Data Communications : Huffman Coding, Convolutional Coding and Viterbi Algorithm

Mahsa Eskandari Ghadi
Student No. 810196597

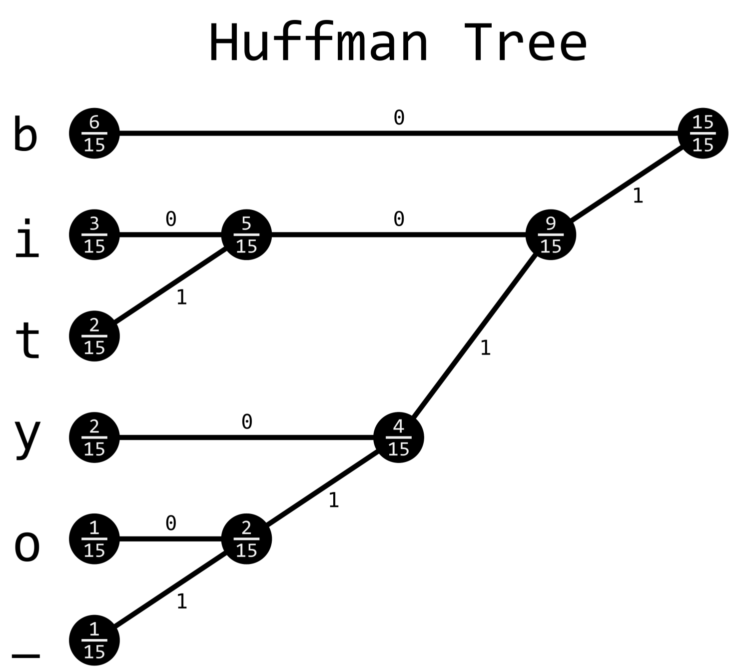
In this project we look at encoding algorithms such as Huffman and Convolutional coding and decoding algorithms such as Viterbi algorithm.

In [12]:

```
from scipy.io import loadmat
import heapq
import string
import numpy as np
import operator
import math
```

Source Coding : Huffman Coding

In Huffman Coding each alphabetic letter has a frequency which helps us determine the codeword for it, that which results in "the bigger the frequency the smaller the codeword" in this way we can have a smaller data in the end than having the same size for all or randomly assigning codewords.



In [13]:
```
alphabet = list(string.ascii_lowercase)
print("The alphabet is:", alphabet)
```

```
The alphabet is: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

In [14]:
```
table = loadmat('freq.mat')
frequencies = table['freq']
print("Frequencies: \n", frequencies)
```

```
Frequencies: 
 [[0.08167]
 [0.01492]
 [0.02782]
 [0.04253]
 [0.12702]
 [0.02228]
 [0.02015]
 [0.06094]
 [0.06966]
 [0.00153]
 [0.00772]
 [0.04025]
 [0.02406]
 [0.06749]
 [0.07507]
 [0.01929]
 [0.00095]
 [0.05987]
 [0.06327]
 [0.09056]
 [0.02758]
 [0.00978]
 [0.0236 ]
 [0.0015 ]
 [0.01947]
 [0.00102]]
```

As explained before the value of frequency makes a difference in choosing the codewords; this indicates **sorting**.
a **Heap** is a maximally efficient implementation of an abstract data type called a priority queue. Heap basically **sorts itself** no matter when or where you want to insert a or remove an item from it.

We define a HeapNode as shown below: for each alphabetic letter we have a node that has it's own frequency and as the Huffman code algorithm needs us to, we define 2 childs for each node; a left and a right. We want the heap to sort these nodes depending on their frequencies therefore a "greater than" function is defined to sort in that manner.

In [15]:

```
class HeapNode:

    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __gt__(self, other):
        return self.freq > other.freq
```

a Huffman Encoder is an object that has methods that can encode given an alphabetic string and decode given a numerical binary string via other methods which will be explained.

- make_dict simply makes a dictionary that uses letters as keys and frequencies as values.
- make_heap makes a heap depending on the frequencies assigned to each letter in the previous method.
- merge_nodes merges the 2 smallest frequencies and pushes the new node to the heap
- recursive_make_codes assigns 0 to the left edge and 1 to the right edge and does this recursively for all nodes.
- make_codes pops the root from the heap and calls recursive_make_codes for the first time.
  the encode method calls the above methods and is trivial.
  Huffman Encoder as an additional attribute **"reverse_mapping"** which is the opposite dictionary to encoding meaning the keys are the codewords and the values are the alphabetic letters.

In [16]:

```
class HuffmanEncoder:

    def __init__(self, input_text, frequencies):
        self.input_text = input_text
        self.frequencies = frequencies
        self.heap = []
        self.codes = {}
        self.reverse_mapping = {}

    def make_dict(self):
        return {k:v for k,v in zip(alphabet, frequencies)}

    def make_heap(self, freq_dict):
        for key in alphabet:
            node = HeapNode(key, freq_dict[key])
            heapq.heappush(self.heap, node)

    def merge_nodes(self):
        while(len(self.heap) > 1):
            node1 = heapq.heappop(self.heap)
            node2 = heapq.heappop(self.heap)

            merged = HeapNode(None, node1.freq + node2.freq)
            merged.left = node1
            merged.right = node2

            heapq.heappush(self.heap, merged)


    def recursive_make_codes(self, root, current_code):
        if(root == None):
            return

        if(root.char != None):
            self.codes[root.char] = current_code
            self.reverse_mapping[current_code] = root.char
            return

        self.recursive_make_codes(root.left, current_code + "0")
        self.recursive_make_codes(root.right, current_code + "1")


    def make_codes(self):
        root = heapq.heappop(self.heap)
        current_code = ""
        self.recursive_make_codes(root, current_code)

    def get_encoded(self, text):
        encoded_text = ""
        for character in text:
            encoded_text += self.codes[character]
        return encoded_text

    def encode(self):
        freq_dict = self.make_dict()
        self.make_heap(freq_dict)
```

```python
        self.merge_nodes()
        self.make_codes()
        self.encoded_text = self.get_encoded(self.input_text)
        return self.encoded_text

    def decode(self):
        current_code = ""
        decoded_text = ""

        for bit in self.encoded_text:
            current_code += bit
            if(current_code in self.reverse_mapping):
                character = self.reverse_mapping[current_code]
                decoded_text += character
                current_code = ""

        return decoded_text

```

[https://bhrigu.me/blog/2017/01/17/huffman-coding-python-implementation/ (https://bhrigu.me/blog/2017/01/17/huffman-coding-python-implementation/)](https://bhrigu.me/blog/2017/01/17/huffman-coding-python-implementation/)

In [17]:

```python
huffman_encoder = HuffmanEncoder("mahsaeskandari", frequencies)
huffman_encoder.encode()
```

Out[17]: '001111110011001111110100011100101111110101011111111001011011'

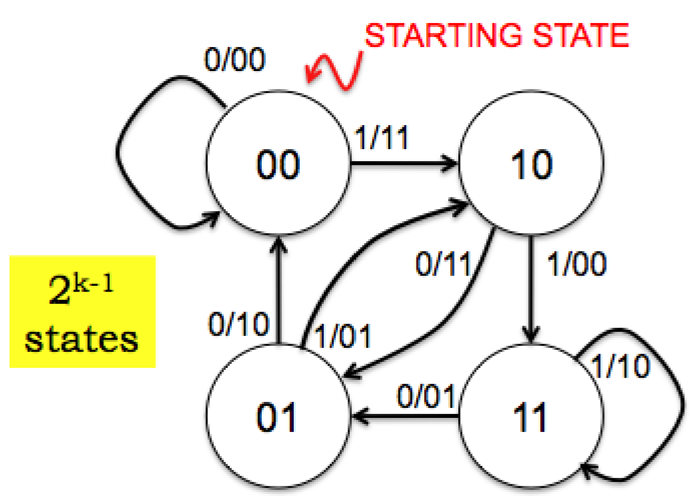
In [18]:

```python
huffman_encoder.decode()
```

Out[18]: 'mahsaeskandari'

Channel Coding : Convolutional Coding

Convolutional encoder on the other hand uses a state machine to encode a binary string. Figure Below:



To implement this state machine we view each state as a function where depending on the state (function) different actions are taken and to move from one state to another the current function simply calls another function. the starting state is called by the encode method to start off the state machine. Each time a state function is called a bit is read from the input bits and removed from the original then an action is taken depending on that bit and the next state is called. Once the string is empty the function at that moment returns.

In [30]:

```
class ConvolutionalEncoder:

    def __init__(self, input_bits):
        self.input_bits = list(input_bits)
        self.encoded = []

    def zerozero(self):
        if len(self.input_bits) == 0:
            return

        bit = self.input_bits.pop(0)
        if bit == '0':
            self.encoded.append('00')
            self.zerozero()
        else:
            self.encoded.append('11')
            self.onezero()

    def onezero(self):
        if len(self.input_bits) == 0:
            return
        bit = self.input_bits.pop(0)

        if bit == '0':
            self.encoded.append('11')
            self.zeroone()
        else:
            self.encoded.append('00')
            self.oneone()


    def oneone(self):
        if len(self.input_bits) == 0:
            return

        bit = self.input_bits.pop(0)
        if bit == '0':
            self.encoded.append('01')
            self.zeroone()
        else:
            self.encoded.append('10')
            self.oneone()

    def zeroone(self):
        if len(self.input_bits) == 0:
            return

        bit = self.input_bits.pop(0)
        if bit == '0':
            self.encoded.append('10')
            self.zerozero()
        else:
            self.encoded.append('01')
            self.onezero()

    def encode(self):
```

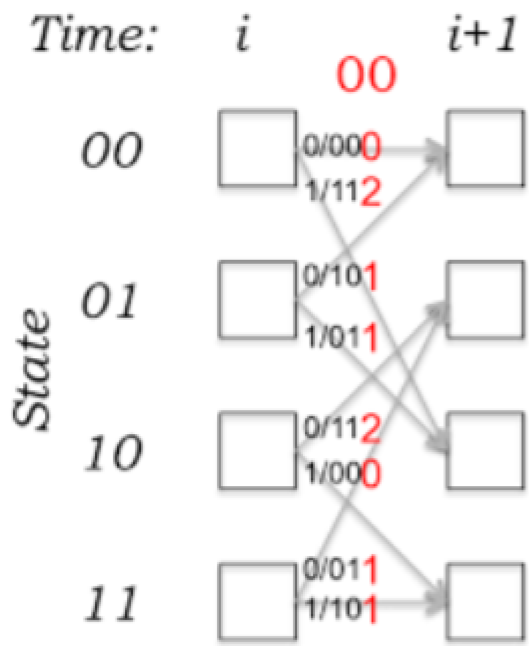
```python
        self.zerozero()
        return ''.join(self.encoded)
```

In [37]:
```python
convolutional_encoder = ConvolutionalEncoder('100111')
convolutional_encoder.encode()
```

Out[37]: '111110110010'

Viterbi Algorithm

To implement the Viterbi Algorithm we need a Trellis diagram. Each Trellis node has 2 paths for each **parent**. Considering we're in a **to_state** we will need the **from_state** for it's **path metric** and the weight of the edge for **branch metric** to determine which way is the most likely way that has been taken when encoding so that we can trace that back and decode the encoded thing:D



So by path1 and path2 I mean a list of, the code on the edge (the 11 in 0/11), the **Hamming Distance** of the code on the edge and the received bits (or the branch metric), the number of the from_state, the decoded bit (the 0 in 0/11) respectively for each from_state. And that's all we need.

In [38]:
```python
class TrellisNode:

    def __init__(self, path1, path2):
        self.path1 = path1
        self.path2 = path2
```

We're holding 4 nodes at a time since Viterbi is a **Dynamic Programming** algorithm; We only need the current nodes to find the next ones.

- hamming is a method that given two lists of bits determines the hamming distance of a the lists for example for [1,0] and [0,1] the hamming distance is '2'. It will come in handy later.

- calculate_branch_metrics takes the first two bits of the received data then removes those two bits from the original received data so that once it's empty the decoding is over. After this it calculates the hamming distance between them and saves that to the paths explained before (again for each path of each node)
- When branch metrics are known we can move on to calculating the path metrics via the calculate_path_metrics method; which calculates the path metrics of the new nodes according to the equation below:

  $$PM[s, i+1] = min(PM[a, i] + BM[a \rightarrow s], PM[b, i] + BM[b \rightarrow s])$$

  Everytime we reach a new PM list the min of that is declared to be the most likely state and the path is saved.
- the decode method simply calls the above methods in order.

In [33]:

```
class ViterbiDecoder:

    def __init__(self, encoded):
        self.encoded = encoded
        #[code, hd, from_state, decoded]
        self.nodes = [TrellisNode(['00', 0, 0, 0], ['10', 0, 1, 0]), Tre
        self.PMs = [0, 0, 0, 0]
        self.path = []
        self.res_path = []

        # compute hamming distance of two bit sequences
    def hamming(self, s1, s2):
        return sum(map(operator.xor,s1,s2)) #cool right?

    def calculate_branch_metrics(self):

        encoded_bits = list(self.encoded[:2])
        self.encoded = self.encoded[2:]

        for i, bit in enumerate(encoded_bits):
            encoded_bits[i] = int(bit)

        for node in self.nodes:
            edgebits = list(node.path1[0])

            for i, bit in enumerate(edgebits):
                edgebits[i] = int(bit)

            node.path1[1] = self.hamming(encoded_bits, edgebits)

            edgebits = list(node.path2[0])
            for i, bit in enumerate(edgebits):
                edgebits[i] = int(bit)

            node.path2[1] = self.hamming(encoded_bits, edgebits)

    def calculate_path_metrics(self):
        #[code, hd, from_state, decoded]
        newPMs = [0, 0, 0, 0]
        for i, node in enumerate(self.nodes):
            values = [self.PMs[node.path1[2]] + node.path1[1], self.PMs[

            newPMs[i] = min(values)
            if values.index(min(values)) == 0:
                self.path.append(node.path1[3])
            else:
                self.path.append(node.path2[3])
        self.PMs = newPMs

    def viterbi_step(self):

        most_likely_state = min(self.PMs)
        self.res_path.append(self.path[self.PMs.index(most_likely_state)

    def decode(self):

```

```python
        while self.encoded:
            self.calculate_branch_metrics()
            self.calculate_path_metrics()
            self.viterbi_step()

        for i, path in enumerate(self.res_path):
            self.res_path[i] = str(path)
        return ''.join(self.res_path)
```

In [39]:

```python
viterbi_decoder = ViterbiDecoder('111110110010')
viterbi_decoder.decode()
```

Out[39]: '000111'

To capture reality better it is likely for a noise to influence the output of channel encoding. Putting it all together we have the results shown as below:

In [35]:

```python
import noise
```

In [36]:

```python
huffman_encoder = HuffmanEncoder("mahsaeskandari", frequencies)
source_encoded = huffman_encoder.encode()

convolutional_encoder = ConvolutionalEncoder(source_encoded)
channel_encoded = convolutional_encoder.encode()
channel_encoded = list(channel_encoded)

for i, bit in enumerate(channel_encoded):
    channel_encoded[i] = int(bit)

noised = noise.noise(channel_encoded)

for i, bit in enumerate(noised):
    noised[i] = str(bit)

noised = ''.join(noised)

viterbi_decoder = ViterbiDecoder(noised)
viterbi_decoder.decode()
```

Out[36]: '001001000011001001000100001001001000000001011100111001011011'

In [ ]: