Greedy Algorithms : Huffman Codes

Dr. G P Gupta

Data Compression

- Suppose we have 1000000000 (1G) character data file that we wish to include in an email.
- Suppose file only contains 26 letters {a,…,z}.
- Suppose each letter a in {a,…,z} occurs with frequency f_a.
- Suppose we encode each letter by a binary code
- If we use a fixed length code, we need 5 bits for each character
- The resulting message length is $5(f_a + f_b + \cdots f_z)$
- **Can we do better?**

Huffman Codes

- Widely used and very effective for data compression
- Savings of 20% - 90% typical space
  – *depending on the characteristics of the data*
- Huffman's greedy algorithm uses a table of frequencies of character occurrences to build up an optimal way of representing each character as a binary string.

Binary String Representation - Example

- Consider a data file with:
  – 100K characters
  – Each character is one of {a, b, c, d, e, f}
- Frequency of each character in the file:

	a	b	c	d	e	f
frequency	45K	13K	12K	16K	9K	5K

- *Binary character code*: Each character is represented by a unique binary string.
- ***Intuition***:
  Frequent characters ⇔ shorter codewords
  Infrequent characters ⇔ longer codewords

Binary String Representation - Example

	a	b	c	d	e	f
frequency	45K	13K	12K	16K	9K	5K
fixed-length	000	001	010	011	100	101
variable-length(1)	0	101	100	111	1101	1100
variable-length(2)	0	10	110	1110	11110	11111

How many total bits needed for fixed-length codewords?
100K * 3 = 300K bits
How many total bits needed for variable-length(1) codewords?
45K*1 + 13K*3 + 12K*3 + 16K*3 + 9K*4 + 5K*4 = 224K
How many total bits needed for variable-length(2) codewords?
45K*1 + 13K*2 + 12K*3 + 16K*4 + 9K*5 + 5K*5 = 241K

Huffman Codes

- Most character code systems (ASCII, unicode) use fixed length encoding
- If frequency data is available and there is a wide variety of frequencies, **variable length encoding can save** 20% to 90% space
- Which characters should we assign shorter codes; which characters will have longer codes?

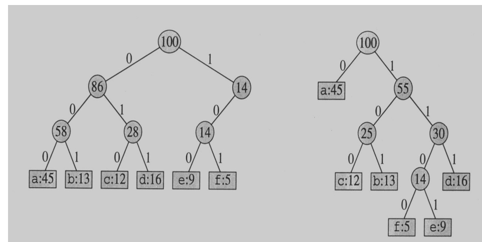
How to decode?

- At first it is not obvious how decoding will happen, but this is possible if we use **prefix codes**.
- codes in which no codeword is also a prefix of some other codeword. Such codes are **called prefix codes.**

- No encoding of a character can be the prefix of the longer encoding of another character, for example, we could not encode *t* as 01 and *x* as 01101 since 01 is a prefix of 01101

Prefix Codes

- By using a binary tree representation we will generate prefix codes provided all letters are leaves



Prefix codes

- A message can be decoded uniquely.

- Following the tree until it reaches to a leaf, and then repeat!

- Draw a few more tree and produce the codes!!!

Some Properties

- Prefix codes allow easy decoding
  - Given a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100
  - Decode 001011101 going left to right, 0|01011101, a|0|1011101, a|a|101|1101, a|a|b|1101, a|a|b|e

- An optimal code must be a full binary tree (a tree where every internal node has two children)
- For *C* leaves there are *C-1* internal nodes
- The number of bits to encode a file is

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

where ***f(c)* is the freq of *c***, $d_T(c)$ is the **tree depth of *c***, which corresponds to the code length of *c*

Optimal Prefix Coding Problem

- Input: Given a set of *n* letters ($c_1, \ldots, c_n$) with frequencies ($f_1, \ldots, f_n$).

- Construct a full binary tree *T* to define a prefix code that minimizes the average code length

$$\text{Average}(T) = \sum_{i=1}^{n} f_i \bullet \text{length}_T(c_i)$$

Greedy Algorithms

- Many optimization problems can be solved using a greedy approach
  - The basic principle is that local optimal decisions may may be used to build an optimal solution
  - But the greedy approach may not always lead to an optimal solution overall for all problems
  - The key is knowing which problems will work with this approach and which will not
- We will study
  - **The problem of generating Huffman codes**

Greedy algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - My everyday examples:
    - Driving in Los Angeles, NY, or Boston for that matter
    - Playing cards
    - Invest on stocks
    - Choose a university
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- Greedy algorithms tend to be easier to code

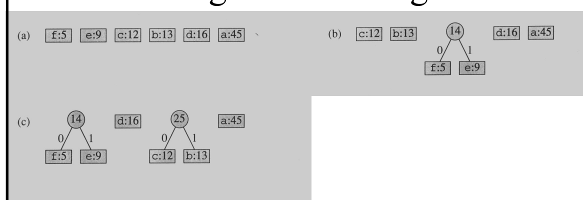
David Huffman's idea

- A Term paper at MIT

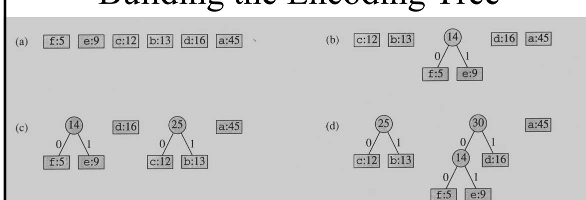
- Build the tree (code) bottom-up in a greedy fashion

Building the Encoding Tree

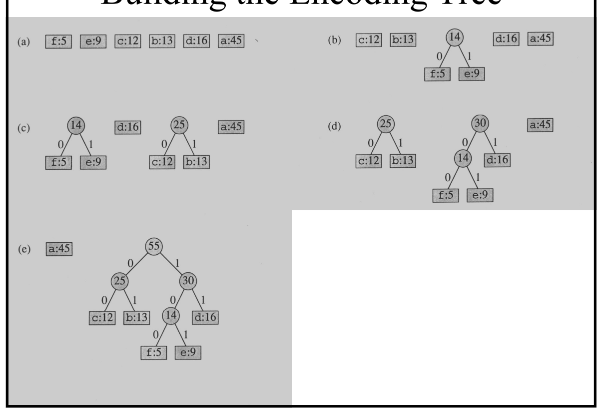
Building the Encoding Tree



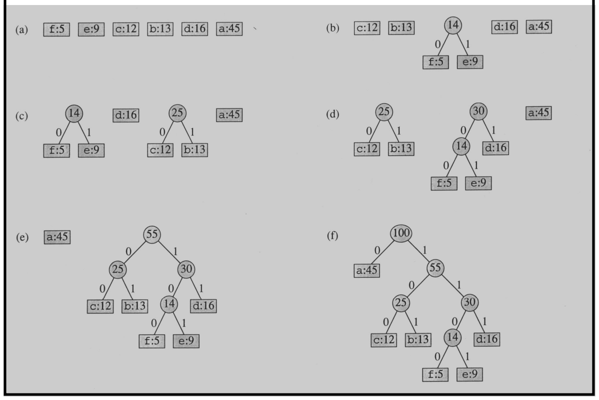
Building the Encoding Tree



Building the Encoding Tree



Building the Encoding Tree



The Algorithm

```
HUFFMAN(C)
n ← |C|
Q ← C
for i ← 1 to n − 1
    do allocate a new node z
       left[z] ← x ← EXTRACT-MIN(Q)
       right[z] ← y ← EXTRACT-MIN(Q)
       f[z] ← f[x] + f[y]
       INSERT(Q, z)
return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

- An appropriate data structure is a ***binary min-heap***
- **Rebuilding the heap is** ***lg n*** and ***n-1*** extractions are made, so the **complexity is** O(*n lg n*)
- The encoding is NOT unique, other encoding may work just as well, but none will work better