Assignment 7 Huffman Coding

Jasmine Lorenz

CSE 13S – Winter 24

Purpose

The purpose of the Huffman Coding program is to efficiently compress and decompress data. In this case, we're asked to help Summer free up storage so she can have more pictures of Tank the cat on her computer. However, she still wants to keep the existing files. Huffman Coding aims to minimize the overall size of the data while preserving its contents. This program enables us to compress large volumes of data into smaller representations, perfect for the task at hand.

Questions

1. Describe the goal of compression. (As a hint, why is it easy to compress the string "aaaaaaa"?)
    - The goal is to reduce the size of data. Compression aims to store or transmit data using fewer bits than the original, while preserving the essential information. It's simple to compress the string "aaaaaa" because each character is the same. This means there is a higher degree of redundancy, or repeated patterns.

2. 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 keeps all the original data intact when making files smaller. It's great for things like text where you can't afford to lose any details. Lossy compression decreases the amount of bits or data in a file to make it smaller, but it looses quality. Those can be used in JPEG files which can be pictures. Huffman Coding is a form of lossless compression and works great with text files. Lossy compression won't work on those files because you'll mess up the text.

3. Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case?
    - Yes it can accept any files, as it operates on streams of data, treating them as sequences of symbols, where each symbol could be a character, byte, or any type of data. However, Huffman coding may not always make a file smaller. It depends on the redundancy of the file. If there's a lot of repetition, Huffman coding can reduce the size, but if not, it might not make much of a difference or could even increase the size.

4. How big are the patterns found by Huffman Coding? What kind of files does this lend itself to?
    - The size of patterns found by Huffman Coding depends on how often different parts of the data show up. It's good for files with lots of repeating stuff, like text files. If some things in the file happen a lot more than others, Huffman Coding can shrink the file size by giving shorter codes to the common things and longer codes to the rare ones. So, it's handy for things like written documents.

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

    - My front-facing camera had a resolution of `2316 x 3088` with `899 KB` of storage. My back-camera had a resolution of `3024 x 4032` with `2 MB` of storage.

6. 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?

    - My back-facing camera image would be `36.58 MB`. It could be smaller because of compression technique done by the image file.

7. 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.

    - My compression ratio is `0.0546`.

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

    - Not really, since there's not much leftover redundancy for Huffman coding to use. Trying to compress it again might even make the file bigger because JPEG has already removed a lot of unnecessary bits.

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

    - Yes, there can be different ways to make a file as small as possible. Compression algorithms can rearrange and encode data in different ways while still achieving the smallest possible file size. For example, lossy compression for JPEG files can make trade-offs in how it discards information, resulting in difference compression outcomes.

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

    - No, in Huffman coding, internal nodes do not have symbols. Internal nodes in the Huffman tree represent combinations of symbols rather than individual symbols themselves. Combinations are formed during the construction of the Huffman tree as the algorithm merges nodes with the lowest frequencies to create parent nodes. The leaf nodes of the tree, or the endpoints, represent individual symbols in the input data, while the internal nodes represent combinations of symbols.

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

    - Creating a histogram helps us see which symbols appear more often in the data. With this information, we can assign shorter codes to the symbols that show up frequently, and longer codes to the ones that don't. This strategy makes the compressed file smaller. If we randomly assigned codes without looking at how often symbols appear, we wouldn't get the same efficiency in compression.

12. 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?

- Morse code is like a language made of dots and dashes used for communication. It's good for sending messages but not efficient for compressing text into binary because it doesn't adapt to the frequency of letters. On the other hand, Huffman coding is a more advanced system that assigns shorter codes to letters that appear more often, making the message smaller. If we added more symbols to Morse code for common things like spaces and newlines, it might help a bit, but it still wouldn't be as efficient as Huffman coding because it doesn't adjust based on how often things appear in the message.

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

    - I have a text file that is 10 MB in size and shrunk to 3 MB after Huffman coding. When dividing original / compressed: `10 / 3`, the compression ratio is approximately `3.33`. This means that the compressed file is roughly `3.33` times smaller than the original file.

Program Usage

To properly use the program, enter: `huff -i -infile -o outfile` to initiate the data compresser.

To compile the data decompressor, enter: `dehuff -i infile -o outfile`.

To run specific tests on the respective function files, enter `make`, `make all`, or `make <specific test>`. Then enter `./<specific test>` to check the functionality of the functions

Testing

1. Bit reader test: `brtest.c`

    - Our test `brtest.c` checks if our program can read into files without error and reads the correct amount of bits. Also assures that the functions are able to free any allocated memory if needed. To run the test, enter `make brtest`, then run the binary executable `./brtest`. If our `bitreader.c` implementation matches the outputs of the test, the program prompts a successful message. If incorrect outputs are detected, the program prints out the line where the error has occurred in our `bitreader.c` file.

2. Bit writer test: `bwtest.c`

    - Our test `bitwriter.c` checks if our program can write into files without error and writes the correct amount of bits. Also assures that the functions are able to free any allocated memory if needed. To run the test, enter `make bwtest`, then run the binary executable `./bwtest`. If our `bitwriter.c` implementation matches the outputs of the test, the program prompts a successful message. If incorrect outputs are detected, the program prints out the line where the error has occurred in our `bitwriter.c` file.

3. Node test: `nodetest.c`

    - Our test `nodetest.c` checks if our node functions properly create the nodes and frees the memory when needed. To run the test, enter `make nodetest`, then run the binary executable `./nodetest`. If our `nodetest.c` implementation matches the outputs of the test, the program prompts a successful message. If incorrect outputs are detected, the program prints out the line where the error has occurred in our `nodetest.c` file.

4. Priority Queue test: `pqtest.c`

    - Our test `pqtest.c` checks if our queues are being properly formulated. It will detect if the creation of our bit trees are properly made. However, `node.h` functions must be able to compile in order for `pqtest.c` to give proper results. To run the test, enter `make brtest`, then run the binary executable `./pqtest`. If our `pqtest.c` implementation matches the outputs of the test, the program prompts a successful message. If incorrect outputs are detected, the program prints out the line where the error has occurred in our `pqtest.c` file.

5. Huff test: `huff-x86`

    - Our test `huff-x86` is a binary executable that returns the results of our implementation of `huff.c`. We can verify if the files are being compressed as accordingly. Since it's already good to compile, simply run `./huff-86 -i <infile> -o <outfile>` to test.

6. Dehuff test

    - Same as the above, our test `dehuff-x86` is a binary executable that returns the results of our implementation of `dehuff.c`. We can verify if the files are being decompressed as accordingly. Since it's already good to compile, simply run `./dehuff-86 -i <infile> -o <outfile>` to test.

Program Design

This program uses a group of functions and header files, respectively `bitreader.h`, `bitwriter.h`, `node.h`, `pq.h`, `huff.c`, `dehuff.c` to compile a Huffman Coding program. The `bitreader` and `bitwriter` functions serve to read and write into the files we choose. `node.c` and `pq.h` functions handle the creation of the tree data structure, something we need for our actual Huff Coding implementation. `huff.c` and `dehuff.c` take the previous functions mentioned and compress or decompress the data in the files. In their respective areas of functionality, errors in bit manipulation refer to the bit functions. Data handling and storing refer to the functions that create the trees. Finally, if files are being incorrectly compressed or decompressed, it's likely occurring in the `huff.c` or `dehuff.c` files.

Function Descriptions

1. Bit reader functions

    - `BitReader *bit_read_open(const char *filename)`
      Opens a binary or text file `filename` to read using `fopen()`, and returns a pointer to a `BitReader`. Returns NULL if the file fails to open.

    - `void bit_read_close(BitReader **pbuf)`
      Closes the binary or text file `*pbuf`. Frees all the allocated memory of the file and sets `*pbuf` to NULL. Returns a fatal error if file fails to close.

    - `uint32_t bit_read_uint32(BitReader *buf)`
      Using `bit_read_bit()` 32 times, it reads 32 bits from `*buf` starting with the least significant bit. Then it returns the word.

    - `uint16_t bit_read_uint16(BitReader *buf)`
      Using `bit_read_bit()` 16 times, it reads 16 bits from `*buf` starting with the least significant bit. Then it returns the word.

- `uint8_t bit_read_uint8(BitReader *buf)`
  Using `bit_read_bit()` 8 times, it reads 8 bits from `*buf` starting with the least significant bit. Then it returns the byte.

- `uint8_t bit_read_bit(BitReader *buf)`
  Reads a single bit from the values in `*buf`, and returns the bit.

2. Bit writer functions

   - `BitWriter *bit_write_open(const char *filename)`
     Allocates a new `BitWriter`. Opens a binary or text file `filename` to write using `fopen()` and returns a pointer to that `BitWriter`. If allocation fails, return NULL.

   - `void bit_write_close(BitWriter **pbuf)`
     Flushes any data pointed to by `*pbuf`, then frees all memory allocated in the `BitWriter` object and sets `*pbuf` to NULL. Returns a fatal error if there is a failure in freeing memory.

   - `void bit_write_bit(BitWriter *buf, uint8_t bit)`
     Writes a single bit from the values pointed to by `buf` using `fputc()`. Returns a fatal error if a failure occurs.

   - `void bit_write_uint16(BitWriter *buf, uint16_t x)`
     By calling `bit_write_bit()` 16 times, it writes 16 bits by starting with the LSB of `x`.

   - `void bit_write_uint32(BitWriter *buf, uint32_t x)`
     By calling `bit_write_bit()` 32 times, it writes 32 bits by starting with the LSB of `x`.

   - `void bit_write_uint8(BitWriter *buf, uint8_t byte)`
     By calling `bit_write_bit()` 8 times, it writes 8 bits by starting with the LSB of `x`.

3. Node functions

   - `Node *node_create(uint8_t symbol, uint32_t weight)`
     Allocates a `Node`, sets the `symbol` and `weight` fields to its respective variables, and returns a new node pointer. If allocation fails, return NULL instead.

   - `void node_free(Node **node)`
     Frees all the allocated memory of `**node` and `*node`, then sets `*pnode` to NULL.

   - `void node_print_tree(Node *tree)`
     For debugging purposes, prints out the a tree of nodes from `tree`.

4. Priority Queue Functions

   - `PriorityQueue *pq_create(void)`
     allocates a `Priority Queue` object and returns a pointer to it. Returns NULL if allocation fails.

- `void pq_free(PriorityQueue **q)`
  Frees all existing memory of `*q`, calls `free()` on `*q`, and finally sets `*q` to NULL.

- `bool pq_is_empty(PriorityQueue *q)`
  Returns `true` if `*q` is empty, `false` is otherwise. It functions by storing NULL into the queue's `list` field.

- `bool pq_size_is_1(PriorityQueue *q)`
  Returns `true` if `*q` contains only one element, and returns `false` if otherwise.

- `void enqueue(PriorityQueue *q, Node *tree)`
  Inserts a tree `*tree` into `*q` and the tree with the lowest weight is the head. If the queue is empty, the new element will become the new first element of the queue.

- `Node *dequeue(PriorityQueue *q)`
  Removes and returns the queue element with the lowest weight. Returns a fatal error if queue is empty

- `void pq_print(PriorityQueue *q)`
  Prints the trees of the queue `*q`

5. Huff Functions

   This file must refer to this Code Struct:

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

- `void huff_write_tree(BitWriter *outbuf, Node *node)`
  This function takes a pointer to a `Bitwriter` and a pointer to a `Node`. It traverses the Huffman tree and writes information about each node, whether it's internal or a leaf node. If it's a leaf node, it's a symbol as well.

- `uint32_t fill_histogram(FILE *fin, uint32_t *histogram)`
  This function takes in a pointer to a file and a pointer to an unsigned 32-bit histogram array. It reads the bytes from `*fin`, counts their frequencies, and constructs a histogram of byte frequencies stored in the `histogram` array.

- `Node *create_tree(uint32_t *histogram, uint16_t *num_leaves)`
  This function takes in an array of integers called `histogram` and a pointer to a `uint32_t` integer called `num_leaves`. It creates a Huffman tree based on a histogram of byte frequencies, and returns it.

- `void fill_code_table(*code_table, *node, code, code_length)`
  This function takes in four parameters: a pointer to a `Code` struct `code_table`, a pointer to a `Node` struct `node`, a `uin64_t` integer `code`, and a `uint8_t` integer `code_length`. It's a recursive function that traverses the tree and fills in the code table for each lead node's symbol.

- `void huff_compress_file(*outbuf, *fin, filesize, num_leaves, *code_tree,*code_table)`
  This function takes in 6 parameters: a pointer to a `BitWriter`, a `FILE`, a `uint32_t` integer `filesize`, a `uint16_t` integer `num_leaves`, a pointer to a `Node` struct, and a pointer to a `Code` struct. It then writes a Huffman coded file in order to begin the compression of data.

6. Dehuff Functions

- `void dehuff_compress_file(FILE *fout, BitReader *inbuf)`
  This function takes in a Huffman coded file `fout` and a BitReader `inbuf` to decompress the file. It essentially uses stack pointers to Node elements to access the nodes in the tree. Then the stack stores allocated nodes as they are assembled into the code tree. The function has an implementation of `stack_pop()` and `stack_push()` to push a Node pointer into the stack, or pop it out of the stack. Together, they formulate a function that decompresses a Huffman coded file.

Command Line Options

Both `huff.c` and `dehuff.c` should accept these command line options while compiling.

- -i : Sets the name of the input file. Requires a filename as an argument.
- -o : Sets the name of the output file. Requires a filename as an argument.
- -h : Prints out a help message to stdout.