TK6434 – Data Structures and Algorithms

Project #2

You are urged to work in groups of two. Each group should submit ONE program per group. Be sure to include name and matric number of each person in your group in the README file that belongs with the submission. *Do not wait until the last minute to start this assignment*.

The resulting program will be a complete and useful compression program although not, perhaps, as powerful as standard programs like *compress* or *zip* which use slightly different algorithms permitting a higher degree of compression than Huffman coding.

Because the compression scheme involves reading and writing in a bits-at-a-time manner as opposed to a char-at-a-time manner, the program can be hard to debug. In order to facilitate the design/code/debug cycle, you should take care to develop the program in an incremental fashion. If you try to write the whole program at once, you probably will not get a completely working program!

Design, develop, and test so that you have a working program at each step.

Build a program by adding working and tested pieces.

To compress you should use the Huffman coding algorithm. The algorithm has four steps. You should understand each of these steps before starting to code. The first three steps are the basis for Part I which you should design, implement, and test before proceeding to the next part and step four.

First, the process of Huffman compression is discussed, then some details about the program follow.

1. To compress a file, count how many times every character occurs in a file. These counts are used to build weighted nodes that will be leaves in the Huffman tree. The word *character* is used, but we mean *8-bit chunk* and this chunk-size could change.

2. From these counts build the Huffman tree. First create one node per character, weighted with the number of times the character occurs, and insert each node into a priority queue. Then choose two minimal nodes, join these nodes together as children of a newly created node, and insert the newly created node into the priority queue. The new node is weighted with the sum of the two minimal nodes taken from the priority queue. Continue this process until only one node is left in the priority queue. This is the root of the Huffman tree.

3. Create a table or map of characters (8-bit chunks) to codings. The table of encodings is formed by traversing the path from the root of the Huffman tree to each leaf, each root-to-leaf path creates an encoding for the value stored in the leaf. When going left in the tree append a zero to the path; when going right append a

one. All characters/encoding bit pairs may be stored in some kind of table or map to facilitate easy retrieval later.

4. Finally, read the input file a second time. For each character/8-bit chunk read, write the encoding of the character (obtained from the map of encodings) to the compressed file.

The Huffman Compression Algorithm

Introduction

The Huffman Compression algorithm is an algorithm used to compress files. It does this by assigning smaller codes to frequently used characters and longer codes for characters that are less frequently used.

Code: A code is a sequence of zeros and ones that can uniquely represent a character.

A file is a collection of characters. In a file certain characters are used more than others. The number of bits required to represent each character depends upon the number of characters that have to be represented. Using one bit we can represent two characters. i.e. 0 represents the first character and 1 represents the second character. Using two bits we can represent 2^2 i.e 4 characters.

00 - first character
01 - second character
10 - third character
11 - fourth character.

In general if we want to represent n characters we will need 2^n bits for representing one character. The ASCII code uses 7 bits to represent a character. Therefore 2^7= 128 bits can be represented using ASCII code.

The ASCII code and the above codes used above to represent characters are known as fixed length codes. This is because each character has the same bit length. i.e the number of bits required to represent each character is the same. In ASCII code every character requires 7 bits. Using variable length codes for each character we can reduce the size of a file considerably. *By assigning smaller codes for more frequently used characters and larger codes for less frequently used characters, a file can be compressed considerably.*

EXAMPLE-1

For example consider a file consisting of the following data

AAAAAAAAAABBBBBBBBBCCCCCCDDDDDEE

Frequency: *The number of times a character occurs in a file is usually called its frequency.*

The frequency of

A is 10
B is 8
C is 6
D is 5
E is 2

If each character is represented by using three bits then number of bits require to store this file will be

3*10 + 3*8 + 3*6 + 3*5 +3*2 = 93 Bits.

Now suppose we represent the character

A by the code 11
B by the code 10
C by the code 00
D by the code 011
E by the code 010

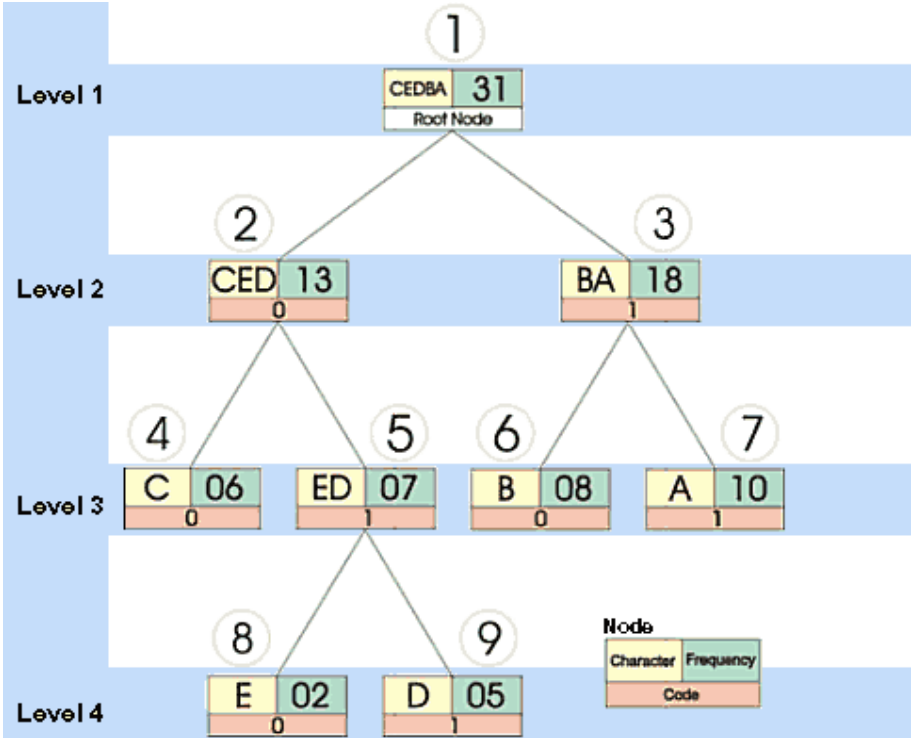
then the size of the file becomes 2*10 + 2*8 + 2*6 + 3*5 + 3*2 = 69 bits. A certain amount of compression has been achieved. In general much higher compression ratios can be achieved by using this method.

As you can see frequently used characters are assigned smaller codes while less frequently used characters are assigned larger codes. One of the difficulties of using a variable-length code is knowing when you have reached the end of a character in reading a sequence of zeros and ones. This problem can be solved if we design the code in such a way that no complete code for any character is the beginning of the code for another character. In the above case A is represented by 11. No other code begins with 11. Similarly B is assigned the code 00. No other code begins with 00. Such codes are known as prefix codes.

In general, **we can attain significant savings if we use variable-length prefix codes that take advantage of the relative frequencies of the symbols in the messages to be encoded**. One particular scheme for doing this is called the Huffman encoding method, after its discoverer, David Huffman.

A Huffman code can be represented as a binary tree whose leaves are the characters that are encoded. At each non-leaf node of the tree there is a set containing all the characters in the leaves that lie below the node. In addition, each leaf is assigned a weight (which is the frequency of the character), and each non-leaf node contains a weight that is the sum of all the weights of the leaves lying below it.

For Example-1 the Huffman tree is as shown below



Each non leaf node of the tree has two child nodes , the left child node and the right child node. *The non leaf node is known as the parent node of these two child nodes.* Similarly the parent of a parent node is the grandparent of the child nodes. The parent, grand parent, great grandparent etc. are collectively called the ancestors of a child node. The child nodes are called the descendants of the ancestors. In the tree above node 3 is the parent of nodes 6 and 7. Node 1 is the parent of 3. It is also the grandparent of nodes 6 and 7. Generally nodes 3 and 1 are the ancestors of nodes 6 and 7. Nodes 6 and 7 are the descendants of nodes 3 and 1.

Note that

1. ***if n characters are present in a file then the number of nodes in the huffman tree is*** *2n-1.*
2. ***If there are n nodes in a tree then there can be at most*** *(n+1)/2* ***levels, and at least*** *$\log_2(n+1)$ levels.*
3. ***The number of levels in a huffman tree indicates the maximum length of code required to represent a character.***

The codelength of a character indicates the level in which the character lies. *If the code length of a character is n then it lies in the* ***(n+1)th*** *level of the tree.*

For example the code length of character D is 011. the code length is 3. Therefore this character must lie in the 4th level of the tree.

The code for each character is obtained by starting from the root node and traveling down to the leaf that represents the character. When moving to a left child node a '0' is appended to the code and when moving to a right child node a '1' is appended to the code.

To get the code for the character 'A' from the tree,

1. we first start at the root node (i.e node 1).

2. Since the character 'A' is the descendant of the right child node (We decide which branch to follow by testing to see which branch either is the leaf node for the character or is its ancestor) we move to the right and append a '1' to the code for character 'A'.

3. Now we are on node 3. The leaf node for character 'A' lies to the right of this node, so we again move to the right and append a '1' to its code.We have now reached node 7 which is the leaf node for the character 'A'.

4. Thus the code for character 'A' is 11. In similar fashion the codes for other characters can also be obtained.

You can see that codes of characters having higher frequencies are shorter than those having lower frequencies.

Generating Huffman Trees

Given a set of characters and their relative frequencies, how do we construct the ``best'' code? (In other words, which tree will encode messages with the fewest bits?) Huffman gave an algorithm for doing this and showed that the resulting code is indeed the best variable-length code for messages where the relative frequency of the symbols matches the frequencies with which the code was constructed.

To generate the Huffman Tree for a file

1. The file is scanned from beginning to the end and the number of times each character occurs is stored in an array.

2. The size of this array must be atleast 2n-1 where n is the number of characters used in the file. Generally we use an array of 511 elements. Since one byte can represents 256 different characters, we will need 2*256-1 =511 elements. Each element of this array represents a node of the tree.

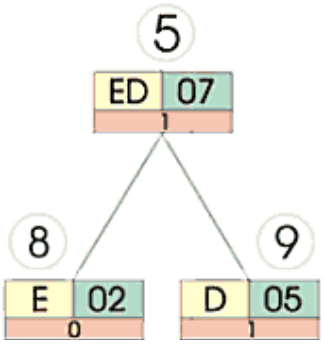
3. The first element of this array gives us the frequency of the first character. Similarly the second element gives the frequency of the second character.

4. After the file is scanned, the first 256 elements will contain the frequencies of each of the 256 characters. The remaining 255 elements will be empty.

5. The next step is to find two nodes having the least frequency.

6. Take one of the empty elements from the array and make it the parent of these two nodes.

7. *The frequency of the parent node is the sum of the frequencies of these two nodes.* **The left and right child nodes can be interchanged**, it does not matter if node 8 is the left or right child node of the parent node 5.



8. Now find the next two nodes having least frequencies, after removing the two child nodes and adding the parent node to the search list. Continue this process until only one node is left in the search list. This node is the root node of the tree.

Compressing Files

Once the Huffman tree is generated, the file is scanned again and each character in the file is replaced by its corresponding code from the tree.Once this is done, the character and the codes for each character along with the length of each code must be saved in the file in the form of a table. This table is needed during the decompression process.The frequency of the characters do not have to be saved because it is not needed for the decompression process. The size of this table depends on the file being compressed and usually ranges from 500 to 1200 bytes.

Decompressing Files

Before decompressing the file, we have to first regenerate the huffman tree from the table that was saved along with the compressed file.

To do this we use the following algorithm

1. We initialize an array having 511 elements to zero.
2. We take one element from the array and make it the root node.The root node is initially the current node.
3. We then read the i^{th} (initially the first) bit of the code for the j^{th}(initially the first) character in the table.
4. If the i^{th} bit of the code read is a '1', we take another element from the array and make it the right child node of the current node, if it is a '0' then the node is made the left child node of the current node. If the current node already has a left or right child node we skip this step.
5. If the bit currently read is the last bit of the code then this node holds the character represented by the code.
6. This node is then made the current node.
7. i is incremented
8. Steps 3 to 5 are repeated until all the bits of the code for the j^{th} character are read.
9. The current node is reset to the root node.
10. j is incremented and i reset to 0
11. steps 3 to 10 are repeated until the codes for all the characters are read.

Once this process is completed we will obtain the complete Huffman tree that was used to compress the file. This tree is used search for the character that was represented by the code in the compressed file.

Now the actual decompression process begins.

1. The current node is set to the root node.

2. A sequence of zeros and ones are read from the compressed file. For every '0' read we move to the left child node of the current node and for every '1' read we move to the right child node of the current node and set it as the new current node.
3. If the current node is a leaf node then we output the character that is represented by this node. The current node is reset to the root node.
4. Steps 2 to 3 are repeated until all the bytes in the file are read.

Once this process is complete the output file contains the decompressed data.

Conclusion

This compression ratio achieved by this algorithm is usually in the range of 20 to 40 % . Once it was believed that no other algorithm could achieve a better compression ratio.However recently compression algorithms (like the LZW compression algorithm) have been developed which produce much better compression ratios. So this compression algorithm is now mostly used in conjunction with other compression algorithms to achieve a better compression ratio.