Data Structures – LECTURE 10

Huffman coding

- Motivation
- Uniquely decipherable codes
- Prefix codes
- Huffman code construction
- Extensions and applications

Chapter 16.3 pp 385—392 in textbook

Motivation

- Suppose we want to store and transmit very large files (messages) consisting of strings (words) constructed over an alphabet of characters (letters).
- Representing each character with a fixed-length code will not result in the shortest possible file!
- Example: 8-bit ASCII code for characters
  - some characters are much more frequent than others
  - using shorter codes for frequent characters and longer ones for infrequent ones will result in a shorter file

Example

Σ	**a**	**b**	**c**	**d**	**e**	**f**
Frequency (%)	45	13	12	16	9	5
Fixed-length	000	001	010	011	100	101
Variable-length	0	101	100	111	1101	1100

Message: abadef → 000001000011100101
0101011111011100

A file of 100,000 characters takes:

- 3×100,000 = 300,000 bits with fixed-length code
- (.45×1 + .13×3 + .12×3+ .16×3 + .09×4 + .05×4) ×100,000 = 224,000 bits on average with variable-length code (25% less)

Coding: problem definition

- Represent the characters from an input alphabet **Σ** using a variable-length code alphabet *C*, taking into account the occurrence frequency of the characters.
- Desired properties:
  - The code must be *uniquely decipherable*: every message can be decoded in only one way.
  - The code must be *optimal* with respect to the input probability distribution.
  - The code must be *efficiently decipherable* → prefix code: no string is a prefix of another.

Uniquely decipherable codes (1)

- Definition: The code alphabet $C = \{c_1, c_2, \ldots, c_n\}$ over the original alphabet Σ is *uniquely decipherable* iff every message constructed from code-words of *C* can be broken down into code-words of *C* in only one way.
- Question: how can we test if *C* is uniquely decipherable?
- Lemma: a code *C* is uniquely decipherable iff no *tail* is a code-word.

Terminology

- Let *w*, *p*, and *s* be words over the alphabet *C*. For *w* = *ps*, *p* is the prefix and *s* is the suffix of *w*.
- Let *t* be a non-empty word. *t* is called a tail iff there exist two messages $c_1c_2\ldots c_m$ and $c'_1c'_2\ldots c'_n$ such that:
  - c_i and c'_j are code-words and $c_1 \neq c'_1$ ($1 \leq i \leq n$, $1 \leq j \leq m$)
  - *t* is a suffix of c'_n
  - $c_1c_2\ldots c_m t = c'_1c'_2\ldots c'_n$
- The length of a word *w* is $l(w)$. *w* is non-empty when $l(w) > 0$. *l* is the maximum length of a code-word in *C*.

Uniquely decipherable codes (2)

Proof: a code C is uniquely decipherable (UD) iff no tail is a code-word.

- If a code-word c is a tail then by definition there exist two messages $c_1c_2\ldots c_m$ and $c'_1c'_2\ldots c'_n$ which satisfy

  $$c_1c_2\ldots c_m c = c'_1c'_2\ldots c'_n \text{ while } c_1 \neq c'_1$$

  Thus there are two ways to interpret the message.
- If C is not UD, there exist messages which can be interpreted in more than one way. Let μ be the shortest such an ambiguous message. Then $\mu = c_1c_2\ldots c_k = c'_1c'_2\ldots c'_n$ that is, all c_i's and c_j's are code-words and $c_1 \neq c'_1$. Thus, without loss of generality, c_k is a suffix of c'_n → c_k is a tail.

Test for unique decipherability

1. For every two code-words, c_i and c_j ($i \neq j$) do:
   – If $c_i = c_j$ then halt: C is not UD.
   – If for some word s either $c_is = c_j$ or $c_js = c_i$ then put s in the set of tails T
2. For every tail t in T and every code-word c_i in C do:
   – If $t = c_j$ then halt: C is not UD.
   – If for some word s either $ts = c_j$ or $c_js = t$ then put s in the set of tails T.
3. Halt: C is UD.

Time complexity: $O(n^2l^2)$

Example 1

- $C = \{00,10,11,100,110\}$

  1. Tails: 10.0 = 100 → $t = 0$
     11.0 = 110 → $t = 0$
  2. Tails 0.0 = 00 → $t = 0$ C is *UD*

Example 2

- $C = \{1,00,101,010\}$

  1. Tails: 1.01 = 101 → $t = 01$
  2. Tails 01.0 = 010 → $t = 0$
     0.10 = 010 → $t = 10$
     10.1 = 101 → $t = 1$

  1 is a code-word! C is not *UD*

Counter-example: 10100101 has two meanings:

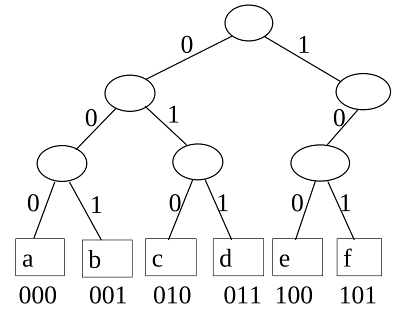
1.010.010.1

101.00.101

Prefix codes

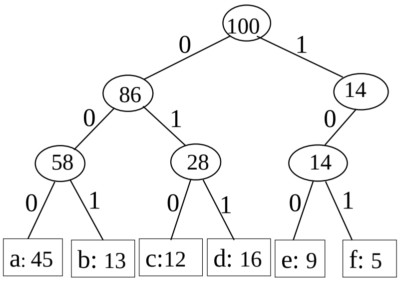
- We consider only prefix codes: no code-word is a prefix of another code-word. Prefix codes are uniquely decipherable by definition.
- A binary prefix code can be represented as a binary tree:
  – leaves are a code-words and their frequency (%)
  – internal nodes are binary decision points: "0" means go to the left, "1" means go to the right of a character. They include the sum of frequencies of their children.
  – The path from the root to the code-word is the binary representation of the code-word.

Example: fixed-length prefix code (1)



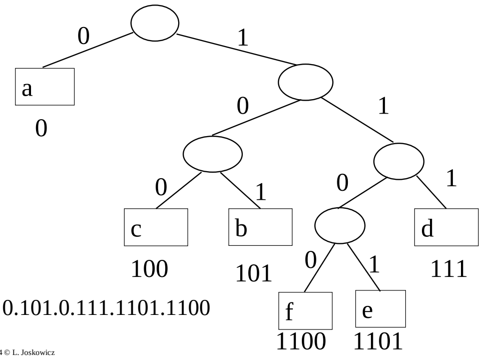
Message: 000.001.000.011.100.101 → abadef

Example: fixed-length prefix code (2)



Frequency (%)

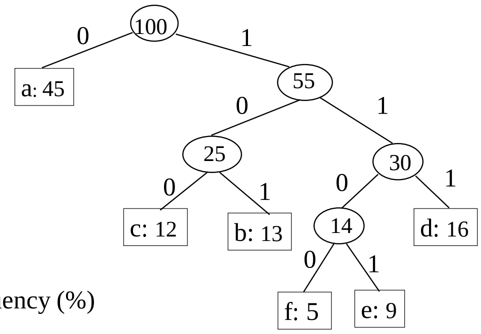
Example: variable-length prefix code (1)



Message: 0.101.0.111.1101.1100

→ abadef

Example: variable-length prefix code (2)



Frequency (%)

Optimal coding (1)

- An optimal code is represented as a full binary tree
- For a code alphabet $C = \{c_1, c_2, \ldots, c_n\}$ with $|C|$ code-words, all with positive frequencies $f(c_i) > 0$, the tree for an optimal prefix code has exactly $|C|$ leaves and $|C| - 1$ internal nodes.
- Definition: The cost of a prefix tree is defined as number of bits $B(T)$ required to encode all code-words

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

where $d_T(c)$ is the depth in T (length) of code-word c.

Optimal coding (2)

- Example
  - Fixed-length code:

    $(.45\times3 + .13\times3 + .12\times3 + .16\times3 + .09\times3 + .05\times3) = 3$
  - Variable-length code:

    $(.45\times1 + .13\times3 + .12\times3 + .16\times3 + .09\times4 + .05\times4) = 2.24$
- Optimal code: the code with the lowest cost:

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

- Theorem: Optimal coding is achievable with a prefix code.

Huffman code: decoding

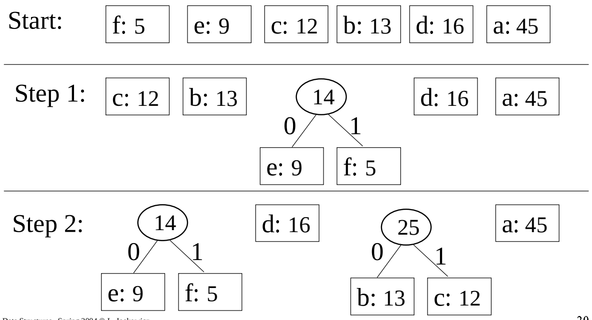
- Huffman invented in 1952 a greedy algorithm for constructing an optimal prefix code, called a Huffman code.
- Decoding:
  1. Start at the root of the coding tree T, read input bits.
  2. After reading "0" go left
  3. After reading "1" go right
  4. If a leaf node has been reached, output the character stored in the leaf, and return to the root of the tree.

  Complexity: $O(n)$, where n is the message length.

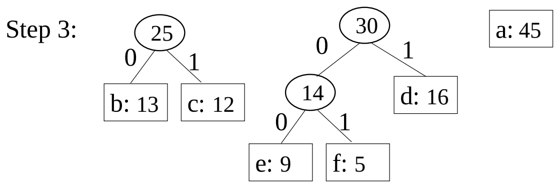
Huffman code: construction

- Idea: build the tree bottom-up, starting with the code-words as leafs of the tree and creating intermediate nodes by merging the two least-frequent objects, up to the root.
- To efficiently find the two least-frequent objects, use a minimum priority queue.
- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the merged objects.

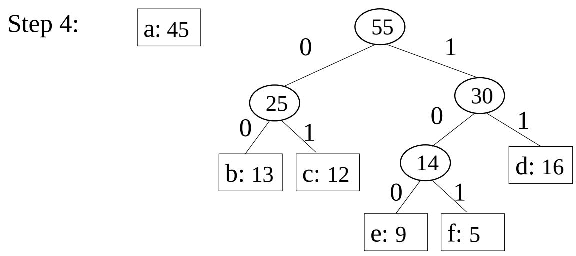
Example: Huffman code construction (1)



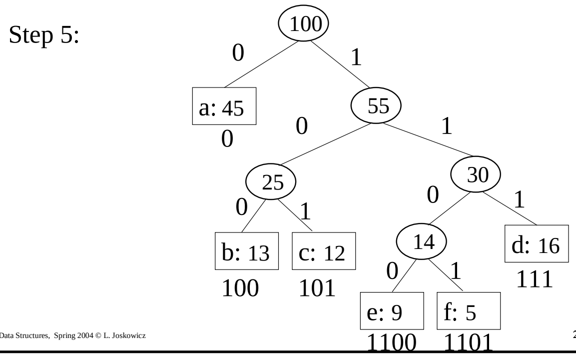
Example: Huffman code construction (2)



Example: Huffman code construction (3)



Example: Huffman code construction (3)



Huffman code construction algorithm

Huffman(C)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ **to** $n - 1$
  do allocate a new node z
    $left[z] \leftarrow x \leftarrow$ Extract-Min(Q)
    $right[z] \leftarrow y \leftarrow$ Extract-Min(Q)
    $f(z) \leftarrow f(x) + f(y)$
    Insert(Q, z)
return Extract-Min(Q)

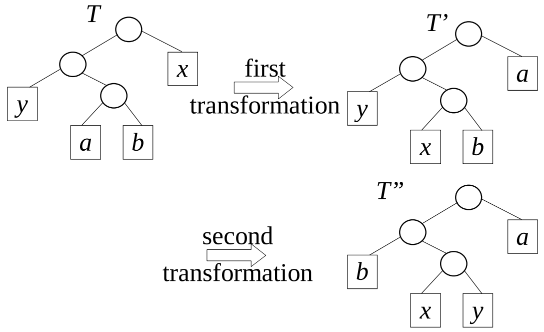
Complexity: $O(n \lg n)$

Optimality proof (1)

Lemma 1: Let C be a code alphabet and x, y two code-words in C with the lowest frequencies. Then there exists an optimal prefix code tree in which x and y are sibling leaves.

Proof: take a tree T of an arbitrary optimal prefix code where x and y are not siblings and modify it so that x and y become siblings of maximum depth and the tree remains optimal. This can be done with two transformations.

Optimality proof (2)



Optimality proof (3)

- Let a and b two code-words that are sibling leaves at maximum depth in T. Assume that $f(a) \le f(b)$ and $f(x) \le f(y)$. Since $f(x)$ and $f(y)$ are the two lowest frequencies, $f(x) \le f(a)$ and $f(y) \le f(b)$.
- First transformation: exchange the positions of a and x in T to produce a new tree T'.
- Second transformation: exchange the positions of b and y in T' to produce a new tree T''.
- Show that the cost of the trees remains the same.

Optimality proof (4)

First transformation:

$$\begin{aligned}
B(T) - B(T') &= \sum_{c\in C} f(c)d_T(c) - \sum_{c\in C} f(c)d_{T'}(c) \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_{T'}(x) + f(a)d_{T'}(a)] \\
&= [f(x)d_T(x) + f(a)d_T(a)] - [f(x)d_T(a) + f(a)d_T(x)] \\
&= (f(a) - f(x))(d_T(a) - d_T(x)) \\
&\ge 0
\end{aligned}$$

because $0 \le f(a) - f(x)$ and $0 \le (d_T(a) - d_T(x))$

Since T is optimal, $B(T) = B(T')$

Optimality proof (5)

Second transformation:

$$\begin{aligned}
B(T') - B(T'') &= \sum_{c\in C} f(c)d_{T'}(c) - \sum_{c\in C} f(c)d_{T''}(c) \\
&= [f(y)d_{T'}(y) + f(b)d_{T'}(b)] - [f(y)d_{T''}(y) + f(b)d_{T''}(b)] \\
&= [f(y)d_{T'}(y) + f(b)d_{T'}(b)] - [f(y)d_{T'}(b) + f(b)d_{T'}(y)] \\
&= (f(b) - f(y))(d_{T'}(b) - d_{T'}(y)) \\
&\ge 0
\end{aligned}$$

because $0 \le f(b) - f(y)$ and $0 \le (d_{T'}(b) - d_{T'}(y))$

Since T' is optimal, $B(T') = B(T'')$

Conclusion from Lemma 1

- Building up an optimal tree by mergers can begin with the greedy choice of merging together the two code-words with the lowest frequencies.
- This is a greedy choice since the cost of a single merger is the sum of the lowest frequencies, which is the least expensive merge.

Optimality proof: lemma 2 (1)

<u>Lemma 2</u>: Let T be an optimal prefix code tree for code alphabet C. Consider any two sibling code-words x and y in C and let z be their parent in T. Then, considering z as a character with frequency $f(z) = f(x) + f(y)$, the tree $T' = T - \{x,y\}$ represents an optimal prefix code for the code alphabet $C' = C - \{x,y\} \cup \{z\}$.

<u>Proof</u>: we first express the cost $B(T)$ of tree T as a function of the cost $B(T')$ of tree T'.

Optimality proof: lemma 2 (2)

- For all c in $C - \{x,y\}$, $d_T(c) = d_{T'}(c)$ and thus
$$f(c)\, d_T(c) = f(c)\, d_{T'}(c).$$
- Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we get:
$$f(x)\, d_T(x) + f(y)\, d_T(y) = [f(x) + f(y)](d_{T'}(z) + 1)$$
$$= f(z)\, d_{T'}(z) + [f(x) + f(y)]$$
- Therefore, $B(T) = B(T') + [f(x) + f(y)]$
$$B(T') = B(T) - [f(x) + f(y)]$$

Optimality proof: lemma 2 (3)

We prove the lemma by contradiction:

- Suppose that T does not represent an optimal prefix code for C. Then there exist a tree T'' whose cost is better than that of T: $B(T'') < B(T)$.
- By Lemma 1, T'' has two siblings, x and y. Let T''' be the tree with the common parent of x and y replaced by leaf z with frequency $f(z) = f(x) + f(y)$. Then:
$$B(T''') = B(T'') - [f(x) + f(y)]$$
$$< B(T) - [f(x) + f(y)]$$
$$= B(T')$$
yielding a contradiction to T' being an optimal code for C'.

Optimality proof: Huffman algorithm (1)

<u>Theorem</u>: Huffman's algorithm produces an optimal prefix code.

<u>Proof</u>: By induction on the size of the code alphabet C, using Lemmas 1 and 2.

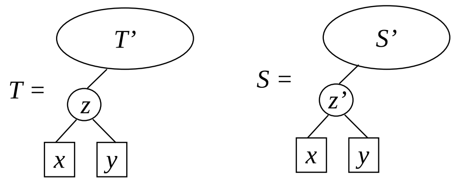
- For $|C| = 2$ it is trivial, since the tree has two leaves, assigned to "0" and "1", both of length 1.

The induction step (1)

- Suppose the Huffman algorithm generates an optimal code for a code of size n, let us prove this for C with $|C| = n + 1$.
- Let T be the tree generated for C by the Huffman algorithm, Let x and y be two nodes with minimal frequencies that the Huffman algorithm picks first. Suppose in contradiction that S is a tree for $|C|=n+1$, which is strictly better than T: $B(S) < B(T)$. By Lemma 1, we can assume that S has x, y as siblings.
- Define the node z' to be their parent, S' to be the sub-tree of S without x and y, T' to be the sub-tree of T without x, y.
- T' is the Huffman code generated for $C - \{x,y\} \cup \{z\}$ with $f(z) = f(x) + f(y)$. S' describes a prefix code for $C - \{x,y\} \cup \{z'\}$ with $f(z') = f(x) + f(y)$.

The induction step (2)

Compare now S' and T':



$B(S') = B(S) - [f(x)d_S(x) + f(y)d_S(y)] + f(z')d_S(z')$

Since $d_S(x) = d_S(y) = d_S(z') + 1$, we get:

$B(S') = B(S) - f(x) - f(y)$ and similarly,

$B(T') = B(T) - f(x) - f(y)$

The induction step (3)

But now if $B(S) < B(T)$ we have that $B(S') < B(T')$.

Since $|S'| = |T'| = n$, this contradicts the induction assumption that T', the Huffman code for $C - \{x.y\} \cup \{z\}$ is optimal!

Extensions and applications

- *d*-ary codes: we merge the *d* objects with the least frequency at each step, creating a new object. whose frequency is the sum of the frequencies
- Many more coding techniques!