Assignment 8 – Huffman Coding

Johann Lee

CSE 13S – Fall 2023

Purpose and How to use.

The purpose of this project is the use Huffman Coding to compress data files. The key component of Huffman coding is to determine which bytes or "symbols" of the input file are most common and switch their representations to use fewer bits. To compensate for this swap, Huffman Coding will use more bits when representing less common symbols. By using this process, less bits are used to represent the file and therefore creating a compressed file.

Using this program consists of first having the needed files and running "make". This will create our binary "huff" and "dehuff". From there a help menu can be printed using "huff -h" which will explain how to use the program. "huff" and "dehuff" require two inputs using "-i" and "-o". These are infile and outfile respectively. With these the command to run the binary is "./huff -i "file" -o "file"". This will output a compressed file that is named whichever file was chosen for -o. To decompress the file run it with "./dehuff -i "file" -o "file"". This will output a decompressed file that should be identical to the original file.

Program Design

There is a significant portion of provided code that I will be exploring within the following functions. The first function is for the Bit Writer.

- This function is based off the following struct

```
Provided by asgn8.pdf by Dr. Kerry Veenstra page 2,3

typedef struct BitWriter BitWriter;

#include "bitwriter.h"
struct BitWriter {
FILE *underlying_stream;
uint8_t byte;
uint8_t bit_position;
};
```

- The first function something that opens the binary filename for write using fopen() and returning a pointer to a BitWriter

```
Provided by asgn8.pdf by Dr. Kerry Veenstra page 3

def bit_write_open(filename):
    allocate a new BitWriter
    open the filename for writing as a binary file,
    storing the result in FILE *f
```

```
        store f in the BitWriter field underlying_stream
        clear the byte and bit_positions fields of the BitWriter to 0
        if any step above causes an error:
            return NULL
        else:
            return a pointer to the new BitWriter
```

- The next function is to close the opened file. This is done using the values in the BitWRiter pointed to by *pbuf, flush any data in the byte buffer, close underlying_stream, free the BitWriter object and setting *pbuf to NULL.

```
            Provided by asgn8.pdf by Dr. Kerry Veenstra page 3

    def bit_write_close(BitWriter **pbuf):
        if *pbuf != NULL:
        if (*pbuf)->bit_position > 0:
            /* (*pbuf)->byte contains at least one bit
            that has not yet been written */
            write the byte to the underlying_stream using fputc()
        close the underlying_stream
        free the BitWriter
        *pbuf = NULL
```

- Now that we can open and close files we need a method to write bits. This is done with the following function. This function writes a single bit using values in the BitWriter pointed to by buf.

```
            Provided by asgn8.pdf by Dr. Kerry Veenstra page 3

    def bit_write_bit(buf, bit):
        if bit_position > 7:
            write the byte to the underlying_stream using fputc()
            clear the byte and bit_position fields of the BitWriter to 0
        set the bit at bit_position of the byte to the value of bit
        bit_position += 1
```

- We also need methods to write bits in forms of 8, 16, and 32. These are what the following functions do.

```
            Provided by asgn8.pdf by Dr. Kerry Veenstra page 4

    def bit_write_uint8(buf, x):
        for i = 0 to 7:
            write bit i of x using bit_write_bit()

    def bit_write_uint16(buf, x):
        for i = 0 to 15:
            write bit i of x using bit_write_bit()

    def bit_write_uint32(buf, x):
        for i = 0 to 31:
            write bit i of x using bit_write_bit()
```

Our next function file is the Bit Readers.

- These functions are based off the followings struct

```
        Provided by asgn8.pdf by Kerry Veenstra page 5

/* bitreader.h */
typedef struct BitReader bitReader;

/* bitreader.c */
#include "bitreader.h"
struct BitReader {
    FILE *underlying_stream;
    uint8_t byte;
    uint8_t bit_position;
}
```

- The first function like Bit Writing opens the file.

```
        Provided by asgn8.pdf by Kerry Veenstra page 5

def bit_read_open(filename):
    allocate a new BitReader
    open the filename for reading as a binary file,
    storing the result in FILE *f

    store f in the BitReader field underlying_stream
    clear the byte field of the BitReader to 0
    set the bit_position field of the BitReader to 8
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new BitReader
```

- The next functions follows the same pattern and closes a file

```
        Provided by asgn8.pdf by Kerry Veenstra page 6

def bit_read_close(BitReader **pbuf):
    if *pbuf != NULL:
        close the underlying_stream
        free *pbuf
        *pbuf = NULL
        if any step above causes an error:
            report fatal error
```

- We also need our main reading function

```
        Provided by asgn8.pdf by Kerry Veenstra page 6

def bit_read_bit(buf):
    if bit_position > 7:
        read a byte from the underlying_stream using fgetc()
        bit_position = 0
    get the bit numbered bit_position from byte
    bit_position += 1;
```

```
        if any step above causes an error:
            report fatal error
        else:
            return the bit
```

- We also need a method to read in 8, 16, and 32 bits

```
            Provided by asgn8.pdf by Kerry Veenstra page 6, 7

    def bit_read_uint8(buf):
        uint8_t byte = 0x00
        for i in range(0, 8):
            read a bit b from the underlying_stream
            set bit i of byte to the value of b
        return byte

    def bit_read_uint16(buf):
        uint8_t byte = 0x0000
        for i in range(0, 16):
            read a bit b from the underlying_stream
            set bit i of byte to the value of b
        return byte

    def bit_read_uint32(buf):
        uint8_t byte = 0x00000000
        for i in range(0, 32):
            read a bit b from the underlying_stream
            set bit i of byte to the value of b
        return byte
```

Our next set of functions are based in the Node file. These nodes are used to make the binary tree.

- Our first item is our struct

```
            Provided by asgn8.pdf by Kerry Veenstra page 7

    /* node.h */
    typedef struct Node Node;

    struct Node {
        uint8_t symbol;
        uint32_t weight;
        uint64_t code;
        uint8_t code_length;
        Node *left;
        Node *right;
    };
```

- We also need a method to create a Node

```
            Provided by asgn8.pdf by Kerry Veenstra page 8

    def node_create(symbol, weight):
        allocate a new Node
        set the symbol and weight fields of Node
```

```
    to function parameters symbol and weight
    if any step above causes an error:
        return NULL
    else:
        return a pointer to the new Node
```

- Since we created a node we also need a method to free it

```
            Provided by asgn8.pdf by Kerry Veenstra page 8

def node_free(Node **pnode):
    if *pnode != NULL:
    free(*pnode)
    *pnode = NULL
```

- Now that we have the nodes we need a method of printing our tree that contains all of our nodes. This will be used for mostly debugging and diagnostics.

```
            Provided by asgn8.pdf by Kerry Veenstra page 9

void node_print_node(Node *tree, char ch, int indentation) {
    if (tree == NULL)
        return;
    node_print_node(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_node(tree->left, '\\', indentation + 3);
}

void node_print_tree(Node *tree){
    node_print_node(tree, '<', 2);
}
```

Now we need to work on creating a Priority Queue to store pointers to trees. This will use the weight field of each node in order to create queue entry and such. This query will be implemented using linked lists.

- First we need our two struct which for this file is the ListElement and PriorityQueue

```
            Provided by asgn8.pdf by Kerry Veenstra page 9 10

/* pq.h */
typedef struct PriorityQueue PriorityQueue;

/* pq.c */
typedef struct ListElement ListElement;
```

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

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

- With these structs we can now create an assortment of functions to interact with the queue.

```
This function allocates a PriorityQueue object and returns a pointer
to it. Return NULL if there is an error

PriorityQueue *pq_create(void){
    allocate priority queue

    if it is null
        return null

    initialize the list to NULL

    return priorityqueue
}

Since we created a priority queue we also have to be able to free it

void pq_free(PriorityQueue **q){
    if *q is not NULL
        free(*q)
        set *q to NULL
}

We also have to check if a pq is empty

bool pq_is_empty(PriorityQueue *q){
    return q == null or q->list == null
}

bool pq_size_is_1(PriorityQueue *q){
    return q is not null and q list is not null and q list next is null
}

bool pq_less_than(ListElement *e1, LIstElement *e2){
    if e1 null or e2 null
        return false

    if e1 tree weight < e2 tree weight
        return true
    else e1 tree weight > e2 tree weight
        return false
    else if e1 tree weight == e2 tree weight
        if e1 e1 tree symbol < e2 tree symbol
            return true
```

```
            else if e1 tree symbol > e2 tree symbol
                return false
    }
```

- Going back to given functions we now have the enque function which inserts a tree into the priority queue

```
            Provided by asgn8.pdf by Kerry Veenstra page 11

    def enqueue(node, tree):
        Allocate ListElement new_element
        set the tree field to the value of the tree function parameter
        if the queue is empty:
            point the queue to new_element
        elif pq_less_than(new_element, q->list):
            New element E1 goes before all existing elements of the list
            Q -----> E2 --> E2 ...
            ^
            E1
            insert new element E1 as the first element of the list
        else:
            Either the new element E3 goes at the end of the list:
            Q --> E1 --> E2
            ^
            E3
            Or the new element E3 goes before an existing element E4:
            Q --> E1 --> E2 -----> E4 ...
            ^
            E3
            In both cases, we are looking for existing element E2,
            and then we put the new element E3 after it.
            find existing element E2 (either E2->next is NULL or E3 < E2->next)
            insert new element after E2
```

- We also need a method to dequeue a tree by removing the queue element with the lowest weight then returning it

```
            if q null and q list null
                exit fatal

            create variables to hold current weight, lowest weight

            while loop find lowest weight in queue

            once ran remove tree
            return tree
```

- Another given diagnostic function is a method to print the priority queue

```
            Provided by asgn8.pdf by Kerry Veenstra page 12

    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");
    }
```

We are now onto the Huffman Coding portion of our function.

- There are five steps when considering huffman coding which is reading the file and counting the frequency of each symbol. We then create a code tree from the histogram based on the symbol frequency. Afterwards, we fill a 256 entry code table, one entry for each byte value. We then rewind the input file and create a huffman coded output file from our input file.
- Our first function is to create a huffman tree

```
            Provided by asgn8.pdf by Kerry Veenstra page 13

    Node *create_tree(uint32_t *histogram, uint16_t *num_leaves)

    1. Create and fill a Priority Queue.
        Go through the histogram, and create a node for every non-zero histogram entry.
        Write the number of nodes added to the Priority Queue to *num_leaves.
        Initialize each node with the symbol and weight of its histogram entry.
        Put each node in the priority queue.

    2. Run the Huffman Coding 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

        *make sure to free the created pq

    3. Dequeue the queue's only entry and return it.
```

- Once we have our huffman tree created we then have to fill the code table mentioned above. This is done with a recursive function that traverses the tre and fills in the code table for each leaf node's symbol.

```
            Provided by asgn8.pdf by Kerry Veenstra page 13

    we first have the used struct
    typedef struct Code {
        uint64_t code;
        uint8_t code_length;
```

```
} Code;

With this struct we then run the recursive

if node is internal:
    /* Recursive calls left and right. */

    /* append a 0 to code and recurse *
    /* (don't neeed to append a 0; it's already there)
    fill_code_table(code_table, node->left, code, code_length + 1);

    /* append a 1 to code and recurse */
    code |= (uint64_t) 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;
```

- Now that we have our code table we are ready to begin compressing our input file.

```
        Provided by asgn8.pdf by Kerry Veenstra page 14

This code compresses the the file
def huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)
    write uint8_t 'H' to outbuf
    write uint8_t 'C' to outbuf
    write uint32_t filesize to outbuf
    write uint16_t num_leaves to outbuf
    huff_write_tree(outbuf, code_tree)
    while true:
        b = fgetc(fin)
        if b == EOF:
            break
        code = code_table[b].code
        code_length = code_table[b].code_length
        for i in range(0, code_length):
            write bit (code & 1) to outbuf
            code >>= 1

This code actually writes the code tree
def huff_write_tree(outbuf, node):
    if node->left == NULL:
        /* node is a leaf */
        write bit 1 to outbuf
        write uint8 node->symbol to outbuf
    else:
        /* node is internal */
        huff_write_tree(outbuf, node->left)
        huff_write_tree(outbuf, node->right)
        write bit 0 to outbuf
```

- We also need our main that runs our entire huff.c program.

```
int main(argc, argv)
    check argc has correct amount of arguments

    const char infile = argv[x]
    const char outfile = argv[y]

    open input file

    bit_write_open(outfile)

    create histogram[256]
    filesize = fill_histogram(input_file, histogram)

    initialize num_leaves
    create huff tree

    create code_table[256]
    fill code table with code table, huff tree, 0, 0

    reset read to beginning of file using fseek

    run huff_compress_file using outputfile, inputfile, filesize, numleaves, hufftree, and

    close inputfile
    bitwriteclose outputfile
    return 0
```

Since we have created our compression code we need a way to decompress the code so that we can get a readable result after using it in its compressed form.

- This is done with a single function that reads the code tree then uses it to decompress the compressed file. It uses stack pointers to store nodes and decompresses using this logic.

```
            Provided by asgn8.pdf by Kerry Veenstra page 15

def dehuff_decompress_file(fout, inbuf)
    read uint8_t type1 from inbuf
    read uint8_t type2 from inbuf
    read uint32_t filesize from inbuf
    read uint16_t num_leaves from inbuf
    assert(type1 == 'H')
    assert(type2 == 'C')
    num_nodes = 2 * num_leaves - 1
    Node *node
    for i in range(0, num_nodes):
        read one bit from inbuf
        if bit == 1:
            read uint8_t symbol from inbuf
            node = node_create(symbol, 0)
        else:
            node = node_create(0, 0)
        node->right = stack_pop()
        node->left = stack_pop()
        stack_push(node)
```

```
    Node *code_tree = stack_pop()
    for i in range(0, filesize):
        node = code_tree
        while true:
            read one bit from inbuf
            if bit == 0:
                node = node->left
            else:
                node = node->right
            if node is a leaf:
                break
        write uint8 node->symbol to fout
```

- The debuff function requires the push and pop stack functions which are implemented as such

```
Node *stack_pop(Node *stack[64], int *top){
    check if top is less than 0
        if so error

    Node temp = stack[top]
    top--
    return temp
}

void stack_push(Node *stack[64], Node *obj, int *top){
    if top is greater than 64
        error

    top++
    stack[top] = obj
}
```

- Since we have our needed functions we can now create our main that will run the entirety of our dehuff.

```
int main(argc, argv)
    check if argc is the right amount of argument
        if not return error

    const char infile = argc[2];
    const char outfile = argc[4];

    bitreader inputfile = bitreadopen(infile)

    file outputile = fopen(outfile)

    dehuff_decompress_file(outputfile, inputfile)

    close outputfile
    close inputfile
    return 0
```

Result

This project was confusing, to say the least. Tracking all the necessary functions and files and how they connect was something that I think I struggled with. Throughout this project I think I gained a much more deep understanding of trees, linked lists, and how they are utilized in terms of Huffman's compression method. I was able to finish this project without any errors or glitches within my program. This includes a lack of any memory leaks when scanned by Valgrind. Scan-build also runs cleanly and returns no bugs.

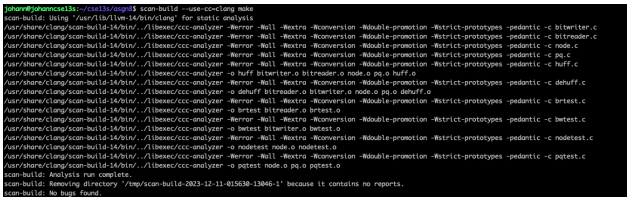

Figure 1: As seen by scan-builds output there are no errors

My project as mentioned before passed all tests including the pipeline and matches both the compression reference and the decompression reference.

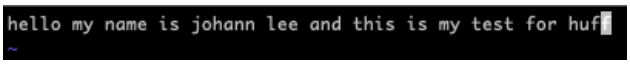

Figure 2: This is the test.txt file and its contents that I will use on both the reference binary and my binary

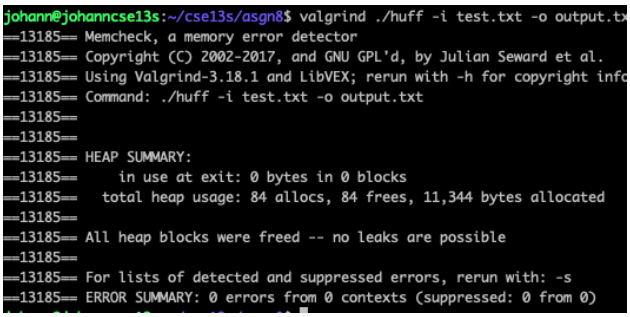

Figure 3: This is an image of my Valgrind run-on huff with results showing no leaks and successfully compressing test.txt

Figure 4: This is an image of the reference output.txt as seen by the window header. It is located within resources/asgn8.

Figure 5: This on the other hand is my huff output.txt and it can be seen that they have identical outputs.

Figure 6: This is the finalref.txt reference and it matches the initial test.txt from before as it should.

Figure 7: This is my final.txt and just like the reference binary, it matches the initial test.txt meaning that the compression and decompression was successful

Going into this final project I was very nervous about whether I would be able to finish it but now that I have I feel that I am that much more prepared for the final with a fortified baseline on understanding assignments and C as a whole entity rather than the small bits and pieces that quizzes are often about. I will of course be studying these bits and pieces as the final comes up, however, I am grateful for being able to do these assignments.

References

asgn8.pdf by Dr. Kerry Veenstra pages 2-15