THE MINISTRY OF EDUCATION AND SCIENCE OF THE REPUBLIC OF KAZAKHSTAN

INTERNATIONAL INFORMATION TECHNOLOGIES UNIVERSITY

Faculty of Information technologies

Department of Computer Systems, Software Engineering and Telecommunications

Project report

Course: Information theory

Student: Nurtai Maksat

Group: CSSE-1605

Accepted : senior-lecturer L.A. Kozina

Almaty 2015

CONTENTS

INTRODUCTION

This project was written to learn how to use algorithms to transmit information through a noisy channel. In this project, the Huffman and Hamming algorithms were used.

In computer science and information theory, a Huffman code is a particular type of optimal prefix code that is commonly used for lossless data compression. The process of finding and/or using such a code proceeds by means of Huffman coding, an algorithm developed by David A. Huffman while he was a Sc.D. student at MIT, and published in the 1952 paper "A Method for the Construction of Minimum-Redundancy Codes".

In telecommunication, Hamming codes are a family of linear error-correcting codes. Hamming codes can detect up to two-bit errors or correct one-bit errors without detection of uncorrected errors. By contrast, the simple parity code cannot correct errors, and can detect only an odd number of bits in error. Hamming codes are perfect codes, that is, they achieve the highest possible rate for codes with their block length and minimum distance of three. Richard W. Hamming invented Hamming codes in 1950 as a way of automatically correcting errors introduced by punched card readers. In his original paper, Hamming elaborated his general idea, but specifically focused on the Hamming(7,4) code which adds three parity bits to four bits of data.

PROJECT IMPLEMENTATION

First, you need the data, that is, the text to work. We will retrieve this data from the txt file. Let me show you the structure of the project. It looks like this:

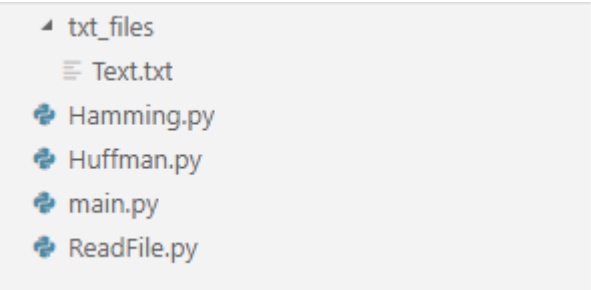

Figure 1. Project structure

The project was written in Python. The main file of my project is main.py. The very first step is to read the Text.txt file. To do this, the main.py file calls a method from the ReadFile.py file that reads the data from the file.

```
file = ReadFile('txt_files/Text.txt', analize=True)
```

Figure 2. From main.py 22 line

As you can see, we pass the “analize” parameter, so that after reading the file, it will automatically read the probabilities of the characters and write to the file.

```
def __read(self):
  try:
    read = open(self.file_name, 'r')
    if self.analize:
      self.__analize(read)
      self.__writeToFile()
  except:
    print("File Not Found!")

def __analize(self, read):
  for text in read:
    self.file_text = text
    for item in text:
      if item in self.data:
        self.data[item]['count'] += 1
      else:
        self.data[item] = {'count':1, 'probability':0}
      self.count += 1

def __writeToFile(self):
  writeFile = open('txt_files/Probability.txt', 'w')
  for key, value in sorted(self.data.items()):
    value['probability'] = round(value['count']/self.count, 10)
    if key == '\n':
      key = '\\n'
    string = key+"  -  Probability: "+str(value['probability'])+'\n'
    writeFile.write(string)
```

Figure 3. ReadFile.py

After launch, we will see it as output written in the file:

```
1    -  Probability: 0.1718538566
2   ,  -  Probability: 0.0094722598
3   -  -  Probability: 0.0027063599
4   .  -  Probability: 0.0094722598
5   1  -  Probability: 0.0027063599
6   5  -  Probability: 0.00135318
7   9  -  Probability: 0.00135318
8   A  -  Probability: 0.00135318
9   C  -  Probability: 0.00135318
10  D  -  Probability: 0.00135318
11  F  -  Probability: 0.0027063599
12  ...
```

Figure 4. Probability.txt

Huffman algorithm

The next thing we need to do is implement the Huffman code algorithm to take the prefix code at the heart of our probabilities. The idea behind the Huffman coding is based on the frequency of occurrence of a character in a sequence. The symbol that occurs in the sequence most often receives a new very small code, and the symbol that occurs least often receives, on the contrary, a very long code. This is necessary, since we want, when we have processed the entire input, the most frequency symbols occupy the least space, and the rarest - more. For this algorithm, you will need a minimal understanding of the structure of the binary tree and the priority queue. To build a tree, I used an array. The array was used in this way, with each iteration of the loop I added the first two elements to the array, sorted in ascending characters. After each iteration (adding to the array), I again sorted the list in ascending order. The array looked like this when the words "banana" were used:

```
[{'mark': [['n', 'b'], 'a'], 'probability': 1.0}]
```

The tree would look like this:

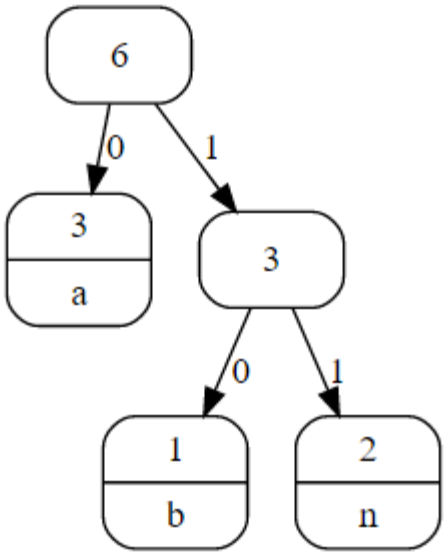

Figure 5. Example of Huffman tree

The following is the screenshots of the code:

```python
class Huffman:
    def __init__(self, dictionary):
        self.key = {}
        self.combinates = self.__sort(dictionary)
        self.__generateKey(self.__sort(dictionary))

    # Sort list by probabilities
    def __sort(self, dictionary):
        arr = []
        dictionary = dict(sorted(dictionary.items(), key=lambda x: x[1]['probability']))
        for key, value in dictionary.items():
            arr.append({ 'mark':key, 'probability':value['probability'] })
        for i in range(len(arr)):
            for j in range(len(arr)):
                if arr[i]['probability'] == arr[j]['probability']:
                    first = ord(arr[i]['mark'])
                    second = ord(arr[j]['mark'])
                    if first < second:
                        temp = arr[i]
                        arr[i] = arr[j]
                        arr[j] = temp
        return arr

    # Sort the list alphabetically when the probabilities are equal
    def __sortList(self, arr):
        arr = sorted(arr, key=lambda x: x['probability'])
        return arr

    def __setCode(self, mark, code):
        for item in mark:
            if type(item)==list:
                self.__setCode(item, code)
            else:
                if item not in self.key:
                    self.key[item] = code
                else:
                    self.key[item] = code + self.key[item]
```

```
39      def __generateKey(self, arr):
40          length = len(arr)-1
41          while length > 0:
42              first = arr.pop(0)
43              second = arr.pop(0)
44
45              if(first['probability'] <= second['probability']):
46                  self.__setCode(first['mark'], '0')
47                  self.__setCode(second['mark'], '1')
48              else:
49                  self.__setCode(first['mark'], '1')
50                  self.__setCode(second['mark'], '0')
51
52              temp = {
53                  'mark': [second['mark'], first['mark']],
54                  'probability': second['probability']+first['probability']
55              }
56              arr.append(temp)
57
58              arr = self.__sortList(arr)
59              length -= 1
60          writeFile = open('txt_files/Keys.txt', 'w')
61          for key, value in sorted(self.key.items()):
62              if key == '\n':
63                  key = '\\n'
64              string = key+"\t-\t"+value+"\n"
65              writeFile.write(string)
66
66
67      def encript(self, text):
68          encripted = ''
69          for item in text:
70              encripted += self.key[item]
71          return encripted
72
73      def decript(self, text):
74          decripted = ''
75          code = ''
76          for item in text:
77              code += item
78              if code in self.key.values():
79                  index = list(self.key.values()).index(code)
80                  decripted += list(self.key.keys())[index]
81                  code = ''
82          return decripted
83
```

Figure 6. Huffman.py

The result of the Huffman algorithm is recorded in the Keys.txt file.

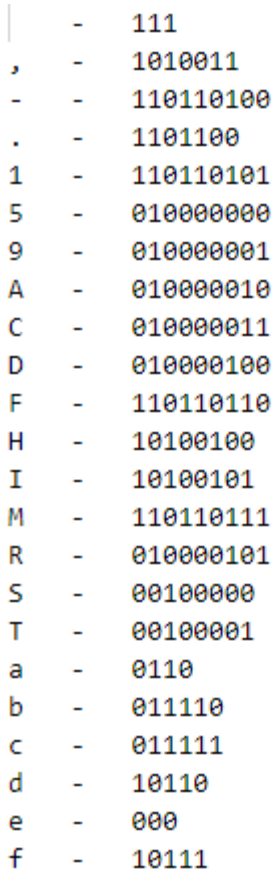

Figure 7. Keys.txt

Hamming algorithm

The second step is to use the Hamming algorithm. In coding theory, Hamming(7,4) is a linear error-correcting code that encodes four bits of data into seven bits by adding three parity bits. It is a member of a larger family of Hamming codes, but the term Hamming code often refers to this specific code that Richard W. Hamming introduced in 1950. At the time, Hamming worked at Bell Telephone Laboratories and was frustrated with the error-prone punched card reader, which is why he started working on error-correcting codes. The Hamming code adds three additional check bits to every four data bits of the message. Hamming's (7,4) algorithm can correct any single-bit error, or detect all single-bit and two-bit errors. In other words, the minimal Hamming distance between any two correct code words is 3, and received words can be correctly decoded if they are at a distance of at most one from the code word that was transmitted by the sender. This means that for transmission medium situations where burst errors do not occur, Hamming's (7,4) code is effective (as the medium would have to be extremely noisy for two out of seven bits to be flipped).

```
29    def encode(self, data):
30      data = self.__checkInputData(data)
31      length = len(data)
32      result = ''
33      i, j = 0, 4
34      while j<=length:
35        temp = data[i:j]
36        r1, r2, r3 = self.__xor(temp)
37        result += temp+r1+r2+r3
38        i, j = j, j+4
39      return result
40
41    def decode(self, data):
42      length = len(data)
43      result = ''
44      i, j = 0, 7
45      while j<=length:
46        temp = data[i:j]
47        rg1, rg2, rg3 = self.__xor(temp)
48        rc1, rc2, rc3 = temp[4], temp[5], temp[6]
49        s1 = str(int(rc1)^int(rg1))
50        s2 = str(int(rc2)^int(rg2))
51        s3 = str(int(rc3)^int(rg3))
52        error = self.errors[s1+s2+s3]
53        fix = str(int(not int(temp[error])))
54        temp = temp[:error]+fix+temp[error+1:]
55        result += temp[:4]
56        i, j = j, j+7
57      return result
```

Figure 8. Hamming. Encode and Decode functions

CONCLUSION

After the end of the project, as well as during its implementation, I learned a lot about these algorithms. Where to use them and where they are now often used. And also learned some information about the channel. The one that the channels are different and that when transmitting information noises are made. Subsequently, our information changes. If we talk about algorithms, then Huffman is used to compress information. And Hamming eliminates the possibility of changing information.

REFERENCES

1. https://youtu.be/dM6us854Jk0
2. https://en.wikipedia.org/wiki/Huffman_coding
3. https://www.geeksforgeeks.org/computer-network-hamming-code/
4. https://www.tutorialspoint.com/error-correcting-codes-hamming-codes