SYRUP and SAP: Exploring a special case of polygon simplification

Mike Buchanan

May 1, 2008

1 The Problem

A wide variety of computational topics have to deal with what is essentially *data smoothing*. Data smoothing goes by many names, including dimensionality reduction, noise removal, summarization, and, relevant to this paper, simplification. What all these fields have in common is moving a large data set to a smaller one that still has all the phenomena of interest. This is useful for a variety of reasons, ranging from aesthetics to performance.

This paper will be looking specifically at polygon simplification, with an eye towards speeding performance. I will use *polygon* in its normal sense; that is, a closed finite sequence of line segments in the plane. Polygons are not required to be convex, but must not self-intersect.

However, polygon simplification in general is well-studied and well-understood, so there is not much work for me to do here. Instead, I will focus on a more restricted class of polygons that came up in my work, namely *self-avoiding polygons* (SAPs). A self-avoiding polygon is named with reference to a self-avoiding walk; it is a planar random walk that does not repeat any points except that it ends where it started. Equivalently, it is a polygon whose vertices are lattice points and whose edges are rectilinear (such that all of its angles are right). It is hoped that the special structure of SAPs will allow simple but effective means of simplification.

While our original polygons will be self-avoiding, we will *not* require that our output, simplified versions be self-avoiding (i.e. their vertices need not fall on lattice points and their edges need not be rectilinear).

In this paper, I will describe different metrics for simplification, then four different methods for performing simplification, and then discuss which method is best.

2 What is Simplification?

In order to have a well-posed problem, we need some better metric for simplification. If our only goal is to minimize the number of vertices used, it is easy to

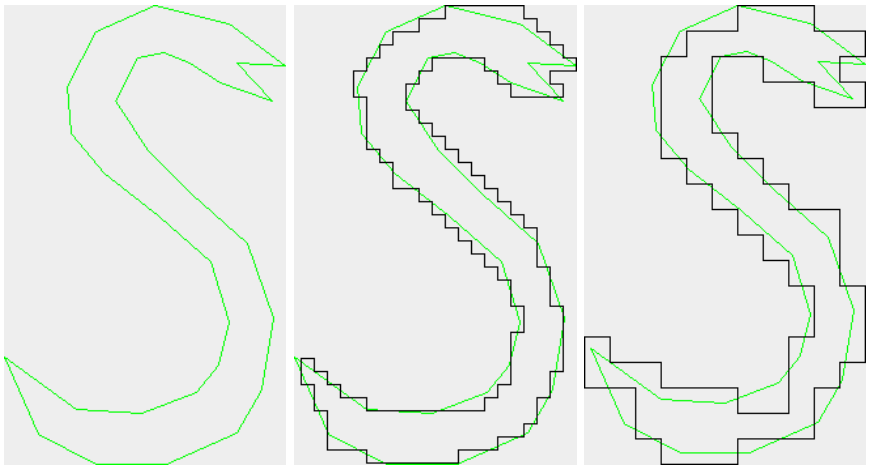

Figure 1: A snake polygon we will use as a running example, and two different self-avoiding versions of it with differently-spaced lattice points.

pick three non-colinear vertices that describe a triangle.

There are a number of possible ways to quantify simplification. One common way, called the *Hausdorff Distance* is to compute the maximum distance from a point on the original polygon to the nearest point on the simplified polygon[3]. (This is an application of the Hausdorff metric between compact sets of a metric space.) However, this is not entirely satisfying. It ignores all points but the farthest, and thus could rate two radically different simplifications, one much better than the other, equally. It is also not symmetric, which is not a problem *per se*, but is disappointing.

Instead of Hausdorff Distance, I will use *Hamming Distance*. Hamming Distance is typically thought of as a metric on strings, indicating the number

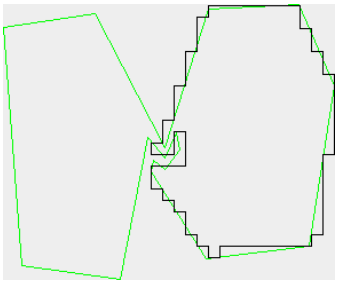

Figure 2: The Hausdorff Distance from the SAP to the green polygon is low, but the Hausdorff Distance from the green polygon to the SAP is high. No sane simplification algorithm will create the green polygon, but this sort of situation still casts doubt on the metric.

of positions in which have different symbols. In a discrete space, Hamming Distance on polygons is quite analogous. If we had two bitmaps representing polygons and expressed them as binary strings, the Hamming Distance between those strings would be the Hamming Distance between those polygons. More formally, the Hamming Distance between two polygons is the symmetric difference in the areas of those polygons. That is, it is the area of all regions that are contained in exactly one of the two polygons.

I will not use Hamming Distance exactly, though. The problem with Hamming Distance is that it is not normalized: large polygons are naturally Hamming-farther apart than small polygons. To account for this, I normalize the Hamming Distance by the area of the union of the two polygons (that is, the area of all regions that are contained in *at least* one of the two polygons).

However, even this metric is not enough. One flaw in some simplification techniques is that they tend to systematically reduce the area of the original polygon, and for some purposes, namely mine, this is a problem. So we also track the relative difference in area between the original polygon and its simplified form.

Also, since the problem domain is inherently visual, we also make some aesthetic considerations.

3 Solutions

3.1 Setting up a Framework

In order to assess the possible polygon simplification algorithms, I needed a framework for testing them. To this end, I created the program *SYRUP*, an acronym for "Simplifying Your Really Ugly Polygons". Syrup allows the user to simplify a given SAP using one of several methods, and see the results overlaid and the statistics about how good the simplification is.

However, there is a fair bit of work hidden in the phrase "given SAP". The original dataset of SAPs that I needed simplified is no longer available to me, so I decided the best solution would be to generate SAPs on the fly. Furthermore, since the SAP dataset I originally wanted to simplify was designed to describe regions that were general regions, not SAPs, I decided that these SAPs should be generated to approximate more general regions.

So, Syrup begins with the user drawing a polygon (not self-avoiding). This is then converted into a SAP. The algorithm for doing so is not particularly clever nor particularly concerned with accuracy, so I will not go into it here.

3.2 Douglas-Peucker

The canonical polygon simplification algorithm is known as the Douglas-Peucker algorithm[1]. The algorithm is best for simplifying *polylines*, that is, non-closed finite sequences of line segments. As such, we will describe the algorithm for polylines and then discuss its application to polygons.

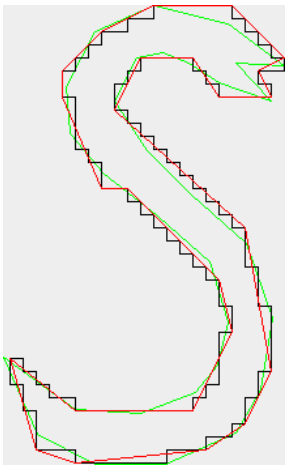

Figure 3: A Douglas-Peucker simplification of the snake (in red), with tolerance equal to the distance between lattice points. The green polygon is the same as it was above, that is, the source of the self-avoiding polygon, provided for comparison.

The procedure of the algorithm follows:

1. Given a polyline described by the points $\{p_0, p_1, \ldots, p_n\}$, consider the line $\{p_0, p_n\}$.
2. Find the point in $\{p_1, \ldots, p_{n-1}\}$ that is farthest from the line segment $\{p_0, p_n\}$. Call this p_f.
3. If the distance of p_f from that line segment is less than some specified tolerance, stop the algorithm, using as your line $\{p_0, p_n\}$.
4. If not, recursively simplify the polylines $\{p_0, p_1, \ldots, p_f\}$ and $\{p_f, \ldots, p_n\}$ and take as your final polyline the concatenation of those two.

Applying the algorithm to a polygon is straightforward. Simply treat the polygon as a polyline that goes around the polygon in a circuit, ending where it began. This polyline is degenerate at first (i.e. a point), but it quickly expands to be a proper polyline and then a proper polygon, assuming it recurses at least twice. The downside of this application is that the point at which you consider the polygon to "start" and "stop" is guaranteed to be included in the simplified version, and thus results of the simplification vary greatly depending which point is chosen.

Note that the inclusion of a tolerance parameter in the algorithm allows you to trade off simplification (as measured in number of vertices) and fidelity (as measured in Hamming Distance). Furthermore, the Douglas-Peucker algorithm guarantees that no point on the simplified polygon be farther than *tolerance*

units away from the original polygon, thus guaranteeing a small Hausdorff Distance.

Naive implementations of Douglas-Peucker (like mine) run in expected time $O(n \log n)$. However, clever use of convex hulls allows this to be reduced to $O(n \log^* n)$, where $\log^*$ denotes the iterated logarithm function, which is a constant for all but theoretical purposes[2]. Thus, there is not significant room to run faster than the industry standard, at least asymptotically.

3.3 Critical Edges

A very basic algorithm for simplifying SAPs (inapplicable to general polygons) relies on the identification of what I call "critical edges". The motivation stems from the fact that most SAPs tend to proceed stepwise for large sections, and the only really important behavior is when they turn. Thus, define a *critical edge* to be one whose two adjacent edges lie in the same half-plane described by the critical edge. That is, critical edges are like the bottom of a 'U', not the middle of a 'Z'. The algorithm, then, will simply consist of placing points on the critical edges and ignoring all others.

Pseudocode will clarify:

1. Consider each pair of adjacent vertices (i.e. edge) (p_0, p_1), (p_1, p_2), ..., (p_{n-1}, p_n), (p_n, p_0).
2. Check if the edge is critical by examining the relative location of the two vertices that bracket those two.
3. If the edge is critical, add the average of its two endpoints to the simplified polygon.

Unfortunately, this is not guaranteed to produce a valid polygon. While it will choose at least four points, the segments connecting them may cross. This rarely occurs in practice.

As something of an aside, it may seem that taking the average of the critical edge is not optimal. I did some experimentation with pushing the point outward in the direction that the polygon "was already going", but the results were unsatisfactory.

I do not know of any bounds on the badness of this algorithm's results. It is not hard to construct cases where it does quite badly, and they are by no means pathological: one such case is the square. Its running time is, at least, linear.

3.4 Staircase Simplification

A related algorithm that I devised for SAPs is what I will call *Staircase Simplification*. The assumption behind the algorithm is that the SAP is "trying" to be a regular polygon (which is a good assumption for my test framework, which calls into doubt the validity of my testing to some extent). The algorithm attempts to find the line segments of the underlying polygon.

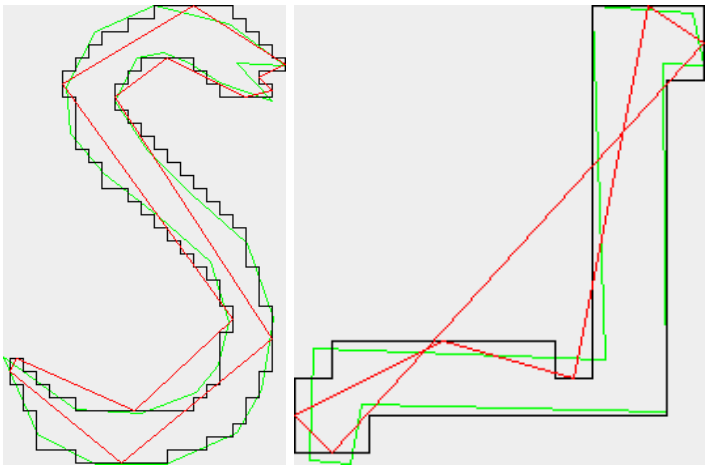

Figure 4: The critical-edges simplification of the snake, and an example of the algorithm producing a non-polygon.

To do so, it makes some basic assumptions about how line segments will be represented in a SAP. A set of sequential edges in a SAP come from the same underlying line segment if:

1. They move in at most one vertical direction (up or down) and one horizontal direction (right or left). Put another way, angles alternate between concavity and convexity.

2. All of the line segments in one direction are of length 1.

3. All of the line segments in the other direction differ in length by at most 1.

The algorithm then simply attempts to find the minimal decomposition of the SAP's edges into underlying edges.

The approach of my implementation is greedy. Starting at some point, proceed around the polygon. When you reach an edge that could not be grouped with the edges you've seen since the last point placed, add the near point of that edge to the simplified polygon.

I do not believe the staircase simplification method is capable of producing degenerate polygons, but have no proof of this fact.

3.5 Optimization Methods

A rather different approach to polygon simplification is to treat it as an optimization problem. That is, find the polygon with n vertices that is closest in some metric to the given polygon. It is a $2n$-dimensional problem, but that's not a major hurdle.

Figure 5: The staircase simplification of the snake.

First, we need to choose a metric. I chose the Hamming Distance, for most of the reasons outlined above. I did not include area ratio, because I think for most applications minimizing the Hamming Distance will minimize the absolute different in area.

Then, we need some optimization method. For a proof-of-concept, I implemented simple hill-climbing. The algorithm follows:

1. Select n points from the original polygon.
2. Randomly select one of the points and a random way it could be perturbed.
3. Check if that perturbation would reduce Hamming Distance.
4. If so, make that perturbation. (If not, don't.)
5. Repeat from step 2 a predetermined large number of times.

Like critical edges simplification, optimization is not guaranteed to produce a valid polygon, unless that constraint it explicitly enforced (a time-consuming check to make).

The tricky part of the algorithm is step 3, checking the Hamming Distance. Computing the Hamming Distance seems to require finding the intersection and union of two polygons and computing the area of both. Those are difficult algorithms to implement quickly, though they can be done in sub-quadratic time. For this proof-of-concept, I elected to use a simpler, but slower, method: brute force. I iterate over the polygons, and for each point check whether it is contained in one or both polygons. I considered using a Monte-Carlo approach, randomly sampling points from the region, but initial testing suggested this was unreliable until the number of points became very close to the number sampled in the brute force method.

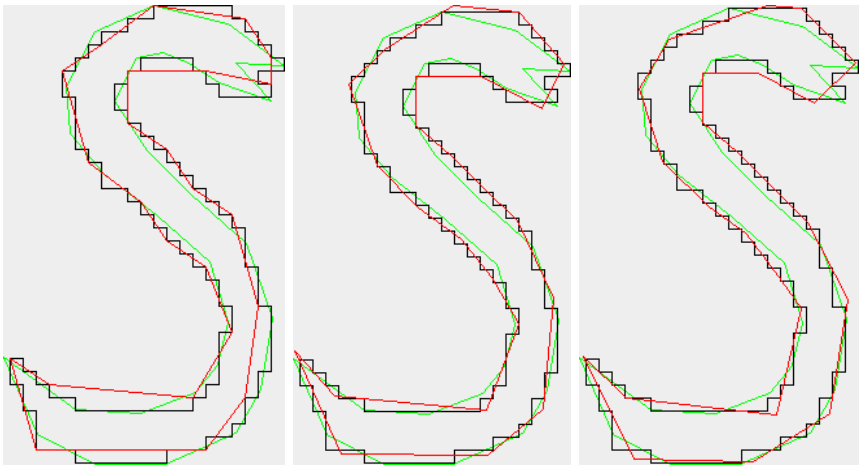

Figure 6: Optimized simplification of the snake after 0, 1000, and 3000 iterations.

However, there is one big runtime optimization to be made to computing the Hamming Distance. Specifically, to make the decision whether to perturb or not, the only thing you need to know is the sign of the difference between the two polygons, and you know that the polygons differ in only one point. So instead of iterating over all of the polygons, iterate over just the region containing the two edges joined to the point to be perturbed. This optimization has no worst-case effect on runtime, but in practice improves runtime *substantially*. It also causes the somewhat unexpected effect that the optimization algorithm runs faster when it is working with more points (i.e. in a higher-dimensional space), since each computation of the metric will be over a smaller region.

For actual use, one would probably want a more sophisticated method than simple hill-climbing. While polygon simplification seems to have few local minima, it does have some, so a method like simulated annealing would be preferable. Using an existing optimization package would be wise, though most would not be able to take advantage of the fact noted above that computing the goodness of a step is much easier than computing the Hamming Distance two polygons.

4 Conclusion

There were some trade-offs between the algorithms, but one seemed to be a clear winner. While critical edge simplification was the easiest to code, and runs extremely quickly, its output is more like a modern-art version of the input than a simplification. It may be suitable for some purposes where one is more concerned with topology than anything else, but is not appropriate for normal use.

Staircase simplification is better, providing a fairly good simplification with only marginally higher running time. Still, it makes some obvious "bad choices", and may fare poorly on data from SAPs imitating non-polygonal regions. It may be useful as input to an optimizing simplifier.

Treating simplification as an optimization problem did not turn out too well. It's hard to know how much of this is because of my admittedly-naive proof-of-concept implementation and how much an inherent fault, but optimization was orders of magnitude slower than other methods, and produced comparable results. It is probably best reserved for fine-tuning the results of another optimization method.

Douglas-Peucker simplification was the clear winner. It consistently quickly produced a simplified polygon of very low Hamming Distance. Its tunable parameter is extremely useful for specifying the simplification desired. In short, it's the gold standard of polygon simplification for a reason, and nothing about SAPs seems to dislodge it from its position.

References

[1] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica*, 10(2):112–122, 1973.

[2] John Hershberger and Jack Snoeyink. *Algorithms and Data Structures*, chapter Cartographic Line Simplification and polygon CSG formulae in $n \log^* n$ time. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1997.

[3] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, New York, 1997.