Building Huffman tree with arbitrary probabilities

The example given in the previous lecture was very simple. When we actually implement the code to build the Huffman tree given symbols associated with arbitrary probabilities, depending on the data structure we choose, some data structures will perform better than others (e.g. tree vs. sorted list). For our project, Jarek is OK with us using simpler data structures.

Revisit the algorithm for building a binary tree

Build (){
    ...
    read next bit from the stream representing a tree
    if the bit represent a leaf
        return a leaf node
    else{
        leftChild = Build();
        rightChild = Build();
        return makeNode(leftChild, rightChild);
    }
}

Note that the only decision necessary is whether a node is a leaf. Hence, we can construct the tree from a bit stream, where 0 represents a leaf and 1 an internal node. We can create the bit stream from a tree as follows:

Algorithm for encoding a binary tree structure in depth-first-search manner

Encode (node){
    if it is a leaf node
        string += 0;  // 0 – leaf bit
    else{
        string += 1;  // 1 – internal node or root
        Encode( leftChild(node) );
        Encode( rightChild(node) );
    }
}

Methods for sending the encoding information and encoded data

• Method 1: Dictionary
  o Send dictionary (a=0, b=01, c=...)
  o Send encoded stream
• Method 2: Huffman tree (more efficient)
Different encoding schemes

- Entropy is the “theoretical average cost”
- Huffman code uses slightly more bits than Entropy if the probabilities are not powers of 2
- Arithmetic code
  - We think we understand how it works, but we can't explain it very well. Maybe someone else will help us out with this.
  - Problem => the interval might become very small, then we will need to send enough bit to get to the corresponding interval, dividing the space into half at a time. The method is more complex; the resulting code is close to entropy.

Steps for Compressing Polygonal Curve

1) Normalize coordinates
2) Quantize
3) Predict
4) Encode residues

Normalizing coordinates

- Map the coordinate system from Xmin-Xmax to 0-1
- \( X' = (X-Xmin) / (Xmax - Xmin) \)
• After the normalization step, the coordinates look like (0.012345, 0.2355). We would like to use integer representation instead.
• Depending on the resolution, we choose to use, for example, B bits to represent the coordinates (i.e., instead of 0 to 1.0, we now have 0 to $2^B-1$). Now the new coordinate can be obtained using

$$X''=\text{int}(X' \times 2^B)$$

• If some bits end up being truncated when we quantize the coordinates, some information will be lost. Therefore, this is a lossy step.
• In most cases, B=11 is a good enough choice. For example, how many bits would be required to tell Georgia Tech from the Atlanta airport on a map of the United States? Assuming the US is 4000 miles across, 12 bits would bring us to the mile level, 14 bits to the quarter mile.
• The maximum error caused by quantization is half of the diagonal of the chosen pixel size.

**Prediction**

• Instead of always representing each point in its real quantized coordinates, we can predict the position of a point based on the position of previous points, then use the difference of the actual position and predicted position to represent the point.

Some methods for making the guess $G_n$ based on points $V_1, ..., V_n$

- Always guess origin : not good
- $G_i = V_{i-1}$ (good if edges are short relative to curve length)
- $G_i = V_{i-1} + (V_{i-1} - V_{i-2}) = 2V_{i-1} - V_{i-2}$ (Linear interpolation)
- $G_i = V_{i-1} + (V_{i-1} - V_{i-2}) + ( (V_{i-1} - V_{i-2}) - (V_{i-2} - V_{i-3}) ) = 3V_{i-1} - 3V_{i-2} + V_{i-3}$ (Quadratic interpolation)

- Note: typically, higher order will not work better due to higher inertia. (Getting past perturbations in the curve takes longer.)
- These may have some problems if the edges are not uniform in length.
- Extrapolating predictor (all of the above)
- Interpolating predictor
  - Send $V_1$ and $V_n$, and interpolate to guess the rest
  - For example, $G_{n/2}=(V_1+V_n)/2$; $G_{n/4}=(V_1+V_{n/2})/2$; etc.
  - 4-point subdivision is also possible: Given $V_1$, $V_2$, $V_3$, and $V_4$:
    $$M_1=(V_1+V_2)/2, M_2=(V_0+V_3)/2, D=M_1-M_2, V'=M_1+D/8.$$  

- Known as Split-and-tweak

**Encoding Residues**

• The correction vectors are called residues.
• Prediction gives $G_i$ for each vertex $V_i$. The residue is $\Delta_i=V_i-G_i$.
• Each residue is a pair of integers; each integer is a symbol for a Huffman code.
• If our prediction is good, we will be able to use, with high probability, fewer bits to represent the residues (since (0,0), (1, 0), etc. will occur more frequently).
• If our prediction is bad, almost every residue coordinate is different. This would require all B bits to represent, resulting in little compression. Reducing B can reduce this effect, at the cost of quantization error.