CPE202 revised 02/21/2020 v4

Project 5 (Optional)

Real File Compression By Huffman Coding

You can earn up to extra 100 points (in the project category) for successful completion of this project assignment.

Objective

To understand how data compression works, and also to learn how data is stored in computers.

Prerequisites

You need to have successfully project 3 before starting this assignment.

Background

Information is stored as binaries in computers. Binaries consist of only 0s and 1s. This means that the value of each bit is either 0 or 1. Usually eight bits are handled as one unit called “byte”. One bit can express a number between 0 and 1. Two bits upto 3. Three bits 7 and so on. N bits can represent numbers up to 2N. So, one byte, which is eight bits, can represent values up to 255 including 0. This is why each character in alphabet is represented by one byte with additional characters such as . , / ? % $ and other characters you see in ASCII code table (256 characters in total).

ord(‘a’) = 97. 97 in binary is

0	1	1	0	0	0	0	1

The Huffman code represents characters with bits, not bytes. So, we need to be able to store Huffman code in a file by bit, not by byte. However, Python does not allow us to store information by bit: it only allows us to store information in a file by byte. Hence, we need to pack Huffman codes into bytes, and we need to invent a way to mark the end of encoding because it is not guaranteed that the entire length of encoding in Huffman code is divisible by 8: We might end up with extra bits in the last byte, and we have to make sure that a decoder knows that those extra bits are not the part of the encoded string. To mark the end of encoding, we use

EOT (the end of text) character specified in ASCII table. This means that we always include the EOT character in our Huffman tree with its frequency being one, and add the Huffman code of the EOT character at the end of the encoded string. The ASCII code of EOT is 3.

Modifications to be made to your project 3 code

Create following functions:

def tree_preord(hufftree) that takes a Huffman tree and produces the tree description given in the second bullet under Header Information. Namely a string of characters produced by a preorder traversal of a Huffman tree writing 0 for a non-leaf node or 1 for a leaf node followed by '-' followed by the ord() value of the character stored in the leaf node with no spaces or separating characters followed by '-' (we use '-' as the separator).

def binalize(serialized), which converts the return value of tree_preord function, serialized, into list of bytes, and returns it, converting each character into byte. Use int(int_char, 10).to_bytes(1, byteorder='big')) for the conversion, where int_char is one of '0', '1', and other numerical strings. Convert '-' to integer using ord() and convert to byte. Append 255 at the end of the binary string which is 11111111 in binary marking the end of the header.

def to_byte_list(char_list), convert a list of one character string to a list of bytes. You will use this function to pack a huffman encoded string into bytes. For example, '111010111' will be converted to '1110101110000000', which is two bytes with zero padding on the right. Use int(byte_char, 2).to_bytes(1, byteorder='big')) to convert a string of eight 0s or 1s to a byte.

def right_zfill(chars, size), zero pads a string on the right. Use format function. For example,

```
'{:<08d}'.format(190)
'19000000'
```

def huffman_encode_bin(in_file, out_file), encodes the content of a file, in_file, in huffman code and stores the encoded in a binary file, out_file. Add a huffman code for EOT char (the ascii code of EOT is 3) at the end of the encoded string and pass it to to_byte_list() function. Store the header and then the encoded string. The header is the serialized string of a huffman tree produced by your tree_preord function but converted to binary by your binalize function. You may omit the '-' character in the result of the tree_preord function, when you convert it to binary in order to keep the size of the file as small as possible (you may not be able to achieve

any compression when you try to compress a small file if you keep the '-' in the header.). In that case, you need to remember to add '-' back when you decode the header.
To write binary into a file, use
with open(out_file, 'wb') as out:
out.write(byte)

def restore_tree(preordstr, tree=None), restores a huffman tree from a serialized string of the huffman tree. Take a look at the specification about the tree_preord function in the project 3 instruction.

def convert_bin(num, b), converts an integer to a binary string. For example, 8 will be converted to '00001000'. Use zfill(8) to zero pad your string on the left. For example, *'1111'.zfill(8)* will return '00001111'.

def get_header_encoded(in_file), extracts the header, a binary string of serialized huffman tree, and the body, huffman encoded string, from a binary file. Convert the header to a string (1) whose format is the same as that of the string generated by the tree_preord function: if the integer is not 0 nor 1, that is an ord value of a character because "*0 for a non-leaf node or 1 for a leaf node followed by '-' followed by the ord() value of the character stored in the leaf node with no spaces or separating characters followed by '-' (we use '-' as the separator, but you might have decided to omit it when you convert the header into binary)*". Also, convert the encoded text to strings of '1's and '0's (2). Return both the header string (1) and the encoded text string (2) from this function.

To open a binary file, use
with open(in_file, 'rb') as infile:
byte = infile.read(1)

To convert a byte to an integer number, use
int.from_bytes(byte, byteorder=sys.byteorder)
,which will return an int.

The number 255, which is 11111111 in binary marks the end of the header. Stop the decoding when you encounter the huffman encoding for EOT character.

def decode(tree, encoded_string), using the restored HuffmanTree, tree, decode the encoded_string, a string of '1's and '0's into normal string. Return the decoded string. You may have already implemented this function in Project 3.

def huffman_decode_bin(in_file, out_file), decodes the content of a binary file, in_file, containing the header and huffman encoded string into an ASCII string and store the string in a

file, out_file. Call get_header_encoded() from this function. Call convert_bin() to convert an integer to a string of ‘0’s and ‘1’s. Then, restore a HuffmanTree from the header. Pass the restored HuffmanTree, and the Huffman encoded string to decode function to convert the encoded string into normal string. Then, write the decoded string into the out_file.

Modify your Project 3 code to use **huffman_encode_bin** and **huffman_decode_bin, but try to reuse your existing code as much as possible (You can still use your existing functions to count frequencies and to build a huffman tree. After converting binary to ascii string, you can use the existing code for decoding).**

Test

The result of encoding a text file to a binary encoded file and decoding the decoded file should be identical to the original text file. As a result of encoding, the size of the encoded file should be smaller than that of the original file or if larger, the size should be close to the original size. Larger the size of the original file ,more compression you should be able to gain out of your huffman encoding except for some edge cases where each character in the file is unique.

Submission

Demo your work to your instructor or TA by the due date.