Huffman Coding

Sareh Jalalizad

Spring Term 2023

Introduction

For this assignment, we were supposed to work with Huffman decoding/encoding algorithm. Huffman encoding is a lossless data compression algorithm. In this project, we implemented a basic Huffman encoding and decoding algorithm using Elixir. We implemented several functions to build a Huffman tree, create an encoding table, and encode/decode a message using the Huffman codes. We also performed benchmarks to measure the performance of our implementation.

The table

The first step of the task includes counting the frequency of all characters in the given text and building a binary tree from it. The freq function is implemented by using a map where the characters are the keys, and their frequency is the corresponding value. It recursively traverses the list of characters, updating the frequency map for each character it encounters. The Map.update/4 function is used to increment the count of each character in the map. If the character is already in the map, its count is incremented by 1 (using &(&1 + 1)); otherwise, a new entry is added to the map with a count of 1.

```
def freq(sample) do freq(sample, %{}) end
def freq([], freq) do freq end
def freq([char | rest], freq) do
    updated_map = Map.update(freq, char, 1, &(&1 + 1))
    freq(rest, updated_map)
end
```

The tree function simply calls freq to get the frequency distribution and then passes it to the huffman function to build the Huffman tree.

The Huffman tree

The Huffman tree is built by sorting the character frequencies, combining tuples with the lowest frequency counts, and creating a binary tree with each character represented as a leaf node.

```
def huffman(freq) do
  build_tree(Enum.sort_by(freq, fn {_, f} -> f end))
end

def build_tree([{char1, _}]) do char1 end
def build_tree([{char1, freq1}, {char2, freq2} | rest]) do
build_tree(insert({{char1, char2}, freq1 + freq2}, rest)) end

def insert({char, freq}, []) do [{char, freq}] end
def insert({char1, freq1}, [{char2, freq2} | rest]) do
  if freq1 < freq2 do
     [{char1, freq1}, {char2, freq2} | rest]
  else
     [{char2, freq2} | insert({char1, freq1}, rest)]
  end
end
```

The build_tree function checks if the list has only one tuple, it returns the character represented by that tuple. If the list has more than one tuple, it combines the frequency counts of the first two tuples (lowest frequency) and inserts a new tuple into the list. The insert function is a helper function that inserts a new tuple into a sorted list based on the frequency count of the tuple.

Huffman encoding

The next function implemented is the encode_table function. This function takes a Huffman tree and generates an encoding table for the characters in the tree. The encoding table maps each character to its corresponding Huffman code. The function traverses the tree and records the path to each leaf node, which corresponds to a character in the tree. The path to the leaf node is represented as a list of 0s and 1s, where 0 represents a left child and 1 represents a right child in the tree.

```
def encode_table(tree) do generate_codes(tree,[]) end
def generate_codes({left, right}, path) do
    generate_codes(left, [0 | path]) ++
    generate_codes(right, [1 | path]) end
```

```
def generate_codes(char, path) do [{char, Enum.reverse(path)}] end
```

Once the encoding table is built, we can encode given data using the encode function. This function takes a list of characters and an encoding table as input. It looks up the Huffman code for each character in the encoding table and appends the corresponding path for that character to the result list. The function recursively traverses the list of characters until it reaches the end, and then returns the compressed list of ones and zeroes.

```
def encode([], _) do [] end
def encode([char | rest], table) do
  find(char, table) ++ encode(rest, table)
end

def find(_, []) do [] end
def find(char, [{char, path} | _]) do path end
def find(char, [_ | rest]) do find(char, rest) end
```

Huffman decoding

The decode function takes a sequence of bits and a decoding table and decodes the sequence back into the original characters. The function works by decoding each character from the input sequence using the Huffman decoding table. It starts by decoding the first character using the decode_char function and then recursively decodes the remaining characters until the end of the input sequence is reached.

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

We implemented a benchmark function to measure the performance of each function for a larger text. Running each of the functions with the text from "Kallocain" gets the result that shows in the following table. We could observe that creating the Huffman tree and encoding text were the fastest functions in the algorithm. However, decoding the text took more time than other functions. the reason for that is probably that for the decoding we need to continuously search for bit sequences in the table multiple times until we find them. The algorithm was able to compress the "Kallocain" text to around 55% of its original size.

Functions	**Time**
Build Tree	74
Encode Table	0
Encode	131
Decode	2057

Table 1: Benchmark for the Huffman implementation, run time in (ms)