CS 445- Algorithms

Another example of greedy algorithm: Huffman codes

Credit: Artur Czumaj

Compression

- Take a text (or an object)
- Encode it so that
  - Less space is used, or energy to transmit it.
  - No information is lost – we can reconstruct the original text
  - Often: be able to quickly reconstruct original text

Lossless compression

Huffman Coding

- Coding: each character → unique binary string
- Example (not so good for compression):
  ASCII codes-each character uses 8 bits
  - Doesn't give good compression rate
- Different characters will have different lengths
  - More frequent (common) characters will have shorter coding
    - letter "e" is most frequent ~ 12%
  - rare characters will have longer coding
    - letter "z" is very rare

Huffman coding

- Easy to assign bit-strings to letters
- How to ensure (unique) reconstruction?

A → 01 How to decode 010101

B → 0101 AB or BA or perhaps AAA?

Definition: Prefix codes:

no codeword is a prefix of another codeword

Huffman coding / prefix codes

Prefix codes:

no codeword is a prefix of another codeword

char	code
A	1
B	01
C	001
D	0001

- Easy encoding:
  - Construcs the strings *s* of bits by concatenating codewords of chars
- Easy decoding:
  - given *s*, we find the first few bits (prefix) that forms a char – there can be only one such prefix.
  - Remove this prefix from *s* and repeat.

D A D A B C A

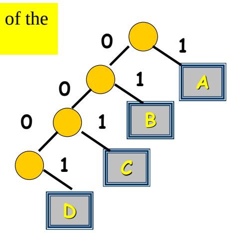
0001 1 0001 1 01 001 1

Huffman coding

- Codewords are presented by a binary tree
- Each leaf stores, and represents a character
- Node with two children – left ~ 0; right ~ 1
- codeword = path from the root to the leaf storing given characters

The code represented by the lefs of the tree is a prefix code (why?)

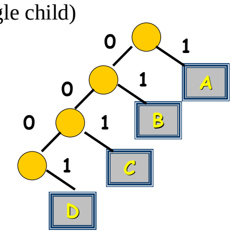
char	code
A	1
B	01
C	001
D	0001



Huffman coding

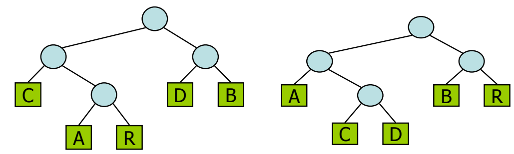
- Codewords are presented by binary trees
- We can always aim at getting full binary trees
  - (no node with a single child)

char	code
A	1
B	01
C	001
D	000



Huffman codes and full trees

- Given a text string *X*, find a prefix code for the characters of *X* giving smallest encoding for *X*
  - Frequent characters should have short codewords
  - Rare characters should have long codewords
- Example
  - *X* = ABRACADABRA ("R" is rate, "A" is frequent)
  - *T*1 encodes *X* into 29 bits
  - *T*2 encodes *X* into 24 bits



Huffman codes and full trees

- Given a file *X*, find a prefix code for the characters of *X* giving smallest encoding for *X.*
  - Frequent characters should have short codewords
  - Rare characters should have long codewords
- **More practical scenario**:
  - Given frequencies of possible characters in a language, find a prefix code that gives smallest encoding of a string from the language

Huffman codes-cont

- Σ - alphabet. X – input file to encode.
- $f(x)$ = how many times x appears X.
- Let $w(x)$ denote the binary code of a char $x \in \Sigma$.
- The size of the encoded file is therefor
  $\sum_{x \in \Sigma} f(x) \; w(x)$
- The **depth** of a leaf $w(x)$ of the encoding tree is the distance from the root to the leaf = $|w(x)|$
- Given a coding tree T, the cost of of the tree is
  $cost(T) = \sum_{x \in \Sigma} f(x)$ depth$(w(x))$

Problem: Find a tree T of minimum cost.

Greedy algorithm for generating opt tree

Start: Each character is a tree by itself (so we have a forest of $|\Sigma|$ trees. Store them in a heap Q.

Repeat until one tree is left:

Find two nodes u,v with the lowest frequencies.

Create a new internal node, w with u,v nodes as its children (either node can be either child) and the sum of their frequencies as the new frequency

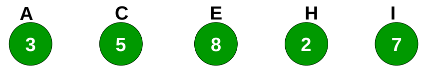
```
for i=1 to n - 1 {
 ALLOCATE-NODE(w)
 left[w]=  u =EXT_MIN(Q)
 right[w]= v =EXT-MIN(Q)
 f[w] =f[u] + f[v]
 INSERT(Q, w)
}
return EXTRACT-MIN(Q)
```

Credit for next several slides:

Nelson Padua-Perez and William Pugh

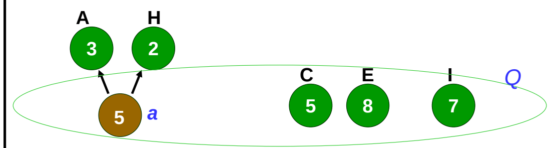
Huffman Tree Construction 1

The number indicate the frequency



•The two least-frequent nodes are A,H
•The algorithm replaces them with one new node ***a***.
•Its frequency is the sum of frequencies of these two nodes

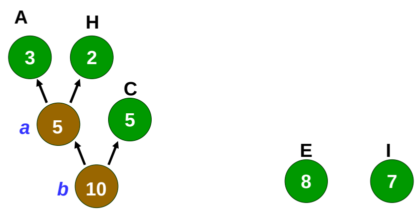
Huffman Tree Construction 2



The frequency of a is the length of the encoded binary file taken by A and H

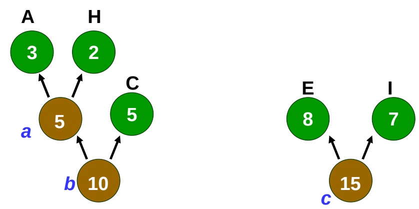
•The two least-frequent nodes are A,H
•The algorithm replaces them with one new node ***a***.
•Its frequency is the sum of frequencies of these two nodes

Huffman Tree Construction 3

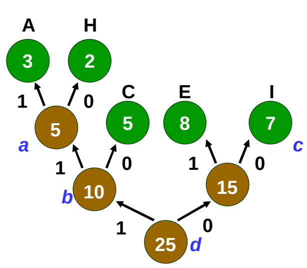


The two least frequent nodes were ***a*** and C, and they were replaced by a node ***b*** whose frequency is the sum of their frequencies – 10.

Huffman Tree Construction 4



Huffman Tree Construction 5



E	=	01
I	=	00
C	=	10
A	=	111
H	=	110

Huffman codes

- Good implementations:
  - $O(n \log n)$ time, where $n = |\Sigma|$
- Using priority queues (aka binary heaps):
  - Initially, store all characters in a priority queue wrt the frequencies (as the keys)
  - Removal of two nodes with lowest freqs: DELETEMIN
  - Inserting of a new node: INSERT
  - $O(\log n)$ operations DELETEMIN / INSERT ➔ $O(n \log n)$ time

Huffman codes

- Correctness:

Huffman codes-correctness

Assume by induction that the algorithm works correctly for all alphabets with less than *n* characters.

- Optimum tree (recall: not unique):
  - Is a full binary tree (all internal nodes have 2 children)
  - There is always an optimal tree in which two nodes with minimum frequencies are siblings
    - (if this is not the case in an optimal tree, we can always replace one with the sibling of the other, getting an equally-cheap tree)
  - If we remove any two sibling leaves (but leave their parent) then we're left with an optimum tree for the same alphabet but with a new char that replaces the two leaves – freq of this char is freq of that node
- This is exactly what Huffman algorithm produces