Assignment 3b

Huffman Decoding

For this project, you will implement a program to decode text that has been encoded using Huffman coding. This project assumes that the **compressed** input file is in the format that was output from Assignment 3a. (The compressed file is the one generated using the HuffmanBitWriter class. Do not write code to decode the non-compressed version of the file.)

For decoding, you will need to recreate the Huffman tree that was used to determine the Huffman codes used for encoding. The header in the input file contains the character frequency information needed for this. After reading in the frequency information, you will be able to use the same create_huff_tree() function that you wrote for the encoding portion of the assignment.

Once the Huffman tree is recreated, you will be able to traverse the tree using the encoded 1's and 0's from the input file. Use the HuffmanBitReader class to read the individual 1's and 0's. The decoding process starts by beginning at the root node of the Huffman tree. A 0 will direct the navigation to the left, while a 1 will direct the navigation to the right. When a leaf node is reached, the character stored in that node is the decoded character that is added to the decoded output. Navigation of the Huffman tree is then restarted at the root node, until all of the characters from the original file have been written.

2 Recreating Huffman Tree, Decoding

HuffmanBitReader

This class is provided to you in the huffman_bit_reader module. You will use this class to read the header information from the compressed file and also to read individual bits from the compressed file.

Implementation

- You should start with your huffman.py file from the first portion of the assignment, and add the functions necessary for the decode portion of the assignment.
- Write a function called **huffman_decode(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 using the header information. If the encoded_file does not exist, your program should raise the FileNotFoundError exception. If the specified output file already exists, its old contents will be overwritten.
- Before recreating the Huffman tree you will need to create a **list_of_freqs** from the information stored in the header. For example, a header of "97 3 98 4 99 2" is associated with an original file of "aaabbbbcc" (3 a's, 4 b's, 2 c's). Use the HuffmanBitReader class to read the header.
- Write a function, **parse_header(header_string),** that takes a string input parameter (the first line of the input file) and returns a list of frequencies. The list of frequencies should be in the same format that cnt_freq() returned in the first part of this assignment (a list/arrary with 256 entries, indexed by the ASCII value of the characters). For the example above, list_of_freqs[97:100] would be [3, 4, 2, 0].
- One you have re-created the list of freqs, pass it to your function **create_huff_tree(list_of_freqs)** to recreate the Huffman Tree used for encoding**.**

- Once you have recreated the Huffman tree, you should have all the information you need to decode the encoded text and write it back out to the **decode_file**.
- Stop reading bits using the HuffmanBitReader when you have written the correct number of characters to the output file. The number of characters to write can be determined from the information in the header file.

3 Tests

- Write sufficient tests using unittest to ensure full functionality and correctness of your program.
- When testing, always consider *edge conditions* like the following cases:
  - Single character files
  - Empty files
  - In your code, you will likely need to "special case" these types of input files
- Assuming that they are working correctly, leave in your tests for the encoding part of the project (3a). You will be required to have 100% code coverage.
- There are sample tests provided in the provided **huffman_decode_tests.py** file. You may/should copy them over to your huffman_tests.py file.

4 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. 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.

5 Submission

You must submit (commit and push to GitHub) the following files:

- **huffman.py**: The functions from the first portion of the assignment, and the newly specified and functions
  - **parse_header(header_string)** takes in input header string and returns Python list (array) of frequencies (256 entry list, indexed by ASCII value of character)
  - **huffman_decode(encoded_file, decode_file)** decodes encoded file, writes it to decode file
- **huffman_tests.py**, containing testcases for the functions specified by the assignment
  - This file should contain tests only for the functions specified above and should only test functionality required by the specification. These tests must run properly on a valid instructor solution and will be tested to see if they can catch bugs in incorrect solutions.
- **huffman_helper_tests.py**, containing all testcases used in developing your solution
  - This file should contain all tests for your solution, including tests for any helper functions that you may have used. These tests will be used to verify that you have 100% coverage of your solution. Your solution must pass these tests.