Huffman

Ayda Ghalkhanbaz

Spring Term 2024

Introduction

This report covers the implementation details of Huffman coding algorithm. The logic behind this algorithm says that each character can be coded into a binary number (bit sequence), where frequently used characters are represented by shorter sequences while less used characters are coded into longer sequences. We will go through the implementation details of this coding algorithm and do some benchmarking. The results are then discussed in later sections.

Method

The general idea is to build a tree with all characters as leaves of the tree and then number the branches with either 0 or 1 (as the characters should then be represented with a binary number). As mentioned earlier, the more often used characters should be closer to the root of the tree while less used characters are placed at the bottom.

The first step of implementing the tree involved determining the frequency of each character in a given text. This was done by using the in-built Maps, where keys are the character and the values are their frequencies. So as we go through a list of characters, we add every character to the map and count the number of each character in the text.

The next step is to build the tree itself. What is done in this step is to sort the map we got from the previous step and convert it to a list. Then for each two least frequent tuples in the list (these tuples are characters with their frequencies), we combine them to one node. This repeats until the list becomes empty.

The third step involves creating an encoding table. By traversing the tree in a depth-first manner, we can generate code for each character, for instance the left branches are represented by 0 while right branches represent the bit 1. Now encoding the text is only iterating over each character, looking up the corresponding binary code from the table and appending it to a binary sequence that represents the compressed text.

The last step can be done in two ways; where one might be a smarter choice than the other. The first approach is to take a sequence of binary code, starting from one bit, and search for the corresponding character in a decoding table (similar to the encoding table). If there is a hit for that one bit so the found character is returned, but if no character is found, we add the next bit in the sequence to the first one and search for the character in the table. We keep repeating this process until the entire text is decoded. Below you'll find the code snippet demonstrating the mentioned decoding algorithm.

```
def decodeDum([], _)  do [] end
# decoding the binary sequence
def decodeDum(seq, table) do
  {char, rest} = decode_char(seq, 1, table)
  [char | decodeDum(rest, table)]
end
# searching for the corresponding character
def decode_char(seq, n, table) do
  {code, rest} = Enum.split(seq, n)
  case List.keyfind(table, code, 1) do
    {char, _} -> {char, rest}
    nil -> decode_char(seq, n+1, table)
  end
end
```

The smarter decoding is done by following the binary sequence in the tree until we find a character. For instance, starting from the root and depending on whether the binary sequence starts with zero or one we go through the left or right branch, respectively. We keep continuing this traversal till a character is found.

The code snippet bellow illustrates the `decode/2` and `decode/3` functions, employed for this approach. Note that we do not need any deocoding table here, instead we use the Huffman tree directly.

```
# calling helper function, decode/3
def decode(encoded, tree) do decode(encoded, tree, tree) end
# let the bits in the sequence guide you through the tree
# 0 -> goes to the left branch
# 1 -> goes to the right branch
def decode([], char, _) do [char] end
def decode([0|rest], {zero, _one}, tree) do
  decode(rest, zero, tree)
```

```
end
def decode([1|rest], {_zero, one}, tree) do
  decode(rest, one, tree)
end
def decode(encoded, char, tree) do
  [char|decode(encoded, tree, tree)]
end
```

Result

The benchmark was done on an input text with 318997 characters and a byte size of 333816, and 77 characters in total. Building the Huffman tree and encoding the text (including creating the encode table) took about 30 ms each. However table 1 in bellow, shows the run time for decoding the text using the two mentioned approaches.

First Approach	Second Approach
1630	10

Table 1: *Execution time of two approaches of decode function in ms*

Discussion

The benchmarking done for this task may not be good enough to give an understanding of the performance and in real life time complexity. However, by looking at the results presented in table 1, we can see that the first approach is as slow as we expected from theory. The fact that the first approach has a time complexity of $\mathcal{O}(n)$ can be approved by looking at the implementation, where n is the length of the encoded binary sequence. The reason is that in the worst-case scenario we need to go through the decoding table linearly over and over so that finally we get a hit.

On the other hand, the second approach searches through a tree and therefore has a time complexity of $\mathcal{O}(log(n))$ where n is the number of leaves in the tree. This leads to a faster execution time compared to the first approach.