CS 3430: S22: Scientific Computing
Assignment 12
Encoding and Decoding with Huffman Trees

Vladimir Kulyukin
Department of Computer Science
Utah State University

April 16, 2022

1 Learning Objectives

1. Encoding
2. Decoding
3. Huffman Trees
4. Variable-Length Codes

Introduction

In this assignment, we'll learn how to do variable-length encoding and decoding with Huffman trees. Recall that the objective of encoding is to take a code map and assign each symbol (e.g., a character or a number) in a message a specific code (i.e., a bit string). I will use the terms *symbol* and *character* synonymously in this assignment. The former is more generic than the latter in that it includes numbers, characters, pictograms, etc. One of the best known encoding algorithms is the Morse Code. There are two types of codes: fixed-length and variable-length. In a fixed-length code, every character is assigned the same number of bits. For example, in standard ASCII, every character is assigned exactly 7 bits, which gives us $2^7 = 128$ possible character encodings. In general, from Information Theory we know that if we want to encode n different characters, also known as symbols, we need $log_2(n)$ bits per symbol. Thus, if our alphabet consists of 8 characters: A, B, C, D, E, F, G, and H, we need $log_2(8) = 3$ bits per character.

Problem 1 (0 points)

Review your notes or the pdfs of Lectures 24 and 25 on Huffman trees to become comfortable with such concepts as encoding, decoding, fixed- and variable-length codes, encoding/decoding invariant equations, symbol frequencies (aka weights and relative frequencies), symbol weight/frequency maps, node megers, and the Huffman tree generation algorithm we discussed in the last lecture (Lecture 25). I will use the terms *weight* and *frequency* synonymously.

Problem 2 (5 points)

Huffman Tree Nodes

Our first task is to implement a Huffman tree node class out of which we'll then build Huffman trees. The constructor of the class `HuffmanTreeNode` takes a set of characters and their weights. This must be either manually defined and computed from a dataset. If a node's a leaf, its symbol set consists of only one character. I've included in the assignment my source code in `HuffmanTreeNode.py`. I'd like to quickly draw your attention to two magic methods in `HuffmanTreeNode.py`: `__str__` and `__eq__` that we can use in debugging.

```
def __str__(self):
    return 'HTN(' + str(self.__symbols) + ', ' + str(self.__weight) + ')'

def __eq__(self, htn):
    return self.getWeight() == htn.getWeight() and \
           len(self.getSymbols().difference(htn.getSymbols())) == 0 and \
           len(htn.getSymbols().difference(self.getSymbols())) == 0
```

The method `__str__()` is called every time you call `str(x)` where `x` is a `HuffmanTreeNode` object. Here's an example.

```
>>> from HuffmanTreeNode import HuffmanTreeNode
>>> htn01 = HuffmanTreeNode(symbols=set(['A']), weight=8)
>>> str(htn01)
'HTN({'A'}, 8)'
>>> htn02 = HuffmanTreeNode(symbols=set([3, 4, 1]), weight=10)
>>> str(htn2)
'HTN({1, 3, 4}, 10)'
>>> print('My nodes are {} and {}'.format(htn01, htn02))
My nodes are HTN({'A'}, 8) and HTN({1, 3, 4}, 10)
```

The method `__eq__()` is called every time we call `x == y` where both `x` and `y` are `HuffmanTreeNode` objects. Here's how `__eq__()` is defined in the huffman node class.

```
def __eq__(self, htn):
  return self.getWeight() == htn.getWeight() and \
         len(self.getSymbols().difference(htn.getSymbols())) == 0 and \
         len(htn.getSymbols().difference(self.getSymbols())) == 0
```

This method returns true if the nodes have the same weights and there is no difference between the symbol sets. Let's take a quick look at `test_ht_ut00()` in `HuffmanTreeUTs.py`. In the following code segment, both assertions pass because `htn01` and `htn02` are equal (i.e., have the same symbol sets and weights).

```
htn01 = HuffmanTreeNode(symbols=set(['A', 'B', 'C']), weight=10)
htn02 = HuffmanTreeNode(symbols=set(['B', 'C', 'A']), weight=10)
assert htn01 == htn02
assert htn02 == htn01
```

In the next code segment both assertions pass, because `htn03` and `htn03` are **not** equal (their weights are equal but their symbol sets are not). All symbols in `htn03` are in the symbols of `htn04` but not vice versa.

```
htn03 = HuffmanTreeNode(symbols=set([1, 2, 3]), weight=5)
htn04 = HuffmanTreeNode(symbols=set([2, 3, 1, 5, 4]), weight=5)
assert htn03 != htn02
assert htn02 != htn03
```

From Huffman Nodes to Huffman Trees

Our Huffman tree class (`HuffmanTree`) will have only one member variable – the root, as shown in the code segment below from `HuffmanTree.py`. Once we have the root, we can get to all nodes by using `HuffmanTreeNode.getLeftChild()` and `HuffmanTreeNode.getRightChild()` (see `HuffmanTreeNode.py` on how these methods are defined).

```
from HuffmanTreeNode import HuffmanTreeNode

class HuffmanTree(object):
    def __init__(self, root=None):
        self.__root = root

    def getRoot(self):
        return self.__root
```

We iteratively construct a Huffman tree from from a list of Huffman node objects. We find the two nodes with the lowest weights and merge them to produce a new node (aka *merger node*) that has these nodes as its left and right children and whose weight is the sum of the weights of the two children (i.e., the two nodes being merged). We remove the two child nodes from the node list and add the new node back to the list. The procedure goes on until there is only one node left in the list. This node becomes the *root node* of the Huffman tree and the procedure terminates by returning this root node. That's our *code map* in the form of the Huffman tree for the alphabet we're interested in encoding and decoding. You may want to make a pause here and review Slides 10–27 in Lecture 25 for the examples of how Huffman Trees are constructed from a list of Huffman nodes.

Here's an implementation of the method for merging two nodes in `HuffmanTree.py`. Note that it's a static method, i.e., it doesn't require an `HuffmanTree` object and can be called as `HuffmanTree.mergeTwoNodes(n1, n2)`.

```
    @staticmethod
    def mergeTwoNodes(htn1, htn2):
        symbols = set(htn1.getSymbols())
        for i in htn2.getSymbols():
            symbols.add(i)
        n = HuffmanTreeNode(symbols=symbols, weight=htn1.getWeight() \
                                + htn2.getWeight())
        n.setLeftChild(htn1)
        n.setRightChild(htn2)
        return n
```

Implement the static method `HuffmanTree.fromListOfHuffmanTreeNodes()` that takes a list of `HuffmanTreeNode` objects and returns a `HuffmanTree` object constructed according to the Huffman tree construction algorithm outlined in Lecture 25.

Let's run `test_ht_ut01()` in `HuffmanTreeUTs.py` to test this method.

```
char_freqs01 = [('A', 4), ('B', 3), ('C', 1), ('D', 1)]
def test_ht_ut01(self, tn=1):
    hnodes = [HuffmanTreeNode(symbols=set([kv[0]]), weight=kv[1])
              for kv in HuffmanTreeUTs.char_freqs01]
    ht = HuffmanTree.fromListOfHuffmanTreeNodes(hnodes)
    assert ht is not None
    HuffmanTree.displayHuffmanTree(ht)
```

Here's the Huffman tree displayed with `HuffmanTree.displayHuffmanTree()` that my implementation constructs.

```
{'B', 'A', 'D', 'C'}:9
        {'A'}:4
        {'B', 'D', 'C'}:5
               {'D', 'C'}:2
                        {'C'}:1
                        {'D'}:1
               {'B'}:3
```

The above output of my Huffman tree can be type-drawn as follows.

```
                 ({'B','A','D','C'}, 9)
                  /                 \
                 /                   \
             ({'A'}, 4)      ({'B', 'D', 'C'}, 5)
                                 /          \
                                /            \
                       ({'D', 'C'}, 2)     ({'B'}, 3)
                        /        \
                       /          \
                  ({'C'}, 1)  ({'D'}, 1)
```

You may recall that, as we discussed in Lecture 25, David Huffman proved that there can be several equivalent Huffman trees constructed from the same symbol weight map (or the same list of Huffman tree nodes). The tree your algorithm will construct will depend on how you find the two nodes with the lowest weights and arbitrarily break the ties and may not look exactly the same as my tree. That said, however, the symbol code *lengths* in your tree must be the same as in my tree above. In other words, the code of `'A'` is 1 bit long; the code of `'B'` is 2 bits long; the codes of `'C'` and `'D'` are 3 bits long each. I've added a detailed debugging trace my implementation produces in a comment right before `test_ht_ut01(self, tn=1)` in `HuffmanTreeUTs.py`.

Implement the methods `encodeSymbol()` and `encodeText()` to encode individual characters and texts (aka messages), respectively. Actually, `encodeText()` simply concatenates into one string the outputs of `encodeSymbol()` for each individual character in the text from left to right. The direction is important, because if we were to encode messages in Arabic or Hebrew, for example, we'd go right to left. So, once we've implemented `encodeSymbol()`, `encodeText()` is a simple `for`-loop.

Both methods should return strings of characters `'0'` and `'1'`. We'll talk about bit fiddling later. The method `test_ht_ut03(self, tn=3)` in `HuffmanTreeUTs.py` constructs two alternative Huffman trees from the same list of huffman nodes and prints out the codes of individual characters as follows.

```
Huffman Tree Codes:
encode(A) = 0
encode(B) = 100
encode(C) = 1010
encode(D) = 1011
encode(E) = 1100
encode(F) = 1101
encode(G) = 1110
encode(H) = 1111
Alternative Huffman Tree Codes:
encode(A) = 0
encode(B) = 111
encode(C) = 1000
encode(D) = 1001
encode(E) = 1010
encode(F) = 1011
encode(G) = 1100
encode(H) = 1101
```

Implement the method `decode()` in `HuffmanTree.py` that takes a string of the '0' and '1' characters and decodes it with a given Huffman tree. The code segment below is from `test_ht_ut04(self, tn=4)` and tests the encode/decode invariant equations on Slide 4 of Lecture 25.

Here's my output.

```
txt0 = AABCCDEFG
enc0 = 00100101010101011110011011110
txt1 = AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA
enc1 = 00100101010101011110011011110
dec0 = AABCCDEFG
dec1 = AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA
```

Bit Fiddling

The above implementation of a Huffman tree is conceptually adequate insomuch as it generates 0-1 encodings of messages and decodes the encodings back into original messages. However, it's not adequate from the point of view of data compression in that it doesn't *reduce* the size of the encoded text. Why? Because it returns strings of 0-1 *characters* instead of sequences of *bits*. To address this problem, we need to implement a Huffman tree class that generates raw bytes instead of characters. Bit fiddling is not within the scope of this class. Since bit fiddling, in and of itself, doesn't have much to do with scientific computing (it's just a technique), I won't dive into it here. My hope is that you either have had some exposure to it in a computer architecture or an OS class or, if not, will get some exposure to it some day.

To experiment with bit fiddling, I've given you my implementation of a binary Huffman tree class in `BinHuffmanTree.py`. The class uses the methods in `HuffmanTree.py` but encodes texts as bit sequences (i.e., byte arrays) and decodes bit sequences back into texts. The constructor of `BinHuffmanTree` takes the root node of a `HuffmanTree` object. The method `encodeText()` of `BinHuffmanTree` takes a text message and emits the encoding of the message as a byte array, the number of bytes in the encoding, and the number of padded bits, i.e., the bits we have to add to the end of the encoding to make its length divisble by 8 (i.e., 8 bits per symbol like in extended ASCII).

The method `encodeTextToFile()` of `BinHuffmanTree` takes a text message and a file path and produces two files. The first file has the extension `.bin` and contains the bytes of the encoded message (i.e., the actual encoding). The second file has the extension `.txt` and contains two lines, each of which has one integer: the first integer denotes the number of bytes in the `.bin` file and the second integer denotes the number of the padded bits added to generate `.bin` file.

The method `decodeTextFromFile()` of `BinHuffmanTree` takes a file path, adds the `.bin` and `.txt` extensions to the file path to open the needed files and then decodes the `.bin` file to produce the original text.

Let's run `test_ht_ut06(self, tn=6)` and see the results. This unit test encodes and decodes two texts defined below.

```
txt0 = 'AABCCDEFG'
txt1 = 'AABBBBHHHFFGGGDDDDEEEECCCCCCCCDDDDDHHHHHAAAAAA'
```

On my Bionic Beaver (Ubuntu 18.04 LTS) with Python 3.6.7, the following files are generated in `data/`. The first column gives the number of bytes.

```
 4 Apr 16 12:27 test_txt0.bin
 4 Apr 16 12:27 test_txt0_pb.txt
```

```
 9 Apr 16 12:27 test_txt0.txt
20 Apr 16 12:27 test_txt1.bin
 5 Apr 16 12:27 test_txt1_pb.txt
46 Apr 16 12:27 test_txt1.txt
```

The files `test_txt0.bin` and `test_txt0_pb.txt` encode `txt0` saved in `test_txt0.txt`. The extension `_pb` stands for padded bits. The combined size of the two encoded files, `test_txt0.bin` and `test_txt0_pb.txt`, is $4 + 4 = 8$ bytes, whereas the size of the file with the original text, `test_txt0.txt`, is 9 bytes. In other words, we managed to save a byte.

The files `test_txt1.bin` and `test_txt1_pb.txt` encode `txt1` saved in `test_txt1.txt`. The size of the two encoded files, `test_txt1.bin` and `test_txt1_pb.txt`, is $20 + 5 = 25$ bytes, whereas the size of the file with the original text, `test_txt1.txt`, is 46 bytes. This time we managed to get a compression of 46 - 25 = 21 bytes, which is not bad at all!

Encoding and Decoding the Great "Moby Dick" by Herman Melville

The zip archive for this assignment contains the files `data/moby_dick_ch01.txt` and `data/moby_dick_ch02.txt` that contain the first two chapters of the great "Moby Dick" by Herman Melville. I downloaded these from Project Gutenberg https://www.gutenberg.org/ebooks/2701.

Recall that in Lecture 25 we posed the question of where the symbol weight maps come from. One answer is the come from the dataset. In our case, it comes from the two chapters of "Moby Dick." The class `CharFreqMap.py` gives a concrete answer to this question through the static method `computeCharFreqMap()` that creates a dictionary mapping all characters in a given text file to their frequencies in that file. If you save all chapters of "Moby Dick" in one text file, it'll create the frequency map of the entire novel. The static method `freqMapToListOfHuffmanTreeNodes()` in the class `HuffmanTree` takes this map and creates a list of `HuffmanTreeNode` objects.

The unit tests `test_ht_ut07(self, tn=7)` and `test_ht_ut08(self, tn=8)` compute character frequencies maps, build the corresponding Huffman trees, and use them to encode the first two chapters of "Moby Dick." Let's run them and look at what they save in `data/`. The first column gives the number of bytes for each file.

```
12285 Apr 16 12:44 moby_dick_ch01.txt
 6722 Apr 16 12:41 moby_dick_ch01.bin
    7 Apr 16 12:41 moby_dick_ch01_pb.txt
 8030 Apr 16 12:45 moby_dick_ch02.txt
 4412 Apr 16 12:41 moby_dick_ch02.bin
    7 Apr 16 12:41 moby_dick_ch02_pb.txt
```

Let's analyze what just happened. The size of `moby_dick_ch01.txt` (the actual text of the first chapter) is 12,285 bytes (the first line above). The total size of the two encoding files of the first chapterf (i.e., `moby_dick_ch01_pb.txt` and `moby_dick_ch01.bin`) is 6,722 bytes + 7 bytes = 6,729 bytes. This gives us 12,285 - 6,729 = 5,556 bytes of compression, which is a really good reduction. The size of `moby_dick_ch02.txt` is 8,030 bytes. The size of the two encoding files of the second chapter (i.e., `moby_dick_ch02_pb.txt` and `moby_dick_ch02.bin`) is 4,412 + 7 = 4,419. This gives us 8,030 - 4,412 = 3,618 bytes in savings. For the first two chapters we reached a compression of 9,174 bytes. We did well!

Let me put a final touch of digital oil to my painting of an orchard of Huffman trees in our last assignment of this semester. Huffman trees generated from gigabytes/terabytes of data are precious and should, therefore, be persisted. The `BinHuffmanTree.py` has two methods `persist()` and `load()` that use `pickle` to persist generated trees and read them back into a running Python later. You can run, if you want, the unit tests `test_ht_ut09(self, tn=9)`, `test_ht_ut10(self, tn=10)`, and `test_ht_ut11(self, tn=11)` to persist your Huffmann trees.

The methods `test_ht_ut09()` and `test_ht_ut10()` persist the Huffman trees built from chapters 1 and 2 as `moby_dick_ch01_bht.bin` and `moby_dick_ch02_bht.bin`, respectively. My `data/` directory after running these two tests contains the following files.

```
7765 Apr 16 12:55 moby_dick_ch01_bht.bin
7600 Apr 16 12:55 moby_dick_ch02_bht.bin
```

These two files can be repeatedly used to encode any messages (characters, sentences, paragraphs, pages) from the first two chapters of "Moby Dick." The method `test_ht_ut11()` loads both of them and uses them to decode the encodings of both chapters and compare them against the original texts of those chapters.

What To Submit

You should write your code in `HuffmanTree.py`. As usual, please zip all your files (except for the data directory files – we already have those!) into `hw12.zip` and submit your zip in Canvas. When your zip contains all the necessary files, grading your submission goes much faster.

Happy Hacking!

Thank You!

This is the last assignment for CS3430: S22: Scientific Computing. I've had a lot of fun teaching this class and learned a lot from your questions, suggestions, and comments. It's been a great run! There are many wonderful instructors and great classes at the USU CS Department. Tons of fascinating stuff to choose from. So, thank you for throwing your hats on my scientific computing hanger this semester!

A special thank you to all of you who have been coming to my F2F classes.