**Huffman Coding (greedy technique) used for File Compression**

A large size of files is a big problem when your computer has limited memory. This problem is solved by compressing such large files to reduce their sizes.

Consider alphabets with frequencies

a=50 b=10 c=30 d=5 e=3 f=2

M=100-character (message consist of 100 alphabets) to encode this, we have to write in 0,1 form ASCII encoding technique (0-127) range to represent no of bits, require 7bits

127=1111111(in binary) to represent every alphabet am....f we need 7 bits to encode all these alphabets.......7\*100=700 bits require. But we have limited space and we want to reduce this size. To solve this problem, we will use Huffman tree algorithm.

**Building of HUFFMAN TREE**

* Made a leaf node for every alphabets.
* Every Leaf node contains the frequency of that character.
* Made a new internal node.
* The frequency of this new node is the sum of frequency of those two nodes. Which has smallest frequency.
* Select two minimum frequency f=2, e=3 then add result will compare with other alphabets.
* And build tree by merge values and compare with other alphabets.

**Code of Bit**

* Element on left side assign value=0
* Element on right side assign value=1 then
* Count value till leaf node how many bits require and then multiply no of bits to frequency
* Formula=frequency\*code of bit
* Add and divide by 100

**Pseudocode**

The steps you’ll take to do perform a Huffman encoding of a given text source file

into a destination compressed file are:

**1. Count frequencies:** Examine an input file’s contents and count the number of occurrences

of each character, and store them in a map using the huff map class you’ll write.

**2. Build encoding tree:** Build a binary tree with a particular structure, where each node

represents a character and its count of occurrences in the file. A priority queue is used

to help build the tree along the way.

**3. Build encoding map**: Traverse the binary tree to discover the binary encodings of each character.

4. **Encode data:** Re-examine the input file’s contents, and for each character, output the encoded binary version of that character to the destination file.

Huffman algorithm is based on greedy approach to get optimal result(code length)

**Algorithm Huffman(X)**

**Input:** String x of length n with d distinct characters

**Output:** coding tree for X

Compute the frequency f (c) of each character c of X.

(c1, f[c1] ) , (c2, f[c2] ) ,…, (cn, f[cn] )

Initialize a priority queue Q

For i=1 to n-1 Do

Create a single-node binary tree storing c

T = allocate new node.

Insert T into Q with key f (c).

While Size ( )> 1 do

*F1* Q.min()

*T1* Q.removeMin()

*F2* Q.min()

*T2* Q.removeMin()

Create a new binary tree T with left sub tree T1 and right sub tree T2

T.left=T1 T.right=T2 Insert T into Q with key f1 +f2

T.f = T1.f1 + T2.f2

return Tree Q.removeMin()

**Example:**

As an example, suppose we have a file named try.txt whose contents are: ad ad ead. In the original file, this text occupies 10 bytes (80 bits) of data, including spaces and a special “end-of-file” (EOF) byte.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Byte** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **character** | **‘a’** | **‘d’** | **‘ ’** | **‘a’** | **‘d’** | **‘ ’** | **‘e’** | **‘a’** | **‘d’** | **EOF** |
| **ASCII** | **97** | **100** | **32** | **97** | **100** | **32** | **101** | **97** | **100** | **256** |
| **Binary** | **01100001** | **01100100** | **00100000** | **01100001** | **01100100** | **00100000** | **01100101** | **01100001** | **01100100** | **N/A** |

**Count Frequencies:**

In Step 1 of Huffman’s algorithm, a count of each character is computed. This frequency table is represented as a map:

{' ':2, 'a':3, 'd':3, 'e':1, EOF:1}

**Build encoding tree:**

#(10)

0 / \1

#(4) #(6)

0/ \1 0/ \1

‘ ’(2) #(2) a(3) d(3)

0 / \1

e(1) EOF(1)

**Buils encoding map:**

{' ':"00", 'a':"10", 'd':"11", 'e':"010", EOF:"011"}

**Encoded data:**

Using the encoding map, we can encode the file’s text into a shorter binary representation. Using the preceding encoding map, the text "ad ad ead" would be encoded as:

**1011001011000101011011**

The following table details the char-to-binary mapping in more detail. The overall encoded contents of the file require 22 bits, or a little under 3 bytes, compared to the original file size of 10 bytes.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Char** | **‘a’** | **‘d’** | **‘ ’** | **‘a’** | **‘d’** | **‘ ’** | **‘e’** | **‘a’** | **‘d’** | **EOF** |
| **Binary** | **10** | **11** | **00** | **10** | **11** | **00** | **010** | **10** | **11** | **011** |

**Decoding a File:**

You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are given the same encoding tree above, and we are asked to decode a file containing the following bits:

**1011001011000101011011**

Using the Huffman tree, we walk from the root until we find characters, then output them and go back to the root.

* We read a 1 (right), then a 0 (left). We reach 'a' and output a. Back to the root **1011001011000101011011**
* We read a 1 (right), then a 1 (right). We reach 'd' and output d. Back to root. **1011001011000101011011**
* We read a 0 (left), then a 0 (left). We reach ' ' and output a space. **1011001011000101011011**
* We read a 1 (right), then a 0 (left). We reach 'a' and output a. Back to the root. **1011001011000101011011**
* We read a 1 (right), then a 1 (right). We reach 'd' and output d. Back to root. **1011001011000101011011**
* We read a 0 (left), then a 0 (left). We reach ' ' and output a space.  **1011001011000101011011**
* We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'e' and output e. **1011001011000101011011**
* We read a 1 (right), then a 0 (left). We reach 'a' and output a. Back to the root. **1011001011000101011011**
* We read a 1 (right), then a 1 (right). We reach 'd' and output d. Back to root. **1011001011000101011011**
* We read a 0, 1, 1. This is our EOF encoding pattern, so we stop. The overall decoded text is “ad ad ead. ”

Algorithm for Decoding tree is given below:

**Decode(root, inputString)**

output = ” ”

create pointer which point towards current node

current = root

FOR i=1 to size of inputString

if (inputString[i]=0)

current=current.left

else

current=current.right

if current.left==NULL and current.right==NULL

output=current.data

current=root

return output