Introduction to Algorithm

Project Report

Member: Liner Zhang

CUG No: 20231003496

IIT No: A20563408

Teacher: Mr. Peng

Date: December 16, 2024

Contents

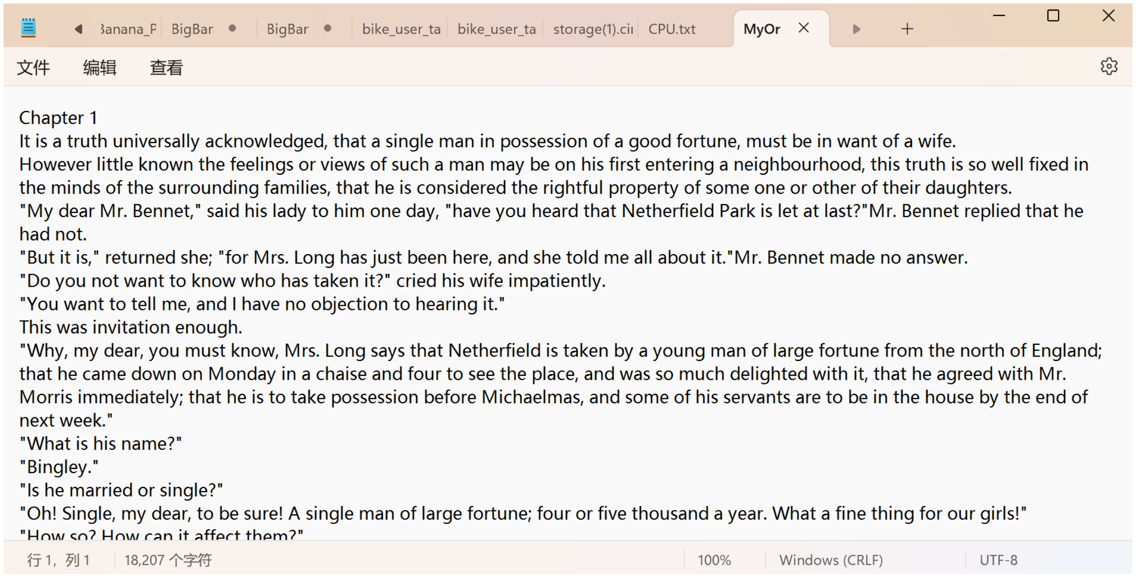
1 Description of the Project

1.1 Program Instruction

Users simply need to fill in the ”MyOriginalText.txt” file and click the ”Run” button. The program will automatically generate the optimal Huffman encoding method, output the parameters k and n, and calculate the shortest length of the encoded code. It will then save the results in several files: the Huffman dictionary will be saved to ”MyCodeWorkBk.txt”, the encoded text will be saved to ”MyEncodingFile.txt”, and the decoded text will be saved to ”MyDecodingFile.txt”. Additionally, the program will generate the Huffman tree in two formats: as text in ”MyHuffmanFile.txt” and as an image in ”MyHuffmanTree.png”.

1.2 Program Demonstration

I provide a piece of material from *Pride and Prejudice*, which contains over 3000 words. The contents of the file ”MyOriginalText.txt” are shown in Figure 1.



```
Chapter 1
It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.
However little known the feelings or views of such a man may be on his first entering a neighbourhood, this truth is so well fixed in
the minds of the surrounding families, that he is considered the rightful property of some one or other of their daughters.
"My dear Mr. Bennet," said his lady to him one day, "have you heard that Netherfield Park is let at last?"Mr. Bennet replied that he
had not.
"But it is," returned she; "for Mrs. Long has just been here, and she told me all about it."Mr. Bennet made no answer.
"Do you not want to know who has taken it?" cried his wife impatiently.
"You want to tell me, and I have no objection to hearing it."
This was invitation enough.
"Why, my dear, you must know, Mrs. Long says that Netherfield is taken by a young man of large fortune from the north of England;
that he came down on Monday in a chaise and four to see the place, and was so much delighted with it, that he agreed with Mr.
Morris immediately; that he is to take possession before Michaelmas, and some of his servants are to be in the house by the end of
next week."
"What is his name?"
"Bingley."
"Is he married or single?"
"Oh! Single, my dear, to be sure! A single man of large fortune; four or five thousand a year. What a fine thing for our girls!"
"How so? How can it affect them?"
```

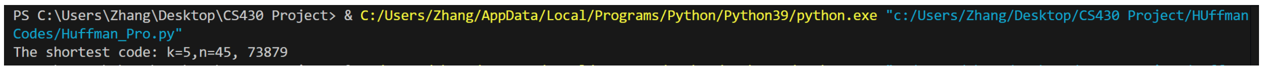
Figure 1: MyOriginalText.txt

The output of the program, representing the best-case result, along with the other related output files, is shown in Figures 2-7.

From Figure 2, we can determine that the shortest length of the encoded code is 73,879, which occurs when the top forty-five most frequent five-letter words are added to the Huffman tree.

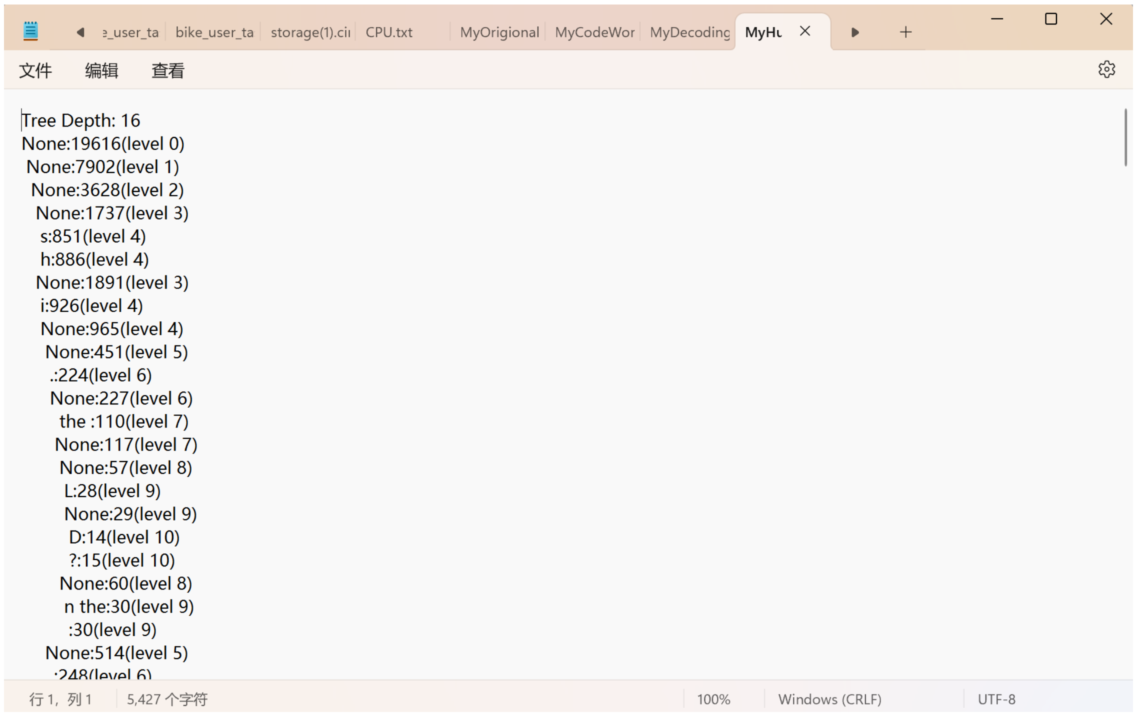
From Figure 5, we can indeed identify several five-letter words.

From Figure 7, we can see the decoded text is the same as the origional text, which means the Huffman encoding method is effective.



```
PS C:\Users\Zhang\Desktop\CS430 Project> & C:/Users/Zhang/AppData/Local/Programs/Python/Python39/python.exe "c:/Users/Zhang/Desktop/CS430 Project/HUffman Codes/Huffman_Pro.py"
The shortest code: k=5,n=45, 73879
```

Figure 2: Program Output



```
Tree Depth: 16
None:19616(level 0)
 None:7902(level 1)
  None:3628(level 2)
   None:1737(level 3)
    s:851(level 4)
    h:886(level 4)
   None:1891(level 3)
    i:926(level 4)
    None:965(level 4)
     None:451(level 5)
      .:224(level 6)
      None:227(level 6)
       the :110(level 7)
       None:117(level 7)
        None:57(level 8)
         L:28(level 9)
         None:29(level 9)
          D:14(level 10)
          ?:15(level 10)
        None:60(level 8)
         n the:30(level 9)
         :30(level 9)
     None:514(level 5)
```

Figure 3: MyHuffmanFile.txt

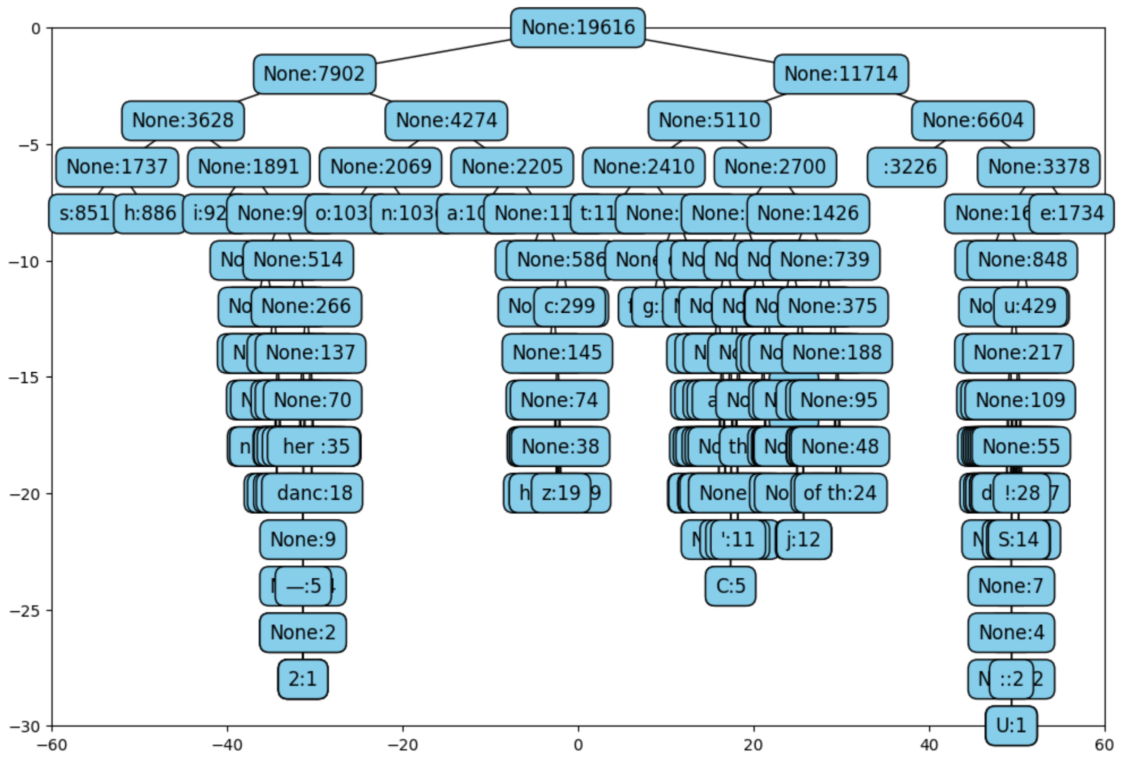
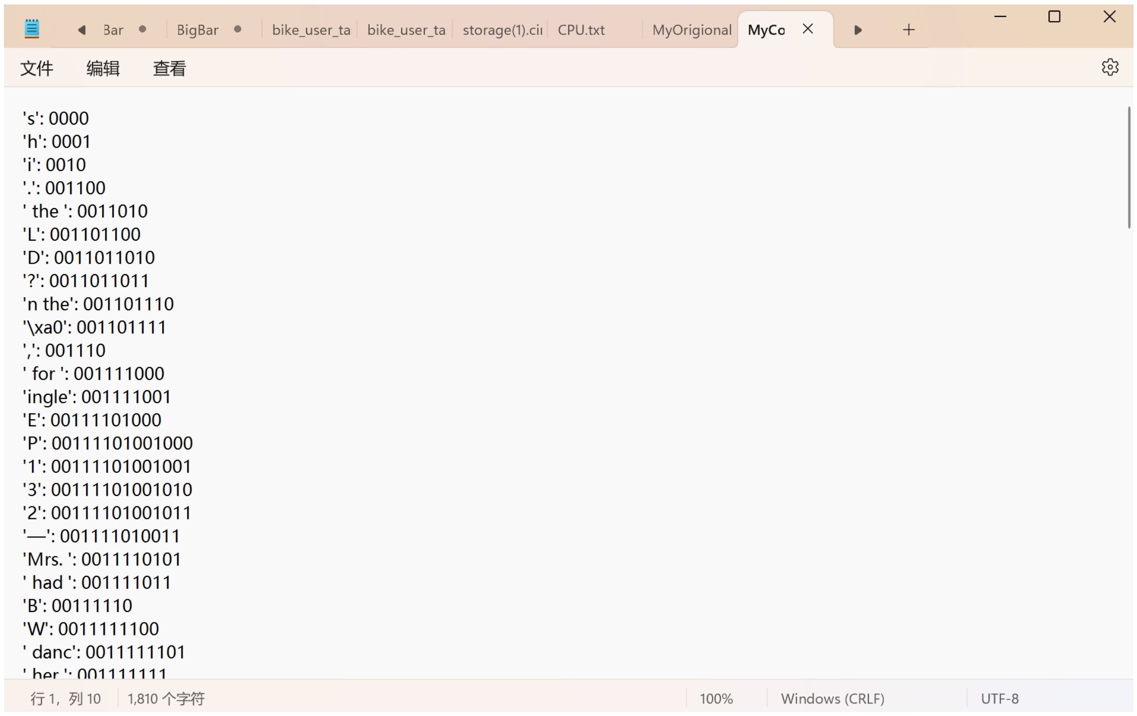
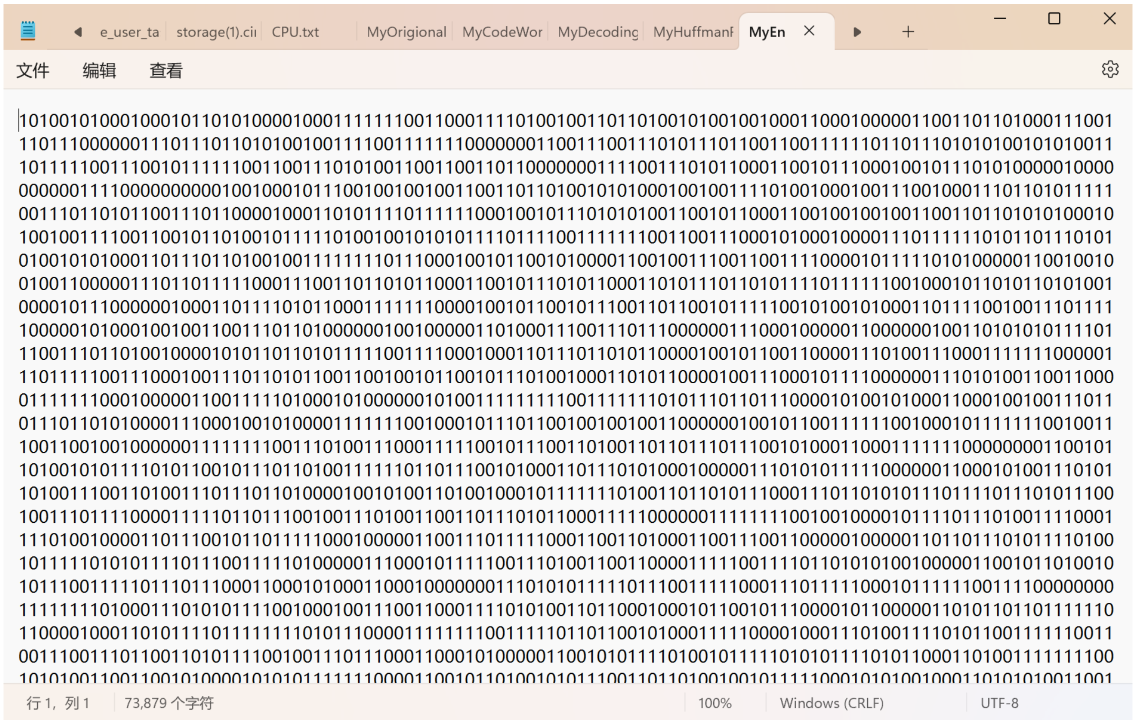


Figure 4: MyHuffmanTree.png



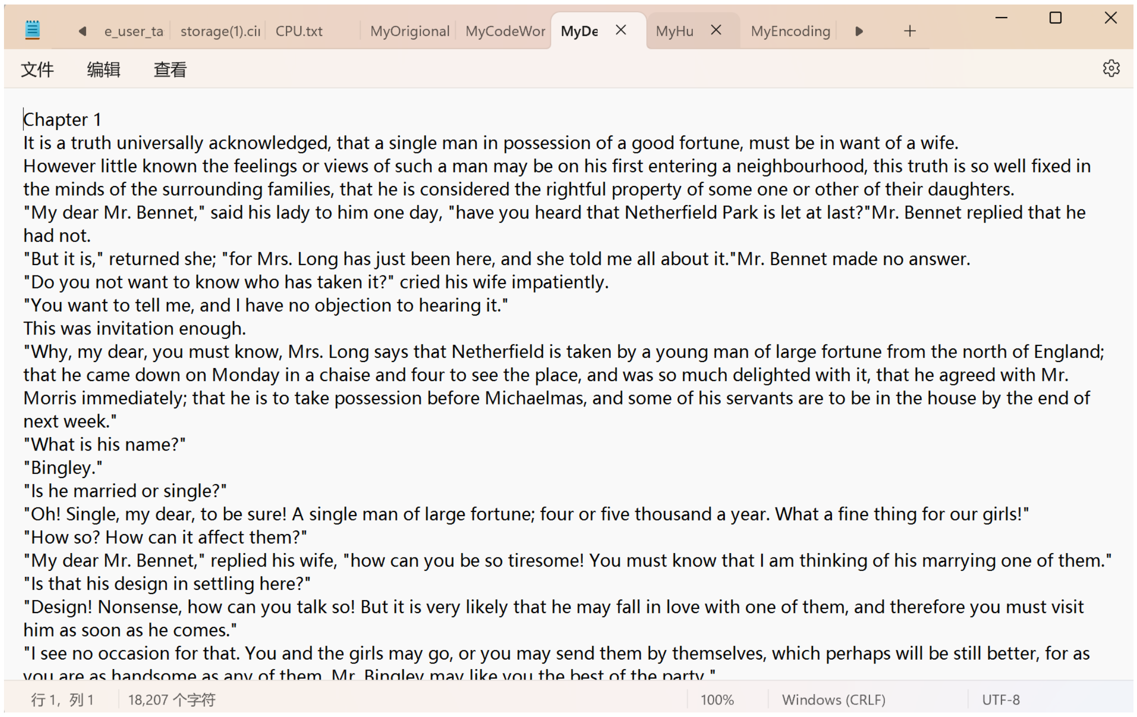
```
's': 0000
'h': 0001
'i': 0010
'.': 001100
' the ': 0011010
'L': 001101100
'D': 0011011010
'?': 0011011011
'n the': 001101110
'\xa0': 001101111
',': 001110
' for ': 001111000
'ingle': 001111001
'E': 00111101000
'P': 00111101001000
'1': 00111101001001
'3': 00111101001010
'2': 00111101001011
'—': 001111010011
'Mrs. ': 0011110101
' had ': 001111011
'B': 00111110
'W': 0011111100
' danc': 0011111101
' her ': 001111111
```

Figure 5: MyCodeWorkBk.txt



```
1010010100010001011010100001000111111100110001111010010011011010010100100100011000100000110011011010001110 01
1101110000001110111011010100100111100111111100000001100111001110101110110011001111110110111010101001010100 11
1011111001110010111111001100111010100110011001101100000001111001110101100011001011100010010111010100000100 00
0000001111000000000010010001011100100100100110011011010010101000100100111101001000100111001000111011010111 11
0011101101011001110110000100011010111101111110001001011101010100110010110001100100100100110011011010101000 10
1001001111001100101101001011111010010010101011110111100111111100110011100010100010000111011111101011011101 01
0100101010001101110110100100111111110111000100101100101000011001001110011001111000010111110101000001100100 10
0100110000011101101111000111001101101011000110010111010110001101011101101011101111110010001011010110101001
0000101110000000100011011110101100011111110000100101100101110011011001011111001010010100011011110010011101111
1000001010001001001100111011010000000100100000110100011100111011100000011100010000001100000010011010101011101
1100111011010010000101011011010111110011110001000110111011010110000100101100110000111010011100011111111000001
1101111100111000100111011010110011001001011001011101001000110101100001001110001011110000000111010100110011000
0111111100010000001100111110100010100000001010011111111100111111101011101101110000101001010001100010010011011
0111011010100001110001001010000111111100100010111011001001001001100000010010110011111100100010111111110010011
1001100100100000011111111001110100111000111110010111001101001101101110110010100011000111111110000000001100101
1010010101111010110010111011010011111110110111001010001101110101000100000111010101111100000011000101001110101
1010011100110100111011101101000010010100110100100010111111110100110110101110001110110101011101111011101011100
1001110111100001111101101110010011101001100110111010110001111100000011111111001001000010111101110100111110001
1110100100001101110010110111111000100000110011101111100011001101000110011100110000010000011011011101011110100
1011111010101111011100111110100000011100010111110011101001100110000111110011110110101010010000011001011010010
1011100111110111011100011000101000110001000000011101010111110111001111100011101111100010111111001111000000000
1111111101000111010101111001000100111001100011110101001101100010001011001011100001011000001101011011011111110
1100001000110101111011111111101011100001111111100111110110110010100011111000010001110100111101011001111110011
0011100111011001101011110010011101110001100010100000110010101111010010111110101011110101100011010011111111100
```

Figure 6: MyEncodingFile.txt



Chapter 1
It is a truth universally acknowledged, that a single man in possession of a good fortune, must be in want of a wife.
However little known the feelings or views of such a man may be on his first entering a neighbourhood, this truth is so well fixed in the minds of the surrounding families, that he is considered the rightful property of some one or other of their daughters.
"My dear Mr. Bennet," said his lady to him one day, "have you heard that Netherfield Park is let at last?"Mr. Bennet replied that he had not.
"But it is," returned she; "for Mrs. Long has just been here, and she told me all about it."Mr. Bennet made no answer.
"Do you not want to know who has taken it?" cried his wife impatiently.
"You want to tell me, and I have no objection to hearing it."
This was invitation enough.
"Why, my dear, you must know, Mrs. Long says that Netherfield is taken by a young man of large fortune from the north of England; that he came down on Monday in a chaise and four to see the place, and was so much delighted with it, that he agreed with Mr. Morris immediately; that he is to take possession before Michaelmas, and some of his servants are to be in the house by the end of next week."
"What is his name?"
"Bingley."
"Is he married or single?"
"Oh! Single, my dear, to be sure! A single man of large fortune; four or five thousand a year. What a fine thing for our girls!"
"How so? How can it affect them?"
"My dear Mr. Bennet," replied his wife, "how can you be so tiresome! You must know that I am thinking of his marrying one of them."
"Is that his design in settling here?"
"Design! Nonsense, how can you talk so! But it is very likely that he may fall in love with one of them, and therefore you must visit him as soon as he comes."
"I see no occasion for that. You and the girls may go, or you may send them by themselves, which perhaps will be still better, for as you are as handsome as any of them, Mr. Bingley may like you the best of the party."

Figure 7: MyDecodingFile.txt

2 Algorithm Design

In designing the algorithm, I used the classic binary tree method to implement Huffman encoding. However, in addition to counting the frequency of individual letters, I also considered combinations of letters. To determine how many letter combinations to include and how many of the most frequent combinations to consider, I set two parameters to iterate and compare, in order to identify the parameter combination that produces the shortest encoded length. The detailed algorithm and code are as follows.

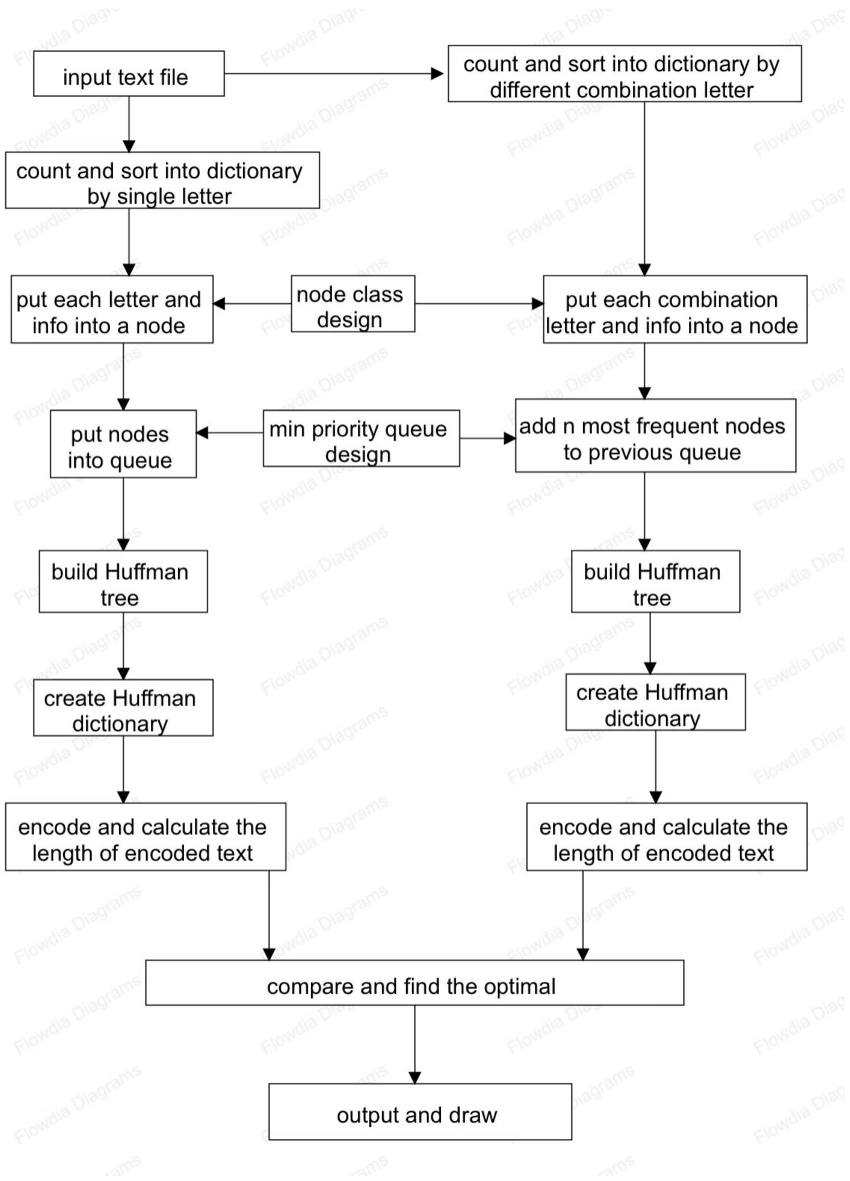

input text file
count and sort into dictionary by different combination letter
count and sort into dictionary by single letter
put each letter and info into a node
node class design
put each combination letter and info into a node
put nodes into queue
min priority queue design
add n most frequent nodes to previous queue
build Huffman tree
build Huffman tree
create Huffman dictionary
create Huffman dictionary
encode and calculate the length of encoded text
encode and calculate the length of encoded text
compare and find the optimal
output and draw

Figure 8: Flowchart

2.1 Data Processing

Before starting the algorithm, the input document needs to be processed. First, the content of the document must be read and stored as a string.

Based on my requirements, I also need to calculate the frequency of individual letters as well as the frequency of letter combinations, where k represents the number of letters in the combination. Since I will later need to map each letter to its corresponding frequency, I use a dictionary to store this mapping.

Data Processing Code

```
def parse_file(input_file):
    with open(input_file, 'r', encoding='utf-8') as f:
        file = f.read()
    return file
def count(file):
    counts = defaultdict(int)
    for i in range(len(file)):
        counts[file[i]] += 1
    sorted_counts = sorted(counts.items(), key=lambda x: x
       [1], reverse=True)
    return sorted_counts
def count_1(file, k):
    counts_three = defaultdict(int)
    for i in range(len(file)):
        if i + k - 1 < len(file):
            counts_three[file[i:i+k]] += 1
    sorted_3 = sorted(counts_three.items(), key=lambda x: x
       [1], reverse=True)
    return sorted_3
```

2.2 Huffman Tree Construction

Considering the algorithm, which requires continuously extracting the characters (or character combinations) with the least frequency and adding their frequencies together, a minimum queue is used here. In the queue class, it is specified that each time a new element is added, it should be placed in the correct position to ensure the elements in the queue are always sorted in descending order, and that the extracted element is always the minimum value.

Additionally, since a tree needs to be built later, a Node class is designed to define the parent-child relationships and to store information associated with each node.

When constructing the Huffman tree, all characters (or character combinations) involved in the encoding are first placed into the queue. The add_to_tree function is then used to repeatedly extract the elements with the smallest frequencies, merge them, establish parent-child relationships, and put the merged elements back into the queue. This process continues until only one element remains in the queue, which becomes the root of the tree.

Queue Class Code

```
class min_queue:
    def __init__(self):
        self.queue=[]
    def enqueue(self, node):
        if not self.queue:
            self.queue.append(node)
        else:
            i = 0
            while i < len(self.queue) and self.queue[i].
               get_value() >= node.get_value():
                i += 1
            self.queue.insert(i, node)
    def dequeue(self):
        return self.queue.pop(-1)
    def length(self):
        return len(self.queue)
```

Node Class Code

```
class Node:
    def __init__(self,char,value):
        self.char=char
        self.value=value
        self.left=None
        self.right=None
        self.code=None
        self.parent=None
    def __lt__(self, other):
        return self.value < other.value
    def left_child(self,node):
        self.left=node
    def right_child(self,node):
        self.right=node
    def set_parent(self,node):
        self.parent=node
    def get_left(self):
        return self.left
    def get_right(self):
        return self.right
    def get_parent(self):
        return self.parent
    def get_value(self):
        return self.value
    def get_char(self):
        return self.char
    def set_code(self,code):
        self.code=code
```

Huffman Tree Construction Code

```
def add_to_tree(queue):
    if queue.length()==1:
        root=queue.dequeue()
        return root
    else:
        node1=queue.dequeue()
        node2=queue.dequeue()
        new_node=Node(None,node1.get_value()+node2.get_value
            ())
        new_node.left_child(node1)
        new_node.right_child(node2)
        node1.set_parent(new_node)
        node2.set_parent(new_node)
        queue.enqueue(new_node)
        return add_to_tree(queue)
        #add return !!!!!!!the problem is here!!!!!
def HuffmanTree_1(sorted_counts):
    queue=min_queue()
    for char,freq in sorted_counts:
        queue.enqueue(Node(char,freq))
    return add_to_tree(queue)
def HuffmanTree_2(sorted_counts,sorted_3,n):
    queue=min_queue()
    for char,freq in sorted_counts:
        queue.enqueue(Node(char,freq))
    for i in range(n):
        queue.enqueue(Node(sorted_3[i][0],sorted_3[i][1]))
    return add_to_tree(queue)
```

2.3 Huffman Dict Construction

After obtaining the Huffman tree, the next step is to find the corresponding encoding for each leaf by traversing the tree. A depth-first search (DFS) approach, similar to left-child-first, is used. The add_to_tree function is repeatedly called to explore the tree from top to bottom and from left to right. When traversing, a 0 is added to the encoding for a left child, and a 1 is added for a right child. This process continues until a node's left and right children are both NIL, indicating it is a leaf node. The traversal then backtracks layer by layer, eventually determining the encoding for each leaf.

Huffman Dict Construction Code

```
def find_leaf(node, code,HuffDict):
    if node is None:
        return HuffDict#{char:code}
    if node.get_left() is None and node.get_right() is None:
      # It's a leaf node
        HuffDict[node.get_char()] = code
        return HuffDict
    if node.get_left() is not None:
        HuffDict = find_leaf(node.get_left(), code + "0",
           HuffDict)#until found the lefest node
    if node.get_right() is not None:
        HuffDict = find_leaf(node.get_right(), code + "1",
           HuffDict)
    return HuffDict
def HuffmanDict(root):
    node=root
    HuffDict={}
    HuffDict=find_leaf(node,"",HuffDict)
    return HuffDict
```

2.4 Huffman Encoding

After obtaining the Huffman dictionary, the text can be encoded by matching each character with its corresponding entry in the dictionary. For character combinations, the next k characters in the text are checked to see if they exist as keys in the dictionary, and they are then encoded accordingly.

Huffman Encoding Code

```
def encode(file,HuffDict,k):
    encoded_text=""
    i=0
    while i<len(file):
        if file[i:i+k] in HuffDict:
            char=file[i:i+k]
            i+=k
        else:
            char=file[i]
            i+=1
        encoded_text+=HuffDict[char]
    return encoded_text
```

2.5 Optimal Huffman Solution

After refining the encoding process, the next step is to compare encoding lengths for different parameters in order to find the optimal solution. The Huffman tree for individual characters is used as a baseline, with its encoding length denoted as l_1, which is considered the shortest encoding by default. For different values of k (i.e., different combinations of characters), the top n-frequent substrings are added to the single-character Huffman tree to generate new encodings. All relevant information for each parameter is stored in the info dictionary for later use. By continuously comparing the lengths of encodings generated with different parameters to the shortest encoding length, the optimal encoding method is identified, and all related information is returned.

Optimal Solution Code

```
def compare(file):
    ZLE=1
    sorted_counts=count(file)
    root_1=HuffmanTree_1(sorted_counts)
    dic1=HuffmanDict(root_1)
    encoded_1=encode(file,dic1,0)
    l1=len(encoded_1)
    info={}
    for k in range(1,20):
        sorted_3=count_1(file,k)
        for n in range(1,50):
            root_2=HuffmanTree_2(sorted_counts,sorted_3,n)
            dic2=HuffmanDict(root_2)
            encoded_2=encode(file,dic2,k)
            l2=len(encoded_2)
            info[(k, n)] = [l2, encoded_2, dic2, root_2]
    L=l1
    for par,inf in info.items():
        if inf[0]<L:
            ZLE=0
            P=par
            L=inf[0]
            encoded_final=inf[1]
            D=inf[2]
            R=inf[3]
    if ZLE:
        print("The␣shortest␣code:␣n=k=0␣,",l1)
        return encoded_1,dic1,root_1
    else:
        print(f"The␣shortest␣code:␣k={P[0]},n={P[1]},",L)
        return encoded_final,D,R
```

2.6 Huffman Tree Visualization

After finding the optimal encoding method, the tree can be drawn based on the information from the optimal encoding. However, due to the large number of nodes and deep structure, it is difficult to represent the tree accurately using only a text-based format. Therefore, the tree structure is printed in a text format, while a rough visualization of the tree is generated using a plot.

Tree Description Code

```
def draw_tree(root, file_name):
    with open(file_name, "w", encoding="utf-8") as f:
        def calculate_depth(node):
            if node is None:
                return 0
            left_depth = calculate_depth(node.get_left())
            right_depth = calculate_depth(node.get_right())
            return max(left_depth, right_depth) + 1
        depth = calculate_depth(root)
        f.write(f"Tree■Depth:■{depth}\n")
        def print_tree(node, level, f):
            if node is None:
                return
            else:
                f.write(f"{'■'■*■level}{node.get_char()}:{
                   node.get_value()}(level■{level})\n")
                print_tree(node.get_left(), level + 1, f)
                print_tree(node.get_right(), level + 1, f)
        print_tree(root, 0, f)
```

Tree Painting Code

```
import matplotlib.pyplot as plt
def draw_binary_tree(root, file_name=None):
    fig, ax = plt.subplots(figsize=(12, 8))
    def plot_tree(node,x,y,dx,dy,level,ax):
        if node is None:
            return
        ax.text(x, y, f'{node.get_char()}:{node.get_value()}'
            , ha='center', va='center',
                fontsize=12, bbox=dict(facecolor='skyblue',
                    edgecolor='black', boxstyle="round,pad=0.5
                    "))
        if node.get_left() is not None:
            ax.plot([x, x - dx], [y, y - dy], color='black',
                lw=1)  # Line from parent to left child
            plot_tree(node.get_left(), x - dx, y - dy, dx /
                2, dy, level + 1, ax)
        if node.get_right() is not None:
            ax.plot([x, x + dx], [y, y - dy], color='black',
                lw=1)
            plot_tree(node.get_right(), x + dx, y - dy, dx /
                2, dy, level + 1, ax)
    ax.set_xlim(-60, 60)
    ax.set_ylim(-30, 0)
    #ax.axis('off')
    plot_tree(root, 0, 0, 30, 2, 0, ax)
    plt.savefig(file_name,bbox_inches='tight')
```

2.7 Huffman Decoding

The decoding process is the same regardless of the value of k. This is because, during traversal, it is possible to check if the next k characters in the sequence exist in the Huffman dictionary, and whether they exist or not does not affect the decoding.

Since all encoded characters are leaf nodes, the traversal from left to right (or from the root downward) will not cause any conflicts. When a match is found, decoding can proceed directly without ambiguity.

Decoding Code

```
def decode(encoded_text,HuffDict):
    decoded_text=""
    current_code=""
    for code in encoded_text:
        current_code+=code
        for char,char_code in HuffDict.items():
            if current_code==char_code:
                decoded_text+=char
                current_code=""
    return decoded_text
```

2.8 Main Function Implementation

Finally, in the main function, the above functionalities are linked together, and after obtaining the optimal solution, the results are printed to various txt files.

Main Function Code

```
def main():
    input_file='HUffmanCodes\MyOrigionalText.txt'
    file=parse_file(input_file)
    encoded_text,HuffDict,tree_root=compare(file)
    with open('HUffmanCodes/MyCodeWorkBk.txt', 'w',encoding='
       utf-8', errors='replace') as f:
        for char, code in HuffDict.items():
            f.write(f"{repr(char)}:■{code}\n")
        f.close()
    with open('HUffmanCodes\MyEncodingFile.txt','w') as f:
        f.write(encoded_text)
        f.close()
    # print(encoded_text)
    decoded_text=decode(encoded_text,HuffDict)
    with open('HUffmanCodes\MyDecodingFile.txt', 'w',
       encoding='utf-8', errors='replace') as f:
        f.write(decoded_text)
    # print(decoded_text)
    draw_tree(tree_root,"HUffmanCodes\MyHuffmanFile.txt")
    draw_binary_tree(tree_root, "HUffmanCodes/MyHuffmanTree.
       png")
if __name__ == "__main__":
    main()
```

3 Algorithm Analysis

In this analysis, I will not go through the code line by line, as that would be somewhat impractical. Instead, I will focus on analyzing the space complexity of the main data structures and the time complexity of the core algorithms.

I assume the length of the input text is N.

3.1 Space Complexity

The $file$ is a string that reads all characters from the input text, storing every character, so its space complexity is $\mathcal{O}(N)$.

The $counts$ dictionary stores each character and its corresponding frequency, so its space complexity is $\mathcal{O}(N)$. The $counts\_three$ dictionary also stores each character and the frequency of the character along with the following k consecutive characters, thus its space complexity is $\mathcal{O}(N-k+1)$, which simplifies to $\mathcal{O}(N)$.

If the space complexity of each $Node$ is considered as 1, then the $queue$ contains all the Nodes composed of characters and their corresponding frequencies, so the space complexity of the queue is also $\mathcal{O}(N)$.

The $dct1$ and $dic2$ store the Huffman codes for each character (character combination), so the space complexity is $\mathcal{O}(N)$.

The lengths of the encoded strings, $encoded\_1$ and $encoded\_2$, both depend on the length of the input text and the Huffman code lengths of the individual characters. Even in the worst-case scenario, the length of these encoded strings will not exceed the sum of the code lengths of all characters in the input text, hence the space complexity for the encoded strings is $\mathcal{O}(N)$. Similarly, the space complexity for the decoded string $decoded\_text$ is also $\mathcal{O}(N)$.

3.2 Time Complexity

The $count$ function traverses the entire input file, and its time complexity is $O(N)$. The $sorted()$ function sorts $counts.items()$, and its time complexity is $O(N \log N)$.

The add_to_tree function is a recursion. Because each recursive call involves an $enqueue$ operation on the min_queue, which has a time complexity of $O(N)$ due to the need to traverse the queue to find the proper position for insertion. As we know, the recursion follows the pattern $T(N) = N + T(N-1)$, with $T(1) = 1$, and there will be $N-1$ recursive calls. Therefore, the overall time complexity is $O(N^2)$.

The $find\_leaf$ function is also a recursion. The basic case returns immediately when it reaches a leaf node, which has a time complexity of $O(1)$ and its induction case makes two recursive calls, one for the left child and one for the right child of the current node if they exist. The time complexity of the $find\_leaf$ function depends on the height of the Huffman tree, as each recursive call corresponds to a move down one level of the tree. In the worst case, the tree is completely unbalanced and resembles a linked list, where the height of the tree is equal to the number of nodes, making the time complexity $O(N)$.

The $encode$ function uses a while loop to traverse all the characters. Since for each character, we need to check the subsequent k characters to determine if they form a valid substring in the

Huffman dictionary, the time complexity is $O(N * k)$.

The $decode$ function employs a for loop to iterate through the encoded string. During each iteration, it examines the Huffman dictionary to determine if a match exists and to perform the decoding. Given that the dictionary should contain approximately about $(26 + n)$ entries, where 26 represents the English alphabet and n accounts for additional symbols, the time complexity of this function is $O(N * n)$.

4 Discussions and Conclusions

I believe there are better ways to shorten the encoded text to reduce space. For instance, the range of values I've chosen for k and n might not be optimal and there could be better values for k and n. Alternatively, we could go beyond focusing on a single optimal k and consider integrating various letter combination lengths to find a more comprehensive and superior solution.

At the same time, I think there is room for improvement in my code as well. It could be made more concise, saving both space and time complexity. For example, since I wrote two versions of many functions with similar functionalities to obtain results for different combinations of letters, there is a lot of repetition in many operations, perhaps they could be merged using selection structures. During decoding, the constant traversal is very time-consuming; perhaps a reverse lookup dictionary could be created to simplify the decoding process.