Assignment 7 – Huffman Coding

Jason Maheru

CSE 13S – Winter 24

Purpose

- The purpose of this assignment is to create two programs. One program can compress the data of large files and the other program can decompress the data of a given compressed file.

Questions

- Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaaaaaaaaaa")
- 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?
- Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?
- How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?
- Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage (in bytes)?
- 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?
- 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.
- Do you expect to be able to compress the image you just took a second time with your Huffman program?1 Why or why not?
- Are there multiple ways to achieve the same smallest file size? Explain why this might be.
- When traversing the code tree, is it possible for a internal node to have a symbol?
- Why do we bother creating a histogram instead of randomly assigning a tree.
- 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
- Using the example binary, calculate the compression ratio of a large text of your choosing 2

1. The goal of compression is to reduce the data size of a file without losing any significant info from that file. It is easy to compress a string that is redundant with the same letter because it can be stored as the letter and how many times it is repeated. For example, for the given string it would be stored as "a16."

2. With loosy compression you can not decompress the file that was compressed but with loseless compression you can compress and decompress the file. Huffman coding is loseless compression. JPEG is lossy compression. Yes, you can lossily compress a text file but you will end up losing text within the file since you will not be able to decompress it for the lost info.

3. The program can only accept text and binary files. Any other files will result in an error from the program. I'm guessing there could be special cases where using Huffman coding won't make the file smaller.

4. The patterns found by the Huffman Coding are generally very large and occur more frequently. It is well-suited for text files and source code.

5. The picture was taken in 4k resolution (12MP). The photo takes up 2.9 MB which is 2,900,000 bytes.

6. I would have expected the picture to take up 36,578,304 bytes. I guess that the photo is compressed losslessly but without losing actual quality in the picture.

7. The compression ratio is 0.079.

8. I do not expect to be able to compress the image a second time using the Huffman program because the allowed file inputs of the program are .txt files or binary files and on top of that Huffman program is made to run to check for repeating patterns in the file so it can be reduced.

9. Yes there can be multiple ways to achieve the smallest file size.

10. No it is not possible for the internal node to have a symbol in it because they are located in the leaf nodes of the tree and the internal nodes have the combination of characters/symbols.

11. You want to create a histogram before the tree because it helps you find out the frequencies of each symbol in the data making everything much easier.

12. Huffman coding compresses data by assigning shorter codes to more common symbols. Morse code, on the other hand, represents characters with a fixed-length sequence of dots and dashes, making it less efficient for compressing text files.

13.

Testing

- Testing for this program should be fairly easy due to most test files being provided to check our functions files. Not only that we are provided the -x86 of both the programs so I can create test functions to make sure the output from my program matches the outputs of the given -x86 files.

How to Use the Program

- To run the program it is very straightforward. You first run the Makefile, the compiler, by running the following command `make`. Then you run the program with the following command `./<program-name> <command><inputfile> <command><outputfile>`.

Program Design

Main Data Structures:

- Tree Node: Represents a node in the Huffman Tree. It contains fields for the symbol, weight, code, code length, left child, and right child.

- Priority Queue.

Pseudocode

BitWriter.c

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

BitWriter *bit_write_open(const char *filename);
2  allocate a new BitWriter
3  open the filename for writing as a binary file, storing the result in FILE *f
4  store f in the BitWriter field underlying_stream
5  clear the byte and bit_positions fields of the BitWriter to 0
6  if any step above causes an error:
7      return NULL
8  else:
9      return a pointer to the new BitWriter


void bit_write_close(BitWriter **pbuf);
2  if *pbuf != NULL:
3      if (*pbuf)->bit_position > 0:
4          /* (*pbuf)->byte contains at least one bit that has not yet been written */
5          write the byte to the underlying_stream using fputc()
6      close the underlying_stream
7      free the BitWriter
8      *pbuf = NULL


void bit_write_bit(BitWriter *buf, uint8_t bit);
2  if bit_position > 7:
3      write the byte to the underlying_stream using fputc()
4      clear the byte and bit_position fields of the BitWriter to 0
5  set the bit at bit_position of the byte to the value of bit
6  bit_position += 1


void bit_write_uint16(BitWriter *buf, uint16_t x);
2  for i = 0 to 15:
3      write bit i of x using bit_write_bit()

void bit_write_uint32(BitWriter *buf, uint32_t x);
2  for i = 0 to 31:
3      write bit i of x using bit_write_bit()


void bit_write_uint8(BitWriter *buf, uint8_t byte);
2  for i = 0 to 7:
3      write bit i of x using bit_write_bit()
```

BitReader.c

```
1 /* bitreader.c */
2  #include "bitreader.h"
3 struct BitReader {
4  FILE *underlying_stream;
5  uint8_t byte;
```

```
6   uint8_t bit_position;
7 };

BitReader *bit_read_open(const char *filename);
2   allocate a new BitReader
3   open the filename for reading as a binary file, storing the result in FILE *f
4   store f in the BitReader field underlying_stream
5   clear the byte field of the BitReader to 0
6   set the bit_position field of the BitReader to 8
7   if any step above causes an error:
8       return NULL
9   else:
10      return a pointer to the new BitReader

void bit_read_close(BitReader **pbuf);
2   if *pbuf != NULL:
3       close the underlying_stream
4       free *pbuf
5       *pbuf = NULL
6       if any step above causes an error:
7           report fatal error


uint32_t bit_read_uint32(BitReader *buf);
2   uint32_t word = 0x00000000
3   for i in range(0, 32):
4       read a bit b from the underlying_stream
5       set bit i of word to the value of b
6   return word;


uint16_t bit_read_uint16(BitReader *buf);
2   uint16_t word = 0x0000
3   for i in range(0, 16):
4       read a bit b from the underlying_stream
5       set bit i of word to the value of b
6   return word;

uint8_t bit_read_uint8(BitReader *buf);
2   uint8_t byte = 0x00
3   for i in range(0, 8):
4       read a bit b from the underlying_stream
5       set bit i of byte to the value of b
6   return byte

uint8_t bit_read_bit(BitReader *buf);
2   if bit_position > 7:
3       read a byte from the underlying_stream using fgetc()
4       bit_position = 0
5   get the bit numbered bit_position from byte
6   bit_position += 1;
7   if any step above causes an error:
8       report fatal error
9   else:
10      return the bit
```

Node.c

```
Node *node_create(uint8_t symbol, uint32_t weight);
2   allocate a new Node
```

```
3   set the symbol and weight fields of Node to function parameters symbol and weight
4   if any step above causes an error:
5       return NULL
6   else:
7       return a pointer to the new Node

void node_free(Node **node);
2   if *pnode != NULL:
3       node_free(&(*pnode)->left)
4       node_free(&(*pnode)->right)
5       free(*pnode)
6       *pnode = NULL

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

void node_print_tree(Node *tree);
2   if (tree == NULL)
3       return;
4   node_print_node(tree->right, '/', indentation + 3);
5   printf("%*cweight = %d", indentation + 1, ch, tree->weight);
6   if (tree->left == NULL && tree->right == NULL) {
7       if (' ' <= tree->symbol && tree->symbol <= '~') {
8           printf(", symbol = '%c'", tree->symbol);
9       } else {
10          printf(", symbol = 0x%02x", tree->symbol);
11      }
12  }
13  printf("\n");
14  node_print_node(tree->left, '\\', indentation + 3);
```

PriorityQueue.c

```
1 /* pq.c */
2 typedef struct ListElement ListElement;
3 struct ListElement {
4   Node *tree;
5   ListElement *next;
6 };
7 struct PriorityQueue {
8   ListElement *list;
9 };

PriorityQueue *pq_create(void);
    PriorityQueue *pq = calloc(1, sizeof(PriorityQueue));

    if (pq == NULL) {
        exit(1);
    }

    return pq;

void pq_free(PriorityQueue **q);
    if (*q != NULL) {
        free(*q);
        *q = NULL;
    }

bool pq_is_empty(PriorityQueue *q);
    if (q->list == NULL) {
```

```
        return true;
    } else {
        return false;
    }

bool pq_size_is_1(PriorityQueue *q);
    if (q->list == NULL) {
        return false;
    } else if (q->list->next == NULL) {
        return true;
    } else {
        return false;
    }


bool pq_less_than(ListElement *e1, ListElement *e2)
    if (e1->tree->weight < e2->tree->weight) {
        return true;
    } else if (e1->tree->weight == e2->tree->weight) {

        if (e1->tree->symbol < e2->tree->symbol) {
            return true;
        } else {
            return false;
        }

    } else {
        return false;

void enqueue(PriorityQueue *q, Node *tree);
    //Insert a tree into the priority queue. Keep the tree with the lowest weight at the head (that
          is, next to bedequeued). There are three possibilities to consider:
    //    The queue currently is empty.
    //    The new element will become the new first element of the queue.
    //    The new element will be placed after an existing element.

Node *dequeue(PriorityQueue *q);
    //Remove the queue element with the lowest weight and return it. If the queue is empty, then
         report a fatal error.
    //CHECKING FOR ERROR FIRST
    if (pq_is_empty(q) == true) {
        exit(1);
    } else {
        ListElement *remove = q->list;
        Node *r_node = remove->tree;
        q->list = q->list->next;
        free(remove);
        return r_node;
    }

void pq_print(PriorityQueue *q);
2   assert(q != NULL);
3   ListElement *e = q->list;
4   int position = 1;
5   while (e != NULL) {
6       if (position++ == 1) {
7           printf("============================================\n");
8       } else {
9           printf("--------------------------------------------\n");
```

```
10      }
11      node_print_tree(e->tree, '<', 2);
12      e = e->next;
13  }
14  printf("============================================\n");
```

Huff.c

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

#define HELP
Function fill_histogram(file: FILE, histogram: uint32_t[]) -> uint32_t:
    if histogram is NULL:
        exit(0)

    file_count := 0

    for i from 0 to 255:
        histogram[i] := 0

    histogram[0x00]++
    histogram[0xFF]++

    while (read_value := fgetc(file)) is not EOF:
        histogram[read_value]++
        file_count++

    return file_count

Node *create_tree(uint32_t *histogram, uint16_t *num_leaves) {
        int i = 0;
        while (i < 256) {
            if (histogram[i] != 0) {

                Node *new = node_create((uint8_t) i, histogram[i]);

                if (new == NULL) {
                    pq_free(&queue);
                }

                enqueue(queue, new);
                (*num_leaves)++;
            }

            i++;
        }

    while Priority Queue has more than one entry
        Dequeue into left
        Dequeue into right
        Create a new node with symbol = 0 and weight = left->weight + right->weight
        node->left = left
        node->right = right
        Enqueue the new node

    //Dequeue the queues only entry and return it.
    Node *tree = dequeue(queue);
```

```
    pq_free(&queue);

fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)
    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;

void 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

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

FUNCTION main(argc, argv):
    input = NULL
    output = NULL
    options = "i:o:h"
    WHILE (user = GET_OPTION(argc, argv, options)) != -1 DO
        SWITCH user:
            CASE 'i':
                input = FOPEN(optarg, "rb")
                IF input == NULL THEN
                    PRINT("huff: error with input file %s\n", optarg)
                    PRINT(HELP)
                    EXIT(1)
                ELSE
                    BREAK
            CASE 'o':
```

```
            output = BIT_WRITE_OPEN(optarg)
            IF output == NULL THEN
                PRINT("huff: poorly formatted -o\n")
                EXIT(1)
        CASE 'h':
            PRINT(HELP)
            EXIT(0)

    IF input == NULL THEN
        PRINT("huff: -i(input) is required\n")
        PRINT(HELP)
        EXIT(1)
    ELSE IF output == NULL THEN
        PRINT("huff: -o(output) is required\n")
        PRINT(HELP)
        EXIT(1)

    petals = 0
    hist = CALLOC(256, sizeof(uint32_t))
    file = fill_histogram(input, hist)
    tree = create_tree(hist, &petals)
    code_table = CALLOC(256, sizeof(Code))
    fill_code_table(code_table, tree, 0, 0)
    FSEEK(input, 0, SEEK_SET)
    huff_compress_file(output, input, file, petals, tree, code_table)
    FREE(hist)
    FREE(code_table)
    NODE_FREE(tree)
    BIT_WRITE_CLOSE(output)
    FCLOSE(input)
    RETURN 0
```

Dehuff.c

```
#include "bitreader.h"
#include "bitwriter.h"
#include "node.h"
#include "pq.h"

#include <assert.h>
#include <inttypes.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define SIZE 64
struct Node *stack[SIZE];

#define HELP
//global var
int top = -1;

//push
void stackpush(struct Node *node) {
    if (top >= SIZE - 1) {
        fprintf(stderr, "The stack's full.\n");
        exit(EXIT_FAILURE);
```

```
    }

    stack[++top] = node;
}

//pop
struct Node *stackpop(void) {
    if (top < 0) {
        fprintf(stderr, "The stack's empty.\n");
        exit(EXIT_FAILURE);
    }

    return stack[top--];
}

void dehuff_decompress_file(FILE *fout, BitReader *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

FUNCTION main(argc, argv):
    output = NULL
    input = NULL
    options = "i:o:h"
    WHILE (user = GET_OPTION(argc, argv, options)) != -1 DO
        SWITCH user:
            CASE 'i':
                input = BIT_READ_OPEN(optarg)
                IF input == NULL THEN
                    PRINT("dehuff: error with input file %s\n", optarg)
                    PRINT(HELP)
                    EXIT(1)
```

```
            ELSE
                BREAK
        CASE 'o':
            output = FOPEN(optarg, "wb")
            IF output == NULL THEN
                PRINT("dehuff: poorly formatted -o\n")
                EXIT(1)
        CASE 'h':
            PRINT(HELP)
            EXIT(0)

    IF input == NULL THEN
        PRINT("dehuff: -i(input) is required\n")
        PRINT(HELP)
        EXIT(1)
    ELSE IF output == NULL THEN
        PRINT("dehuff: -o(output) is required\n")
        PRINT(HELP)
        EXIT(1)

    dehuff_decompress_file(output, input)

    FCLOSE(output)
    BIT_READ_CLOSE(input)

    RETURN 0
```

Function Descriptions

BitWriter.c

```
BitWriter *bit_write_open(const char *filename);
    -Open binary or text file, filename for write using fopen() and return a pointer to a newly
        allocated BitWriter struct. You must check all function return values and return NULL if
        any of them report a failure.

void bit_write_close(BitWriter **pbuf);
    - Using values in the BitWriter pointed to by *pbuf, flush any data in the byte buffer, close
        underlying_stream, free the BitWriter object, and set the *pbuf pointer to NULL

void bit_write_bit(BitWriter *buf, uint8_t bit);
    -  writes a single bit, bit, using values in the BitWriter pointed to by buf. This function
        collects 8 bits into the buffer byte before writing it using fputc(). You must check all
        function return values and report a fatal error if any of them report a failure.

void bit_write_uint16(BitWriter *buf, uint16_t x);
    - Write the 16 bits of function parameter x by calling bit_write_bit() 16 times.

void bit_write_uint32(BitWriter *buf, uint32_t x);
    - Write the 32 bits of function parameter x by calling bit_write_bit() 32 times.

void bit_write_uint8(BitWriter *buf, uint8_t byte);
```

BitReader.c

```
BitReader *bit_read_open(const char *filename);
    - Open binary filename using fopen() and return a pointer to a BitReader. On error, return NULL

void bit_read_close(BitReader **pbuf);
```

```
    - Using values in the BitReader pointed to by *pbuf, close (*pbuf)->underlying_stream, free the
        BitReader object, and set the *pbuf pointer to NULL.

uint32_t bit_read_uint32(BitReader *buf);
    - Read 32 bits from buf by calling bit_read_bit() 32 times

uint16_t bit_read_uint16(BitReader *buf);
    - Read 16 bits from buf by calling bit_read_bit() 16 times.

uint8_t bit_read_uint8(BitReader *buf);
    - Read 8 bits from buf by calling bit_read_bit() 8 times

uint8_t bit_read_bit(BitReader *buf);
    - It reads a single bit using values in the BitReader pointed to by buf.
```

Node.c

```
Node *node_create(uint8_t symbol, uint32_t weight);
    - Create a Node and set its symbol and weight fields. Return a pointer to the new node. On
        error, return NULL.

void node_free(Node **node);
    - Free the children of *pnode, free *pnode, and set *pnode to NULL

void node_print_tree(Node *tree);
    - provided print a tree however u want.
```

PriorityQueue.c

```
PriorityQueue *pq_create(void);
    - Allocate a PriorityQueue object and return a pointer to it. If theres an error, return NULL.

void pq_free(PriorityQueue **q);
    - Call free() on *q, and then set *q = NULL

bool pq_is_empty(PriorityQueue *q);
    - We indicate an empty queue by storing NULL in the queues list field. Return true if thats the
        case

bool pq_size_is_1(PriorityQueue *q);
    - If the Priority Queue contains a single element, then return true. Otherwise return false.
bool pq_less_than(ListElement *e1, ListElement *e2)
    - The pq_less_than() function compares the tree->weight values of two ListElement objects,
        returning true if the weight of the first element is less than the weight of the second
        element. If the weights of the elements are equal, then compare their tree->symbol values,
        and return true if the symbol of the first element is less than the symbol of the second
        element.
    This function is not used outside of pq.c, and so it is not declared in pq.h.

void enqueue(PriorityQueue *q, Node *tree);
    - Insert a tree into the priority queue.
Node *dequeue(PriorityQueue *q);
    - Remove the queue element with the lowest weight and return it. If the queue is empty, then
        report a fatal error.

void pq_print(PriorityQueue *q);
    -  It prints the trees of the queue q.
```

References