Assignment 7 – Huffman Coding

Caden Roberts

CSE 13S – Winter 24

Purpose

In this assignment we will work on compressing and decompressing files. Our algorithm will work by determining which symbols are most common, changing their representations into smaller bits, and changing less common symbols into larger bit representations. We will also write a program that can undo the "huff" compression.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

- Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaa")

  ANS:

  The goal of compression is to make file size shorter. The string "aaaaaaaaaaaaaaa" is taking up many chars to represent this, however, we could shorten this string representation considerably, especially because we are compressing a single repeated letter.

- What is the difference between lossy and lossless compression? What type of compression is huffman coding? What about JPEG? Can you lossily compress a text file?

  ANS:

  JPEG is commonly commpressed "lossily". This is because "lossy" compression allows a file to be compressed even further, despite risking some detail or quality. Huffman compressing is a lossless compression. Using lossy compression on a text-file is ill-advised becuase every piece of information is highly important in a text-file.

- Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?

  ANS:

  Huffman coding is best at shortening files with lots of repeating characters. While most files can be processed with Huffman, if there is an even character distribution the file may not compress much further.

- How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?

ANS:

The patterns range from small for repeating characters to large for less frequent characters. This is particularly useful for text files.

- Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in bytes)?

ANS:

My iPhone 14 resolution is about 2532 by 1170 pixels, and 12-megapixel photo size is approximately 3.6 MB.

- If each pixel takes 3 bytes, how many bytes would you expect the picture you took to take up? Why do you think that the image you took is smaller?

ANS:

Total pixels = 2778 × 1284 = 3,566,952 pixels Expected file size = 3,566,952 pixels × 3 bytes/pixel = 10,700,856 bytes (approximately 10.7 MB) The actual image size is smaller than this expected value: photos are typically compressed to reduce file size while maintaining image quality. Compression algorithms remove redundant information, resulting in a smaller file size.

- What is the compression ratio of the picture you just took? To get this, divide the actual size of the image by the expected size from the question above. You should not get a number above 1.

ANS:

3.6/10.7 = 0.336 is the compression ratio of my pciture.

- Do you expect to be able to compress the image you just took a second time with your Huffman program? Why or why not?

ANS:

It depends if there are redundant patterns present in the current image compression or not. If not, the Huffman program won't find a simpler way to represent/compress the file.

- Are there multiple ways to achieve the same smallest file size? Explain why this might be.

ANS:

Lossless Huffman algorithms often do achieve the same smallest file size. If there are all choosing the same representations for repeated patterns then they will end up with the same compressed file.

- When traversing the code tree, is it possible for a internal node to have a symbol?

ANS:

In a properly constructed Huffman tree, only the leaf nodes have symbols associated with them. These leaf nodes represent the actual symbols found in the input data and are assigned unique Huffman codes during the tree construction process. Internal nodes, on the other hand, do not represent specific symbols and are used solely for organizing and constructing the tree.

- Why do we bother creating a histogram instead of randomly assigning a tree.

ANS:

The histogram provides valuable information about the frequency distribution of symbols in the input data. This statistical analysis helps identify which symbols occur most frequently and which ones are less common. This information is crucial for constructing an efficient Huffman tree that minimizes the average code length.

- Relate this Huffman coding to Morse code. Why does Morse code not work as a way of writing text files as binary? What if we created more symbols for things like spaces and newlines?possible

ANS:

While Morse code and Huffman coding share some similarities in their use of variable-length codes, they serve different purposes and are not directly interchangeable. Huffman coding is better suited for data compression and encoding, especially for text files, due to its adaptive encoding scheme based on symbol frequencies. Morse code - to its disadvantage - has a limited symbol set, fixed-length encoding, and lack of binary representation.

- Using the example binary, calculate the compression ratio of a large text of your choosing

ANS:

```
cadenroberts@cse13svm:~/cawrober/asgn7$ ./huff-arm -i test_files/independence.txt -o out.txt
cadenroberts@cse13svm:~/cawrober/asgn7$ cd independence.txt
-bash: cd: independence.txt: No such file or directory
cadenroberts@cse13svm:~/cawrober/asgn7$ cd test_files
cadenroberts@cse13svm:~/cawrober/asgn7/test_files$ wc -c independence.txt | awk '{print $1 *
    8}'
242680
cadenroberts@cse13svm:~/cawrober/asgn7/test_files$ cd ..
cadenroberts@cse13svm:~/cawrober/asgn7$ wc -c out.txt | awk '{print $1 * 8}'
145312
cadenroberts@cse13svm:~/cawrober/asgn7$
```

145312 / 242680 = 0.599 compression ratio.

Testing

I will test many circumstances and combinations of command line inputs, along with different types of input files to huff and dehuff. I will also test each function individually to ensure proper functionality. I will use valgrind and brtest.c, bwtest.c, nodetest.c, and pqtest.c.

How to Use the Program

Running ./huff or dehuff -i inputfile -o outputfile is the basic format to run our huff and dehuff program on the command line. We can also run ./huff -v, for example, to have a USAGE messaged and warning about needing -i included displayed, and ./huff -h will simply display the USAGE message.

```
cadenroberts@cse13svm:~/cawrober/asgn7$ ./huff -v
huff: -i option is required
Usage: huff -i infile -o outfile
      huff -v -i infile -o outfile
      huff -h
cadenroberts@cse13svm:~/cawrober/asgn7$ ./huff -h
Usage: huff -i infile -o outfile
      huff -v -i infile -o outfile
      huff -h
```

Program Design

We utilize user-defined types node, BitWriter, ListElement and PriorityQueue. These structs rely on uint8/16/32_t's and pointers to eachother. The huff and dehuff algorithms will do the work of compressing or decompressing a given file according to command line specifications and by utilizing the user-defined data types and functions from their respective files.

Pseudocode

Psuedocode for my huff and dehuff logic is as follows:

HUFF:

```
function fill_histogram(fin, histogram) returns filesize:
    for i from 0 to 255:
        histogram[i] = 0
    increment histogram[0x00]
    increment histogram[0xff]
    filesize = 0
    while (byte = get_next_byte(fin)) is not EOF:
        increment histogram[byte]
        increment filesize
    return filesize

function create_tree(histogram) returns root_node:
    pq = priority_queue_create()
    if pq is NULL:
        return NULL
    num_leaves = 0
    for i from 0 to 255:
        if histogram[i] > 0:
            node = node_create(i, histogram[i])
            if node is NULL:
                priority_queue_free(&pq)
                return NULL
            enqueue(pq, node)
            increment num_leaves
    while not pq_size_is_1(pq):
        left = dequeue(pq)
        right = dequeue(pq)
        if left is NULL or right is NULL:
            node_free(left)
            node_free(right)
            priority_queue_free(&pq)
            return NULL
        parent_weight = left.weight + right.weight
        parent = node_create(0, parent_weight)
        if parent is NULL:
            node_free(left)
            node_free(right)
            priority_queue_free(&pq)
            return NULL
        parent.left = left
        parent.right = right
        enqueue(pq, parent)
    root_node = dequeue(pq)
    priority_queue_free(&pq)
    return root_node
```

```
function fill_code_table(code_table, node, code, code_length):
    if node is NULL:
        return
    if node.left is NULL and node.right is NULL:
        code_table[node.symbol].code = code
        code_table[node.symbol].code_length = code_length
    else:
        fill_code_table(code_table, node.left, code, code_length + 1)
        code = code OR (1 << code_length)
        fill_code_table(code_table, node.right, code, code_length + 1)

function huff_write_tree(outbuf, node):
    if node is NULL:
        return
    if node.left is NULL and node.right is NULL:
        bit_write_bit(outbuf, 1)
        bit_write_uint8(outbuf, node.symbol)
    else:
        huff_write_tree(outbuf, node.left)
        huff_write_tree(outbuf, node.right)
        bit_write_bit(outbuf, 0)

function huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table):
    bit_write_uint8(outbuf, 'H')
    bit_write_uint8(outbuf, 'C')
    bit_write_uint32(outbuf, filesize)
    bit_write_uint16(outbuf, num_leaves)
    huff_write_tree(outbuf, code_tree)
    set_file_pointer(fin, 0)
    for i from 0 to filesize - 1:
        byte = get_next_byte(fin)
        if byte is EOF:
            break
        code = code_table[byte].code
        code_length = code_table[byte].code_length
        for j from 0 to code_length - 1:
            bit_write_bit(outbuf, (code AND 1))
            code = code >> 1

function main(argc, argv):
    if argc < 2:
        print "Bad Input Files"
        print USAGE
        return 1
    infile_path = NULL
    outfile_path = NULL
    while there are arguments to parse:
        opt = get_next_option(argc, argv)
        switch opt:
            case 'h':
                print USAGE
                return 0
            case 'i':
                infile_path = optarg
                break
            case 'o':
                outfile_path = optarg
                break
            default:
```

```
            print "huff: Invalid option"
            print USAGE
            return 1
    if infile_path is NULL or outfile_path is NULL:
        print "huff: -i and -o options are required"
        print USAGE
        return 1
    fin = open(infile_path, "rb")
    histogram[256] = {0}
    filesize = fill_histogram(fin, histogram)
    num_leaves = 0
    code_tree = create_tree(histogram, num_leaves)
    if code_tree is NULL:
        print "Error creating Huffman tree"
        close(fin)
        return 1
    code_table[256] = {0}
    fill_code_table(code_table, code_tree, 0, 0)
    outbuf = open(outfile_path)
    if outbuf is NULL:
        print "Error opening input file"
        close(fin)
        node_free(code_tree)
        return 1
    huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table)
    print_tree(code_tree)
    close(fin)
    close(outbuf)
    node_free(code_tree)
    return 0
```

DEHUFF:

```
function stack_push(stack, top, node):
    if top >= MAX_STACK_SIZE - 1:
        print "stack overflow lol"
        exit(EXIT_FAILURE)
    stack[++top] = node

function stack_pop(stack, top):
    if top < 0:
        print "stack overflow lol"
        exit(EXIT_FAILURE)
    return stack[top--]

function dehuff_decompress_file(fout, inbuf):
    type1 = bit_read_uint8(inbuf)
    type2 = bit_read_uint8(inbuf)
    filesize = bit_read_uint32(inbuf)
    num_leaves = bit_read_uint16(inbuf)
    assert type1 == 'H' and type2 == 'C'
    num_nodes = 2 * num_leaves - 1
    stack[MAX_STACK_SIZE]
    stack_top = -1
    for i from 0 to num_nodes - 1:
        bit = bit_read_bit(inbuf)
        if bit == 1:
            symbol = bit_read_uint8(inbuf)
            node = node_create(symbol, 0)
        else:
```

```
            node = node_create(0, 0)
            if stack_top >= 1:
                node.right = stack_pop(stack, stack_top)
                node.left = stack_pop(stack, stack_top)
        stack_push(stack, stack_top, node)
    code_tree = stack_pop(stack, stack_top)
    for i from 0 to filesize - 1:
        node = code_tree
        while node.left is not NULL or node.right is not NULL:
            bit = bit_read_bit(inbuf)
            node = node.left if bit == 0 else node.right
        fputc(node.symbol, fout)
    node_free(code_tree)

function main(argc, argv):
    infile_path = NULL
    outfile_path = NULL
    while there are arguments to parse:
        opt = get_next_option(argc, argv)
        switch opt:
            case 'h':
                print USAGE
                return 0
            case 'i':
                infile_path = optarg
                break
            case 'o':
                outfile_path = optarg
                break
            default:
                print "dehuff: Invalid option"
                print USAGE
                return 1
    if infile_path is NULL or outfile_path is NULL:
        print "dehuff: -i and -o options are required"
        print USAGE
        return 1
    inbuf = bit_read_open(infile_path)
    if inbuf is NULL:
        print "Error opening input file"
        return 1
    fout = open(outfile_path, "wb")
    if fout is NULL:
        print "Error opening input file"
        bit_read_close(&inbuf)
        return 1
    dehuff_decompress_file(fout, inbuf)
    close(fout)
    bit_read_close(&inbuf)
    return 0
```

Function Descriptions

- BitWriter *bit_write_open(const char *filename); Takes a string as a file name, and initializes a BitWriter struct, storing 0 in bit_positions and byte, and filename in underlying_stream. Returns a pointer to a BitWriter and NULL if any step fails.
- void bit_write_close(BitWriter **pbuf); Takes a pointer to a pointer to a BitWriter and frees the

memory assosciated with the BitWriter. Sets the pointer to the BitWriter to NULL.

- void bit_write_bit(BitWriter *buf, uint8_t bit); Takes a pointer to a BitWriter, collected 8 bits from bit, storing them in buffer byte and then writing them to underlying_stream.
- void bit_write_uint8(BitWriter *buf, uint8_t x); Takes a pointer to a BitWriter, functioning like bit_write_bit but calling that function 8 times instead of fputc 8 times.
- void bit_write_uint16(BitWriter *buf, uint16_t x); Takes a pointer to a BitWriter, functioning like bit_write_uint8 but calling the bit_write_bit function 16 times instead of 8 times.
- void bit_write_uint32(BitWriter *buf, uint32_t x); Takes a pointer to a BitWriter, functioning like bit_write_uint16 but calling the bit_write_bit function 32 times instead of 16 times.
- BitReader *bit_read_open(const char *filename); Takes a string filename, creates a BitReader and initializes it with filename and 0. Returns a pointer to the BitReader.
- void bit_read_close(BitReader **pbuf); Takes a pointer to a pointer to a BitReader, freeing the memory assosciated with the BitReader and setting the pointer to the BitReader to NULL.
- uint8_t bit_read_bit(BitReader *buf); Uses a pointer to a BitReader and calls fgetc 8 times. Returns the uint8_t bit.
- uint8_t bit_read_uint8(BitReader *buf); Read 8 bits from buf by calling bit_read_bit() 8 times. Return the uint8_t byte.
- uint16_t bit_read_uint16(BitReader *buf); Read 16 bits from buf by calling bit_read_bit() 16 times. Return the uint16_t byte.
- uint32_t bit_read_uint32(BitReader *buf); Read 32 bits from buf by calling bit_read_bit() 32 times. Return the uint32_t byte.
- Node *node_create(uint8_t symbol, uint32_t weight); Takes a uint8_t symbol and weight and allocates a Node, setting its symbol and weight fields, and returning a pointer to the new node if successful and NULL otherwise.
- void node_free(Node **pnode); Takes a pointer to a pointer to a node and frees all memory associated with the node, setting the pointer to the node to NULL.
- void node_print_tree(Node *tree); Takes a pointer to a node and calls node_print_node with tree, '¡', and 2, and prints its associated tree.
- void node_print_node(Node *tree, char ch, int indentation); Recursively calls itself to print the tree, updating ch and indentation as needed.
- PriorityQueue *pq_create(void); Allocates a priority queue, returning a pointer to it on success and NULL otherwise.
- void pq_free(PriorityQueue **q); Takes a pointer to a pointer to a priority queue, frees all associated memory, and sets the pointer to the priority queue to NULL.
- bool pq_is_empty(PriorityQueue *q); Takes a pointer to a priority queue and returns true if the list field is NULL and false otherwise.
- bool pq_size_is_1(PriorityQueue *q); Takes a pointer to a priority queue and returns true if it contains only one element.
- bool pq_less_than(ListElement *e1, ListElement *e2); Takes 2 pointers to ListElements and returns true if the first element's weight field is less than the second element's weight field.
- void enqueue(PriorityQueue *q, Node *tree); Takes a pointer to a priority queue and a pointer to a Node (tree) and inserts the tree into the queue.

- Node *dequeue(PriorityQueue *q); Takes a pointer to a priority queue and removes the queue element with the lowest weight, then returns the pointer to the Node of the deleted tree.
- void pq_print(PriorityQueue *q); Takes a pointer to a priority queue and prints all of its trees.
- uint32_t fill_histogram(FILE *fin, uint32_t *histogram); Takes in a file pointer and uint32_t pointer and updates the histogram array of uint32_t values with the number of each of the unique byte values of the input file, returning the total size of the input file.
- Node *create_tree(uint32_t *histogram, uint16_t *num_leaves); Takes in a pointer to a uint32_t and a uint16_t, creates and returns a pointer to a new Huffman Tree, and also updates the number of leaf nodes in the tree.
- void fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length); Takes pointers to a Code and a Node and a uint64_t and uint8_t, and recursively calls itself to traverse the tree and fill in the Code Table for each leaf node's symbol.
- void huff_compress_file(BitWriter * outbuf, FILE * fin, uint32_t filesize, uint16_t num_leaves, Node * code_tree, Code * code_table); Takes pointers to a BitWriter, file, Code, and Node, and a uint32_t and uint16_t and creates a huffman coded file.
- void huff_write_tree(FILE *outbuf, Node *node); Takes pointers to a File and a Node, and writes the tree to the file.
- void dehuff_decompress_file(FILE *fout, BitReader *inbuf); Takes pointers to a file and BitReader and creates a dehuffman coded file.