Huffman Coding

William Frid

Spring 2024

Introduction

Huffman code is an algorithm used to compress data without any quality loss. It's an old algorithm created back in 1952 by a man called David A. Huffman at MIT that's still in use today very widely. In a nutshell, it uses a binary tree, and an encoding table to shorten the encoding of each character and store these in as short branches as possible in the tree, where a shorter branch means a more frequently used character.

This report covers a basic implementation of the Huffman code using Elixir to give an introduction to a popular and rather simple way of compression. As this report is rather short, it's recommended that you read it alongside the full code which is available on GitHub at `https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2`.

Encoding

The first step of the Hoffman encoding is to calculate the number of times each character occurs in the string one wishes to encode. This can be done in multiple different ways, such as seen in the snippet below. This function returns a list where each character ASCII code is mapped to an integer representing the number of occasions.

```
def freq(sample) do freq(sample, %{}) end
 def freq([], frq) do Map.to_list(frq) end
 def freq([char | rest], frq) do
   case Map.get(frq, char) do
     :nil ->
       freq(rest, Map.put(frq, char, 1))
     f ->
       freq(rest, Map.put(frq, char, f + 1))
   end
 end
```

Note that the list one finds via the code above needs to be sorted by the number of occasions of each character in ascending order.

The next step is to generate the Huffman tree, a binary tree built with a twist. Assume we have the the list $[\{100, 1\}, \{99, 1\}, \{98, 2\}, \{97, 4\}]$ and pass this to `huffman/1` (note that 97 stands for the character "a", 98 for "b", and so on). The function then detects that it's a list and not a proper tree. Thus it combines the characters in a tuple and pairs this inside a tuple with the total amount of occasions ($f1 + f2$).

This newly created node is then sent further to `insert/2` where the recursion is used to place its character data correctly in the soon-to-be tree. The if-statement in `insert/2` determines whatever to put into the left or right, and in what branch. The rule here is to put the most frequently used characters on as short branches as possible to the right.

```
def huffman([{tree, _}]) do tree end
def huffman([{tree1, f1}, {tree2, f2} | rest]) do
    node = {{tree1, tree2}, f1+f2}
    huffman(insert(node, rest))
end

def insert(node, []) do [node] end
def insert(node1, [node2 | rest] = nodes) do
    if (elem(node1, 1) <= elem(node2, 1)) do
        [node1 | nodes]
    else
        [node2 | insert(node1, rest)]
    end
end
```

Following the example written above (and ignoring the amounts written), we would get the flow shown below:

$[\{100, 1\}, \{99, 1\}, \{98, 2\}, \{97, 4\}]$
$[\{\{100, 99\}, 2\}, \{98, 2\}, \{97, 4\}]$
$[\{\{\{100, 99\}, 98\}, 4\}, \{97, 4\}]$
$[\{\{100, 99\}, 98\}, \{97\}]$

The next step is to generate an encoding table, which is used for encoding a given string. Creating an encoding table out of a Huffman tree can be done through depth-first reversal through the tree. The idea is to start with an empty map, then follow each branch down to each leaf, store the character found there, and combine this with the path taken back up to the root of the tree. This might sound like a lot of code but it can be as simple as seen in the snippet below.

```
def encode_table(tree) do encode_table(tree, [], %{}) end
def encode_table({zero, one}, path, table) do
```

```
    table = encode_table(zero, [0 | path], table)
    encode_table(one, [1 | path], table)
end
def encode_table(char, path,table) do
    Map.put(table, char, Enum.reverse(path))
end
```

To use the encoding table generated with `encode_table/1` is then as simple as creating an empty list, and starting reading the given string to encode and insert each path to this character in the tree. For instance: 'abc...' would with the Huffman tree seen above result in '101001...'.

Decoding

Finally comes the decoding. This can be easily done by traversing down the tree until a leaf is found. Assuming we have the encoded string '101001...', then we would first take a step to the right, and notice that this is the character "a". We put this aside and start over, only this time with '01001...'. We first take a left, then a right, and find a "b". Next, we have '001...' and thus take to left, and one right, to find a "c". And on it goes.

This is a great way of reusing the Huffman tree we generated and avoiding the need for for instance a decoding table which would only slow down the algorithm. Seeing this solution right away might however be tricky, thus the importance of analyzing the resources generated through the algorithm.

A clear example of this code is visible in the snippet below.

```
def decode(encoded, tree) do decode(encoded, tree, tree) end
def decode([], _, _) do [] end
def decode([0 | rest], {zero, _}, root) do
    decode(rest, zero, root)
end
def decode([1 | rest], {one, _}, root) do
    decode(rest, one, root)
end
def decode(encoded, char, root) do
    [char | decode(encoded, root, root)]
end
```

Benchmarking

To measure the speed of this implementation of Huffman code there are two things to take into consideration; the length of the data, and the size of the alphabet. The length of the data is self-explanatory, but the size of the

alphabet might be harder to understand. This is due to the increased size of the Huffman tree which leads to longer encoding. You can simplify this by saying that a shorter alphabet leads to better compression. The test run for this report simply used randomly generated text with a fixed alphabet size of 41, but with a 3 different size to explore how the time encoding and decoding changes.

The results in table 1 do show a somewhat linear time complexity as the time it takes doubles as the number of characters given doubles. This is a desirable attribute of the implementation as it shows that it's effective and scales well with larger amounts of data (required for real-life compression where the data is of greater size).

Chars.	**Bytes**	**Compressed**	**Time(ms)**
100000	100000	53000	80
200000	200000	106000	178
300000	300000	160000	238
400000	400000	213000	463

Table 1: Benchmark comparing time to compress different amounts of bytes.

Discussion

One could adjust how the Huffman tree is built to prevent too long branches from forming to improve the performance. This could be through adjusting the algorithm so that it builds a more balanced tree. Doing so does however require more computations and would slow down the encoding, which is a drawback when speed is in force.

The benchmark that was used for this report isn't thorough enough to give a full understanding of the performance. Additional benchmarks where comparisons between different alphabets should have been run as this has a major effect on especially the compressed byte size.

Conclusion

Hoffman code is a simple yet effective way of encoding and decoding data. It gives a great understanding of the principles behind encoding and can built on for improved performance. It also shows great versatility despite having been invented way back.