Prof. Sergio A. Alvarez
Fulton Hall 410–B
Computer Science Department
Boston College
Chestnut Hill, MA 02467 USA

http://www.cs.bc.edu/∼alvarez/
alvarez@cs.bc.edu
voice: (617) 552-4333
fax: (617) 552-6790

CS383, Algorithms
Notes on Lossless Data Compression and Huffman Coding

It is convenient to minimize the space needed to store data. Larger files will take longer to transfer over a data link, and will more quickly fill up disk quotas. Data compression techniques such as those used in common compression utilities allow reducing file sizes by exploiting redundancies in the data contained in them. *Lossless* coding techniques do this without compromising any information stored in the file. *Lossy* techniques may achieve even greater compression, but only by providing an approximate reconstruction of the original data. We discuss lossless binary coding, Shannon's lower bound on the code length in terms of entropy, and the Huffman coding algorithm, a greedy approach to generating optimal prefix codes for lossless data compression.

1 Binary Codes

We encode data as strings of bits (binary digits). For example, suppose that a particular file contains text written only the three characters A, B, C. We can encode each character using 2 bits as follows:

$$A = 00,\ B = 01,\ C = 10$$

Doing this already provides a savings over, say, using an ASCII or Unicode representation, which would spend 8 bits or more per character.

More generally, we can encode an alphabet containing n different symbols by using $\lceil \log_2 n \rceil$ bits per symbol.

1.1 Variable length codes

However, we can sometimes do even better by using different code lengths for different symbols of the alphabet. Suppose that A's are significantly more common than B's and C's. We can quantify this in terms of the fraction of characters in the file that match each of the three possible candidates. For example, we might know that these probabilities are as follows:

$$P(A) = 0.5,\ P(B) = 0.25,\ P(C) = 0.25$$

Since A is more common, we could save space by encoding A using a single bit:

$$A = 0,\ B = 01,\ C = 10$$

The expected number of bits per character for this new code will be the weighted sum $1 * P(A) + 2 * (P(B) + P(C)) = 0.5 + 1 = 1.5$. Indeed, we save 0.5 bits per character (25%) in this way.

1.2 Prefix codes

You may have noticed that there's a difficulty with the particular encoding used in the preceding example, since some bit sequences will be ambiguous, that is, they will match more than one possible character sequence. For example, does the sequence 010 correspond to AC or BA? One way of preventing this problem is to require that the binary code of each character cannot be a prefix of any other. Any encoding that satisfies this property is known as a *prefix code*. We will discuss Huffman's algorithm for finding optimal prefix codes below. First, however, we will present a fundamental lower bound on the code length of any possible coding scheme.

2 Entropy; Shannon's coding theorem

The absolute lower limit on the number of bits needed to encode a probabilistic data source was discovered by Claude Shannon, a mathematician and electrical engineer working at Bell Labs in the mid 20th century. Shannon found that the concept of *entropy* from statistical mechanics plays a key role. Entropy measures the degree of disorder in a mechanical system, which is related to the number of possible states of the system.

2.1 Entropy of a data source

Consider an alphabet containing n symbols. Assume that strings over this alphabet are generated probabilistically, with the i-th symbol having probability p_i. Then the entropy of the generation process (in bits) is given by the following expression:

$$H = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i}$$

The term $\log_2 \frac{1}{p_i}$ is the number of bits needed to encode equiprobable outcomes that occur with probability p_i each. For example, the value $p_i = 1/4$ corresponds to $\frac{1}{p_i} = 4$ equally likely outcomes, which would require $\log_2 4 = 2$ bits to encode. The overall expression for the entropy is a weighted sum of these bit counts, each weighted by the probability of the corresponding symbol of the alphabet.

2.2 Shannon's lossless coding theorem

Claude Shannon obtained a fundamental result that establishes a tight lower bound on the average code length that is needed to encode a data source without loss of information. Shannon's theorem states, in essence, that any lossless compression technique must use at least as many bits per symbol, on average, as the entropy of the data source. Furthermore, the theorem asserts the existence of codes that come arbitrarily close to achieving this limit (in other words, given any positive value ϵ, there will exist a code that uses at most $H + \epsilon$ bits per symbol on average to code data from a source with entropy H).

Example. Consider the example discussed above in section 1.1, for which the symbol probabilities are as follows:

$$P(A) = 0.5,\ P(B) = 0.25,\ P(C) = 0.25$$

The entropy of this data source is

$$H = 0.5 \log_2 2 + 0.25 \log_2 4 + 0.25 \log_2 4 = 1.5 \text{ bits per symbol}$$

Thus, any lossless compression scheme will use at least 1.5 bits to encode each symbol from this source, on average. The variable length code that we found above actually achieves this limit. By Shannon's theorem, we know that this is the best possible result, and that no further compression is possible without some loss of information.

3 Huffman coding algorithm

We now discuss one of the best known algorithms for lossless data compression. As mentioned above, it is desirable for a code to have the prefix-free property: for any two symbols, the code of one symbol should not be a prefix of the code of the other symbol. Huffman's algorithm is a greedy approach to generating optimal prefix-free binary codes. Optimality refers to the property that no other prefix code uses fewer bits per symbol, on average, than the code constructed by Huffman's algorithm.

Huffman's algorithm is based on the idea that a variable length code should use the shortest code words for the most likely symbols and the longest code words for the least likely symbols. In this way, the *average* code length will be reduced. The algorithm assigns code words to symbols by constructing a binary coding tree. Each symbol of the alphabet is a leaf of the coding tree. The code of a given symbol corresponds to the unique path from the root to that leaf, with 0 or 1 added to the code for each edge along the path depending on wehether the left or right child of a given node occurs next along the path.

3.1 Huffman pseudocode

Huffman's algorithm constructs a binary coding tree in a greedy fashion, starting with the leaves and repeatedly merging the two nodes with the smallest probabilities. A priority queue is used as the main data structure to store the nodes. The pseudocode appears below.

Algorithm 1: Huffman Coding
Input: Array $f[1...n]$ of numerical frequencies or probabilities.
Output: Binary coding tree with n leaves that has minimum expected code length for f.

```
HUFFMAN(f[1...n])
(1)   T = empty binary tree
(2)   Q = priority queue of pairs (i, f[i]), i = 1...n, with f as comparison key
(3)   foreach k = 1...n − 1
(4)     i = EXTRACTMIN(Q)
(5)     j = EXTRACTMIN(Q)
(6)     f[n + k] = f[i] + f[j]
(7)     INSERTNODE(T, n + k) with children i, j
(8)     INSERTREAR(Q, (n + k, f[n + k]))
(9)   return T
```

Example. Consider the text string "a basket of bananas and a large train and a fantastic anaconda as a matter of fact". Here are the characters that appear in this string, with their frequencies:

```
A:20 B:2 C:3 D:3 E:3 F:4 G:1 I:2 K:1
L:1 M:1 N:8 O:3 R:3 S:4 T:7 blank:16
```

With this string as input, the Huffman coding algorithm will first merge two characters that appear singly in the string. There are more than two such characters, so I will assume that the tie is broken by proceeding in alphabetical order: G is merged with K, creating a new node labeled $GK : 2$ that replaces the G and K nodes in the priority queue and that will have G and K as children in the binary coding tree. L and M are likewise merged. At this point, the available symbols and their frequencies in the priority queue are as follows:

```
B:2 GK:2 I:2 LM:2 C:3 D:3 E:3 O:3 R:3
F:4 S:4 T:7 N:8 blank:16 A:20
```

B and GK will be merged now since they have the lowest frequencies (again, ties are broken alphabetically), and then I and LM, C and D, and E and O. The priority queue now contains the following nodes:

```
R:3 BGK:4 F:4 S:4 ILM:4 CD:6 EO:6
T:7 N:8 blank:16 A:20
```

In the next few rounds, R is merged with BGK, F with S, and ILM with CD.

```
EO:6 RBGK:7 T:7 FS:8 N:8 ILMCD:10
blank:16 A:20
```

After more merging:

```
EORBGK:13 TFS:15 blank:16 NILMCD:18
A:20
```

and

```
A:20 EORBGKTFS:28 blankNILMCD:34
```

and

```
AEORBGKTFS:48 blankNILMCD:34
```

In the final round, the remaining two nodes are merged, becoming children of the root node of the binary coding tree. The levels of the resulting tree are as follows:

```
all:82

AEORBGKTFS:48 blankNILMCD:34

A:20 EORBGKTFS:28 blank:16 NILMCD:18

EORBGK:13 TFS:15 N:8 ILMCD:10
```

```
EO:6 RBGK:7 T:7 FS:8 ILM:4 CD:6

E:3 O:3 R:3 BGK:4 F:4 S:4 I:2 LM:2 C:3 D:3

B:2 GK:2 L:1 M:1

G:1 K:1
```

Codes are assigned to leaves by associating the bit 0 with the left child and 1 with the right child of each node along the path from the root to the leaf. For example, the code for T will be 0110, since the path from the root to T first goes left, then right, then right, and finally left. Leaves at depth d will have codes containing d bits. The total compressed (encoded) length will be the sum of the lengths of all of these leaf codes, each multiplied by the number of times that the corresponding character appears in the text string. In this example, the total length will be

$$2 * (20 + 16) + 3 * (8) + 4 * (7) + 5 * (25) + 6 * (4) + 7 * (2) = 287 \text{ bits}$$

How does this compare with the uncompressed length? Well, there are 17 distinct characters in the original string. Using a fixed length code, we would need 5 bits per symbol to represent the 17 character choices. Since the string has length 82, the total number of bits required would be $5 * 82 = 410$. By using the variable length code found by Huffman coding, we end up needing only $287/82 = 3.5$ bits per symbol, a savings of 30%.

3.2 Time complexity of Huffman's algorithm

As with Dijkstra's algorithm for the single source shortest paths task in graphs, the running time of Algorithm 1 depends on how the priority queue is implemented. Assuming that a heap is used, each insertRear and extractMin operation will require time $O(\log n)$, where n is the number of elements in the priority queue. Since these operations are performed a constant number of times in each iteration of the main loop, and since $O(n)$ iterations are carried out altogether, the total running time will be $O(n \log n)$. By the way, the initial construction of the heap may easily be completed in time $O(n \log n)$ also, since it suffices to perform n individual insertions.

4 Optimality of Huffman coding

We show that the prefix code generated by the Huffman coding algorithm is optimal in the sense of minimizing the expected code length among all binary prefix codes for the input alphabet.

4.1 The leaf merge operation

A key ingredient in the proof involves constructing a new tree from an existing binary coding tree T by eliminating two sibling leaves a_1 and a_2, replacing them by their parent node, labeled with the sum of the probabilities of a_1 and a_2. We will denote the new tree obtained in this way merge(T, a_1, a_2). Likewise, if A is the alphabet of symbols for T, that is, the set of leaves of T, then we define a new alphabet A', denoted merge(A, a_1, a_2), as the alphabet consisting of all symbols of A other than a_1 and a_2, together with a new symbol a that represents a_1 and a_2 combined.

Two important observations related to these leaf merging operations follow.

1. Let T be the Huffman tree for an alphabet A, and let a_1 and a_2 be the two symbols of lowest probability in A. Let T' be the Huffman tree for the reduced alphabet $A' = \text{merge}(A, a_1, a_2)$. Then $T' = \text{merge}(T, a_1, a_2)$. In other words, the Huffman tree for the merged alphabet is the merge of the Huffman tree for the original alphabet. This is true simply by the definition of the Huffman procedure.

2. The expected code length for T exceeds that of $\text{merge}(T, a_1, a_2)$ by precisely the sum p of the probabilities of the leaves a_1 and a_2. This is because in the sum that defines the expected codelength for the merged tree, the term dp, where d is the depth of the parent a of these leaves, is replaced by two terms (corresponding to the leaves) which sum to $(d+1)p$.

4.2 Proof of optimality of Huffman coding

With the above comments in mind, we now give the formal optimality proof. We show by induction in the size, n, of the alphabet, A, that the Huffman coding algorithm returns a binary prefix code of lowest expected code length among all prefix codes for the input alphabet.

- (Basis) If $n = 2$, the Huffman algorithm finds the optimal prefix code, which assigns 0 to one symbol of the alphabet and 1 to the other.

- (Induction Hypothesis) For some $n \geq 2$, Huffman coding returns an optimal prefix code for any input alphabet containing n symbols.

- (Inductive Step) Assume that the Induction Hypothesis (IH) holds for some value of n. Let A be an input alphabet containing $n+1$ symbols. We show that Huffman coding returns an optimal prefix code for A.

  Let a_1 and a_2 be the two symbols of smallest probability in A. Consider the merged alphabet $A' = \text{merge}(A, a_1, a_2)$ as defined above. By the IH, the Huffman tree T' for this merged alphabet A' is optimal. We also know that T' is in fact the same as the tree $\text{merge}(T, a_1, a_2)$ that results from the Huffman tree T for the original alphabet A by replacing a_1 and a_2 with their parent node. Furthermore, the expected code length L for T exceeds the expected code length L' for T' by exactly the sum p of the probabilities of a_1 and a_2.

  We claim that no binary coding tree for A has an expected code length less than $L = L' + p$.

  Let T_2 be any tree of lowest expected code length for A. Without loss of generality, a_1 and a_2 will be leaves at the deepest level of T_2, since otherwise one could swap them with shallower leaves and reduce the code length even further. Furthermore, we may assume that a_1 and a_2 are siblings in T_2. Therefore, we obtain from T_2 a coding tree T_2' for the merged alphabet A' through the merge procedure described above, replacing a_1 and a_2 by their parent labeled with the sum of their probabilities: $T_2' = \text{merge}(T_2, a_1, a_2)$. By the observation above, the expected code lengths L_2 and L_2' of T_2 and T_2' respectively satisfy $L_2 = L_2' + p$. But by the IH, T' is optimal for the alphabet A'. Therefore, $L_2' \geq L'$. It follows that $L_2 = L_2' + p \geq L' + p = L$, which shows that T is optimal as claimed.

4.3 Non-prefix codes

We have claimed above that Huffman's algorithm yields optimal (shortest length) prefix codes. Previously, we discussed the fact that Shannon's coding theorem establishes the shortest coding

length as being the entropy of the data source. Does this mean that the Huffman code achieves the Shannon limit?

Consider a binary data source with the following statistics:

$$P(A) = 127/128,\ P(B) = 1/128$$

The entropy of this source is just under 0.07 bits per symbol. However, the Huffman code for it obviously uses 1 bit per symbol and hence is clearly far from the Shannon limit.

How could one possibly encode this source using less than 1 bit per symbol? Here's a possibility. Given the probabilities above, most symbols will be A's, and we'd expect long runs of only A's in the data. In fact, one would expect a B only every 128 characters or so. We can leverage this fact by coding entire runs at once instead of requiring one code per individual symbol. The code string could be split up into segments delimited by a special "delimiter" code followed by, say, 8 bits for the binary positional encoding of the run length. To account for the possibility of particularly long runs for which the run length cannot be coded in 8 bits, an extra 9th bit could be included in each segment to signal whether or not the next 8-bit word should be added to the run length code for the current segment. In this scheme, an extra B could be encoded simply as an A-run of length 0.

For example: let 11 be the delimiter code, and allow 3 bits for the run length (for brevity in this example) and one extra bit for continuation. The string $A^5BA^{10}BBA^6BA^9$ would be encoded as 111010110011110100110000111100110011110010 11. Changing the word length back to $8 + 1$ bits, the expected code length for the probabilities given above would be approximately 11 code bits (one delimiter plus one $8 + 1$ bit word) for every 128 data bits, since that is how often a B would be expected in the data stream. This gives a code length of just under 0.09 bits per symbol, which is far better than the best prefix code and not too far from the Shannon limit.