Huffman Coding

Aditya Gupta

March 2023

Introduction

In this assignment, we implement the Huffman algorithm for lossless text compression. This algorithm assigns short codes for frequently occurring characters and longer ones for less frequent characters.

Implementation

Character Frequency Map and the Creating the Huffman Tree

We create a map that stores the occurrence of each character in the input text. This frequency table is then converted to a list of tuples sorted based on the frequency count to aid us with building the Huffman Tree for the input text. To create the Huffman tree, we combine the character nodes with the lowest frequency and add their frequencies together. We insert this into the list in a sorted manner and continue this procedure until we are left with just one remaining node. We disregard the frequency in the node tuple and end up with the final Huffman tree.

```
def huffman_tree([{tree, _}]) do tree end
def huffman_tree([{h1,hf1}, {h2, hf2}|rest]) do
    huffman_tree(insert({{h1, h2}, hf1 + hf2}, rest))
end

def insert({a, f}, []) do
    [{a, f}]
end
def insert({a, f1}, [{b, f2}|rest]) do
    if f1 < f2 do
      [{a, f1}, {b, f2}|rest]
    else
      [{b, f2} | insert({a, f1}, rest)]
    end
end
```

Encoding Characters

To assign bit codes for all characters we employ a recursive approach that involves traversing the tree while doing a depth-first search for the character and generating the code along the way. For each time a left branch in the tree is taken we add a “0” to the code and a “1” every time we take the right branch. When we reach the leaf node containing the character, we add the table and its corresponding code into a map which represents the encoding table. The code is stored in the form of a list. To make the process a bit more efficient we add 1s or 0s to the beginning of the code instead of the end of the list, and in the end, we reverse the list representing the code when entering it into the table to get the correct code for the character.

```
def encode_table(tree) do
    encode_table(tree, %{}, [])
end
def encode_table({left, right}, table, code) do
    table = encode_table(left, table, [0|code])
    encode_table(right, table, [1|code])
end
def encode_table(char, table, code) do
    Map.put(table, char, Enum.reverse(code))
end

def encode_text([], _, acc) do acc end
def encode_text([head|rest], table, acc) do
    if(Map.get(table, head)!=nil) do
      acc = acc ++ Map.get(table, head)
      encode_text(rest, table, acc)
    else
      encode_text(rest, table, acc)
    end
end
```

Decoding Characters

We can obtain the decoding table by just inverting the encoding table such that its keys and values are swapped. Decoding the Huffman encoded characters was quite confusing at the beginning, as I forgot that the encoding is such that each character can be assigned a code with different lengths. But upon realizing my mistake it was fairly simple to implement a decoding method for the encoded text. All the Huffman codes have a unique prefix such that no 2 characters begin with the same prefix, meaning we will not encounter any false matches when decoding. We read the bits one at a time and we accumulate the bits to form a key which eventually matches an entry in the decode table. When a match is found we retrieve the character and reset the key accumulator for the next character encoded in the bits.

```
def decode(table) do
    Map.new(table, fn {key, value} -> {value, key} end)
end

def decode(bits, table) do
    decode(bits, table, '', '')
end

def decode([], _, _, result) do result end
def decode([head|rest], table, key, result) do
    key = key ++ [head]
    temp = Map.get(table, key)
    if temp != nil do
      decode(rest, table, '' , result ++ [temp])
    else
      decode(rest, table, key, result)
    end
end
```

Benchmarking

The time taken to build the Huffman tree seems to increase in a nearly linear manner. Most of this is probably spent calculating the frequency table, after which the process would take about the same time as the number of unique characters in the text is fixed. After this, we examine the time taken to create the encoding table and it doesn't seem to increase because of the fixed number of unique characters in the text. This would increase in case a different text is used or a text in a different alphabet is used. The encoding and decoding times appear to grow linearly as well, with the encoding taking more time than decoding. The overall compression seems to be close to 54% with the approach working better when working with datasets with more characters.

“n” Characters	Tree Build Time(ms)	Encode Table Build Time(ms)	Encode Time(ms)	Decode Time(ms)	Source Size(bytes)	Encoded Size(bytes)	Compression Ratio
10	0	0 (size 9)	0	0	11	4	0.364
100	0	0 (size 26)	0	0	106	52	0.491
1000	6	0 (size 37)	13	1	1065	548	0.515
10000	18	0 (size 54)	1822	405	10487	5543	0.529
100000	79	0 (size 74)	410025	84357	104746	55989	0.535
200000	111	0 (size 78)	751247	266235	209311	112373	0.537

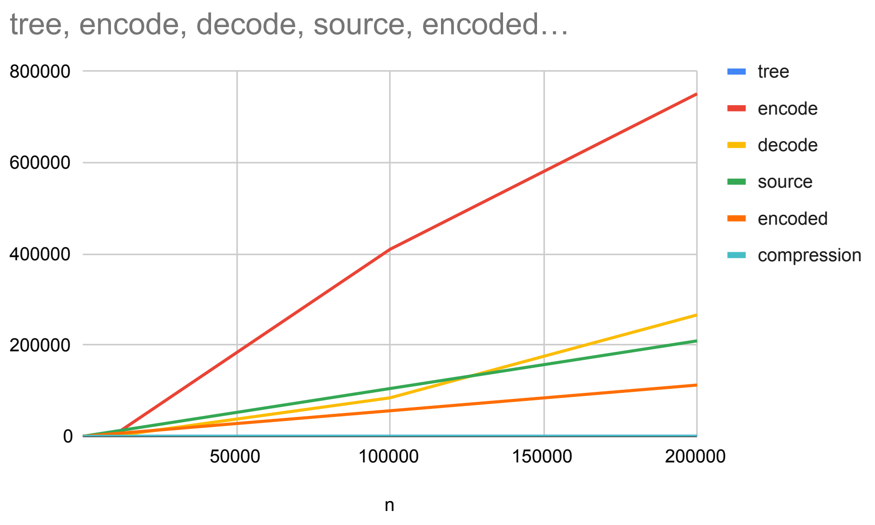

Figure 1: Graph Showcasing data in the table above