Huffman Coding

Darin Critchlow
CSIS 2430-001

Objective:

Implement Huffman Coding

1. **Use this file: Metamorphosis.txt**
2. **Convert the file to binary and measure the size.**
3. **Implement Huffman Coding to compress the file. (You will calculate the frequencies of all punctuations, etc., build a tree, then use the tree to compress the file (remember, it will be in BINARY - LEAVE IT in this format.)**
4. **Measure the newly compressed file.**
5. **Answer the question: What is the delta shrinkage (Entropy) from the first file to the second?**
6. **Use the frequencies found on page 771 #27 and reimplement.**
7. **Measure the compression again.**
8. **Compare/contrast (what did you find?, why was there much of a difference?, etc.)**

What Worked:

Everything worked properly

What Didn't Work:

Everything worked properly

Comments:

The first Huffman code resulted in a 46.28% compression using the binary code taken from the frequency of letters in the Metamorphosis document. When I started the first part of this assignment I was sure that the compression would be much greater than just using the frequencies in the second portion. As I was coding and looking at the binary codes that it produced I was pleased with how many bits were shaved off from the original.

When I was researching Huffman code I realized that the less characters you have in your input the compression is much higher. As I was pondering this I was trying to figure out how this would in fact be possible. With the first part of the assignment I had covered everything from spaces, carriage returns, tabs, etc. How would it be possible that less characters would give greater compression?

I started to code the final portion of this assignment and as I was working through the code and looking at my print statements as I was forming the Huffman code there definitely was less characters and bits. Even though there were less bits in the Huffman code in the second part it did not take any bits off of the carriage returns, tabs, spaces, etc ...

I calculated the size of the second converted file by counting how many alpha characters that had Huffman codes associated with it and also how many non-alphanumeric characters there were. I then multiplied the non-alphanumeric characters by 8 bits and then added that to the number of alpha characters. The resulting compression was good, 37.24%, but not quite as high as the first portion of the assignment.

The non-alphanumeric codes that did not get converted was the difference and it was interesting to see how much they would actually play a role. This exercise was nice to be able to go through some other code and research different ways to form these data structures in Python.

Code:

```
#!/usr/bin/python
from __future__ import division
from itertools import groupby
from heapq import heapify, heappop, heappush

# Frequencies from pg 771 #27
frequencies = [('A', 0.0817), ('B', 0.0145), ('C', 0.0248),
               ('D', 0.0431), ('E', 0.1232), ('F', 0.0209),
               ('G', 0.0182), ('H', 0.0668), ('I', 0.0689),
               ('J', 0.0010), ('K', 0.0080), ('L', 0.0397),
               ('M', 0.0277), ('N', 0.0662), ('O', 0.0781),
               ('P', 0.0156), ('Q', 0.0009), ('R', 0.0572),
               ('S', 0.0628), ('T', 0.0905), ('U', 0.0304),
               ('V', 0.0102), ('W', 0.0264), ('X', 0.0015),
               ('Y', 0.0211), ('Z', 0.0005)]

class Node(object): #[1]
    """
    Tree nodes are simple objects, but they need to keep track of four
    things: Which item they store (if any), the combined weight of them
    and their children, and their left and right child nodes.
    """
    left = None
    right = None
    item = None
    weight = 0

    def __init__(self, i, w):
        self.item = i
        self.weight = w

    def setChildren(self, ln, rn):
        self.left = ln
        self.right = rn

    def __repr__(self):
        """Print out the status of the nodes to debug the tree"""
        return "%s - %s -- %s _ %s"\
            % (self.item, self.weight, self.left, self.right)

    def __cmp__(self, a):
        """Use the heapq module to order the nodes"""
        return cmp(self.weight, a.weight)
```

[1] TechRepublic. Huffman coding in Python. [Online] Available from: http://www.techrepublic.com/article/huffman-coding-in-python/ [Accessed 7 Apr 2014].

```
def huffman(input): #[1]
    """Use the groupby function of the itertools module to calculate the
    original weights, then use a heapq priority queue to rank the nodes"""
    itemqueue =  [Node(a,len(list(b))) for a,b in groupby(sorted(input))]
    heapify(itemqueue)
    """At the end of this step, itemqueue has only one element,
    the root node of the tree."""
    while len(itemqueue) > 1:
        l = heappop(itemqueue)
        r = heappop(itemqueue)
        n = Node(None, r.weight+l.weight)
        n.setChildren(l,r)
        heappush(itemqueue, n)

    codes = {}

    def code_it(s, node):
        """
        Walk through the tree to work out the encoding for each item.
        Traverse the whole tree in one go and store the results in a
        dictionary
        """
        if node.item:
            if not s:
                codes[node.item] = "0"
            else:
                codes[node.item] = s
        else:
            code_it(s+"0", node.left)
            code_it(s+"1", node.right)

    # Recursive function to accumulate the encoding as we walk down
    # the tree and branch at each non-leaf node
    code_it("",itemqueue[0])

    return codes, "".join([codes[a] for a in input])

def my_huffman(mylist, input):
    """
    Take the frequencies of letters in typical English text and build a
    Huffman code
    """
    itemqueue = [Node(a,b) for a,b in sorted(frequencies,
        key=lambda frequencies: frequencies[1])]

    while len(itemqueue) > 1:
```

[1] TechRepublic. Huffman coding in Python. [Online] Available from: http://www.techrepublic.com/article/huffman-coding-in-python/ [Accessed 7 Apr 2014].

```
        l = heappop(itemqueue)
        r = heappop(itemqueue)
        n = Node(None, r.weight+l.weight)
        n.setChildren(l,r)
        heappush(itemqueue, n)

    codes = {}

    def code_it(s, node):
        """
        Walk through the tree to work out the encoding for each item.
        Traverse the whole tree in one go and store the results in a
        dictionary
        """
        if node.item:
            if not s:
                codes[node.item] = "0"
            else:
                codes[node.item] = s
        else:
            code_it(s+"0", node.left)
            code_it(s+"1", node.right)

    # Recursive function to accumulate the encoding as we walk down
    # the tree and branch at each non-leaf node
    code_it("",itemqueue[0])

    return codes
def huffman_text(codes, data):
    huff_text = []
    counter = 0
    counter_alpha = 0
    for i in data.upper():
        if i.isalpha():
            huff_text.append(huff_2.get(i))
            counter_alpha += len(huff_2.get(i))
        else:
            counter += 1
            huff_text.append(i)
    return "".join(huff_text), counter, counter_alpha

if __name__ == "__main__":
    print 'Build a Huffman code based upon the frequency of characters',
    print 'in the Metamorphosis.txt file','\n'
    with open('Metamorphosis.bin', 'rb') as f:
        data = f.read()
```

[1] TechRepublic. Huffman coding in Python. [Online] Available from: http://www.techrepublic.com/article/huffman-coding-in-python/ [Accessed 7 Apr 2014].

```
    huff = huffman(data)
    with open('huffman.bin', 'wb')as f2:
        f2.write(huff[1])
    with open('huffman.bin', 'rb')as f3:
        myfile = f3.read()
    original_byte_length = len(data) * 8
    myfile_byte_length = len(myfile)
    percent = myfile_byte_length / original_byte_length

    print 'myfile\n>> %d bytes' %(myfile_byte_length),\
        '\nOriginal file\n>> %d bytes' %(original_byte_length),\
        '\nCompression of %.2f%%' %(100 - round(percent * 100, 2))

    print

    huff_2 = my_huffman(frequencies, data.upper())
    english_huff = huffman_text(huff_2, data)
    with open('huffman2.bin', 'wb')as f2:
        f2.write(english_huff[0])
    second_file_byte_length = (english_huff[1] * 8) + english_huff[2]
    percent = second_file_byte_length / original_byte_length
    print 'second file\n>> %d bytes' %(second_file_byte_length),\
        '\nOriginal file\n>> %d bytes' %(original_byte_length),\
        '\nCompression of %.2f%%' %(100 - round(percent * 100, 2))
```

[1] TechRepublic. Huffman coding in Python. [Online] Available from: http://www.techrepublic.com/article/huffman-coding-in-python/ [Accessed 7 Apr 2014].

```
[darin@darin-HP:~/Documents/CSIS2430Spring2014/Assignments/Huffman_Coding]$./huffman.py
Build a Huffman code based upon the frequency of characters in the Metamorphosis.txt file

myfile
>> 513413 bytes
Original file
>> 955656 bytes
Compression of 46.28%

second file
>> 599772 bytes
Original file
>> 955656 bytes
Compression of 37.24%
```