Assignment 4 Report

Robert Millward, William Corlette,

Philip Rajala, Brandon Forster

COP 3503, 11/25/12

The data structures we use are hash-tables, priority queue, binary trees

Our general runtime for the entire Huffman process takes O(nlg(n))

**Assignment Implementation and make files**

We have our program pass the Huffman tree by appending it to the top of an encoded file.

Included are two make files. Upon the call ‘./huffmanencode <filename>’ Encoder.java is run. When ‘./huffmandecode<filename>’ is inputted, our make file launches Decoder.java. Regardless of the input, Launcher.java is launched to read in the file to be encoded and interpret whether to pass it to Encoder.java or Decoder.java.

In here we read in the characters which has a runtime of O(n), n being each character

**Encoding**

Upon reaching Encoder.java, a frequency table is constructed although the table is merely an array implementation where the index is the ASCII value. Upon completion of the frequency table, a priority queue is constructed to be used in the main Huffman encoding algorithm. Lastly, encoder performs a very important task by creating a separate text file containing the tree. This tree is then picked up by decoder and used to decode the encoding. This allows the two separate java files to ‘talk’ to each other.

//make priority queue

For(i=0 to ASCII alphabet size)

If(frequency table[i]>0)

Add a new node to the priority queue

Runs in 256 or O(1)

//make trees out of frequencies

while(priority queue size>1)

remove left node from priority queue

remove right node from priority queue

make the tree from these nodes

Runs in O(1)

public void fillHash(hash array, current node, string code)

If(current node is not a leaf)

fillHash(hash array, current node, string code+0)

fillHash(hash array, current node, string code+1)

else(when node is a leaf)

look up hash table for binary value of char

Runs in O(lg(n))

public void printTree(current node, string Hash)

if(current node isn’t empty)

print the node and all its children using preorder recursion

Runs in O(n) time

**Decoding**

Upon reaching Decoder.java, decoder copies down the tree printed in the text file by Encoder.java. It then traverses the tree looking for leaf nodes. It traverses the tree using the encoded file. If it hits a char ’0’ it traverses left down the tree if it hits a char ‘1’, it heads right down the tree. Upon reaching a leaf node, it looks into the node class to find the character.

For(i=0 to this.getText.length i++)

If(this.getText.char==0)

Traverse left

Else

Traverse right

If(temp.isLeaf)

Write the char id at that leaf

else

go to the top of the tree.

Runs in O(nlg(n))

**Huffman ‘compression’**

Because this is an academic exercise and not an actual compression algorithm, we are not actually compressing the file since the ASCII values are not being turned into true binary but rather, more characters. What this means is that the ‘compression’ is actually going to generally be more than the actual size of the file. For example, if the character ‘C’ is represented in binary as 11000, than our file actually has the binary represented as 5 characters (5\*8 bits) whereas the original represents it only as 1 character (1\*8 bits).

**One of our compressed ratios:**

Caesar:

Actual encode:

574576 / 122153 ~ 4.703

Pretending our chars are bits and dividing by 8

(574576/8) = 71822

71822 / 122153 ~ 0.585

Hamlet

Original file: 88.1 KB

Encoded file: 416.9 KB

416.9/8=52.1125KB if compressed

Ratio:45.9%

Macbeth

Original file: 106.7 KB

Encoded file: 502 KB

502/8=62.75 KB if compressed

Ratio: 58.81%

Othello

Orginal file:59.1 KB

Encoded file:276.6 KB

276.6/8=34.575 KB if compressed

Ratio: 58.503%