CSE 13S
Assignment 5 - Design

Jackson Browne - jbrowne2@ucsc.edu

October 28, 2021

1 Introduction

The key idea within this program is to implement Huffman Coding, which is a form of static encoding. Huffman Coding uses something called *entropy*, which is a measure of the amount of information in something, in our case symbols, or characters, in a file. Static encoding is going to assign the least number of bits to the most frequently occurring symbol. Therefore we may use the ideas discussed in the sections below to implement a way to parse a file, get a histogram of its characters, and assign the amount of bits to each according to their frequency.

2 Encoder

The encoder will reach in an input file and find the Huffman encoding of its contents. The algorithm must compute a histogram of the symbols in the file, construct a Huffman tree (see §4), construct a code table (see §5), emit an encoding of the Huffman Tree to an out file, then finally for each symbol emit its code to the output file.

We may do the following steps to encode our file

- Read the infile and make a histogram of each possible symbol (256)

  For each symbol in the histogram with a frequency greater than zero, create a Node and push it to the queue.

- While there are two or more nodes in the priority queue, dequeue twice. Dequeue the left child then the right. We join these two nodes together using the node_join function and enqueue the resulting parent.

- Once there is only one node left in the priority queue, we have our huffman tree.

- Construct a code table. We fill this by travering the huffman tree and keep track of each left and right child we visit. This is a post-order traversal. If we visit the left child, add a one to the stack, vice versa for a right child. Once we hit a leaf node, we save the code into the code table in respect to the leafs symbol.

- Write our header to the outfile
- Write the constructed huffman tree to the outfile using dump_tree()
- Starting at the beginning of the infile, write the code for each symbol to the outfile with write_code(). Then flush_codes().

2.1 Histogram

For our histogram in particular, we know that we are working with ASCII-symbols, of which there are 256 symbols. Therefore we may construct an array of 256 items with indices 0-255, and use the symbols ascii value as its array index. In the array itself, the at the specified index is going to the be frequency of our symbol. We will do this until we reach the end of the file.

2.2 Huffman Tree

Using our histogram we can easily construct our Huffman Tree by using a *heap*. See §4 for more on the construction of the Priority Queue. The process for this consists of the steps as follows. We start by taking the two lowest priority symbols currently in the priority queue. We join them by setting their parent node to symbol '$' where its frequency is the sum of the frequencies of the children being joined. We then put that back into the Priority Queue and *pretend* its children "do not exist". We do this process until we are left with only a single node in our Priority Queue.

2.3 Code Table

The code table consists of an array where each index of the table represents a symbol and the value associated with it is going to the be symbols code. We will be using a *stack* of bits as we traverse our tree in order to calculate our code table. As we step through our Huffman tree, we will keep track of which child nodes we are visiting. We notate a left child visit as a 0 and a 1 for a right child. We will continue this process until we reach a leaf node. How do we know we've reached a leaf node? When there are *no children*. We will then pop the values off the stack until the stack is empty and add the resulting bits into our code table. For example say we traverse the left child twice and the right once to reach a symbol of 'a'. This would mean our code for 'a' would be 001. It is important to note that these values will also later be used as instructions for decoding.

3 Decoder

The process for decoding a file is much more streamlined and easier to understand than that of encoding it to begin with. Since we have our output of the encoding, which essentially contains instruction to decode it, we can just simply work backwards.

To begin with, we are given our dumped tree from our input (encoding output) file. Using this dumped tree, we can simply follow its "instructions". We read in the input file bit-by-bit and traverse down our Huffman tree following the same instructions we used to encode it in §2.3 where a left child visit is a 0 and a right child visit is a 1. Whenever we reach a leaf node, we emit its symbol and we start back again from the root. This process is going to be some-what similar to that of the depth-first-search we used previously.

- Read the header from the infile and confirm the magic number is accurate.
- Set the permissions using fchmod()
- Read the dumped tree size from the header. Read the infile into a array tree_size bytes long.
- Use rebuild_tree() to reconstruct the tree:

  Iterate over the contents of the tree dump.

  If the element is 'L' then we have a leaf node, create a node and push it to the stack.

  If the element is a 'I' then we pop the stack once to get the right child of the interior node. Then we pop again to get the left. Join the two children together and push the parent onto the stack.

  Once there is only one node left in the stack, the resulting node is the root of the Huffman Tree.
- Read the infile one bit at a time. Traverse the tree with respect to the bit read.

  If we have a bit 0, visit the left child, vice versa for a 1.

  We do the above until we reach a leaf node. Once we have a leaf node, write its symbol to the outfile and reset the current node to the root.

  We repeat this process until we have the original file size.

4 Priority Queue

Our Priority Queue in this program in particular is going to be utilizing a *min-heap*. The implementation for Priority Queue in our case is going to be very straight forward. Since we have already implemented a Heap-Sort algorithm, all we need to do is to change a few things in order to implement it for a min-heap. Granted we are *not* going to be running heap-sort every-time we need to fix the heap. In-fact we are not even going to be needing to run fix-heap every-time as-well. Knowing how a heap works will work to our benefit later when it comes to not only building and ordering the priority queue but also when we later need to turn it into a Huffman Tree. Since we are working in a min-heap we are going to want to have your values with the lowest frequency be dequeued first, which means they need to be essentially "last" in our priority queue. The reason a Priority Queue is going to be a much more efficient implementation rather

than a simple insertion sort method is because not only is it already in "tree" format, but the symbol with the smallest frequency is going to be at the very end of the queue, allowing us to grab it *very* quickly.

4.1 Functions For Priority Queue

PriorityQueue *pq_create(uint32_t capacity)
Constructor for the Priority Queue.
pq_delete(PriorityQueue **q)
Destructor for Priority Queue.
pq_empty(PriorityQueue *q)
Checks if the size of the queue is zero.
pq_full(PriorityQueue *q)
Checks is the size is equal to the capacity.
pq_size(PriorityQueue *q)
Returns the number of nodes in the queue.
enqueue(PriorityQueue *q, Node *n)
Enqueues a node into the queue. We can do this by inserting it at the end and backtracking until we find its position.
dequeue(PriorityQueue *q, Node **n)
Removes the item with the highest priority from the queue. This is the last item in the queue.
pq_print(PriorityQueue *q)
Prints the Priority Queue for debugging purposes..

5 Codes

When we are constructing our Huffman tree we are also working with a stack of bits in order to create a code for each symbol. This was discussed previously in §2.3. Our Code ADT is going to be implemented as a stack of bits with a 32 bit unsigned integer representing the top and an array of bits of MAX_CODE_SIZE (which is really just 256/8 bits, which turns out to be 32 bytes).

One very important thing to note when we are constructing our Code ADT is that it is going to be opn the *stack*, not the heap. This means that we will not be using any dynamic memory allocation like malloc in our Code ADT, so there is no need to free any memory. We will very simply be creating a stack, setting the top to zero, and zeroing our the bits array.

5.1 Functions For Code ADT

code_init(Code *c)
Create the Code struct.

code_size(Code *c)
Return the number of bits in Code.
code_empty(Code *c)
Check if the length of Code is zero.
code_full(Code *c)
Checks if the length of Code is 256 bits.
code_set_bit(Code *c, uint32_t i)
Sets the bit at index i to 1.
code_clr_bit(code *c, uint32_t i)
Clears the bit at i by setting it to 0.
code_get_bit(Code *c, uint32_t i)
Gets a bit at index i.
code_push_bit(Code *c, uint8_t bit)
Pushes a bit into the *Code*. The value is given by the parameterized bit.
code_pop_bit(Code *c, uint8_t *bit)
Pops a bit off the *Code*. The value is passed back into a pointer.
code_print(Code *c)
Prints the Code struct.

6 I/O

Instead of using the already buffered I/O functions we will be using the non-buffered syscall functions found in unistd.h and fcntl.h such as read(), write(), open(), close(). In oreder to read and write bytes we will also be using two seperate variables fro bytes_read and bytes_written.

When we are reading bytes from a file, we will be writing a wrapper function to loop through the file to read 4096 bytes, or 4KB, of data at a time until we have completed reading the file. We will put this data into a buffer.

When it comes to writing to a file, this process is going to operate very similarly to the reading operation specified above. We have to implement a loop to write out all the bytes in the buffer.

For reading a single *bit*, it is not so simple as just simply reading it directly. What we have to do is to read a *block* of *bytes* and use bit-wise operations in order to read a single bit at a time. We do this until we have cleared the buffer out of all the bits and we fill it back up for as many times as it takes to read all the bytes from the specified input file. The buffer for this function is the same as aforementioned, a block of 4096 bits which will be static to the file. The process for writing bits is going to be much the same as reading them.

6.1 Functions For I/O

int_read_bytes(int infile, uint8_t *buf, int nbytes)
This function in particular is going to be implemented like a *wrapper*. We are going to be reading a file 4096 bits at a time until we have either read all the bytes specified by *nbytes* or until all the bytes in the file have been read. The number of bytes read is returned from the function. We will be using the standard I/O functions from <unistd.h> read(), write() and <fcntl.h> open() and close().
write_bytes(int outfile, uint8_t *buf, int nbytes)
Very similarly to our read_bytes function above we are going to be using looping called to write all *nbytes* from the buffer 4096 bits at a time.
read_bit(int infile, uint8_t *bit)
Since we cannot individually read bits we must instead read them in using bytes and then to bit logic to get one bit at a time. We will be using a *static* buffer that we will fill with bytes and use an index denoted by a bit pointer to keep track of which bit to return. We can get individual bits by treating the buffer like a bit vector and using a combination of AND and SHIFT operations.
write_bit(int outfile, Code *c)
Similar to reading bits, we will be doing the same process for writing them out into our Code ADT. We will be doing bit logic since reading and writing individual *bits* is not as straightforward as writing bytes.
flush_codes(int outfile)
Since it is possible to have remaining bits in the buffer after using write_code(), this function aims to write out any leftover buffered bits,.

7 Huffman Coding Module

The main idea behind our Huffman Coding Module is to do most of the brunt work when it comes to setting things up for the encoding and decoding. The functions contained within our interface for the Huffman Coding Module are used in previous parts of the program, such as building the tree. and Given our histogram create from our I/O, we will be using this histogram to create our Huffman tree using a Priority Queue. The histogram in question will have ALPHABET indices, where each index is a possible symbol.