Assignment 8 – Huffman Coding

Asaveri Rao

CSE 13S – Fall 2023

Purpose

The main purpose is to get familiar with Huffman coding, where it compresses a data file. In its core is is supposed to make the entire compression process much more efficient and more concise. After the bytes are taken into consideration in the input file, the most common symbols are taken and converted to less bits. The opposite logic is utilized for less common symbols. The Huffman coding and Dehuffman coding are utilized. As seen in the same, Huffman is to compress and Dehuffman is to decompress.

How to Use the Program

This program consists of four header files: bitreader.h, bitwriter.h, node.h, pq.h. To correspond with these files, there are four c files, bitreader.c, bitwriter.c, node.c, and pq.c. To actually run the program the user has to run the four command line options to see the correct output and if the program is running. The four command line options are -i, -o, -h, -v. There are also a few test case files to run, they can be separately called, however there is also a shell script file that contains all the test cases to run called runtests.h.One of the major components of this program to work is the Makefile. To properly compile the program, the programmer has to run make format, make clean, make and if it compiles without producing no errors that means it successfully compiled.

Program Design

There are a few different working components to this program. There are four different header files: bitreader.h, bitwriter.h, pq.h, node.h. In short, the main purpose of bitreader.h is to read one bit at a time of a binary file. Bit writer is to to do the same as bitreader however, it is to write one bit at a time into a binary file. Node.h is to create a binary tree. Pq.h is to store pointers to tree. In correspondence to these header files there are four c files, bitreader.c, bitwriter.c, node.c, pq.c. These eight file act as the base for the huff.c and dehuff.c files to properly execute. I will go more in depth about what the huff.c and dehuff.c file do in the program design section. However for a brief description, in the huff.c file it is supposed to enact five major steps to properly compress the provided files. These steps consist of reading the file, taking note of the frequency of each symbol, create a histogram, and from that histogram derive a code tree, and then create a code table. Then translate Huffman coded output file. Now for the dehuffman file, it is working in the opposite direction, decompressing the file instead. In this file it mostly reads the code tree already created and then decompresses through stack pointers in node.c.

Data Structures

There are a few different data structures that are utilized in this program. In the header files, structures like priority queues, binary trees, nodes, structs, linked lists are used. Most of these data structures were used for a specific reason of efficiency and effectiveness.

Algorithms

Due to this program being on the more complicated side, there are two different files that each have their own main function, huff.c and dehuff.c. As mentioned before, huff.c is for the compression of files and dehuff.c is for the decompression of files. I've included brief pseudo code of both files below.

Here is the pseudo code for huff.c:

```
function fill_histogram:
    initialize an array 'histogram' of size HISTOGRAM SIZE =  0
    ensure at least two values in the histogram are non-zero
    for each byte in input file:
        increment the histogram entry for that byte
        increment filesize
    return filesize
function create_tree:
    create a priority queue pq
    number of leaves is equal to 0
    for each entry in histogram with non-zero frequency:
        create new node with symbol, frequency
        enqueue node into pq
        increment number of leaves
    while pq has more than one node:
        dequeue two nodes left, right
        create new parent node with combined weight
        set left, right as children of parent
        enqueue parent node back into pq
    return remaining node in pq as the root of the Huffman tree
function fill_code_table
    if node is a leaf:
        store code and code length in code table at node's symbol index
    else:
        recursively call fill code table for left child
        recursively call fill code table for right child

function huff compress file:
    write header information to output buffer
    write the Huffman tree to output buffer by calling huff write tree
    for each byte in input file:
        get its Huffman code and length from code table
        write the Huffman code to output buffer bit by bit
function huff write tree
    if node is a leaf:
        write a bit 1 followed by the node's symbol to output buffer
    else:
        recursively call huff write tree for left, right children
        write a bit 0 to output buffer
    main function:
    check command line arguments
    open input file
    open output file
    create, initialize histogram
    call fill histogram
    create Huffman tree
    create code table
    create BitWriter
```

```
    compress the file using huff compress file
    close files, free resources
    return success status
```

Here is the pseudo code for dehuff.c:

```
define NodeStack structure with:
    a stack array to hold Node pointers
    an integer 'top' to represent the top of the stack
function stack init:
    set stack's top = -1
function stack is empty
    return true if stack's top = -1
    else false
function stack_push:
    if stack is not full:
        increment stack's top
        place node at new top of stack
        return true
    else:
        return false
function stack_pop
    if stack is not empty:
        retrieve node from stack's top
        decrement stack's top
        return node
    else:
        return null
function dehuff decompress file
    read first two characters from input buffer
    assert they are H,C
    read filesize and number of leaves from input buffer
    calculate number of nodes as 2 times (number of leaves) - 1
    initialize a NodeStack
    for each node in the range of number of nodes:
        read a bit from input buffer
        if bit = 1:
            read symbol from input buffer
            create a new node with the symbol
            push the node onto the stack
        else:
            pop right child from stack
            pop left child from stack
            create a new internal node
            assign left, right children to new node
            push the new node onto stack
    pop the root of the Huffman tree from stack
    for each byte in the range of filesize:
        start at the root of the Huffman tree
        while the current node is not a leaf:
            read a bit from input buffer
            move to the left child if bit = 0
            else move to right child
        write the symbol of leaf node to output file
```

```
    free Huffman tree
main function:
    check command line arguments
    open input, output files
    create BitReader for input file
    call dehuff decompress file to decompress file
    close files, free resources
    return success status
```

Function Descriptions

Due to the complexity of this program, there are a lot of functions used in the header and parallel c files. I will be breaking it down briefly by header file. Bitreader.c file:

bitread open:

```
function bitreadopen:
    allocate memory for a BitReader object
    if memory allocation fails:
        NULL
    open filename in binary read mode
    if file open fails:
        free allocated memory
        NULL
    initialize byte to 0
    set the bit position to 8
    return BitReader object
```

bitread close:

```
bit_read_close:
    if BitReader pointer and pointed BitReader = valid:
        close file
        free BitReader memory
        set the BitReader pointer = NULL
```

uint8 bit reader:

```
function bit read bit:
    if buffer = invalid
        handle error
    if bit position is greater than 7:
        read byte from the file
        if end of file:
            handle error
        set buffer byte = read byte
        reset bit position = 0
    extract a bit from the buffer byte
    return the extracted bit
```

uint8t bitreaduint8

```
function bit read uint8
    initialize a uint8t value = 0
    for 8 times
```

```
        read bit from buffer
        add bit to value at corresponding position
    return composed 8-bit value
```

uint16t bitreaduint16

```
function bit read uint16
    initialize uint16t value =  0
    for 16 times:
        read bit from buffer
        add bit to value at corresponding position
    return composed 16-bit value
```

uint32t bitreaduint32

```
function bit read uint32
    initialize uint32_t value = 0
    for 32 times:
        read bit from buffer
        add bit to value at corresponding position
    return composed 32-bit value
```

bitwriter.c:
bit write open:

```
function bit write open
    allocate memory for BitWriter object
    if memory allocation fails:
        NULL
   open filename in binary write mode
    if file open fails:
        free allocated memory
        NULL
    initialize byte = 0
    set bit position = 0
    return BitWriter object
```

bit write close:

```
function bitwriteclose:
    if BitWriter pointer, pointed BitWriter = valid:
        if bit position is greater than 0
            write byte to file
        close file
        free BitWriter memory
        set BitWriter pointer to NULL
```

bit write bit:

```
function bitwritebit:
    if buffer = invalid:
        return
    if bit position is greater than 7:
        write current byte to file
        reset byte, bit_position
```

```
    set bit at bit position in buffer byte
    increment bit position
```

bit write uint8:

```
function bitwriteuint8
    for each bit in x:
        extract bit from x
```

bit write uint16:

```
function bit write uint16
    for each bit in x:
        extract the bit from x
```

bit write uint32:

```
function bit write uint32
    for each bit in x:
        extract the bit from x
```

node.c: create:

```
function node create
    allocate memory for new Node
    if allocation fails:
        NULL
    set Node's symbol = symbol
    set Node's weight = weight
    initialize Node's left and right = NULL
    set Node's code =  0
    set Node's code_length = 0
    return new Node
```

node free:

```
function node free
    if Node pointer, pointed Node = valid:
        free Node memory
        set Node pointer = NULL
```

node print tree:

```
function node print tree
    call node print node with tree, '<' character, and initial indentation
```

pq.c:
pq create:

```
function pq create()
    allocate memory for PriorityQueue
    if allocation fails:
        NULL
    initialize list in PriorityQueue = NULL
    return PriorityQueue
```

pq free:

```
function pqfree:
    if PriorityQueue pointer, pointed PriorityQueue = valid:
        initialize current pointer to the list head
        while current is not NULL
            store current in temporary pointer
            move current to next element
            free temporary pointer
        free PriorityQueue memory
        set PriorityQueue pointer = NULL
```

pq is empty:

```
function pq is empty
    return true if the list in PriorityQueue = NULL
    return false
```

pq size is 1:

```
function pq size
    return true if the list in PriorityQueue is not NULL
    return false
```

pq less than:

```
function pqlessthan
    if weight of e1's tree = less than that of e2's tree
        true
    else if weights are equal and symbol of e1's tree is less than that of e2's tree
        return true
    return false
```

enqueue:

```
function enqueue
    allocate memory for a new ListElement
    if allocation fails:
        return
    set tree of new element to tree
    set the next of new element = NULL
    if queue is empty or new element should be first
        insert new element at beginning of the list
    else
        find the correct position to insert the new element
        insert the new element at that position
```

dequeue

```
function dequeue
    if queue = empty
        handle error
        exit
    remove first element from list
```

```
    store its tree in a temporary variable
    free removed element
    return stored tree
```

pq print:

```
function pq_print(PriorityQueue q)
    assert that q is not NULL
    initialize a pointer to the start of the list
    initialize position counter to 1
    while current element is not NULL
        print header for first element or separator for others
        call node_print_tree for the tree in the current element
        move to the next element
        increment position counter
    print footer
```

Error Handling

There are several errors that can occur with this program. If the program does not run properly or execute correctly, an error message will be printed. This was specifically implemented to make it easier to figure out where the issue is occurring. More specifically if a file that hasn't been specified or provided is called, or an invalid command line option is input, the program will report an error message and stop executing. Now moving on to the huff.c and dehuff.c files, there are certain order of verification steps that occur if any complications arise. It checks for what was entered on the command line, if bitwriter and bitreader are valid, histogram, code table, compression and decompression checks. If all of these are verified and checks out properly, that means there are no errors.

Testing

For the testing of this program there are a few different things that needed to be done. There are a few different test files provided. One of them is called brtest.c, bwtest.c, nodtest.c, and pqtest.c to test out all possible header and c files. The brtest.c file tests the bitreader.c functions. The bitwriter.c file tests the bitwriter functions. The nodetest.c file tests the functions defined in node.c. The pqtest.c runs the functions in pq.c, which is about the priority queue. There is also a runtests.sh file that is a shell script that runs all the tests together. Valgrind also needs to be run, to make sure there are no memory leaks.

Results

After writing and compiling this program, I can say that it executed successfully. The c files in correspondence to the header files had the correct function definitions, which were correctly utilized in huff.c and dehuff.c. I ran make, make format, make clean, make all, make huff, make brtest, make bwtest, along with the other test cases. After compilation there were no errors. I ran runtests.sh to check if the program was functioning correctly, and it also executed properly. I created my own test case of input.txt. I attached screenshots of the results below of this test case of input.txt into compressed.huff and the decompressed version into decompressed.txt. I ran valgrind, which produced no memory leaks, and ran successfully.

Figure 1: input txt file content

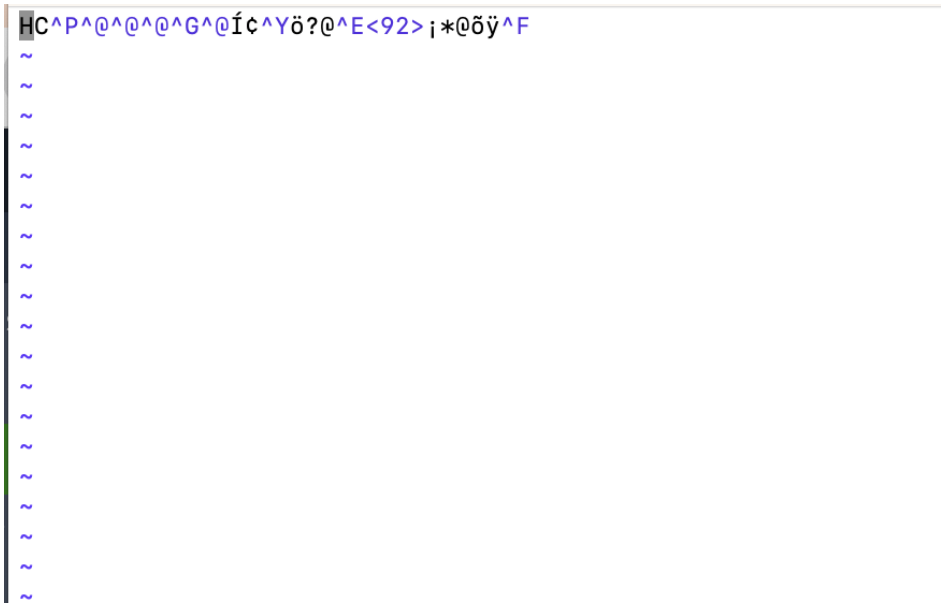

Figure 2: compressed file content

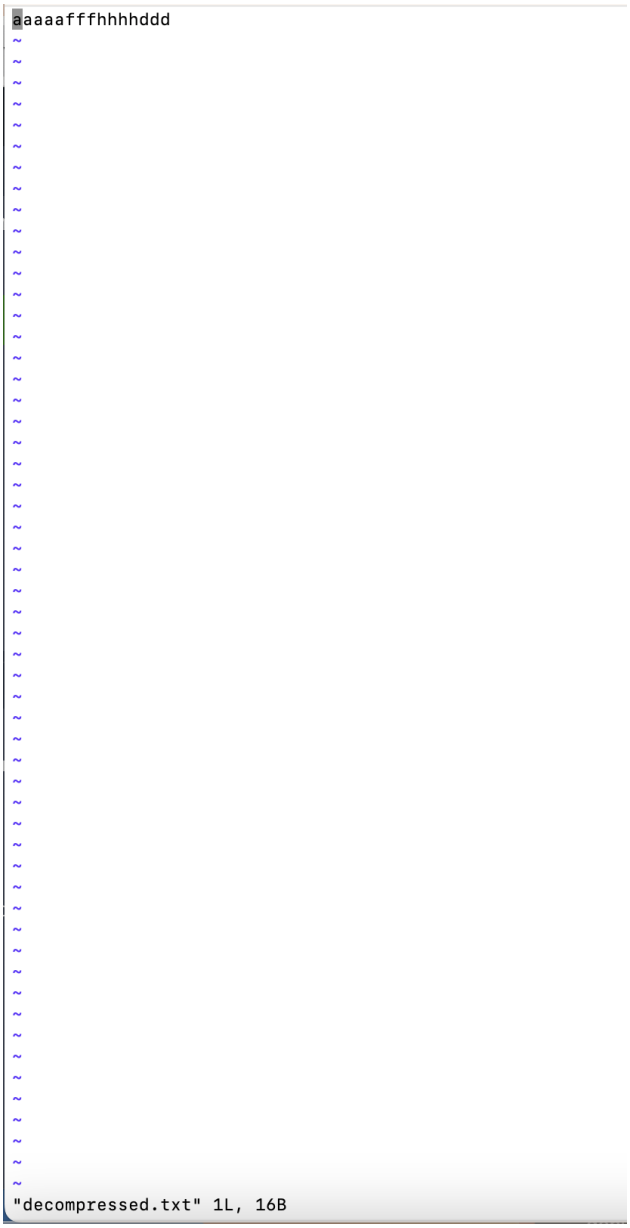

Figure 3: decompressed file content

```
asa@asarao:~/cse13s/asgn8$ ./huff -i input.txt -o compressed.huff
asa@asarao:~/cse13s/asgn8$ ls
bitreader.c  bitwriter.h  brtest.o  compressed.huff  huff       Makefile  nodetest    pq.h      pqtest.o
bitreader.h  bitwriter.o  bwtest    dehuff           huff.c     node.c    nodetest.c  pq.o      report.pdf
bitreader.o  brtest       bwtest.c  dehuff.c         huff.o     node.h    nodetest.o  pqtest    resources
bitwriter.c  brtest.c     bwtest.o  dehuff.o         input.txt  node.o    pq.c        pqtest.c  runtests.sh
asa@asarao:~/cse13s/asgn8$ vim compressed.huff
asa@asarao:~/cse13s/asgn8$ vim input.txt
asa@asarao:~/cse13s/asgn8$ ./dehuff -i compressed.huff -o decompressed.txt
asa@asarao:~/cse13s/asgn8$ ls
bitreader.c  bitwriter.h  brtest.o  compressed.huff   dehuff.o  input.txt  node.o      pq.c    pqtest.c    runtests.sh
bitreader.h  bitwriter.o  bwtest    decompressed.txt  huff      Makefile   nodetest    pq.h    pqtest.o
bitreader.o  brtest       bwtest.c  dehuff            huff.c    node.c     nodetest.c  pq.o    report.pdf
bitwriter.c  brtest.c     bwtest.o  dehuff.c          huff.o    node.h     nodetest.o  pqtest  resources
asa@asarao:~/cse13s/asgn8$ vim decompressed.txt
```

Figure 4: command line