Huffman Coding

Bailey Fenzl

Ohio University, Athens, OH 45701

1 Introduction

Data compression has always been an important aspect of computer science. Data compression allows the storage of the same amount of information in less space, and has various applications, including MP3 and JPEG file formats. On a low level, compression requires information to be encoded in a way that requires fewer bits than the original. Huffman coding is an algorithm that is used in the application of encoding information. Mostly, it is used as a means of creating a character encoding system. To my knowledge, there exists one other implementation of Huffman Codes in Coq written by Laurent Thry. His implementation differs from mine in that while mine covers only the basics, his his much more thorough and ultimately provides a proof of correctness to the algorithm.

Overall, my implementation of Huffman coding is more simplistic, and focusses more on the core uses of Huffman codes: generating an encoding system, encoding messages, and decoding messages. The ultimate goal of this project is to prove the identity function, that decoding an encoded messaging will simply return the original message.

2 Implementation

2.1 The Basic Tree Structure

```
Inductive tree : Type :=
| leaf : natprod → tree
| node : nat → tree → tree → tree.
```

The foundation of this implementations lies in the initial tree structure. One of the main aspects of Huffmans algorithm is that the generated tree will always be a full binary tree- meaning that each node will either be a leaf, or will have two subchildren. We reflect this in our implementation by only having constructors for leaves and nodes. Leaves are composed of a pair of natural numbers. The first number in the pair will represent a character in the alphabet, while the second character will represent the probability of that character appearing

2.2 Generating a Tree

```
Fixpoint Huffman_helper' (ls : list tree) : (list tree) :=
  match ls with
```

```
  | nil → []
  | h :: nil → cons h nil
  | h1 :: h2 :: tail →
    treelist_sort ((node ((get_weight h1) + (get_weight h2)) h1 h2) :: tail)
  end.

Fixpoint Generate_Huff_Tree (ls : list tree) (n :nat) : list tree :=
  match n with
  | 0 → nil
  | 1 → ls
  | S n → Generate_Huff_Tree (Huffman_helper' ls) n
  end.
```

Our next step is to generate the encoded tree with the given alphabet. For our implementation, we reflect this with two different functions. The main function takes two arguments: a list of trees, and a natural number that represents the size of the list of trees. In the starting state, it is assumed that the provided list of trees is the alphabet that we with to encode.

The helping function serves the purpose of finding the two trees with the lowest probabilities, and then making them subtrees of a newly generated node, and assigning the weight of said new node to be the sum of its two subtrees. More technically, this is accomplished by sorting the list of trees by weight, and then selecting the first two elements of the newly sorted list. The new node is then generated with the two subtrees, and appended to the remaining list of trees. In the grand scheme of Huffmans algorithm, this function represents a single iteration of the main loop.

The main function essentially controls how many times the helper function iterates over the list. In our case, the helper function should always iterate over the list n - 1 times, where n is the size of our initial alphabet. At the end, it will return a single tree.

2.3 Genrating an Encoded Alphabet

```
Inductive Huff_alpha : Type :=
| Huffman_alpha : nat → nat → list nat → Huff_alpha.
```

We will first define another data type for a character that includes the generated alphabet. It is similar to the pair that we used to define a character, but with the addition of another list of natural numbers. This list will represent the character encoded into binary.

```
Fixpoint Make_Huff_Alpha_helper (t : tree) ( ls :list nat) : list Huff_alpha :=
  match t with
  | node _ lt rt →
    (Make_Huff_Alpha_helper lt (add_end ls 0) ) ++ (Make_Huff_Alpha_helper rt (add_end ls 1) )
  | leaf lf → cons ((Huffman_alpha (fst lf) (snd lf) ls)) nil
  end.
```

```
Fixpoint Make_Huff_Alpha (ls : list tree) : list Huff_alpha :=
  match ls with
  | t :: [] → Make_Huff_Alpha_helper t nil
  | [] → cons (Huffman_alpha 0 0 [ 0;0]) nil
  | h :: t → cons (Huffman_alpha 0 0 [ 0;0]) nil
  end.
```

For generating an encoded alphabet, we will once more use two functions. The first main function takes a single tree. For the flexibility of combining the Tree generating function and this function, the tree will be passed as a list. This main function checks if the list is indeed just a single element, and then passes it onto the helper function. If its not a single element, a default value is returned.

The helper function traverses a tree to each node, passing a list of natural numbers with it. This list is initially empty, but as the tree is traversed, a 0 or 1 is appended to it. When the traversal reaches a note, this list will contain the binary value for that character, and then that character is returned. In the end, a fully encoded alphabet is produced.

2.4 Encoding

```
Fixpoint single_char_encode ( n : nat) (ls : list Huff_alpha) : list nat :=
  match ls with
  | nil → nil
  | h :: t → match h with
                | Huffman_alpha char _ hls → if beq_nat n char then hls else (single_char_encode n t)

                end
  end.

Fixpoint encode (ms : list nat) ( alph : list Huff_alpha) : (list nat) :=
  match ms with
  | nil → nil
  | h :: t → (single_char_encode h alph) ++ (encode t alph)
  end.
```

Encoding a list intro binary is simple. We encode a single character, and then append it to the remainder of the list that will also be encoded. Our single character encoding function takes a character, and an encoded alphabet. The function traverses the alphabet until it finds the character, and returns the characters encoded form. The encoding function takes a list of characters (our decoded message), and an alphabet. It traverses the message, converting each character into binary and finally returns a fully encoded list.

2.5 Decoding

```
Fixpoint decode_helper ( char : list nat) ( alph : list Huff_alpha) : nat:=
  match alph with
  | nil → 0
  | h :: t → match h with
```

```
        | Huffman_alpha c p hls → if ls_equiv char hls then c
                                  else decode_helper char t
        end
  end.

Fixpoint make_decode ( ls : list nat) (alph : list Huff_alpha) (pass : list nat) : (list nat) :=
  match ls with
  | nil → if (ls_equiv pass nil) then nil else cons (decode_helper pass alph) nil
  | h :: t → if in_alpha (pass ++ (cons h nil)) alph then
               (cons (decode_helper (pass ++ (cons h nil)) alph) nil) ++ (make_decode t alph [])
             else make_decode t alph (pass ++ (cons h nil))
  end.
```

Decoding, when the encoded characters are all different lengths is more complicated than encoding. For our approach, we will require an additional auxiliary list. To decode, we will traverse the encoded list, adding binary characters to the auxiliary list as we go. With each addition to the auxiliary list, we will check to see if the list corresponds to an encoded character in an alphabet. If it is, then the list will be cleared, and the decoded character will be appended to the final output. If not, then the list remains and the next character is read in.

Our helper function takes in an encoded character that we already know to be defined in our alphabet, and then matches with the corresponding decoded character in our alphabet. If in error the alphabet happens to be an empty list, or the character isnt actually in the alphabet, a default value will be returned.

The main function takes in an encoded list, the alphabet, and the auxiliary list which should initially be empty. In the case where our current working encoded list is not empty, we check if the current auxiliary list is an encoded character in the alphabet. If its in the alphabet, we find the respective decoded character, and then recursively append the decoded character to the rest of the list that is to be decoded. In addition, the auxiliary list is reinitiated to only contain the next encoded character and passed onto the next function iteration. For the next case, If the auxiliary list is not in the alphabet, then we simply append the next character to be read in, and is passed on for the next iteration. In the event that the encoded message passed is an empty list, we can assume that whatever remains in the auxiliary list is the last character in our message, and thus is decoded and the entire decoded list is returned.

2.6 Reflextion on the Implementation

Overall, I think the current state of the implementation serves more as a prototype, than it is as a fully complete project. I believe there are a few aspects of the implementation that could be expanded or improved on One of the main things is to include support for strings as characters, as opposed to representing them as natural numbers. Furthermore I would represent binary characters as their own type.. Lastly, I would like to have probability represented as a positive real number. With string support, I would also add the feature of calculating the probability for a character by being provided a long string of characters from

an alphabet. In addition, I would like to add a few functions for calculating statistics, such as expected value and standard deviation of code lengths.

```
Fixpoint Huffman_helper (ls : list tree) : (list tree) :=
  match ls with
  | nil → [] (*should never hit this case*)
  | h :: nil → cons h nil
  | h1 :: h2 :: tail →
    Huffman_helper (treelist_sort ((node ((get_weight h1) + (get_weight h2)) h1 h2) :: tail))
  end.

Definition Huffman_tree (ls : list tree) : (list tree) :=
  Huffman_helper ( treelist_sort ls).
```

Another issue that I encountered was when implementing the tree generator, I was originally going to implement the tree generation with only a single function (as seen above), however I ran into the error that Coq could not detect that the function was decreasing. I might look into seeing if theres a way to convince Coq that my function does work.

As for a minor issue, it happened a few times in my implementation where I would pass on a single tree as a list of trees. I realize that may not be the best approach and I may revise the code so when a single tree is expected, just a single tree is passed. Also, the base alphabet is represented as a list of trees, and I believe it would be best if I had base alphabets represented as their own types instead.

3 Final Theorem

Final Theorem. For presenting the theorem, we will take a bottom-up approach, starting with the auxiliary lemmas. While in the end we were unable to prove the final theorem, the current state of the theorem reveals a great deal of information to help improve later implementations.

```
Lemma single_char_iden:
  forall (alpha : list Huff_alpha) (x : nat),
  (char_in_alpha x alpha) = true → decode (single_char_encode x alpha) alpha = cons x nil.
```

Our first lemma is that we need to prove an identity function for a single character, that decoding a character thats encoded will return the original character. We are assuming that that the character is in our alphabet. This is because our single character encoding function was designed with the assumption in mind that the provided character was in the provided alphabet. For our approach, we perform induction on the alphabet. In the case where the alphabet in empty, we observe that this is obscene because we know that our character is in the alphabet, however a character cannot be in an empty alphabet. There was some difficulty with the induction case, as there seemed to be a few contradictions in the context.

Lemma Identity_helper_gen :
forall (l1 l2 : list nat) (alpha : list Huff_alpha),
decode (l1 ++ l2) alpha = (decode l1 alpha) ++ (decode l2 alpha).

This next lemma is to prove that decoding a list appended to another list is the same as decoding a list, and appended it to another decoded list. This can be proved with induction. We get stuck on the inductive case and we are unsure of how to proceed at the moment.

Lemma identity_helper :
forall (x : nat) (hls : list nat) (alpha : list Huff_alpha),
decode (single_char_encode x alpha ++ encode hls alpha) alpha
= decode (single_char_encode x alpha) alpha ++ decode (encode hls alpha) alpha.

This is similar to the more generalized form, except that its referring to an encoded single character being appended to the rest of an encoded list. This holds by the more generalized form.

Theorem identiy :
forall (hls : list nat) (alpha : list Huff_alpha),
decode (encode hls alpha) alpha = hls.

We perform induction on the Huffman list. The base case is simple, and we are able to prove the inductive case by the Identity helper lemma, and by the single character identity lemma. Because the single character identity lemma has the hypothesis that the character is in the alphabet, we must also prove it. Unfortunately, with our current implementation this is not possible at the moment.

3.1 Reflextion on the Final Theorem

The current theorem is a work in progress, and there are still many improvements to be made. For one, more work will need to be done with the single character encoding function to check and see if the character is in the alphabet, as right now its just assumed. I do believe that the generalized form of the helper lemma can be proved, but more work will need to be done on it. Overall, I believe with an improved implementation proving this Theorem is possible.

4 Conclusion

I believe that in the current state, there is a lot more work that could be done to make this project more complete. But overall, My implementation of Huffman Coding in Coq serves as a solid prototype, and includes the most fundamental aspects of the applications of Huffmans algorithm.

Anything below this line will be deleted, this is just for reference

4.1 Subsection1

This is a subsection.

4.2 Subsection2

This is a second subsection.
This is a citation. [**?**]
This is a chunk of code:

```
Definition f(x : nat) := S x.
Definition g(y : nat) := f y.
```

This is inline code `Fixpoint f(x : nat) := ...` typeset within a line of text.

Para1. This is a paragraph, or subsubsection.

5 Conclusion

but anything below this line won't be deleted

References