File Compression Using Huffman Coding

2020310830 SungHwan Cho

November 30, 2024

1 Introduction

The goal of this assignment is to implement a program that compresses and uncompresses files using Huffman coding. ASCII text file named 'input.txt' containing any ASCII characters (0-127). We have to make 2 output files, 'output1.txt' and 'output2.txt'.

- output1.txt: Construct a Huffman tree based on the ASCII text in the input file and store the binary encoding result along with the tree.
- output2.txt: Decode the tree and binary data from the 'output1.txt' file to save both the binary code for each character and the original text.

2 Implementation

2.1 Constructing Tree

To create a Huffman tree, the frequency of each character in the input must first be recorded. This frequency data is essential because the Greedy Algorithm relies on it. After recording the frequencies, a Min Heap is constructed based on this information.To construct the Huffman Tree using the Min Heap, the Greedy Algorithm is applied as follows:

1. Pop the node with the smallest frequency from the Min Heap.
2. After heapifying, pop the next minimum frequency node from the Min Heap.
3. Create a new blank node with a frequency equal to the sum of the two nodes' frequencies, and add this new node to the Min Heap.
4. Repeat steps 1–3 until there is only one node left in the Min Heap.
5. The remaining node in the Min Heap becomes the root node of the Huffman Tree.

2.2 Encoding

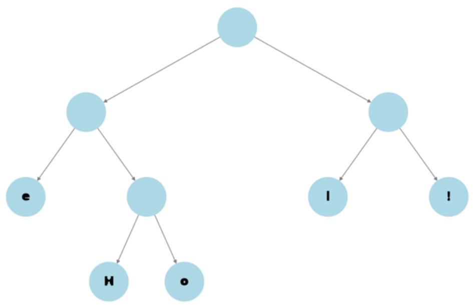
Encoding is divided into two processes: visualizing the tree and converting the original text into binary.

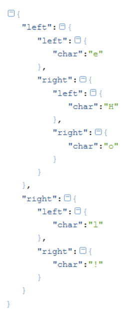
For visualizing the tree, the **JSON format** was adopted. Using commonly used characters such as **( )** or ***** for visualization could cause confusion during the decoding process, as it would be unclear whether a character is part of the original input or a separator used for visualization.

The method of converting the text into binary involved performing a DFS(Depth First Search) on the tree. When reaching a leaf node, the binary code for the corresponding character was stored in a table. This table was then used to convert the original text into its binary representation.

2.3 Decoding

Encoding is also divided into two processes: reconstructing the tree and decoding the binary code to the original text.





```
{
  "left":{
    "left":{
      "char":"e"
    },
    "right":{
      "left":{
        "char":"H"
      },
      "right":{
        "char":"o"
      }
    }
  },
  "right":{
    "left":{
      "char":"l"
    },
    "right":{
      "char":"!"
    }
  }
}
```

Figure 1: Original Tree and representation with JSON

To reconstruct the tree, the input in JSON format is used to establish the connections between the nodes. After the tree is successfully created, decoding is performed by traversing the graph based on the binary code sequence.

The detail of decoding process proceeds as follows:

1. Traverse the tree following the sequence one by one. Once a leaf node is reached, the character value of that leaf node is returned.
2. After reaching the leaf node, return to the root and continue the search with the remaining sequence.
3. This process repeats until the entire binary sequence is decoded.

3 Performance Analysis

To express the time complexities of various operations using the Big O notation, we define the variables as follows:

(i) n: The total number of input characters.

(ii) v: The total number of nodes in the Huffman Tree

(iii) B: The total length of the encoded binary code

Because the total number of nodes in the Huffman tree cannot exceed twice the number of ASCII characters(128), v can considered constant.

make minheap	encoding	make Huffman tree	decoding
$O(vlogv)$	$O(nlogv)$	$O(vlogv)$	$O(B)$

Table 1: Time complexity of each operation

3.1 Make Min Heap

To construct the Min Heap, we add v nodes, performing a heapify up operation for each node. Since the heapify operation takes $O(logv)$, and we do this for all v nodes, the total time complexity is $O(vlogv)$.

3.2 Make Huffman Tree

(i) *Encoding*: Constructing the tree during encoding involves repeatedly removing the two smallest nodes from the heap, combining them, and reinserting the new node. This process is repeated $v - 1$ times, with each operation taking $O(vlogv)$. Hence the total time complexity is $O(vlogv)$.

(ii) *Decoding*: Constructing the tree from the JSON input involves parsing the string and creating nodes directly. Each node is processed once, giving $O(v)$.

3.3 Encoding

Assigning binary codes to characters involves performing a DFS traversal of the tree. For each character, the traversal takes time proportional to the tree's depth $O(logv)$. Encoding the input text involves text involves repeating this process n times so the total time complexity is $O(nlogv)$

3.4 Decoding

Generating binary codes involves creating binary codes for each node in the tree, which takes $O(v)$. Each bit in the encoded sequence is processed by traversing the tree. This takes $O(B)$, where B is the total number of bits. The length of bits of each encoded character cannot exceed the depth of the tree, so $B \approx n \log v$. Therefore, the time complexity of this step can be represented as $O(n \log v)$.

3.5 Overall Time Complexity

Combining all the process, since v is a constant, the overall time complexity in term of n is $O(nlogv)$.

4 Code Explanation

4.1 Node Structure

```
typedef struct Node
    char character; // ASCII of this character
    uint frequency; // the frequency of this character in input
    struct Node *left, *right; // pointers of left, right children
```

4.2 Functions

4.2.1 main()

First it handles the input.txt and encodes this text using Huffman Tree. Then it decodes the result of the encoded text.

Algorithm 1 Main Algorithm

```
1: Phase 1: Encoding
2: Count character frequencies using INPUT_HANDLER(input_file).
3: Build a min-heap using CREATE_PQ(pq).
4: Construct Huffman Tree with HUFFMAN_ENCODING(pq).
5: Save Huffman Tree and encoded binary to file.
6: Phase 2: Decoding
7: Parse Huffman Tree from file using PARSE_JSON_TREE(input2).
8: Generate Huffman codes with GENERATE_HUFFMAN_CODE(root2).
9: Decode binary data back to text using DECODE_HUFF(input2, output2).
```

4.2.2 create_pq(Node* pq)

Before constructing the Huffman tree, build a Min Heap for using the Greedy Algorithm.

Algorithm 2 Create Priority Queue (Min-Heap)

```
1: for each character with non-zero frequency do
2:     Create a new node and insert it into pq.
3:     Adjust the heap using HEAPIFY_UP(pq).
4: end for
```

4.2.3 huffman_encoding(Node* pq)

Construct the Huffman tree using Min Heap until the size of Min Heap is 1 (root node).

Algorithm 3 Construct Huffman Tree

```
1: while more than one node in pq do
2:     Extract two nodes with smallest frequencies.
3:     Create a new node combining their frequencies.
4:     Insert the new node into pq.
5: end while
6: return the remaining node as the tree root.
```

4.2.4 save_tree(FILE* output, Node* root)

Save the Huffman tree to JSON format.

Algorithm 4 Save Huffman Tree as JSON

```
1: if root is a leaf node then
2:     Write the character as a JSON leaf node.
3: else
4:     Write a JSON object with left and right subtrees.
5:     Recursively save left and right subtrees.
6: end if
```

4.2.5 generate_huffman_code(Node* root, char* code, int depth, char** code_table)

Using the Huffman tree, we can generate the Huffman code of each ASCII character which is the result of encoding. And save each character's bits to code_table.

Algorithm 5 Generate Huffman Codes

```
1: if root is a leaf node then
2:     Save the current code in code_table[root.character].
3: else
4:     Add '0' to code and recurse left.
5:     Add '1' to code and recurse right.
6: end if
```

4.2.6 parse_json_tree(FILE* input)

Start of the decoding process. First we have to reconstruct the tree from JSON format input.

Algorithm 6 Parse JSON Huffman Tree

```
1: if leaf node then
2:     Create and return a new leaf node.
3: else
4:     Recursively parse left and right subtrees.
5:     Create and return an internal node.
6: end if
```

4.2.7 decode_huff(FILE* input, FILE* output, Node* root)

Decode the binary data using the Huffman tree. We simply move to the left or right based on the bits in the input.

Algorithm 7 Decode Huffman Encoded Data

```
1: Set current to the root.
2: while not end of file do
3:     if bit is '0' then
4:         Move to the left child.
5:     else if bit is '1' then
6:         Move to the right child.
7:     end if
8:     if current is a leaf node then
9:         Write current.character to output.
10:        Reset current to root.
11:    end if
12: end while
```

5 Conclusion

Using Huffman coding, the original string could be compressed and represented efficiently. Implementing this required an understanding of various data structures and algorithms, including Min Heap, recursive functions, and DFS. The time complexity of the algorithm is $O(nlogv)$, where v represents the limited set of ASCII characters. Given the nature of ASCII, v can be treated as a constant, enabling the implementation of an efficient algorithm.