Lecture 1 Introduction

Coded Bitstream: The binary string representing the whole coded (compressed) data.

Codeword: A binary string representing either the whole coded bitstream or one coded data symbol (you will tell the difference from the context).

Bit Rate (or bitrate): Average number of bits per original data element after compression.

Compression Ratio (CR):

$$\text{Compression Ratio (CR)} = \frac{\text{size of uncompressed data (in bits)}}{\text{size of coded bitstream (in bits)}} = \frac{|I|}{|b|}.$$

Entropy: the minimum average number of bits/symbol possible.

Lecture 2 Lossless Compression Part I

2.1 Huffman Coding

Algorithm 1 Huffman Coding

Input Alphabet $\{a_1, a_2, \ldots, a_n\}$ and symbol probabilities $\{p_1, p_2, \ldots, p_n\}$

Create a node for each symbol a_i // these nodes will be the leaves.

While (there are two or more uncombined nodes) **do**

    Select 2 uncombined nodes a and b of minimum probabilities

    Create a new node c of probabilities $p_a + p_b$, and make a and b children of c

Label the tree edges: left edges with 0, right edges with 1

The codeword of each alphabet symbol a_i (a leaf) is the binary string that labels the path from the root down to leaf a_i

Output The codewords of the alphabet symbols

2.2 Huffman Decoding

Algorithm 2 Huffman Decoding

Input A coded bitstream $b_1 b_2 \ldots b_N$ (and we have the Huffman tree).

Initialize: $i = 1$, and let node pointer ptr point at the tree root

While $(i < N)$ **do**

    If $b_i == 0$, let ptr go to left child, else go to the right child

    If ptr is pointing to a leaf node

        Append to the output the symbol corresponding to that leaf;

        Reset ptr back to the root

        $i = i + 1$

    Else $i = i + 1$

Output The reconstructed data (will be identical to the original data).

2.3 The Prefix Property

The prefix property: a coding scheme where every alphabet symbol is coded with a codewordis said to have the prefix property if no codewordis a prefix of another codeword. Huffman coding has the prefix property because every codeword is a path from the root to a leaf in the Huffman tree, and no leaf is on the path from the root to another leaf.

2.4 Block Huffman Coding

Algorithm 3 Block Huffman Coding

1: **Input** An input bitstream
2: Break the input into a series of blocks
3: Treat every block as a new (macro) symbol
4: Take all possible macro-symbols (as a new macro-alphabet)
5: Compute the probabilities of the individual macro-symbols
6: Apply Huffman coding on the macro-symbols, getting a tree and codewordsfor the macr-symbols
7: Code the original input by replacing each block by its corresponding codeword
8: **Output** The codewords of the input

Block Huffman Decoding is similar, getting back the blocks, which are appended to the output.

2.5 Run-length Encoding (RLE)

Run: Each maximal stretch of identical symbols.

RLE coding method: Replace each run by (a, L) where 'a' is the symbol repeating in the run, and L is the length of the run.

M: The number of bits that will be allocated to each length.

- Determine ahead of time the maximum L, then set $M = \log(\max L)$. Otherwise, per input, find the max L, set $M = \log(\max L)$, allocate a fixed number of bytes in the header to store the value of M so the decoder knows it.

- (Run-splitting) Set M to a reasonable, non-wasteful value even if $M < \log(\max L)$. If a run has $L > M$, then split the run into $q+1$ runs: $L = q * (2^M - 1) + r$ where $r < 2^M - 1$.

Algorithm 4 Run-length Encoding (Run-splitting)

1: **Input** An input bitstream
2: Process the input with RLE coding method
3: Fix the number of bits (say M bits) that will be allocated to each length
4: If a run has $L > M$:
5: $\quad$ $L = q * (2^M - 1) + r$ where $r < 2^M - 1$
6: $\quad$ Split the run into $q+1$ runs: the first q runs are of length $2^M - 1$ (needs M bits)
7: $\quad$ The last run of length r (needs $\leq M$ bits)
8: Apply Huffman coding on the 'a'
9: **Output** The codewords of the input

2.6 RLE Decoding

Algorithm 5 RLE Decoding

1: **Input** A coded bitstream $b_1 b_2 \ldots b_N$
2: Apply Huffman Decoding on the 'a' and next M bits
3: **Output** The reconstructed data (will be identical to the original data)

2.7 Golomb Coding

More Probable Bit (MPB): If the input is a binary stream, then count the number of of 0's (say N_0) and the number of 1's (say N_1), calculate the probability

$$p = \max\left(\frac{N_0}{N_0 + N_1}, \frac{N_1}{N_0 + N_1}\right)$$

to find the more probable bit in the input stream.

Tail Bit: The other bit which is not the MPB.

Parameter m: The optimal value of m is the nearest power of 2 to

$$p \times \frac{\ln 2}{1 - p}$$

where p is the probability of the more probable bit in the input stream.

Algorithm 6 Golomb Coding (Assume b is the MPB)

Input An input bitstream and the parameter m

Break the input into runs of the form $b^i\bar{b}$

Code each Golomb run $b^i\bar{b}$ as $\bar{b}^q by$ where

    Divide i by m integer division, we get quotient q and a remainder r

    y is the binary representation of r, using $\log m$ bits

If the last run has a tail

    tail? $= \bar{b}$

Else

    tail? $=$ **Null**

Output The final coded bitstream: MPB code_1 $\text{code}_2 \ldots \text{code}_n$ tail?

2.8 Golomb Decoding

Algorithm 7 Golomb Coding (Assume b is the MPB)

1: **Input** A coded bitstream $b_1 b_2 \ldots b_N$ and the parameter m

2: Grab first bit b as the MPB

3: Set $k = 2$ // index of the bits in the coded bitstram

4: Scan the bitstream rightward from position k looking for successive 1's, keeping a count q, and incrementing k along the way

5: When a 0 is met, read the next $\log m$ bits as y, set $k = k + \log m$

6: Set $r = b2d(y)$ // binary to decimal conversion

7: Compute $i = q \times m + r$

8: Append $b^i\bar{b}$ to the output

9: Repeat from 2 until the coded bitstreamis exhausted

10: If the last bit (tail?) is b, strip the final bit from the output

11: **Output** The reconstructed data (will be identical to the original data)

2.9 Differential Golomb Encoding

Algorithm 8 Differential Golomb Encoding (Assume b is the MPB)

Input An input bitstream $x_1 x_2 \ldots x_n$ and the parameter m

Transform x to $z = z_1 z_2 \ldots z_n$ where $z_i = x_i - x_{i-1} \forall i > 1$, and $z_1 = x_1$

Delete the alternating negatives of the tails, we get $z' = z'_1 z'_2 \ldots z'_n$

Record sign of 1^{st} tail, by one bit $s : s = 1$ if 1^{st} tail > 0, $s = 0$ otherwise

Code z' using Golomb coding, and prefix s to the output

Output The final coded bitstream: MPB code_1 $\text{code}_2 \ldots \text{code}_n$ tail?

2.10 Differential Golomb Decoding

Algorithm 9 Differential Golomb Decoding (Assume b is the MPB)

Input A coded bitstream $b_1 b_2 \ldots b_N$ and the parameter m

Set $s = 1^{\text{st}}$ bit of the coded bitstream, and remove it from the latter

Golomb-decode the rest of the coded bitstream into $z' = z'_1 z'_2 \ldots z'_n$

If $s == 0$, set the first Golumbrun's tail to -1

Alternate the signs of the remaining tails in z', getting $z = z_1 z_2 \ldots z_n$

Set $x_1 = z_1$, and for $i = 2$ to n do: $x_i = x_{i-1} + z_i$

Output The reconstructed data $x_1 x_2 \ldots x_n$ (will be identical to the original data)