Assignment 7 – Huffman Coding

William Zhou

CSE 13S – Winter 24

Purpose

Audience for this section: Pretend that you are working in industry, and write this paragraph for your boss. You are answering the basic question, "What does this thing do?". This section can be short. A single paragraph is okay.

Do not just copy the assignment PDF to complete this section, use your own words.

The purpose of this program is to imitate the Huffman coding algorithm. The program is split into two parts. One part is responsible for compressing a file. The other part is responsible for decompressing a file. Note that no data will be lost from the compression and decompression process as the Huffman algorithm is lossless.

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Be sure to test inputs with delays.

In this assignment, test files are provided for me to compress and decompress. I'm also given already working implementations of compression and decompression. I will run my program alongside the given implementation on each test file and compare outputs to make sure my program is working as intended.

Test files are also provided to me to test my function implementations in bitwriter.c, bitreader.c, node.c, and pqtest.c. I can utilize those to make sure my implementations are correct.

I will utilize valgrind to ensure all my dyamically allocated memory is being freed and that there are no leaks.

As I'm coding, print statements every 5-10 lines of code to check for functionality will be employed to make sure my code is working how I want it to. This way I can debug my code in bite sized, more manageable amounts.

As I'm coding, once I finish the node_print_tree() function, I can use that to view the Huffman trees being produced and compare them with trees made using a website from the link belowgiven the same file input to ensure proper functionality. ***https://cmps-people.ok.ubc.ca/ylucet/DS/Huffman.html***

How to Use the Program

Audience: Write this section for the user of your program. You are answering the basic question, "How do I use this thing?". Don't copy the assignment exactly; explain this in your own words. This section will be longer for a more complicated program and shorter for a less complicated program. You should show how to compile and run your program. You should also describe any optional flags or inputs that your program uses, and what they do.

To show "code font" text within a paragraph, you can use `\lstinline{}`, which will look like this: `text`.

For a code block, use `\begin{lstlisting}` and `\end{lstlisting}`, which will look like this:

```
Here is some code in lstlisting.
```

[1] This question is a whole lot more vague than it has been the last few assignments. Continue to answer it with the same level of detail and thought.

```
def 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
```

Figure 1

And if you want a box around the code text, then use `\begin{lstlisting}[frame=single]` and `\end{lstlisting}`

which will look like this:

```
Here is some framed code (lstlisting) text.
```

Want to make a footnote? Here's how.[2]

Do you need to cite a reference? You do that by putting the reference in the file `bibtex.bib`, and then you cite your reference like this[1][2][3].

To use this program, first type in "make" to create the proper executable files. If you want to compress, type in "huff -i" followed by the file you want compressed, followed by a "-o", followed by where you want the resulting compressed file to go. If you want to decompress, type in "dehuff -i" followed by the file you want decompressed, followed by a "-o", followed by where you want the resulting decompressed file to go.

Program Design

Audience: Write this section for someone who will maintain your program. In industry you maintain your own programs, and so your audience could be future you! List the main data structures and the main algorithms. You are answering the basic question, "How is this thing organized so that I can have a chance of fixing it?". This section will be longer for a more complicated program and shorter for a less complicated program.

In this program, each ADT impelementation is in its own file. Priority queue is implemented in pq.c. Node is implemented in node.c. Bitreader is in bitreader.c. Bitwriter is in biwrter.c. The main functions performing the compression are in huff.c.

Pseudocode

Give the reader a top down description of your code! How will you break it down? What features will your code have? How will you implement each function.

Here is the pseudocode for the main compression algorithm:

Pseudocode for the main decompression algorithm:

Function Descriptions

For each function in your program, you will need to explain your thought process. This means doing the following

1. The inputs of every function (even if it's not a parameter)
2. The outputs of every function (even if it's not the return value)
3. The purpose of each function, a brief description about a sentence long.

[2]This is my footnote.

```
def dehuff_decompress_file(fout, 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
```

Figure 2

4. For more complicated functions, include pseudocode that describes how the function works

5. For more complicated functions, also include a description of your decision making process; why you chose to use any data structures or control flows that you did.

Do not simply use your code to describe this. This section should be readable to a person with little to no code knowledge. **DO NOT JUST PUT THE FUNCTION SIGNATURES HERE. MORE EXPLANATION IS REQUIRED.**

Bitwriter functions:

BitWriter *bit_write_open(const char *filename);

Takes in a char string. Returns a pointer to a bitwriter. Purpose is to open a binary of textfile specified.

void bit_write_close(BitWriter **pbuf);

Takes in a double pointer to a BitWriter struct. Returns nothing. Purpose is to remove data from byte buffer, close underlying_stream, free bitWriter object, and set the pointer to NULL. Checks all function return values and reports a fatal error if any return values report a failure.

void bit_write_bit(BitWriter *buf, uint8_t bit);

Takes in pointer to bitwriter and an int. Returns nothing. Purpose of this function is to collect 8 bits into a buffer and then writing it using fputc(). Checks all return values and reports a fatal error if any of them report a failure.

void bit_write_uint8(BitWriter *buf, uint8_t x);

Takes in pointer to bitwriter and an int. Returns nothing. Calls bit_write_bit() 8 times and writes 8 bits.

void bit_write_uint16(BitWriter *buf, uint16_t x);

Takes in pointer to bitwriter and an int. Returns nothing. Calls bit_write_bit() 16 times and writes 16 bits.

void bit_write_uint32(BitWriter *buf, uint32_t x);

Takes in pointer to bitwriter and an int. Returns nothing. Calls bit_write_bit() 32 times and writes 32 bits.

Bitreader functions:

BitReader *bit_read_open(const char *filename0);

Takes in a char string and returns a bitreader pointer. Opens the binary file specified to read it and store the result.

void bit_read_close(BitReader **pbuf);

Takes in double pointer of a bitreader. Returns nothing. Closes underlying_stream, frees the BitReader object, and sets pointer to NULL. If any return values report failure, function reports a fatal error.

uint8_t bit_read_bit(BitReader *buf):

Takes in pointer to bitreader. Returns a bit. Purpose is to read in bits.

uint8_t bit_read_uint8(BitReader *buf);

Takes in a pointer to BitReader. Returns an int. Calls bit_read_bit() 8 times and reads 8 bits.

uint16_t bit_read_uint16(BitReader *buf);

Takes in a pointer to BitReader. Returns an int. Calls bit_read_bit() 16 times and reads 16 bits.

uint32_t bit_read_uint32(BitReader *buf);

Takes in a pointer to BitReader. Returns an int. Calls bit_read_bit() 32 times and reads 32 bits.

Node functions:

Node *node_create(uint8_t symbol, uint32_t weight);

Takes in 2 ints. Returns a pointer to a new node. Purpose is to create a new node. Symbol and weight are set by the respective int values provided.

void node_free(Node **pnode);

Takes in double pointer to a node. Returns nothing. Purpose is to free *pnode and its children, and set pointer to NULL.

void node_print_tree(Node *tree);

Takes in pointer to a node. Returns nothing. Purpose is to give a visual of the node to help with debugging.

Prority Queue functions:

PriorityQueue *pg_create(void);

Takes in nothing. Returns a pointer to a priorityqueue. Creates priorityqueue object. If there is an error, return null.

void pq_free(PriotiyQueue **q);

Takes in double pointer to priority queue. Returns nothing. Frees up the priority queue and sets pointer to null.

bool pq_is_empty(PriorityQueue *q):

Takes in pointer to priority quee. Returns a boolean. Returns true if queue is empty. Returns false otherwise.

bool pq_size_is_1(PriorityQueue *q):

Takes in pointer to priority queue. Returns a boolean. Returns true if the queue has one element. Otherwise returns false.

bool pq_less_than(ListElement *e1, ListElement *e2);

Takes in two listelement objects. Returns a boolean. Returns true if weight of the first element is less than weight of the second element. If weights are tied, then returns true if their respective symbol of the first element is less than the symbol of the second element.

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

Takes in a pointer to a priorityquee and a pointer to a node. Returns nothing. Purpose is to insert the node tree into the priority queue. Tree with the lowest weight is at the head.

Node *dequeue(PriorityQueue *q);

Takes in pointer to a priority queue. Returns a node pointer. Purpise is to remove the queue elemtn with the lowets weight and return it. If queue is empty, report a fatal error.

void pq_print(PriorityQueue *q);

Takes in pointer to a prioority queue. Returns nothing. Purpose is to print the trees inside the queue.

Huffman functions:

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

Takes in pointer to a file and int. Returns an int. Purpose of the functino is to update a histogram array of int values with the number of times each unique byte occured in the file.

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

Takes in 2 ints. Returns a node poniter. Purpose is to create a new Huffman tree.

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

Takes in 2 ints, a node, and a code table. Purpose is to recusively fill in the code table for each leaf node's symbol.

void huff_compress_file(outbuf, fin, filesize, num_leaves, code_tree, code_table);

Takes in bitwriter, file, 2 ints, and a pointer to a code tree and code table. Returns nothing. The purpose is to write a huffman compressed file.

void dehuff_compress_file(FILE *fout, BitReader *inbuf);

Takes in a file and bitreader pointer. Returns nothing. Purpose of the function is to get a huffman compressed file and decompress it into its original state.

Questions

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

1. (a) The goal of compression is to reduce the file size of something. There are various means of doing so. "aaaaaaaaaaa" can be easily compressed using hufman coding. Since it is the most common (and only) character, you can assign it a small prefix code like 0 and the resulting encoding value would just be 000000000000.

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?**

   (a) The key difference between lossy and lossless compression is that lossy compression permanently discards information while lossless doesn't. As a result lossy typically results in a larger reduction in file size. Huffman coding is a lossless compression type as no information is lost through creating prefix codes and making an encoding value. JPEG is a lossy compression format as information is permanently discarded to reduce file size. You can lossily compress a text file but depending on the situation that might not be a good idea. When you lossily compress a text file you may end up losing important information from that text file.

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

   (a) My program can theoretically accept any type of file as input. The effectiveness of Huffman coding will vary depending on the file. If there are very few repeating characters, Huffman coding's effectiveness will diminish substantially. In some cases, it may actually make the file larger from the large length of some codes assigned to the less frequent symbols.

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

   (a) Question is a little bit confusing.

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

   (a) The picture is 1080p resolution. It takes up 2.9e+6 bytes.

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?**

   (a) 2073600 pixels * 3 bytes/pixel = 6220800 bytes. I think the image I took is smaller because a built in compression process probably executed to reduce the size of my photo.

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.**

   (a) 2.9/6.2 = 0.467

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?**

   (a) I don't think running the Huffman program to compress an already compressed image will do a whole lot. After the first compression everything is already encoded into a string consisting of 1's and 0s. Performing compression again would just convert the 1's and 0's into an encoding string that is identical to the original (or have the 1's and 0's flipped). This does not make the file any smaller.

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

   (a) Yes, there are multiple ways to achieve the same smallest file size. I can prove this by providing a simple example. Say we have a text file with "aabbab". We can compress this file using Huffman coding and convert the text file into the encoding string by assigning A to 1 and B to 0 or vice versa as both letters have the same frequency. This will lead to the encoding string "110010" or "001101". The encoding strings are different, but they are the same size and do just as good of a job of compressing the original text file.

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

    (a) No, internal nodes will not have symbols. Only the leaves of the tree correspond to symbols in the input data. The internal nodes are meant to help create the hierarchical structure of the tree, not to represent symbols.

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

    (a) We create a histogram so that we can identity what symbols occur the most. With this information in mind, we can give the symbols that occur more frequently a shorter code. It is in our best interest to do this as this leads to more optimal compression and thus smaller file sizes.

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?**

    (a) Morse code does not work as a way of writing text files as binary. The issue is that Morse code only has representations for the alphabet and numbers. You could remedy this by just creating representations for other symbols, but the more you add, the longer some of the symbols have to be because there cannot be overlaps in representation. At a certain point this will become very memory inefficient and simply not worth it.

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

    (a) 0.58 I used candid.txt

References

[1] Wikipedia contributors. C (programming language) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/C_(programming_language), 2023. [Online; accessed 20-April-2023].

[2] Robert Mecklenburg. *Managing Projects with GNU Make, 3rd ed.* O'Reilly, Cambridge, Mass., 2005.

[3] Walter R. Tschinkel. Just scoring points. *The Chronicle of Higher Education*, 53(32):B13, 2007.