Huffman

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open a window onto the Huffman compressing algorithm implementation, following the guidelines presented in the lecture. Of course, in order to be able to solve such a problem, it is very important to highlight the essential ideas that stand behind it. First and foremost, the frequency of each letter is important since depending on it, the Huffman tree is then going to be constructed. At the same time, this tree will also serve as the base to building up the encoding table in order to obtain the compressed final result. Last but not least, the decoding will be done by reading the through the Huffman tree alongside the encoded text.

Step 1: The letter frequency list

The first step in building the Huffman compressing algorithm consists in calculating the frequency of each occurring letter. In order to do so, the *freq/2* method will be implemented. First and foremost, its one parameter only version will call it with the text argument, alongside a newly created map where the letters and their frequencies will be stored (key-value association).

It should also be taken into account that we will work with the sample text under the form of a char list. In addition:

- the Map module will be used in order to put and get the elements from the list;
- base case: if the char list is empty, return the map under the form of a list;
- analyze the head of the list: if the element has already been added to the list, call the *freq/2* method with the tail and updated map as argument (it has increased element frequency by one); otherwise, do the same thing, except add the new element to the list with the frequency equal to one;

```
def freq(sample) do freq(sample, %{}) end
def freq([], frq) do Map.to_list(frq) end # turn the map into a list
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))
    #Map.get/2 returns the value associated to the key: the character frequency
  end
end
```

Step 2: The Huffman Tree

The main idea that stands behind the Huffman tree is ordering from top to bottom the letter frequency nodes from the most frequent to the least frequent. Having said that, in the end, the letters at the bottom will have the longest encoding codes, but since rarely do they occur, this won't come as a major problem. In other words, in order to do so, we have to sort the frequency list, not by the character, but by their associated frequency. And that is precisely why, a function, as presented below will be implemented alongside using the Enum module.

```
freq = Enum.sort(freq, fn({_, f1}, {_, f2}) -> f1 < f2 end)
#the frequency list is now in ascending order and ready for the Huffman tree
```

Since in the implementation, the Huffman tree will be built from bottom to top, the list is sorted in ascending order and the following steps will then, be taken:

- implement two methods: one that takes care of creating nodes (*huffman/1)* and the other one that correctly inserts the nodes (*insert/2)*;
- base case of the first method: if there is one tree structure left only, return that specific tree (this is the final Huffman tree);
- create a node made out of the first two list elements: the node will be a tuple made out of two elements - the tuple of the two given trees and the other one, the sum of the two frequencies;
- continue this recursive step with the result after having correctly inserted the node in the tree;
- base case of the second method: if the remaining list in the inserting method is empty, return the list of the node itself;
- if the frequency of the node that is to be inserted is strictly less than the frequency of the first remaining node, then add it to the left in the

tree, namely as head in the list - otherwise: add the remaining list's first node as header and keep on inserting to the right of the tree until the node that was to be initially inserted is in the correct spot;

```
def huffman([{tree, _}]) do tree end
def huffman([{tree1, f1}, {tree2, f2} | rest]) do
  node = {{tree1, tree2}, f1+f2} #create a new node
  huffman(insert(node, rest)) #call recursive step and insert node
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
```

After having taken all these steps, the Huffman tree should be correctly implemented, whose top is marked by the most frequent letters.

Step 3: The encoding table

From a theoretical point of view, the encoding consists in placing a zero on all left branches and a one on all the right ones. After having done that, all that is left to do is read the resulting codes following the branches until a letter is reached. The resulting codes will then be part of the encoding table. On the other hand, from a technical point of view, the *encode_table/3* will be implemented, which will make use of the Huffman tree and accumulate in a path list the letter code, which will be added to the table once a leaf (letter) is reached.

- use pattern matching to match the Huffman tree with the initial encoding tuple {zero,one};
- for each child call the recursive step while also adding a zero or a one to the list depending on the taken branch;
- when a letter is reached (it is not a tuple), add it to the table alongside the path which has to be reversed since the tree is read using depth first traversal;

Once the table is done, the encoding part is almost done. All that there needs to be done is to create a new method *encode/2* which takes each character in the list at a time and maps it to its encoding code. The final result is at the same time the final compressed text under the form of a list.

Step 4: Decoding

Finally, once a text is encoded using Huffman's algorithm, the easiest and most efficient way to decode it is to read the encoding while simultaneously traversing the tree. Once a letter (a leaf) is met it means there is a match and most importantly, *we have to go back up to the tree's root in order to continue the reading from where we left off in the encoding list.* The root will always be sent as parameter and will be represented by the tree itself and therefore, the *decode/3* method will be created as follows:

- base case: if the encoding list is empty, deliver the empty list;
- if in the encoding a zero is met, pick the zero branch in the tree and keep on exploring with the rest of the encoded list;
- if in the encoding a one is met, pick the one branch in the tree and keep on exploring with the rest of the encoded list;
- if a leaf is met (not a tuple for it to be a tree), it means that a letter has been reached: add it to the list and keep on decoding the rest, by always going back to the root;

```
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)
  #regardless of what is after, follow and decode the zero branch
end

def decode([1 | rest], {_, one}, root) do
  decode(rest, one, root)
  #regardless of what is after, follow and decode the one branch
end

def decode(encoded, char, root) do #a letter has been met
  [char| decode(encoded, root)] #add it to the list and decode the rest
  #in the method argument: encoded = rest of the encoded code list
end
```

Benchmark

In order to evaluate how efficient the Huffman algorithm implementation is, the following benchmark has been run in order to see how long it takes to encode and at the same time, decode a large text, namely the provided excerpt from "Kallocain" by Karin Boye.

Operation	ExecutionTime[us]
encoding	100K
decoding	30K
encode + decode	156K

Table 1: Benchmark analysis of the encoding and decoding time.

As it can be noticed above, the algorithm is pretty efficient especially when considering the lengthy chosen paragraph. It should also be taken into account that initially, the decoding and encoding execution times have been measured separately, while in the end they have been measured both at the same time, as one operation only.

Looking back, as far as this algorithm is concerned, I would say that the most challenging part was constructing and inserting the nodes in the Huffman tree.