Assignment 8 – Huffman Coding

Aiden Trager

CSE 13S – Fall 2023

Purpose

This Huffman Coding program aims to provide an efficient solution for data compression. By implementing the Huffman Coding algorithm, the program compresses data by assigning shorter codes to more frequently occurring symbols, reducing overall file size. In an industry setting, this tool serves as a valuable asset for optimizing storage and transmission of data.

How to Use the Program

Start by:

```
make all
```

Followed by:

```
./huff *more will be needed*
```

The usage can be described by the -h flag:
Usage:
huff -i infile -o outfile
huff -v -i infile -o outfile
huff -h
And when it needs to be decoded:

```
./dehuff *more will be needed*
```

The usage can be described by the -h flag:
Usage:
dehuff -i infile -o outfile
dehuff -v -i infile -o outfile
dehuff -h

Program Design

Data Structures The program utilizes the following key data structures:

- Arrays: Used for storing and manipulating data efficiently.
- Structs: Employed to represent nodes in the Huffman tree, containing symbol and frequency information.

The choice of these data structures prioritizes simplicity and efficiency in managing symbol frequencies and constructing the Huffman tree.

Algorithms The core algorithms for Huffman Coding are outlined below:
Huffman Compression Algorithm:
1. Build a frequency histogram of symbols in the input file.
2. Create a priority queue of nodes based on symbol frequencies.
3. Build the Huffman tree by repeatedly merging nodes with the lowest frequencies.
4. Generate a code table by traversing the Huffman tree.
5. Compress the input file using the generated codes.
Huffman Decompression Algorithm:
1. Read the code table from the compressed file.
2. Reconstruct the Huffman tree.
3. Decode the compressed file using the Huffman tree.
4. Output the decompressed data.
Pseudocode for both of these is given in the assignment pdf, so I will neglect to show it here.

Function Descriptions

0.0.1 BitWriter Functions:

BitWriter *bit_write_open(const char *filename);

- Opens a binary file specified by `filename` for writing using `fopen()` and returns a pointer to a newly created `BitWriter` structure.
- Initializes the underlying stream and the internal byte buffer.
- Returns NULL on failure, and it is essential to check all function return values.

void bit_write_close(BitWriter **pbuf);

- Flushes any remaining data in the byte buffer, closes the underlying stream, frees the `BitWriter` object, and sets the pointer to NULL.
- It is crucial to check all function return values and report fatal errors if any occur.

void bit_write_bit(BitWriter *buf, uint8_t bit);

- Writes a single bit (`bit`) to the binary file using values in the `BitWriter` structure.
- Collects 8 bits into the buffer before writing it using `fputc()`.
- Checks all function return values and reports fatal errors if any occur.

void bit_write_uint8(BitWriter *buf, uint8_t x);

- Writes the 8 bits of the function parameter `x` by calling `bit_write_bit()` 8 times.
- Ensures correct bit alignment within the binary file.

void bit_write_uint16(BitWriter *buf, uint16_t x);

- Writes the 16 bits of the function parameter `x` by calling `bit_write_bit()` 16 times.
- Ensures correct bit alignment within the binary file.

void bit_write_uint32(BitWriter *buf, uint32_t x);

- Writes the 32 bits of the function parameter `x` by calling `bit_write_bit()` 32 times.
- Ensures correct bit alignment within the binary file.

0.0.2 BitReader Functions:

BitReader *bit_read_open(const char *filename);

- Opens a binary file specified by `filename` for reading using `fopen()` and returns a pointer to a newly created `BitReader` structure.
- Initializes the underlying stream and the internal byte buffer.
- Returns NULL on failure, and it is essential to check all function return values.

void bit_read_close(BitReader **pbuf);

- Closes the underlying stream and frees the `BitReader` object, setting the pointer to NULL.
- Checks all function return values and reports fatal errors if any occur.

uint8_t bit_read_bit(BitReader *buf);

- Reads a single bit from the binary file using values in the `BitReader` structure.
- Manages the byte buffer and ensures correct bit extraction.

uint8_t bit_read_uint8(BitReader *buf);

- Reads 8 bits from the binary file by calling `bit_read_bit()` 8 times.
- Collects these bits into a `uint8_t` starting with the least significant bit.

uint16_t bit_read_uint16(BitReader *buf);

- Reads 16 bits from the binary file by calling `bit_read_bit()` 16 times.
- Collects these bits into a `uint16_t` starting with the least significant bit.

uint32_t bit_read_uint32(BitReader *buf);

- Reads 32 bits from the binary file by calling `bit_read_bit()` 32 times.
- Collects these bits into a `uint32_t` starting with the least significant bit.

0.0.3 Node Functions:

Node *node_create(uint8_t symbol, uint32_t weight);

- Creates a new `Node` and sets its `symbol` and `weight` fields.
- Returns a pointer to the new `Node` on success and NULL on failure.

void node_free(Node **pnode); Frees the memory occupied by the `Node` pointed to by `*pnode` and sets it to NULL.

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

- Diagnostics and debugging function for printing the tree structure.
- Prints a sideways view of the binary tree using text characters.

0.0.4 Priority Queue Functions:

PriorityQueue *pq_create(void);

- Allocates a `PriorityQueue` object and returns a pointer to it.
- Returns NULL on failure.

void pq_free(PriorityQueue **q); Frees the memory occupied by the `PriorityQueue` pointed to by `*q` and sets it to NULL.

bool pq_is_empty(PriorityQueue *q); Returns true if the priority queue is empty (list field is NULL), otherwise returns false.

bool pq_size_is_1(PriorityQueue *q); Returns true if the priority queue contains a single element, otherwise returns false.

void enqueue(PriorityQueue *q, Node *tree);

- Inserts a tree into the priority queue, keeping the tree with the lowest weight at the head.
- Handles various cases, including an empty queue or inserting before/after existing elements.

Node *dequeue(PriorityQueue *q);

- Removes the queue element with the lowest weight and returns it.
- Reports a fatal error if the queue is empty.

void pq_print(PriorityQueue *q); Diagnostic function for printing the trees of the queue.

0.0.5 Huffman Coding Functions:

uint32_t fill_histogram(FILE *fin, uint32_t *histogram);

- Updates a histogram array with the number of occurrences of each unique byte value in the input file.
- Returns the total size of the input file.

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

- Creates a Huffman tree from the histogram and returns a pointer to the root node.
- Updates `num_leaves` with the number of leaf nodes in the tree.

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

- Recursively fills a code table for each leaf node's symbol in the Huffman tree.
- The code table is an array of `Code` objects, each containing a code and code length.

void huff_compress_file(BitWriter *outbuf, FILE *fin, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table); Writes a Huffman-coded file using the provided BitWriter, input file, file size, Huffman tree, and code table.

0.0.6 Huffman Decoding Functions:

void dehuff_decompress_file(FILE *fout, BitReader *inbuf); Reads a Huffman-coded file using the provided BitReader and writes the decompressed output to the specified file.

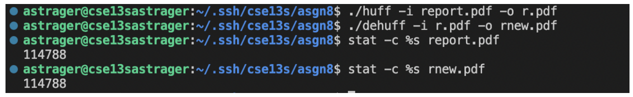

Figure 1: File is the same size before and after

Results

In evaluating the program's performance, it successfully compresses and decompress files, achieving the intended purpose of data optimization. The program has been tested on various inputs and with each one of them was able to compress and decompress them. This program is a lossless form of data compression. It is lossless because the file is the same as the original input after decompressing it. That is shown above.

References