Huffman 8

Simon Yllmark

March 4, 2024

1 Introduction

For this assignment I did a text compression program. The idea is that the program will take a sample text and create a Huffman tree, where the character will be represented as numbers and the positioning of that those characters in that tree will depend on the frequency of how often those characters occur in the sample text.

Then the program will create a encoding and decoding table. The idea is that the encoding table will map character to bit sequence and the decoding table will map bit sequence to character. In this implementation the decoding table is the encoding table.

Then when the initial work is done the program will be able to encode a piece of text of any length, the idea is that the encoding of a message would same memory space because a UTF-8 character is represented using 8 bits but using the encoding method the same text could be represented using a smaller sequence of bits. Then using the decode table the message will be decoded and become more readable for the person who wants to read that text.

2 Tree

Before creating a Huffman tree the frequency of the characters appearing in the sample text need to be calculated and returned as list of tuples that contain the element and a count value stating how often that character appear in the text.

Then that frequency list of tuples will be sorted using a merge sort algorithm where the characters with the highest frequency appear first. Then the program creates the Huffman tree using the huffman() function where the placing of the characters will depend on the frequency that they occur in the sample text. A code snippet can be viewed below:

```
def tree(sample) do
  freq = freq(sample)
  freq = Enum.sort(freq, fn({_, x}, {_, y}) -> x < y end)
  huffman(freq)
end
```

3 The table

The encoding/decoding table will contain the information in how to find a certain character in the Huffman tree. For example if we have character A in the table and the path to that character is bit sequence [1,0,1] that means in the Huffman tree go to the right branch, then the left and then one right and there you will find that value. Then instead of writing character A as a bit sequence of 8 bits it can be written as a bit sequence of 3 bits. A code snippet can be viewed below:

```
def encode_table(tree) do
  encode_table(tree, [])
end
def encode_table({left, right}, path) do
  leftp = encode_table(left, path ++ [0])
  rightp = encode_table(right, path ++ [1])
  leftp ++ rightp
end
def encode_table(tree, path) do
  [{tree, path}]
end
```

4 Encode/Decode

If one would give the program a text message then using the encoding table the program would be able to encode that message to a smaller message. Then using the same table the program would be able to decode that message. Thus, one could send someone a message using a sequence of bits with a Huffman tree and a decoding table then that person would be able to decode that message. This would be more useful if the receiver had the Huffman tree and the decode table then the message giver would only have to send a bit sequence.

5 Bench

This is the code that calculates the bench mark from a data set ”data.txt” that can be viewed on the canvas page for this assignment can be viewed

below:

```
    {text, length} = read("data.txt")
    {tree, tree_time} = time(fn -> tree(text) end)
    {encode_table, encode_table_time} = time(fn -> encode_table(tree) end)
    {decode_table, decode_table_time} = time(fn -> decode_table(tree) end)
    {encode, encode_time} = time(fn -> encode(text, encode_table) end)
    {_, decoded_time} = time(fn -> decode(encode, decode_table) end)

    e = div(length(encode), 8)
    r = Float.round(e / length, 3)

    def time(func) do
    {func.(), elem(:timer.tc(fn () -> func.() end), 0)}
  end
```

The result from the code can be viewed below:

```
Tree Build Time: 44544 us
Encode Table Time: 0 us
Decode Table Time: 0 us
Encode Time: 178073 us
Decode Time: 1870130 us
Compression Ratio: 0.521
```

6 Conclusion about benchmarks

To decode characters take around twice the amount of time as to encode things, according to the bench marks form section 5. After extra viewing of the code I have concluded that the encode() time complexity is $O(n^2 + n)$ that is around $O(n^2)$ because first the program need to check if the character is in the tree and then get the bit sequence for that character, then the program would have to append that bit sequence to a list.

The decode() functions must then have a time complexity that is higher than for the encode() function because how else can the function take 10 times the time as the other with the same input. The time complexity of this operation is $O(n * x^2)$ where x is the average length of a character bit sequence and n is the length of the input. Because the bit sequence have to be checked bit by bit and it is done n times.

7 Conclusion

It can be viewed in section 5 that the text in the data set ”data.txt” was compressed to half the size using the encode() function, thus the same in-

formation was compressed to half it original size in terms of bits. This is a strategy for compressing written information and thus saving memory. This would work well for text written in English because the English alphabet only consist of 29 characters.

One drawback is that the information encoded becomes useless if there is no decoding table that can decode the encoded information. Thus, there would be risk if information is sent to a recipient that does not have the decoding table. There is also the other scenario where a person has the decoding table that is not the intended recipient of a message.