Simplification and Compression of Two Dimensional Curves

Alexander Powell
Georgia Institute of Technology
dlex@cc.gatech.edu

Brian Whited
Georgia Institute of Technology
fender@cc.gatech.edu

Overview

We present a method for compressing 2D curves using a lossy compression scheme that involves quantization of the original data and Huffman encoding. The system consists of two distinct parts—the compressor and the decompressor. The compressor quantizes the data and stores it in an efficient manner. The decompressor reads back this compressed data and uses a number of techniques to attempt to recreate the original curve as accurately as possible.

Compression

There are multiple distinct steps to creating the final compressed file. These are normalization, quantization, guesses and residues, symbol generation, building a Huffman tree, and encoding data. Each step must be executed sequentially.

Normalization

The first step is to take the input coordinates and scale them so that they range from 0 to 1. We did this in a way that forces the resulting curve to have a consistent aspect ratio with the original. For this reason, our normalization involves only three parameters: minimum x value, minimum y value, and a scale factor.

We start by finding the minimum and maximum x and y values of all of the coordinates. The scale factor is defined as $\max(\Delta x, \Delta y)$. For each coordinate, we subtract the minimum x and y values from the position and then scale by dividing the values by the scale factor.

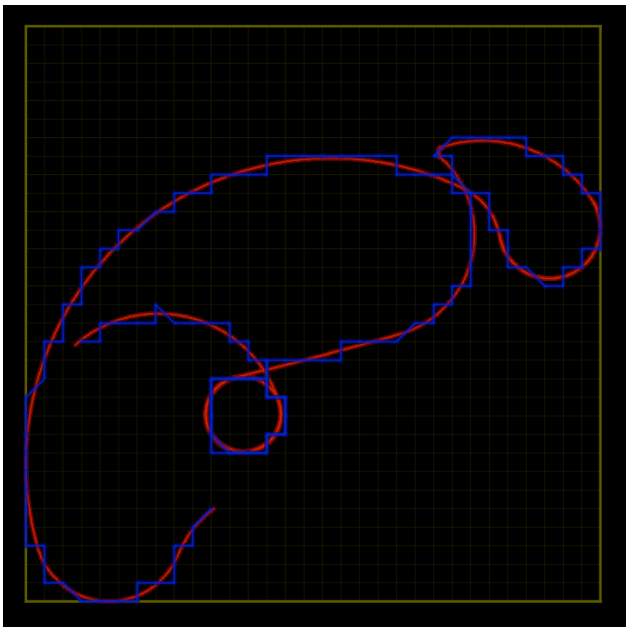


***Figure 1**: Quantization of the original curve (red) results in a discretized curve (blue). In our lossy compression algorithm, the quantization step involves the most drastic loss of data.*

Quantization

Quantization is the process of discretizing the curve by converting the original floating point coordinates into integer coordinates that fall on a regular grid.

We originally considered quantization by truncating the values and offsetting the final grid by 1/2 the grid spacing. The result was a properly quantized representation of the curve, but our sample space was reduced by one grid space in each dimension, which was unacceptable. Our solution to this was to quantize our curve onto a regular grid of size 2^b-1 rather than 2^b. This allowed us to round the values, rather than truncate them, and resulted in no shrinking of the sample space when converting back and forth from quantized numbers to real numbers.

Guesses and Residues

A guess for a point involves using previous points on the curve to predict where the next point will occur. We use this guess instead of the actual location of the next point so that we reduce the amount of storage necessary to represent that point. We calculate the guess by calculating the "difference of the differences" of the previous three points. This is shown in pseudocode form in Listing 1. From this guess, we then store only how far off the actual point is (Δx, Δy) because we assume that these delta values (also known as residues) will be small and will occur frequently in our data.

```
D[0] = P[0]; // Send first point
D[1] = P[1] - P[0];
D[2] = P[2] - P[1];
For the rest of points in P {
    d1 = P[i-2] – P[i-3];
    d2 = P[i-1] – P[i-2];

    // Calculate difference
    // of differences
    dd = d2 – d1;

    // Our guess point
    guess = P[i-1] + dd;

    // How much we're off by
    D[i] = P[i] – guess;
}
```

***Listing 1**: Prediction mechanism*

Finding Symbols

Once we have all of the residues stored, we define unique symbols to represent the values of the residues (both the x and y values separately). We count the number of symbols necessary to regenerate the curve and calculate the frequency of each (Listing 2).

Building a Huffman Tree

The Huffman tree determines a minimal binary code that is associated with each symbol. We start by building an array of Huffman nodes, or binary search tree nodes that store in a single node both the symbols themselves and the frequency of occurrence. Once we have the tree, we can determine the unique code for each symbol by recording the path taken to get to the leaf node that is the symbol. By convention, going from parent to left progeny is represented by a 0 and going from parent to right progeny is represented by a 1. For example, if you have to go left then right to get to a symbol from the root, the code for that symbol is "01".

Data Encoding

We now have all of the information we need to encode the file with the exception of a binary representation of the Huffman tree itself (called the codebook). We used a pre-order traversal of the tree, and at each node we add a 0 if it is an internal node and a 1 for a leaf. This results in a very efficient representation of the tree.

```
Array s;  // our symbols
For all Points i in P {
    If not s.contains(P[i].x)
        s.add(P[i].x);
    If not s.contains(P[i].y)
        s.add(P[i].y);
}

// Put them in order,
// smallest first
sort(s);

// Calculate frequency of each
Array freq;
For all values i in s {
    For all vectors j in D {
        If (D[j].x == s[i])
            freq[i]++;
        If (D[j].y == s[i])
            freq[I]++;
    }
}
```

***Listing 2**: Finding symbols*

Our file's header consists of 17 bytes. Stored in the header is the minimum x and y value and scale necessary to unnormalize the data (4 bytes each, 12 bytes total), the number of vertices in the curve (4 bytes), and the quantization value b (1 byte). There are more compact ways to store the header, such as removing the number of vertices and inferring that from the data, but this method was simple to implement and is close to optimal.

Next, we write the Huffman tree as a string of 0's and 1's. If the number of bits is not a multiple of 8,

we pad the end with zeros so that we are byte-aligned. The decoding function will stop when the number of 0's is more than the number of 1's at the end of a byte.

Then the symbols are written in-order (via the in-order traversal of the tree). These symbols are 4 bytes each. Optimizations can be done here, since most of the time the symbols represent values that can be contained in 1 byte, but since we cannot guarantee that without extra effort, we use 4 bytes. One simple alternative would be to send a value in the header specifying how many bits would be necessary per symbol.

Following the symbols, we send the first point. The x and y values of the first point are each 4 byte integers.

Finally, we send the encoded residues. We must pad the end of the bit string with zeros if necessary so that we are byte-aligned.

Decompressor

In addition to reading the quantized curve from the compressed file, the decompressor considers a number of methods that take quantized data and produce a curve that resembles the original curve as closely as possible. Among these is a simple curve smoothing tool, and three curve subdivision techniques. The difference between the curves (error) is measured by Hausdorff distance.

Curve Smoothing

A very simple curve-smoothing tool is provided because the quantized curve often has jagged edges that resemble stair steps. Solving this problem with subdivision can be tedious and usually preserves too many jagged edges. Our alternative solution is to form a curve from the midpoints of each of the edges, and then tack on the original and final points to maintain curve length. This approach works well to smooth out a curve and can be applied more than once. The only downside to this method is that, like many other smoothing techniques, it can result in a shrinking of certain areas when applied repeatedly. On smooth curves, we usually apply the midpoint trick 1 or 2 times before continuing with other techniques for restoring the curve (Figure 2).

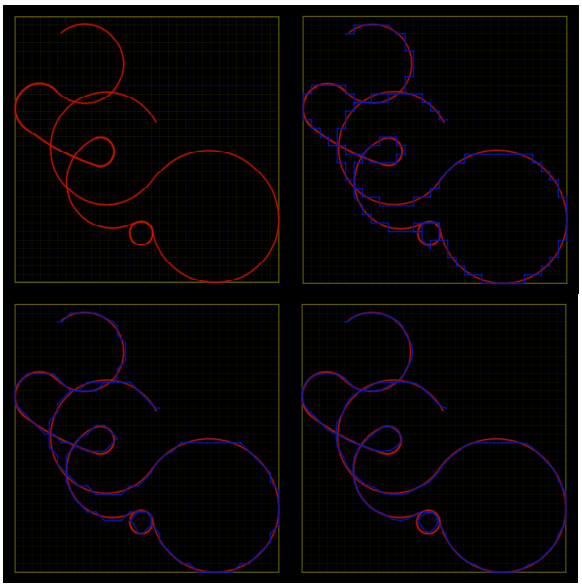


Figure 2*: The original curve (top-left) is quantized into the representation we use for compression (top-right). After one level of basic midpoint smoothing (bottom-left), the main jagged edges are toned down. After three levels of smoothing (bottom-right), the curve is almost restored, despite a compression ratio of nearly 100:1.*

Curve Subdivision

We used a couple of subdivision techniques to aid in smoothing our final curve. The first technique (split-and-tweak) was desirable because our past experience showed that it smoothed sharp edges dramatically. 4-point subdivision was also considered, but since it interpolates vertices, it maintains too much of the jagged structure of the quantized curve. Jarek's subdivision technique, which can be viewed as a combination of the principles between both 4-point and split-and-tweak subdivision, proved to be the most useful in combating the sloppiness of the discretized curve.

Hausdorff Distance Calculation

We calculated the Hausdorff distance by an approximation technique where we sample the edges of one curve in order to find the nearest points on the edges of the other. Though this is not a very efficient method for finding the Hausdorff distance, it runs fast enough on our personal computers with a reasonable number of vertices and samples per curve.

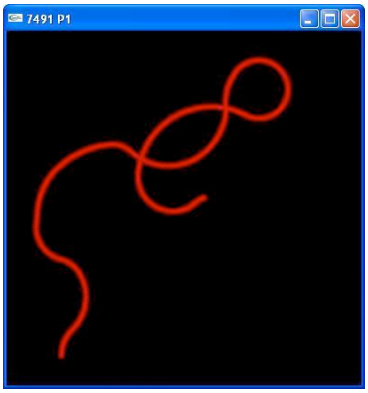
Results

All of the curves shown below were generated randomly using only a few parameters (number of vertices, average edge length, edge length variation, turning angle range, and probability of changing angle or length of the next segment.

If Elvis Were a Peanut

Number of Vertices:	1000
Average Length:	0.00951424
Log_2(length):	-6.7157
File Size:	9704 bits
Bits per Vertex:	9.704
Bits per Coordinate:	4.852
% Header:	1.40148
Entropy:	4.31051
Excess:	0.125621

Diving Miss Daisy



Number of Vertices:	1000
Average Length:	0.001
Log_2(length):	-9.96578
File Size:	6704 bits
Bits per Vertex:	6.704
Bits per Coordinate:	3.352
% Header:	2.02864
Entropy:	3.0424
Excess:	0.101762

Future Work

It would be interesting to add a few more dimensions to our curve compression technique. A possible use of a 3 to 4 dimensional curve would be legal signatures, such as those necessary to make a credit card transaction at the point-of-sale. The third dimension can be time and can be used to check whether the timing of the signature is consistent with the writer's other signatures. In addition, a fourth dimension, pressure, might be considered as a means to further assess the validity of the signature. Each of these dimensions can be compressed and stored for later analysis.