Lab 3 Analysis

EN.605.202
Data Structures
Summer 2022

Jose Márquez Jaramillo

Engineering and Applied Science Programs for Professionals
Johns Hopkins University
United States
August 9th, 2022

Abstract

This lab analysis presents a methodology for Huffman coding. It describes in detail the data structures used, the implementation implications using a Priority Queue and a Binary Tree, the mechanism for breaking ties, and explanations of the enhancements incorporated in the code.

Contents

1 About Huffman coding

Huffman coding is a lossless data compression algorithm. Given data represented as some quantity of bits, the Huffman encoder transforms data into fewer bits. Compressed data uses less storage and can be communicated faster than uncompressed data [2].

A particular characteristic of Huffman coding is using a frequency table to assign variable length codes to input characters. Lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code, and the least frequent character gets the largest code. Another important characteristic related to Huffman coding is that decoding a Huffman compressed message requires having the frequency dictionary used for its encoding. For Huffman encoded data, the dictionary must be included along with the compressed data to enable decompression. Even though that dictionary adds to the total bits used, its storage cost is smaller than those bits saved through the compression process [1].

Huffman compression requires three main steps:

1. Building a Huffman tree,
2. getting the Huffman codes,
3. and compressing or decompressing the message.

1.1 Building a Huffman tree

The data members on a Huffman tree depend on the node type. Leaf nodes will have two data members; a character from the input and an integer frequency for that character. Internal nodes will have in addition left and right child nodes.

1.1.1 Breaking ties

There are consequences to improperly organizing the characters while building the tree. This happens both when feeding new items into the tree as well as when assigning children to Internal nodes. According to the assignment and the course office hours, the tie-breaking mechanism implements the following logic:

1. Elements with higher frequencies go to the right and have higher priority.
2. If two elements have the same frequency, internal nodes go to the right and have higher priority.
3. If two elements have the same frequency and are internal nodes, the node with the lowest ranked (A-Z) first character goes left and has less priority. The character representation of nodes is ordered alphabetically; therefore, the first character is always the lowest ranked character for those in a particular node.

1.2 Getting the Huffman codes

Once a Huffman tree has been created and codes exist for each character, each character corresponds to a leaf node. The Huffman code for a character is built by tracing a path from the root to that character's leaf node, appending 0 when branching left or 1 when branching right.

1.3 Compressing or decompressing the message

To compress an input string, the Huffman codes are first obtained for each character. Then each character of the input string is processed, and corresponding bit codes are concatenated to produce the compressed result. In order to decompress Huffman code data, one can use a Huffman tree and trace the branches for each bit, starting at the root. When the final node of the branch is reached, the result has been found. The process continues until the entire item is decompressed.

2 Description of program and methods

The program is ”packaged” according to the specifications in the guidelines. The folder SourceCode contains all of the code generated and used for implementation. An effort was made to make the file parsing as accurate as possible, and several different test cases were implemented to alleviate potential incidences. Enhancements beyond those initially specified in the lab were also implemented to check whether a read line contains a valid prefix expression and the level of information generated in output.

2.1 Data structures

2.1.1 Dictionaries

The main method for data storage used is the Python Dictionary. It is used extensively throughout the program.

2.1.2 Priority Queue

A priority queue acts like a queue in that one de-queues an item by removing it from the front. However, in a priority queue the logical order of items inside the queue is determined by their priority. The highest priority items are at the front of the queue, and the lowest priority items are at the back. When an item is en-queued the new item may move to the front [3]. The implementation of this priority queue is located in the file `priorityqueue.py` The implementation used in this program uses a Python list for storage and has the following methods:

- `__init__` initializes a new instances by creating an empty list.
- `__str__` is an operator overloading of the print function, it prints a concatenation of the items in the priority queue.
- `isempty` returns a True boolean if the priority queue is empty.
- `enqueue` inserts a new item into the queue. This method preserves the order necessary for Huffman encoding, these include the mechanism for breaking ties previously discussed. Inserting items into the priority queue will be $O(n)$ since as the item is inserted, it needs to be sorted and placed according to its priority in the queue.
- `swap` is a helper method used for swapping items in the queue while en-queuing.
- `dequeue` is the method used to remove items from the priority queue. In this instance, the lowest priority item is required, and therefore, the method pops the last item in the queue (the one with the lowest priority).

2.1.3 Binary Tree

Node classes have been defined to create the tree. Since the tree is made up of internal and leaf nodes, two classes are defined. The main difference is that `LeafNode` has None type left and right children while `InternalNode` has been assigned left and right children. Both classes contain:

- a character (for `InternalNode` this is a string ordered alphabetically),
- a frequency used to en-queue and for children assignment,
- and left and right nodes (`LeafNode` has None type left and right children).

The binary Huffman tree and the associated functions to the Huffman tree are stored in `huffmantree.py`. These functions include:

- `createhuffman` calls the priority queue class and creates a new priority queue with all of the leaf nodes. Then it builds a series of internal nodes by popping the last two items in the priority queue and assigning them to the left or right according to the mechanism for breaking ties. The internal node is then en-queued in the priority queue. The loop finishes once the priority queue only contains one internal node (the root node). The function returns the root internal node.
- `gethuffmancodes` recursively traverses the tree and creates a dictionary with all of the leaf nodes in the tree along as keys and the corresponding code.
- `preorderhuffman` prints out a preorder traversal of the tree to the command line. It prints the output from left to right on the screen.
- `compresshuffman` uses the Huffman codes to create a list of the compression codes.
- `decompresshuffman` traverses the Huffman tree given an encoded binary string. It traverses the tree from the root down. If the bit is a zero, it goes left and right otherwise. It moves onto the next character once it encounters a leaf node.

3 Enhancements

3.1 Pre-processing as an enhancement

The pre-processing happens inside the `readfiles.py` file, there are two functions that read in the text and frequency files and store them in dictionaries. When reading the text to be decoded or encoded, each line has a unique dictionary with the following:

- Original string from the file,
- valid string which excludes non-alphanumeric characters (such as spaces and symbols),
- a frequency dictionary of the lines characters,
- a bit count of the valid characters,
- and an uppercase representation of the valid string.

The frequency dictionary by each line is later used as an enhancement to add non-existing characters to the frequency table used for building the Huffman Tree.

3.2 Output as an enhancement

The `processfiles` function in the `readfiles.py` file brings together the program and generates a file that contains:

- a print-out preorder traversal of the Huffman tree generated;
- an output summary which contains the file name, whether new characters were added to the and which ones, and the number of saved bits by the compression;
- a line-by-line printout of the original string, the valid string processed, and the compressed value;
- and finally the frequency table resulting from the operations.

This information is also printed to the command line. When running the file, the user must input:

- The path to the frequency table,
- the path to the text to be compressed or decompressed,
- the character 'C' for compression or the character 'D' for decompression,
- the instruction 'Y' for scanning and comparing the frequency table with the text previous to compression or 'N' for processing without comparing,
- and the path to the output file.

For example: **FreqTable.txt ClearText.txt C Y CompressedText.txt**

4 What I learned and what I would do differently

4.1 What I learned

I have summarized above several concepts which have contributed to the development of this lab, below I will explicitly list some of the concepts that I learned as a consequence of going through this exercise:

- Huffman coding
- Use and implementation of priority queues
- Implementing a binary tree through a linked list
- The effect and importance of breaking ties

4.2 What I would do differently

I would implement the priority queue using a heap instead of a python list. A binary heap would allow the program to en-queue and de-queue in $O(\log n)$ instead of $O(n)$ as it currently happens with the current implementation.

References

[1] Huffman coding | greedy algo-3, July 2022.

[2] R. Lysecky and F. Vahid. *Data Structure Essentials*. zyBooks.com, 2021.

[3] B. N. Miller and D. L. Ranum. *Problem Solving with Algorithms and Data Structures Using Python SECOND EDITION*. Franklin Beedle Associates.