Data Compression Tool Based on Huffman Code

Yihua Hu, hi8722

May 2, 2022

1 Introduction

This project is to design an integrated tool for file compression and decompression based on Huffman coding. As we know, Huffman coding algorithm is a classical algorithm in data compression which builds compression construction based on binary tree. This algorithm encodes characters using a variable-length code table, which is obtained by a method of getting the frequency statistics of occurrence of the source symbols. Characters with higher frequencies use shorter encodings and vice verse, characters with lower frequencies use longer encodings, which decreases the average length of the encoded files or encoded strings and save the storage space, therefore, achieving the purpose of lossless compression. In this project, we will achieve data compression through designing and implementing suitable data structures and combining different functional components.

2 Project Goals

In this project, a set of tools will be designed and implemented to achieve Huffman coding-based compression and decompression on different types of files, including ASCII text files and binary files. The following are the tools in the set:

- A standalone tool to get the frequency statistic of source symbols from a given file.
- A standalone tool to build a Huffman encoding table and its corresponding decoding table based on the given frequency statistic of source symbols.
- A standalone tool to encode a file when given Huffman encoding table.
- A standalone tool to decode a Huffman coding-based compressed file when given Huffman decoding table.
- A tool for file compression and decompression that integrates the above functions.

The basic goal of this project is to implement the above-mentioned toolset to achieve losslessy compression and correctly recover files after decompression, and on this basis, we expect to compress a file as much as possible without increasing the complexity of the project.

3 Designs

3.1 Struct Node

In this project, struct node represents nodes of the Huffman tree and contains the following variables:

- an unsigned char c for storing the source symbol;
- a long integer *count* to record the number of occurrence of the symbol in a file;
- three node pointers to represent the left child $left\_c$, the right child $right\_c$, and the parent *parent* of this node in the Huffman tree;
- a string *coding* to represent the Huffman code of this symbol.

This struct has two constructors, the main difference being that the parameters of the constructors are different. The first parameterless constructor is used to initialize the node's pointer variable as nullptr, while the second constructor takes an integer as a parameter to define a non-leaf node in the Huffman tree, which is the sum of the *count*s of the left and right children of the node. In addition, this struct also declares an operator overloadding for adding two node objects. This operator overloadding takes two node objects as parameters and return the sum of the *count*s of two objects.

3.2 Struct compare

This structure is defined for the use of the minimum heap. C++ provides the syntax $priority\_queue < int, vector < int >, greater < int >> g = gq$; for defining a heap, but since the definition is for a maximum heap, which means that the elements in the heap are sorted in descending order, in this project we need to define our own comparator *compare* rather than directly use *greater* so that the nodes in the heap will be sorted from smallest to largest according to the count of the nodes.

One interesting thing about implementing this comparator is that I named the comparator *smaller* at first, but it didn't work as well as expected until I changed the name to *compare* and got results that met my expectations.

3.3 Class HuffCode

This class is the main and key part of the whole project. It defines several variables and all the functions needed for the project, including file compression, encoding, getting frequency statistics, etc. The following are the variables declared in the class:

- a long integer $original\_bytes$ is initialized as 0 and will be equal to the number of bytes in the file to be compressed;
- a long integer $total\_bits$ is used to count the number of bits in the compressed file so that we can compute and display the compressed ratio or compressed rate;
- an integer $unique\_chars$ records the total number of different characters/symbols that appear in the file to be compressed;
- a node pointer *root* indicates the root node of the built Huffman tree;

- a string array *coding_array* with size 256. Each subscript value of the array has a corresponding ASCII code, for example, a subscript value of 65 corresponds to the character A. In this case, we can represent the corresponding character according to the subscript value. And each element of the array indicates the number of occurrences of the character corresponding to that subscript. If the number of occurrences is 0, then the position code is empty, otherwise it is the Huffman coding corresponding to the character.

- a long integer array $chars_count$ with size 256 is to store the occurrence of each unique character in the file. Since we will choose to represent each character in ASCII, we can put the ASCII code of the character in relation to the subscript of the array, for example, we know that the ASCII code of the character space is 32, then *chars_count*[32] will represent the number of times the space appears;

- an array $info$ storing nodes with size 511. Since there are at most 256 unique characters, there will be at most 256 * 2 - 1 = 511 nodes in a Huffman tree, in this case, this array is initialized with at most size 511.

3.4 Header Information

The header information contains some initial information in the compressed file and is necessary when doing file decompression. Essentially, the Huffman tree/the Huffman encoding table is required to be stored in the compressed file because the decompression tool needs this tree/this table to decode the data. In addition, the header information also contains the total number of bytes in the given file, i.e., the file to be compressed. This data is of importance since the program can know when to stop decompressing based on it.On the other hand, the total number of unique characters also needs to be stored in the compressed file so that we can know where is the end of the header information coding part.

After writing this header information into the compressed file, the encoding of the content of the original file will be stored into the compressed file.

4 Implementation Details

4.1 Get frequency statistics

This tool is to scan the given file and get the total number of unique characters appearing in the file and the number of occurrence of each symbol in the file.

1) Open the file for reading in binary mode (rb).

2) Initialize every element in array *chars_count* to be 0.

3) Use functions fseek and ftell to obtain the number of bytes of the given file *original_bytes*.

4) Iterates through the entire file, reading in one byte at a time and updating the number of occurrences of the character read in.

5) Iterate through the array *chars_count*, and count the number of non-zeros in the array and assign it to *unique_chars*.

6) Close the file.

One key point in this part is after I moved the file pointer to the end of the file by using fseek() to get the number of bytes of the given file, I needed to call function fseek() again to move the file pointer to the beginning of the file so that the file can be read in correctly. If this step is not done, only the last byte is read in.

4.2 Build a Huffman tree

This function is to build a Huffman tree from a given symbol frequency statistics. Therefore, a Huffman encoding table and its matching decoding table can be constructed after building the tree.

1) Get the symbol frequency statistics by using the previous tool. In this step, we can get the array *chars_count* and *unique_chars*.

2) Initialize a minimum heap named *tree* by using the comparator *compare* that we defined above. Syntax: $priority\_queue < node*, vector < node* >, compare > tree$;

3) Define a node pointer pointing to the array $info$, traverse the array *chars_count*. If the value of the array element is greater than 0, that means the character corresponding to the subscript of the array has appeared in the file, then record the character (node->c) and the number of occurrences of the character (node->count) to the address pointed to by the node pointer, and push the node pointer into the minimum heap *tree*. Then move the node pointer to point to the next empty element of the $info$ array.

4) After step 3), there are *unique_chars* * 2 - 1 node pointers in the heap *tree*. And the node pointer points to the (*unique_chars* * 2 - 1)th position of array $info$. To build the tree, repeat the following steps until there are less than 2 elements in the heap:

   4.1) Pop out the first two elements in *tree*. Since *tree* is a minimum heap, those two elements are the symbols with minimum occurrences in the file.

   4.2) Create a new node pointer using the second constructor in Struct Node, and assign it to be the parent node of the two nodes in step 4.1.

   4.3) Add this parent node into array $info$, push it into *tree*, and move the node pointer to point to the next empty element of the $info$ array.

5) The only one element in *tree* after step 4 is the root node of the Huffman tree which is to be built. Pop it out and use it to initialize the Huffman coding for each character.

6) The initialized code of the root node is empty, and then traverse the entire tree in pre-order. Every time the left sub-tree is added with a code of 0, and the right sub-tree is added with a code of 1. When reaching a leaf node, the code of the leaf node is the currently obtained code.

7) After step 6, each ASCII character that appears in the file has its corresponding Huffman coding. Then update the Huffman encoding table *coding_array*.

4.3 Encode a file

The implementation of this part is relatively simple. First, define a string *encoding_thing* and initialize it to null. When given a Huffman encoding table, each time a byte of the file is read (i.e. one character), then

encoding_thing = *encoding_thing* + the Huffman coding of the character read. Based on *original_bytes*, which is the total number of bytes of the original file, a for loop can be used to determine if we reach the end of the file or not.

However, the point of importance here is that since we used to reach the end of the file when scanning and counting the character information of the file, and closed the file after finishing the count. Therefore, after reopening the file we need to move the file pointer to the head of the file first.

4.4 Compress a file

There are two parts in the compressed file. The first part is the compressed header information. The second part is the compressed original content.

4.4.1 Header information

As mentioned in section 3.4, the header information contains the total number of bytes in the original file, the number of unique ASCII characters that appear in the original file and the Huffman tree.

The first one written to the compressed file is the total number of bytes in the original file, which takes up 8 bytes since it is a long integer. After that the number of ASCII characters that appear in the original file is written. Since the maximum number is 256 and the range of one byte is 0 to 255, if *unique_chars* is equal to 256, the result of writing is 0, then this information only needs to occupy one byte. When writing to this part of the Huffman tree, for each leaf node, i.e. each ASCII character, we write not only the character of that node, but also the length of the Huffman coding of the character, and then the Huffman coding of the character. Each character occupies one byte, and the length of the Huffman coding of the character also occupies one byte. One thing to consider here is that the Huffman coding of a character may be less than 8 bits or more than 8 bits. Since we write every 8 bits to the compressed file, if it is less than 8 bits, it will be merged with the following characters until it meets 8 bits. If the coding is more than 8 bits, the coding is split into two parts, and the first 8 bits are written to the file first, while the remaining bits are treated according to the rule of less than 8 bits.

Take string "lab" as an example. There are 3 ASCII characters in this string and their occurrences are all 1. After building the Huffman tree, the Huffman coding for these three characters are: 0 for l, 10 for a and 11 for b. Then, in order to store the tree, the first byte will be the character a, followed by the ASCII code 2 for STX (00000010), since the length of the Huffman coding for character a is 2. The next thing to write is the Huffman coding 10 for character a. Since it is not yet 8 bits long, it needs to be combined with the next character b. Since the ASCII code for character b is 98 (01100010), writing the third byte is 10011000, which is the character with ASCII code 152. 152 is obtained by the Huffman coding of character a combining with the high six bits of character b. The next byte written is the ASCII code 128 character, 10000000, which is equal to the lower two bits of b combining with the higher six bits of the Huffman coding length of b (00000010). The following byte is the character with ASCII code 182 (10110110). The top two bits of 10110110, 10, comes from the lower two bits of the length of the Huffman coding of b, the next 11 comes from the Huffman coding of b, and 0110 is the upper four bits of the character l.Then, the next byte will be the character with ASCII code 192 (11000000) where 1100 is the lower four bits of character l and 0000 is the higher four bits of the length of the Huffman coding of l. The final part for this tree is 00010 where 0001 is the lower four bits of the length of the Huffman coding of l and the final 0 is the Huffman coding for character l. Since it is not yet 8 bits long, this part will be combined with the file content and

then written into the compressed file, and then the complete 8 bits will be 00010 + the Huffman coding of l + the Huffman coding of a = 00010010 = ASCII code 18 DC2.

4.4.2 File content

Compared to the storage of header information, the compressed writing of file content is relatively simple. Traverse the entire file, read in one character at a time and convert to its Huffman coding, which needs to be written according to the 8-bit rule mentioned earlier due to the variable length of the coding. If it is less than 8 bits, continue reading in the next character until it meets the 8-bit length. When the last character is read, if the byte is less than 8 bits at this time, the low bit is written to the compressed file after completing 8 bits with 0. And then close the file.

4.5 Decode a file

We introduced the compression of the header information in section 4.4.1, so for the decompression program, we first need to take out the first 8 bytes, which is equal to the total number of bytes of the original file *original_bytes*. The next byte represents the number of ASCII characters that have appeared in the source file, i.e., *unique_chars*. If the value of *unique_chars* is 0, then it means that all 256 ASCII characters are present in the source file. After getting this information we can reconstruct the Huffman tree of the source file completely based on *unique_chars*. In a for loop, we first read two bytes at a time, the first byte representing a character and the second byte representing the length of the Huffman encoding of that character. Based on this length value n, we can know the Huffman encoding of this byte as the first n bits of the next byte or if n ¿ 8, then the next next byte also needs to be written. And the process of Huffman coding in reading characters is also the process of building a tree.

Let's still use the "lab" example mentioned earlier as an example. (1) The first byte after taking *original_bytes* and *unique_chars* is character a, and the next byte is ASCII character STX which is equal to integer 2, which means that the length of the Huffman coding of character a is 2. Then the next byte is ASCII character with code 152 (10011000), thereby, the first two bits 10 being the Huffman coding of character a, and the remaining bits are 011000. Then defining a node pointer which is equal to the root node of the Huffman tree, once a 1 is read, the node pointer will move to its right child node and once a 0 is read, the node pointer will move to its left child node. So back to the character a, since its Huffman coding is 10, so the path from the root node to the node representing a is root node -> the right child node of the root node -> the left child node of the right child node of the root node (node a). Then the next byte is the character with ASCII code 128 (10000000), so the next character is the combination of the remaining bits and the higher 2 bits of 10000000, which is 01100010 (binary), or 98 (decimal). (2) So the next character is b and the remaining bits are 000000. Then read the next byte which is the character with ASCII code 182 (10110110). Pick the high 2 bits and combine it with the remaining bits, then we get 00000010, which is the length of the Huffman coding of character b, equal to 2, and the new remaining bits are 110110. Then based on the length information, we can know that the Huffman coding of character b is 11, and the remaining bits are 0110. And similar to character a, we can also find the path from the root node to the node representing b: root node -> the right child node of the root node -> the right child node of the right child node of the root node (node b). (3) Read in the next byte which is the character with ASCII code 192 (11000000). So combine the remaining bits with the high 4 bits of the current byte, then we get 01101100, which is 108 in decimal and the ASCII character is l, and the remaining bits turn to be 0000. Continue to get the next

byte then we get 00010010. Combining the remaining bits and the high 4 bits of the current byte we can get 00000001 and the updated remaining bits are 0010, which means the length of the Huffman coding of character l is 1. So read in one bit from the remaining bits then 0 is the Huffman coding of character l. The path from the root node to the node representing l: root node -> the left child node of the root node (node l). And the rest 010 belongs to the compression content for the source file content.

In sum, after decompressing the header information, we are able to construct the Huffman decoding table and build the Huffman tree. Starting from the root node and using the constructed Huffman tree, when we read a bit value 0, we move to the left sub-tree, and when we read a bit value 1, we move to the right sub-tree until we meet a leaf node and add the character of the leaf node to the decoded string. Repeat this process until all the bits are read. Take the remaining bits 010 mentioned above as an example. We read a 0 and then we move to the left child of the root node and we find that this is a leaf node with character l, therefore, l is added to the decoding string. Then we continue to read a bit 1, moving to the right child of the root node, and then read a bit 0, we are now reaching the left child node of the right child of the root node, which is a leaf node with character a, so a is added to the decoding string and the decoding string is currently "la".

4.6 Decompress a file

The tool to decompress a file is similar with to decode a file, including the decompression of the header information and the way to re-build the Huffman tree. The only difference is that whenever a leaf node is read, instead of adding the character of the leaf node to the decoding string, the character is written to the decompression file.

5 Tests and Benchmarks

5.1 Tools tests

The following are some screenshots which verify the availability and accuracy of the above tools. Take the title of this project "Data Compression Tool Based on Huffman Code" as an example. Figure 1 shows the frequency statistics, figure 2 shows the Huffman encoding table, figure 3 shows the test results of three functions, encoding, compression and decoding, and figure 4 shows the test result of the decompression function.

From the figures, we can see that those tools works.

5.2 Benchmarks

The following files are from The Silesia Corpus, which can be downloaded from `http://www.data-compression.info/Corpora/SilesiaCorpus/index.html`

6 Summary & Discussion

For me this project exercise was an interesting practice. When it comes to Huffman coding, I can give a simple example to illustrate the principle of the algorithm. But when it came time to actually implement

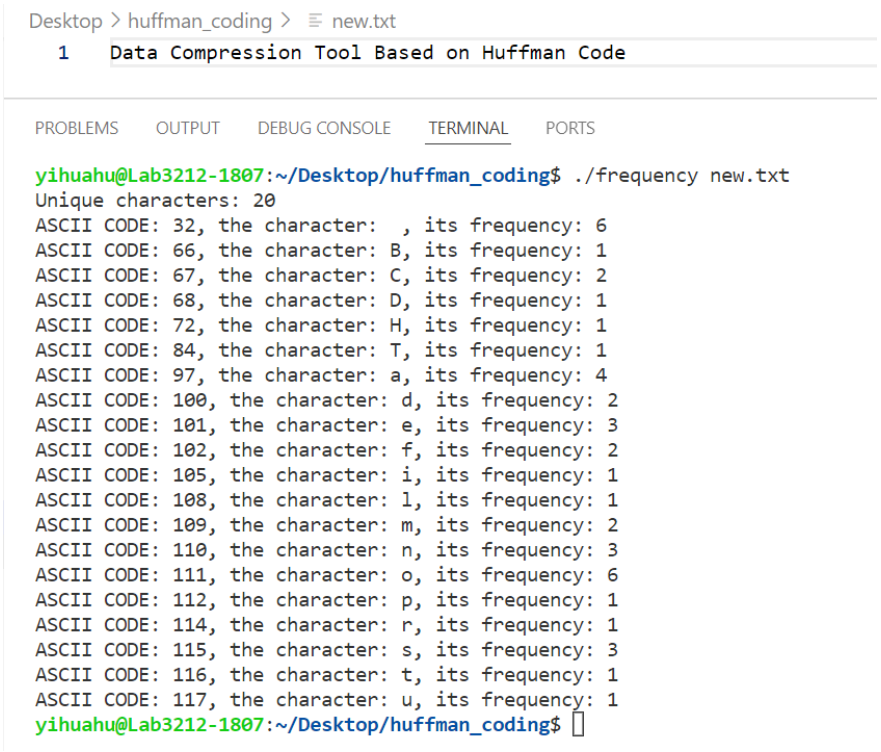

Figure 1: Frequency stats

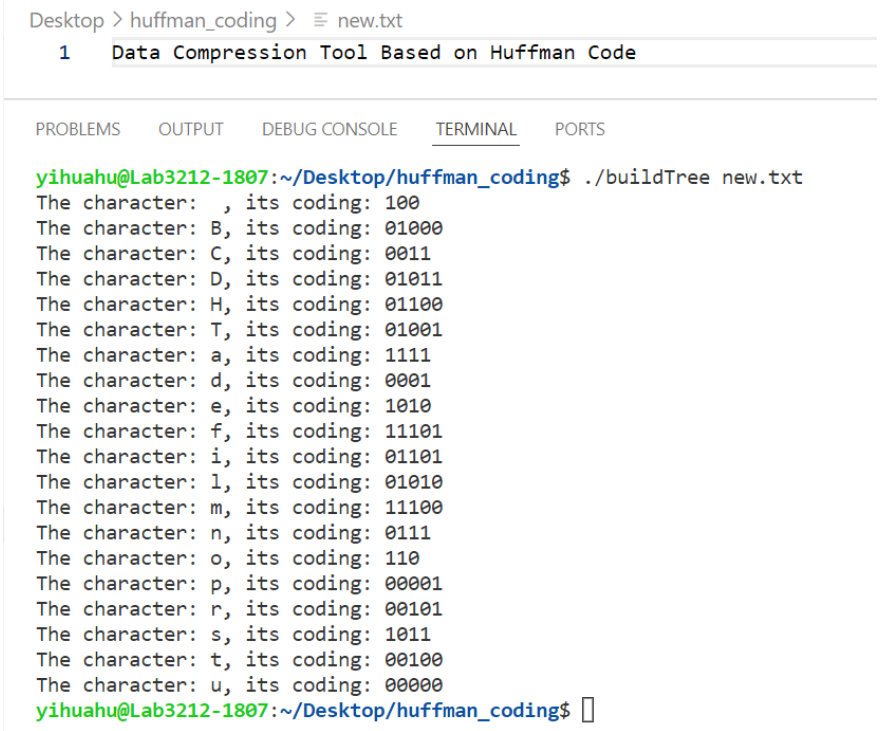

Figure 2: Huffman Encoding table

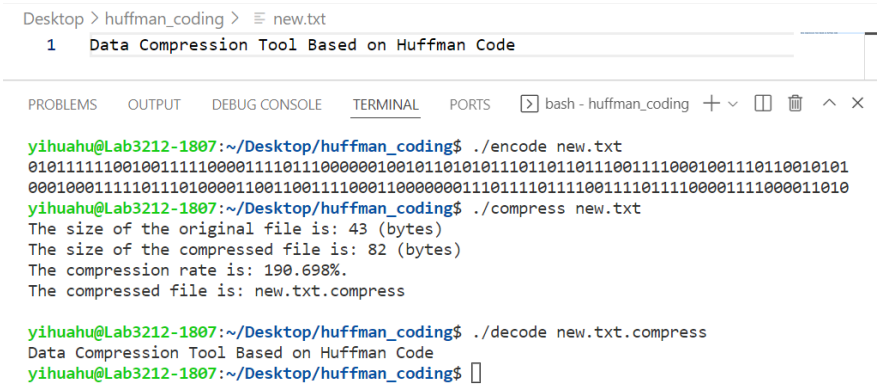

Figure 3: Encode & Compress & Decode

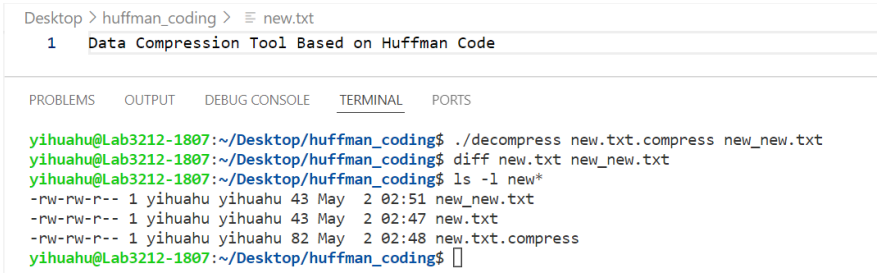

Figure 4: Decompress

Table 1: Compression results

File	Original Size (bytes)	Compressed Size (bytes)	Compression Ratio
dickens	10192446	5826280	1.749
mozilla	51220480	39977674	1.281
mr	9970564	4623723	2.156
nci	33553445	10224124	3.282
ooffice	6152192	5125025	1.200
osdb	10085684	8342702	1.209
reymont	6627202	4032239	1.644
samba	21606400	16547396	1.306
sao	7251944	6844056	1.060
webster	41458703	25929207	1.600
x-ray	8474240	7022134	1.207
xml	5345280	3711399	1.440

the algorithm from scratch and apply it to data compression, I had a thought process from point to line to surface, from designing tree nodes, to building trees with minimal heaps, to thinking about how to store the necessary auxiliary information to compress the files as much as possible. From the class presentation and section 5 we can see that all requested functions have been completed. And as we can see from Table 1, for some files, the algorithm can compress up to one-third of the original file size, while for some files, the algorithm only compresses one-tenth. So the compression program still has room for improvement. On the other hand, what we are considering here is the compression and decompression of a single file, while in our daily life we often do the compression and decompression of a folder containing multiple files. Therefore, we can try to compress and decompress the folder.