Huffman Codes

Glen G. Langdon, Jr.

©1999, Glen G. Langdon, Jr.
Chapter 3

Huffman Codes

Based partly on lecture notes of David Huffman

3.1 Introduction

To review what we’ve covered, Chapter One, *Introduction to Data Compression*, points out that the data compression field employs computer algorithms and data files in digital form. Knowledge of the symbol probabilities is needed before many of the existing coding techniques can achieve good compression.

Chapter Two, *Coding in General*, discussed examples of Variable Length Code (VLC) design. One design compressed a 16-bit data sequence to a 14-bit code string. Instead of achieving compression, a bad designed VLC code example expanded the resulting code string to 21 bits. **Compression** does not come for free, it requires knowledge of symbol frequencies. Chapter Two also presents the use of table lookup for encoding and decoding.

The early coding techniques for data compression were usually one of two types. The earliest type was the Fixed-to-Variable Length (FVL) codes. The term fixed length refers to the original data employing a fixed number of bits per data item.

An example of a fixed-length code is the 7-bit ASCII code. The 8-bit byte is commonly the smallest accessible unit for early computer memories. The 16-bit short integer was a popular computer register size in the minicomputer era of the 1970s. The trend continues with newer processors using 32-bit and 64-bit register sizes.

In using FVL coding, each fixed-length data item is **mapped** to, or associated with, a **code word** whose length varies according to the relative frequency of the symbol being encoded. The mapping from symbol to code word is not done in a haphazard manner. Each data item is mapped to a code word according to that data item’s probability such that the more probable symbols receive the shorter code words.

This chapter shows that the Huffman algorithm [Huf 52] is a method for **optimally** assigning code words to symbols given that we accurately know and use the symbol probabilities.

We acknowledge that some of this material is based on a class handout by the authors’ colleague Prof. David A. Huffman, when giving a substitute lecture for the author.

3.2 Prefix codes and the Huffman method

We draw on the following covered in Chapter Two.

1. For a prefix code, no code word is the prefix of another code word.
2. For a prefix code, each code word must be identified with a leaf of the code tree.
3. For optimal compression, the longest code words are assigned to the least probable symbols.

The code word set is in one-to-one correspondence with the code tree’s set of leaf nodes. A code tree leaf is assigned to each symbol of alphabet $A$. The assigned code word is the binary “roadmap” of 0 and 1 tree branches (or path segments) along the path from the root node to the symbol’s leaf node.

The traversal from the root node to each given leaf node **automatically** generates a code word set with the **prefix property**: no code word is the prefix of another code word. Figure 3.1(a) has the prefix property. This means, given code word 0, that symbol a can be decoded immediately if it is the first bit value of the remaining encoded code string to be decoded.

An example of a set of code words that does not have the prefix property appears in Figure 3.1(b): \{0, 01, 11\}. The code \{0, 01, 11\} reverses the order of the bits in the prefix code word set \{0, 10, 11\} for Figure 3.1(a). The prefix property is violated for the new (reversed) set of bit strings because code word 0 is a prefix of code word 01.

In Figure 3.1(c), notice that 0 is the parent node of 01, so if 0 is a code word, then 01 cannot be a prefix...
code word. Consider decoding a code string with this non-prefix code and the next two bits are 01. Similarly, neither code word 110 or 111 has the prefix property if 11 is a code word.

The prefix property always implies that once we recognize a code word, we can decode it. This means if 11 is a code word, we can decode 11 as soon as it becomes the first bit of the yet to be decoded part of the code string.

Without the prefix property, the decoder may need look deep into the code string past several code words to determine the length of the next code word and decode the next symbol: what a bother! Thus, non-prefix codes are rarely used, if at all.

3.3 The Code Design Problem

A code associates a code word with each data symbol, and the code string is a concatenation of code words (placing code words side by side). The design of the code requires an estimate of the symbol probabilities, since compression is achieved by assigning shorter code words to more frequent events. The use of a code designed with an incorrect estimate of the probabilities can generate a code string that is larger than the original data.

Huffman’s code-construction algorithm is optimal in that no other association of code words to symbols for the given symbol frequencies can compress better.

Often, more than one code table can deliver optimal performance. The set of code words created by the Huffman algorithm necessarily have the prefix property as a built-in property.

A suboptimal technique called the Shannon-Fano code existed at the time graduate student David Huffman took a course from Professor Fano at MIT. The contribution of David Huffman was to discover an optimal algorithm to assign code words to symbols according to their probabilities. Huffman also proved the optimality of the method. A second benefit (to Huffman) was that Fano exempted him from taking the final exam.

3.4 The Shannon-Fano algorithm

The code design problem was first addressed with an “almost” optimal algorithm called the Shannon-Fano coding algorithm. The Shannon-Fano algorithm is a “top-down” algorithm. In contrast, the Huffman algorithm is a “bottom-up” algorithm.

3.4.1 The General Coding Scenario

Consider the following scenario. Data file \( S \), comprised of 7-bit ASCII characters is very large and we are running out of disk space. The 7-bit ASCII symbol alphabet \( \mathcal{A} \) has \( M = 128 \) symbols \( (2^7 = 128) \).

Our task is to encode data file \( S \) with an optimal prefix code, which is a statistical technique (short code words for popular symbols). For lossless data compression, we need to use a statistical technique, which means we get our best compression when we know the relative frequencies of the symbols to be compressed.

The Shannon-Fano coding method (Huffman coding’s precursor) and the Huffman algorithm both have the same starting point: assign a number to each symbol value that is proportional to the relative frequency of that symbol in the data file. These numbers ideally come from the histogram of the symbols found in the data file.

Typically, the number associated with each ASCII character \( a_m \) is its frequency count \( \text{Ct}(a_m) \). Symbol \( a_m \)’s relative frequency should be \( \frac{\text{Ct}(a_m)}{N} \), where \( N \) is the total of all the counts of the individual symbols (i.e., bytes) such as \( a_m \) found in file \( S \). The number \( N \) should also be the file length in bytes.

Chapter One has mentioned histograms and empirical probabilities. Most studies of the compressibility of a data file \( S \) will count the frequencies of each byte.

3.4.2 The Shannon-Fano Method

Given a set of symbol counts, the Shannon-Fano method first sorts the set of symbol counts (or
frequencies) into an ordered list according to magnitude. Let the list of magnitude ordered count numbers run from left to right in decreasing order. The Shannon-Fano algorithm maintains the order of the lists and sublists during its execution (in contrast to the Huffman algorithm covered later in the chapter).

In what follows, we employ the term count to indicate the numbers the algorithm acts upon. Following the sort operation on the count magnitudes, the next step of the Shannon-Fano algorithm is the partitioning recursion.

The partitioning step locates the partition point: a point between two adjacent counts in the ordered list. The partition point of the ordered count list creates two sublists (one to the right of the partition point and the other to the left) with the following property: the two sublists so defined have approximately equal count totals. The list subdivision process continues until there are no more sublists with two or more counts.

**Definition 1 (Partition Point)** More formally, the partition point is that position between two count values in the list under consideration yielding the smallest absolute difference between the sums of the two sublists created by that partition point.

When dealing with the full list of counts, the partition point at this first level divides the list of symbol counts into two sublists. The left list becomes the left sibling of the root node, and the other list the right sibling. The first partitioning step creates a two-sibling binary tree of depth one below the root.

At the second level, the partition step is recursively applied each of the right and left sibling sublists, unless one or both of the sublists consists of a single count. A single-count sublist is a leaf of the code tree. At the third level, the partition step can apply to one, two, three, or four sublists, depending on the outcome of the previous level. At each recursive step, at least one new level of a binary tree is created. If a recursion leaves a two-element sublist, then the next recursion assigns each count as a one-element sublist, which creates two leaves of a common parent node. The partitioning step is recursively applied to each subset that contains more than one symbol. The recursion stops when each remaining sublist contains exactly one count value.

### 3.4.3 Shannon-Fano Code Tree Construction

**Example 1** Consider the following frequency counts totaling 165. The symbols of alphabet A, \{a, b, c, d, e, f, g\}, have the following respective frequency counts:

\[\text{Root} = \{50, 30, 30, 20, 20, 10, 5\}.\]

**Figure 3.2: Code Tree for Shannon-Fano Example**

Partition the list between the two count values of 30, such that the sum on the left is: 50 + 30 = 80. The reader can verify that the sum on the right is 85 and the absolute value of the difference is 5. We now have two magnitude-ordered lists (nodes X1 and X2), children of the root:

\[X_1 = \{50, 30\}, X_2 = \{30, 20, 20, 10, 5\}.\]

The left side is node X2 of Figure 3.2, and is a two-element sublist. Each symbol of a two-symbol sublist becomes a leaf. In this case, leaves a (count 50) and b (count 30) at the second level below the tree root. The right side needs further subdivision. Following, is the right sublist's best partition point represented as the two resulting sublists, X3 and X4, as child nodes of node X2:

\[X_3 = \{30, 20\}, X_4 = \{20, 10, 5\}.\]

The left side is the two-element sublist, so count values 30 and 20 become third level leaves with respective symbols c and d. The right side (X4) only has three counts, and its resulting third level partition is:

\[e = \{20\}, X_5 = \{10, 5\}.\]

The left side, count 20 becomes another third-level leaf, e, and the right side is the two-element sublist (represented as parent node X5 of Figure 3.2) with child node value (10 and 5) respectively representing fourth-level sibling leaves for respective symbols f and g:

\[f = \{10\}, g = \{5\}.\]
The outcome of the Shannon-Fano approach is the
definition of a binary tree, see Figure 3.2, whereby
the code words are obtained from recording the
branch paths of 0s and 1s from the root to each leaf.
The branch numbers are shown in the figure for the
root node and the left sibling only. (See problems at
end of chapter.) Note that in Figure 3.2 that interior
nodes also have paths from the root. For example,
the path to node X3 is 10, and the path to node X5
is 111.

3.5 The Huffman algorithm

Both the Shannon-Fano and the Huffman
algorithms associate each symbol with a number
proportional to its expected relative frequency for
the data to be compressed. For the Huffman
algorithm it is customary (but not mandatory) to list
the symbols in frequency order. The starting point
for Huffman’s algorithm can be the set of counts,
frequencies, relative frequencies, and/or probability
estimates for each symbol to be coded. The The
scaling of the actual numbers used in the algorithm
don’t matter as long as the value associated with
each symbol is representative of its portion of the
data files to be compressed.

The beginning part of the Huffman algorithm
parallels that of the Shannon-Fano algorithm: the
values associated the symbols (eg., counts, etc.) are
sorted in magnitude order. For Huffman’s algorithm,
the sorting step is not necessary (as in Shannon).
The sorting step is a convenience in locating the two
smallest values.

The original publication ([Huf 52]) of the Huffman
algorithm is applicable to any code symbol set:
binary, ternary (three code symbol values), decimal,
etc. The Huffman algorithm pre-dates the digital
revolution that made computers commonplace.

Fifty years ago, the early teletype networks
employed relatively slow electro-mechanical devices
such as relays. The teletype networks employed a
fixed-rate code, where each symbol consumes a
constant duration of communication time to transmit
and to receive one character or symbol. Printers also
could print each character at the maximum rate for
the electro-mechanical devices. Rate control assured
the symbols would not arrive faster than the printer
could print them.1

With today’s computer systems running faster that
communications lines, and large memory capacities,
elastic buffering prevents data loss. Variable-rate
coding on the internet for compressing large amounts

of data is commonplace for data such as images and
video.

Consider alphabet \( A \) of \( M \) data symbols, and a
class of data files with some estimated set of
“typical” frequency counts. We assume the binary
code alphabet, since binary is currently the number
system of most if not all digital computers.

First we describe the steps in Huffman’s algorithm,
then show why it is optimal for the assumptions
associated with prefix codes. The Huffman code, like
the Shannon-Fano, is based on numbers related to
the symbol frequencies or the symbol probabilities.
The algorithm, as described here, is intended for a
human to perform by hand.

The Huffman algorithm produces a resulting prefix
code similar to Figure 3.2: a binary code tree with
one leaf per symbol, such that the code words for
each symbol is the path along the branches from the
root node to each symbol’s respective leaf node.

The initial condition for the algorithm is also the
same as for the Shannon-Fano algorithm: a list of
numbers sorted in magnitude order, the largest value
on the left or top of the list. The algorithm proceeds
with a Recursion Loop until the list is reduced to a
single element, and a binary code tree has been built
that has the same properties of a prefix code found
in Figure 3.2.

3.5.1 The Huffman Code Recursion

As with the Shannon-Fano code, we begin with a
single List of \( M \) symbol counts, one count \( \text{Ct}(a_m) \)
per symbol \( a_m \). Although other views are possible,
we describe the Huffman method as having a	hree-step recursion:

Given: an \( M \) symbol alphabet with each symbol
associated with its count, one symbol per node,
nodes sorted according to symbol count magnitude in
List[M], and recursion variable \( i \) initialized to \( M \). The
following recursion builds a code tree from the
bottom up.

1. Assign the two List[\( i \)] nodes with the two
smallest counts as siblings to a newly created
parent node.

2. Associate the new parent node with the sum of
the counts of its siblings.

3. Create NewList[\( i \)] from List[\( i \)] by removing
the two siblings of Step 1 and add the parent node
of Step 2 with its count. NewList[\( i \)] has one less
element than List[\( i \)].

The author's job programming part of a process control
application for Westinghouse in the early 1960s taught him how
slow the teletype printers really were, since that was our output
device for printing our program listing.

\footnote{A possible exception is that although this figure’s Shannon-
Fano prefix code is optimal, not all Shannon-Fano prefix codes are optimal}
4. **Add and Label Branches** from the parent node created this recursion to each of the two sibling nodes that were identified in Step 1.

5. **Re-sort NewList[i]** to magnitude order.

6. **IF** NewList[i] has at least two nodes to be combined, **THEN** re-execute this recursion with List[i-1] = NewList[i] and recursion variable i = i-1; **ELSE** having just produced the root node, exit the Huffman algorithm with the tree structure complete with its root node.

In summary, this “bottom-up” algorithm produces a new interior node of the code tree at each recursion step. The recursion steps are repeated until the List reduces to the single element that becomes the root of the binary code tree.

**Example 2 (Huffman code)** The Huffman code algorithm is applied to the Shannon-Fano code problem of Figure 3.2.

**Initial Condition for First trip through the Recursion Loop:**

\[
\text{List}[7] = \{50, 30, 30, 20, 20, 10, 5\}.
\]

For this first trip through the recursion loop, we break the operation into smaller steps than the second and other trips through the recursion loop.

1. Perform the **Combine and Remove** Step: the two smallest counts, (10, 5), are combined and the parent \{15\} is added to the List while nodes \{10, 5\} are removed from the List.

2. Perform the **Add and Label Branches** Step: **Add** a branch from node 15 to each sibling 10 and 5, and **Label** one sibling branch 0 and the other sibling branch 1.

3. Perform the **Resort List** Step: **Remove** counts 10 and 5 from the List[7], and insert count 15 into the new 6-element NewList[7] in its proper sort-order3 position.


**Observation 1** Each recursion is designed to reduce the size of the remaining List[i] by one element.

**Initial Condition for Second Trip through the Recursion Loop** is List[6] = \{50, 30, 30, 20, 20, 15\}. **Note:** at this point, we have created the node identified as X5 on the Shannon-Fano code tree of Figure 3.2.

The Second Trip through the recursion step begins by combining the count values of the two smallest nodes 20 and 15 of List[6] to create new parent node 35. **Note:** at this point, we have created node 35 that corresponds to X4 on the Shannon-Fano code tree of Figure 3.2.

The newly created NewList[6] is: \{50, 30, 30, 30, 35\}. From new parent node 35, the branch to child node 20 has label 0, and from parent node 35 the branch label to child node 15 is 1. Sorting NewList[6] we create List[5]: \{50, 35, 30, 30\}. The Second Trip through the recursion loop has produced five-element List[5] from six-element List[6].

**Observation 2** After two recursions, we reduced the 7-symbol code design problem to a 5-symbol problem. List[5] re-enters the recursion step.

The recursion for the Third Trip begins with re-sorted List[5]: \{50, 35, 30, 30, 20\}. Nodes 30 and 20 create new node 50. From new node 50, the branch to node 20 is labeled 1, and the branch to node 30 is labeled 0. NewList[5] is \{50, 35, 30, 30\}, which sorts to the final

\[
\text{List}[4] = \{50, 50, 35, 30\},
\]

which is used to re-enter the recursion loop.

**Note:** at this point, we have created the node identified as X3 on the Shannon-Fano code tree of Figure 3.2.

**Observation 3** The first three nodes created by the Huffman algorithm are the last three nodes generated by the Shannon-Fano algorithm for this same problem.

The Fourth trip through the recursion creates a code tree node the differs from the nodes in the Shannon-Fano tree of Figure 3.2. We re-enter the recursion with List[4]: \{50, 35, 30, 30\}. From new interior node 65, branch 0 goes to leaf node 30 (symbol b), and branch 1 goes to interior node 35. After removing nodes 35 and 30 and adding node 65 to NewList[4], the result after sorting NewList[4] is List[3]: \{65, 50, 50\}.

To continue, the Fifth instance of the recursion begins with List[3]: \{65, 50, 50\}. The two nodes of count 50 are combined into the new parent node 100, of count 100 to form NewList[3] = \{65, 100\}.

The branch from new interior node 100 to leaf node 50 has label 0, and the branch from node 100 to interior tree node 50 has label 1. The new List[2], After sorting NewList[3], the recursion loop is re-entered with List[2] = \{100, 65\}.

The Sixth (and last) recursion step combines counts 100 and 65 of List[2] into the root node 165.
NewList[2] = {165}. The path from root 165 to interior node 100 is 0, and to interior node 65 is 1. Since NewList[2] becomes List[1] with only one element, and the condition for terminating the recursion is met.

### 3.5.2 Technical Basis for the Huffman Algorithm

We have already covered the notion of a code tree for prefix codes. Based on the properties of code trees for prefix codes, Huffman had an important observation. With a binary code alphabet, Huffman observed:

**Observation 4 (Longest codewords are pairs)**

The longest code word has the most bits, and its leaf is furthest from the root node of the Huffman tree. Since the symbol with the smallest probability gets the longest code word, the key idea of the Huffman algorithm is: the longest code words come in pairs.

Prefix code trees for the binary alphabet have two siblings per parent node. A parent node of a binary tree has exactly two sibling nodes (not one node and not three nodes). Suppose, for other than the unary alphabet of one symbol) one of the longest code words is assigned to a symbol without a sibling to share the parent node. This condition immediately leads to a suboptimal code.

**Example 3 (Assumptions)** Assume a prefix code has respective code words \{0, 10, 110, 1110\}, with the following properties:

1. the longest code word length is 4,
2. symbol d is assigned code word 1110, and
3. no symbol has been assigned to sibling code word 1111.

Let prefix code word set \(CdX = \{0, 10, 110, 1110\}\). \(CdX\) satisfies the properties of assumptions defined by Example 3, since no code word belonging to \(CdX\) is a prefix of 0, 10, 110, or 1110. The longest prefixes of \(CdX\) members \{0, 10, 110, 1110\} are respectively \{null, 1, 11, and 111\} and none of these prefixes are also code words belonging to \{0, 10, 110, 1110\}.

The code word set \(CdX\) also satisfies items 1, 2, and 3 of Example 3. Let symbols \(\{a, b, c, d\}\) respectively be assigned code words \{0, 10, 110, 1110\}.

**Observation 5 (Bottom-up)** A symbol assigned the longest code word must have a sibling symbol assigned the same length code word.

The bottom-up nature is based on forming sibling pairs that most deserve to be siblings with a common parent node. These siblings must be closest to each other in probability value, and the symbol with the smallest probability deserves a sibling except when more interested in a property of the code other than optimal coding efficiency.

By focusing on combining sets representing symbols and subsets into larger sets, the Huffman algorithm's recursion step is significantly different from the Shannon-Fano recursion. Instead, Shannon-Fano partitions larger sets into successively smaller subsets. The difference between Shannon-Fano and Huffman is basically the difference between a top-down algorithm (Shannon-Fano) and a bottom-up algorithm (Huffman):

- The Shannon-Fano recursion begins with all symbols belonging to the same set, recursively chooses and subdivides one of the multiple-symbol sets into two sets. The Shannon-Fano algorithm terminates when each symbol is its own single-element subset.
- The Huffman recursion begins with each symbol comprising a single-element set, that consists of its count. These single-element sets become the leaves of the code tree. The recursion step chooses two individual sets, and combines them into a single set. The count of the new set is the sum of the counts of the sets that comprise it. If symbol a of count 13 is combined with symbol b of count 20, then the new interior node has count 33.
- The Huffman algorithm begins with leaves associated with values, and builds new interior nodes at each step that are associated with larger values.
- The Huffman recursion terminates there are no more leaves, or node sets to combine. The remaining set is the root node of the binary tree. The value associated with the root node is the total of all the values associated with the symbols of the alphabet.

Thus, in terms of sets and subsets: the Huffman algorithm ends where the Shannon-Fano algorithm begins, whereas the Shannon-Fano ends where the Huffman algorithm begins.

Some published algorithms show the second phase, top-down, assigning the path branch numbers of 0 and 1. However, branch labels can also be assigned on the way up, as we have shown here. The second phase determines the code words in prefix order: the branch labels from the root become the first code word bit.

\[^4\]Such cases do exist, such as whenever the code is subjected to a noisy environment that can alter the values of 0 and 1.
Comparison of Huffman versus Shannon-Fano

In Example 2, the example problem that demonstrated the Shannon-Fano code was re-worked using the Huffman algorithm. Let us compare the results.

Observation 6 (Same performance) In both cases, Shannon-Fano and Huffman, the code word length vector is 2 for a and b, 3 for c, d, and e, and 4 for f and g.

For the given code word lengths in bits, and the symbol frequency counts, we can calculate the total code length $CL$ for the Huffman code using the inner product of the frequency count vector: 

\[
\{50, 30, 30, 20, 20, 10, 5\}
\]

and the code word length vector:

\[
\{2, 2, 3, 3, 3, 4, 4\}
\]

Since the two code trees have the same length vector, the code length $CL = 430$ bits. The per-symbol code length is $\frac{CL}{N} = \frac{430}{165} = 2.606 \text{ bps}$.

This calculation is the same result as for the Shannon-Fano version.

The code length vector used here is the same as for Shannon-Fano, as are the counts. This example illustrates the following point: the Huffman algorithm generates an optimal code tree for a given distribution; however, there may be other optimal code trees. Also note: The Shannon-Fano code of Figure 3.2 cannot be generated by the Huffman algorithm.

Merging smallest counts is sufficient but not necessary

Observation 7 (Not Necessary) A counter-example shows that combining the two smallest counts at each step is a sufficient condition for prefix code design, but not a necessary condition.

Counter-example 1 (Not necessary) Note that List[A] of in the creation of the Huffman code, \{50, 30, 50, 35\}, corresponds to the nodes of Figure 3.2 that are two branches from the root node: \{50, 30, 50, 35\}.

These nodes are respectively labeled \{a, b, X3 and X4\}. The Shannon-Fano code resulting from this tree yields the same code length as the tree of Figure 3.3. Therefore, combining the two smallest counts is a sufficient condition for optimality. Figure 3.2 is a counter-example to the notion that combining the two smallest counts is a necessary condition for optimality.

The tree branches connecting the original M numbers from first list to successive list forms the basis for a binary code tree. The root is the final list with the single number that is the sum of the M numbers of the initial list. The M tree leaves are the numbers in the initial list, one number per symbol. Each of the M symbols has its own unique path to the root node. What remains is to label the branch pairs with respective labels 0 and 1, in going from the root through nodes representing the intermediate numbers to the M leaves.

Clearly the tree resulting from the Huffman algorithm can have more than one code tree associated with the original numbers. Simply exchanging the 0 and 1 labels on the sibling branches, in various combinations, will change the values of the symbols' code words. A less common way for Huffman trees of different shapes occurs when three or parent nodes of the code tree being created share the same count number as the smallest count. There are 3 possible combinations for the next super-symbol.

```
00 a 50
10 b 30
010 c 20
011 d 20
110 e 20
1110 f 10
1111 g 5
```

Figure 3.3: Huffman Code Tree for Shannon-Fano Example

The resulting Huffman code tree differs from the outcome of the Shannon-Fano code for the same problem. The Shannon-Fano code tree is shown in Figure 3.2, for comparison. Although the shape of the tree is different, the code word lengths for both Figures 3.2 and 3.3 are still respectively \{2, 2, 3, 3, 3, 4, 4\}. For Example 1, the compression performance is the same.

3.5.3 Why is the Shannon-Fano method sub-optimal?

Consider the way sibling pairs are formed by the Shannon-Fano process. See Example 1, beginning with all the counts in a single list. The lists are subdivide into sublists, but the ability for independent sublists to combine with other sublists is limited. In contrast, in the example Huffman code, leaf symbol b was able to be a sibling under interior node 65 and have interior node 35 as its sibling.

See Figure 3.3, and note that the dotted lines associated with lines of the tree that cross each other are not possible with the Shannon-Fano code. Note
that the dotted lines in the tree are related to nodes
touched by the two crossed lines. Not possible with
the Shannon-Fano algorithm. All interior node 65.
Notice also nowhere are the relative frequencies of
the interior nodes calculated. ‘Weights’ of the
interior nodes

3.6 Example Designs of Huffman Codes

Example 4 A six-symbol Huffman code tree

Let symbol alphabet \( \mathcal{A} = \{a, b, c, d, e, f\} \), of
respective distribution:

\[ D_0 = \{ \frac{4}{5}, \frac{5}{27}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9} \} \].

The numerators can be viewed as frequencies after
noting that 54 is a common denominator:

\[ \{ \frac{24}{54}, \frac{2}{54}, \frac{6}{54}, \frac{6}{54}, \frac{6}{54}, \frac{1}{54} \} \].

Figure 3.4 is the result of applying the Huffman
Algorithm to build the code tree for six-symbol
problem.

![Huffman Code Tree for Distribution D0](image)

Steps in Code Tree Construction

The tree of Figure 3.4 is determined as follows.
The symbols are ordered according to decreasing
frequency count.\(^5\) In the tree shown, we have omitted
the six symbol identifiers. The leaves of the tree
respond to each of the numbers \{24, 9, 8, 6, 6, 1\}.  

Note: The node circles enclose the associated
count value, eg, count 7, is associated with the first
parent (interior) node appears in parentheses just
below the circle. At each next step we combine the
smallest two count numbers into the next node.

1. For the first step, we have a choice; we can
combine the smallest valued leaf (frequency 1)
with either of the two leaves of frequency count
6. The closest leaf is chosen.

2. For the second step, we are now dealing with
count set \{24, 9, 8, 6, 7\}. Again we combine the
two smallest frequencies, in this case leaf 6 and
interior node (1) of count 7, forming interior
node (2) whose count is 6 + 7 = 13. The new set of
counts is \{24, 9, 8, 13\}.

3. In the third step, node (3) is created by
combining leaves 8 and 9, with interior node (3)
having count 9 + 8 = 17. The new counts for step
4 are \{24, 17, 13\}.

4. In the fourth step, node (4) is created by
combining interior nodes 17 and 13, so node (4)
has count 17 + 13 = 30. The new counts are \{24,
30\}.

5. In the final step, root node (5) is created by
combining the interior nodes of counts 24 and
30, so the root node has count 24 + 30 = 54. The
tree is complete.

Steps in Code word Assignment

1. First we must ensure that the two paths from
each interior node each of its two sibling nodes
are distinguished by the first sibling path having
label 0 and the second sibling path having label
1. In Figure 3.4, the upper of the two paths
from the interior nodes carry label 0, and the
lower sibling path carries label 1.

2. The code word for each leaf symbol is the
sequence of sibling path labels from the root
node to the leaf node. In other words, the code
word is the concatenation of path segment labels
from the root to the respective leaf node.

In the case of symbol a, whose count is 24, the
code word is 0 because for a there is only one path
segment from the root to the leaf, and its label is 0.
The next leaf, symbol b of count 9, has three path
segments from root to leaf, and its code word is 100.
Leaf node c of count 8 has path segments 101. Note
that leaf c of count 8 is a sibling of b, so their code
words differ in only the last position. Two other leaf
nodes are siblings; e of count 6 and f of count 1.
Their code words are respectively 1110 and 1111.
Again, note the difference in the last bit position,
because they are siblings.
Code performance of six-symbol example

For the resulting codetree, and the frequencies employed to design it, the average code length is calculated as follows:
\[
\frac{1}{54} \times (54 + 30 + 17 + 13 + 7) = \frac{121}{54} = 2.241 \text{ bps.}
\]

This calculation assumes that there are 54 symbols in the data string, and Figure 3.4 represents the calculation for the Huffman code. The summand 54 of the left factor of the equation comes from the fact that each code word has a first bit, as noted in the circled 54 at the root of the Huffman tree. The number 30 comes from node 30 (code words that begin with a 1). These 30 code words (of the 54) each have a second bit to be added. The numbers 17 and 13 are due to the third bit in each of the code words respectively beginning with 10 and 11. The summand 7 comes from node 7 because the 7 code words beginning with 111 have a fourth bit to be counted.

So for 54 symbols, we divide the total of 121 (= 54+30+17+13+7) by a total count of 54 and get 2.241 bits per symbol. Returning to the Naive Approach example, we see there was 2.389 bits per symbol, versus the Huffman method of 2.241, or a savings of (2.389 - 2.241) = 0.148 bps. No prefix code can do better than the Huffman bound of 2.241 bps.

Two more tree-construction examples

Example 5 A nine-symbol Huffman code tree.

Let the nine symbol alphabet have the following distribution:
\[D_9 = \{12, \frac{9}{48}, \frac{8}{48}, \frac{6}{48}, \frac{4}{48}, \frac{3}{48}, \frac{2}{48}, \frac{2}{48}, \frac{2}{48}\}\]

The leaf labels on the tree are the numerators above common denominator 48.

For Figure 3.5 the average (per-symbol) code length is:
\[
\frac{1}{48} \times (48 + 28 + 20 + 16 + 11 + 8 + 5 + 4) = \frac{140}{48} = 2.917.
\]

Note: for the numbers feeding the Huffman algorithm we use probabilities times 48. We can use the numerators as the symbol “counts” of a typical data string.

Example 6 A seven-symbol Huffman code tree.

Let the seven symbol alphabet have the following distribution: \[D_7 = \{\frac{12}{48}, \frac{9}{48}, \frac{8}{48}, \frac{6}{48}, \frac{4}{48}, \frac{3}{48}, \frac{2}{48}\}\].

The leaf labels on the tree are the numerators above common denominator 48.

The average code length for Example 6 is
\[
\frac{1}{16} \times (16 + 8 + 8 + 4 + 4 + 2) = \frac{42}{16} = 2.625.
\]

Consider Figure 3.6: in all cases the number on each pair of branches to each of the two siblings is the same, i.e., 8 and 8, 4 and 4, and 2 and 2, and 1 and 1. With identical counts, the probability for each branch of the pair is the same: each branch has probability \(\frac{1}{2}\). No loss in coding efficiency occurs if each of the two Huffman code tree branches from a parent node has equally probable siblings. With equally probable siblings at each parent node, the number of binary digits required to identify a given event \(ct(i)\) exactly equals the self-information \(il(i)\).

3.6.1 Adaptive Huffman Coding

Adaptive Huffman codes were first discovered by Newton Faller, then an employee of IBM do Brasil in Rio. The discovery became his Master’s thesis at the Federal University of Rio de Janeiro, and was later published in [Fal73]. The code “learns” the symbol probabilities by dynamically using the symbol counts to adjust the code tree. The decoder must use the
same initial counts and count incrementation
algorithm used by the encoder so the
encoder-decoder pair “maintain the same tree”. A
more accessible reference to the adaptive Huffman
code was done by Gallager [Gal77]. Gallager’s
treatment is more extensive than Faller’s, and
provides additional insights through the definition of
the sibling property.

3.7 Summary

A prefix code corresponds to a code tree. Each
prefix code word is associated with a leaf as
summarized below. The Shannon-Fano code tree
construction algorithm creates sub-optimal codes.

In contrast, the Huffman code tree construction
algorithm (below) always generates an optimal prefix
code in two phases:

1) the generation of a tree from its leaves
culminating in a single root node by combining each
of the least probable symbols or interior nodes into a
new parent node at the next higher tree level. Each
such next level has one less symbol or node to deal
with

2) the assignment of a sequence of 0s and 1s to
each branch of an interior tree node beginning with
the root node. For each parent (interior) node: value
0 is assigned to the left sibling’s branch and value 1
is assigned to the right sibling’s branch. The
respective code word assigned to each symbol is the
concatenation of the branch labels along the path
from the root to the respective leaf symbol.

Given the probability value of a each symbol, and
knowing the number of bits in each symbol’s code
word, permits a calculation of the average code
length per symbol (in bps) for the code.

Problems

1. For the Shannon-Fano code of Figure 3.2, what
are the code words respectively assigned to
the three leaves of depth three?

2. The frequencies for alphabet \( A_1 \) are \{420, 100,
60, 60, 23, 20, 11, 6\}.

(a) Determine two possible Huffman tree
shapes for this set of frequencies, each
based on a choice of which of these nodes to
combine. Show your work and the choices.

(b) Do any of the code word lengths of any
symbols change when choosing one tree
instead of the other? (Yes or No, give
reason.)

(c) Calculate the per-symbol code length in
bps for this set of frequencies.

3. Frequencies for alphabet \( A_2 \) are \{10, 2, 2, 1, 1\}.

(a) Create the Huffman tree for the code based
on these frequencies.

(b) Calculate the expected per-symbol code
length in bits per symbol.

4. The prefix property reads digits from left to
right. We create a “postfix” code from a prefix
code by reversing (or “mirror imaging”) the
digit sequence for each prefix code word. Given
a set of prefix code words \{0, 10, 110, 111
code words 10 and 110 respectively become 01
and 011, while 0 and 111 are unchanged.
Suppose we code a data string with the postfix
version of the above code.

(a) Can we uniquely convert the coded data
back to the original data, knowing only the
original prefix code, and that the code
word?

(b) Suppose \( a, b, c \) respectively have code
words 0, 10, and 11; whose mirror image is
0, 01, 11. Encoding \( a b c b b a \) we get
0011010100. Describe a simple method,
knowing the prefix code that was mirrored,
and given a reasonably large memory
capacity relative to the code string size,
that “works every time” for recovering the
original sequence \( a b c b b a \) one symbol\(^6\)
at a time.

\(^6\) Not necessarily the first
Bibliography

