Huffman Coding

Axel Månson Lokrantz

Spring Term 2023

Introduction

Huffman coding is a lossless data compression algorithm i.e a form of compression that preserves all the original data. It was first described by David Huffman in 1992 and utilizes a binary tree as its base. In this assignment we will implement a Huffman tree to encode and decode chunks of data and run some performance tests. The program was written in Elixir together with Daniel Dahlberg at KTH.

Github link

A link to the program in its entirety can be seen here.

A Huffman Tree

The algorithm at its core works by building a binary tree based on the frequency of characters. Characters that appear often will have a higher frequency number and will be placed closer to the root and vice versa. When traversing the tree to find a character and picking the left-child, we write down a zero and for every right-child a 1. We repeat this for every character in the set of data and end up with a list of zeroes and ones which will be a much smaller representation than we had originally.

Building the Tree

Before implementing the tree we define the type of nodes. A leaf node is connected to a single character and a node contains a left- and a right child.

```
defmodule Node do defstruct [:left, :right] end
defmodule Leaf do defstruct [:key] end
```

Next we have to break the data into separate characters with their corresponding frequency and convert them to leaf nodes. Once done we have an ordered set of leaf nodes which will be used to build the binary tree.

```
def huffman_tree([{root, _freq}]) do root end
def huffman_tree([{node_a, freq_a}, {node_b, freq_b}
| rest]) do
    new_node = %Node{left: node_a, right: node_b}
    total = freq_a + freq_b
    queue = [{new_node, total}] ++ rest
    queue
    |> Enum.sort_by(fn {_node, frequency} -> frequency end)
    |> huffman_tree()
  end
```

We pop the the two nodes of the queue and create a new parent node with the sum of their frequencies and prepend it to the queue. The queue is then sorted, to make sure the lowest frequencies are placed at the head of the queue. The function is called recursively until the base case is met.

Encoding

Once the tree is built we need to be able to keep track of each character. This is done by writing down the path we take to reach a character. Taking left three times and then right to reach the letter 'a', for example, would result in the code `0001`.

```
  def find(tree, character, path \\ [])
  def find(%Leaf{key: key}, character, path) do
    case key do
      ^character -> {character, Enum.reverse(path)}
      _ -> nil
    end
  end
  def find(%Node{left: left, right: right}, character, path) do
    find(left, character, [0|path]) ||
    find(right, character, [1|path])
  end
```

The function `find` lets us do a simple lookup for a single character. To get a complete list of all the characters in the tree we create another function `table` as seen below.

```
  def table(sample) do
    tree = freq(sample)
    |> huffman_tree()
    sample = to_atom_list(sample)
    map = Enum.map(sample, fn char -> find(tree, char) end)
    map = Enum.reverse(map)
  end
```

We can now encode a given set of data through the `encode` function. The function returns a compressed list of ones and zeroes.

```
def encode(sample \\ sample()) do
   map = table(sample)
   Enum.reduce(map, [], fn {_, element}, acc ->
   acc ++ element end)
 end
```

Decoding

With the ability to encode a set of data we should also be able to decode it back to its original state. The procedure is quite straight forward, all we need is a schematic (represented by `table`) over all the characters on how to interpret them. The argument `n` starts at 1 and is where we split sequence. Thus, the first snippet of code we look for is either a zero or a one. If neither of those are represented by a character we increment n by one until we find the character we are looking for.

```
def decode([], _) do [] end
def decode(seq, table) do
    {char, rest} = decode_char(seq, 1, table)
    list = [char | decode(rest, table)]
    List.to_string(list)
end

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
```

Benchmark

Chars	Encode (ms)	Decode (ms)	Tree (ms)	Compression (ratio)
10	0.03	0.03	0.07	0.4
100	0.08	1.6	0.3	0.5
1000	13	70	3	0.5
10000	700	6000	11	0.5

Table 1: Benchmark for the Huffman implementation. Measurements in milliseconds.

For really large values of n ($n > 10000$) the benchmark takes a long time. The implementation is by no means optimized. For example instead of traversing the Huffman tree every time we encode a text and lookup the Huffman code, we can do a complete traversal once and save the result in a map. This way, we can easily access the Huffman code for each character without the need to traverse the tree each time. The queue can also be improved. instead of sorting the queue after every insert, we can use a priority queue that automatically orders the elements based on their priorities. This will allow us to avoid unnecessary sorting operations and improve the efficiency of the algorithm.