Assignment 6 – Huffman Coding

Yazmyn Sims

CSE 13S – Spring 2023

Purpose

This program aims to produce a data compression using Huffman's Code.

How to Use the Program

The user will have to type command line options (-i: Sets the name of the input file, -o: Sets the name of the output file, -h: will remind the user how to use the command line options) into vim followed by a filename after i and o. For example user input could look like:

./huff -i files/zero.txt -o files/zero.huff

OUTPUT: The output will be a compressed file.

Program Design

Data Structures

- Struct: groups several related variables into one place. Like an array that can hold elements of different types.
    - Used in bitwriter.c, node.c, pq.c, and huff.c.
- Buffer: use a single, constant-size buffer as though they link end to end
    - Used in bitwriter.c, huff.c
- File: representations for data
    - Used in bitwriter.c and huff.c
- Priority Queue: a queue where each element has its own level of priority. High priority is served first.
    - Used in pq.c and huff.c.
- Tree: a collection of nodes connected by edges
    - Used in node.c, pq.c and huff.c.

Algorithms

- Huffman Coding Algorithm

Example Pseudocode for compressing a file:

```
huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
8 'H'
8 'C'
32 filesize
16 num_leaves
huff_write_tree(outbuf, code_tree)
for every byte b from inbuf
    code = code_table[b].code
    code_length = code_table[b].code_length
    for i = 0 to code_length - 1
        /* write the rightmost bit of code */
        1 code & 1
        /* prepare to write the next bit */
        code >>= 1
```

Function Descriptions

BitWriter Functions

- BitWriter *bit_write_open(const char *filename);
    - Open filename using write_open() and return a pointer to a BitWriter struct. On error, return NULL.
- void bit_write_close(BitWriter **pbuf);
    - Using values in the BitWriter struct 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 x);
    - Writes a single bit using values in the BitWriter struct pointed to by buf. Collects 8 bits into the buffer byte before writing the entire buffer.
- void bit_write_uint8(BitWriter *buf, uint8_t x);
    - Write 8 bits of the uint8_t x. Start with the LSB(least-significant-bit).
- void bit_write_uint16(BitWriter *buf, uint16_t x);
    - Write 16 bits of the uint16_t x. Start with the LSB.
- void bit_write_uint32(BitWriter *buf, uint32_t x);
    - Write 32 bits of the uint32_t x. Start with the LSB.

Node Functions

- Node *node_create(uint8_t symbol, double weight);
    - Create a Node and set its symbol and weight fields. Return a pointer to the new node.
- void node_free(Node **node);
    - Free the *node and set it to NULL.
- void node_print_tree(Node *tree, char ch, int indentation);
    - (optional) for debugging purposes

Priority Queue Functions

- PriorityQueue *pq_create(void);

    - Allocate a PriorityQueue object and return a pointer to it. (use calloc())
- void pq_free(PriorityQueue **q);
    - Call free() on *q, and then set *q = NULL.
- bool pq_is_empty(PriorityQueue *q);
    - Indicate an empty queue by storing NULL in the queue's list field. Return true.
- bool pq_size_is_1(PriorityQueue *q);
    - Huffman Coding algorithm fills the Priority Queue and then runs a loop until the queue contains a single value.
- void enqueue(PriorityQueue *q, Node *tree);
    - Insert a tree into the priority queue.
    - Keep the tree with the lowest weight at the head
    - Allocate a new ListElement *e, and set e->tree = tree.
    - The enqueuing operation performed depends on the queue's current state.
    - Empty the queue
    - Insert the new element at the beginning of the queue.
    - New element goes after one of the existing elements.
    - follow the linked list until you find the last queue entry or until you find a queue entry whose next field points to a tree whose weight is greater than the tree.
    - Then insert e after the queue element that you found.
- bool dequeue(PriorityQueue *q, Node **tree);
    - If the queue is empty, return false
    - else remove the queue element with the lowest weight, set e to point to it, set parameter *tree = e->tree, call free(e), and return true
- void pq_print(PriorityQueue *q);
    - Here's a diagnostic function. It prints the trees of the queue q.
- bool pq_less_than(Node *n1, Node *n2)
    - enqueue() compares the weights of two Node objects. The order of the nodes is determined by the weights.
    - include a tie-breaker in the comparison function, using the symbol value

Huffman Coding Functions

- uint64_t fill_histogram(Buffer *inbuf, double *histogram)
    - updates a histogram with the number of each of the unique byte values of the input file.
    - also returns the total size of the input file.
    - Parameter inbuf provides access to the input file using read_uint8().
    - Parameter histogram points to a 256-element array of doubles.
    - Clear all elements of this array, and then read bytes from inbuf using read_uint8().
    - For each byte read, increment the proper element of the histogram.
    - return the total size of the file
- Node *create_tree(double *histogram, uint16_t *num_leaves)
    - This function creates and returns a pointer to a new Huffman Tree.
    - returns the number of leaf nodes in the tree.
    - Create and fill a Priority Queue
    - Run the Huffman Coding algorithm.
    - Dequeue the queue's only entry and return it.
- fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)
    - traverses the tree and fills in the Code Table for each leaf node's symbol.

- huff_compress_file(outbuf, inbuf, filesize, num_leaves, code_tree, code_table)
    - Write a Huffman Coded file.
    - Parameters:
        - BitWriter *outbuf — Use this parameter with calls to bit_write_bit(), bit_write_uint8(), bit_write_uint16(), and bit_write_uint32() to write the output file.
        - Buffer *inbuf — Use with calls to read_uint8() to read the input file
            - buffer must be closed/reopened before this function can re-read the file.
        - uint32_t filesize — The size of the file, returned by the call to fill_histogram().
        - uint16_t num_leaves — The leave number in Code Tree, returned by create_tree()
        - Node *code_tree – A pointer to the Code Tree, returned by create_tree().
        - Code *code_table — A pointer to the Code Table, prepared by fill_code_table().
- huff_write_tree(outbuf, node)
    - A recursive routine that writes the code tree.

Results

I was unable to figure this assignment out. Sorry.

References

N/A

N/A

Figure 1: N/A