Project 3

Huffman Coding with Priority Queue

Background

For this project, you will implement a program to encode 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 will be presented in a lecture.

Note:

- This project is based on Lab 7. Complete Lab 7 before starting this project.
- 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.

Implementation

Start by downloading a zip file for this project. The zip file contains huffman.py, huffman_coding.py, huffman_tests.py, and some sample text files to be used in tests. Add functions and data definitions to these files, if not otherwise stated. You should develop incrementally, building test cases for each function as it is implemented. You are also going to use min_pq.py from Lab 7.

1. Data Definition

Let’s write a recursive data definition for a Huffman tree, which is a possibly empty binary tree of HuffmanNodes:

Huffman Tree is one of None or HuffmanNode

The definition is very similar to the definitions of binary trees we have seen before except that the class for each node is called HuffmanNode. This means that each HuffmanNode is a Huffman Tree (either the root node of the entire tree or a subtree).

In huffman.py, write a definition for HuffmanNode class. The class is required to have four fields: char for storing a character, freq for the character’s frequency, left for storing a pointer for the left subtree, and right for the right subtree. Make sure you implement, __eq__ and __repr__ methods as well.

As usual, follow the six steps of the design recipe.

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**.

You are also required to implement your own version of python builtin method __lt__ in the HuffmanNode class. Implementing __lt__ overloads < operator. You want to do so so that you can compare two HuffmanNode objects with < operator. This method, **__lt__(self, other),** returns true if the tree itself comes before other tree. 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 the tree is larger than one node, compare the frequencies of the root nodes and use the letter with the smallest ASCII value in the two nodes as a tiebreaker. Store the smallest letter in the new node. 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'. **This function will be used when you build a min heap of Huffman Nodes**.

2. Functions

The following bullet points provide a guide to implement some of the data structures and individual functions of your program. **You are free to add helper functions but do not change the name of functions and their signatures specified below.**

- **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 encoded text to out_file
- **huffman_decode(list_of_freqs, encoded_file, decode_file)** : decode encoded_file and write the decoded text to decode_file.

2.1 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. Let's call this array **list_of_freqs.** The list_of_freqs is an array (list) of strings of the size 256. The index in this array corresponds to each character's ord() value. For example, the frequency of the letter 'a' is stored at the index 97.

CAUTION: There can be issues with extra characters, such as a newline character, in some systems.

2.2 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.

Write a function, **create_huff_tree(list_of_freqs),** that builds a Huffman tree from a given list of the number of occurrences of characters, **list_of_freqs**, returned by **cnt_freq()** and returns the root node of that tree.

1. Start by creating an array (list) of individual Huffman trees each consisting of a single HuffmanNode containing the character and its occurrence counts. **Ignore characters whose frequencies are 0.**
2. Then, build a **min heap** (The Minimum Priority Queue described in section 3) by heapifying the array.
3. Building the actual tree used for huffman code involves dequeuing a node with the lowest frequency count from the minheap twice and connecting the two nodes to the left and right field of a newly created internal Huffman Node as presented in the lecture. The node that dequeued (del_min) first should go in the left field. In the newly created internal Huffman Node, store the smaller character in ASCII code table order as its character (i.e. new_huff_node.char = min(huff_node1.char, huff_node2.char)).
4. Note that when connecting two HuffmanNodes to the left and right field of a new

internal Huffman Node, that this new node is also a HuffmanNode, but does not contain an actual character to encode. Instead this new internal node should contain an occurrence count that is the sum of the left and right child occurrence counts as well as the smallest (in terms of ord() number) of the left and right character representation.

5. After creating the new internal Huffman Node joining the two dequeued HuffmanNodes, insert the new node (Huffman Tree) back into the **min heap**.
6. This process of joining two Huffman Trees (nodes) from the top of the min heap is continued until there is a single node left in the heap, which is the root node of the Huffman tree. **create_huff_tree(list_of_freqs)** then returns this root node (tree).

2.3 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 and returns an array of huffman code. 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.

For returning an array of huffman code, you may want to initialize a Python list of strings that is initially consists of 256 Nones in **create_code**. For example, create a list by [None] * 256. 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 Nones. Return this list.

2.4 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.

1. The function will call **cnt_freq(in_file)** which will create and return an array (list) of strings of the size 256 and store the frequency of each unique character

presents in the in_file in the array at the index corresponding to the ord() value of the character.
2. Then the **huffman_encode** function passes the array **list_of_freqs** to **create_huff_tree(list_of_freqs), which will ignore characters whose frequencies are 0**.
3. The **huffman_ecode** function then calls **create_code(root_node)**, passing the root node of the Huffman tree the create_huff_tree returns, which in turn returns a list containing Huffman code of each character present in the in_file.
4. Lastly, the **huffman_ecode** function encode characters present in the **in_file** using the Huffman code returned by the **create_code** function, and write the encodings to the **out_file**.

Writing the generated code for each character into a file will actually enlarge the file size , not 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. We leave this to an optional project, which will be introduced toward the end of this course.

2.5 Huffman Decoding Header Information Note

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:

1. 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()**.
2. 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 ord() value of 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.
3. 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 called **huffman_decode(list_of_freqs, encoded_file, decode_file)** (use that exact name) that reads a list of frequencies of characters, **list_of_freqs,** an encoded text file, **encoded_file,** and writes the decoded text into an output text file, **decode_file.** To test **huffman_decode function,** produce the **list_of_freqs** by passing a non-encoded text file to the **cnt_freq().** Pass the **list_of_freqs** to **huffman_decode().** Call **create_huff_tree(list_of_freqs)** in the **huffman_decode function** to build a huffman tree**,** which in turn will be used to decode huffman code to a character.

3. Minimum Priority Queue

Finding the smallest HuffmanTree takes O(n) if you search it using sequential search, or sorting the list of trees in ascending order using the insertion sort. But if you use an efficient data structure such as heap, you can get (del_min) the smallest Huffman Tree in a list in O(log n).

Use your **min_pq.py file** from lab 7.

4. Tests

Write sufficient tests using unittest to ensure full functionality and correctness of your program. Start by using the supplied **huffman_tests.py**, **test1.txt, test2.txt, and test3.txt** to begin testing your programs. Test your encoding output by comparing it with supplied test outputs: test1.out, test2.out, and test3.out.

When testing, always consider *edge conditions* like the following cases:

– 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”.

5. 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.

6. Submission

You must submit the following files to the grader and polylearn:

- min_pq.py, huffman.py, huffman_coding.py
    - The functions specified and any helper functions necessary.
- huffman_tests.py
    - Include test cases you used in developing your programs.
    - Text files you used if any in addition to supplied text files.