Huffman Code Implementation in Haskell

Breno Fatureto
breno.afb@gmail.com

***Abstract*—We describe the implementation of a Huffman Tree encoder and decoder in Haskell. The implementation leverages Haskell's purely functional approach to develop a elegant implementation.**

I. Introduction

Huffman codes are prefix codes that are optimum for a given distribution of symbols [Sayood]. Building a Huffman tree can be described as a three-step process:

1) Given the input, compute the relative weight of each symbol.
2) From the distribution, build the Huffman tree.
3) From the tree, assign a binary code to each symbol.

The second step consists of repeatedly merging each symbol's nodes based on the weight. This process, as will be shown later, is very easily implemented using recursion and Haskell's algebraic data types.

While the underlying implementation is able to operate on a stream of arbirary symbols from an ordered set, for practicality we constrain the possible input symbols to two types: `Word8`, for binary data, and `Char`, for Unicode characters. The main advantage of such constraint is that Haskell's compiler can automatically derive code for serializing to and from binary files. And, since any file is represented in a binary format, we can encode any input using the binary mode, albeit at a possible loss of efficiency. The tool is also able to take an encoded file and decode it back to the original file.

The project is available at GitHub. It uses the Stack build tool. Usage instructions are available in the project's readme.

II. Implementation

A. Counting symbol frequencies

If the input type is a list of symbols, we can easily implement the procedure by recursively traversing the list and updating a store of values. However, Haskell generalizes the notion of a traversable structure using the 'Traversable' typeclass. A typeclass defines specific behavior that a type is able to accomplish. Therefore, we can execute a state update function for each symbol in a traversable stream.

The `updateCount` function takes a single symbol and updates it's count in a *monadic State* context. After that, the `freqs` function takes an input stream and updates the store in sequence by combining each individual state update into a single large state update.

B. Building the tree

The second step leverages Haskell's algebraic data type system to describe the structure of the tree recursively. We define a recursive data type `Tree`, which can either be a `Leaf`, containing a symbol and the corresponding weight, or a `Node`, which contains the corresponding collection of symbols, the aggregated weight, and the recursive nodes.

```
data Tree a n = Leaf n a
              | Node n [a] (Tree a n) (Tree a n)
```

The core operation done on this datatype is `merge` which, given two `Tree`, combines them into a single `Node` which contains the sum of the weights and the union of the symbols of each `Tree`.

We then use the `merge` operation iteratively to assemble the tree. This is done by taking each symbol and its weight and transforming it into a `Leaf`, then examine the `Trees` with lowest weight and merging them. This procedure is done until there is a single `Tree` left.

C. Constructing Codes

First, we define the operation `getCode`, which, given a `Tree` and a symbol, returns the symbol's corresponding bitstring. Since it is possible that the given symbol is not present in the `Tree`, the function can fail. This behavior is captured with the polymorphic type `Maybe`.

Applicative syntax is used to handle the non-deterministic operation of searching for a symbol in the tree. Given a `Node`, we attempt to find the symbol in each branch, and return whichever result (if any) succeeds.

The process of building a table of codes for each symbol consists of executing `getCode` for each symbol in the alphabet and collecting the results.

D. Encoding

Encoding consists of, given an input stream of symbols and a table associating each symbol to a code, convert the input to its Huffman code representation. This is done with the `encode` function. One detail that we have to keep track of is the number of padding bytes prepended to the encoded representation, since the output has to be a whole number of bytes. This information, along with the encoded, padded representation is returned.

Haskell's generics mechanism allows us to easily embbed this information in a structured data type which can be serialized to and from a binary file. For this purpose, we define two data types, `EncodingDataT` and `EncodingDataB` for text and binary files, respectively.

E. Decoding

The process of decoding can be summarized as, given an input bitstream and a table associating each prefix bitstring with a symbol, produce the stream of symbols represented by the bitstring. This is where Huffman's code prefix property comes into play: since each code is not a prefix of any other code, we just need to keep taking symbols until we find a corresponding entry in the table.

In our implementation, this is done using a conjunction of state operations under the `StateT` monad for accumulating into the output, as well as failable lookup operations under the `Maybe` polymorphic type. Such mecahisms allows us to safely, as well as succintly, describe the decoding algorithm.

F. Results

In order to test the implementation, we test the program on a set of files. The set is composed of 6 text files and one image. The text files include the contents of a book (`domcasumurro.txt`), snippets of computer code (`TEncEntropy.txt` and `TEncSearch.txt`), as well as random sequences of characters (`fonte.txt`, `fonte0.txt`, `fonte1.txt`).

We measure the compressed raw size of the encoded file (as reported by the program, excluding metadata) and compare it against the original size, obtaining the compression ratio. All text files are compressed in both binary and text mode, and the image is compressed in binary mode.

We also examine the entropy obtained with each method (T for Unicode text mode, B for binary mode) with the average symbol length of each source.

File sizes are measured in bytes, while entropy and average symbol lengths are presented in bits/symbol.

Filename	File size	Compressed (T)	Compressed (B)	Ratio (T)	Ratio (B)
TEncEntropy.txt	19415	12882	12882	0.6635	0.6635
TEncSearch.txt	253012	164196	164196	0.6490	0.6490
domcasumurro.txt	389670	217025	227984	0.5569	0.5851
fonte.txt	1000000	368693	368693	0.3687	0.3687
fonte0.txt	1000	209	209	0.2090	0.2090
fonte1.txt	1000000	415793	415793	0.4158	0.4158
lena.bmp	263290	-	247810	-	0.9412

Filename	ASL	Entropy (T)	Entropy (B)
TEncEntropy.txt	5.3079	5.2719	5.2719
TEncSearch.txt	5.1917	5.1581	5.1581
domcasumurro.txt	4.6000	4.5619	4.6430
fonte.txt	2.9495	2.8940	2.8940
fonte0.txt	1.6680	1.5835	1.5835
fonte1.txt	3.3263	3.3033	3.3033
lena.bmp	7.5296	-	7.5097

III. Evaluation

We evaluate this implementation by comparing it with the `zip` utility commonly available in Unix systems. This utility uses the PKZIP compression scheme. The specific version used is the one available on macOS 11.1.

We measure the original file size and compressed file size in bytes as reported in `ls`. We then compute the compression ratio obtained by dividing the compression file size by the original file size.

Filename	Size	Compressed Size	Ratio
TEncEntropy.txt	19415	14356	0.7394
TEncSearch.txt	253012	165855	0.6555
domcasumurro.txt	389670	219091	0.5622
fonte.txt	1000000	368845	0.3688
fonte0.txt	1000	292	0.2920
fonte1.txt	1000000	416105	0.4161
lena.bmp	263290	252557	0.9592

Filename	Size	Compressed Size	Ratio
TEncEntropy.txt	19415	4409	0.2271
TEncSearch.txt	253012	38759	0.1532
domcasumurro.txt	389670	153834	0.3948
fonte.txt	1000000	428496	0.4285
fonte0.txt	1000	501	0.5010
fonte1.txt	1000000	489832	0.4898
lena.bmp	263290	227772	0.8651

We see that our tool obtained much better results in the random text sequences, and slightly better in the natural-language book. The `zip` tool obtained better results in the image file, and much better results in the code snippets.

IV. Conclusion

We present an implementaion of a Huffman tree encoder and decoder in Haskell. The encoder is able to achieve average symbol length close to the source's entropy. The observed compression ratio was best when encoding random strings of text, as compared to the widely availale `zip` utility.