Huffman Coding

Hassan Nazar

January 29, 2015

Abstract

The "Huffman Coding" seminar task was aimed to give the student some elementary insight to Erlang programming and the concept "functional programming". This report will run through the process of creating the Huffman Coding implementation in Erlang and discussions on time complexity as well as algorithm efficiency...

1 Introduction

The "Huffman Coding" seminar task was aimed to give the student some elementary insight to Erlang programming and the concept "functional programming", As well as give the student some knowledge about what Huffman coding is, what it is for and more. The main task of the seminar was to implement a functional program in which the Huffman coding was implemented. Main tasks such as frequency counting, tree building, encoding and decoding was going to be implemented though the Erlang language, and was to be tested in the end. This report will run through the process of creating the Huffman Coding implementation in Erlang and discussions on time complexity as well as algorithm efficiency will also be taken into consideration.

2 Attempt

2.1 Quick intro to Huffman Coding

In computer science and computer technology, a Huffman Code is the optimal prefix code, the algorithm was developed by David A. Huffman while in his years of studies. Huffman Coding uses a specific method of representing symbols which results in a prefix code, a bit string that represents some particular symbol. The main reason to have a optimal prefix language like the one David A. Huffman invented is due to the fact that this is much more light weight on the system. This is hopefully understood more clearly by the following example: A Computer uses something called ASCII[American System Code for Information Interchange], and is a character encoding scheme. It encodes every character to a 7-bit binary integer. With this said we can for every byte(8 bits) have a maximum of 1 character. Whilst if we use the Optimal prefix algorithm known as Huffman Code we can encode the a given string to a binary tree which will yield that we can store up to 4 characters in one byte. The Huffman coding is thus a compression technique, used to reduce the number of bits that is needed

to send a message. It uses a standard which says "The more common letters, The less bits", and thus those letters who are less common should need more bits for their representation. This is done by creating a upside down tree known as Huffman tree and by labeling every path down to the letter as either "0" or "1" depending on if the letter is in the right path or left path of the tree. This way we can represent every number as a set of binary values. Thus reducing the memory used and also making the message more easily and faster sent to the receiver.

2.2 Basic Setup

To be able to order the words for Huffman Coding, we firstly need to know how often the words appear in the sample string given. For this we were asked to choose a data structure of our choice and implement a simple frequency counter for every character that appears in the string. The main contention of this is to later on be able to structure all the characters in a balanced binary tree format with nodes. Through this we will be able to label every character with a base 2 bit value, which is called encoding. The code below is a visual representation of the implementation of such a frequency counter. It has a couple of important parts with are crucial to explain. The first part of the implementation will sort every character in the string so that we(in a consecutive order) have all letters after each other. Through this we can create a recursive Erlang function to count every following character that is similar to the previous one.

```
% 2.0 Freq Implementation
freq(Sample)->
    freq(lists:sort(Sample),[]).
freq([], Freq)->
    Freq;
freq([Char | Rest],Freq) ->
    {Block, MBlocks}=lists:splitwith(fun (Block) -> Block==Char end, Rest),
    freq(MBlocks,[{Char, 1 + length(Block)} | Freq]).
```

We simply create a tuple Block,MBlocks (Se Figure 1) with which we split the big list of characters at every point there is a consecutive number of repeating characters. If we have [a,a,b,b,c] then this is splits up to [a,a] and [b,b,c]. (splitwith function takes in two parameters, 1: The condition, 2: The list to split). After this we send the rest of the characters [b,b,c] back to the recursive function(for further splitting) while we set the frequency to the length of the splinted block [a,a]. Thus we get that the character "a" (in this situation) has the value 2 in frequency, and so forth.

2.3 The Huffman Tree Implementation

When the frequency counter is all set up we can go forth to try and build the actual Huffman tree. The tree will be used further on for the compression of ASCII values for more memory efficiency. To make the Huffman Tree we will use the frequencies as our main meat and bone. Something to remember at this point is that Huffman Coding had a "standard" that the most repeating letters will have the least binary compression value, whilst the least appearing

numbers will use more binary numbers for their representation. This is why we are interested in the frequency of the letters given. We start by having the frequency values in a consecutive order, from lowest to largest frequency value. And we will start building the tree by always joining those 2 letters who have the least frequency value, and through this we will build up the tree. When the tree is finished, we will have the letters with highest frequency with the lowest binary representation value.

```
% 2.1 Tree Building Implementation
huffman([{X, _} | []]) ->
    X;
huffman(Freq)->
[{Key1, Fre1},{Key2, Fre2} | Rest] = lists:keysort(2, Freq),
huffman([{{Key1,Key2}, Fre1 + Fre2} | Rest]).
```

In our implementation we start by getting the creating a tuple inside of a list where which we will get the first two values from the "keysort" function. We sort recursively so that we always get the lowest value in the "key1" and "key2" variables, what we have acquired now is the two characters which had the lowest frequency. This so that we can correctly implement the Huffman tree. After this we merge these 2 characters into one "node" with the name Key1,Key2 and with the frequency ..Fre1 + Fre2. The node is now called by its 2 branches names, this will be used later on when we are implementing the Huffman Codes function in 1.2. Through this recursive method we get a binary tree with nodes and leaves to represent the string given. Question given in the report: 1) Implement something that works and then look at this: what happens if we represents a leaf by the tuple node, Char, Freq, na, na and a node by node, na, Freq, Left, Right, would things be easier? - We noticed that although this would work, the way we had implemented it used only that information which is necessary for the tree so that we can later on encode through it. We are not in need of the frequency after the tree is built neither are we in need of nodes or left or right assignments. A tree for the string "foobar" would be: 111,98,97,114,102, Where 98,97 is a node with the letters "b" and "a", which is combined with the node 114,102 of letters "r" and "f" to create a new node 111,98,97,114,102 with "o" as the leaf. This can also show that the most used letter "o" will have the least binary bit representation. The same implementation with the above given form would have a lot of unneeded information but will possibly look better for the eye.

2.4 The Huffman Codes

When the tree structure is finished we can go on to label every character with a "0" or "1" so that we can later on use this for encoding another string. The simple implementation was to create a recursive function who labels every tuple with L, R where L stands for "Left" and R for "Right". We go through every tuple in the tree and label every step with a number. This is how we will be able to later encode a string and its characters to bit string values.

2.5 Huffman Encoding

This is where we are presented with a text string through which we are going to use the table of Huffman codes implemented in 1.2 to encode the string into a sequence of bit values. The table implemented has the form [111,[0], 98,[1,0,0], 97,[1,0,1], 114,[1,1,0], 102,[1,1,1]...] And the only main thing to do really is to search through this list and append the acquired bit-sequences to satisfy as a representation of the string given. The full Erlang implementation for this is given below.

```
%%------------------------------------------------------------------------
%% Encode Implementation:
%%------------------------------------------------------------------------

encode(Text, Table) ->
Dict = dict:from_list(Table),
encode(Text, Dict, []).

encode([], _Dict, Result) ->
Result;

encode([Char | Rest], Dict, Result) ->
Newvar = dict:fetch(Char, Dict),
Newlist = lists:append([Result, Newvar]),
encode(Rest , Dict, Newlist).
```

Although the aim was simple, it was quite a lot trickier in practice. The first function takes in the Text to encode and the Table we will use to encode from. We simply start by creating a dictionary though the "from list" function. The "from list" function converts the Key - Value list List to a dictionary so that we can later on get key value pairs from it. This function ends with that we call the function shown on the bottom of the image. The third function gets the Text value that is to be encoded and also the dictionary of the Table list we created in the first function. We then use the fetch function to look up the first character of the "Text" string in the dictionary we created. The dict:fetch(Char,Dict) function returns the value associated with Key in the dictionary Dict. fetch assumes that the Key is present in the dictionary and an exception is generated if Key is not in the dictionary. The binary code representation of the character is then appended in to a list and the rest of the characters in the String are sent back through the recursive function for the same purpose. When the String list is finally empty, it means that we have iterated through every character recursively, and thus we get sent to the middle most function that returns the Result list of all the binary bit values.

Questions given by the teacher:

Q1: Stop here an ponder what the time complexity is. You will of course have a linear factor, depending on the length of the text, since the text is encoded character by character but you might have other factors. What is the time complexity of looking up a character in the table?

Ans: We have to run through the full Text string which is n. For every character in the string we have to search the dictionary. A search in the dictionary should take at most n searches, this gives us the Time complexity:O(N^2)

Q2: What is the complexity of producing the final sequence of bits? Ans: As we have N characters and we will have most probably do N appends to the list, thus this should yield the time complexity: O(N^2)

2.6 Huffman Decoding

```
%%----------------------------------------------------------------------
%% Decode Implementation:
%%----------------------------------------------------------------------

decode([], _Table, Result) ->
Result;

decode(Seq, Table, Result) ->
{Char, Rest} = decode_char(Seq, 1, Table),
Newvar = Char,
Newlist = lists:append([Result, [Newvar]]),
decode(Rest, Table, Newlist).

decode_char(Seq, N, Table) ->
{Code, Rest} = lists:split(N, Seq),
case lists:keyfind(Code, 2, Table) of
{C,_} ->
{C, Rest};
false ->
decode_char(Seq, N+1, Table)
end.
```

At this point we wanted the code to be able to decode a String of bit values to a readable string. If we suppose that the message has been recieved by the client and that it now is to be shown to him then we would need to use such a decoder function. The Implementation is pretty straight forward, we take each character and search for its bit-sequence, when it is found we send that character to a list which grows for every recursion.

3 Summary

Understanding the task was pretty straight forward and easy. After some reading on the topic i believe i got a good understanding of the subject. The real issue was the implementation of the various tasks, creating the frequency counter, Encoding and Decoding were major parts that took alot of time. All in all it went well!