Isabella Morgan

CSCI 3333-06 Fall

Project 5: Huffman's Algorithm

My implementation follows the classic Huffman coding algorithm with six main phases. Phase 1 is the frequency analysis, where the input file in this case "bible.txt" is scanned character by character, and counting the occurrences of each unique character. For this phase, while n is the file size, this phase provides O(n) time complexity. Phase 2 constructs the node list; each character frequency pair is assigned to a binary tree node. Then each pair is inserted into a vector that's sorted by frequency. Using a custom insertSorted function, the insertion maintains ascending order. Phase 3 is tree building, repeatedly two nodes with the lowest frequencies are extracted, a parent node is then created with their combined frequency and then reinserted. This greedy approach continues until only one node remains, which is the root of the Huffman tree. This phase has O(m^2) complexity where m is the number of unique characters, since each iteration involves sorted insertion. Phase four performs a recursive depth-first traversal of the tree, appending "0" for left edges and "1" for right edges. This phase generates optimal prefix-free codes for each character. Phase 5 is the encoding phase; where we read the input file again, replacing each character with its Huffman code and writing the result to the output file. Lastly, phase 6 does the decoding, we use the Huffman tree to decode the bit string by traversing from root to leaf for each character, then verify that the decoded file matches the original.

Dr. Z provided the outline for the nodeType class which creates a binary tree node that contains five fields: character(char), frequency count (int), Huffman code string, and the left and right child pointers. Comparison operators were implemented to enable sorted insertion based on frequency. Some other structures I used was a map<char,int> for frequency counting, vector<nodeType*> for the sorted node list, and map<char>int for storing character to code mappings. I tried to utilize them well enough to provide efficient lookup and insertion operations that improve the algorithm's performance. Additionally, I chose to use recursive functions for tree traversal, which provides maintainable and cleaner code might I add. The sorted vector approach for node management is straight but not optimal, I believe a priority queue would provide better performance. However, for typical text files with approximately 100 unique characters, the difference in performance is negligible.

For the extra credit portion, the bit-level compression, I added functions to pack eight bits into each byte and store the bit length as metadata, this makes it possible for true compression by using bits rather than character representations of “0” and “1”. The bit-packing algorithm pads the final byte with zeros and stores the exact bit count, ensuring no data loss during decompression. This approach achieves compression ratios of 50-60%.