Assignment 7: A (Huffman Coding) Tree Grows in Santa Cruz

Shirin Rokni
shrokni

December 2, 2022

Citations

This PDF was generated using a LaTeXtemplate made by Jessie Srinivas (CSE13s tutor).
All pseudocode was provided by the assignment pdf, course staff (section leaders, tutors, TAs), or the professors lecture slides.

Purpose

This assignment creates two programs which are the huffman encoder and decoders. The encoder will read in an input file, finding the Huffman encoding of its contents, and use the encoding to compress the file. The decoder will read in a compressed input file and decompress it, expanding it back to its original, uncompressed size.

---

1 Test Harness

1.1 Building

To build the program, type the following in your terminal

```
make
```

To remove any compiler-generated files, type the following in your terminal

```
make clean
```

To remove all files that are compiler generated as well as the executable.

```
make spotless
```

1.2 encode command-line options

when running encode, you can use the following flag options

```
./encode [-hv] [-i infile] [-o outfile]
```

these are what the flags do

```
-h              Program usage and help.
-v              Print compression statistics.
-i infile       Input file to compress.
-o outfile      Output of compressed data.
```

1.3 decode command-line options

when running decode, you can use the following flag options

```
./decode [-hv] [-i infile] [-o outfile]
```

these are what the flags do

```
-h            Program usage and help.
-v            Print compression statistics.
-i infile     Input file to compress.
-o outfile    Output of compressed data.
```

---

2 Node

2.1 node.h

The header file for node.c. It allows other programs to access the functions within the node source code. It uses the standard library functions stdbool.h and stdint.h. stdbool.h is incredibly necessary as some functions defined in this file are boolean functions. stdint.h is necessary because there is a defined struct within the header file which has some variables with integer type names, and thus need the stdint.h library to declare it as such a variable.

2.2 node.c

purpose: An abstract data type. Huffman trees are composed of nodes, with each node containing a pointer to its left child, a pointer to its right child, a symbol, and the frequency of that symbol. Nodes are integral for the creation of huffman trees.

2.2.1 node_create

purpose: The constructor for a node.

Defining the node structure with mallocing the size. Then, the function initializes the left and right Nodes, the symbols and the frequency. The left and right node are initialized to 0 specifically. Finally, return a node.

2.2.2 node_delete

purpose: The deconstructor for a node.

Free the node pointer and then set it to NULL.

2.2.3 node_join

purpose: Joins a left child node and right child node, returning a pointer to a created parent node.

Creating a parent node type 'node' and passing in the arguments '$', and the sum of its left child frequency and right child frequency. Finally, return the parent node.

2.2.4 node_print

purpose: A debug function to verify that your nodes are created and joined properly.

First, check for if the passed in node is equal to NULL, and if its, print that. Then, check for if the node is a control variable or is not printable, in which the symbol will be uniquely printed. If not, then it will be directly printed as a character. Finally, the frequency is printed.

2.2.5 node_cmp

purpose: To compare the frequency of two nodes.

If the first node's frequency is greater than the second node's frequency, the function returns true, otherwise it returns false.

2.2.6 node_print_sym

purpose: To only print the symbol of the node.

The symbol of the node is printed.

---

3 Priority Queues

3.1 pq.h

The header file for pq.c. It allows other programs to access the functions within the pq source code. It uses the standard library functions stdbool.h and stdint.h, as well as calling the node.h header file in the local directory. stdbool.h is necessary as some functions defined in this file are boolean functions, the same logic is used for why stdint.h is necessary. node.h is used in various functions within the .c file, and thus is logically defined in the header file.

3.2 pq.c

purpose: An abstract data type. The encoder will make use of a priority queue of nodes. A priority queue functions like a regular queue, but assigns each of its elements a priority, such that elements with a high priority are dequeued before elements with a low priority.

3.2.1 Priority Queue struct

purpose: A way to group several related variables into one place.

The struct initializes the size and capacity uint32_t type variables and the inside pointer of a pointer to a Node.

3.2.2 pq_create

purpose: The constructor for a priority queue.

Defining the pq struct with mallocing the size. It also defines the variable 'inside' to the calloced size of capacity. It sets the 'size' variable to 0 and the 'capacity' variable to the argument passed into the function. It finally returns a priority queue.

3.2.3 pq_delete

purpose: The deconstructor for a priority queue.

If a pointer to a priority node exists, then the function frees it and sets it to NULL. It also frees the priority queue and sets it to NULL.

3.2.4 pq_empty

purpose: Returns true if the priority queue is empty and false otherwise.

If the 'size' variable equals 0, then that means that the priority queue is empty and the function returns true. Otherwise, it returns false.

3.2.5 pq_full

purpose: Returns true if the priority queue is full and false otherwise.

If the 'size' variable is equivalent to the 'capacity' variable, then that means that the priority queue has reached its capacity and returns true. Otherwise, the function returns false.

3.2.6 pq_size

purpose: Returns the number of items currently in the priority queue.

This function simply returns the value of the 'size' variable.

3.2.7 l_child

purpose: To return the left child in a heap sort.

Returns two times the passed in value plus one.

3.2.8 r_child

purpose: To return the right child in a heap sort.

Returns two times the passed in value plus two.

3.2.9 parent

purpose: To return the parent in a heap sort.

Returns the passed in value minus 1 divided by two.

3.2.10 up_heap

purpose: Sorts the path from insertion to root.

While the index is positive and the parent is smaller than the child, then the parent and child values swap and the index becomes the parent value.

3.2.11 down_heap

purpose: Sorts the path from the root, checking both children for greater than the parent.

While the left child is less than the size of the priority queue, the following occurs. If there is no right child, then the larger value is the left child. Otherwise, if the left child is greater than the right child, then the larger value is the left child, otherwise the larger value is the right child. If the current node is less than the larger node, then break because that is the correct ordering. Otherwise, the two nodes swap and the node becomes the larger variable.

3.2.12 enqueue

purpose: Enqueues a node into the priority queue.

If the priority queue is full prior to enqueing the node, the function returns false (can be detected using the pq_full function).

The passed in node becomes the last node in the priority queue. Then, up_heap is called and the priority queue is properly fixed. Then, the size variable will be incremented to account for the new node and 'true' is returned.

3.2.13 dequeue

purpose: Dequeues a node from the priority queue, passing it back through the double pointer n. The node dequeued should have the highest priority over all the nodes in the priority queue.

If the priority queue is empty prior to dequeuing a node, the function returns false. Otherwise the passed in double pointer n becomes the highest priority queue (the first node in the priority queue). Then, the code sets the node from the tail pointer to the front of the node. Next, the size variable is decremented and down_heap is called in order to fix the priority queue. Finally, the function returns true.

3.2.14 pq_print

purpose: A debug function to print a priority queue.

It prints the size of the priority queue just for clarity. Then, it loops until the size of the priority queue is reached, printing each of the nodes with node_print.

---

4 Code

4.1 code.h

The header file for code.c. It allows other programs to access the functions within the code source code. It uses the standard library functions stdbool.h, stdint.h, and defines.h. stdbool.h is necessary as some functions defined in this file are boolean functions. stdint.h is necessary because there is a defined struct within the header file which has some variables with integer type names, and thus need the stdint.h library to declare it as such a variable. defines.h is used in various functions within the .c file, and thus is logically defined in the header file.

4.2 code.c

purpose: An abstract data type. A way to maintain a stack of bits wile traversing the tree in order to create a code for each symbol.

4.2.1 code_init

purpose: The constructor for a code.

Set the top variable to 0. Then, iterate through the array bits and set them to 0. Finally, return code.

4.2.2 code_size

purpose: Returns the size of the Code, which is exactly the number of bits pushed onto the code.

Simply return the 'top' variable.

4.2.3 code_empty

purpose: Returns true if the Code is empty, and false otherwise.

If the code_size of the passed in Code type variable is equivalent to 0, then the Code is empty and the function returns true. Otherwise, it returns false.

4.2.4 code_full

purpose: Returns true if the Code is true and false otherwise.

If the code_size of the passed in Code type variable is equivalent to ALPAHBET, then the Code is empty and the function returns true. Otherwise, it returns false.

4.2.5 code_set_bit

purpose: Sets the bit at index i in the Code, setting it to 1. If i is out of range, return false.

If the passed in index is positive and less than the ALPHABET value, then it sets the bit index i to 1 and the function returns true. Otherwise, it returns false.

4.2.6 code_clr_bit

purpose: Clears the bit at index i in the Code, clearing it to 0. If i is out of range, return false.

If the passed in index is positive and less than the ALPHABET value, then it sets the bit index i to 0 and the function returns true. Otherwise, it returns false.

4.2.7 code_get_bit

purpose: Gets the bit at index i in the Code. If i is out of range, or if bit i is equal to 0, return false.

If the passed in index is positive and less than the ALPHABET value, then the function checks for if the bit at that index exists by equaling it to 1, and the function returns true. Otherwise, it returns false.

4.2.8 code_push_bit

purpose: Pushes a bit onto the Code. The value of the bit to push is given by bit.

If the Code is full prior to pushing it, then the function returns false (can be checked with code_full). Otherwise, if the bit is equivalent to 1, then the function calls code_set_bit. Else, it calls code_clr_bit. Finally, the top variable is incremented and the function reurns true.

4.2.9 code_pop_bit

purpose: Pops a bit off the Code. The value of the popped bit is passed back with the pointer bit.

If the Code is empty prior to pushing it, then the function returns false (can be checked with code_empty).Otherwise, the function pops the bit from the stack to the pointer with functions like code_get_bit and code_clr_bit. It also decrements the top variable. Finally, the function returns true if it does find the bit, otherwise it returns false.

4.2.10 code_print

purpose: A debug function to help you verify whether or not bits are pushed onto and popped off a Code correctly.

The function returns the top variable.

---

5 I/O

5.1 io.h

The header file for io.c. It allows other programs to access the functions within the io source code. It uses the standard library functions stdbool.h, stdint.h, and code.h. stdbool.h is necessary as some functions defined in this file are boolean functions. stdint.h is necessary because there is a defined struct within the header file which has some variables with integer type names, and thus need the stdint.h library to declare it as such a variable. code.h is used in various functions within the .c file, and thus is logically defined in the header file.

5.2 io.c

purpose: An abstract data type. Functions defined by the following I/O module will be used by both the encoder and decoder.

5.2.1 read_bytes

purpose: This will be a useful wrapper function to perform reads.

While the total read data is not equal to the number of bytes that are meant to be read, the read() function is called that passes in the buffer, the infile and the nbytes. Finally, the bytes_read variable which tracks the bytes_read statistic is equaled to the number of totally read bytes and that value is returned from the function.

5.2.2 write_bytes

purpose: This functions is very much the same as read_bytes(), except that it is for looping calls to write().

While the total written data is not equal to the number of bytes that are meant to be written, the write() function is called that passes in the buffer, the infile and the nbytes. Finally, the bytes_written variable which tracks the bytes_written statistic is equaled to the number of totally write bytes and that value is returned from the function.

5.2.3 read_bit

purpose: Reading in a block of bytes into a buffer and dole out bits one at a time.

The function first initializes the bytes, buffer and 'i' variable which will act as a local index. It then checks for if i is equal to 0, in which then call read_bytes and equal that value to the bytes variable. It will also multiply that value by 8 for the total byte that is read. Then, the function passes the bit back through *bit using the get_bit function. Then, it increases the local index by 1 and mod by 8. If the index is less than the read_bytes value, then the function returns true, otherwise it returns false.

5.2.4 flush_codes

purpose: The sole purpose of this function is to write out any leftover, buffered bits

The function writes any leftover bits with the write_bytes function. This can be done by first dividing index by 8 and equaling it to a variable called 'flush'. The, if index mod 8 right shifted by one and subtracted by one exists, then the array buffer at the index of an incremented flush is equaled to itself and the if check. Finally, write_bytes is called with the arguments outfile, array_buffer and flush.

5.2.5 write_code

purpose: When the buffer of BLOCK bytes is filled with bits, write the contents of the buffer to outfile

The function first initializes a total_size variable which will be equivalent to the size of the passed in Code type variable. The size will be determined with code_size. Then, the function loops through the values until it reaches the total_size value. Next, an each_bit variable is created which is equivalent to the code_get_bit function. If 'each_bit' is equal to 1, then the function sets that bit with the contents of the code_set_bit function and increments the index. Otherwise, it clears the bit with the contents of the code_clr_bit function but also increments the index. If the index value is equivalent to the BLOCK value times 8, then write_bytes is called and the index is set to 0.

---

6 Stacks

6.1 stack.h

The header file for stack.c. It allows other programs to access the functions within the stack source code. It uses the standard library functions stdbool.h, stdint.h, and node.h. stdbool.h is necessary as some functions defined in this file are boolean functions. stdint.h is necessary because there is a defined struct within the header file which has some variables with integer type names, and thus need the stdint.h library to declare it as such a variable. node.h is used in various functions within the .c file, and thus is logically defined in the header file.

6.2 stack.c

purpose: A stack of nodes is necessary in the decoder to reconstruct a Huffman tree.

6.2.1 Stack Struct

purpose: A way to group several related variables into one place.

The struct initializes the 'top' and 'capacity' variables which are both uint32_t type variables. It also initializes the node type pointer called 'items'.

6.2.2 stack_create

purpose: The constructor for a stack.

Defining the stack struct with mallocing the size. It also defines the variable 'items' to the calloced size of capacity. It sets the 'top' variable to 0 and the 'capacity; variable to the argument passed into the function. It finally returns a stack.

6.2.3 stack_delete

purpose: The destructor for a stack.

If the pointer to a stack exists, then the function frees it and sets it to NULL. If the stack exists, then the function frees it and sets it to NULL.

6.2.4 stack_empty

purpose: Returns true if the stack is empty and false otherwise.

If the 'top' variable equals 0, then that means that the stack is empty and the function returns true. Otherwise, it returns false.

6.2.5 stack_full

purpose: Returns true if the stack is full and false otherwise.

If the 'top' variable is equivalent to the 'capacity' variable, then that means that the stack has reached its capacity and returns true. Otherwise, the function returns false.

6.2.6 stack_size

purpose: Returns the number of nodes in the stack.

The function simply returns the value of the 'top' variable.

6.2.7 stack_push

purpose: Pushes a node onto the stack.

When the stack_full function is called and if that output equals to false, then a node is pushed to stack by making the top of the stack equal to the passed in node and the top variable incremented. The function also returns true. Otherwise, that means that the stack is full of nodes and the function returns false.

6.2.8 stack_pop

purpose: Pops a node off the stack, passing it back through the double pointer n.

When the stack_empty function is called and if that output equals to false, then a node is popped to stack by making the top of the stack equal to the passed in node after the top variable is decremented. The function also returns true. Otherwise, that means that the stack is empty and the function returns false.

6.2.9 stack_print

purpose: A debug function to print the contents of a stack

The function loops until the top of the stack is reached, in which all of the items of the stack are printed with node_print.

---

7 Huffman

7.1 huffman.h

The header file for huffman.c. It allows other programs to access the function within the huffman source code. It uses the standard library function stdint.h. It also calls the node.h, code.h and defines.h header files which are located in the local directory. These header files are used in various functions within the huffman.c file, and is thus logically defined in the huffman.h header file.

7.2 huffman.c

purpose: An abstract data type. An interface for a Huffman coding module.

7.2.1 build_tree

purpose: Constructs a Huffman tree given a computed histogram.

First, the function initializes a priority queue. Then, it initializes five different node type variables: left, right, parent, 'node' and output. The, it iterates through the Alphabet, and if the histogram's index at that point is not equal to 0 (meaning that it is in the histogram), then a node is created with node_create with the index and hist[i] as the arguments. The function also enqueues the node into the priority queue.

Outside of the for-loop, there is a while loop which checks for if the size of the priority queue is not equal to 1, in which the left and right nodes are dequeued and the parent node becomes the node_joined left and right node. Then, the parent node is also enqueued. Then, the output node is also dequeued in order to get the last node of the priority queue and the priority queue is deleted. Finally, the output node is returned.

7.2.2 build_codes

purpose: Populates a code table, building the code for each symbols in the Huffman tree.

Before this function is initialized, a variable 'c' is initialized as a Code type so that it isn't initialized every iteration. A bool is also set that is equal to false. In the function, if the bool is false, then code_init() is called and the bool is set to true so that code_init() is not repeatedly called.

Then, if the root argument is not null, then it does the following. If the left and right nodes of the root are equal to 0, then the table with the symbol as the index are equal to c. Otherwise, the following occurs. The function uses code_push_bit with the arguments of the code pointer and 0. Then, build_codes is called again with the arguments being the left node and the table. Next, code_pop_bit is called with the code pointer and a bit variable.

The function then uses code_push_bit with the arguments of the code pointer and 1. Then, build_codes is called again with the arguments being the right node and the table. Next, code_pop_bit is called with the code pointer and a bit variable.

7.3 dump_tree

purpose: Conducts a post-order traversal of the Huffman tree rooted at root, writing it to outfile.

First, the function makes variables that are equivalent to the characters 'L' and 'I'. Then, the function checks for if the root exists, and if so to call dump_tree again on the left and right roots. Then, if the left and right roots are equal to 0, then the function calls write_bytes with the arguments being outfile, the L variable and 1, and the outfile, the symbol and 1. Otherwise, the function calls write_bytes with the arguments outfile, the I variable and 1.

7.3.1 rebuild_tree

purpose: Reconstructs a Huffman tree given its post-order tree dump stored in the array tree_dump

First the function initializes a stack type variable 'stack' with stack_create. Then, it initializes four different node type variables: left, right parent and 'node'. Then, it iterates through the nbytes value. If the tree with the index i is equal to the 'L' character, then the index is incremented, and node is equaled to node_create with the arguments tree[i] and 0. Also, stack_push is called with the arguments stack and node. Otherwise, the left and right nodes are popped with stack_pop. The parent node is equaled to the joined of the left and right nodes and the parent is pushed. Finally, the output node is popped with stack_pop to get the rebuilt huffman tree, the stack is deleted and output node is returned.

7.3.2 delete_tree

purpose The destructor for a Huffman tree

If the root exists, then delete_tree is called with the left and right nodes. Finally, the root is deleted and the pointer is set to NULL.

---

8 Encoder

purpose: This file contains the main program and will be responsible for encoding a file into a compressed version by utilizing the Huffman tree.

8.1 encode.c

First, Main is called with the arguments argc and aargv. Then, the infile and outfile variables are made with calls to STDIN_FILENO AND STDOUT_FILENO. Additionally, opt and verbose variables are set to 0. Then, the function creates a temporary file for stdin, and this step is crucial so that when a user uses the STDIN_FILENO command-line input, then the function can accurately be encoded.

Then, a while loop is initialized which will go through all of the command-line flags. If the 'h' flag is called, then the helper function is printed to stderr and the function returns 0. If the 'i' flag is called, then the infile variable is set to the output of the open() function with the arguments optarg and O_RDONLY. If the 'o' flag is called, then the outfile variable is set to the output of the open() function with the arguments O_WRONLY — O_CREAT — O_TRUNC. If the 'v' flag is called, then the verbose variable is set to 1. If an incorrect flag is used, then the default function is called which prints the help function again to stderr but returns a nonzero value.

Now, the function creates a buffer size 4096, a code table size 256, and a histogram size 256. Additionally, a characters variable is initialized which will count all of the special symbols.

Now, a while-loop is initiated which goes on for as long as the read_bytes function returns a value greater than 0. In the while-loop, the function calls write_bytes with the temporary file as the argument. Then, the function goes through a for-loop in which the histogram buffer value, if equal to 0 (meaning that the character has not yet been put into the buffer), the characters variable is incremented. Then, the size of the buffer is incremented.

Then, the infile is set to the temporary file and lseek is called.

Now, the function builds the code table from the tree. What this means is, that a node 'tree' is created with the function build_tree and the argument histogram. Then, the build_codes function is called with the arguments tree and code.

Then, the function sets up the header. The header is set to the name 'perm' and then calls fstat and fchmod. fstat has the arguments infile and perm, while fchmod has the arguments infile and perm.st_mode and outfile and perm.st_mode. Next, each of the header arguments are set. So, header.magic equals the MAGIC variable. header.permissions equals the perm.st_mode variable. header.tree_size equals the (3 * characters) - 1 value. Finally, header.file_size equals perm.st_size.

Then, write_bytes is called with the header being written to the outfile.

Next, dump tree is called which will dump the tree to the outfile. Then, starting at the beginning of infile, the function writes the corresponding code for each symbol to outfile with lseek.

Next, the function reads in the data from the input again and prints the appropriate code out from the code table. This can be done with write_code. Finally, all of the code remaining in the buffer is flushed out with flush_codes.

Finally, the verbose statement is called. It first prints the uncompressed file size which is simply perm.st_size. Then, it prints the compressed file size which is the statistic bytes_written. Finally, it prints the spaced saved which is 100 * (1 - bytes_written / perm.st_size).

The function concludes by deleting the tree and then closing the files and returning 0.

---

9 Decoder

purpose: This file will decode the encoded message from encode.c back into the original message.

9.1 decode.c

First, Main is called with the arguments argc and aargv. Then, the infile and outfile variables are made with calls to STDIN_FILENO AND STDOUT_FILENO. Additionally, opt and verbose variables are set to 0.

Then, a while loop is initialized which will go through all of the command-line flags. If the 'h' flag is called, then the helper function is printed to stderr and the function returns 0. If the 'i' flag is called, then the infile variable is set to the output of the open() function with the arguments optarg and O_RDONLY. If the 'o' flag is called, then the outfile variable is set to the output of the open() function with the arguments O_WRONLY — O_CREAT — O_TRUNC. If the 'v' flag is called, then the verbose variable is set to 1. If an incorrect flag is used, then the default function is called which prints the help function again to stderr but returns a nonzero value.

Then, the header is initialized and the function will read in the header using read_bytes with the arguments infile, the bit masked uint8_t value of the header and the size of the header.

Next, a struct called 'perm' is initialized. fstat is called with the arguments infile and perm and the fchmod is called to get permissions for the outfile/

Then, the function checks for if the magic number matches and if not, prints an error message and returns -1.

Next, the function makes the tree the size of the header tree size and then reads in the dumped tree with read_bytes. Then, a node type called 'tree' will be the output of rebuild_tree with the arguments head.tree_size and sz_tree.

Then, two nodes 'node' and 'root' are created which will be swapped in following functions. Then, a while loop is initialized which will go on while the file is still being read and read_bit does not output a false value. If the node right and left node's do not exist, then write_bytes is called and the roots will be swapped and a counter will be incremented. If the read_byte equals one, then the function goes down the right node, otherwise it goes down the left.

Finally, the verbose statement is called. It first prints the compressed file size which is simply bytes_read. Then, it prints the decompressed file size which is the statistic bytes_written. Finally, it prints the spaced saved which is 100 * (1 - bytes_read / bytes_written).

The function concludes by deleting the tree and then closing the files and returning 0.