit sequences, resulting in more efficient compression.

2 Theoretical Analysis

Here's an overview of the Huffman coding process:

1. **Frequency Analysis:**
   - Calculate the frequency of each symbol (character) in the input data.
2. **Priority Queue (Min-Heap):**
   - Create a priority queue (min-heap) based on the frequencies of the symbols.
3. **Huffman Tree Construction:**
   - Build a binary tree by repeatedly combining the two nodes with the lowest frequencies until a single node (the root) is left. Each internal node has a weight equal to the sum of its children's weights.
4. **Code Assignment:**
   - Traverse the Huffman tree to assign binary codes to each symbol. Assign '0' for a left branch and '1' for a right branch.
5. **Generate Huffman Codes:**
   - The binary codes for each symbol are the paths from the root to the respective leaves.
6. **Encode (Compression):**
   - Replace each symbol in the original data with its corresponding Huffman code.
7. **Decode (Decompression):**
   - Use the Huffman tree to decode the compressed data back to the original symbols.

The efficiency of Huffman coding lies in its ability to create a compact representation for more frequent symbols, resulting in a variable-length code. This is in contrast to fixed-length codes, where each symbol is represented by the same number of bits.

3 Experimental Analysis

3.1 Program Listing

GitHub Link: https://github.com/vij-sameerb5/6212_PROJ2

3.2 Time Complexity

1. **Building Huffman Tree (build_huffman_tree function):**
   - The function uses a priority queue to repeatedly combine the two nodes with the lowest frequencies until a single node (root) is left.
   - The loop runs until there is only one node in the queue.
   - In each iteration, two nodes are removed from the queue, and one new node is added.
   - The time complexity of this process is O(n log n), where "n" is the number of symbols.
2. **Generating Huffman Codes (generate_huffman_codes function):**
   - The function traverses the Huffman tree and assigns binary codes to each symbol.
   - The traversal visits each node once.

- The time complexity of this process is O(n), where "n" is the number of symbols.

3. **Encoding (encode function):**
   - The function iterates through each character in the input text and retrieves its corresponding Huffman code.
   - The time complexity is O(m), where "m" is the length of the input text.
4. **Decoding (decode function):**
   - The function iterates through each bit in the encoded text and checks if the current code matches any Huffman code.
   - In the worst case, it might need to iterate through all codes for each bit.
   - The time complexity is O(m * n), where "m" is the length of the encoded text, and "n" is the number of symbols.

The overall time complexity is dominated by the construction of the Huffman tree, so it can be expressed as O(n log n), where "n" is the number of symbols in the input text. The encoding and decoding processes contribute linear factors but do not change the overall complexity.

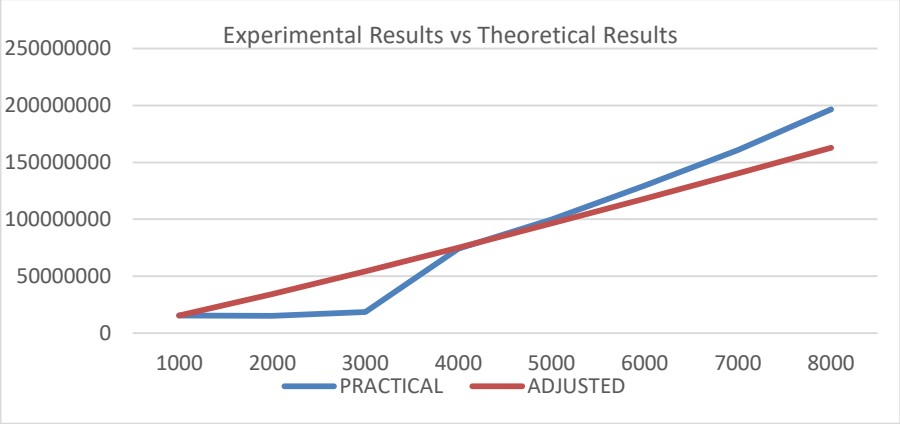
3.3 Data Normalization Notes

I normalized the value with constant 1570 that, I got by dividing one of the experimental value with the theoretical value, the value I got is 1570, now I have taken it as the constant, as in the context of time complexity analysis Multiplying or dividing a function by a constant does not change its big-O notation.

Output Numerical Data

n	PRACTICAL	ADJUSTED
1000	15651500	15646280.88
2000	15258200	34432549.2
3000	18559100	54404016.8
4000	74348600	75145125.09
5000	99678660	96458540.77
6000	129600000	118228033.6
7000	160844710	140376792.9
8000	196512800	162849820.3

3.4 Graph



4 Conclusions

Huffman coding is a highly efficient and widely employed algorithm for lossless data compression. By assigning shorter variable-length codes to more frequent symbols, it achieves optimal compression for diverse datasets. The algorithm's key strength lies in its construction of a binary tree, where the paths from the root to the leaves represent unique codes for each symbol. The time complexity for building the Huffman tree is O(n log n), with subsequent encoding and decoding processes having linear complexities. With applications in file compression and network protocols, Huffman coding's adaptability and ability to achieve significant compression ratios make it a fundamental and versatile solution in the field of data compression, ensuring that it continues to play a crucial role in various domains.