Assignment 7 - the Huffman decoder

Catherine Bronte

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.

This program builds a basic compressor and decompressor called the Huffman coder. First the program figures out the most and least common characters in a text file. Then is assigns longer codes for the less common characters and shorter codes for the more common ones. Then a new file is written based on the codes and you get your compressed file. If you want to take a Huffman compressed file and decompress it, is essentially does the same thing but backwards, and it will return the uncompressed version.

Questions

Please answer the following questions before you start coding. They will help guide you through the assignment. To make the grader's life easier, please do not remove the questions, and simply put your answers below the text of each question.

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

  The goal of compression is to find some shorter way to represent the characters or bytes in a file, so the file becomes smaller and easier to transmit. For example, it would be very easy to compress "aaaaaaa" because instead of storing every ascii table 8 bit value however many times there are the letter a, you can just store that many bits as ones, because there is nothing the character a needs to distinguish from. Basically making a = 1 instead of 97, or 01000001, which saves space. The less variety the greater the compression ability because you need less to distinguish each item, so the codes can be smaller.

- 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?

  Lossy means some of the original data of the file is lost permanently (strategically picked data of course, not just random bytes) in order to save space, and lossless means some data is compressed so it can be rebuilt later to its full value. You basically loose nothing. Huffman is lossless because you can get back to the exact original data through decompression. Jpeg is usually lossy because our eyes aren't perfect so we wont super notice if some detail is permanently lost. BUT, you can do lossless compression for an image if you want. You can absolutely compress a text file because text usually has some patterns you can compress by.

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

  Yes, as long as it is a verified file type, like jpeg or txt, and it has byte representation you can access. At the end of the day the program does not look at the file type, it just reads the bits of the file, so as long as there are bits, its good to go. However, sending every file through a huffman tree will not always result in compression. Sometimes, depending on the patterns you get in the file, and how long

it is, you can have cases where you get expansion. This is when the resulting file is larger than the starting file.

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

  Depends on the specific text file you are trying to use but it can be anywhere from one bit long for when you have text files that are just "aaaa" or it can be like 8 bits if you are trying to capture every character in ASCII. In theory, it could be 20 bits long if you have that kind of character variety, it really just depends.

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

  It is 1.8 mega bytes. The resolution is 3088 by 2320 pixels.

- 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?

  In theory this should take 21 megabytes. They probably get it down to 1.8 through a combination of lossy and lossless compression. Lossy for the colors and details we cant see and lossless to compress the patterns.

- 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.

  1.8 / 21 = .0857

- Do you expect to be able to compress the image you just took a second time with your Huffman program?1 Why or why not?

  No, usually you get expansion after the first round of Huffman compressing. Trying to compress random bits as apposed to bits that follow a pattern usually lead to expansion. And as such, the bits saved is a file after being compressed usually follow a random parrten, whithout knowing the huffman codes.

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

  You bet, I'm sure there are wizards who come up with this stuff all of the time. We learned different ways to do this in class but essentially compression is a customisation process that really depends on how you want to compress it. You can lossless compress large files many different ways and probably get similar results.

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

  No, because this means that one symbol will be a prefix for another symbol, which does not happen in Huffman coding. That's like, the entire point. For that not to happen.

- Why do we bother creating a histogram instead of randomly assigning a tree

  Because the beauty of the system and what really generates the compression is that the most common characters get the smaller codes. If we randomly assign codes of varying length, then our common letters can get codes longer than their original representation, which will expand the file rather than compress.

- 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?possible

  Both are similar in that both forms are intended to shorten communication. Morse code does not work so well in binary because you need spaces between each character to distinguish them, and that takes a great deal of space in binary. In theory you can create additional space codes for the space meant to separate words instead of characters, but this just takes more space and makes things longer, which is inefficient.

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

  (dont forget to do this xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx)

Testing

List what you will do to test your code. Make sure this is comprehensive. [1] Remember that you will not be getting a reference binary [2].

The testing I am thinking of does not particularly involve writing tests as it does comparing data between the example Huffman program. If I plug in a text file to the given program, and get a compressed outfile, my outfile from my program should match it since they are using the same coder. Also as always, print statements are very handy when it comes to checking how far your program gets before breaking.

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.

If you would like to compress a text file, simply do ./huff -i (file.txt or jpeg or whatever) -o (where you would like the file to go.txt). If you would like to decompress then do ./dehuff -i (your compressed file.txt) -o (where you would like the file to go.whatever). If you need help at any point you can type in ./huff or ./dehuff -h. This will print out a help message explaining what thee inputs do. Note, you can only effectivly decompress files that have been compressed by a huffman compressor.

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].

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.

(not a section??)

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.

I will only be including pseudo code for the two main programs huff and dehuff since the remainder of the program is described in the functions section.

huff:

define main with input of int argc and char double pointer argv, return int fizz var assigned as a char pointer, used for reading in files opt var is an int for options in command line options char pointer funky out is for name of outfile

command line options section while options does not equal -1 if i, set fizz to optarg if fizz is null, exit with error if o, funky out is set to optarg if h, print help message and exit default to error with help messsage

fizz is opened as fi uint32-t histogram is allocated file-size = fill histogram output (fi, histogram)

num leaves is declared as uint16-t complete fano tree = create tree output(histogram, numleaves)

code table is allocated fill code table(code table, complete fano tree, 0, 0)

outfile bw is allocated with bit write open output(funky out)

huff compress file(outfile bw, fi, file size, num leaves, complete fano tree, code table);

[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.

[2] The output of your binary is not the only thing you should be testing!

all memory is freed exit 0
dehuff:
everything from the main definition to the end of the command line options block is the same as huff
new in is a bitreader that is allocated with the bit read open function(fizz) dehuff decompress file(funky out, new in) everything is closed and freed exit 0 with no error

Function Descriptions

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

- bit write open: Takes a filename pointer and returns a bit-write pointer. Makes a new bitwriter and sets underlying stream to the file pointer. Byte and but position also get set to 0.
- bit write close: Takes a double pointer to a bit writer, and returns void. Closes the file associated with the underlying stream, and frees all of the data associated with the bitwriter.
- bit write bit: Takes a pointer to a bit writer, and a uint8-t value, returns void. If the bit position index is greater than 7, byte gets written to the buffer. Regardless of that, the bit position index of byte gets set to bit (the uint8-t input value). Then bit position gets incremented by 1.
- bit write uint8: Takes a pointer to a bit writer and a uint8-t value, and returns void. Calls bit write bit 8 times to fill up the struct byte. Bits are filled in right to left.
- bit write uint16: Takes a pointer to a bit writer and a uint8-t value, and returns void. Calls bit write bit 16 times to fill up the struct byte. Bits are filled in right to left.
- bit write uint32: Takes a pointer to a bit writer and a uint8-t value, and returns void. Calls bit write bit 32 times to fill up the struct byte. Bits are filled in right to left.
- Bit read open: Takes a filename pointer and returns a bit reader. Allocates memory for a new bit reader, and sets underlying stream to the file pointer. Byte and but position also get set to 0.
- bit read close: Takes a double pointer to a bitreader and returns void. Closes the file assigned to bit reader and frees all associated memory.
- bit read bit: Takes a bit reader pointer and returns a uint8-t value. If bit position is greater than 7, a new character is read into temp, checked if it is -1, and then assigned to byte. Then through a few bit wise operations, the bit at bit position is retrieved and assigned to one-bit. Bit position is incremented by one, and then one bit is returned.
- bit read uint8-t: Takes a bit reader and returns a uint8-t. Calls bit read bit 8 times to retrieve 8 bits, and each one is bit wised stored to it proper position in byte. Byte is then returned.
- bit read uint16-t: Takes a bit reader and returns a uint16-t. Calls bit read bit 16 times to retrieve 16 bits, and each one is bit wised stored to it proper position in word. word is then returned.
- bit read uint32-t: Takes a bit reader and returns a uint32-t. Calls bit read bit 32 times to retrieve 32 bits, and each one is bit wised stored to it proper position in word. word is then returned.
- file return: takes a bit reader and returns a file pointer. This function goes into the bit reader and returns the file pointed to by underlying stream.
- node create: takes a uint8-t symbol and a uint32-t weight and returns a node pointer. Allocates memory for a new node while setting the nodes weight to weight, symbol to symbol, code to 0, code length to 0, left to NULL, and right to NULL. The node is then returned.
- node free: Takes a double node pointer and returns void. frees node-¿left and right, frees the node its self, and sets node to NULL.

- node print node: takes a Node pointer, a character, and an int. returns void. prints out the node and all of its recursive lefts and rights. Associated with node-print-tree.
- pq create: takes void and returns a priority queue pointer. Allocates space for a priority queue, and returns the pointer from the allocation.
- pq free: Takes a double priority queue pointer and returns void. Traverses through all list elements in priority queue untill is hits null, and through each pass, frees the previous element. Then at the the end of the loop, it frees the current element, the priority queue, and then sets it to NULL
- pq is empty: Takes a priority queue pointer and returns a bool. If the priority queues list section is null it returns true, if not it returns false.
- pq size is 1: Takes a priority queue pointer and returns a bool. If pq is empty is false and (q¿list)¿next == NULL, it returns true. If not it returns false.
- pq less than: takes two list element pointers and returns bool. If the weight of the first element is smaller than the second, true is returned. If they are equal then if their symbols are also equal then true it returned. If not either of those then it returns false.
- enqueue: Takes a priority queue pointer and a node called tree. Returns void. Allocates a new list element, sets its tree to the given tree and its next to null. If the given priority queue is empty the queues list is just set to the new element. If the new element is smaller than the existing element in the list, the new element becomes the first item in the list, with the existing first being placed second. If that does not happen then a loop traverses the list, checking if the new element is smaller than the existing elements. If it is it is inserted where the original element was. If it makes it to the end, it is just inserted there.
- dequeue: takes a priority queue pointer and returns a node pointer. a node returner is set to tree from the priority queue list. Then the list is updated, everything is freed and det tuo null, and returns returner
- fill histogram: takes a file pointer and a uint32-t pointer histogram. returns a uint32-t. Filesize is set to zero. while i = 0 ¡ 256, index i of histogram is set to zero. two items are added to histogram 00 and ff. A character is read from the infile, and if it is -1 the program exits with error. Current item is set to this character. Untill the character read is -1, histogram adds in the read character, filesize is incrimented, and a new character is read in. filesize is returned.
- create tree: Takes a uint32-t pointer histogram and a uint16-t pointer num leaves. returns a node pointer. A new priority queue is made called my queue. while i = 0 ¡ 256, if histogram at index i does not equal zero, a new node is made with the histogram information, num-leaves is incrimented, and the node is enqueued into the new queue. Node left and right are set to null. While our queue has more than one item in it, left and right are dequeued, a new node is created, the left and rights of the new node are set to left and right, and the new node is enqueued. The queue is then dequeued and returned, with the information being freed.
- fill code table: takes a code pointer code table, a node pointer, a uint64-t code and uint8-t code length. Returns void. If the node is not a leaf the function is recursivly called with the child and code length being incrimented by one. If it is a leaf, the code is added to the code table
- huff write tree: Takes a bitwriter pointer and a node pointer. Returns void. If the node is not a child, write one to outbuf, and write the symbol to outbuf. If it is not a leaf recursivly call with the child, and then write zero to outbuff.
- huff free tree: Takes a node pointer with a fano tree in it and returns void. Frees all memory associated with the tree.

- Huff compress file: Takes a bit writer pointer, a file pointer, a filesize, the number of leaves, a code tree and a code table. Returns void. The file is rewound to the begining, and H, C, num leaves, file size, and code tree is written to outbuff. While true, if the character read in is eof, break. Write a bit for the lenght of the code at the position of the code table, decided by the character read in.
- dehuff decompress file: Takes a file pointer (fout) and a bit reader (inbuf). Returns void. A crude stack is made and all of the file basics are read in. Number of nodes in file is calculated. a new node is set to null. For every node in the file, read in from the inbuff. If it is one, make a new node from reading in 8 bits, and set it to the node. If its not 1, Make a new node have have its left and right set to two item in the stack. The node is added to the stack. A code tree is derived from the stack. For i in the filesize, every node in the code tree is retraced back to its origional character, and the character is written out.

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.