Jinsung Park
jpark598

Assignment 6: Huffman Coding
Design

Overview

Huffman coding is an approach to compressing a file by counting the frequency of a character. It optimizes the truncation by replacing the most used characters with the smallest bit. Program constructs a Huffman tree using priority queue and uses that to construct a code table.

In this lab, I implement Huffman encoding and decoding. For encoding, the program generates a histogram of the frequencies of each character in a file. There are 255 characters in total. It will use that histogram to create a Huffman tree using priority queue. Evaluating each node in the tree, it creates a code. Program adds a constructed header which contains information about the file to the outfile.

Command-line options:

- h: help message
- i: name of infile
- o: name of outfile
- v: turn on option to print compression statistics

For decoding, the program will reconstruct the Huffman tree from the dumped tree. It reads each bit at a time from the infile, traversing down the tree. Once there is nothing to read, the program ends.

command-line options:

- h: help message
- i: name of infile
- o: name of outfile
- v: turn on option to print decompression statistics

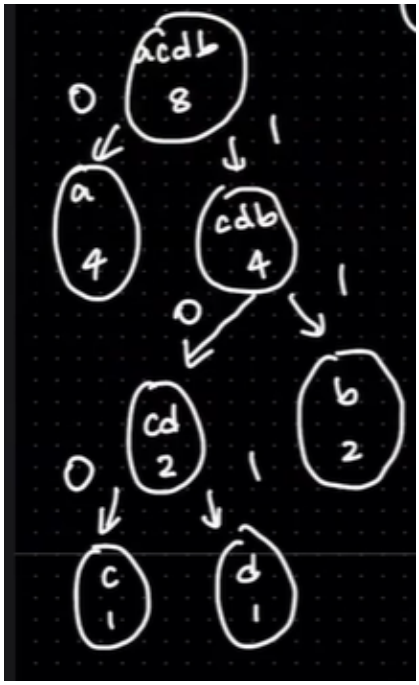
Top level

```
Histogram:
        for i in file:
                histogram[i]++
```

To determine which character occurs the most to optimize space, I create a histogram. It measures the frequency of all 255 characters.

```
Huffman_tree:
        for word in histogram:
                pq_push(word)
        while(pq_size > 1):
                pq_push(node_join(pq_pop(), pq_pop())
```

Huffman tree is a priority queue that represents the most occurring character at the front and least occurring character in the back. It will first shove all characters in an ordered list. Then, it will join nodes until there is only one root node left.



The Huffman tree looks very much like this. Taken from Enguen's lab section (5/11/2021)

```
code_construction(code ):
        if(left_node != null):
                code_push(code, 0)
                code_construction(code)
                code_pop(code)
        if(right_node != null):
                code_push(code,1)
                code_construction(code)
                code_pop(code)
        if(left_node == null and right_node == null):
                codes[symbol(node)] = code
```

Using the huffman tree, this code populates the code array with bits that represent each character. It is essential that the program finds all leaves in the tree so that code for every symbol can be generated. The piece of code above will iterate through left and right branches. Every left branch it goes to, it adds 0 to the end of code and every right branch it goes to, it adds 1. Once it finds a leaf which is signified by having no left or right nodes from the current node, it will add that code to the table of code which will act as a lookup table for encoding messages. Code works essentially as a replacement for the uncompressed characters but is smaller in size because it is often smaller than 8 bits while all characters in the uncompressed file are represented by 8bits.

```
file_encode:
        for each char ch in file:
                write_file(code[ch]);
```

This will read the infile, then use the “code” cheat sheet to the outfile in a compressed form. Nothing more to be said. It just reads the file, figures out what that character should be in code, then writes that code into the outfile.

```
create_postorder_treedump(node):
        if(node != null):
                a = left_node(node)
                b = right_node(node)
                if(a==null and b==null):
                        write(‘L’)
                        write(symbol(node))
                else:
                        write(‘I’)
        return null
```

Create a treedump by visiting each node. Writes L in front of a symbol if it is a leaf Or I to represent an internal branch. Treedump goes along with the outfile so the person who receives the file can decode it. There are multiple ways of doing this. Hence, this pseudocode may look a little different from the one I’ve used in my program but they are virtually the same with minor differences in code clarity.

```
reconstruct_postorder_treedump:
        n = new_node()
        while i < length(treedump):
                if(treedump[i] = ‘L’):
                        stack_push(treedump[i+1])
                        i+=2
                if(treedump[i] = ‘I’):
                        right = new_node(stack_pop())
                        left = new_node(stack_pop())
                        stack_push(node_join(n, left, right)
                        i += 1
        return n
```

Once the decoder has received the encoded file, it must decode it. But before it can do that, it needs to reconstruct nodes so it can figure out what code corresponds to which character. This program will read the treedump, pushing a node into a stack when ‘L’ is encountered, and when ‘I’ is encountered, it pops two items and joins them. It continues until the stack only has one node. This reverses the creation of postorder treedump.

Once the nodes have fully joined and what remains is a single root of the tree remaining in the stack, all is well done.

```
file_decode:
        n = nodes(root)
        while(file_end != true):
                b = readbit
                if(b == 0):
                        n=left_node(n)
                        if(left_node(n) == null and right_node(n) == null): //found leaf
                                write_file(symbol(n))
                                n = nodes(root)
                else:
                        n = right_node(n)
                        if(left_node(n) == null and right_node(n) == null):
                                write_file(symbol(n))
                                n = nodes(root)
```

File_decode will traverse the nodes to determine the character we need to print to outfile. It reads one bit at a time. If the current bit is 0, it will look at the left node. If the current bit is 1, it looks at the right node. If the current node is a leaf, it will write to the outfile and go back to the root once again.

Reading and writing file (I/O)

Reading files is undoubtedly an important part of this lab. I am tasked to create a function to read and write bytes using open(), close(), write(), and read(). io.h manages all reading and writing the program conducts.

write_bytes is a wrapper for writing bytes into an outfile. It is a wrapper because it ensures that writing is only done up until nbytes have been read or the buffer is empty.

read_bytes is very similar to write_bytes. Instead of writing, it will read from infile until it reaches the end of until nbytes of characters have been read.

```
read_bit(bit):
        if index == bytes_read * 8:
                refill buffer
                bytes_read = number of bytes of read into buffer
                index = 0
        bit = buffer[index/8] >> (index%8);
        if index == 8 * bytes_read and index != BLOCK * 8:
                return false
        index+=1
        return true
```

read_bit will read a bit in the buffer. And return through a bit pointer. If there is no more bits to be read or in other words, it will return false and true otherwise.

```
write_code(c):
        for i < size(c):
                buffer[index/8] |= c.bits[i] << (index %8)
                index+=1
                if index == BLOCK * 8:
                        write to file
                        index = 0
```

Outside of this function, the program will read characters off of the infile. Everytime it reads a character, it will find the code value of that character and call write_code. Write_code will use that given code to write into a buffer.Note that it will write MSB on the left hence why I am bit shifting left. Once the buffer is full, it will write to file, and continue looping until all bits in the code have been read off.

```
flush():
        write_bytes(buffer)
```

Flush is used to write out any bits in the buffer that may not been written because index did not hit that magical number BLOCK * 8 in write_code. Flush is necessary because of the way write_code function is written, it will only write to outfile if the buffer is full.

Permission and reread of file

When piping an infile to encoder, it is not possible to traceback the origin by any means (any known means). This means lseek does not work on the file. This is important because my program requires that I read through the infile twice: once to generate a histogram by counting the number of occurrences of each character and once more to read and encode the messages.

```
int val = lseek(infile)
if val == -1:
        lseek(0, tempfile)
        file = tempfile
else:
        file = infile
```

To bypass this issue, I create a temp file using tempfile(). As I read the infile for the first time, will make a copy of the infile into the temp file. If I determine that lseek is not possible, I will read from the tempfile, if not I will lseek the original infile to reread the file again.
If lseek is not possible, I cannot get fstat to find the permission of the original infile so I do nothing. If lseek is possible, I will find the permission through fstat and chmod the outfile to set permission.

Priority Queue

Priorityqueue is an interesting piece of the program that makes the world go around. Yes literally, it spins endlessly. There are multiple ways to go about creating this “PriorityQueue” but the most efficient and easiest I found was using an insertion type queue that inserts nodes based on frequency. Of course, I could have used a min heap but that certainly would have been a lot more code to write.

```
enqueue(node):
        view_index = top
        top += 1
        while frequency(pq[view_index]) < frequency(node):
                pq[view_index+1] = pq[view_index]
        pq[view_index] = node
```

Above is a very simple demonstration of my enqueue logic. It will get the index at the end of the queue and slowly move up the elements to give space in the middle. If the frequency is less than or equal to the node you are going to add, stop the loop and insert the node there.

```
dequeue(node):
        node = pq[0]
        head+=1
```

Dequeue is very easy. First get the first item in the list, then move up the header. Note that the queue is looping so every increment in index is ((head+1) mod capacity)

Stack

Stack is used for reading the tree dump and reconstructing the tree of nodes. As you read the tree dump, you insert each symbol into the stack if you find an ‘L’. If you see a ‘I’, pop two nodes and join them together. Pseudocode is provided above.

Code

Essential part of encoding and the backbone of huffman’s fundamental usage. Code is constructed based on the frequency of the character. The less frequent it is, the larger its code is. Presumably, this means that a file with high entropy will not compress as much as a file with not a lot of unique characters. This is so because if there are only two characters for instance, those characters can be represented by 0 and 1. While a normal uncompressed file will use up 8 bits for each character, it will be ⅛ the size of that.

Code consists of an array of 8bits (those only one bit will be used). Each element in the array will use up one bit in the encoded message.

Statistics

statistics for the encoder is given by the following formula(courtesy of the assignment doc):

$$100 \times (1 - (\text{compressed size} / \text{uncompressed size})).$$

statistics for decoder is given by the following formula(courtesy of the assignment doc):

$$100 \times (1 - (\text{compressed size} / \text{decompressed size})).$$

compressed, uncompressed, decompressed file size can simply be calculated by counting the number of bytes that are being written or using fstat if you can.

Design process

Throughout the process of understanding the constituent of the lab, I watched the lab sections multiple times and drew nodes tree to better visualize what needs to be done.

- I first began with creating Node which is the fundamentals of how we structure the data. Then the priority queue came next. I chose insertion sort type enqueuing because I believed it to be the harder approach and ultimately faster in terms of computational runtime. However, I do not know if that is the case after speaking with others.
- IO is clearly a big hurdle that must be faced in this lab. Instead of relying on <stdio.h> library to read and write files, I must create one from scratch to suit the needs of the program. Confusing times were had.
- After completing the IO, I had to work with writing code into the program. LSB to MSB. I figured the bits had to be in reverse order when printing. And flushing was no problem.
- Decode was a lot easier than encode, after all, everything is laid out in front of you. Since my nodes and IO worked, there was no issue figuring out the rest.
- When piping into a file, lseek is not possible. In such cases, permissions cannot be read. I will instead have to use default file perms.
- It is also very important to make sure int size is sufficient for what it's intended to be used for.