Nadav Zinger
ECE 368
11/3/2015

Project 2 Report: Huffman Coding

Introduction (project description):

In this project, our objective was to write two different programs, huff.c and unhuff.c. huff.c takes an argument of an input ascii text file. The goal of this program is to create a Huffman tree and write a compressed binary file with a header and a sequence. The header is instructions on how to re-create the Huffman tree, and the sequence is the path for each character in the original input text file. The output file will have ".huff" appended to the input file name. The second program is unhuff.c. This program takes an input of a .huff file and outputs the original ascii text file. Therefore the output of unhuff.c should be the same as the input of huff.c and the input of unhuff.c is the output of huff.c. unhuff.c reads binary .huff input file, it first encounters the header and re-constructs the Huffman tree. The program then reads the sequences, traverses the tree according to them and outputs the original ascii text file.

Huff.c Solution:

The first step of writing huff.c is to create the tree. To do this, I used the concept of counting the frequency of each character in the input file, sorting them based on that frequency (ascending order), then combining low frequency characters together and re-inserting their combined frequency into the sorted queue until all frequencies in the queue are combined together. To keep track of the character index and its frequency, I made an array with index 0-255 and initialized all values in it to zero. I then parsed through every character the input file and incremented its index for every encounter so that the array would look like this: Array[ascii value] = frequency. Once I had this array I then made a Queue using a linked list that would not only keep track of the index and frequency but sort all characters (represented by the index) based on their frequencies. The linked List stored 3 key values: index,frequency, and a tree. At first the tree value of every link in the list is null. I then parsed through the linked list and created tree nodes for the links that had their tree null. I would look at 2 links at a time, pop those two links create tree nodes for them then combine those nodes and add their frequencies to a parent node. My tree data structure had the letter/index value and a frequency (and left/right ofcourse). I then constructed a new link (node) for the linked list with the new built tree as the the tree value in the link. The new built tree has its root node letter as -1 in order for me parse through it later and see if its a leaf node. The leaf nodes will have the character index as the letter. Once this new link is created it is inserted into the original linked list, the combined frequencies of the two links before it will push the new link further into the linked list. This means after the first two links are popped and combined not all the links in the sorted linked_list will have null trees. Eventually their will be one link left with the completed Huffman tree within it. Once the tree is built I started to encode (compress) the output file. The first part was to create the header, I decided to use post order traversal to give directions on how to re-build the tree (easier for me to unhuff). Since I used post order traversal, it was extremely difficult for me to write a bit based header so instead my header was char based which decreases my possible compression ratio. Using post-order recursive traversal, I wrote a 1 every time I encountered a leaf node, followed by the character at that leaf node. Every time I hit a non-leaf node I wrote a 0 into the file. After my traversal I put a new line character to distinguish between header and sequence. For the sequence I created a code book which was a 2-d array with the first column being characters in the leaf node and the rest of the columns were its path (0 and 1's). so the first row would

look something like this: g0001-1. The -1 signifies the end of the sequence if the sequence size is less than the max height of the tree. Before traversing my code book, I wrote a int size (after the new line I mentioned before) into the output file. This size represents the number of characters int the ascii input text file and will help me rewrite my characters in unhuff.c. I then traverse my codebook by traversing the input file again. I match the row to the correct index value and then go through its column converting the 0's and 1's to bits and storing them in the correct bit spot of a byte using bit shifting and bit masking. Once the byte is full I write it to the file.

Unhuff.c Solution:

For my unhuff.c I traversed the input file until I hit a character that not a 1 or a 0, this signifies the end of the header. While traversing the input file I use a similar method to reconstruct my tree in huff.c. The way the header works is a 1 is always followed by a character. If I hit a 1 I grab the character and put it in a tree node. I store this node in a linked list. If I hit a 0 I pop two links off a linked list and combine their tree nodes (leafs and roots). It eventually always works out to end up with a single link and the complete Huffman tree within it. Once the tree is reconstructed I read the size in order to set a loop condition while (size > 0). The rest of the file is the compressed sequence. I read in a byte and then enter my loop. I get each bit from the byte read in and traverse the tree based on the bits I read. Once all the bits in the byte are read, I read in another byte from the file. I traverse my tree with the 1's and 0's from the bits I read and when I hit a leaf node (node->letter != -1 → explained in huff.c solution), I output that character onto the output file and decrement size. When size is 0 the loop quits and all the characters from the orignal ASCII input are rewritten to the output file.