ECE 36800: Data Structures and Algorithms
Project 2 Report
Kunwar Digraj Singh Jain

Homework ID: 536358

Introduction

The project that we are worked on was based on Huffman compression and decompression which is a “home-made” zip and unzip program. It takes in a file and removes all the bits which aren’t necessary and stores them in a file in such a way that the original file can be generated from new compressed file by utilizing the header data which is provided in the compressed file. Such type of coding is known as greedy algorithm since we take up a little extra space to reduce a lot of other space. The unnecessary bits that are removed are the ones that are found in the binary values of 1 and 0. For these two numbers, 1 is compressed to 0x01 and 0 is compressed to 0x00 and only 1 bit is stored for either of them. This way, we reduce the space by 7 bits.

Compression

For the compression algorithm, we first read the file into an array and determine the recurrence of each character in the file. For example, if the file is “go go gophers”, then character g occurs 3 times, o and space occurs 2 times and all other characters 1 time. Therefore, the weight of g is 3, space is 2 and so on for other characters. Using these weights, we create a priority binary tree for all the characters. The characters with the lowest weights are combined and made into a binary tree with the character with the lower ascii value on the left node of the new root node and the one with the higher ascii value on the right node of the node. The weight of the new node is the sum of the weights of its two leaf nodes. The new node is placed in the appropriate place and the same process continues and the binary tree is constructed. It is done so, so that the characters that are visited frequently do not take much time to be accessed thus reducing the time needed to compress and decompress the file.

Once the binary tree is prepared, we are ready to write a new file which is the compressed file. First, we need the binary tree in the form of a table. So for example, to get to g, we need to go left-left-left-right, in the table g will have the value 0001. Here, the 0001 are not characters but bits. So we need 4 more bits to make a character and then write that character to the file. Second, we need to add a header to the file. Here, we save the binary tree that we created in the file. This can be recreated during decompression. After the header is saved, we save the whole file go go gophers in the form of bits as shown above. So if o is at left-left-right-right and the space is at right-left-left, the starting bit stream would be 00010011 10000010 011, which is just go go. Here, we used 2 characters and 3 bits to store 5 characters. The compression is much bigger in bigger files since the number of characters repeated is increased. To make the compressed file, we first insert the binary tree as header information in to the output file. Once the header information is

stored, we insert a 0xFF character which acts as a barrier between the header information and the compressed file. This is how the file is compressed.

For the binary tree, an example output is shown below. In the table, it shows how a character can be accessed once the binary tree is constructed.

G	00
O	01
P	1110
H	1101
E	1100
R	1111
S	100
	101

In the table above, to access H, we need to go right-right-left-right. Similarly, all the elements can be accessed once the binary tree is constructed and such a table has been formed.

Decompression

In the decompression program, the header file is read and the original binary tree is reconstructed. We stop constructing the tree once we encounter the 0xFF character. Once the tree is reconstructed, we start reading the bits for the reconstruction of the original file. For every 0 bit that is encountered, we go left in the binary tree and for every 1 bit encountered, we go right in the binary. Once a leaf node is encountered, i.e. node->left and node->right are NULL, the character in that node is printed and we return to the top node and start the same process again. We keep doing this till the end of the file is reached where we say that the decompression is completed and we exit the program. This is how the file is decompressed.

Due to time limitation, I was unable to properly optimize my code because of which it only works for small files.