Class Notes CS 3137

1 Creating and Using a Huffman Code. Ref: Weiss, page 433

1. FIXED LENGTH CODES: Codes are used to transmit characters over data links. You are probably aware of the ASCII code, a fixed-length 7 bit binary code that encodes 2^7 characters (you can type 'man ascii' on a unix terminal to see the ascii code). Unicode is the code used in JAVA, which is a 16 bit code.

2. Since transmitting data over data links can be expensive (e.g. satellite transmission, your personal phone bill to use a modem, etc) it makes sense to try to minimize the number of bits sent in total. Its also important in being able to send data faster (i.e in real time). Images, Sound, Video etc. have large data rates that can be reduced by proper coding techniques. A fixed length code doesn't do a good job of reducing the amount of data sent, since some characters in a message appear more frequently than others, but yet require the same number of bits as a very frequent character.

3. Suppose we have a 4 character alphabet (ABCD) and we want to encode the message **ABACCDA**. Using a fixed length code with 3 bits per character, we need to transmit 7*3=21 bits in total. This is wasteful since we only really need 2 bits to encode the 4 characters ABCD. So if we now use a 2 bit code (A=00, B=01, C=10, D=11), then we only need to transmit 7*2=14 bits of data. We can reduce this even more if we use a code based upon the frequencies of each character. In our example message, A occurs 3 times, C twice, B and D once each. If we generate a variable length code where A=0, B=110, C=10 and D=111, our message above can be transmitted as the bit string:

   ```
   0  110  0  10  10  111 0 = 13 bits
   A   B   A   C   C   D  A
   ```

   Notice, that as we process the bits each character's code is unique; there is no ambiguity. As soon as we find a character's code, we start the decoding process again.

4. CREATING A HUFFMAN CODE: The code above is a *Huffman* code and it is optimal in the sense that it encodes the most frequent characters in a message with the fewest bits, and the least frequent characters with the most bits.

5. It is also a prefix-free code. There can be no code that can encode the message in fewer bits than this one. It also means that no codeword can be a prefix of another code word. This makes decoding a Huffman encoded message particularly easy: as the bits come by, as soon as you see a code word, you are done since no other code word can have this prefix. The general idea behind making a Huffman code is to find the frequencies of the characters being encoded, and then making a Huffman tree based upon these frequencies.

6. The method starts with a set of single nodes, each of which contains a character and its frequency in the text to be encoded. Each of these single nodes is a tree, and the set of all these nodes can be thought of as a "forest" - a set of trees. We can put these single node trees into a priority queue which is prioritized by decreasing frequency of the characters occurrence. Hence, the node containing the least frequent character will be the root node of the priority queue.

7. Algorithm to make a Huffman tree:

(a) Scan the message to be encoded, and count the frequency of every character

(b) Create a single node tree for each character and its frequency and place into a priority queue (heap) with the lowest frequency at the root

(c) Until the forest contains only 1 tree do the following:

- Remove the two nodes with the minimum frequencies from the priority queue (we now have the 2 least frequent nodes in the tree)
- Make these 2 nodes children of a new combined node with frequency equal to the sum of the 2 nodes frequencies
- Insert this new node into the priority queue

8. Huffman coding is a good example of the separation of an Abstract Data Type from its implementation as a data structure in a programmijng language. Conceptually, the idea of a Huffman tree is clear. Its implementation in Java is a bit more challenging. Its always important to remember that programming is oftentimes creating a mapping from the ADT to an algorithm and data structures. Key Point: If you don't understand the problem at the ADT level, you will have little chance of implementing a correct program at the programming language level.

   A possible Java data structure for a huffTreeNode is:

```
public class huffTreeNode implements Comparable {
  public huffTreeNode(int i, int freq, huffTreeNode l, huffTreeNode r){
    frequency = freq;
    symbol = i;
    left = l;
    right = r;
  }
  public int compareTo(Object x){
     huffTreeNode test= (huffTreeNode)x;
     if (frequency>test.frequency)
        return 1;
     if (frequency==test.frequency)
        return 0;
     return -1;
  }
int symbol;
huffTreeNode left,right;
int frequency;
}
```

9. MAKING A CODE TABLE: Once the Huffman tree is built, you can print it out using the printtree utility. We also need to make a code table from the tree that contains the bitcode for each character. We create this table by traversing the tree and keeping track of the left branches we take (we designate this a 0) and the right branches we take (the opposite bit sense - a 1) until we reach a leaf node. Once we reach a leaf, we have a full code for the character at the leaf node which is simply the string of left and right moves which we have strung together along with the code's length. The output of this procedure is an array, indexed by character (a number between 0 and 127 for a 7 bit ascii character), that contains an entry for the coded bits and the code's length.

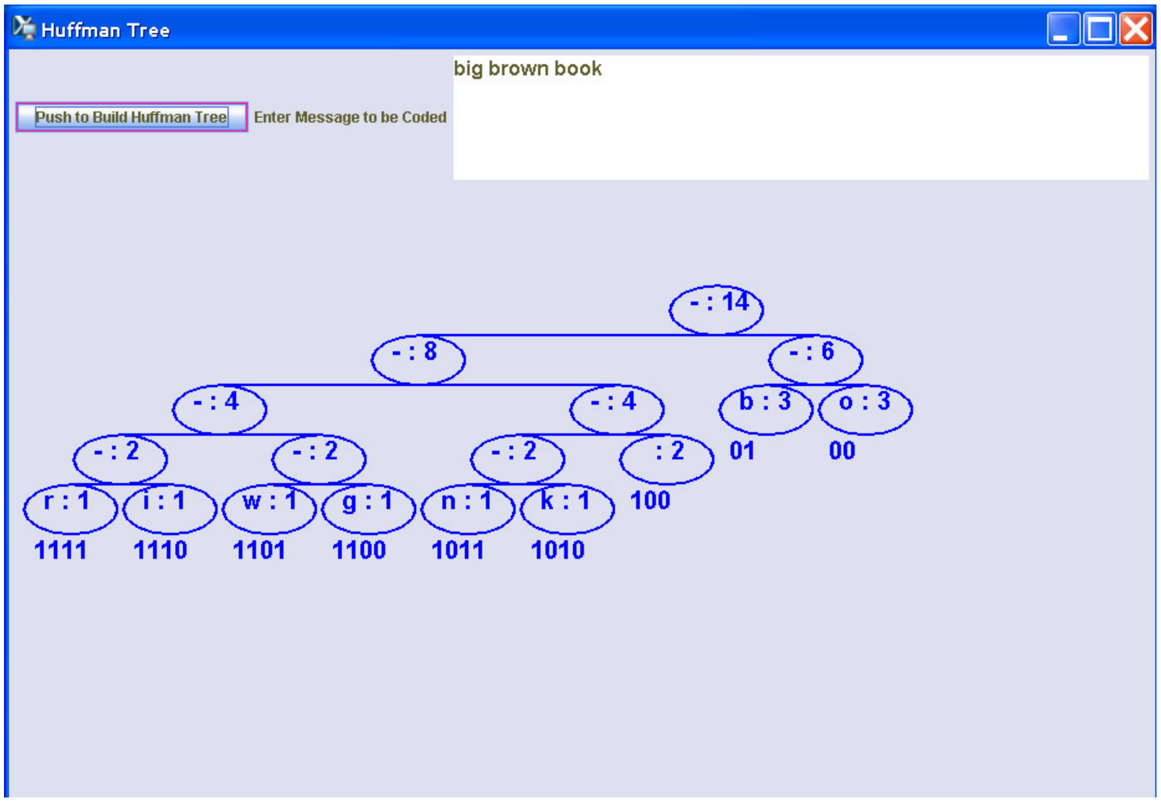

Figure 1: Huffman Coding Tree for string: big brown book

Here is a test run of a Huffman Coding program

```
Here is the original original message read from input:

big brown book

Here is the Code Table
CHAR ASCII FREQ BIT PATTERN CODELENGTH
        32   2   100          3
   b    98   3   01           2
   g   103   1   1100         4
   i   105   1   1110         4
   k   107   1   1010         4
   n   110   1   1011         4
   o   111   3   00           2
   r   114   1   1111         4
   w   119   1   1101         4

Here is the coded message,

011110110010001111100110110111000100001010
b i   g   _  b r   o w   n   _  b o o k
14 characters in 42 bits, 3 bits per character.
```

2 How to Write a Program to Make a Huffman Code Tree

To make a Huffman Code tree, you will need to use a variety of data structures:

1. An array that encodes character frequencies: FREQ_ARRAY
2. A set of binary tree nodes that holds each character and its frequency: HUFF_TREE_NODES
3. A priority queue (a heap) that allows us to pick the 2 minimum frequency nodes: HEAP
4. A binary code tree formed from the HUFF_TREE_NODES: This is the HUFFMAN _TREE

Here is the Algorithm:

1. Read in a test message, separate it into individual characters, and count the frequency of each character. You can store these frequencies in an integer array, FREQ_ARRAY, indexed by each characters binary code. FREQ_ARRAY is data structure #1. For debugging, you should print this array out to make sure the characters have been correctly read in and counted.
2. You are now ready to create a HEAP, data structure #3. This heap will contain HUFF_TREE_NODES which are data structure #2. When we make the heap, using the insert method for heaps, we create a HUFF_TREE_NODE for each individual character and its frequency. The insert uses the character frequency as its key for determining position in the heap. The HUFF_TREE_NODE as defined in your class notes on Huffman trees implements the Comparable Java interface, so you can compare frequencies for insertion into the HEAP. Remember, you ALREADY HAVE THIS CODE FOR MAKING A HEAP FROM THE WEISS TEXT, GO USE IT! For debugging, you may want to print the heap out and see if its ordered correctly by using repeated calls to deleteMin(). Be careful, as this debugging of the heap also destroys the heap, so once you get the heap correct, remove this debugging code!
3. The HUFFMAN_TREE, data structure #4, is formed by repeatedly taking the 2 minimum nodes from the HEAP, creating a new node that has as its character frequency the sum of the 2 nodes frequencies, and these nodes will be its children. Then we place this node back into the HEAP using the insert function.
4. The method above takes 2 nodes from the HEAP, and inserts a new node back into the HEAP. Eventually, the HEAP will only have 1 item left, and this is the HUFFMAN_TREE, data structure #4.
5. The HUFFMAN_TREE is just a binary tree. So all the binary tree operations will work on it. In particular, you can print it out to see if its correct. You can also number the tree nodes with x and y coordinates for printing in a window using the algorithm in exercise 4.38 of Weiss.

3 Proving Optimality of a Huffman Code

1. We can think of code words as belonging to a a binary tree where left and right branches map to a 0 or 1 bit encoding of the code. Every encoded character will be a node on the tree, reachable in a series of left-right (meaning a 0 or 1 bit) moves from the root to the character's node.

2. Huffman codes are **prefix-free** codes - every character to be encoded is a leaf node. There can be no node for a character code at an internal node (i.e. non-leaf). Otherwise, we would have characters whose bit code would be a subset of another character's bit code, and we wouldn't be able to know where one character ended and the next began.

3. Since we can establish a one-to-one relation between codes and binary trees, then we have a simple method of determining the optimal code. Since there can only be a finite number of binary code trees, we need to find the optimal prefix-free binary code tree.

4. The measure we use to determine optimality is to be able to encode a specified message with a set of characters and frequency of appearance of each character in the minimum number of bits. If we are using a fixed length code of say 8 bits per character, then a 20 character message will take up 160 bits. Our job is to find a new code who can encode the message in less than 160 bits, and in fact in a minimum number of bits. The key idea is to use a short code for very frequent characters and longer code for infrequent characters.

5. We need to minimize the number of bits for the encoding. A related measure is the Weighted Path Length (WPL) for a code tree.

$$WPL \;=\; \sum_{i=1}^{N} F_i * L_i \quad , \quad Num - of - Bits \;=\; \sum_{i=1}^{N} N * F_i * L_i \tag{1}$$

   Let F_i be the frequency of character i to be encoded (expressed as a fraction). Let L_i be the length in bits of the code word for character i. We can recognize that the length L_i is simply the number of levels deep for each character in the binary code tree. Each 0 or 1 in the code word is equivalent to going to the next level of the tree. So a 3 bit code 010 would have a leaf at the third level of the tree which is the code for that character. $Num - of - Bits$ just adds up each character's contribution to the total length of the encoded message based upon the frequency of occurrence. $N * F_i$ is simply the number of occurrences of character i in a message of N characters, and L_i is the length of the code for that character in bits. Since N is a constant, we can delete it to get a more compact measure WPL.

6. To minimize WPL, we can only affect the term L_i which is the length of the code for each character i. We need to show how to build a code tree that has minumum Weighted Path Length. We do this proof by induction, and we also introduce a new tree. Define T_{opt} as the optimal code tree that has the minimum WPL. Since there can only be a finite number of code trees, we could build them all and find this one, so it must exist. The core of the proof is to show that a Huffman code tree T_{huff} is identical to this tree, and is therefore also the optimal code tree.

7. We will first prove the **basis** case for the minimal encoding of 2 characters is identical for T_{opt} and T_{huff}. We will show that the two least frequencies must be the deepest nodes in the tree. For T_{opt}, we know that optimal code tree for two characters must be 2 siblings off of a single root node. To see this, suppose there was an extra node in the tree. This must be an interior node with 1 child. But if we simply moved the single child up to the interior node's position, we would have a shorter code. Hence we cannot have a single child interior node and the 3 node tree (root and 2 siblings) is optimal. For T_{huff}, we can see that the construction process of the Huffman algorithm will generate exactly the same tree for a 2 character code. So we now have proved that T_{opt} and T_{huff} are identical for a 2 character code. NOTE: we can actually only show that they are identical up to a transposition of 0's or 1's since we have a choice of

which node is a left child and which is a right child. However, the code length L_i will be the same, and this is what we are trying to minimize.

8. Now, perform the **induction** step. We assume that for a code of N characters, T_{opt} and T_{huff} are identical. Now show that it is true for a code of $N+1$ characters.

9. In the trees of $N+1$ characters, we know that the two characters with the smallest frequencies F_1 and F_2 are siblings at the deepest node by the basis case. Now, take these two deepest leaves which are siblings off of each tree T_{opt} and T_{huff}. Replace these two leaves with a *single* node that encodes the sum of the frequencies of these two characters, $F_1 + F_2$. We can then think of these two characters as a single meta-character encoding characters 1 and 2. We can define the new trees T^*_{opt} and T^*_{huff} which are the trees which encode N characters formed by removing the two deepest sibling leaves and replacing them with a single node. So we can now recalculate the WPL for both T_{opt} and T_{huff}:

$$WPL(T_{opt}) = WPL(T^*_{opt}) + (F_1 * L_1) + (F_2 * L_2) \tag{2}$$

$$WPL(T_{huff}) = WPL(T^*_{huff}) + (F_1 * L_1) + (F_2 * L_2) \tag{3}$$

Since T_{opt} is defined to be the minimum cost tree for $N+1$ characters, we are assuming that:

$$WPL(T_{opt}) < WPL(T_{huff}) \tag{4}$$

substituting (2) and (3) into (4) we get:

$$WPL(T^*_{opt}) + (F_1 * L_1) + (F_2 * L_2) < WPL(T^*_{huff}) + (F_1 * L_1) + (F_2 * L_2) \tag{5}$$

$$WPL(T^*_{opt}) < WPL(T^*_{huff}) \tag{6}$$

However we assumed in the induction step that T_{huff} was minimal cost on N nodes, and this is exactly $WPL(T^*_{huff})$. Therefore, it cannot be true that $WPL(T^*_{opt}) < WPL(T^*_{huff})$, and therefore we have shown that the Huffman code tree on $N+1$ nodes is of no less cost than T_{opt}, and must also be minimal.