Algorithms Fall 2019

Lecture : Greedy Algorithms - Huffman Codes

Imdad Ullah Khan

Contents

1 Compression

Compression is a tool used to facilitate storing large files by reducing their size. There are two different types of goals associated with compression:

- Maximize ease of access, manipulation and processing
- Minimize size−especially important when storage or transmission is expensive.

Naturally, these two objectives are often at odds with each other. We will focus on the second objective. In general, data cannot be compressed. For example, we can not, without any loss of information, represent all m-bit strings using $(m-1)$-bit strings, since there are 2^m possible m-bit strings and only 2^{m-1} possible $(m-1)$-bit strings. So when is compression possible?

- If only a relatively small number of the possible m-bit strings appear, compression is possible.
- If the same "long" substring appears repeatedly, we could represent it by a "short" string.
- If we relax the requirement that every string have a unique representation, then compression might work but make "similar" strings identical.

Lossless and lossy: Let D be the original document and D' be the recovered document (the one decompressed after some kind of compression). In lossless compression, we require that $D = D'$. This means that the original document can always be recovered exactly from the compressed document. Examples include: Huffman coding, Lempel–Ziv (used in .gif images) etc. In lossy compression, D' is close enough but not necessarily identical to D. Examples include: .mp3 (audio), .jpg (images) and .mpg (videos) etc.

Adaptive and non-adaptive: Compression algorithms can be either adaptive or non-adaptive. A non-adaptive algorithm assumes prior knowledge of the data (e.g., character frequncies) whereas an adaptive algorithm assumes no knowledge of the data, but builds such knowledge.

The purpose of introducing the above terminology and definitions is to make you familiar with compression and you can easily justify different techniques of compression with regards to their nature.

Lets see compression with a concrete example **(MP3 compression)**, first there is the process of conversion from analog to digital (which is beyond the scope of this lecture), followed by compressing the digital signal. , In the MP3 audio compression scheme, a sound signal is encoded in three steps.

1. It is digitized by sampling at regular intervals, yielding a sequence of real numbers $s_1, s_2, \ldots, s_T$. For instance, at a rate of $44,100$ samples per second, a 50-minute symphony would correspond to $T = 50 \times 60 \times 44100 \approx 130$ million measurements.

2. Each real-valued sample s_i is quantized: approximated by a nearby number from a finite set $\sum$. This set is carefully chosen to exploit human perceptual limitations, with the intention that the approximating sequence is indistinguishable from $s_1, s_2, \ldots, s_T$ by the human ear.

3. The resulting string of length T over alphabet $\sum$ is then encoded in binary.

This example shows the process of compression and encoding scheme. Since computers ultimately operate on sequences of bits (i.e., sequences consisting only of the symbols 0 and 1), one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into long strings of bits. We will discuss different techniques for encoding strings of characters as binary codes.

2 Binary Codes

A binary code represents text, computer processor instructions, or any other data using a two-symbol system. The two-symbol system used is often the binary number system's 0 and 1.

Example 1. *Suppose that we have a* $100,000$ *characters data file that we wish to store in computer. The file contains only 6 characters, appearing with the following frequencies:*

	a	b	c	d	e	f	Total
Frequency in 000's	45	13	12	16	9	5	100

A binary code encodes each character as a binary string or codeword. We would like to find a binary code that encodes the file using as few bits as possible i.e. compresses it as much as possible. There are two different coding schemes to represent characters in binary.

Fixed Length Codes:

In this approach, fixed number of source symbols are encoded into a fixed number of output symbols. i.e. each codeword has the same length. For example, English characters $a - z$ and some punctuations make a total of 32 different characters, so for every character to have a unique fixed length binary code, what should be length of a codeword? It will need 5 bits for each character to be uniquely represented, as $2^5 = 32$. ASCII and Unicode are common examples of fixed length codes. Here the problem is length of codeword (number of bits for a code), it grows very big and one can face storage problem. To overcome this issue, we try to reduce the number of bits somehow.

Variable Length Codes:

In this approach, codewords may have different lengths. The motivation for variable length code is that frequencies of different characters may vary in a document and as our goal is to minimize the overall size of binary encoded files with lossless encoding, most frequent characters can be encoded with shorter codewords.

Example 2. *Lets see an example of English alphabet, coding and translation of a small string. Then we will see that how total length is reduced substantially in comparison with fixed length coding scheme. We will see it mean compression is achieved ? Lets see.*

Characters	a	b	c	d	e	f	Total
Frequency	45k	13k	12k	16k	9k	5k	100k
Fixed-Length Code	000	001	010	011	100	101	3×100k
Var. Length Code	0	101	100	111	1101	1100	224k

The fixed length-code requires $300,000$ bits to store the file. The variable-length code uses only $224,000$ bits, saving about 25% in size, i.e. the file is compressed by about 25%, saving a lot of space! Can we do better than this?

Example 3. *Lets take an another example, suppose we have the following codes for the string* ***aabbaaabcd****.*

Characters	a	b	c	d	Total	String
Frequency	5	3	1	1	10	*aabbaaabcd*
Fix-Len Code	00	01	10	11	$2 \cdot 10 = 20$ bits	00000101000000011011
Var. Len Code	0	10	110	111	$1 \cdot 5 + 2 \cdot 3 + 3 \cdot 1 + 3 \cdot 1 = 17$ bits	00101000010110111

Example 4. *Lets see another example, can we further decrease the length of codewords? It can be seen in the following example that there is another variable-length coding scheme which does even more compression than that of the previous one,*

Characters	a	b	c	d	Total	String
Frequency	5	3	1	1	10	*aabbaaabcd*
Fix-Len Code	00	01	10	11	$2 \cdot 10 = 20$ bits	00000101000000011011
Var. Len Code	0	10	110	111	$1 \cdot 5 + 2 \cdot 3 + 3 \cdot 1 + 3 \cdot 1 = 17$ bits	00101000010110111
Var. Len Code2	0	1	01	10	$1 \cdot 5 + 1 \cdot 3 + 2 \cdot 1 + 2 \cdot 1 = 12$ bits	001100010110

Although this strategy seems like a good strategy, as it minimizes the length of encoding, there is a critical issue with this variable length technique, i.e. ambiguity in decoding of the variable length codes. The following example highlights this issue

Example 5. *Given a code(corresponding to some alphabet* Γ*) and a message it is easy to encode the message. Just replace the characters by the codewords. Example:* $\Sigma = \{a, b, c, d\}$

- *code* $C1$ *(fixed length) given as* $C1 \rightarrow \{a = 00, b = 01, c = 10, d = 11\}$, ***bad*** *is encoded as* 010011
- *code* $C2$ *(variable length) given as* $C2 \rightarrow \{a = 0, b = 110, c = 10, d = 111\}$, ***bad*** *is encoded as* 1101111
- *code* $C3$ *(variable length) given as* $C3 \rightarrow \{a = 1, b = 110, c = 10, d = 111\}$, ***bad*** *is encoded as* 1101111

Decoding: *Given an encoded message, decoding is the process of converting it back into the original message.*
For example relative to $C1$, 010011 *is uniquely decodable to* ***bad****. But, relative to* $C3$*; 1101111 is not uniquely decipherable since it could have encoded either* ***bad*** *or* ***acad****. here* $a = 1$ *is prefix of* $b = 110$*.*

Therefore, the unique decipherability property is needed in order for a code to be useful. In order to ensure a code is unique, we define prefix-free codes.

Prefix Free Codes:

In this variable length encoding scheme, no codeword is a prefix of another codeword, hence named prefix free codes. This must be true for all pairs of codes. There is no ambiguity while decoding prefix free codes.

Example 6. $\{a = 0, b = 110, c = 10, d = 111\}$ *is a prefix-free code.*

In example 5, $C2$ *is a prefix free code and* 1101111 *is uniquely decodable to* ***bad*** *whereas* $C3$ *is not a prefix free code. In example 4, Var. Len Code is prefix free whereas Var. Len Code2 is not prefix-free.*

Visualizing Codes as Binary Trees:

Binary codes can be represented as binary trees. Nodes are labeled with characters and edges to the left and right child of a node are labeled 0 and 1 respectively. The bits along the path from the root to a node is the code for the character at that node. If the symbols or characters are labeled only at the leaf nodes, the codes are prefix free. A character can be decoded easily by reading the encoded string from left to right starting at the root of the tree and moving to left and right child depending on the encoded bit being 0 or 1 until a leaf node is reached, which is labeled with the character the code substring decodes to. The encoded string from the next bit is again read in the binary tree starting from the root till a lead node is reached, which represents the next decoded character. The process is repeated until the encoded string is exhausted. Clearly, this can be easily automated and there is no ambiguity.

Figure 1 and figure 2 show the trees for two codes of example 2. Note that these are not binary search trees, since the leaves node not appear in sorted order and internal nodes do not contain character keys.

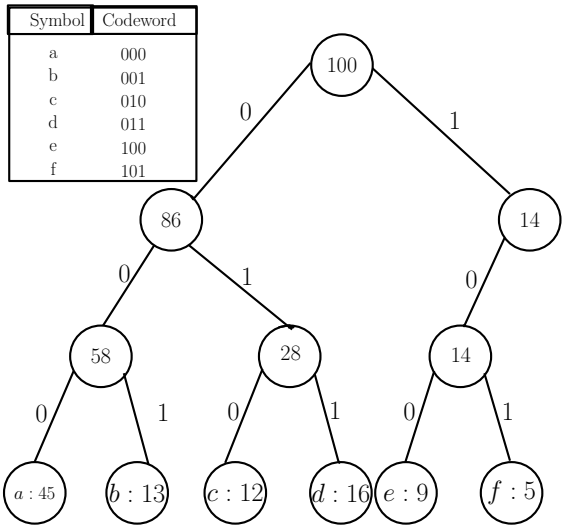

Figure 1: Tree Corresponding to the Fixed Length coding scheme in Example 2 .

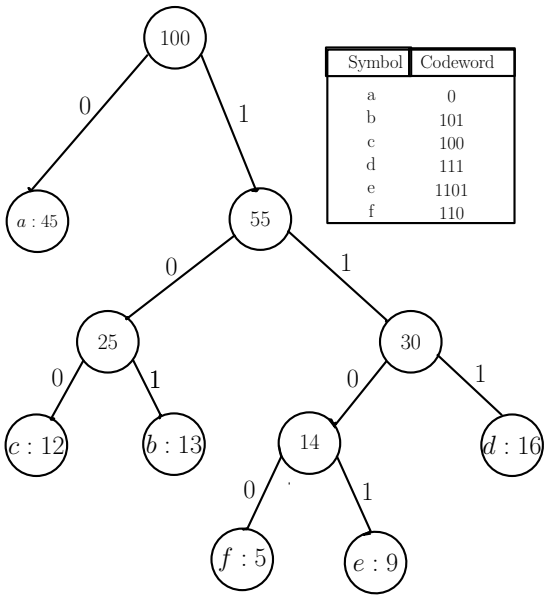

Figure 2: Tree Corresponding to the variable Length coding scheme in Example 2 .

Number of bits needed to encode a symbol is the depth of the corresponding leaves, as shown in above example. For a document D, where frequency of alphabet a_i is $f(a_i)$, compressed with an encoding scheme represented as a binary tree T, where node i is labeled with a_i, the number of bits needed for encoding is

$$B(D) = \sum_{i \in \Sigma} f(a_i) \cdot \ [\text{ depth of } i \text{ in } T]$$

An optimal code for a file is always represented by a full binary tree, in which every non leaf node has two children as shown in Figure 3). The fixed-length code in the above example example is not optimal since its tree (Figure 1) is not a full binary tree, i.e. it contains codewords beginning with $10\ldots$, but none beginning with $11\ldots$.

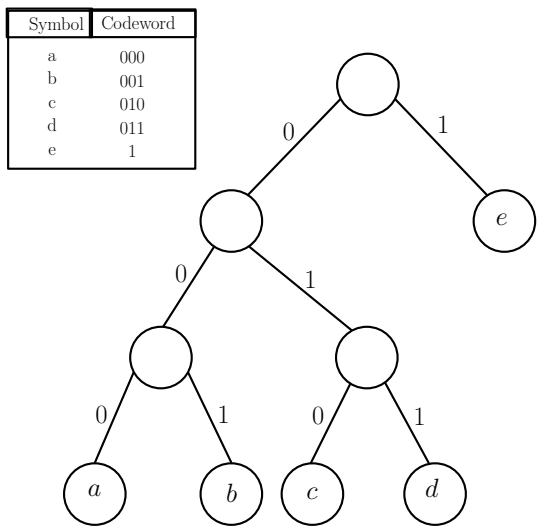

Figure 3: Correspondence between Binary Tree and prefix codes.

Since an optimal code tree T must be a full binary tree, it can said that if C is the alphabet set from which the characters are drawn and all character frequencies are positive, then the tree for an optimal free fix code tree T has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|$ -1 internal nodes.

The problem of finding an optimal encoding can therefore be formulated as follows.

Problem 1. *Given an alphabet* $\Sigma = \{a_1,\ldots,a_n\}$ *with frequency distribution* $f(a_i)$*, find a binary prefix code* C *for* A *that minimizes the number of bits* $B = \sum_{a=1}^{n} f(a_i)L(c(a_i)) = \sum_{a\in\Sigma} f(a_i)\cdot$ *[depth of* a *in* T*] needed to encode a message of* $\sum_{a=1}^{n} f(a)$ *characters, where* $c(a_i)$ *is the codeword for encoding* a_i*,* $L(c(a_i))$ *is the length of the codeword* $c(a_i)$ *and* B *is the sum of code lengths weighted by frequency.*

We begin to devise a solution to this problem by discussing some initial sub-optimal ideas. The **Fano-Shanon code** is a top/down divide and conquer approach which partitions Σ into Σ_1 and Σ_2 each with roughly equal frequency distribution, i.e. half the total frequencies. T_1 for Σ_1 and T_2 for Σ_2 are computed recursively. All codes of T_1 are prepended with a 0 and the other codes with a 1, i.e. T_1 and T_2 are fixed as left and right subtree of new root. These types of prefix codes can be fairly good in practice. However, no version of this top-down splitting strategy is guaranteed to always produce an optimal prefix code. Shannon and Fano knew that their approach did not always yield the optimal prefix code, but they could not see how to compute the optimal code without a brute-force search. This problem was solved by David Huffman.

Huffman developed a greedy approach for constructing optimal prefix free codes. The codes produced by this algorithm are called **Huffman Codes**.

3 Greedy Algorithm to Construct Optimal Prefix Free Codes

The following greedy algorithm constructs the Huffman's Prefix Free Code Tree.

Step 1. *Let* x *and* y *be two characters with lowest frequency (ties broken arbitrarily) from alphabet set* A*. Construct a subtree which has these 2-characters as leaves. Label the root of this subtree as* z*.*

Step 2. *Frequency of this meta-character* z *is the sum of the frequencies of its leaf nodes* x *and* y*. Remove* x, y *from and add* z *to the alphabet set to form* $A' = A \cup \{z\} - \{x, y\}$*. Note that* $|A'| = |A| - 1$*.*

Step 3. *Repeatedly perform steps* 1 *and* 2 *until* A *contains only one character.*

As shown in Figure 4, the more frequent characters have shorter bit codes and less frequent characters have longer bits codes which minimizes the length of the encoding. All codes are prefix-free codes, therefore each code can be decoded to a unique character. Hence, both issues have been resolved.

Example 7. *Consider the alphabet set along with frequencies:*
$\{a : 45, b : 20, c : 10, d : 17, e : 6, f : 5, g : 18\}$.
The code tree generated by the Huffman algorithm shown in Figure 4.
The optimal Huffman codes are:
$a : 0 \quad b : 100 \quad c : 1010 \quad d : 110 \quad e : 10111 \quad f : 10110 \quad g : 111$

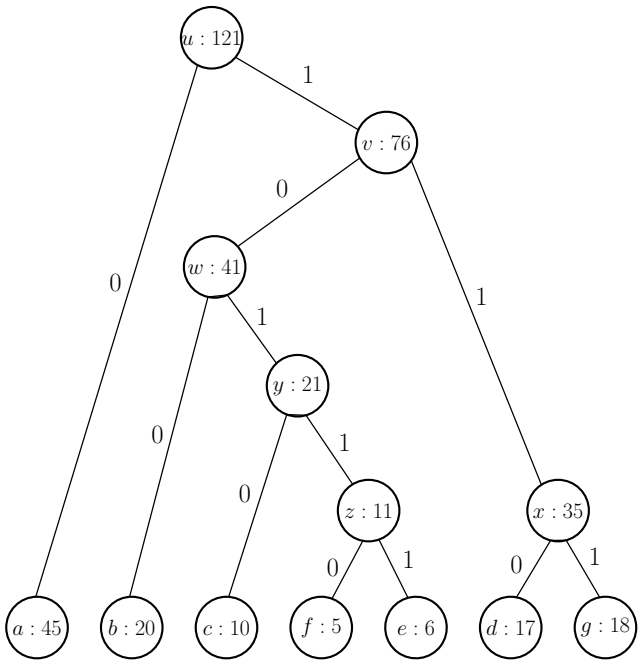

Figure 4: Optimal Huffman Codes

Implementation and Runtime:

The pseudo code for an implementation of Huffman Tree is given in Algorithm 1 takes $O(n)$ preparation time, $n-1$ iterations with 2FindMin operations and other $O(1)$ operations in each iteration. Thus, the total runtime of this implementation is $O(n^2)$, which can be improved by utilizing a more suitable data structure designed for repeated FindMin operations.

Algorithm 1 : HUFFMAN-TREE($S, f(\cdot)$)

for $x \in \mathcal{S}$ **do**
  MakeNode(x) ▷ x is both symbol and pointer
for $i = 1$ to $n-1$ **do**
  $x \leftarrow$ FindMin($\mathcal{S}$)
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x\}$
  $y \leftarrow$ FindMin($\mathcal{S}$)
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{y\}$
  MakeNode(z)
  $z \cdot freq \leftarrow x \cdot freq + y \cdot freq$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{z\}$
return the only node in $\mathcal{S}$

Recall the priority queue data structure. The implementation of Huffman-Tree using a priority queue in Algorithm 2 which performs n Insert operations initially. In each of the $n-1$ iterations, 1 Insert and 2 ExtractMin operations are done. Each of these operations takes $O(\log n)$ time and the total number of operations is $n + 3(n-1)$. Therefore, the total runtime is now reduced to $O(n \log n)$.

Algorithm 2 Huffman-Tree $(\mathcal{S}, f(\cdot))$

$\mathcal{H} \leftarrow$ InitializeHeap$(\mathcal{S}, f(\cdot))$ ▷ min heap keyed by frequencies
for $i = 1$ to $n - 1$ **do**
  $z \leftarrow$ NewNode()
  $x \leftarrow$ ExtractMin$(\mathcal{H})$
  $y \leftarrow$ ExtractMin$(\mathcal{H})$
  $z \cdot left \leftarrow x$
  $z \cdot right \leftarrow y$
  $z \cdot freq \leftarrow x \cdot freq + y \cdot freq$
  Insert$(\mathcal{H}, z)$
return ExtractMin$(\mathcal{H})$ ▷ Return the root of the tree

Correctness:

The Huffman Codes algorithm is correct because by definition because all the alphabets are at the leaf nodes, and the code for each alphabet is the path from the root to the alphabet node. Therefore, code of no alphabet can be a prefix of another and the Huffman codes are prefix-free.

Optimality:

To prove that the Huffman Codes algorithm is optimal, we show that the problem of determining an optimal prefix free code exhibits the greedy-choice and optimal-substructure properties.

Lemma 1 (Greedy Choice)**.** *Let S be a set of characters and x and y be the least and second least frequent symbols in S. Then there exists an optimal prefix free code scheme where the codes for x and y have the same length and differ only in the last bit. i.e. In some optimal tree T, the two least frequent symbols x and y are siblings.*

Proof. Let T be an optimal prefix free code tree, and let a and b be two siblings at the maximum depth of the tree (must exist because T is full). Assume without loss of generality that $f(a) \leq f(b)$. Since x and y are the two least frequent symbols and $f(x) \leq f(y)$, it follows that $f(x) \leq f(a)$ and $f(y) \leq f(b)$.

Let $L(x)$ be the depth of the node for symbol x in T. Since a and b are at the deepest level of the tree, we know that $L(a) \geq L(x)$ and $L(b) \geq L(y)$. Let the encoding cost of T be $B(T)$.

Let T' be the resulting tree if the positions of x and a in T are switched and let T'' be the resulting tree if the positions of y and b in T' are switched, as shown in figure 5

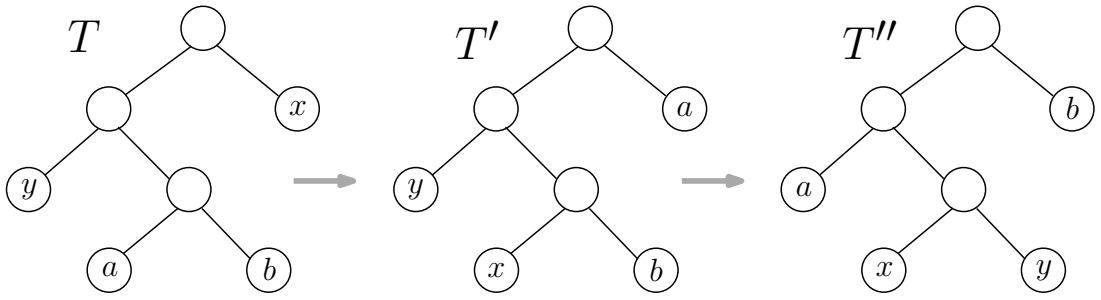

Figure 5: Exchanging a with x and b with y to get T' and T''

$$\begin{aligned} B(T') &= B(T) - f(x)L(x) - f(a)L(a) + f(x)L(a) + f(a)L(x) \\ &= B(T) - (f(a) - f(x))(L(a) - L(x)) \\ &\leq B(T) \end{aligned}$$

By a similar argument as above, $B(T'') \leq B(T)$.

Therefore, since T is optimal, $B(T'') = B(T') = B(T)$ and T' and T'' are optimal trees. Hence, the final tree T'' satisfies the statement of the claim. □

Lemma 2 (Optimal Substructure)**.** *Given an alphabet S with frequencies $f(\cdot)$, let:*

- *x, y be the two most infrequent characters in S (with ties broken arbitrarily)*
- *$z \notin S$ be a new symbol, with $f'(z) \leftarrow f(x) + f(y)$*
- *$S' = S \backslash \{x, y\} \cup \{z\}$*
- *T' be an optimal tree for (S', f')*

Then, T is an optimal tree for (S, f), where T is a copy of T' in which the leaf for z has been replaced by an internal node having x and y as children.

Proof.

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

Suppose T is not optimal. If T is not optimal, then there is a T'' with $B(T'') < B(T)$ in which x and y are siblings. Then,

$$\begin{aligned} B(T'') &= B(T') - f(x) - f(y) \\ &\leq B(T') \end{aligned}$$

Considering T'' with the parent of x and y as a leaf is a better tree than T' for $[S', f'(\cdot)]$ which contradicts the optimality of T'. Therefore, T must have been optimal in the first place.

□

The proof of optimality of Huffman's algorithm, i.e. is produces an optimal prefix code tree for all inputs, is by induction on n, the number of symbols.

Proof. The basis case ($n = 1$) is trivial, since there is only one tree possible. For $n > 2$, then by Lemma 2, the two characters *xandy* of lowest probability are siblings at the deepest level of an optimal tree. Huffman's algorithm replaces these nodes by a symbol z whose frequency is the sum of the frequencies of x and y. The induction hypothesis is that the Huffman's algorithm computes the optimum tree over the resulting alphabet of $n - 1$ symbols. Call it T_{n-1}. Replacing z with nodes x and y results in a tree T_n whose cost is higher by a fixed amount $f(z) = f(x) + f(y)$. Since T_{n-1} is optimal, and the cost of replacement does not depend on the tree's structure, T_n is also optimal. □