Assignment 6 – Huffman Coding

Priya Chaudhari

CSE 13S – Spring 2023

Purpose

The purpose of this program is to implement Huffman compression, which is a lossless data compression algorithm. Huffman compression entails creating a histogram with unique symbols in data and a tree that will analyze the frequencies of symbols from the input data and assigns each symbol to a unique bit-string code. It's a specific type of compression that would work best with input data with low entropy as it optimizes and relies on the repetition of data represented as symbols. This is because it notably has shorter, more efficient code if a symbol frequently reappears. My program specifically will contain one significant source file named huff.c which is responsible for being the "transmitter" stage of my program("Operates on the message to produce a signal suitable for transmission over the channel."). My executable, huff.c, will be responsible for implementing ADT which entails including functions and other source/header files involving nodes (node.c), bit-by-bit writers (bitwriter.c), and concepts like priority queuing (pq.c).

How to Use the Program

In order to use the program, the user needs to enter the command "make" within the terminal which will compile the Makefile, and then enter one of the 3 main command line options to bring data into my program. Below is a list of all the command line options (inputs and outputs) and what the user would need to enter for a specific purpose.

1. -i: the input file which my program will read and set the name to, the user must enter their filename as an argument
2. -o: the output file that my program will write to and set the name to, the user must enter their filename as an argument, and it will default to stdout
3. -h: will print a help message to stdout

Running the following commands will compress and decompress the needed files

- First run one of these which will compress the file with my huff file
- ./huff -i files/zero.txt -o files/zero.huff
- ./huff -i files/one.txt -o files/one.huff
- ./huff -i files/two.txt -o files/two.huff
- ./huff -i files/test1.txt -o files/test1.huff
- ./huff -i files/test2.txt -o files/test2.huff
- ./huff -i files/color-chooser-orig.txt -o files/color-chooser-orig.huff
- Second you decompress the file with the provided decompress file with the corresponding test file you ran previously

- ./dehuff-arm -i files/zero.huff -o files/zero.dehuff
- ./dehuff-arm -i files/one.huff -o files/one.dehuff
- ./dehuff-arm -i files/two.huff -o files/two.dehuff
- ./dehuff-arm -i files/test1.huff -o files/test1.dehuff
- ./dehuff-arm -i files/test2.huff -o files/test2.dehuff
- ./dehuff-arm -i files/color-chooser-orig.huff -o files/color-chooser-orig.dehuff
- Then, run the comparisons command line to report differences between the compress and decompress files
- diff files/zero.txt files/zero.dehuff
- diff files/one.txt files/one.dehuff
- diff files/two.txt files/two.dehuff
- diff files/test1.txt files/test1.dehuff
- diff files/test2.txt files/test2.dehuff
- diff files/color-chooserig.txt files/color-chooser-orig.dehuff

Program Design

The program is designed with 4 primary source files- 3 of which are files (bitwriter.c, node.c, and pq.c) and one main file which will be executable with my main function for this program (huff.c).

0.1 Main Files:

- Files (authored by me): bitwriter.c (contain BitWriter functions), node.c (contains my node functions), pq.c (responsible for implementing the priority queue functions), and Makefile
- Executable/Main file (authored by me): huff.c (main executable which is the encoder for my Huffman compression algorithm)
- Header files (provided and unmodified): bitwriter.h, io.h, node.h, pq.h
- Test files provided: io-x86/_64.a, pqtest.c, nodetest.c, bwtest.c

0.2 Data Structures/Algorithms:

1. BitWriter: in bitwriter.c I will have to implement several functions which will involve opening, closing, and writing bits (in different bit integer sizes like 8, 16, and 32). This is an abstract data type that provides a bit-wise interface for writing files. It contains the following fields:
   (a) underlying_stream: A pointer to a Buffer object used for file writing.
   (b) byte: An 8-bit buffer to collect bits before writing to the underlying stream.
   (c) bit_position: The current position (0-7) within the byte buffer.
2. Node: in node.c I will be demanded to implement the ADT binary tree using nodes which will involve creating, freeing, and printing nodes and the tree to which they belong to. It represents a node in a binary tree used for Huffman coding. It has the following fields:
   (a) symbol: The symbol (byte) associated with the node.

(b) weight: The weight or frequency of the symbol.

(c) code: The Huffman code assigned to the symbol.

(d) code_length: The length of the Huffman code.

(e) left: A pointer to the left child node.

(f) right: A pointer to the right child node.

3. Priority Queue: in pq.c, I will also be creating, freeing, emptying, enqueueing, dequeuing, and printing a priority queue through the use of linked lists. This abstract data type implements a priority queue using a linked list. It contains the following fields:

   (a) list: A pointer to the head of the linked list representing the queue. Each element in the list is a pointer to a Node object.

4. Compression: in huff.c all the above complex ADT structures will be braided with one another to perform Huffman compression

5. ListElement: it represents an element in the priority queue's linked list. It contains the following fields:

   (a) tree: A pointer to the Node object stored in the element.

   (b) next: A pointer to the next element in the linked list.

0.3 Algorithms

Huffman Coding Algorithm

1. Frequency Counting: Scans input data and counts each byte

2. Build Huffman Tree: Uses frequency counting and builds a binary tree with this

3. Assign the Huffman Codes: traverses through the Huffman tree which results in a compressed representation of the data, where the codes are concatenated together.

4. Encoding: encodes the input data by replacing each symbol with its corresponding Huffman code.

5. Decoding: this will read the encoded bits one by one, traverses the tree starting from the root, and moves left or right based on the encountered bit until it reaches a leaf node.

Priority Queue Algorithm

1. Enqueue: inserts a new element (Node) into the priority queue while maintaining the order based on the weight (frequency) of the elements.

2. Dequeue: removes and returns the element with the highest priority (lowest weight) from the priority queue.

3. Merge: Takes the two elements with the highest priority (lowest weight) and merges them in order to build the Huffman tree

Function Descriptions

node.c

1. Node *node_create(uint8_t symbol, double weight : responsible for creating my nodes, will require an initialization of pointer to node struct and memory allocation and returning this pointer.

2. void node_free(Node **node): responsible for freeing the nodes that I created in node_create while involves calling the function free().

3. void node_print_tree(Node *tree, char ch, int indentation): will utilize pseudocode provided to us (provided below).

```
    void node_print_tree(Node *tree, char ch, int indentation) {
        if (tree == NULL)
            return;
        node_print_tree(tree->right, '/', indentation + 3);
        printf("%*cweight = %.0f", indentation + 1, ch, tree->weight);

        if (tree->left == NULL && tree->right == NULL) {
            if (' ' <= tree->symbol && tree->symbol <= '~') {
                printf(", symbol = '%c'", tree->symbol);
            } else {
                printf(", symbol = 0x%02x", tree->symbol);
            }
        }

        printf("\n");
        node_print_tree(tree->left, '\\', indentation + 3);
}
```

pq.c

1. pq_create(): this function creates and initializes a PriorityQueue object. It allocates memory for the object using calloc() and initializes its fields. calloc() is used to allocate memory and initialize all fields to 0 (NULL) values. It basically ensures that the newly created PriorityQueue object starts with an empty list.

2. pq_free(): This function frees the memory allocated for the PriorityQueue object and sets the pointer to NULL. It uses free() to deallocate the memory pointed to by q and then sets the pointer to NULL.

3. pq_is_empty(): a boolean function that checks if the PriorityQueue is empty by looking at its list field. Essentially, it checks if the list of the PriorityQueue is NULL. If it is, the PriorityQueue is considered empty and it returns true otherwise, it returns false.

4. pq_size_is_1(): This function checks if the PriorityQueue has exactly one element. It looks if the list field of the PriorityQueue is not NULL (at least one element) and if the next field of the first element is also NULL (indicating no elements after the first one). If both conditions are met, the function returns true, proving the size of the PriorityQueue is 1 and else it returns false.

5. enqueue(): a void function that takes in a pointer to a PriorityQueue object (q) and a pointer to a Node object (tree) and has no output. This function essentially inserts a Node object into the PriorityQueue while maintaining the order based on the weight of the Nodes. Essentially, this function's logic is the foil to the dequeue function and first allocates memory for a new ListElement object (e) and initializes its fields. Then, it checks the current state of the PriorityQueue and executes the appropriate logic.

6. dequeue(): another void function that takes in a pointer to a PriorityQueue object (q) and a pointer to a pointer to a Node object (tree) and has no output. The function's purpose is to remove the highest priority (lowest weight) Node from the PriorityQueue and return it through the tree parameter.

7. pq_print(): a void function that prints the tree of the queue (q).

```
        void pq_print(PriorityQueue *q) {
            assert(q != NULL);
            ListElement *e = q->list;
            int position = 1;
            while (e != NULL) {
```

```
                if (position++ == 1) {
                    printf("============================================\n");
                } else {
                    printf("--------------------------------------------\n");
                }
                node_print_tree(e->tree, '<', 2);
                e = e->next;
            }
                printf("============================================\n");
}
```

bitwriter.c

1. bit_write_open(): takes in a filename (const char *) and outputs a pointer to a newly created BitWriter object. It opens a file for writing and creates a BitWriter object to write bits to the file. I first ensured the function allocates memory for a BitWriter object and opens the file using the write_open() function. If the memory allocation or file opening fails, appropriate error messages are printed, and the function returns NULL. Otherwise, the BitWriter object is initialized with the underlying file stream and the initial values of the byte and bit_position fields.

2. bit_write_close(): the inputs are a pointer to a pointer to a BitWriter object (pbuf) and has no outputs. This function closes the underlying file stream associated with the BitWriter object, frees the memory allocated for the BitWriter object, and sets the pointer to NULL.

3. bit_write_bit(): the input of this function includes a pointer to a BitWriter object (buf), bit value (uint8_t), and has no outputs. This function writes a single bit to the BitWriter object's underlying stream. This function writes a single bit to the BitWriter object's underlying stream. It increments the bit position and utilizes bitwise operations.

4. bit_write_uint8(), bit_write_uint16(), bit_write_uint32(): the inputs is the pointer to a BitWriter object (buf) and the value to be written (uint8_t, uint16}t, uint32_t) and outputs nothing. It's key purpose is to write an unsigned integer value to the BitWriter object's underlying stream as a sequence of bits. The functions iterate over the bits of the input value, starting from the least significant bit (LSB) to the most significant bit (MSB) using bitwise operations and bit_write_bit().

huff.c

1. fill_histogram(): takes in a buffer pointer (inbuf) and a histogram array pointer (histogram) and outputs the size of the input file in bytes (uint64_t). The purpose of this function reads the bytes from the input buffer and fills the histogram array with the frequency of each byte value. It also applies a hack by incrementing the counts of byte values 0x00 and 0xff by one. It iterates over each element of the histogram array and initializes it to 0. Then, it reads bytes from the input buffer using the read_uint8() function until the end of the file is reached. For each byte read, the histogram and file size are incremented. Lastly, the counts of byte values 0x00 and 0xff are also incremented by one.

2. create_tree(): the inputs of this function is a histogram array pointer (histogram) and a pointer to the variable holding the number of leaves (num_leaves). It then outputs the root node of the Huffman tree (Node *). This function achieves creating a Huffman tree based on the histogram frequencies using a priority queue.

3. fill_code_table(): the inputs of this function are a code table array (code_table), a current node in the Huffman tree (node), the current code (uint64_t), and the length of the current code (code_length). It doesn't have an output as it is a void function. The main purpose of this recursive function is it fills the code table with the Huffman codes for each symbol in the tree. It does this by first checking if the current node is a leaf node (has no left or right child). If it is then it'll store the current code and code length in the code table for the symbol of the leaf node. If the node is not a leaf node, the function

recursively calls itself for the left child. Then, it updates the current code by setting the bit at the code length position to 1 and recursively calls itself for the right child, passing the updated code and code length. This function ensures that the code table is filled with the correct codes for each symbol in the Huffman tree, making it essential for creating an accurate Huffman compression algorithm. The provided pseudocode is what I followed.

```
if node is internal
    /* Recursive calls left and right. */
    fill_code_table(code_table, node->left, code, code_length + 1);
    code |= 1 << code_length;
    fill_code_table(code_table, node->right, code, code_length + 1);
else
    /* Leaf node: store the Huffman Code. */
    code_table[node->symbol].code = code;
    code_table[node->symbol].code_length = code_length;
```

4. huff_write_tree(): the main inputs of this are a BitWriter object (outbuf) and current node in the Huffman tree (node) and no outputs as a void function. The main purpose of this recursive function is it writes the Huffman tree structure to the output buffer in a pre-order traversal manner. It follows the pseudocode provided below.

```
huff_write_tree(outbuf, node)
    if node is an internal node
        huff_write_tree(node->left)
        huff_write_tree(node->right)
        1 0
    else
        // node is a leaf
        1 1
        8 node->symbol
```

5. huff_compress_file(): the main inputs of this BitWriter object for output (outbuf), Buffer object for input (inbuf), File size (filesize), Number of leaves in the Huffman tree (num_leaves), Huffman tree root (code_tree), Code table (code_table). It's a void function therefore it has no output. It's purpose is it performs the Huffman compression of the input file by writing the compressed data to the output buffer. It follows the provided pseudocode below.

```
huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
    8 'H'
    8 'C'
    32 filesize
    16 num_leaves
    huff_write_tree(outbuf, code_tree)
    for every byte b from inbuf
        code = code_table[b].code
        code_length = code_table[b].code_length
    for i = 0 to code_length - 1
        /* write the rightmost bit of code */
        1 code & 1
        /* prepare to write the next bit */
        code >>= 1
```

6. free_tree(): the inputs are the root node of the Huffman tree (node) and it's a void function so no outputs are produced. This is a recursive function that frees the memory allocated for the Huffman

tree nodes. It first checks if the current node is NULL which indicates it's reached the end of a branch. If not, it recursively calls itself for the left and right children of the current node. After that, it frees the memory allocated for the current node using node_free() as a secure way to truly free the tree. This process ensures that all nodes of the Huffman tree are freed properly.

Results

Thankfully, my code successfully achieves what it should and my pipeline passes and command line options to test it work out. I think some of my functions could be cleaner and there definitely needs to be more comments throughout my code to clarify what I am doing in each chunk. I did a good job at that in the beginning for the helper .c files but I noticed while I was writing my executable I was really lacking in writing those comments. I also feel my Makefile could be cleaner or more consolidated but it successfully compiles my program.

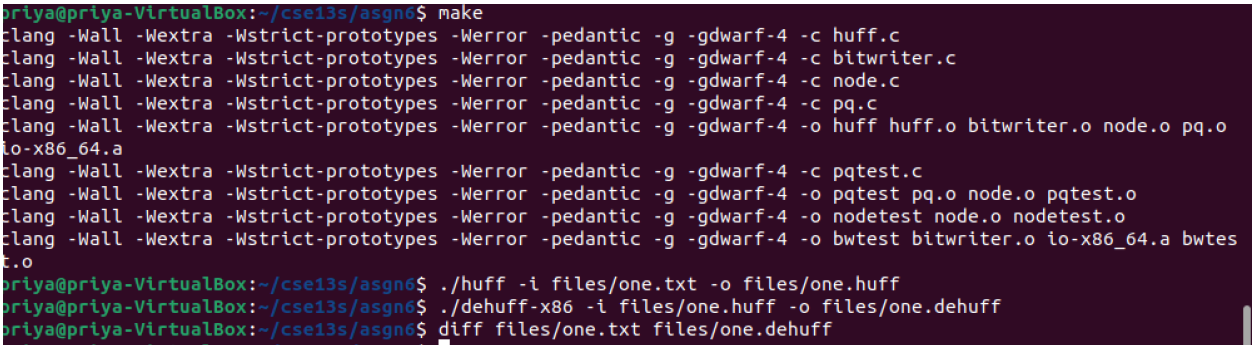

Figure 1: Screenshot of the program running.