Assignment 7 – Huffman Coding Report

Jackson Dawson

CSE 13S – Winter 24

Purpose

The "huff" program takes in a file, compresses the data using Huffman Coding, and writes the compressed data to a file. It requires command line options to specify where you want to provide input from, and where you want the output to go. This program utilizes a priority queue data structure as well as bitwriter, bitreader and huffman coding algorithms in order to effectively and efficiently compress data files. Dehuff uses the bitreader and huffman tree structure to parse the compressed files and restore them. It also utilizes a stack data structure.

Questions

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

  The goal of compression is to reduce the amount of data required to represent information. In the context of huffman coding, compressing the string "aaaaaaaaa" would mean creating a huffman tree consisting of one node which represents "a". This could be done using 1 bit (theoretically 0.5 bits), and in practice means that the string could be represented by the only number of bits as however many letter "a"'s there are in the string.

- 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?

  Lossless compression is compression which preserves the ability to entirely restore the file back to its original form. Lossy compression prioritizes a reduction in size for this ability. However, it is possible to compress lossily and retain a great amount of information upon decompression. Huffman coding is a form of lossless compression, whereas JPEG is a lossy form of compression. It seems that it is not possible, or perhaps preferable to compress text files lossily.

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

  Yes, "huff" can accept any file as input. However, it will not always make the file smaller, and it some cases will increase the size. This is largely due to the fact that huffman coding excells at compressing redundant data, and thus will struggle to reduce the size of a highly unique file.

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

  The patterns found by huffman coding are based on character frequency. The system organizes codes based on how frequency each character is in the given file. This means that huffman coding is most effective for files which contain a biased distribution of some characters over others.

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

I took a photo of my cat. Aww. The photo is 3024px x 4032px, and takes up 2.2 megabytes, or about 2,306,867 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?

  It would seem that the photo should take up 36,578,304 bytes. I bet the photo is compressed! I also bet it was lossily compressed.

- 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.

  The compression ratio I calculated was about 0.0631.

Testing

I will test "huff" and "dehuff" using bash scripts to compare my implementation's output with the output of the included binaries. I will use each of the provided files to ensure thorough testing. I will ensure that a test is run with the -h option selected. Unit testing will be administered for the bitreader, bitwriter, binary tree and priority queue implementations as well. I will create a C file that includes several key functions from each ADT, create objects (such as a queue and/or binary tree), manipulate their data, and then attempt to read, write, traverse and free accordingly. These tests will be compiled as part of "make tests" in the Makefile, and will be checked for memory leaks using Valgrind.

How to Use the Program

Start by compiling the program to make it usable. This is achieved by typing the following commands into the terminal:

```
make clean
make format
make
```

Now the programs "huff" and "dehuff" are ready to use. You may run "huff" and/or "dehuff" with any of the following options:

- **-i:** Sets the file to read from (input file). Requires a filename as an argument. The default file to read from is "stdin".[1]
- **-o:** Sets the file to write to (output file). Requires a filename as an argument. The default file to write to is "stdout".[1]
- **-h:** Prints a help message to "stdout".[1]

An example execution may look like the following, where "huff" will read from "test_files/independence.txt", and write to "test.txt".

```
./huff -i test_files/independence.txt -o test.txt
```

"huff" will read the file provided in the -i argument, proceed to attempt to compress the file, and write the compressed file to the file dictated by the -o argument.

To clean up after yourself, do:

```
make clean
```

You're all done!

Program Design

Main Data Structures:

- Priority Queue
  - A specialized queue that orders its elements based on their priority. In the context of Huffman coding, it is used to ensure that nodes with lower weights (higher priorities) are combined first when building the Huffman tree.
- Binary Tree
  - A hierarchical data structure ideal for storing data in a sorted manner. Each node has up to two children, referred to as the left and right child. In the context of Huffman coding, it is used to represent the Huffman tree where each leaf node represents a symbol and its weight, and internal nodes represent the sum of the weights of their child nodes.
- Code Table
  - A mapping from characters (or symbols) to their corresponding Huffman codes. Each entry in the table represents a character and its code, with the code stored as a bit sequence and its length.
- Histogram
  - A data structure used to count the frequency of each symbol in the input file. This frequency count is used to determine the weight of each symbol in the Huffman tree.
- Node
  - Represents an element in the Huffman tree. It contains the symbol (character), weight (frequency of the symbol), and pointers to the left and right child nodes.
- Stack
  - A linear data structure that follows the Last In, First Out (LIFO) principle. Used in the decompression process to rebuild the Huffman tree from the serialized form.

Main Algorithms:

- Bit Writer
  - The bit writer file contains a number of important functions which allow for writing to a file bit by bit, rather than the more common byte by byte methods.
- Bit Reader
  - The bit reader allows for reading from a file bit by bit.
- Huffman Coding
  - The Huff program utilizes a histogram, followed by a binary tree, followed by a code table, and then finally uses all available information to organize the codes and compressed data into a file.
- Huffman Decoding
  - The Dehuff program reads through the compressed file, reconstructs a binary tree to represent the huffman codes using a stack data structure, and then traverses the tree and writes the respective symbols to the output file.

Pseudocode

The "huff" program follows a structured flow:

1. Read and interpret command line arguments (i.e. get input file and output file names)
2. Open the Input File
3. Fill the histogram to get the distribution of symbols
4. Create the huffman tree representation of the file based on the distribution as collected in the histogram
5. Create a code table to represent each symbol for easy and efficient access
6. Set up the bitwriter for writing the compressed data
7. Write the compressed file
    - Include Header
    - Include huffman tree guide
    - Loop through original file and insert according code for each symbol
8. Close the bitwriter and Input file, and free allocated memory

Function Descriptions

- **BitWriter *bit_write_open(const char *filename);**
    - **Inputs:** const char *filename - The name of the file to be opened for writing.
    - **Outputs:** BitWriter* - A pointer to a BitWriter structure for bit-level writing operations.
    - **Purpose:** To open a file for bit-level writing operations.
- **void bit_write_close(BitWriter **pbuf);**
    - **Inputs:** BitWriter **pbuf - A pointer to a pointer to a BitWriter structure.
    - **Outputs:** None.
    - **Purpose:** To close the bit writer and release any resources it was using.
- **void bit_write_bit(BitWriter *buf, uint8_t bit);**
    - **Inputs:** BitWriter *buf - A pointer to a BitWriter structure; uint8_t bit - The bit to write (0 or 1).
    - **Outputs:** None.
    - **Purpose:** To write a single bit to the buffer.
- **void bit_write_uint8(BitWriter *buf, uint8_t x);**
    - **Inputs:** BitWriter *buf - A pointer to a BitWriter structure; uint8_t x - The 8-bit unsigned integer to write.
    - **Outputs:** None.
    - **Purpose:** To write an 8-bit unsigned integer to the buffer bit by bit.
- **void bit_write_uint16(BitWriter *buf, uint16_t x);**
    - **Inputs:** BitWriter *buf - A pointer to a BitWriter structure; uint16_t x - The 16-bit unsigned integer to write.
    - **Outputs:** None.

  - **Purpose:** To write a 16-bit unsigned integer to the buffer bit by bit.

- **void bit_write_uint32(BitWriter *buf, uint32_t x);**
  - **Inputs:** BitWriter *buf - A pointer to a BitWriter structure; uint32_t x - The 32-bit unsigned integer to write.
  - **Outputs:** None.
  - **Purpose:** To write a 32-bit unsigned integer to the buffer bit by bit.
- **BitReader *bit_read_open(const char *filename);**
  - **Inputs:** const char *filename - The name of the file to be opened for reading.
  - **Outputs:** BitReader* - A pointer to a BitReader structure for bit-level reading operations.
  - **Purpose:** To open a file for bit-level reading operations.
- **void bit_read_close(BitReader **pbuf);**
  - **Inputs:** BitReader **pbuf - A pointer to a pointer to a BitReader structure.
  - **Outputs:** None.
  - **Purpose:** To close the bit reader and release any resources it was using.
- **uint8_t bit_read_bit(BitReader *buf);**
  - **Inputs:** BitReader *buf - A pointer to a BitReader structure.
  - **Outputs:** uint8_t - The next bit from the buffer (0 or 1).
  - **Purpose:** To read a single bit from the buffer.
- **uint8_t bit_read_uint8(BitReader *buf);**;
  - **Inputs:** A pointer to a BitReader structure.
  - **Outputs:** A uint8_t value representing an 8-bit integer read from the BitReader buffer.
  - **Purpose:** To read and return the next 8 bits (1 byte) from the buffer as a uint8_t.
- **uint16_t bit_read_uint16(BitReader *buf);**;
  - **Inputs:** A pointer to a BitReader structure.
  - **Outputs:** A uint16_t value representing a 16-bit integer read from the BitReader buffer.
  - **Purpose:** To read and return the next 16 bits (2 bytes) from the buffer as a uint16_t.
- **uint32_t bit_read_uint32(BitReader *buf);**;
  - **Inputs:** A pointer to a BitReader structure.
  - **Outputs:** A uint32_t value representing a 32-bit integer read from the BitReader buffer.
  - **Purpose:** To read and return the next 32 bits (4 bytes) from the buffer as a uint32_t.
- **Node *node_create(uint8_t symbol, uint32_t weight);**;
  - **Inputs:** A uint8_t value for the symbol and a uint32_t value for the weight.
  - **Outputs:** A pointer to a new Node structure with the specified symbol and weight.
  - **Purpose:** To create and return a new Node structure with the given symbol and weight for Huffman coding.
- **void node_free(Node **pnode);**;
  - **Inputs:** A double pointer to a Node structure.

  - **Outputs:** No return value, but frees the memory allocated for the Node and its children.
  - **Purpose:** To recursively free a Node and all its children from memory.

- **void node_print_tree(Node *tree);**;
  - **Inputs:** A pointer to the root Node of a tree.
  - **Outputs:** No return value, but prints the structure of the tree to stdout.
  - **Purpose:** To visually represent the structure and data of a Huffman tree.

- **PriorityQueue *pq_create(void);**;
  - **Inputs:** No input arguments.
  - **Outputs:** A pointer to a newly created PriorityQueue.
  - **Purpose:** To create and return a new, empty PriorityQueue structure.

- **void pq_free(PriorityQueue **q);**;
  - **Inputs:** A double pointer to a PriorityQueue.
  - **Outputs:** No return value, but frees the memory allocated for the PriorityQueue and its elements.
  - **Purpose:** To free a PriorityQueue and all its elements from memory.

- **bool pq_is_empty(PriorityQueue *q);**;
  - **Inputs:** A pointer to a PriorityQueue.
  - **Outputs:** A boolean value indicating whether the PriorityQueue is empty.
  - **Purpose:** To check if the PriorityQueue is empty.

- **bool pq_size_is_1(PriorityQueue *q);**;
  - **Inputs:** INPUT INFO
  - **Outputs:** OUTPUT INFO
  - **Purpose:** PURPOSE

- **bool pq_less_than(ListElement *e1, ListElement *e2);**;
  - **Inputs:** Two pointers to ListElement structs (`e1` and `e2`) representing the elements to compare.
  - **Outputs:** Returns `true` if the weight of the tree in `e1` is less than the weight of the tree in `e2`, or if their weights are equal and the symbol of the tree in `e1` is less than that of `e2`. Otherwise, returns `false`.
  - **Purpose:** To determine the order of two elements in the priority queue based on the weight (and symbol, if weights are equal) of their corresponding trees. This function helps to maintain the queue's ordering.

- **void enqueue(PriorityQueue *q, Node *tree);**;
  - **Inputs:** A pointer to a PriorityQueue struct (`q`) and a pointer to a Node struct (`tree`) representing the tree to be added to the queue.
  - **Outputs:** No direct output, but the function inserts the given tree into the queue in a position that maintains the queue's order.
  - **Purpose:** To insert a new tree node into the priority queue while maintaining the ordering of the queue based on the weight (and optionally symbol) of the nodes.

- **Node *dequeue(PriorityQueue *q);**;
  - **Inputs:** A pointer to a PriorityQueue struct (`q`).

  - **Outputs:** Returns a pointer to a Node struct representing the tree with the smallest weight (and symbol, if necessary) that was removed from the front of the queue.
  - **Purpose:** To remove and return the tree node with the smallest weight (and symbol, if necessary) from the priority queue, typically for processing or combining in Huffman coding.

- **void pq_print(PriorityQueue *q);**;
  - **Inputs:** A pointer to a PriorityQueue struct (`q`).
  - **Outputs:** No direct output, but prints the entire priority queue to standard output for debugging or inspection.
  - **Purpose:** To provide a visual representation of the current state of the priority queue, including the weight and symbol of each node in the queue.

- **uint32_t fill_histogram(FILE *fin, uint32_t *histogram);**;
  - **Inputs:** A pointer to a FILE struct (`fin`) representing the input file and a pointer to an array of uint32_t (`histogram`) to store the frequency of each byte.
  - **Outputs:** Returns the total number of bytes read from the file. The `histogram` array is populated with the frequency of each byte value (0-255).
  - **Purpose:** To read through an entire file, count the frequency of each byte, and populate a histogram array with these frequencies, which is crucial for Huffman coding.

- **Node *create_tree(uint32_t *histogram, uint16_t *num_leaves);**;
  - **Inputs:** A pointer to an array of uint32_t (`histogram`) containing the frequency of each byte and a pointer to a uint16_t (`num_leaves`) to store the number of leaf nodes created.
  - **Outputs:** Returns a pointer to a Node struct representing the root of the Huffman tree constructed from the frequencies in the histogram.
  - **Purpose:** To construct a Huffman tree based on the frequencies of bytes in the input, facilitating efficient compression by creating a coding scheme that uses shorter codes for more frequent bytes.

- **fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length);**;
  - **Inputs:** A pointer to an array of Code structs (`code_table`) for storing the Huffman codes, a pointer to a Node struct (`node`) representing the current node in the Huffman tree, a uint64_t (`code`) representing the current code being constructed, and a uint8_t (`code_length`) indicating the length of the current code.
  - **Outputs:** No direct output, but recursively fills the `code_table` with Huffman codes for each byte, represented by the Huffman tree.
  - **Purpose:** To traverse the Huffman tree and assign a unique binary code to each byte (leaf node) based on its path from the root, which is essential for compressing the data.

- **void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table);**;
  - **Inputs:** Parameters include a BitWriter pointer (`outbuf`) for the output file, a FILE pointer (`fin`) for the input file, a uint32_t (`filesize`) representing the size of the input file, a uint16_t (`num_leaves`) indicating the number of leaves in the Huffman tree, a Node pointer (`code_tree`) to the root of the Huffman tree, and an array of Code structs (`code_table`) containing the Huffman codes for each byte.
  - **Outputs:** No direct output, but writes the compressed data to the file associated with `outbuf`.
  - **Purpose:** To compress the input file using the Huffman coding scheme represented by `code_tree` and `code_table`, writing the compressed data to the output file.

- **void dehuff_decompress_file(FILE *fout, BitReader *inbuf);**;

- **Inputs:** A FILE pointer (`fout`) for the output file and a BitReader pointer (`inbuf`) for reading the compressed input.
- **Outputs:** No direct output, but decompresses the data read from `inbuf` and writes the decompressed data to `fout`.
- **Purpose:** To decompress data that was compressed using Huffman coding, reconstructing the original data from its compressed form.

References

[1] Kerry Veenstra Jess Srinivas. Huffman coding. CSE 13S Winter 2024 Course Assignment, University of California, Santa Cruz, 2024.