Lab 10 Post-Lab Report

Name: Peter Haynes
Student id: prh7yc
Date: 11/30/19
Section: Tuesdays 12:30

Implementation Description:

To read in my characters and their frequencies I used an unordered map for two reasons. First, it holds both the character and its frequency in an easily accessible grouping. Second, it is easy to count the frequency of an item with an unordered map by incrementing the value of a key every time the key is found in the file. After finding these frequencies, I passed these pairings into my priority heap in a vector that consisted of a structure I called huffmanNodes. These nodes consisted of a character, an integer, and two pointers to other huffmanNodes. The character element stored the printable characters while the integer element stored the freqeuncy of that element in the file. Additionally, while all the huffmanNodes in the vector had their pointers initialized to NULL, these pointers would come in handy when constructing the huffmanTree. The tree of prefix codes branches down from a single huffmanNode which, when the tree is fully constructed, becomes the only huffmanNode stored in the priority heap vector. Using the pointers of the huffmanNode structure, I created intermediary levels for the prefix tree. The huffmanNodes that contained the characters we wanted to access still have pointers to null as they are the leaves of the tree, however our levels of the tree that lead to these leaves contain pointers to either other intermediaries or the leaves themselves while also containing a null character and a frequency of 0. This allows the program to traverse the tree looking only for the leaves. Thus, when writing the codes to the output file, the tree can simply traverse to every leaf keeping track of right and left traversals and then record the string associated with these traversals in tandem with the character at that leaf. Additionally, when constructing the bitstring tree in the decoding step, this makes constructione easy as the program can go left for 0s and right for 1s then know that when it reaches the end of the string that must be a leaf and to insert the character node there. Lastly, in the decoding step, I reaf the bitstrings and their characters into another unordered_map that maps strings to characters. This way, as the program traverses the tree keeping track of its left and right traversals, it can track if the bitstring it has built to that point corresponds to a node that contains a character, and if it does it can report that character. This also allows for files that are just long strings of 1s and 0s with no spaces separating the character strings because the tree can check if the string it has built corresponds to a character at every step rather than going through a given binary string and reporting the character at the end.

Time and Space Complexity:

Compression:

1. Read the source file and determine the frequencies of the characters in the file.
    a. Time: Requires reading n characters from a file and finding each in the unordered map. While the find function is usually constant time, it's worst case is O(n). However, since the unordered_map contains a maximum of 96 elements regardless of the size of the file, runtime is constant with regard to the size of the file. Thus, worst case runtime is linear.
    b. Space: Using the sizeOf() function: the unordered_map uses 40 bytes of memory. Additionally, each character uses 1 byte and each int used 4 bytes. So, since there are 96 possible elements, the worst case space usage is 40 + 96*5 = 520 bytes. However, in practice, the space usage is 40 + n*5 where n is the number of unique bits.
2. Store the character frequencies in a heap (priority queue).
    a. Time: This action is simply inserting the created nodes into the vector in the heap. Since the number of nodes has a constant maximum value (96 printable characters) independent of how many characters are in the file, this time is constant.
    b. Space: 24 bytes for the vector + 8 for every huffmanNode pointer + 24 for every huffmanNode. This is a maximum of 24 + 96*32 = 3096 bytes in the worst case. In practice, the space usage is 40 + n*32 where n is the number of unique bits.
3. Build a tree of prefix codes (a Huffman code) that determines the unique bit codes for each character.
    a. Time: Since the maximum number of huffmanNodes in the tree is independent of the number of characters in the file, constructing the prefix tree does not depend on n and thus the big o runtime is constant.
    b. Space: The space for the huffmanNodes and the vector has already been accounted for, but we require additional space for every intermediary node that connects the tree of which there is a maximum of 95. Thus, in the worst case this is an additional 95*32 = 3040 bytes and in practice n*32 where n is the number of intermediary nodes required to connect the tree.
4. Write the prefix codes to the output file, following the file format above.
    a. Time: Again, since the number of prefix codes has a maximum independent of the number of characters in the file, the big o runtime is constant.
    b. Space: 40 bytes for the unordered_map plus one byte for every char and 24 for every string in the map. With 96 possible elements, worst case space usage is 40

+ 96*25 = 2440 bytes. In practice, usage is 40 + n*25 where n is the number of unique characters.

5. Re-read the source file and for each character read, write its prefix code to the output, following the file format described herein.
    a. Time: Since it must go through every character in the file, the runtime is at least n. Since finding the prefix code for any given character is constant, the runtime is O(n).
    b. Space: Requires no space in the program but one byte for every character in the output file. Thus, the output file requires n bytes where n is the number of characters in the original file.

Decompression:

1. Read in the prefix code structure from the compressed file. You can NOT assume that you can re-use the tree currently in memory, as we will be testing your in-lab code on files that you have not encoded.
    a. Time: Since there is a maximum number of prefix codes independent of the number of characters in the file, the big o runtime is constant. In the worst case, it will require enough time to read the prefix code for all 96 printable characters and insert them into an unordered map.
    b. Space: 40 bytes for the unordered_map plus one byte for every char and 24 for every string in the map. With 96 possible elements, worst case space usage is 40 + 96*25 = 2440 bytes. In practice, usage is 40 + n*25 where n is the number of unique characters.
2. Re-create the Huffman tree from the code structure read in from the file.
    a. Time: Since the Huffman tree maximum size is independent of the number of characters in the file, the big O runtime is constant.
    b. Space: The tree requires 32 nodes for every huffmanNode of which there a maximum of 191 (96 for the ones containing characters and 95 for the ones connecting them) for a worst case space usage of 191*32 = 6112 bytes.
3. Read in one bit at a time from the compressed file and move through the prefix code tree until a leaf node is reached.
    a. Time: Since the number of steps it takes to reach a leaf is independent of the number of characters in the file, big O runtime is constant.
    b. Space: No space to read through the bits.
4. Output the character stored at the leaf node.
    a. Time: Output a character does not depend on the number of characters in the file. Runtime is constant.
    b. Space: Requires 1 byte in the output file for the character found.

5. Repeat the last two steps until the encoded file is finished.
    a. Time: Since the number of bits in the compressed file is determined by the number of characters in the original file and since it traverses the tree once for every character in the file, big O runtime is n.
    b. Space: Requires n bytes in the output file - one byte for every character from the original file.