Post lab report 10

Jiwon Cha

1. A description of your implementation. Describe the data structure used in your implementation and why you selected them.

Compression

First of all, to store the frequencies of each character, I chose an array as my data structure. Since each ASCII character has an integer value, array is a fast and efficient way to store the frequencies of the characters, because each time a character (c) is found, we simply need to increment the value stored at the index of int (c) by one. Also, it is easy to calculate how many characters are distinct because it only involves calculating the number of spots in the array that are not empty.

Then, the stored values (frequencies) and characters are stored in the heap, which is a vector that contains integer values, via a data structure called HuffNode. HuffNode is a node of a Huffman tree which stores 4 variables: 2 pointers (to its left and right nodes), an int value, a char value. It takes in a character and stores it within the node. Moreover, it has an int value that stores the frequency of the input (character) which is initialized to 1 at first. Those Huffman nodes are then pushed onto the heap.

For the final output, I created another array of strings which contains the prefix values of each character. Even though it takes up more memory than I thought it would, it is an efficient way to store the prefix values, print them and calculate the compression ratios and costs of the Huffman tree.

Decompression

First of all, I read in the input file to get the prefix codes for each character. To decode the encoded message, I wrote makeTree function which builds a Huffman tree based on the encoded message. Then, to print the original (decoded) message, I wrote a recursive function called traverse that traverses the Huffman tree and when it reaches a leaf node, it prints out the stored character within that node. The traverse function was used within a while loop so that every time the function successfully prints out the character, the while loop deletes the first n characters of the string so that the traverse function can proceed to the next prefix.

2. An efficiency analysis of all steps in Huffman encoding/decoding.

Compression

Worst case running time

Step 1: Read the source file and determine the frequencies of the characters in the file

- Stores the frequencies in an array (insertion at a specified index): n

Worst case running time: Big-theta(n)

Step 2: Store the character frequencies in a heap

- Insert into the heap: log n
- Number of insertions depends on the input size: n

Worst case running time: Big-theta(n * log n)

Step 3: Build a tree of prefix codes that determines the unique bit codes for each character

- Delete the minimum: log n
- Insert into the heap: log n
- Number of insertions and deletions depends on the input size: n

Worst case running time: Big-theta(n * log n)

Step 4: Write the prefix codes to the output file

- Recursive findCodes function: n
- Setting the value at index of ASCII value of the character to prefix: n

Worst case running time: Big-theta(n)

Step 5: Re-read the source file and for each character read, write its prefix code to the output

- While loop: n
- Addition: constant

Worst case running time: Big-theta(n)

Space complexity
Heap: 32 bytes
Array of frequency (int): 512 bytes
Array of ASCII values (string): 3072 bytes
Huffman node: 24 bytes
Int values: 8 bytes

- Count
- Distinct

Total = 3648 bytes

Decompression
Worst case running time
Step 1: Read in the prefix code structure from the compressed file

- While loop: n

Worst case running time: Big-Theta(n)

Step 2: Read in one bit at a time from the compressed file and move through the prefix code until a leaf node is reached

- While loop: n
- Recursive makeTree function: n

Worst case running time: Big-Theta(n)

Step 3: Write the character stored at the leaf node into the decompressed file

- While loop: n
- Recursive traverse function: n

Worst case running time: Big-Theta(n)

Step 4: Re-read the source file and for each character read, write its prefix code to the output

- 2 while loops

Worst case running time: Big-Theta(n)

Space complexity
String stream: 280 bytes
String: 24 bytes
Huffman node: 24 bytes
Int value: 4 bytes

Total = 332 bytes