

Huffman Coding

Version of September 17, 2016



Outline

- Coding and Decoding
- The optimal source coding problem
- Huffman coding: A greedy algorithm
- Correctness

Example

Suppose we want to store a given a 100,000 character data file. The file contains only 6 characters, appearing with the following frequencies:

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
Frequency in '000s	45	13	12	16	9	5

- A **binary code** encodes each character as a binary string or **codeword** over some given **alphabet** Σ
 - a **code** is a set of codewords.
 - e.g., $\{000, 001, 010, 011, 100, 101\}$
and
 $\{0, 101, 100, 111, 1101, 1100\}$
- are codes over the binary alphabet $\Sigma = \{0, 1\}$.

Given a code C over some alphabet it is easy to **encode** the message using C . Just scan through the message, replacing the characters by the codewords.

Example

$\Sigma = \{a, b, c, d\}$

If the code is

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}$$

then **bad** is encoded as **01 00 11**

If the code is

$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}$$

then **bad** is encoded as **110 0 111**

Fixed-Length vs Variable-Length

- In a **fixed-length code** each codeword has the **same** length.
- In a **variable-length code** codewords may have **different** lengths.

Example

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
fixed-len code	000	001	010	011	100	101
variable-len code	0	101	100	111	1101	1100

Note: since there are 6 characters, a fixed-length code must use at least 3 bits per codeword).

- The fixed-length code requires 300,000 bits to store the file.
 - The variable-length code requires only
 $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000\text{bits}$,
saving a lot of space!
- Goal is to save space!

$$C_1 = \{a = 00, b = 01, c = 10, d = 11\}.$$

$$C_2 = \{a = 0, b = 110, c = 10, d = 111\}.$$

$$C_3 = \{a = 1, b = 110, c = 10, d = 111\}$$

Given an encoded message, **decoding** is the process of turning it back into the original message.

Message is **uniquely decodable** if it can be decoded in only one way.

Example

Relative to C_1 , **010011** is uniquely decodable to **bad**.

Relative to C_2 , **1100111** is uniquely decodable to **bad**.

Relative to C_3 , **1101111** is not uniquely decipherable
it could have encoded either **bad** or **acad**.

In fact, *any* message encoded using C_1 or C_2 is uniquely decipherable. **Unique decipherability** property is needed in order for a code to be useful.

Fixed-length codes are always uniquely decipherable. **WHY?**

We saw before that fixed-length codes do not always give the best **compression** though, so we prefer to use variable length codes.

Definition

A code is called a **prefix (free) code** if no codeword is a prefix of another one.

Example

$\{a = 0, b = 110, c = 01, d = 111\}$ is *not* a prefix code.

$\{a = 0, b = 110, c = 10, d = 111\}$ is a prefix code.

Important Fact: Every message encoded by a prefix free code is uniquely decipherable.

- Because no codeword is a prefix of any other, we can always find the first codeword in a message, peel it off, and continue decoding.

Example

code: $\{a = 0, b = 110, c = 10, d = 111\}$.

$$01101100 = 01101100 = abba$$

We are therefore interested in finding *good* (best compression) prefix-free codes.

- Coding and Decoding
- The optimal source coding problem
- Huffman coding: A greedy algorithm
- Correctness

The Optimal Source Coding Problem

Huffman Coding Problem

Given an alphabet $A = \{a_1, \dots, a_n\}$ with frequency distribution $f(a_i)$, find a binary prefix code C for A that **minimizes** the number of bits

$$B(C) = \sum_{i=1}^n f(a_i) L(c_i)$$

needed to encode a message of $\sum_{i=1}^n f(a_i)$ characters, where

- c_i is the codeword for encoding a_i , and
- $L(c_i)$ is the length of the codeword c_i .

Example

Problem

Suppose we want to store messages made up of 4 characters a, b, c, d with frequencies 60, 5, 30, 5 (percents) respectively. What are the fixed-length codes and prefix-free codes that use the least space?

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store these 100 characters,

(1) the fixed-length code requires $100 \times 2 = 200$ bits,

(2) the prefix code uses only

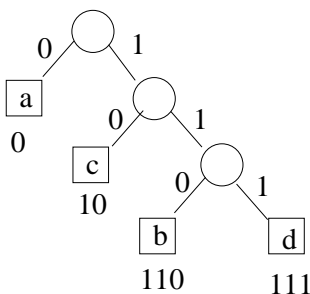
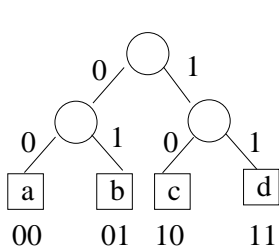
$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150 \text{ bits}$$

a 25% saving.

Remark: We will see later that this is the optimum (lowest cost) prefix code.

Correspondence between Binary Trees and Prefix Codes

1-1 correspondence between **leaves** and **characters**.



- Left edge is labeled 0; right edge is labeled 1
- The binary string on a **path from the root to a leaf** is the **codeword** associated with the character at the leaf.

Minimum-Weight External Pathlength Problem

- d_i , the depth of leaf a_i , is equal to $L(c_i)$, the depth of the codeword associated with that leaf.
- $B(T)$, weighted external pathlength of tree T is same as $B(C)$, the number of bits needed to encode message with corresponding code C

$$\sum_{i=1}^n f(a_i) d_i = \sum_{i=1}^n f(a_i) L(c_i)$$

Definition (Minimum-Weight External Pathlength Problem)

Given weights $f(a_1), \dots, f(a_n)$, find a tree T with n leaves labeled a_1, \dots, a_n that has minimum weighted external path length.

The Huffman encoding problem is equivalent to the minimum-weight external pathlength problem.

- Coding and Decoding
- The optimal source coding problem
- Huffman coding: A greedy algorithm
- Correctness

Set S be the original set of message characters.

① (Greedy idea)

- Pick two **smallest** frequency characters x, y from S .
- Create a subtree that has these two characters as leaves.
- Label the root of this subtree as z .

②

- Set frequency $f(z) = f(x) + f(y)$.
- Remove x, y from S and add z to S
 - $S = S \cup \{z\} - \{x, y\}$.
 - Note that $|S|$ has just decreased by one.

Repeat this procedure, called **merge**, with the new alphabet S , until S has only one character left in it.

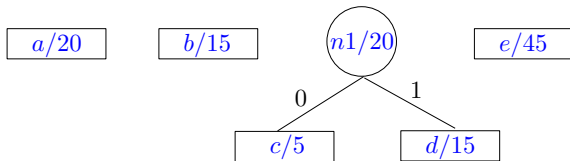
The resulting tree is the **Huffman code tree**.

- It encodes the **optimum** (minimum-cost) prefix code for the given frequency distribution.

Example of Huffman Coding

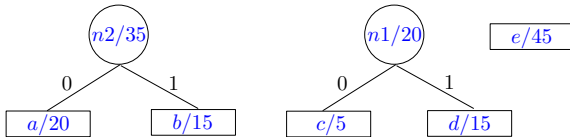
Let $S = \{a/20, b/15, c/5, d/15, e/45\}$ be the original character set S alphabet and its corresponding frequency distribution.

- 1 The first Huffman coding step merges c **and** d .
(could also have merged c and b).



Now have $S = \{a/20, b/15, n1/20, e/45\}$.

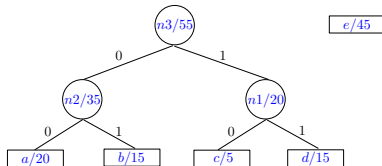
- 2 Algorithm merges a **and** b (could also have merged $n1$ and b)



Now have $S = \{n2/35, n1/20, e/45\}$.

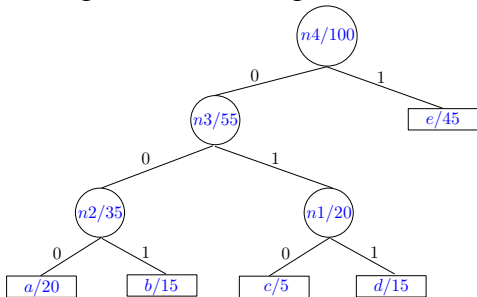
Example of Huffman Coding – Continued

- ❶ Algorithm merges $n1$ and $n2$.



Now have $S_3 = \{\mathbf{n3/55}, e/45\}$.

- ❷ Algorithm next merges e and $n3$ and finishes.



The Huffman code is:
 $a = 000$, $b = 001$,
 $c = 010$, $d = 011$,
 $e = 1$.

Huffman Coding Algorithm

Given character set S with frequency distribution $\{f(a) : a \in S\}$:
The binary Huffman tree is constructed using a **priority queue**, Q ,
of nodes, with frequencies as keys.

Huffman(S)

```
 $n = |S|;$   
 $Q = S;$  // the future leaves  
for  $i = 1$  to  $n - 1$  do  
    // Why  $n - 1$ ?  
     $z = \text{new node};$   
     $\text{left}[z] = \text{Extract-Min}(Q);$   
     $\text{right}[z] = \text{Extract-Min}(Q);$   
     $f[z] = f[\text{left}[z]] + f[\text{right}[z]];$   
     $\text{Insert}(Q, z);$   
end  
return  $\text{Extract-Min}(Q);$  // root of the tree
```

Running time is $O(n \log n)$, as each priority queue operation takes time $O(\log n)$.

- Coding and Decoding
- The optimal source coding problem
- Huffman coding: A greedy algorithm
- Correctness

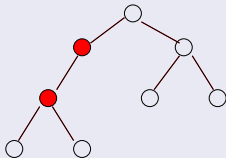
Lemma 1

Lemma (1)

An *optimal prefix code* tree must be “full”, i.e., 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. □

Lemma 2

Lemma (2)

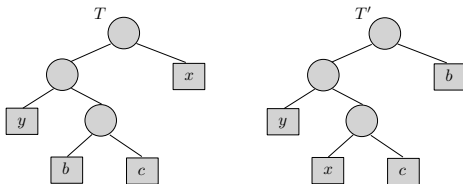
Let T be *prefix code* tree and T' the tree obtained by swapping two leaves x and b in T . If,

$$f(x) \leq f(b), \quad \text{and} \quad d(x) \leq d(b)$$

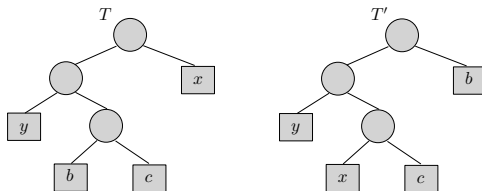
then,

$$B(T') \leq B(T).$$

i.e., swapping a lower-frequency character downward in T does not increase T 's cost.



Lemma 2



Proof.

$$\begin{aligned} B(T') &= B(T) - f(x)d(x) - f(b)d(b) + f(x)d(b) + f(b)d(x) \\ &= B(T) + \underbrace{(f(x) - f(b))}_{\leq 0} \underbrace{(d(b) - d(x))}_{\geq 0} \\ &\leq B(T). \end{aligned}$$

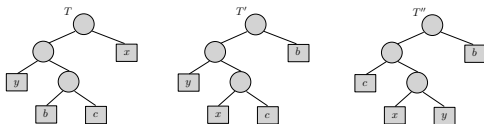


Lemma (3)

Consider the two characters x and y with the *smallest frequencies*. There is an optimal code tree in which these two letters are *sibling* leaves at the *deepest* level of the tree.

Proof: Let

- T be any optimal prefix code tree,
- b and c be two siblings at the deepest level of the tree (such siblings must exist because T is full).



Assume without loss of generality that
 $f(x) \leq f(b)$ and $f(y) \leq f(c)$

- (If necessary) swap x with b and swap y with c .
- Proof follows from Lemma 2.

Lemma (4)

- Let T be a prefix code tree and x, y two sibling leaves.
- Let T' be obtained from T by removing x and y , naming the parent z , and setting $f(z) = f(x) + f(y)$
- Then

$$B(T) = B(T') + f(x) + f(y).$$

$$\begin{aligned} B(T) &= B(T') - f(z)d(z) + f(x)(d(z) + 1) + f(y)(d(z) + 1) \\ &= B(T') - (f(x) + f(y))d(z) + (f(x) + f(y))(d(z) + 1) \\ &= B(T') + f(x) + f(y). \end{aligned}$$

Huffman Codes are Optimal

Theorem

The prefix code tree (hence the code) produced by the Huffman coding algorithm is optimal.

Proof: (By induction on n , the number of characters).

- **Base case** $n = 2$: Tree with two leaves. Obviously optimal.
- **Induction hypothesis:** Huffman's algorithm produces optimal tree in the case of $n - 1$ characters.
- **Induction step:** Consider the case of n characters:
 - Let H be the tree produced by the Huffman's algorithm.
 - Need to show: H is optimal.

Huffman Codes are Optimal

- **Induction step** (cont'd):
 - Following the way Huffman's algorithm works,
 - Let x, y be the two characters chosen by the algorithm in the first step. They are sibling leaves in H .
 - Let H' be obtained from H by
 - (i) removing x and y , (ii) labelling their parent z , and
 - (iii) setting $f(z) = f(x) + f(y)$
 - H has S ; H' has $S' = S - \{x, y\} \cup \{z\}$
 - H' is the tree produced by Huffman's algorithm for S' .
 - By the induction hypothesis, H' is optimal for S' .
 - By Lemma 4, $\mathbf{B}(H) = \mathbf{B}(H') + f(x) + f(y)$.

Huffman Codes are Optimal

- **Induction step** (cont'd):

- By Lemma 3, there exists *some* **optimal** tree T in which x and y are sibling leaves.
- Let T' be obtained from T by removing x and y , naming their parent z , and setting $f(z) = f(x) + f(y)$.
- T' is also a prefix code tree for S' .
- By Lemma 4,

$$B(T) = B(T') + f(x) + f(y).$$

- Hence

$$\begin{aligned} B(H) &= B(H') + f(x) + f(y) \\ &\leq B(T') + f(x) + f(y) \quad (H' \text{ is optimal for } A') \\ &= B(T). \end{aligned}$$

- **Therefore, H must be optimal!**