ID1019: Huffman Coding

Alexander Lundqvist

Spring Term 2023

Introduction

This assignment is about implementing Huffman coding, which is a data compression algorithm. The goal is to create a module with functions that can create a Huffman tree, an encoding table, and encode/decode a text using the tree and table. The instructions provides a guideline on how to approach the implementation, including how to create the frequency table, the Huffman tree, and the encoding table. Finally we measure the performance of the module.

Implementation

To complete the assignment I had to use the code in the course repository. However only the most basic parts we're used, since I didn't really understand all the improvements that were implemented. I'm not going to clutter the report with that code but just provide a brief explanation of how it works.

Huffman

The Huffman algorithm works by assigning variable-length codes to the characters in the input text, based on the frequency of occurrence of each character. The characters that occur more frequently are assigned shorter codes, while those that occur less frequently are assigned longer codes.

We do this by using a tree based data structure. This is because Huffman is an entropy encoding method, meaning it performs best when the input data has a skewed distribution of symbol frequencies. A tree works well for this since its traits are suited for what we want to accomplish with Huffman encoding.

The process of building the Huffman tree begins with calculating the frequency of each character in the sample text. The character frequencies are

used to create leaf nodes, which are then sorted in ascending order of frequency. The algorithm iteratively selects the two nodes with the lowest frequencies and combines them into a new node with a combined frequency. This new node is then inserted back into the sorted list, and the process is repeated until only one node remains, representing the root of the Huffman tree.

Encoding

The encoding process converts an input text into a compressed binary representation using a encoding table that is generated from the Huffman tree. The encoding table maps each character in the input text to a unique binary code based on the character's frequency in the text. It is generated by depth-first traversal, prioritising the left direction.

Decoding

Similar to the encoding process, the algorithm first constructs the decoding table by depth-first traversal. However, is different from encoding because it involves sequentially reading the binary codes from the encoded text and matching them with their corresponding characters in the decoding table.

Benchmarking

To test the system I wrote additional additional functions tt bench/1 and tt bench/2. I didn't use the tt bench/2 but it allows to input any text file if needed. When running the tt bench/1 function which uses the "kallocain.txt" file, and for example an input size of 10000 we get the following output:

```
iex(10)> Huffman.bench(10000)
text of 10000 characters
tree built in 1.0 ms
table of size 54 in 0.0 ms
encoded in 2.0 ms
decoded in 41.0 ms
source 10487 bytes, encoded 5543 bytes, compression 0.529
```

We perform the test multiple times, always doubling `n`. The results are presented in the following table.

We can plot the data to see the relationship between the functions time complexities and the input size. I didn't look up the length of the "kallocain.txt" so the last bench run skews the result but if we look at the graph we can see that decoding is by far more time consuming than encoding which isn't

Text Size	Encoding Time (ms)	Decoding Time (ms)
10000	3.617	45.907
20000	9.123	100.099
40000	11.066	203.059
80000	22.022	460.753
160000	55.234	1045.648
320000	90.587	1934.874
640000	88.024	2014.559

Table 1: Huffman encoding and decoding benchmark results

all that strange since decoding involves scanning through bit sequences of varying length over and over again while encoding is a simple lookup in the table.

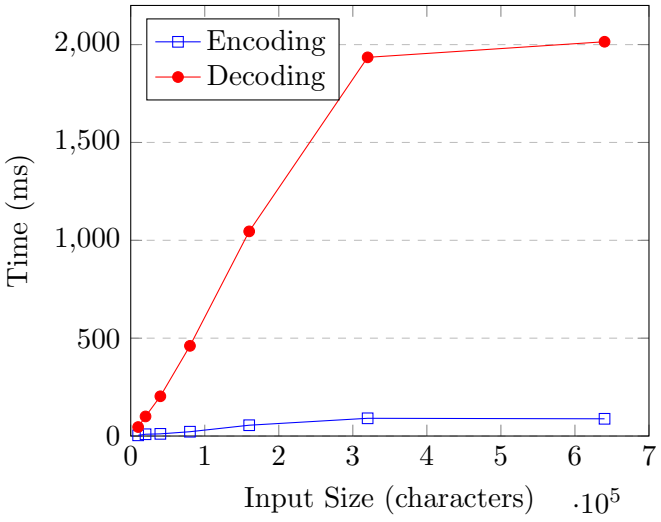

Figure 1: Execution time vs. input size.

Discussion

The time complexity of looking up a character in the Huffman coding table depends on the data structure used to store the table. We could perhaps use another more efficient data structure, such as a hash table, then the time complexity for looking up a character is $O(1)$. However we will instead lose other functionalities that a tree based structure has. In practical applications, the difference in lookup time complexity does not significantly impact the overall performance, as the encoding and especially the decoding processes are usually more computationally intensive.

Some of my own thought on the assignment; I found the task very hard to understand at first. I am still not very comfortable with the language itself and a rusty dealing with tree/graph data structures. It also stated that you had to understand what you were doing and how everything works, so I found myself just doing more reading than actual coding since we are graded for reports, not code.