INFORMATION THEORY PROJECT REPORT

Source Coding for Game of Thrones

Name: Zhang Licheng & Sun Binghui

UESTC ID: 2017200602011 & 2017200602023

UoG ID: 2357767 & 2357779

Date: 28. 11. 2019

Abstract

The purpose of this project is to encode English letters and the space in the English novel Game of Thrones. This experiment was performed by using Huffman coding technique and programming in Python. In this project, counting the frequency of each symbol, creating the Huffman tree, converting the text into binary code and building the executable file were all completed successfully. It concluded that Huffman coding could shorten the number of bits takes to encode a text effectively.

Preliminary

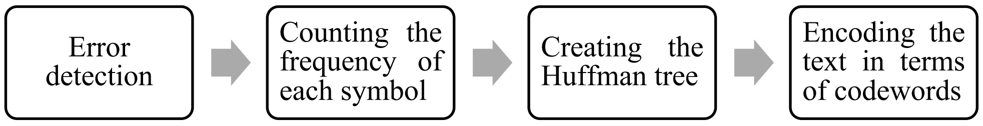
As a predominant coding method, Huffman coding is a way to encode information using variable-length strings to represent symbols depending on how frequently a particular symbol occurring. By making symbols which are used more frequently be shorter while symbols which appear more rarely be longer, it is proved that Huffman coding always shorten the number of bits takes to encode a given text effectively.

The Huffman Coding Steps:

1) List the frequency of each symbol;
2) Select two symbols with the lowest frequencies;
3) Create a binary tree out of these two symbols, labeling one branch with a "1" and the other with a "0";
4) Add the probabilities of the two symbols to obtain the probability of the new subtree;
5) Remove the symbols from the list and add the subtree to the list;
6) Go back through the list and take the two symbols/subtrees with the smallest probabilities and combine those into a new subtree;
7) Remove the original symbols/subtrees from the list, and add the new subtree to the list;
8) Repeat until all of them are combined.

Methodology

We divided the whole source coding process into four main stages as below.


Error detection
Counting the frequency of each symbol
Creating the Huffman tree
Encoding the text in terms of codewords

Stage1: Error detection

In our project, two kinds of common error are considered, Function *'try…except' is utilized to detect* FileNotFound Error and Permission Error.

Stage2: Counting the frequency of each symbol

Since each symbol maps to a particular ASCII code, so we set union in terms of ASCII codes to select symbols needed (26 letters and space) and utilized for loop to count the frequency of each symbol. Function 'findall' is used to match each string in the list of strings with chr(i).

Stage3: Creating the Huffman tree

Several classes and functions are used in the program to construct the Huffman code tree. They are *'TreeNode'*, *'createnodeQueue'*, *'addQueue'*, *'nodeQueue'*, *'freChar'*, *'createHuffmanTree'* and *'HuffmanCodeDic'*.

- Class *'TreeNode'* defines five built-in variables which represent the letter itself, frequency of occurrence, the 01 code of it, left child and right child during the encoded process;
- Function *'createnodeQueue'* uses *'for..in'* and *'. append'* to make the input list all Class *'nodeQueue'* form;
- Function *'addQueue'* adds the new merged tree nodes into the queue and sorts them;
- Class *'nodeQueue'* has two functions. Function *'addNode'* can add new tree node in sequence and function *'popNode'* deletes the smallest node in the tree;
- Function *'freChar'* inputs a two-dimensional list and returns to a new list which is rearranged according to the second dimension of the original one from big to small;

- Function *'createHuffmanTree'* is used to create Huffman binary tree and it makes a traversal while recording necessary information;
- Function *'HuffmanCodeDic'* is used to store information got from above.

Stage4: Encoding the text in terms of codewords

- Function *'TransEncode'* inputs the source novel and translates it into code from;
- Function *'PrintFunction'* prints the code form n words per row.

Result

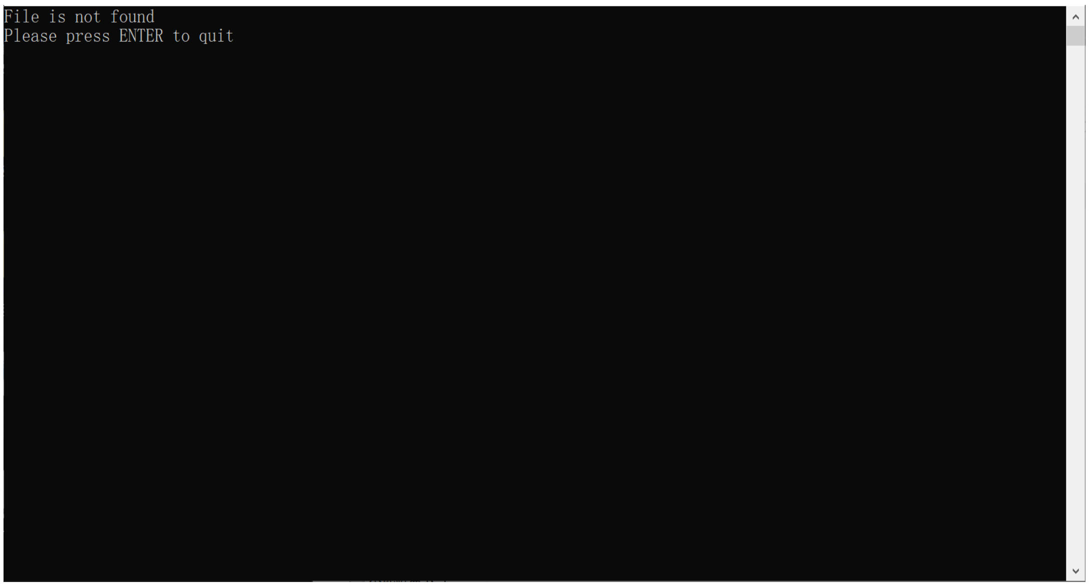
To complete this source coding project, we tried several programming platforms including MATLAB, Python and C++, and determined to utilize Python finally. As mentioned above, we divided the whole source coding process into four main parts: error detection, counting the frequency of each symbol, creating the Huffman tree, and encoding the text. After that, we programmed in terms of these four modules in Python, and the result is obtained as shown in *Table1*.

Symbol	Weight	Codeword	Symbol	Weight	Codeword
space	281965	111	**a**	97696	1001
A	3000	101110100	**b**	18740	110
B	1998	100100	**c**	21754	111
C	1455	1011100011	**d**	66066	11011
D	1794	1011101100	**e**	157340	10
E	978	10111011110	**f**	25057	101100
F	814	10111010110	**g**	26469	101111
G	1005	1001100	**h**	85523	111
H	3974	10001	**i**	69083	0
I	5104	10111	**j**	762	10111010101
J	1832	1011101101	**k**	14042	1100111
K	903	10111011100	**l**	52851	11000

L	2780	101110000	**m**	27948	110010
M	1959	100000	**n**	81760	110
N	1982	100001	**o**	94190	1000
O	934	10111011101	**p**	13911	1100110
P	759	10111010100	**q**	978	10111011111
Q	128	10111010111001	**r**	78829	11
R	2096	100101	**s**	78177	10
S	4774	10110	**t**	98608	1010
T	6094	10111001	**u**	30489	110100
U	186	1011101011101	**v**	8828	1010
V	512	101110101111	**w**	30936	110101
W	2436	100111	**x**	605	10011010
X	6	101110101110001	**y**	25269	101101
Y	1404	1011100010	**z**	609	10011011
Z	0	101110101110000			

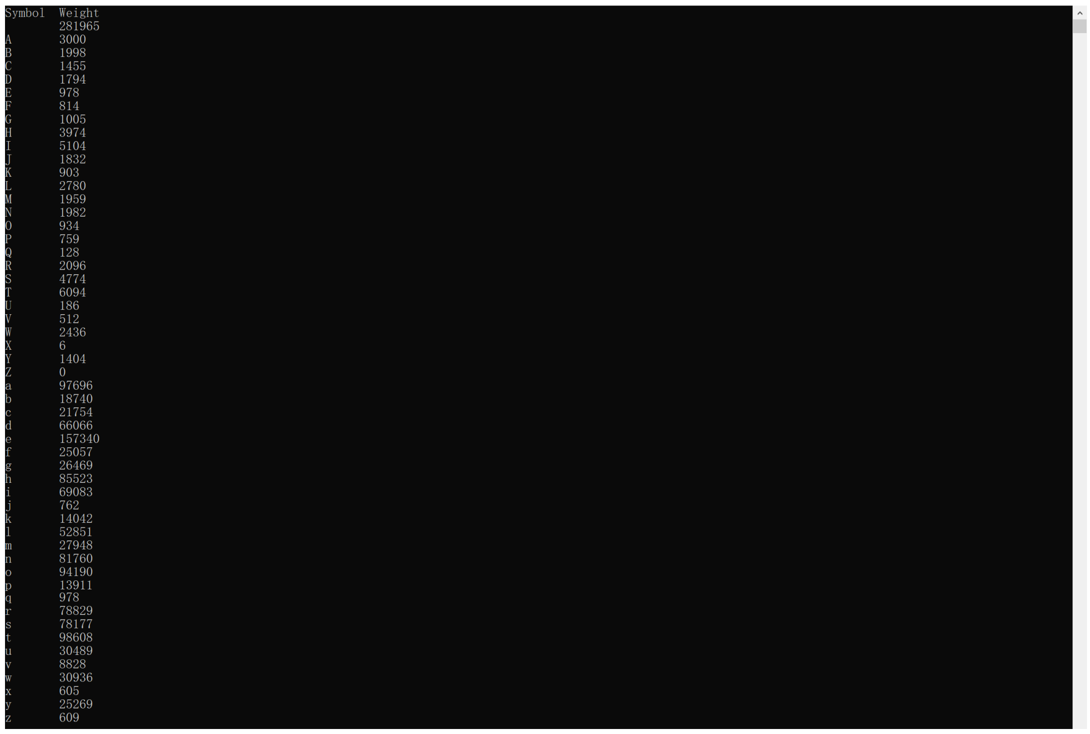
Table1. Result of Huffman Coding

The results in the executable file are shown as follows:



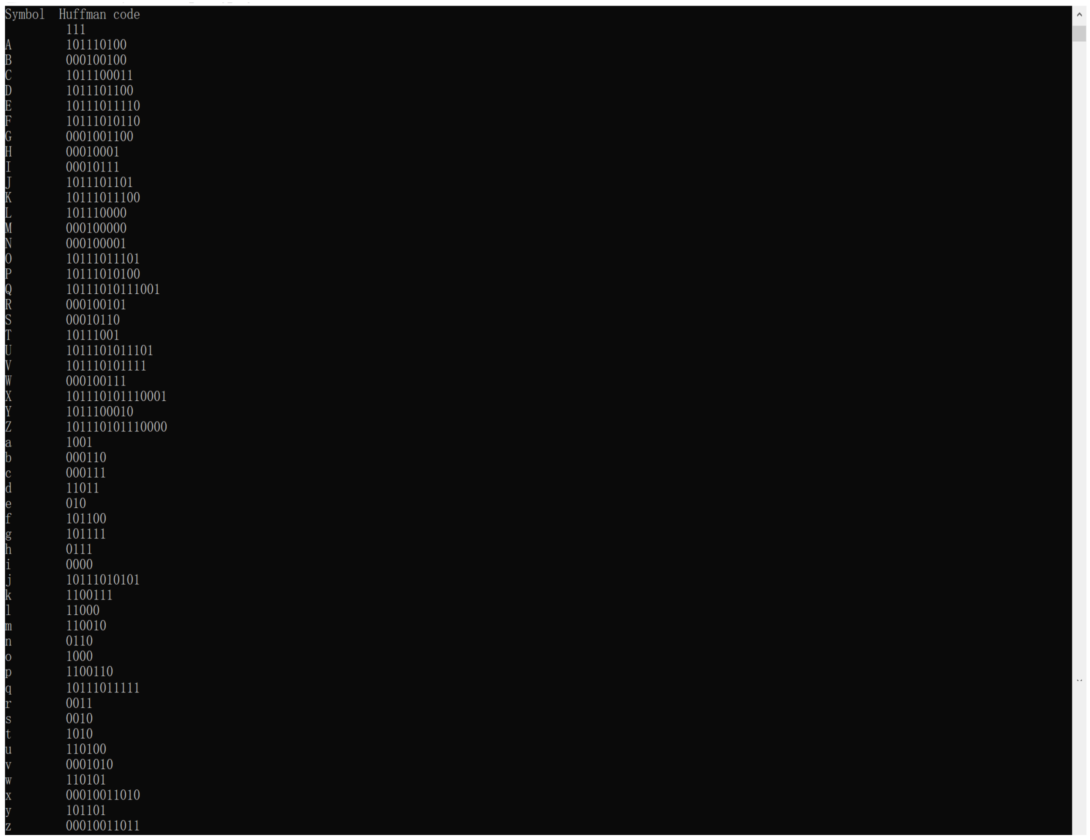
```
File is not found
Please press ENTER to quit
```

Figure1. Result of Error Detection in Executable File



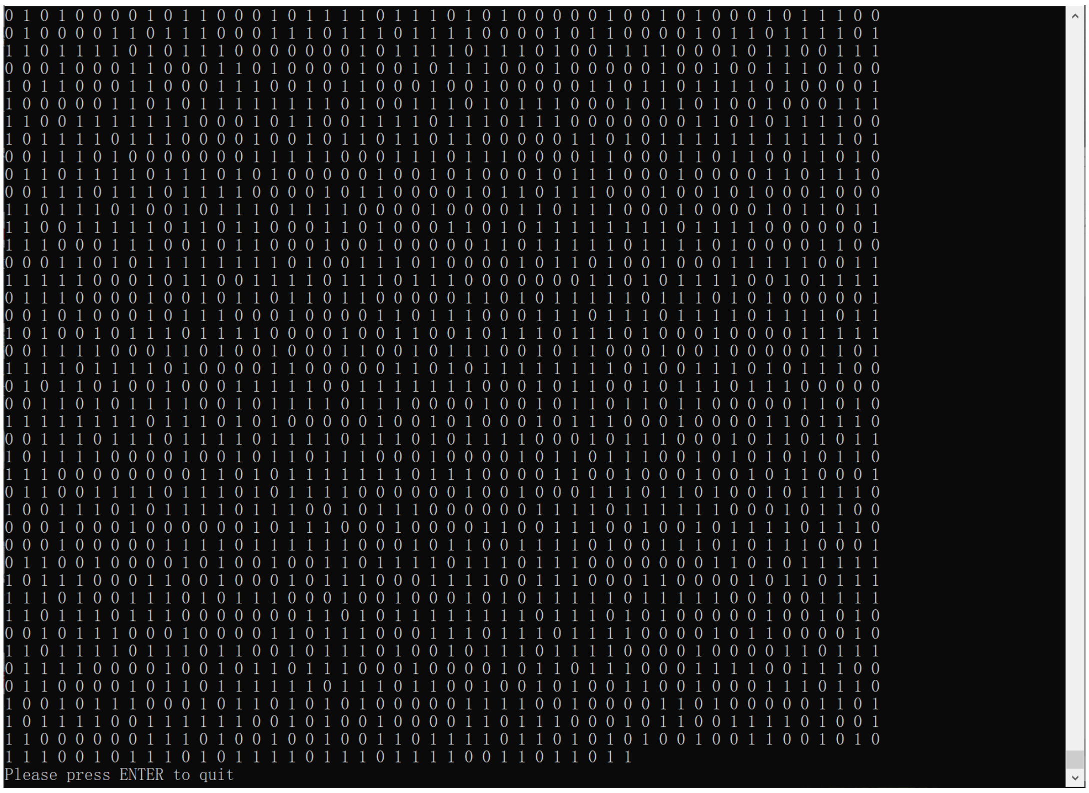
```
Symbol  Weight
        281965
A       3000
B       1998
C       1455
D       1794
E       978
F       814
G       1005
H       3974
I       5104
J       1832
K       903
L       2780
M       1959
N       1982
O       934
P       759
Q       128
R       2096
S       4774
T       6094
U       186
V       512
W       2436
X       6
Y       1404
Z       0
a       97696
b       18740
c       21754
d       66066
e       157340
f       25057
g       26469
h       85523
i       69083
j       762
k       14042
l       52851
m       27948
n       81760
o       94190
p       13911
q       978
r       78829
s       78177
t       98608
u       30489
v       8828
w       30936
x       605
y       25269
z       609
```

Figure2. Result of Weight Counting in Executable File



```
Symbol  Huffman code
         111
A        101110100
B        000100100
C        1011100011
D        1011101100
E        10111011110
F        10111010110
G        0001001100
H        00010001
I        00010111
J        1011101101
K        10111011100
L        101110000
M        000100000
N        000100001
O        10111011101
P        10111010100
Q        10111010111001
R        000100101
S        00010110
T        10111001
U        1011101011101
V        101110101111
W        000100111
X        101110101110001
Y        1011100010
Z        101110101110000
a        1001
b        000110
c        000111
d        11011
e        010
f        101100
g        101111
h        0111
i        0000
j        10111010101
k        1100111
l        11000
m        110010
n        0110
o        1000
p        1100110
q        10111011111
r        0011
s        0010
t        1010
u        110100
v        0001010
w        110101
x        00010011010
y        101101
z        00010011011
```

Figure3. Result of Huffman Coding in Executable File



```
0 1 0 1 0 0 0 0 1 0 1 1 0 0 0 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0
0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 1 0 1
1 1 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 0 0 0 1 0 1 1 1 1 0 1 1 1 0 1 0 0 1 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1
0 0 0 1 0 0 0 1 1 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 1 1 0 1 0 0
1 0 1 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 0 1 0 0 0 0 1
1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 0 1 0 1 1 0 1 0 0 1 0 0 0 1 1 1
1 1 0 0 1 1 1 1 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 0 0
1 0 1 1 1 1 0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1
0 0 1 1 1 0 1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 0 1 1 1 0 0 0 0 1 1 0 0 0 1 1 0 1 1 0 0 1 1 0 1 0
0 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0
0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0
1 1 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1
1 1 0 0 1 1 1 1 1 0 1 1 0 1 1 0 0 0 1 1 0 1 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0 0 1
1 1 1 0 0 0 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 0 0 0 0 1 1 0 0
0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 1 0 0 0 0 1 0 1 1 0 1 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1
1 1 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1
0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1
0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1
1 0 1 0 0 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 1 0 0 0 1 0 0 0 0 1 1 1 1 1
0 0 1 1 1 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 0 0 1 0 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 1 0 0 0 0 0 1 1 0 1
1 1 1 1 0 1 1 1 1 0 1 0 0 0 0 1 1 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0
0 1 0 1 1 0 1 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 1 0 1 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0
0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1 1 1 0 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 1 1 0 0 0 0 0 1 1 0 1 0
1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0
0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 0 0 0 1 0 1 1 0 1 0 1 1
1 0 1 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1 0 1 0 1 0 1 1 0
1 1 1 0 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 0 0 1 0 0 0 1 0 0 1 0 1 1 0 0 0 1
0 1 1 0 0 1 1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 1 1 0 1 1 0 1 0 0 1 0 1 1 1 1 0
1 0 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 0 1 1 0 0
0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 0 1 1 1 0 0 1 0 0 1 0 1 1 1 1 0 1 1 1 0
0 0 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 0 1
0 1 1 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1
1 0 1 1 1 0 0 0 1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1
1 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 0 0 1 0 0 1 0 0 0 1 0 1 0 1 1 1 1 1 0 1 1 1 1 1 0 0 1 0 0 1 1 1 1
1 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 0 1 1 1 0 1 0 1 0 0 0 0 0 1 0 0 1 0 1 0
0 0 1 0 1 1 1 0 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0
1 1 0 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 1 1 1 0 1 0 0 1 0 1 1 1 0 1 1 1 1 0 0 0 0 1 0 0 0 0 1 1 0 1 1 1
0 1 1 1 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 0 1 1 1 1 0 0 1 1 1 0 0
0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 1 1 1 0 1 1 1 0 1 1 0 0 1 0 0 1 0 1 0 0 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0
1 0 0 1 0 1 1 1 0 0 0 1 0 1 1 0 1 0 1 0 1 0 0 0 0 0 1 1 1 1 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 1 1 0 1
1 0 1 1 1 1 0 0 1 1 1 1 1 1 0 0 1 0 1 0 0 1 0 0 0 0 1 1 0 1 1 1 0 0 0 1 0 1 1 0 0 1 1 1 1 0 1 0 0 1
1 1 0 0 0 0 0 0 1 1 1 0 1 0 0 1 0 0 1 0 0 1 1 0 1 1 1 1 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 1 0 0 1 0 1 0
1 1 1 0 0 1 0 1 1 1 0 1 0 1 1 1 1 0 1 1 1 0 1 1 1 1 0 0 1 1 0 1 1 0 1 1
Please press ENTER to quit
```

Figure4. Partial Result of Text Encoding in Executable File

Conclusion

During this experiment, we have done further research about Huffman coding and became more familiar with the Python programming language. By utilizing a variety of functions and classes in Python, we completed counting the frequency of each symbol, creating the Huffman tree, and converting the text into binary code successfully. From our test result, it can be seen that Huffman coding technique shortens the transmission bit length effectively.

Reference

[1] Raymond W. Yeung, *Information Theory and Network Coding*, Springer, 2008;

[2] https://www.cnblogs.com/future-dream/p/10801934.html.

Appendix

The project code in Python:

```python
#Created by Zhang Licheng(2017200602011) & Sun Binghui(2017200602023)

import re

#Error detection
try:
    f1=open(r"A_Game_of_Thrones.txt",encoding='utf-8')
    s=f1.readlines()  # s is a list of strings
    f1.seek(0,0)
    f1.close()
except FileNotFoundError:
    print ("File is not found")
    input('Please press ENTER to quit')
except PermissionError:
    print("You don't have permission")
    input('Please press ENTER to quit')

a=range(97,123) #a-z
b=range(65,91) #A-Z
c=range(32,33) #space
d=set(a) | set(b) #union
e=set(d) | set(c) #union

char2=[]
count2=[]

print ("Symbol\tWeight")
for i in e:  # i is int, mapping to ASCII code of each symbol
    count = 0
    for y in s:
        l=re.findall(chr(i),y)  # Utilize findall function to match each string in the list of strings with chr(i)
        count += len(l)
    count2.append(count)
    char2.append(chr(i))
    print(chr(i),'\t%d'%count)

array=list(zip(char2,count2))


```

```python
# Create tree node
class TreeNode(object):
    def __init__(self, data):
        self.val = data[0]
        self.priority = data[1]
        self.leftChild = None
        self.rightChild = None
        self.code = ""

# Create node queue function
def createnodeQueue(codes):
    q = []
    for code in codes:
        q.append(TreeNode(code))
    return q

# Add node to the queue and create priority queue to make sure it is sorted from large to small
def addQueue(queue, nodeNew):
    if len(queue) == 0:
        return [nodeNew]
    for i in range(len(queue)):
        if queue[i].priority >= nodeNew.priority:
            return queue[:i] + [nodeNew] + queue[i:]
    return queue + [nodeNew]

# Define sequence
class nodeQueue(object):

    def __init__(self, code):
        self.que = createnodeQueue(code)
        self.size = len(self.que)

    def addNode(self, node):
        self.que = addQueue(self.que, node)
        self.size += 1

    def popNode(self):
        self.size -= 1
        return self.que.pop(0)

# Recieve the frequence of letters and space
def freChar(array):

 return sorted(array, key=lambda x: x[1])

# Create Huffman Tree
def createHuffmanTree(nodeQ):
    while nodeQ.size != 1:
        node1 = nodeQ.popNode()
        node2 = nodeQ.popNode()
        r = TreeNode([None, node1.priority + node2.priority])
        r.leftChild = node1
        r.rightChild = node2
        nodeQ.addNode(r)
    return nodeQ.popNode()

# Obtain Huffman Table in terms of Huffman Tree
def HuffmanCodeDic(head, x):
    global codeDic, codeList
    if head:
        HuffmanCodeDic(head.leftChild, x + '0')
        head.code += x
        if head.val:
            codeDic1[head.val] = head.code
        HuffmanCodeDic(head.rightChild, x + '1')

# Encode string
def TransEncode(string):
    m=0
    global codeDic1
    transcode = ""
    for ch in string:
        if ch in char2:
            transcode += codeDic1[ch]
    return transcode

# Print  200 words per row
def PrintFunction(results):
    n = 200  # print  200 words per row
    for q in range(len(results)):
        print(results[q], end=' ')
        if (q + 1) % n == 0:
            print(' ')

codeDic1 = {}

t = nodeQueue(freChar(array))
```

```python
tree = createHuffmanTree(t)
HuffmanCodeDic(tree, '')

list1= sorted(codeDic1.items(),key=lambda x:x[0])

print ("\n")
print ("Symbol\tHuffman code")

for head, x in list1:
    print(head,'\t',x)

print('The encoded novel is:')

test='a4 b+;;;'
list3=''.join(s)
PrintFunction(TransEncode(list3))
print(' ')
input('Please press ENTER to quit')
```