Optimal Coding

CS 251 Design and Analysis of Algorithms

Anis Ur Rahman
Department of Computing
NUST-SEECS
Islamabad

May 12, 2020

Optimal Coding

First some definitions.

- Alphabet = finite set of symbols, for instance
    - English alphabet = $a, b, c, \cdots, z,$
    - Hex values = $0, 1, \cdots, 9, A, B, C, D, E, F.$
- String = sequence of symbols from some alphabet, for instance "This is a string".

Fixed Length Encoding

We know that computers store things as $0'$s and $1'$s.

- Consider a string that is to be encoded as sequence of bits.
- The encoding must be invertible (one-to-one), use as few bits as possible.

Fixed Length Encoding

A simple approach is to choose encoding for characters,

- induce encoding of strings by concatenating codes for each character.

For example,

Letter	Encoding
A	00
B	01
C	10
D	11

The resulting encoding for a string:
ABACBDAAADBAC $\Longrightarrow$ 00010010011100000011010010

Fixed Length Encoding

A simple approach is to choose encoding for characters,

- induce encoding of strings by concatenating codes for each character.

You might have noticed that the approach is wasteful,

- if not exactly 2^k characters, and
- if some sequences never used.

Fixed Length Encoding

A simple approach is to choose encoding for characters,

- induce encoding of strings by concatenating codes for each character.

You might have noticed that the approach is wasteful,

- if not exactly 2^k characters, and
- if some sequences never used.

Consider 11 is never used. Moreover, what if one character occurs very often?

- for instance, AAAAAABAAACAABAADAAAAAAACAAAB.
- If almost all letters are A's, then an encoding that uses fewer bits to represent A and more to represent everything else would save on space.

Variable Length Encoding

Variable Length Encoding encodes characters as bits where different letters may use a different number of bits.

Consider we encode A's using fewer number of bits.

Letter	Encoding
A	0
B	01
C	10
D	11

Recall the encoding on strings must be one-to-one. Not the case with the encoding above, the encoding of A is a prefix of the encoding of B, resulting encoding 010 for AC and BA.

Example

Consider the coding schemes.

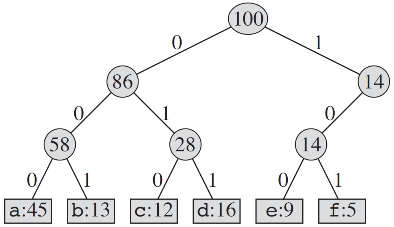
	a	b	c	d	e	f
frequency	45	13	12	16	9	5
fixed	000	001	010	011	100	101
variable	0	101	100	111	1101	1100

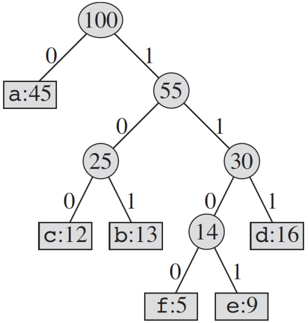
Consider a string S of length $n = 100000$ then

- 3-bit fixed-length codewords will result in total 300000 bits.
- variable-length codewords will result in total 224000 bits ($\sum_i f_i l_i = 45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4$), which is $\approx 25\%$ saving in the total number of bits used to encode S.

Example

Trees corresponding to the coding schemes.





- Each leaf is labeled with a character and its frequency of occurrence.
- Each internal node is labeled with the sum of the frequencies of the leaves in its subtree.

Prefix-Free Encoding

The variable-length codes assigned to input characters are Prefix Codes,

- means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Prefix-Free Encoding

The variable-length codes assigned to input characters are Prefix Codes,

- means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Counter example.

Let there be four characters A, B, C and D, and their corresponding variable length codes be 00, 01, 0 and 1.

- This coding leads to ambiguity because code assigned to C is prefix of codes assigned to A and B.
- If the compressed bit stream is 0001, the de-compressed output may be "CCCD" or "CCB" or "ACD" or "AB".

Prefix-Free Encoding

The variable-length codes assigned to input characters are Prefix Codes,

- means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character.
- This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Letter	Encoding
A	0
B	10
C	110
D	111

Theorem

Claim.

Any prefix-free encoding of an alphabet induces a one-to-one encoding of strings over that alphabet.

Proof.

- Suppose toward contradiction that S and T are two different strings that map to the same sequence of bits.
- Assume that S and T differ on the first character.
- Let c be the first character of S, d the first character of T.
- Let $E(c)$ and $E(d)$ be the encodings of c and d.
- Assume $|E(c)| \geq |E(d)|$; the length of encoding bits.
- Since all bits in encodings of S and T are the same, the first $|E(d)|$ bits of $|E(c)|$ are equal to $E(d)$.
- Hence $E(d)$ is a prefix of $E(c)$, but c was assumed different from d, our encoding is not prefix-free.

Tree View of Prefix-Free Encoding

- Every internal node represents a partial codeword
- Every node has two children, one for appending 0 to the partial codeword, one for appending 1.
- Leaves correspond to actual codewords
- Root is empty

Traversing the Prefix-Free Encoding Tree

- **To encode.** Find path from root to character, concatenate edge labels.
- **To decode.** Starting from the root, follow edge labeled b_1, then edge labeled b_2, ... until we find a leaf. Output that character, and start over from the root.

Lemma.

The tree for any optimal prefix code must be "full", meaning that every internal node has exactly two children.

Proof. If some internal node had only one child then we could simply get rid of this node and replace it with its unique child. This would decrease the total cost of the encoding.

Optimal Prefix-free Encoding

Let n be the length of the string, and C be the cost of the encoding, defined as (length of encoding)/n—the average length of encoding of characters weighted by frequency

$$C = \sum_i f_i l_i$$

f_i: (number of instances of i)/n

l_i: length of the encoding of character i

l_i is also the depth of character i in the encoding tree, and depth of leafs only decreases.

Properties of Optimal Prefix Codes

- **Lemma 1.** Let x, y be symbols with $f_x > f_y$. Then in an optimal prefix code, $|E(x)| \leq |E(y)|$. If x occurs more often than y, its codeword must be no longer than the codeword for y in an optimal prefix code
- **Lemma 2.** If w is a longest codeword in an optimal prefix code, there must be another codeword with equal length
- **Lemma 3.** Let x, y be symbols with the smallest frequencies. Then there exists an optimal prefix code in which they differ only in the final bit
- **Theorem.** The prefix code output by the Huffman algorithm is optimal. Can be shown by induction. See book (page 433-435) for proof.

Our goal is to minimize this cost, or the total number of bits used.

Advantages of Optimal Prefix Codes

1. simplifies encoding
2. unambiguous
3. represented by a full binary tree

Huffman Coding

Huffman developed a nice greedy algorithm for solving this problem cost (optimum) prefix code.

Basic Idea

The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

- The most frequent character gets the smallest code and the least frequent character gets the largest code.

Huffman Coding. Pseudocode

Assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute c.freq giving its frequency.
The algorithm has two major parts.

1. build a Huffman Tree T corresponding to the optimal code in a bottom-up
2. traverse the Huffman Tree and assign codes to characters.

Algorithm

The algorithm to build the tree uses a min-priority queue Q,

- keyed on the frequency attribute, to identify the two least-frequent objects to merge together.
- When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

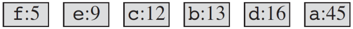
```
Build-Huffman-Tree(C)
  n = |C|
  Q = ENQUEUE(C)
  for i = 1 to n - 1
    allocate a new node z
    z.left  = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q,z)
  return EXTRACT-MIN(Q)   // return the root of the tree
```

Notation

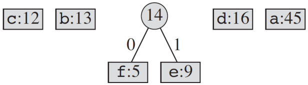
The steps of Huffman's algorithm for the frequencies given in the following example.

- Each part shows the contents of the queue sorted into increasing order by frequency.
- At each step, the two trees with lowest frequencies are merged.
- Leaves are shown as rectangles containing a character and its frequency.
- Internal nodes are shown as circles containing the sum of the frequencies of their children.
- An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child.
- The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.

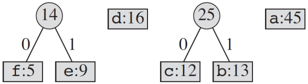
(a) initial set of $n = 6$ nodes, one for each letter. (b)–(e) intermediate stages. (f) final tree.



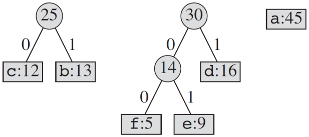
(a)



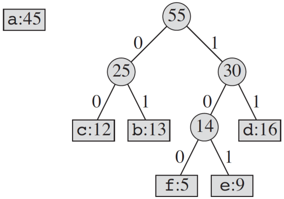
(b)



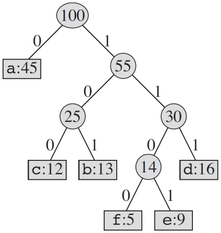
(c)



(d)



(e)



(f)

Running time

Assume that Q is implemented as a binary min-heap.

- For a set C of n characters, we can initialize Q in line 3 in $O(n)$ time using the BUILD-MIN-HEAP procedure.
- The for loop in lines 4–9 executes exactly $n - 1$ times, and since each heap operation requires time $O(lgn)$, the loop contributes $O(nlgn)$ to the running time.

Thus, the total running time of Build-Huffman-Tree on a set of n characters is $O(nlgn)$.

We can reduce the running time to $O(nlglgn)$ by replacing the binary min-heap with a van Emde Boas tree.

Efficient Huffman Coding for Sorted Input

If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in $O(n)$ time.
The $O(n)$ algorithm for sorted input is,

1. Create two empty queues.
2. Create a leaf node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty.
3. Dequeue two nodes with the minimum frequency by examining the front of both queues. Repeat following steps two times
   - If second queue is empty, dequeue from first queue.
   - If first queue is empty, dequeue from second queue.
   - Else, compare the front of two queues and dequeue the minimum.
4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first Dequeued node as its left child and the second Dequeued node as right child. Enqueue this node to second queue.
5. Repeat steps #3 and #4 until there is more than one node in the