Cobarrubias, Ivan
Gaac, Marc Henry
Quijano, Julian Phillip

HUFFMAN ENCODING IMPLEMENTATION

Storing data for either transportation or keeping history is an important aspect in several fields in the modern age. Due to this, compressing said data is also important to save space. To remedy this problem, our group decided to implement Huffman Encoding.

This method requires the use of a data tree in the assignment of a 0 and 1 value for each character present in a document. To build this tree, the program uses a heap to efficiently arrange the characters in terms of their frequency. The build_tree function runs in O(n logn) time complexity, the generation of the code at O(n), and the decode and encode functions run at O(m). It is also a greedy algorithm for lossless compression. At every step, it combines the 2 nodes with the least frequency to ensure higher frequency characters have shorter codes.

One of the difficulties our group experienced in implementing Huffman Encoding is using the tkinter module for a user input. Initially we were using google colab and we later found out that the module is not supported hence the switch to the local based interpreters such as jupyter notebook.

The edge cases are also points of struggle. If the input was a string of a repeating character or a singular character, it only makes a single node during the construction of the Huffman tree. Due to this we decided that the Huffman class would detect such cases and just return the frequency of the said character. Data compression is not needed for these cases as they already take a small amount of space. The class also deals with empty strings in a similar manner: it just says the string is or file is empty.

Due to some constraints, our group only managed to implement the Huffman Encoding for text files but the mechanism is very capable of compressing other things such as images. This program however is more than capable of doing the task it was meant to do.