Assignment 8 – Huffman Coding

Adan Shafei

CSE 13S – Fall 2023

Purpose

Huffman Coding is a popular data compression technique that works by encoding symbols in a file with variable-length codes based on their frequency of occurrence. By doing this, Huffman Coding can significantly reduce the size of a file without sacrificing any of its content. In this program, I will implement two main components - a data compressor (huff) and a data decompressor (dehuff) - each supported by four additional modules: a bit writer, a bit reader, a binary tree, and a priority queue. These modules are responsible for various tasks, such as reading and writing individual bits, creating and manipulating binary trees and priority queues, and constructing the Huffman tree for compression and decompression purposes. With these components working together, this program aims to provide a robust and efficient implementation of Huffman Coding for data compression and decompression.

How to Use the Program

To run any program, we need to make sure that the program compiles with no errors and to do so, I created a Makefile that takes all the header and the helper functions files and compiles them, creating an objective file for each so they could be used when running the main file. Then, it takes the main file, compiles it to make sure there are no syntax errors, and finally creates an executable file for the user that it could run.
To run the Makefile, the user will use that command line, starting with **make format** to fix all the formatting for all the files. The user runs **make clean** to remove any objective files that were created before to avoid any errors, and finally, runs **make** to compile all the files and create an executive file to use.

After compiling the program using Makefile, it creates an executable program for the user to run. The executable file handles multiple command line options to have the user specify how the program should run and produce the output. Those command line options are: -i to read from the input file, -o to write the output in a file, and -h to print the help message.

To use the huffing program, the user will run this command on the command line **./huff -i infile -o outfile** . Using this command will specify which input file to read from and which output file to print to. The same thing to the dehuffing program, the user will run the command **./dehuff -i infile -o outfile** .
If the user was not sure about the command line options and wants to check how to use the program, the user should run the command **./huff -h** to print the following help message:

```
Usage: huff -i infile -o outfile
       huff -h
```

If the user was not sure how to use the dehuff program, the user could run this command **./dehuff -h** to print the following help message:

```
Usage: dehuff -i infile -o outfile
       dehuff -h
```

Program Design

The design of this program contains multiple parts to satisfy the purposes of Huff and Dehuff. The parts of my program are:

1. Data Compressor (huff): This program reads a file, computes the frequency of each character in the file, and uses these frequencies to construct a Huffman tree. It then traverses the tree to determine the Huffman code for each character. The Huffman codes are used to compress the data, and the compressed data is written to a binary file along with the Huffman tree
2. Data Decompressor (dehuff): This program reads the binary file produced by the compressor. It reconstructs the Huffman tree from the file and uses it to decode the Huffman codes back into the original characters.
3. Bit Writer: provides functions for writing individual bits and sequences of bits to a binary file
4. Bit Reader: provides functions for reading individual bits and sequences of bits from a binary file.
5. Binary Tree: provides functions for creating and manipulating binary trees that represent the Huffman codes.
6. Priority Queue: provides functions for creating and manipulating a priority queue, which is used to construct the Huffman tree.

Each component should be implemented as a separate C file with a corresponding header file. The `main` function in the `huff` and `dehuff` programs should handle command-line arguments, call the appropriate functions to perform the compression or decompression, and handle any errors that occur.

Data Structures

In this program, several data structures were defined to facilitate the implementation of the Huffman coding algorithm. Here is a brief description of each:

- BitWriter: This structure is used to write binary data to a file one bit at a time. It includes a pointer to the underlying file stream, a byte buffer to store the bits, and a bit position counter to keep track of the current bit position within the byte.
- BitReader: Similar to BitWriter, this structure is used to read binary data from a file one bit at a time. It also includes a pointer to the underlying file stream, a byte buffer to store the bits, and a bit position counter.
- Node: This structure represents a node in a binary tree, which is used to create the Huffman code tree. Each node includes a symbol (the byte value), a weight (the frequency of the symbol), a code (the Huffman code for the symbol), a code length, and pointers to the left and right children.
- ListElement: This structure is used to create a linked list of nodes. Each element in the list includes a pointer to a node (the tree) and a pointer to the next element in the list.
- PriorityQueue: This structure represents a priority queue, which is used to order the nodes based on their weights when creating the Huffman code tree. The priority queue includes a pointer to the head of a linked list of nodes.

These data structures provide the necessary foundation for implementing the Huffman coding algorithm, which is used for lossless data compression.

Function Descriptions

bitwriter.c file

bit_write_open(): function that opens binary file for write using fopen() and return a pointer

```
allocate a new BitWriter
open the filename for writing as a binary file
if the file was not open
    return NULL
store f in the BitWriter field underlying_stream
clear the byte and bit_positions fields of the BitWriter to 0
```

bit_write_close(): a function that closes the binary file for write using fclose()

```
flush any remaining bits in the byte buffer to the underlying stream
close the underlying stream
free the BitWriter
set the *pbuf pointer to NULL
```

bit_write_bit(): a function that writes a single bit

```
if the buffer is in range (0 to 8)
    flush the byte buffer to the underlying stream
        handle error: Could not write a bit
    clear the byte buffer and reset the bit position
set the bit_position bit of the byte to x
increment the bit position
```

bit_write_uint8(): a function that writes 8 bits at a time

```
if iterating over the range from 0 to 7
    check if the LSB of x is 1
    shift x right by 1 bit, effectively discarding the LSB of x
```

bit_write_uint16(): a function that writes 8 bits at a time

```
if iterating over the range from 0 to 15
    check if the LSB of x is 1
    shift x right by 1 bit, effectively discarding the LSB of x
```

bit_write_uint32(): a function that writes 32 bits at a time

```
if iterating over the range from 0 to 31
    check if the LSB of x is 1
    shift x right by 1 bit, effectively discarding the LSB of x
```

bitreader.c file

bit_read_open(): function that opens binary file for reading fopen() and return a pointer

```
allocate a new BitWriter
checking for error, if there's an error, return NULL
open the filename for writing as a binary file
if the file was not open
    free the memory
    return NULL
store f in the BitWriter field underlying_stream
clear the byte and bit_positions fields of the BitWriter to 0
```

bit_read_close(): a function that closes the binary file for write using fclose()

```
flush any remaining bits in the byte buffer to the underlying stream
close the underlying stream
free the BitWriter
set the *pbuf pointer to NULL
```

bit_read_bit(): a function that reads a single bit

```
if the buffer is in range (0 to 8)
   flush the byte buffer to the underlying stream
      handle error: Could not read the byte
   clear the byte buffer and reset the bit position
set the bit_position bit of the byte to x
increment the bit position
```

bit_read_uint8(): a function that reads 8 bits at a time

```
if iterating over the range from 0 to 7
   check if the LSB of x is 1
   shift x right by 1 bit, effectively discarding the LSB of x
```

bit_read_uint16(): a function that reads 16 bits at a time

```
if iterating over the range from 0 to 15
   check if the LSB of x is 1
   shift x right by 1 bit, effectively discarding the LSB of x
```

bit_read_uint32(): a function that reads 32 bits at a time

```
if iterating over the range from 0 to 7
   check if the LSB of x is 1
   shift x right by 1 bit, effectively discarding the LSB of x
```

Algorithms

node.c file: This file contains data structures that represent a binary tree. Each Node contains a symbol, a weight, a code, a code length, and pointers to left and right children. The Node functions provide functions to create free nodes and print trees.
This file includes three functions which are: node_create(): a function that creates a Node and sets its symbol and weight fields. Return a pointer to the new node. On error, return NULL. A function that creates the node.

```
allocate memory for the new node
assign the given symbol to the new node
assign the given weight to the new node
initialize the code value to 0
initialize the code length to 0
initialize the left child to NULL
initialize the right child to NULL
return the created node
```

node_free(): a function that frees the node and sets it to NULL.

```
check if node is not NULL
   using recursion
   free left node
   free right node
   free the memory allocated
   set the pointer to NULL
```

node_print_node() : a function that prints the node of a tree

```
if tree is empty return
using recursion to print right subtree
print node info
if node is a leaf
```

```
    print ASCII symbol
else
    print non-ASCII symbol
using recursion to print left subtree
```

node_print_tree(): a function that prints the tree in the way that I defined it to print it.

```
calling the function node_print_node to print every node of the tree
```

pq.c file: This file contains data structures that store pointers to trees based on their weights. The Priority Queue is implemented using a linked list, where the tree with the lowest weight is at the head9. The Priority Queue functions provide functions to create, free, enqueue, dequeue, and print the queue.

This file includes multiple functions I will use to build the priority queue. Those functions are:
pq_create(): a function that allocates a priority queue object and returns a pointer.

```
allocating the priority queue in the memory
if the priority queue is NULL the function returns NULL
initialize the list to NULL
function returns pq
```

pq_free(): a function that frees the priority queue

```
if the priority queue is NULL
    function returns
iterate over the priority queue and free each node
freeing the priority queue
    freeing the nodes
freeing the priority queue
setting priority queue to NULL
```

pq_is_empty(): a function that returns true if the priority queue is empty

```
check if the priority queue is NULL or the list is empty
    function returns true
if not function returns a false
```

pq_size_is_1(): a function that returns true if the priority queue has one element

```
check if priority queue is not NULL and the next item it NULL
```

pq_less_than(): a function that checks if the first tree is less than the second tree

```
// based on asgn8.pdf
compare the weights of the trees
if the first weight is less
    function returns true
if weights are equal, compare the symbols
    function returns the comparison result
if first tree is not less than the second tree
    function returns false
```

enqueue(): a function that inserts a tree into the priority queue

```
// based on asgn8.pdf
allocate a new ListElement
    if allocation fails return
set the tree field to the value of the tree function parameter
if the queue is empty
    add the new element
checking if the new element is less than the tree
    new element goes before all existing elements of the list
```

```
else
   find the existing element where the new element should be placed after
insert new element after element 2
```

dequeue(): a function that removes the queue element with the lowest weight and returns it

```
// based on asgn8.pdf
check if the queue is empty
   printing error message and return
get the front element of the queue
update the queue to remove the front element
free the ListElement (but not the tree itself)
return the dequeued tree
```

pq_print(): a function that prints the trees of the queue

```
// based on asgn8.pdf
check if the priority queue is not NULL
start from the beginning of the list
variable to keep track of the position
iterate over the list and print each tree
   print a separator based on the position
   print the tree using a helper function
   move to the next element in the list
print a final separator at the end
```

Huffman Code

Huffing Program: huff.c

fill_histogram(): a function that fills the histogram

```
clearing all elements of the histogram array
at least 2 values of the histogram are not zero
initialize the total size of the file
getting the file name
using a while loop to read bytes from the input file
    updating the histogram
    increment histogram
    increment filesize
rewind the input file to the beginning
return the file size
```

create_tree(): a function that creates the tree

```
create priority queue
iterate over the priority queue to fill the histogram
running the Huffman Coding algorithm
using a while loop if the size of the pq is not 1
   dequeue left
   dequeue right
   create a new node: weight = left->weight + right->weight and symbol = 0
   assigning the left and right of the new node
   enqueue new node
dequeue the queue's only entry and return it
free priority queue
incrementing the number of leaves
returning the tree
```

fill_code_table(): a function that fills the code table

```
if node is null
    the function returns
if the left and right are NULL
    store the Huffman Code for left node
    and return
recursive calls for left and right nodes
appending 1 to code and recurse
```

huff_write_tree(): a function that prints the Huffman Code

```
if the left node is NULL
    writing the leaf node
if not then
    writing the internal node
    writing the left node
    writing the right node
    writing the out buffer
```

huff_compress_file(): a function that compresses the file

```
writing 'H' and 'C' as magic number
writing filesize
writing number of leaves
writing Huffman Tree
rewind the input file to the beginning
initializing the input file
using a while loop to read the input file and write Huffman codes
    retrieve the code for the read character from code_table
    retrieve the code_length for the read character from code_table
        if so, break out of the loop
    retrieve the code for the read character from code_table
    retrieve the code_length for the read character from code_table
    a for loop that runs code_length times
        write the least significant bit of code to output
        right shift the bits of code by 1
```

print_help(): a function that prints the usage message

MAIN FUNCTION:

```
defining option to use in getopt()
defining a file to scan the input file
defining a variable for the output file name
checking the input and output files are provided
    if no input file print error and exit
    if no output file print error and exit
while the user provides an option
    checking the options that were provided (using switch)
    if the option was 'h' print the help message
    if the option was 'i' use the input file
         opening the file with r
         check that finis not NULL
         print error
         print help message
         exit the program
    if the option was 'o' print the output into this file
        open output file
        if the output file is null
        printing error to open the output file
        closing output file
```

```
       exiting the code
       the default case is to break
creating a histogram
initializing the file size and filling the histogram
initializing the number of leaves
creating the tree
if the tree is NULL
    print error and help message
    closing input file
allocating the table
filling the table
compressing the file and printing result to output file
freeing the table
freeing the node
closing input file
making input file null
closing output file
```

Dehuffing Program: dehuff.c

stack_push(): a function that pushes nodes to the stack

```
check if stack has space left
    push node to stack
if not
    stack overflow error - couldn't push to stack
```

stack_pop(): a function that pops nodes from the stack

```
check if stack is is greater ot equal to 0
    pop node from stack
if not
    stack underflow error - couldn't pop from stack
```

dehuff_decompress_file(): function to perform Huffman decoding and write the decompressed data to the output file

```
using the bit read functions to read header information
validate the read information
using the assert functions to check is the header information correct
calculate the number of nodes in the Huffman tree
initialize a stack for building the Huffman tree
initialize top of the stack
iterating over the number of nodes to build the tree for huffman
    read one bit to determine node type
    create a new node based on the bit value
    if bit is 1
       creating the leaf of the node using symbol
    if not
       creating an internal node
       popping the right side
       popping the left side
    push the new node onto the stack
the top of the stack now contains the root of the huffman tree
iterating over the file size to decode the compressed data
    traverse the Huffman tree until a leaf node is reached
    move to the left or right child based on the bit value
    if bit is 0
       move node to the left
    if not
       move node to the right
```

```
if it’s a leaf node, break from the loop
write the symbol of the leaf node to the output file
freeing the memory used for creating the node
```

print_help(): a function that prints the usage message

MAIN FUNCTION

```
defining option to use in getopt()
defining a file to write the output in
definig a variable to take the input file
checking the input and output files are provided
while the user provides an option
    checking the options that were provided (using switch)
    if the option was ’h’ print the help message
    if the option was ’i’ use the input file
    if the option was ’o’ print the output into this file
        taking the output file from the command line (using w to write in the file)
        check that fout is not NULL
            print error
            print help
            exiting the program
    the default case it to break
opening the input file to read it
using the decompressing function to decode the input
closing the input file
closing the output file
```

Error Handling

This program handles errors because it requires opening files, reading from them, and writing to them. If the user did not provide an input file for the program to open and read from, an error will occur, and the program will print the following message **huff: -i option is required** followed by the usage message, and the same thing for the **dehuff** program. Similarly, the program will display an error message **huff: error reading input file** and terminate if there is an error while writing to the file.

If an error occurs while attempting to open the output file for writing, the program will display an error message **huff: -o option is required** followed by the usage message to the user and exit. Similarly, the program will display an error message **huff: error writing to the output file** and terminate if there is an error while writing to the file. This ensures that the program handles errors effectively and provides clear feedback to the user. The **dehuff** program handles errors like the huff program handles them for the output file errors too.

Testing

For testing my code, I will use multiple testing files from Professor Veenstra to test each of the four files I created. For the bit writer and bit reader, there are bwtest.c and brtest.c files that test my code.
Also, for the algorithms used in my program, which are nodes and priority queues, there are testing files that were provided by Professor Veenstra. Those files are nodetest.c and pqtest.c, they will test my code.

To test my programs, I made sure to include the testing files in my Makefile as the executable files that I have, this will compile my files in addition to compiling the test files and ensure that all the files compile correctly based on the compiling commands in the Makefile. After adjusting the Makefile to execute the test files, I run the compiling commands that I normally use to compile; **make format**, **make clean**, and **make**.

To run the testing files, I used a shell script provided by Professor Veensrta, to run the shell script I run **./runtests.sh**, and that will run the test files that will test the functionality of all the files. Running those test files will ensure that all my files work properly and code is called in the main program to create the huff and dehuff program. If no errors occur, my programs passed the tests, and they could be used in the main programs of huffing and dehuffing.

In addition to using the testing files, I will use the valgrind command to ensure no memory leaks in my program. When I read and write to files, I use the memory, and to ensure that I don't have any memory leaks in my program, I use valgrind to check for memory leaks.

Results

Wednesday, December 6, 2023 So far, I have done all the functions and the .c files that I needed to create, include, and use in both huff and dehuff programs. The bit writer, bit reader, node, and priority queue all passed the test files that were provided by Professor Veenstra. For my huff.c I finished writing all the functions, and those passed the test, too, I am currently working on my main function for the huff coding and aiming to finish that by the end of the day.
For my dehuff program, I also coded the sudo code of the function that was provided in the assignment's pdf. I have to write a main function that I know will be relatively similar to the huff main function since both programs generally should work in a similar flow. Once I finish the main function, I will run the test file for my dehuff file and ensure that it works correctly.
During the process of coding the sudo code, I faced some minor bugs that were caused because of incorrect placement or order of code. There was nothing major that consumed a lot of time to fix. After coding the sudo code or each function in one of the files, I compiled the file and ran the test files to ensure that what I was coding worked correctly. When I started working on the huff and dehuff files, I first ran the reference files that I have in the resources to visually see how the programs work and what the output should look like. I also tested a couple of errors to see how to handle the errors.

Sunday, December 10, 2023 After I finished all the code huff program and dehuff programs compiled correctly with no errors. After running the Makefile, compiling all the files, and creating executable files for huff and dehuff, I can run both programs to either encode or decode the files that I provide the program with.
To run the Huffman coding program, I run the command **./huff -i input-file -o output-file**, the program will read the input file, encode the content of the input file by compressing the data that's in it, and print the output in the output file. The results of the compression that will be printed in the output file are going to be binary output.
To run the Huffman decoding program, I run the command **./dehuff -i input-file -o output-file**, the program will read the input file, decode the content of the input file by decompressing the data that's in it, and print the output in the output file. The results of the decompression that will be printed in the output file are going to be text output.
If there is no input file, an error will occur, and the program will not be able to read the input from the file, in this case, the program will print that it needs an input file and the usage message and then exit. The same thing applies to the output file. If it is not provided, the program will print that it requires an output file, print the usage message, and then exit.