# Assignment 6 – Huffman Coding

Eric Lee

CSE 13S – Spring 2023

## Purpose

This program compresses a data file using Huffman coding. The program reads the data file, computes the corresponding Huffman code, and writes the code to a new output file. This compression will allow data to be stored or transferred using less memory while not losing any data.

## How to Use the Program

To run the program, the user first needs to compile and run huff.c. The user can use a shell script called runtests.sh that automatically sets the .txt files in the files directory as the input files and sets the output files. It also decompresses the Huffman coded files and sets those output files too. Finally, it checks for the difference between the decompressed Huffman coded files and the original files. The user can also use command line options "-i" and "-o" to set the input and output filenames.

## Program Design

The program consists of the files bitwriter.c, node.c, pg.c, and huff.c, where the main function is located. The file bitwriter.c writes bits from the input file to the buffer and reads bits from the buffer to the input file. The file node.c makes a binary tree to be used by the Priority Queue, as well as prints the tree. The file pq.c creates a Priority Queue using a linked list to be used by the Huffman coder. The file huff.c actually does all the steps of Huffman coding, including creating a histogram of bytes from the input file, creating a code tree, filling the entry-code table, and writing the Huffman coded file.

### Data Structures

The file bitwriter.c uses a struct BitWriter that has a pointer to a buffer, a 8-bit variable 'byte' to store individual bits before writing to the buffer as a complete byte, and another 8-bit variable 'bit_position' to see how many bits have been written to the variable 'byte'.

The file node.c uses a struct Node that holds all the data needed for each node of the binary tree. It holds data about the node like the symbol, weight, code, and length of the code. It also has a

pointer to its left and right child nodes.

The file pg.c has two structs, ListElement and PriorityQueue. ListElement has a pointer to a binary tree and a pointer to the next element in the linked list. PriorityQueue just has a pointer to the linked list.

The file huff.c has an array of doubles for the histogram, as well as a struct called Code for the Code Table.

## Algorithms

The file bitwriter.c uses the function bit_write_bit to essentially do all of the writing of the bytes from the input file to the buffer. It collects this data one bit at a time, using the variable bit_position from the struct BitWriter to keep track of how many bits have been written before a whole byte can be collected. Each bit is left shifted by the bit_position variable into the byte. When the bit_position reaches the end of a byte, it will write that whole byte into the buffer and flush.

The file pq.c uses the function enqueue to add a new node(tree) to the priority queue. It creates a new ListElement object that points to the tree that needs to be inserted. It then has a few cases to see how to properly insert the ListElement into the priority queue depending on its state. If the priority queue is empty, it simply adds it to the ListElement at the end. If the new tree has a smaller weight than the one at the head of the priority queue, it is inserted at the beginning of the priority queue. If it goes somewhere else, it first traverses the queue to find the spot where the tree in the priority queue has a lesser weight than the new one, and inserts it there. All comparisons of the trees are done with the function pq_less_than. The dequeue function simply removes the tree at the head of the priority queue and sets it to the pointer passed in.

The main function in the file huff.c first fills a double array called histogram with the data from the buffer and returns the size of the file it read from. It then creates a new Huffman tree with the function create_tree using the data from the histogram and returns it, as well as keeping track of the number of leaf nodes in the tree. Then, an array of 256 Code structs is created and filled using the recursive function fill_code_table. The input file is then closed and reopened to start reading it again from the beginning. Then it calls the function huff_compress_file to write the output file according to Huffman coding. The function huff_compress_file calls the recursive function huff_write_tree to write the data from the Huffman tree to the output file and then writes the data from the Code struct array created earlier. Finally all the data is flushed and the files are closed, using the function delete_tree to free all the nodes in the Huffman tree.

## Function Descriptions

The file bitwriter.c has functions bit_write_open and bit_write_close to create the buffer and flush and free its memory. It also has a function bit_write_bit that collects individual bits into bytes and writes them to the buffer. The functions bit_write_uint8, bit_write_uint16, bit_write_uint32, write in sizes of one byte, two bytes, and four bytes by repeatedly calling bit_write_bit.

The file node.c has functions node_create and node_free for creating and setting a node and for freeing a node’s memory. It also has a function node_print_tree but that is mainly used for debugging purposes.

The file pq.c has the functions pq_create and pq_free to allocate and free the memory associated with the priority queue. The function pq_is_empty checks if the priority queue is empty. The function pq_size_is_1 checks if the priority queue has exactly one element, return false if otherwise. The function enqueue and dequeue adds a new ListElement with a tree to the priority queue and removes the ListElement with the tree with the lowest weight. The function pq_print prints out the elements in the priority queue for debugging.

The functions in huff.c have been explained in the Algorithms section.

## Results

The program works as expected, with the decompressed versions of each file matching the original file. It could be improved by adding more compression algorithms, but that is beyond the scope of this project. It can also be improved by outputting the difference between the size of the original and compressed files to see how effective Huffman coding was at compression. This may be useful because Huffman coding may not be the best option for every data file.