CSC 202 Fall 2018

Project 3- Group of 2

Part A: Due (11/3@11:55PM)

Part B: Due (11/7@11:55PM)

Huffman coding counts as 2 projects

For this project, you will implement a program to encode and decode text using Huffman coding. Huffman coding is a method of lossless (the original can be reconstructed perfectly) data compression. Each character is assigned a variable length *code* consisting of '0's and '1's. The length of the code is based on how frequently the character occurs; more frequent characters are assigned shorter codes.

The basic idea is to use a binary tree, where each **leaf node** represents a character and frequency count. The Huffman code of a character is then determined by the unique path from the root of the binary tree to that leaf node, where each 'left' accounts for a '0' and a 'right' accounts for a '1' as a bit. Since the number of bits needed to encode a character is the path length from the root to the character, a character occurring frequently should have a shorter path from the root to their node than an "infrequent" character, i.e. nodes representing frequent characters are positioned less deep in the tree than nodes for infrequent characters.

The key problem is therefore to construct such a binary tree based on the frequency of occurrence of characters. A detailed example of how to construct such a Huffman tree is provided on PolyLearn

Note:

- Every function must come with a signature and a purpose statement.
- You must provide test cases for all functions.
- Use descriptive names for data structures and helper functions.
- You will not get full credit if you use built-in functions that provide the "data structure" functionality that you are supposed to implement. Check with your instructor if you are unsure about a specific function,

Functions

The following bullet points provide a guide to implement some of the data structures and individual functions of your program. Start by creating a file **huffman.py** and add functions and data definitions to this file, if not otherwise stated. You should develop incrementally, building test cases for each function as it is implemented.

PART A

1) Count Occurrences: cnt_freq(filename)

- Implement a function called **cnt_freq(filename)** that opens a text file with a given file name (passed as a string) and counts the frequency of occurrences of all the characters within that file. Use the built-in Python List data structure of size 256 for counting the occurrences of characters. This will provide efficient access to a given position in the list. (In non-Python terminology you want an array.) You can assume that in the input text file there are **only 8-bit characters** resulting in a total of 256 possible

character values. Use the built-in function ord (c) to get an integer value between 0...255 for a given character c (and chr () to go from the integer value back to the character as a Python string. This function should return the list with the counts of occurrences. There can be issues with extra characters in some systems.

2) Data Definition for Huffman Tree

Write a recursive data definition for a Huffman tree, which is a possibly empty binary tree of HuffmanNodes.

- A **HuffmanNode** class represents either a leaf or an internal node (including the root node) of a Huffman tree. A HuffmanNode contains a character value (either as the character itself or the ord() of the character, your choice) and an occurrence count, as well as references to left and right Huffman subtrees, each of which is a binary tree of HuffmanNodes. The character value and occurrence count of an internal node are assigned as described below. You may want additional fields in your HuffmanNodes if you feel it is necessary for your implementation. **Do not change the names of the fields specified in the huffman.py file posted on PolyLearn**.

3) Build a Huffman Tree

Since the code depends on the order of the left and right branches take in the path from the root to the leaf (character node), it is crucial to follow a specific convention about how the tree is constructed. To do this we need an ordering on the Huffman nodes.

- Start by defining a function **comes_before ('a', 'b')** for Huffman trees that returns true if tree '**a'** comes before tree '**b'**. A Huffman tree '**a'** comes before Huffman tree '**b'** if the occurrence count of '**a'** is smaller than that of '**b'**. In case of equal occurrence counts, break the tie by using the ASCII value of the character to determine the order. If, for example, the characters 'd' and 'k' appear exactly the same number of times in your file, then 'd' comes before 'k', since the ASCII value of character 'd' is less than that of 'k'.
- Write a function that builds a Huffman tree from a given list of the number of occurrences of characters returned by **cnt_fref()** and returns the root node of that tree.
  Call this function **create_huff_tree(list_of_freqs)**.
  - Start by creating a list of individual Huffman trees each consisting of single HuffmanNode node containing the character and its occurrence counts. Building the actual tree involves removing the two nodes with the lowest frequency count from the sorted list (This means you will need a **findMin()** function that uses **comes_before()** to find the two "smallest nodes".) and connecting them to the left and right field of a new created Huffman Node as in the example provided. The node that comes before the other node should go in the left field.
  - Note that when connecting two HuffmanNodes to the left and right field of a new parent Node, that this new node is also a HuffmanNode, but does not contain an actual character to encode. Instead this new parent node should contain an occurrence count that is the sum of the left and right child occurrence counts as well as the minimum of the left and right character representation in order to resolve ties in the **comes_before()** function.
  - Once a new parent node has been created from the two nodes with the lowest occurrence count as described above, that parent node is inserted into the list of sorted nodes.
  - This process of connecting nodes from the front of the sorted list is continued until there is a single node left in the list, which is the root node of the Huffman tree.
    create_huff_tree(list_of_freqs) then returns this node.

4) Build a List for the Character Codes

- We have completed our Huffman tree, but we are still lacking a way to get our Huffman codes. Implement a function **create_code(root_node)** that traverses the Huffman tree that was passed as an argument. Traverse the tree from the root to each leaf node and adding a '0' when we go 'left' and a '1' when we go 'right' constructing a string of 0's and 1's. You may want to:

    - use the built-in '+' operator to concatenate strings of '0's and '1's here.
    - You may want to initialize a Python list of strings that is initially consists of 256 empty strings in **create_code**. When **create_code** completes, this list will store for each character (using the character's respective integer ASCII representation as the index into the list) the resulting Huffman code for the character. The code will be represented by a sequence of '0's and '1's in a string. Note that many entries in this list may still be the empty string. Return this list.

5) Huffman Encoding

- Write a function called **huffman_encode(in_file, out_file)** (use that exact name) that reads an input text file and writes, using the Huffman code, the encoded text into an output file. The function accepts two file names in that order: input file name and output file name, represented as strings. If the specified output file already exists, its old contents will be erased. See example files in the test cases provided to see the format.
- Writing the generated code for each character into a file will actually enlarge the file size instead compress it. The explanation is simple: although we encoded our input text characters in sequences of '0's and '1's, representing actual single bits, we write them as individual '0' and '1' characters, i.e. 8 bits. To actually obtain a compressed the file you would write the '0' and '1' characters as individual bits.

PART B

Huffman Decoding

Header Information

Note: In this assignment you will not actually be storing the header information with the encoded file.
In an actual implementation some initial information must be stored in the compressed file that will be used by the decoding program. This information must be sufficient to construct the tree to be used for decoding. There are several alternatives for doing this including:

- Store the character counts at the beginning of the file. You can store counts for every character, or counts for the non-zero characters. If you do the latter, you must include some method for indicating the character, e.g., store character/count pairs. This is conceptually what is being done in the assignment by passing the frequencies into **huffman_decode()**.
- Store the tree at the beginning of the file. One method for doing this is to do a pre-order traversal, writing each node visited. You must differentiate leaf nodes from internal/non-leaf nodes. One way to do this is write a single bit for each node, say 1 for leaf and 0 for non-leaf. For leaf nodes, you will also need to write the character stored. For non-leaf nodes there's no information that needs to be written, just the bit that indicates there's an internal node.
- Use a "standard" character frequency, e.g., for any English language text you could assume weights/frequencies for every character and use these in constructing the tree for both encoding and decoding.

Implementation

- Write a function **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 the character stored in the leaf node with no spaces or separating characters.
- Write a function called **huffman_decode(freqs, encoded_file, decode_file)** (use that exact name) that reads an encoded text file, **encoded_file,** and writes the decoded text into an output text file,

decode_file, using the Huffman Tree produced by your programs **cnt_fref()** and **create_huff_tree(list_of_freqs)**. If the specified output file already exists, its old contents will be erased.

Example:
Suppose the file to be encoded contained: ddddddddddddddddcccccccccbbbbaaff
Numbers in positions of freq counts [97:104] = [2, 4, 8, 16, 0, 2, 0]
Tree: all interior nodes are labeled by a, the leaves containing "a" and "f" are at level 4, ;leaf containing "b" is at level 3, leaf containing "c" at level 2 and leaf containing "d" at level 1.
Returned by tree_preord: "00001a1f1b1c1d"

Tests

- Write sufficient tests using unittest to ensure full functionality and correctness of your program. Start by using the supplied **huffman_tests.py**. **file1.txt** and **encodedfile1_sol.txt** to begin testing your programs.
- When testing, always consider *edge conditions* like the following cases:
  - If the input file consists only of some number of a single character, say "aaaaa", it should write just that character followed by a space followed by the number of occurences. E.g. 'a' 5
  - In case of an *empty* input text file, your program should also produce an *empty* file.
  - If an input file does not exist, your program should abort with an error message, "input file not found".

Some Notes

- When writing your own test files or using copy and paste from an editor, take into account that most text editors will add a newline character to the end of the file. If you end up having an additional newline character '\n' in your text file, that wasn't there before, then this '\n' character will also be added to the Huffman tree and will therefore appear in your generated string from the Huffman tree. In addition, an actual newline character is treated different, depending on your computer platform. Different operating systems have different codes for "newline". Windows uses '\r\n' = 0x0d 0x0a, Unix and Mac use '\n' = 0x0a. It is always useful to use a hex editor to verify why certain files that appear identical in a regular text editor are actually not identical.

Submission

You must submit the following files: The first three count as part A and the last three as part B

- huffman.py

PART A

- **cnt_freq(filename)** returns a Python list of 256 integers the frequencies of characters in file
- **create_huff_tree(list_of_freqs)** returns the root node of a Huffman Tree
- **create_code(root_node)** returns a Python list of 256 strings representing the code
- **huffman_encode(in_file, out_file)** encodes in_file and writes the it to out_file

PART B

- **tree_preord(hufftree)** writes a string representation of the Huffman tree
- **create_huff_tree(list_of_freqs)** Tree based on code
- **huffman_decode(list_of_freqs, encoded_file, decode_file)** decodes encoded file, writes it to decode file

- huffman_tests.py The test cases used in developing your programs
- Up to three text files you used for testing.
- A text file as a report that explains responsibility of each member.
- **Note:**
  - Part A and B must be implemented in same file (huffman.py).
  - All test cases of Part A and B must be implemented in same file (huffman_test.py).
  - The files you use for testing must end with .txt.
  - You can create as many helper functions as you need.