Curve Compression

Nguyen Truong*
Georgia Institute Technology
GVU Center

Howard Zhou†
Georgia Institute of Technology
GVU Center CPL

Abstract

For a given set of vertices, the problem is to compress the data representing these vertices (reduce storage cost), and at the same time, be able to decompress the data so that the original set of vertices could be recovered. We measure the efficiency of the compression scheme (Huffman coding) by comparing the average number of bits for the compressed code to the entropy of the curve. We also look at the Hausdorff error between the original curve and one recovered from decompression: this allows us to rate the correctness of the compression scheme. Furthermore, we propose a simple curve-simplification scheme to further reduce the cost of storing data.

Keywords: curve, compression, simplification, Hausdorff Distance, Huffman coding

1 Introduction

The aim of this paper is to experiment with Huffman coding, in particular, apply the theory to curve compression. Consider an alphabet consisting of n symbols, each with probability of occurrence $P(i), 1 \leq i \leq n$. Clearly if we use fixed length code, we'd need $log_2(n)$ bits to represent each symbol. For example, if our alphabet consists of a, b, c, d, e, and f, then we may assign each symbol to a 3-code: $a = 000, b = 001, c = 010, d = 011, e = 100, f = 101$. This is not very efficient, we can do better! Note that in the fixed length coding scheme, the probability of occurrence for each symbol is not taken into account. Huffman coding improves on this aspect: the scheme tries to assign symbols with high probabilities lesser bits. Intuitively speaking, this makes sense. We'd certainly want to send less bits per symbol A over a channel, given A has a high likelihood of occuring.

Huffman coding is briefly summarized below, although the curious readers are encouraged to consult a book on Information Theory. We are given an alphabet with n symbols, and a function $P(i)$ assigning probability to each symbol. It is assumed that $P(i) \geq 0$ and $Sum(P(i)) = 1$. We sort the probabilities from least to most. Suppose that $P(n)$ and $P(n-1)$ are two smallest probabilites, we then combine these nodes together, and create a new node representing the sum of the probabilities of the two old nodes. Now repeat the process. Each time we do this, we have gotten rid of one probability. In $n-1$ times, we'd have constructed a tree, call T. Each leaf on T is an original probability of a source symbol. Hence by assigning a 0 to a left edge, and a 1 to a right edge under a node, we can represent each leaf with a unique sequence of 0's and 1's.

It should be noted that we do not use this particular scheme of Huffman coding to compress curves, as this requires knowing all the distribution probabilities of symbols beforehand, and this is not ideal for our problem. Surely we can come around this by precomputing the probabilities, but this will require an extra loop, which is not desirable. Instead, we follow an "on-the-fly" approach: the Huffman tree self-adjusts its structure whenever the probability distribution changes. That is, the tree will position itself accordingly, while each symbol is being received.

Hausdorff error is one measure of similarities between objects. Given sets A and B, we define the Hausdorff error as $H(A,B) = max(max(a \in A)min(b \in B), max(b \in B)min(a \in A))$. It can be shown that $H(A,B) = 0$ iff. $A = B$. Although the Hausdorff error is not a good measure in some cases, it is sufficient for our experiments.

2 Approach

2.1 Normalization, Quantization

Quantization reduces the precision of the curve vertex coordinates from real numbers to integer numbers within a given range. During the quantization step, we lose some amount of information(precision), in return, we reduce the entropy of the curve. In our implementation, we first find the min-max box of the given point set and normalize all points so that they are in the interval of $[0,1]$. We then specify how we should quantize our curve by giving B as an external parameter. For any given B, we will quantize every point on the curve so that it lies on the interval of $[0,2^B]$. After the quantization step, we will have a curve that only has integer coordinates in $[0,2^B]$. The maximum error that can occur during the quantization step is bounded by the half diagonal of a quantization cell, hence, it is directly related to B. Therefore, with a curve having about several hundred to a thousand vertex, B is typically chosen to be 9 or 10. we will be using $B = 10$ in our implementation.

2.2 Prediction

We will be using quadratic prediction scheme in our implementation. For an input curve, we store the first 3 vertices: v_0, v_1, v_2, and use these three to predict the next vertex v_3:

$$\begin{aligned} G_3 &= v_2 + (v_2 - v_1) + ((v_2 - v_1) - (v_1 - v_0)) && (1) \\ &= v_0 + 3(v_2 - v_1) && (2) \end{aligned}$$

then the correction vector $D_3 = v_3 - G_3$.

Emperical results have shown that going high than quadratic prediction, the improvement decelerates rapidly, and sometimes, the

*e-mail:nltruong@cc.gatech.edu
†e-mail:howardz@cc.gatech.edu

prediction can be worse. Therefore, we decide to use quadratic prediction as our prediction scheme.

2.3 Compression/Decompression

We used both the original Huffman coding and the adaptive Huffman coding in our curve compression implementation. After prediction, we can separate our data into two parts. The header and the residuals. The residuals are a list of correction vectors for the prediction. Since we have already quantized our curve, the correction vectors D_i's will be integers. Moreover, according to equation (1), D_i's are bounded by 4 times the min-max box. If the curve is relative smooth, then we can expect that D_i's will be clustered. Therefore, we can directly use huffman coding on those D_i's to encode the residuals. The header contains the information that is necessary for the reconstruction of the curve from residuals. We keep the number of vertices, the min-max box, the number of quantization bits, the prediction scheme used (in this case, quadratic) and the first 3 vertices in the header. For the header, although all of the above information can be represented as numbers, the range of those numbers is quite large, and they are reasonably unclustered. Therefore, direct encoding of these numbers would not save us much. However, since numbers use relative small character symbols, encoding those numbers as characters actually make sense here. In our implementation. we encode the header using an adaptive Huffman method. In short, adaptive Huffman encoding encodes symbols as it reads them in and update the Huffman tree in every turn. Since the encoder and decoder initialize the Huffman tree the same way, as they read in the symbols, they will construct the same Huffman tree on the fly. adaptive Huffman encoding is mainly used for encoding tasks where one parse of the whole contents may not be possible, such as stream video transmission. However, we use it here to encode our header for the sake of learning more about different kinds of Huffman encoding.

2.4 Simplification

Have implemented the quantization module, we can simplify a curve by using B-bit quantization and remving all vertices that are identical to their previous neighbor. For subdivision methods to reconstruct original curves from simplified ones, We have implemented three subdivision scheme: B-Spline, 4-Point subdivision and Jarek's subdivision. However, we are not going into detailed discussion about these schemes here.

2.5 Hausdorff Distance

We use a sampling method to calculate the Hausdorff distance approximately. The basic procedure is as follows. pick one curve, and for each vertex on that curve, calculate its distance to every edge of the other curve and keep the minimum of those for each vertex. Then, we find the maximum of all those minima. Now, we switch the curve and repeat. The larger one of the two maxima is the Hausdorff distance we are looking for.

3 Result

3.1 Compression/Decompression (CODEC)

We try our compression technique on 5 different curves, each of which is a result of B-Spline subdivisions on initial control polygons. This saves us time in making test curves, which could end up with several hundred vertices. We shall discuss the shortcoming/fallbacks of this approach, and will provide suggestions for future improvements in the project.

Our first control polygon is a closed-loop star, with no self-intersection. The original curve is colored in red, where as the decompressed curve is represented as blue. Intuitively, we want to measure the "effectiveness" of the compression/decompression scheme by comparing the "closeness" between the two curves. That is, we consider our scheme to be good if the two curves vary minimally from each other. As can be seen in the picture, the blue line follows the red line closely, with no discernible variations as viewed from this level. Note that the bits per vertex number is more than two times the entropy for this particular curve. This is due to the way we implement our compression scheme: we include the header into the compressed file, along with the first three vertices for the curve (refer to the IMPLEMENTATION section). Thus, we'd expect this ratio F/N to E to drop as the number of vertices in a curve increases. This assertion, in fact, can be shown to hold in other test cases. Consider the third curve: it does not have as many vertices as curve 1, and the ratio of F/N to E is more than 3. A similar observation for curve 4 also shows this fact: the ratio F/N to E is approximately 3.5, as its vertices only numbered 256, smallest compare to other curves. Our conjecture is that this ratio will approach entropy as the number of vertices tends to infinity.

A wishful thought is to explore the relationship between entropy and log of the average edge length. From the curve statistics, it is not obvious what this relationship may be. TO BE FILLED IN.

We have not created input files where the polygonal curves are not smooth, i.e. where curves are not created from subdivision, or do not follow any particular mathematical functions. An example of this may be a curve with vertices randomly chosen so that every one of them falls in a circle. The essenstial requirement is that we want jagged curves: this will then keep our vertex predictor off-guard, predicting vertices far away from their real values. This implies the prediction function will be wrong all the time, with residuals vary greatly over the curve. Since we are in effect encoding residuals, a large number of different residuals would force a big Huffman tree, hence would result in a not-so-good compression.

3.2 Simplification and Error Measure

Testings for our simplification scheme works as follow: we simplify the original curve, compress the simplified curve, decompress this compressed curve, then apply one of the three subdivision schemes to the decompressed curve. The results show that, in general, a simplification using 7 bits will suffice in recovering a curve very close to the original curve, measured in Hausdorff error. In all cases, it was shown the Hausdorff errors between the original curve, and those three curves resulted from B-Spline, Four-Point, and Jarek subdivisions, are in the order of 10^{-2} units in real coordinates.

4 Conclusion

Helmut Alt et. al. [Helmut Alt 1991] describes an way to measure the Hausdorff distance between two polygonal curves with complexity $O((p+q)log(p+q))$, where p and q are the number of vertices of polygonal curves A, and B, respectively. This is an improvement over our sampled $O(n^2)$ algorithm. Whether this algorithm is more efficient in practice, that we do not know. Implementing this algorithm would require computing Vor(A) and Vor(B), the Voronoi regions for A and B.

The simplification scheme implemented is a modification of the vertex-clustering method first proposed by Rossignac and Borrel. Our implementation simply removes consecutive vertices which happen to fall in the same cell. If the first vertex and the third vertex fall in the same cell, yet the second vertex does not, our implementation would not remove the third vertex, since it does not violate our criterium. In Rossignac-Borrel's scheme [Rossignac and Borrel 1993], we assign weights to vertices falling in the same cell, then replace all of these with a new vertex, whose position in the cell is based on the value of the weights calculated earlier. If this scheme is to be implemented, then more vertices will be removed when simplification is applied, and hence resulting in even smaller file size.

References

HELMUT ALT, BERND BEHRENDS, J. B. 1991. Approximate matching of polygonal shapes. In *Proceedings of the 7th Annual ACM Symposium of Computational Geometry*, 186–193.

HUFFMAN, D. A. 1952. A method for the construction of minimum-redundancy codes. In *Proceedings of IRE*, 1098–1101.

ROSSIGNAC, J., AND BORREL, P. 1993. *Multi-Resolution 3D Approximation for Rendering Complex Scenes. Modeling in Computer Graphics*. Springer-Verlag.