Huffman - Report

Adrian Hjert

09/Mar/2023

Introduction

The aim of this report is to discuss Huffman coding as well as the results of some performance tests using large texts. Some of the functions used to solve the problem will be discussed in the **Methods** section of this report. The benchmark results will be presented and discussed in the **Results and Conclusions** section of the report.

Methods

Huffman is a coding algorithm which utilizes a binary tree that represents the characters. The characters in this case are sorted in order of frequency where the most common characters end up on the left of the tree and the less frequent characters end up to the right of the tree. The algorithm also follows the idea that combines the two least common characters until only one node remains. This is then used as the root of the tree, whcih then branches to the left if a zero is added to its bit sequence, and to the right if a one is added to the bit sequence. Some of the most important functions will be discussed here. First, we need to actually build the Huffman tree. This is given in the instructions. The function makes use of a supplied sample text which includes all the characters we wish to use. he function then uses the **freq** function to calculate the frequency of the characters and builds the tree using the **huffman** function. These helper functions can be seen in the following code.

```
def freq(sample) do
    freq(sample, [])
  end

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

The **huffman** helper function essentially makes a frequency table and as described previously, returns the root of the Huffman tree. If we do this, the sequence has fewer elements, further compressing the text and reducing the access time for low frequency down to O(1).

Now, we should start to worry about the actual encoding and decoding of the texts. For this we use the following functions: **encode_table/1**, **encode/2**, **decode_table/1** and **decode/2**. Let us begin with the main functions responsible for encoding the text. These functions can be seen in the following code.

```
def encode_table(tree) do
   codes(tree, [])
 end

  def encode([], _table) do [] end
 def encode([char | rest], table) do
   {_, code} = List.keyfind(table, char, 0)
   code ++ encode(rest, table)
 end
```

The first of the two functions takes the Huffman tree we created previously and returns a list of tuples. This list represents the encoding of the characters in the tree. The latter function has two cases; the base case case where we have no list of characters, so we get an empty list, the other case is the one that is a bit more useful to us. It essentially takes a string (our text) and the encode table we created with the **encode_table/1** function and replaces the original characters in the text with their corresponding binary code. Every character in the text is iterated over and the binary code is appended to the relevant character to create an encoded string which is a compressed version of the input text.

Now, to decode the text. The main functions responsible for this can be seen in the following code.

```
def decode_table(tree), do: codes(tree, [])

def decode([], _table) do [] end
  def decode(seq, table) do
   {char, rest} = decode_char(seq, 1, table)
   [char | decode(rest, table)]
  end
```

Similar to the encoding coutnerparts of these functions the first function creates a decoding table using our Huffman tree. The base case of the latter functions behaves in the same way as the encoding counterpart. The other case takes a Huffman tree as the input and returns a lsit of tuples which represent each characters' decoding. The functions responsible for decoding use two helper functions, one of which is also used by the encoding functions. These are as follows.

```
def codes({a, b}, curNode) do
    left = codes(a, [0 | curNode])
    right = codes(b, [1| curNode])
    left ++ right
  end
 def codes(a, code) do
    [{a, Enum.reverse(code)}]
  end

 def decode_char(seq, i, table) do
    {code, rest} = Enum.split(seq, i)
    case List.keyfind(table, code, 1) do
      {char, _rest} ->
        {char, rest}
        nil ->
          decode_char(seq, i + 1, table)
    end
  end
```

The **codes/2** helper function returns a list of pairs representing the characters in the tree and their corresponding binary code. The function then checks if the Huffman tree is a leaf node or an internal node. In the former case, the function returns a list with a single pair, in the latter case, the function recurses on its left child, then the right chilld and concatenates the lists and pairs them. The **decode_char/3** function recursively traverses the Huffman tree table and decodes a single character. It splits the input sequence and attempts to find a matching key-value pair. If a match is found, the decoded character is returned, otherwise the function calls itsef recursively while incrementing i until a match is found or the input sequence ends.

The program was benchmarked using some supplied code and text. The results were collected and can be seen in the **Results and Conclusions** section of this report along with a brief discussion.

Results and Conclusions

The data seen in **Table 1** represents the benchmark results using the supplied *.txt* file.

It can be seen rather obviously in the table that the tree builds rather fast, and equally quick across the board. The same thing goes for the table

Table 1: Showing Benchmark Results

Text Len.	*Tree*	*TabSize*	*Enc.*	*Dec.*	*Src. Size*	*Enc. Size*	*Comp. Rat.*
200	0.0 ms	31 in 0.0 ms	0.0 ms	2.0 ms	225 bytes	109 bytes	0.484
400	1.0 ms	36 in 0.0 ms	0.0 ms	6.0 ms	452 bytes	223 bytes	0.493
800	1.0 ms	38 in 0.0 ms	0.0 ms	8.0 ms	904 bytes	444 bytes	0.491
1600	1.0 ms	43 in 0.0 ms	2.0 ms	29.0 ms	1774 bytes	895 bytes	0.505
1657	1.0 ms	43 in 0.0 ms	1.0 ms	28.0 ms	1835 bytes	926 bytes	0.505

being built at different sizes. The only noticable differences are the decoding times as we get to the longer text lengths and the time seems to depend on very small differences, notice how the decoding time goes from 8 ms at a table size of 38 all the way to 29 ms at 43, a difference of just 5. The compression ratio is also rather similar at around half, indicating that the algorithm generally compresses the text to half its original size.