Lab 1: Huffman Coding

Due date: Sunday, January 21 at 11:55 pm

In this lab we will learn about the whole process of basic huffman encoding and decoding, as described in the lecture slides. A skeleton code file in python is given and you need to finish implementing the functions defined in that file.

Please note that you are NOT allowed to import any other packages (even if you don't use them), but you are free to write any helper functions you need.

This technique works by creating a binary tree of nodes. A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol, the probability (frequency of appearance) of the symbol. More frequent symbols are assigned shorter codewords and less frequent characters are assigned longer codewords. Internal nodes contain sum of occurrence of child nodes and links to two child nodes. Bit '0' indicates left child and '1' represents right child as common convention.

There are several steps to take to finish the huffman encoding and decoding process:

• Count the occurrence of every symbol in the input message.
• Build a huffman tree of the symbols.
• Assign code-word for the symbols in the huffman tree.

• Generate the encoded message based on the code-words.
• Decode the generated huffman code using the huffman tree.

Every step corresponds to a function in the skeleton code. You can add your own helper functions if you like. But you cannot modify the signature of any functions in the skeleton code. Below is a detail description of the functions. You will find more information in the skeleton code.

buildDictionary(message):
In this function, you need to count the occurrence of every symbol in the message and return it in a python dictionary. The keys of the dictionary are the symbols, the values of the dictionary is their corresponding occurrences.

buildHuffmanTree(word_dict):
In this function, you will use the dictionary returned by buildDictionary() to build a huffman tree. The ADT for the tree nodes has been provided in the skeleton code and you should read it to figure out how to use it. There are nodes in huffman trees that are functioning only as a sum of occurrences, so for those nodes their symbol attribute can be None. Please refer to the lecture slides for a detailed explanation of the process. Return the root node of the huffman tree at the end.

assignCodeWord(root, code word=”):
In this function, code-words are assigned to nodes with a symbol in the huffman tree. Those nodes without a symbol won’t have a code-word. You can use recursion to implement this function.

huffmanEncode(message):
This function dose the encode process and returns the encoded message and the huffman tree root node. You will use the three functions implemented above to finish this. The returned message will be a string with only ”0” and ”1” in it. The returned huffman tree will contain code-words so you can use it to decode the message later.

huffmanDecode(message, huffman tree):
This function takes the encoded message and the huffman tree root node and produces the decoded message. The input message contains no separators between code-words. But huffman codes are prefix free, which means every code-word is guaranteed not to be a prefix of any other code-word. Therefore, you can use the entire input to walk down the huffman tree until you meet a node with a symbol, and that marks the end of a code-word. The decode result of the current code-word is the symbol in the current node. The return of this function is the decode message string, and it should be the same as your original input message before encoding.

main():
Implement the main process here. You will use the functions implemented above here. Ask the user for a string input, encode the string, print out the huffman code and its length, then decode the huffman code and print out the decoded message.

Grading:

- Code Quality: 10%
- Comments: 5%
- buildDictionary: 5%
- buildHuffmanTree: 25%
- assignCodeWord: 25%
- huffmanEncode: 15%
- huffmanDecode: 10%
- main: 5%