Huffman-based Compression and Decompression Scheme in Python

By Marc Stern
MS in Data Science at Montclair State University

August 5, 2020

Introduction

Text compression is a way of converting some form of text to binary, or vice-versa. It's application is to reduce the bandwidth of a text document for faster digital transmission. Now, text compression is not directly converting the text to binary, but rather using its process to store text characters, such that they are accessible by some binary sequence. This is the compression scheme, while the decompression scheme works the opposite way, converting binary to text. One method of accomplishing this task is to use the Huffman coding scheme. Its popularity comes from the dynamic sizing of the binary representation of character, ultimately saving memory space. This is different from Unicode for example, whose binary representation is a fixed number of bits. Next, we'll look at my design and implementation of a Huffman coding scheme.

Design

The design of my Huffman-coding scheme is a Class, whose methods include *encode*, *compression*, and *decompression*. First, let's look at the *encode* method since this is the first step towards accomplishing our goal. When a text string is passed in as the argument while calling *encode*, a dictionary is created to count each instance of each character in the text string. Once the dictionary is finalized, its keys will be a character from the string, and its value will be the amount of times that character showed up in the string. This will create a hierarchy for our next step, an implementation of a heap priority queue.

What we want to add into our queue are a tuple, containing a pair of items: the character's frequency, and a binary tree, consisting of one element: the character itself. Once we've created this priority queue, we want to update it, such that each tree, starting with those that have characters of the lowest frequency, is combined with another tree, creating a new tree whose leaves are characters with a mutual parent whose node consists of the sum of the frequency of its leaves. Once this new tree is created, we add it back into the queue, again as a tuple, containing the sum of the frequencies of the characters, and the new tree. We continue this process until there is one element left in the queue, will be a binary tree containing

characters as leaves, and parents that are sums of character frequencies. Our root should be the total number of characters counted in the entire text string.

Now that we have our sorted binary tree, we can traverse the tree to create a binary value for each character. At each step either a 0 or 1 is appended to a list, depending on if a child is its parent's left or right child, respectively. Once this process is complete, because we store the values in a list that always append the right of the list, we reverse the list order, and now we have our binary values. The final step is to add these values to a dictionary. We use two dictionaries, one whose keys are characters, and values are binary, and another which is the reverse.

The *compression* and *decompression* methods use these dictionaries to read a text or binary string, and convert them to either binary or text. It does this by reading a character or binary string, and checks its value in its respective dictionary, and appends to a list. After iterating through the text or binary string, it rewrites in string format, making it legible, and returns the output.

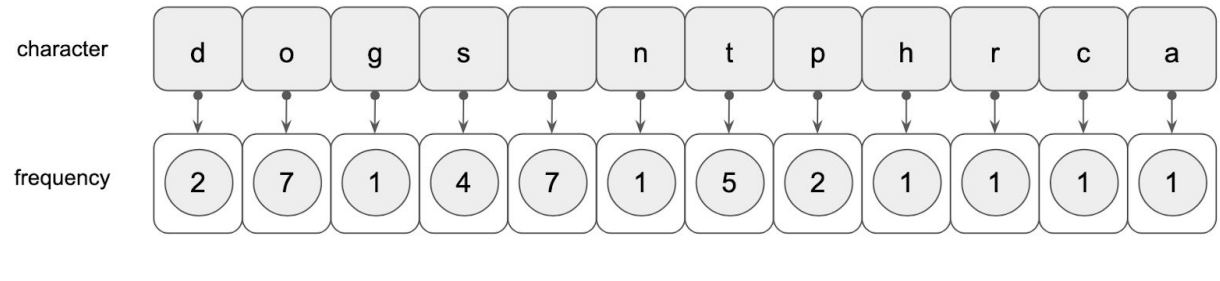
Implementation

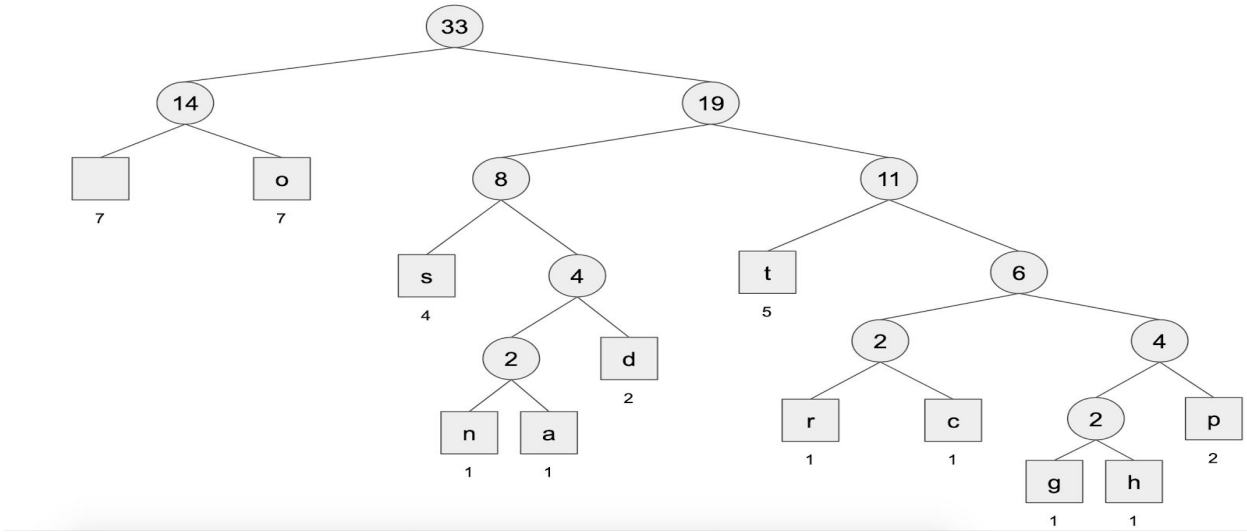
```
from ch08.linked_binary_tree import LinkedBinaryTree
from ch09.heap_priority_queue import HeapPriorityQueue

class Huffman:

    def __init__(self):
        self.text_to_binary = {}
        self.binary_to_text = {}

    def encode(self, s):
        map = {}
        for c in s:
            if not c in map:
                map[c] = 1
            else:
                map[c] += 1

        q = HeapPriorityQueue()

        for key in map:
            t = LinkedBinaryTree()
            r = t._add_root(key)
            q.add(map[key], t)

        while len(q) > 1:
            t1 = q.remove_min()
            t2 = q.remove_min()

            nt = LinkedBinaryTree()
            f = t1[0] + t2[0]

            root = nt._add_root(f)
            nt._attach(root, t1[1], t2[1])

            q.add(f, nt)

        tree = q.remove_min()

        for i in tree[1].preorder():

            if i._node._parent is not None:

                if i._node._left is None and i._node._right is None:
                    l = []
                    me = i._node
                    p = me._parent

                    while p is not None:
                        if me is p._left:
                            l.append('0')
                        elif me is p._right:
                            l.append('1')
                        me, p = p, p._parent

                    lrev = reversed(l)
                    binary = ''.join(lrev)

                    self.text_to_binary[i.element()] = binary
                    self.binary_to_text[binary] = i.element()
```

```
def compression(self, s):
    l = list(s)
    nl = []
    for c in l:
        binary = self.text_to_binary[c]
        nl.append(binary)
    t = ' '.join(nl)
    return t

def decompression(self, s):
    l = s.split()
    nl = []
    for binary in l:
        c = self.binary_to_text[binary]
        nl.append(c)
    t = ''.join(nl)
    return t
```

Visual Representation

The following visual representations are, respectively, the map of character frequency, and the binary tree that are created from the *encode* method. It tells us that the test string “dogs do not spot hot pots or cats” has 12 characters with varying frequencies, and where they live in the tree, such that we can follow the tree and understand how the binary code is assigned.





Testing

The visual representations can help us test our implementation. We can check to see if our encode method is working correctly by comparing an arbitrary value's binary representation, to what our expectation is. For example, the character *o* should have the binary representation of *01*. This is because from the root of the tree, to get to *o*, we move to the left child, and then to the right child. The character *g* will have a longer binary representation, and should be *111100*.

We can test this by using the following code:

```
mytext = 'dogs do not spot hot pots or cats'
h = Huffman()
e = h.encode(mytext)
c = h.compression(mytext)
d = h.decompression(c)
print(d)
print(c)
```

Our results yield the following, printing decompression first, then compression:

dogs do not spot hot pots or cats

1011 01 111100 100 00 1011 01 00 10100 01 110 00 100 11111 01 110 00 111101 01 110 00 11111 01 110 100 00 01 11100 00 11101 10101 110 100

Checking for the letters *o* and *g*, we see their binary values are *01* and *111100* respectively, just as we expected. This shows us that our program runs as expected.