HUFFMAN COMPRESSION

Project 2

Asheem Chhetri

Compression is a tool used to facilitate storing large data sets. There are two different sorts of goals one might hope to achieve with compression:

• Maximize ease of access, manipulation and processing

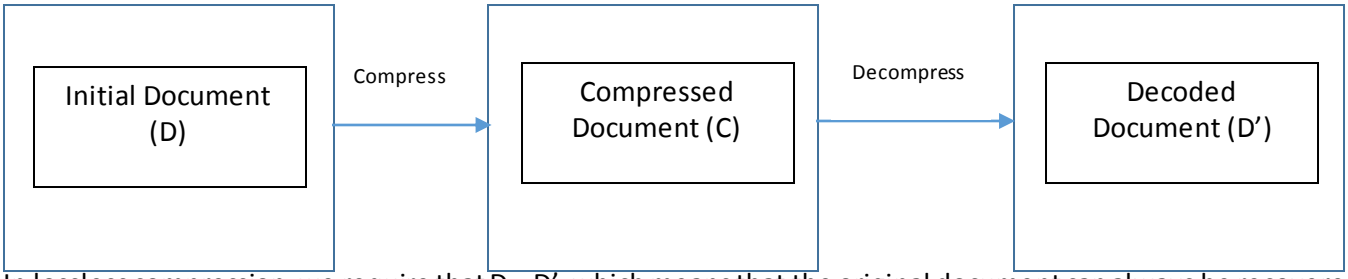
• Minimize size—especially important when storage or transmission is expensive.

In general, data cannot be compressed. For example, we cannot represent all m-bit strings using (m – 1)-bit strings without any loss, since there are 2m possible m-bit strings and only 2m – 1 possible (m – 1)-bit strings unless:

• If only a relatively small number of the possible m-bit strings appear, compression is possible.

• If the same "long" substring appears repeatedly, we could represent it by a "short" string.

Lossless Compression:



In lossless compression, we require that D = D', which means that the original document can always be recovered exactly from the compressed document. Like: Huffman compression

In the nutshell, Huffman compression is done by counting the frequencies of the different ASCII letters in t he file. The program assigns smaller bit values to represent often repeated letters and bigger bit values to less often recurring letters. This way we are able to minimize the number of bits in use.

Eg:

Consider a six-letter alphabet which has following frequencies:

Character	a	b	c	d	e	f
Frequency	45%	13%	12%	16%	9%	5%
Codeword	0	101	100	111	1101	1100
	(1 bit)	(3 bits)			(4 bits)	

Therefore the average number of bits used to encode 100 character is: (45)1+(13)3+(12)3+(16)3+(9)4+(5)4 = 224

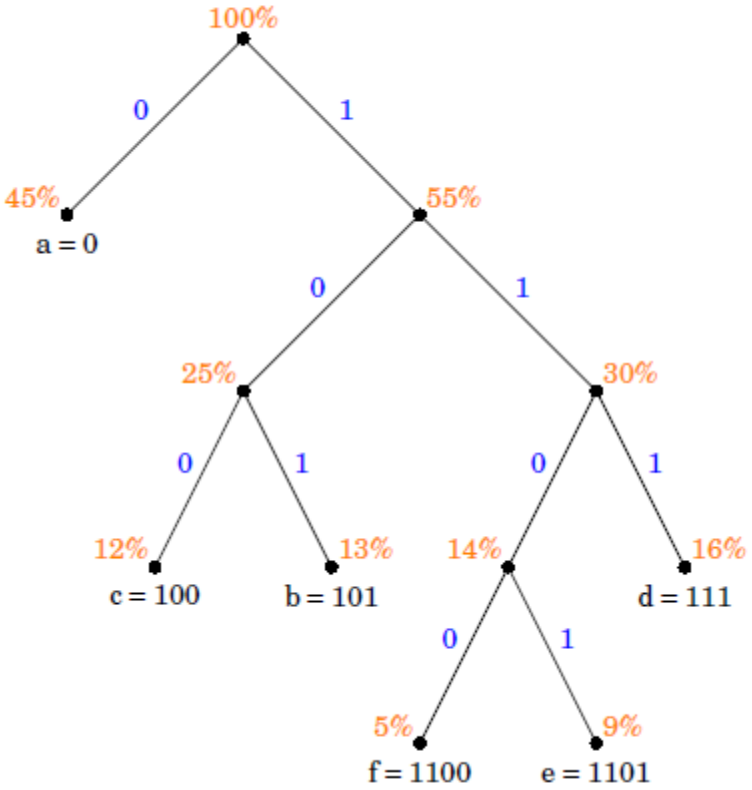
- Given an alphabet $\{a_i\}$ with frequencies: $\{f(a_i)\}$
- We need to find a set of binary code words $C = \{c(a_1),\ldots,ca_n\}$, such that the average number of bits used to represent the data is minimized:

$$B(C) = \sum_{i=1}^{n} f(a_i)\,|c(a_i)|\,.$$

- We can also represent our code as a tree T with leaf nodes $a_1,\ldots,a_n$, then we can minimize it in the following manner:

$$B(T) = \sum_{i=1}^{n} f(a_i)\,d(a_i),$$

- Thus using following algorithm we can create an optimal prefix tree for a given set of characters $C = \{a_i\}$



Algorithm: HUFFMAN-TREE(C)

```
1  n ← |C|
2  Q ← C ▷ a min–priority queue keyed by frequency
3  for i ← 1 to n − 1 do
4       Allocate new node z
5       z.left ← x ← EXTRACT-MIN(Q)
6       z.right ← y ← EXTRACT-MIN(Q)
7       z.freq ← x.freq + y.freq
8       Q.INSERT(z)
9  ▷ Return the root of the tree
10 return EXTRACT-MIN(Q)
```

Compression

- Input file received
- We now create a list of frequencies based on the data received
- A tree is being created from all these nodes
- The two lowest frequency tree nodes are joined together, and their frequencies are added to make the parent node.
- This process repeats itself, until the tree is completed till the root.
- With this tree, we create codebook, where each letter is assigned a number of bits(i.e. 10, 01, 110 etc)
- We then write the header to the file, and contains the binary tree that was used to create the codebook and a new line is inserted.
- We then traverse the file and start storing the letters in the file while referring to the codebook, created earlier.
- File is saved using the extension: ".huff"

A Huffman tree has been used in huff.c program, to generate the header information by traversing in post order. Further it was the used in encoding the input text file by finding the encoded value for each character by traversing from root and then writing the encoded bits into the text file.

Decompression

- Input file is received, which will have the extension ".huff"
- We check the header info, to make sure we are decoding the same file.
- The binary tree is then constructed using the header.
- Now the file read while referring to the binary tree and find each values corresponding letter, till we reach EOF.
- Finally the decoded file is saved with the extension: ".unhuff"

A Huffman tree has been used in unhuff.c, to read the header bytes first and then the code book, read the basic information of the Huffman table, then traverse the tree in binary one digit at a time (if we get a '0' , go to the left child of the node, or go to the right child). When you reach a leaf that with no child, it is the character. Repeat the previous step until the end of the file.

Assumption

- It was assumed that the input file to huff.c will be pure ASCII text file.

Sources Used:

1. MIT course notes on Huffman Compression from class: MIT 6.046 lecture 19
2. I used professor LU's book as a reference, since he was my professor for ECE 264 and I received his book through him as a part of course material. Since he had explained Huffman Coding in his book, it was easy for me to follow his work even though not all part of the work was used but main emphasis was used to display codebook and frequency table to see how the file was being parsed, to determine the correctness of rest of the work.

Compression Ratio:

Based on the text files provided with the assignment:

File name	Original size(bytes)	Size after encoding(bytes)	Compression Ratio	Size after decoding(bytes)	Decoded vs original file match
text0	4	9	0.44	4	TRUE
text1	8	17	0.47	8	TRUE
text2	155	123	1.26	155	TRUE
text3	3,150,000	2,312,212	1.36	3,150,000	TRUE
text4	6,300,000	4,624,146	1.36	6,300,000	TRUE
text5	9,450,000	6,936,434	1.36	9,450,000	TRUE

Final Notes:

- No external flags were used
- ./huff to encode the file
- .unhuff to decode the file
- Accepts only one argument