Huffman Coding in Haskell

Rafał Włodarczyk, Michał Waluś

June 10, 2025

Contents

1 Introduction **2**

2 Initial Requirements **2**
2.1 Parameters 2

3 Huffman Coding Algorithm **2**
3.1 Prefix Codes 3
3.2 Frequency Map 3
3.3 Priority Queue 4
3.4 Huffman Tree construction 4
3.5 Encoding and Decoding 5
3.6 Bytestream Operations 5
3.7 The Main Function 5

4 Dynamic Huffman Coding Algorithm **5**

5 CI **5**
5.1 Build 5
5.2 Exec 6
5.3 Bin 6

6 Showcase **6**

7 Conclusions **6**

8 Future Work **7**

1 Introduction

In 1952 David A. Huffman created an algorithm used for lossless data compression [?], building upon the advancements of Claude Shannon's work on information theory. It has been proven to be optimal for a symbol-by-symbol coding with a known input probability distribution. The algorithm assigns variable-length codes to input characters, the shorter codes being assigned to more frequently occuring characters.

2 Initial Requirements

The program has been designed to fulfill the following requirements:

Write a program that uses Huffman coding (classic or dynamic) to compress files. This project will allow you to practice working with binary data and using tree-like data structures. To handle binary data, you can use the "bytestring" package.

We decided that our take on the implementation will provide a simple command line interface.

Compress: Compresses the input file using Huffman encoding.

```
$ huffman [input-file] -o [output-file]
```

Outputs the compression ratio and sizes.

Decompress: Decompresses the input file.

```
$ huffman [input-file] -d -o [output-file]
```

2.1 Parameters

- [**input-file**]: The file to be processed (required).
- **-o** [**output-file**]: Specifies the output file (required).
- **-d**: Enables decompression mode (default is compression).

3 Huffman Coding Algorithm

As described in [?], Huffman coding is a...

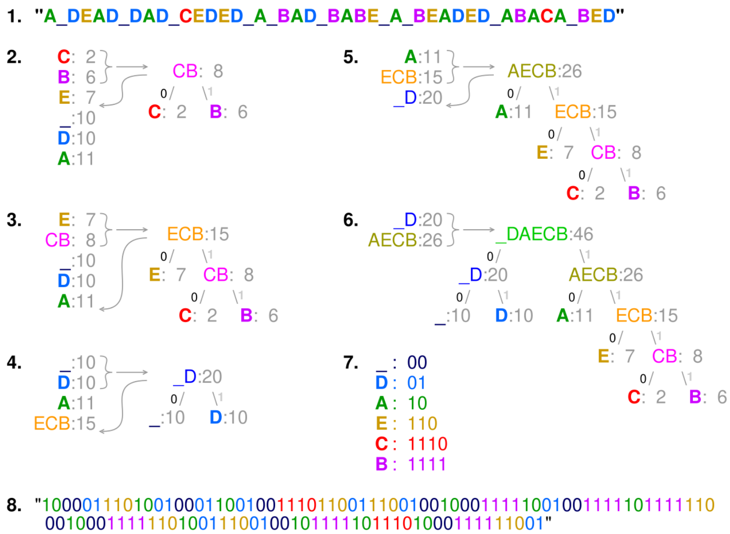

Figure 1: Huffman Coding Algorithm Schema

3.1 Prefix Codes

A set of codes $\{C_1, C_2, \ldots, C_n\}$ is a prefix code if:

$$\forall i, j \in \{1, 2, \ldots, n\}, i \neq j : C_i \not\subseteq C_j \tag{1}$$

In any stream of bits, defined as $S = b_1 b_2 \ldots b_m, b \in \{0, 1\}$, a prefix code can be decoded unambiguously. This feature is crucial for the Huffman coding algorithm, as it allows for deterministical decoding of the compressed data in linear time.

3.2 Frequency Map

A frequency map assigns each character in the input data the number of its occurences. For example it can be easily constructed from a **String** intrinsic.

```haskell
type CharMap = [(Char, Int)]

add :: CharMap -> Char -> CharMap
add [] c = [(c, 1)]
add ((x, y):xs) c
    | x == c = (x, y + 1):xs
    | otherwise = (x, y):add xs c

mapChars :: String -> CharMap
mapChars = mapCharsHelp []
    where
```

```
        mapCharsHelp cm [] = cm
        mapCharsHelp cm (x:xs) = mapCharsHelp (add cm x) xs
```

Listing 1: Constructing a frequency map from a string.

3.3 Priority Queue

In order to build the Huffman tree in linear time, we must preserve the order of the characters based on their number of occurences. A priority queue is a data structure which allows for efficient insertion and extraction of the minimum element.

```
type LeafQueue = [Tree Char Int]

createLQ :: String -> LeafQueue
createLQ = charMapToQueue . mapChars

insertLQ :: LeafQueue -> Tree Char Int -> LeafQueue
insertLQ [] tree = [tree]
insertLQ (t:lq) tree
    | get tree < get t = tree:t:lq
    | otherwise = t:insertLQ lq tree

mergeLQ :: LeafQueue -> Tree Char Int
mergeLQ [] = error "Cannot merge an empty queue"
mergeLQ [t] = t
mergeLQ (t1:t2:ts) = mergeLQ $ insertLQ ts $ Node t1 t2 $
    get t1 + get t2

charMapToQueue :: CharMap -> LeafQueue
charMapToQueue = charMapToQueueHelp []
    where
        charMapToQueueHelp lq [] = lq
        charMapToQueueHelp lq ((c, i):cn) =
            charMapToQueueHelp (insertLQ lq $ Leaf c i) cn
```

Listing 2: Priority queue implementation using a binary tree.

3.4 Huffman Tree construction

We can now construct the Huffman tree (a binary tree with leaves containing characters) from the frequency map each time extracting the two least frequent characters and merging them into a new node.

```
data Tree a b = Leaf a b | Node (Tree a b) (Tree a b) b
    deriving (Eq)
    makeCode :: Tree Char Int  -> Map Char String
    makeCode t = makeCodeHelp t ""
        where
```

```
            makeCodeHelp (Leaf x _) s = singleton x s
            makeCodeHelp (Node t1 t2 _) s = union (
                makeCodeHelp t1 (s ++ "0")) (makeCodeHelp t2
                (s ++ "1"))
makeCode :: Tree Char Int  -> Map Char String
makeCode t = makeCodeHelp t ""
    where
        makeCodeHelp (Leaf x _) s = singleton x s
        makeCodeHelp (Node t1 t2 _) s = union (makeCodeHelp
            t1 (s ++ "0")) (makeCodeHelp t2 (s ++ "1"))
```

Listing 3: Huffman tree construction.

3.5 Encoding and Decoding

The encoding process is straightforward. we traverse the Huffman tree and replace each character with its corresponding code, we decided to use a **Map** intrinsic to store the codes for each character. We can join the coding map with the input data to produce a compressed binary stream.

3.6 Bytestream Operations

The package **bytestring** provides an efficient way to handle binary data in Haskell.

3.7 The Main Function

The main function uses the IO monad to read the input file, perform processing and write the output file.

4 Dynamic Huffman Coding Algorithm

As proposed by [**?**], dynamic Huffman coding allows for encoding data without knowing the frequency distribution beforehand, by maintaining a dynamic tree structure that adapts to the input data as it is processed.

5 CI

5.1 Build

The project uses Stack as a build tool, which allows for simple dependency management and building.

```
$ stack build
```

5.2 Exec

Afterwards the executable can be run with the following command:

```
$ stack exec huffman -- [input-file] -o [output-file]
```

5.3 Bin

To make it available globally, we can copy the executable to a directory in our PATH.

```
$ export PATH=$PATH:/<stack-path>/bin
$ huffman
Usage: huffman <input-file> [-d] -o <output-file>
$ huffman src/Main.hs -o Main.hs.huff
src/Main.hs (1887 bytes) -> Main.hs.huff (1411 bytes)
    [74.77%]
```

6 Showcase

```
$ wget https://pl.wikipedia.org/wiki/Polska -O polska.txt

$ sha256sum polska.txt
645e668ee...9c530  polska.txt

$ stack run -- polska.txt -o polska.txt.huff
polska.txt (1271161 bytes) -> polska.txt.huff (887871 bytes)
     [69.85%]

$ ls -lah polska.txt polska.txt.huff
-rw-r--r-- 1 rafisto rafisto 1.3M Jun  9 14:14 polska.txt
-rw-r--r-- 1 rafisto rafisto 868K Jun  9 23:27 polska.txt.
    huff

$ stack run -- polska.txt.huff -d -o polska.out.txt
polska.txt.huff (887871 bytes) -> polska.out.txt (1271161
    bytes) [143.17%]

$ sha256sum polska.out.txt
645e668ee...9c530  polska.out.txt
```

7 Conclusions

The Huffman coding algorithm with its presented implementation in Haskell has shown to be an effective way to compress text files. We can assume a compression ratio of around 70% for large text files, with the decompression process being lossless and efficient.

8 Future Work

The current implementation can be extended in several ways:

- Support for more complex data types, such as images or binary files.
- Optimization of the priority queue, library **containers** provides a more efficient implementation.
- Implementation of dynamic Huffman coding to adapt to changing input distributions.