Dan Suciu
Project 2 Report

huff.h – huff.h contains prototypes of functions used and the definition of the structure.

huff.c – My Huffman source code works by first reading in every character from the file one by one and incrementing a 255 element array of weights initialized to 0 by 1 at the index of the character read cast as an int. In this way we can use the index of the weight array as the integer definition of the character to which the weight corresponds. My code then goes through the array of weights and creates a node of the structure for each non-zero weight, meaning the character appears in the input file somewhere. As each node is created, it is inserted into the linked list in ascending order. After every element in the weight array has been looked at, an extra node is added representing the pseudo EOF character, being NULL with a weight of 1. Then my code creates the Huffman tree out of the ascending linked list. The way this is done is it merges the first two nodes in the linked list, being the two with the smallest weights, and creates a parent for them where the first node is the left child and the second node is the right child. The parents weight is equal to the sum of the two children's weights, and the new parent node is placed in the linked list in its proper place according to ascending order of weights. This happens until there is only one node in the linked list, which is the root of the Huffman tree. Now that we have the Huffman tree, we can generate the binary encodings based on the root to leaf paths. My function to generate the binary encodings recursively moves through the tree in a pre-order traversal manner, and twiddles a bit onto the 32-bit long encoding variable, which is initialized to 0, depending on the direction moved, a 0 for a left move and a 1 for a right move. Once it finds a leaf node, it saves the encoding into an array of longs, and saves the number of moves to get to the leaf, or sigbits, into an array of integers, both of which are indexed by the integer value of the ASCII character at the leaf. When the function recurs backwards, the bits twiddled onto the encoding are twiddled back off and the number of sigbits is decremented, thus taking into account the movements backwards in the tree after a leaf is found. Once all of the encodings are generated we are ready to write our compressed information. The first thing to do is write the header, which will be a post-order traversal of the binary tree, during which a 0 is written if a non-leaf node is traversed and a 1 is written if a leaf node traversed. After a leaf node is traversed, the immediate next 8 bits is the ASCII value for the char at that node. Once every node in the tree has been represented, we need to make sure that unhuff can tell that the header is finished. We write one extra 0 to the header to assure that the read_header function in unhuff can reach its terminating condition, and then we pad whatever space is left in the byte with 0's. We then write the number of characters read from the original file to the end of the header, which will be used to control our decompression. Then we write the newline '\n' character, which marks the end of the header. Now we write the compressed data to the file, a key part of which is the write_bit function. The write_bit function has a current byte, which is the byte we are packing until it is full, meaning that we can go ahead and write it to the file, and it has a tracker for what the next available position in the byte is, which is where we can put the next bit to be written to the file. When writing our compressed information, we read in once character at a time from the input file and go to that characters ASCII value index of the encoding and sigbits arrays. We store those values and shift the encoding so that the bits are in the far left position. We use a mask with a 1 in the far left position to decide what value the encoding bit has in that far left position. Depending on that we either send a 1 or a 0 into the write_bit function, and then we decrement the sigbits for that encoding and left shift the encoding. We know we have used every bit in the encoding when sigbits reaches 0. We are done

writing compressed data when the read in the EOF character. Then we can write in our own pseudo EOF character, which uses the same logic as above where we know the indices of the encoding and sigbits = 0. Now we pad the file with zeros and are done writing compressed data. We close our files and destroy our tree.

unhuff.h – unhuff.h contains prototypes of functions used and the definition of the structure.

unhuff.c – My unhuff source code works by first reading the header at the beginning of the compressed file. The header is read bit by bit using the read_bit function. The read_bit function works by reading a whole byte from the file, and in a similar way to huff.c keeps track of the current byte and the place of the next bit to read. It isolates the next bit to read with a right shift and a mask with a 1 in the far right bit. It increments the position of the next bit to read and saves the current bit in the global variable which will either be equal to 1 or 0, since the bit read is in the far right position. The read_header reads the header bit by bit. If a 1 is read, which will always be the case in the first pass since it is based on a post-order traversal, we are at a leaf node, so it reads in the next 8 bits which correspond to the ASCII value of the character at that leaf and creates a node. It adds this node with the character data to the end of a linked list, meaning it was one of the most recent added to the list. When a 0 is read from the header, it means that we are at a non-leaf node, and we must create a parent node with our linked list. We call merge two, which takes the last two nodes in the linked list and creates a parent, where the second to last node is the left child and the last node is the right child. The parent takes the place of the children in the linked list. We know we are done reading the header when the next value of the head of the linked list is NULL, meaning that there is only one node in the linked list and all the others have been merged into the tree in their proper locations. The extra 0 written into the header assures that we check to see if head's next value is NULL before reading more bits. Now we have read the whole header and we can ignore the extra padding 0's and return the root to our Huffman tree. We then read in the number of characters that were compressed originally, and the newline to indicate that we have completely finished reading the header. Now we write the decompressed information into the output file. We do this by starting at the top of the tree and moving either left or right depending on if a 0 or a 1 bit were read from the compressed file respectively. When we reach a leaf node in the tree, we write the character in that leaf to the decompressed file, decrement the amount of characters to be decompressed, and start at the top of the tree again. We do this until the amount of characters to be decompressed reaches 0, meaning we have decompressed every character we originally compressed. We are done decompressing the huff file and can destroy the tree and close the files.

Performance:

filename	huff runtime	unhuff runtime	original file size	compressed file size	ratio
text0.txt	0m0.001s	0m0.001s	4 bytes	10 bytes	2.5
text1.txt	0m0.001s	0m0.001s	8 bytes	21 bytes	2.65
text2.txt	0m0.001s	0m0.001s	155 bytes	128 bytes	0.826
text3.txt	0m0.306s	0m0.261s	4.2 MB	3.1 MB	0.738
text4.txt	0m0.624s	0m0.504s	7.3 MB	5.2 MB	0.712
text5.txt	0m0.948s	0m0.790s	10.5 MB	7.3 MB	0.695