Assignment 6 – Huffman Coding

Noah Meisel

CSE 13S – Spring 2023

Purpose

The purpose of this program is to compress data files using Huffman encoding to reduce the number of bits needed to represent each symbol in the data file. The program reads in a data file specified by the user, and after compressing the data into less bytes than the original it will store the new file in another specified file for the user. The reason for creating this sort of program is to help conserve disk space in a computer system, as well as speed up upload/download speeds when sending or receiving a file. The new file can be decompressed with the program dehuff-x86-64.a, which was provided in the asgn6 PDF.[1]

How to Use the Program

To use this program the user will need a file to pass as an argument to the program.

To compile the program, run **make** in the terminal to compile and link all the provided .c files into one executable file "huff".

The executable file "huff" is run in conjunction with the following three command-line options.

```
-i filename: sets a file for the program to read from.
-o filename: sets a file for the program to write to.
-h: Prints a help message with these command-line option descriptions to the console
```

The "huff" executable can also be run with the shell script **./runtests.sh**, which was provided in the asgn6 PDF. [1]

Program Design

This program is split across four different .c files; huff.c, bitwriter.c, node.c, and pq.c. Huff.c is the main file through which the functions implemented in the other three files run. Huff.c contains the actual huffman compression algorithim, and the main switch statement to manage command-line options. Bitwriter.c contains a library of functions to write bits into a buffer and then into the compressed file. Bitwriter.c utilizes a separate library of functions for reading and writing bytes.These functions are described in more detail in the design document for Asgn5[2]. Node.c contains the implementation for the tree data structure, and pq.c contains the implementation for our Priority Queue. The specifications for these data types can be found directly below.

The main body of huff.c consists of four main function calls to execute the huffman encoding process.

```
Call fill_histogram
Call create_tree
Call fill_code_table
Call huff_compress_file
```

Data Structures

Bitwriter

Though not a true ADT, the bitwriter structure is important to see in order to understand the function descriptions that utilize it. The C code for the structure is as follows[1]:

```
struct BitWriter {
    Buffer *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
};
```

Binary Tree

The Binary Tree data structure is implemented as a structure called node. Each node has a symbol to represent it when printing it out, a weight, a code and code length(for use with our huffman compression algorithim) and two pointers to other nodes called left and right. This simple structure can be used to replicate a binary tree, where each node of the tree is accessed by following the pointers from the root node to whichever child you are accessing through the left and right Node pointers.

The Node Structure is defined as follows[1]

```
struct Node {
    uint8_t symbol;
    double weight;
    uint64_t code;
    uint8_t code_length;
    Node *left;
    Node *right;
};
```

Priority Queue

The Priority Queue is a linked list that stores pointers to trees. The weights of each trees root node are used to order each element of the list, creating a priority queue. The actual PriorityQueue structure just contains a pointer to a ListElement, which you can use to access the other elements in the linked list by setting the head of the queue to the next element pointed to by the current head.

The linked list and Priority Queue structure implementation is as follows[1]:

```
struct ListElement {
    Node *tree;
    ListElement *next;
};

struct PriorityQueue {
    ListElement *list;
};
```

Code

The Code structure is used to keep track of the code for each symbol in the Huffman tree as well as the length of each code. The code for it is as follows[1]:

```
typedef struct Code {
    uint64_t code;
    uint8_t code_length;
} Code;
```

Algorithms

Huffman Tree Algorithm[?]

```
while Priority Queue has more than one entry
    Dequeue into left
    Dequeue into right
    Create a new node with a weight = left->weight + right->weight
    node->left = left
    node->right = right
    Enqueue the new node
```

Huffman Traversal Algorithm[?]

```
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;
```

Huffman Compression Algorithim[1]

```
Write 8 bit value H into outbuf
Write 8 bit value C into outbuf
Write 32 bit value filesize into outbuf
Write 16 bit value num_leaves into outbuf
Call huff_write_tree(outbuf,code_tree) to write the tree contents to outbuf.

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
        bit_write_bit(outbuf, code & 1)
        code >>= 1
```

Huffman Write Tree Algorithim for Helper Function[1]

```
if node is internal:
    huff_write_tree(node->left)
    huff_write(tree(node->right)
    bit_write_bit(outbuf, 0)
else:
    bit_write_bit(outbuf, 1)
    bit_write8(outbuf, node->symbol)
```

Enqueue Algorithim

```
Create ListElement e and set its tree element to input tree
If queue is empty:
    set head to e
Else if weight of e's tree < weight of queue heads tree:
    Make e point to item at queue head
    Make queue head point to e
Else:
    Store the head of the queue in a temp ListElement
    While next queue element exists:
        if e's tree < weight of queue heads tree:
            make e point to next queue element
            make next queue element point to e
            exit loop
        make the queue head point to the next item in the queue

    if the loop was exited with no value set:
        make e point to the next item in the queu
        make the next item in the queue point to e

    set the head of the queue to the temp ListElement
```

Function Descriptions

BitWriter *bit_write_open(const char *filename) Takes in a string and returns a BitWriter structure. Pseudocode is as follows

```
USe calloc to allocate memory for a BitWriter structure
Create a pointer to a buffer object called underlying_stream using the write_open function(see asgn5 de
Set the buffer element of the Bitwriter Structure equal to the underlying_stream
return the BiWriter pointer
```

void bit_write_close(BitWriter **pbuf)

Takes in a pointer to a Bitwriter pointer. Writes all the bytes in the structure to the underlying_stream buffer, and then closes the buffer and frees the BitWriter object.

void bit_write_bit(BitWriter *buf, uin8_t)

Takes a BitWriter struct and a byte and writes one bit from that byte into the buffer

```
if the bit position is greater than 7
    write x into the buffer using write_uint8()
    set the byte value of the BitWriter struct to 0
    set the bit position to 0
if (x & 1)
    byte |= (x & 1) << bit_position
increment the bit_position
```

void bit_write_uint8(BitWriter *buf, uin8_t)

Takes in a byte and and the BitWriter struct and writes all 8 bits one by one by calling the bit_write_bit() function.

void bit_write_uint16(BitWriter *buf, uin16_t)

Takes in a byte and and the BitWriter struct and writes all 16 bits one by one by calling the bit_write_bit() function.

void bit_write_uint32(BitWriter *buf, uin32_t)

Takes in a byte and and the BitWriter struct and writes all 32 bits one by one by calling the bit_write_bit() function.

Node *node_create(uint8_t symbol, double weight) Takes in a one byte symbol and a weight value.

Initalizes a new Node object and sets the symbol and weight fields to the passed values respectivley. Returns the new node.

The idea is to use this node structure to to store different values, and through use in our huffman encoding algorithim create a tree by calling this create function for Node *left Node *right.

void node_free(Node **node) Takes in a pointer to a Node pointer. Frees the memory taken up by that node, and sets it to NULL.

void node_print_tree(Node *tree, char ch, int indentation)

Takes in a tree, a character to represent the appropriate tree branch, and the level of indenation for each branch. Prints out a human-readble tree to help visualize our binary tree.

PriorityQueue *pq_create(void)

Takes no arguments and returns a pointer to a Priority Queue object which has it's memory allocated in this function call.

void pq_free(PriorityQueue **q)

Takes in a pointer to a pointer to a PriorityQueue and frees the memory that it takes up.

bool pq_is_empty(PriorityQueue *q)

Takes in a pointer to a PriorityQueue and returns True if Empry, false otherwise.

bool pq_size_is_1(PriorityQueue *q)

Takes in a priorityQueue and returns true if it has one element, false otherwise. To do this, this function checks if the next element in the priority queue is empty(and that the current element is not empty).

void enqueue(PriorityQueue *q, Node *tree)

Takes in a pointer to a PriorityQueue and Tree. Creates a new linked-list element for the priority queue and then proceeds to insert the tree into the priority queue by attaching it to the new ListElement and comparing it to all of the other elements tree weights.

bool dequeue(PriorityQueue *q, Node **tree)

Takes in a pointer to a PriorityQueue and a pointer to a pointer to a Tree. Removes the lowest weight element from the queue, and frees it after storing the tree in the tree parameter. Returns true if succesful, false otherwise.

void pq_print(PriorityQueue *q)

Takes in a PriorityQueue, and prints all of the trees stored in it.

bool pq_less_than(Node *n1, Node *n2)

Takes in two different Trees, n1 and n2, and returns true if n1-¿weight ¡ n1-¿weight, false if n1-¿weight ¿ 2-¿weight, and if neither of those are true it returns the result of n1-¿symbol ¡ n2-¿symbol.

Huffman Encoding has not been started yet, and the functions are not implemented so descriptions have not been created. Their defintions are as follows.

uint64_t fill_histogram(Buffer *inbuf, double *histogram)

Takes in a buffer file and a histogram(a 256 element array that will track the number of each byte in the file) and reads all elements of the file into the histogram. Storing an element in the histogram is done by incriminating the value at index byte.

Node *create_tree(double *histogram, uint16_t *num_leaves)

Takes in histogram and the number of leaves that the new tree should have. Using the histogram the function first creates a priority queue by going through the histogram and creating a new tree node from the histograms elements and adding it to the queue. Then the Program utilizies the Huffman Coding Algorithim(see algorithims) to create the huffman tree. Finally, the last remaining element in the queue is returned.

fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length) Takes in a pointer to a Code structure, a pointer to a node and a value code and code length which should be set to 0 for the first call. THis recursive function traverses the Huffman Tree. The pseudocode for this traversal can be found in the algorithms section.

void huff_compress_file(BitWriter *outbuf, Buffer *inbuf, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table) Takes in a BitWriter, a Buffer, the size of the input file, the number of leaves on trhe code tree, the code tree , and the code table. This function takes in the input file and compresses it by using the functions specified above as well as a recursive helper function(see pseudocode).

huff_write-tree(BitWriter *outbuf, Node *code_tree) Takes in a bit_writer buffer file and a code tree. Uses the huff_write_tree algorithim to write the code tree to the buffer file. The pseudocode can be found in algorithims.

Results

The program successfully reads any text file and compresses it, and valgrind shows no errors when it runs. There were two major roadblocks with getting this program to run; the traversal of the priority queue and, getting the tree created properly. When it came to the priority queue, the main challenge was understanding how to move along the linked list, and really at the core of that was being comfortable incrementing the location the queue was pointing at by setting it equal to the next element. Once I got that done the other major hurdle I encountered was creating the huffman tree. It took multiple hours of debugging and going to the incredible tutor Ben for assistance for me to realize that when I was calling enqueue the values I was passing (Node *left and Node *right) needed to be initialized to NULL, not created using calloc(as that was already done in node_create.

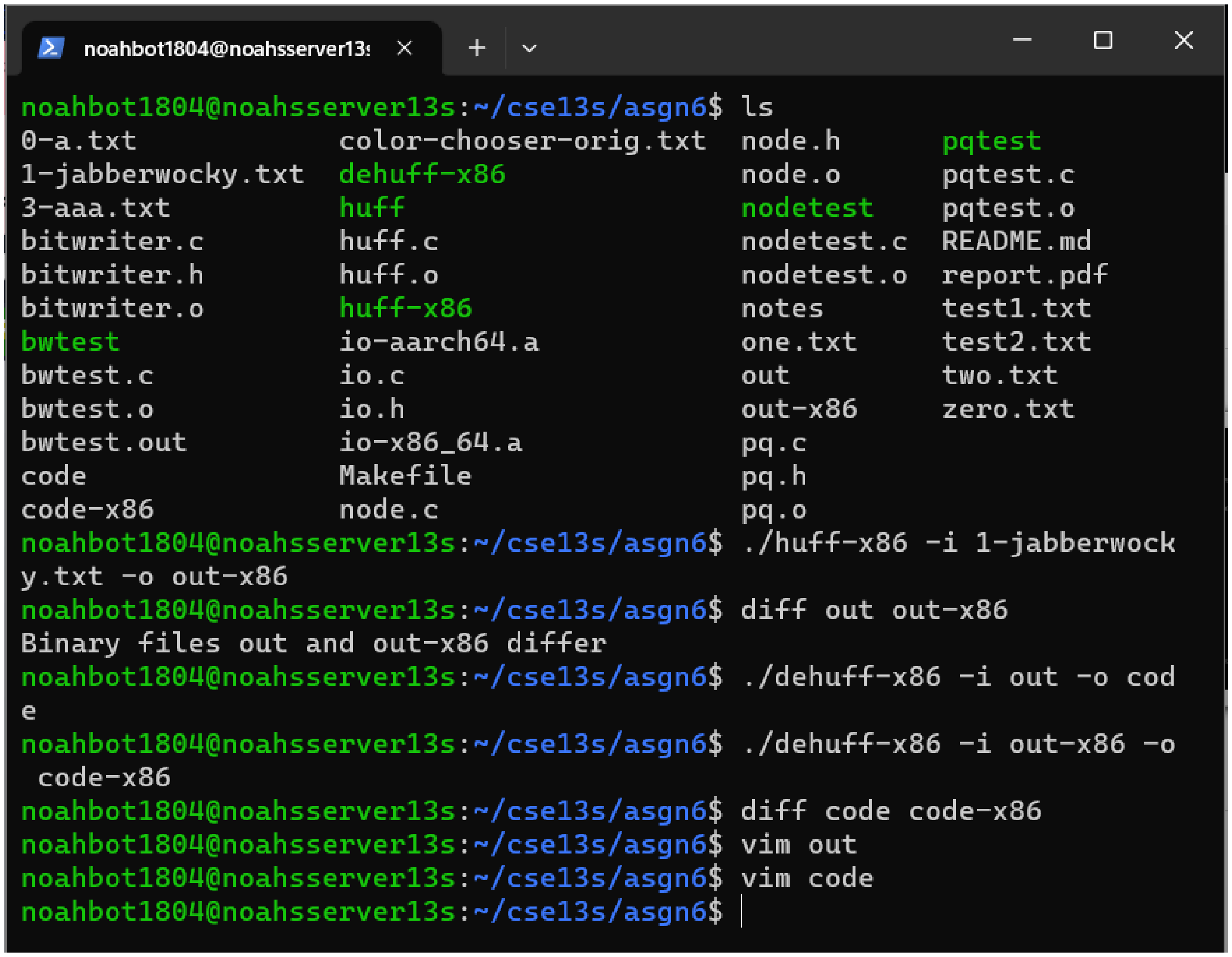
Error Handling

The only ”error” that my program is encountering currently is the compressed file my encoder produces does not match the version that the binary encoder provided in the asgn6PDF[1] produces. I would imagine this is due to hardware differences with my computer and how it stores bytes, but I am really not sure. My program still is able to convert the produced file back into it’s original text format, so it seems to not be an issue in the functionality.

References

[1] Dr. Kerry Veenstra and Ben Grant. *Huffam Coding.* CSE13s, UCSC.

[2] Noah Meisel. *Assingment 5 - Color Blindness Simulator.* CSE13s, UCSC.



noahbot1804@noahsserver13s

```
noahbot1804@noahsserver13s:~/cse13s/asgn6$ ls
0-a.txt              color-chooser-orig.txt  node.h      pqtest
1-jabberwocky.txt    dehuff-x86              node.o      pqtest.c
3-aaa.txt            huff                    nodetest    pqtest.o
bitwriter.c          huff.c                  nodetest.c  README.md
bitwriter.h          huff.o                  nodetest.o  report.pdf
bitwriter.o          huff-x86                notes       test1.txt
bwtest               io-aarch64.a            one.txt     test2.txt
bwtest.c             io.c                    out         two.txt
bwtest.o             io.h                    out-x86     zero.txt
bwtest.out           io-x86_64.a             pq.c
code                 Makefile                pq.h
code-x86             node.c                  pq.o
noahbot1804@noahsserver13s:~/cse13s/asgn6$ ./huff-x86 -i 1-jabberwock
y.txt -o out-x86
noahbot1804@noahsserver13s:~/cse13s/asgn6$ diff out out-x86
Binary files out and out-x86 differ
noahbot1804@noahsserver13s:~/cse13s/asgn6$ ./dehuff-x86 -i out -o cod
e
noahbot1804@noahsserver13s:~/cse13s/asgn6$ ./dehuff-x86 -i out-x86 -o
 code-x86
noahbot1804@noahsserver13s:~/cse13s/asgn6$ diff code code-x86
noahbot1804@noahsserver13s:~/cse13s/asgn6$ vim out
noahbot1804@noahsserver13s:~/cse13s/asgn6$ vim code
noahbot1804@noahsserver13s:~/cse13s/asgn6$ 
```

Figure 1: out is the binary file for my encoder, and out-x86 is the binary file for the provided encoder. They don't match, but upon decoding the new text files have no differences.