Maintaining simplicity in subdivision simplification

Ryan Coleman Jack Snoeyink Diane Souvaine

November 17, 2004

Abstract

The property of simplicity (lack of self-intersection) is difficult to maintain during cartographic line simplification because it requires global knowledges of the lines on a map. In this paper, we show how to maintain structure so that we can test whether a point can be removed without causing intersections, and perform such removals, both in $\log^2 n$ time for sets of polygonal lines with n total segments. Our structure can be used with any importance measure to decide the order to remove points, and can simplify lines while avoiding other lines or points.

1 Introduction

When displaying maps, we frequently want to simplify the representation of linear features. There are many algorithms for simplification in the literature of geographic information systems (GIS) [1], computer vision [4], and computational geometry. The practical literature typically presents heuristics that run in linear or near linear time, while the computational geometry literature tends toward optimal simplification under various criteria (e.g. minimum error for a given number of segments, or minimum number of segments for a given error) using algorithms that run in quadratic to cubic time.

Two desirable and natural restrictions to add to the line simplification problem is that boundaries that do not self-intersect should not be made to self-intersect, and that lines should not pass over point features, but should keep the same homotopy type. Self-intersections, in cartographic applications, usually indicate errors in digitization or processing [6]. Nevertheless, most algorithms simplify collections of polygonal lines by considering each in isolation, and running the risk of introducing intersections and topological changes in the data. The cartographers' favorite algorithm, reported in Douglas and Peucker [**?**], can even introduce self-intersections within one polygonal line. For simplifying sets of lines, such as river banks and elevation contours, a more global approach is needed.

A few simplification methods have been developed in computational geometry that guarantee to preserve map topology. De Berg *et al.* [**?**] give a method that simplifies an n-point polygonal line without passing over itself or over m specified points in $O(n(n+m)\log n)$ time. Estkowski and Mitchell [2] give a heuristic for simplifying parallel lines, such as elevation contours, in quadratic time. Others have used the Voronoi diagram to identify regions in which it is safe to perform simplification without creating intersections [5, 7].

Unfortunately, Estkowski and Mitchell [2] argue that requiring simplicity in subdivision simplification using the original points makes the problem MIN PB-complete and hard to approximate within a factor of $n^{1/5-\delta}$. Even for a single path that winds through a (non simply-connected) region, it is NP-hard to determine if there exists a polygonal line of the same homotopy class that has less than k line segments [3].

At a Dagstuhl seminar in October 2003, Lars Kulik asked if there was a way to test quickly whether the following *shortcutting* operation was possible: remove an point p_i from the interior of a polygonal line, and directly join its former neighbors with a line segment $\overline{p_{i-1}p_{i+1}}$, provided the triangle $\triangle p_{i-1}p_ip_{i+1}$ is empty of points. Such an operation could fit into a number of simplification methods, especially bottom-up schemes in which some importance score can be assigned to vertices on the basis of neighboring angles or vertices. Under such schemes, low priority vertices are candidates for shortcutting, unless they would change homotopy or introduce self-intersections. Successful shortcuts may involve recomputing a constant number of priorities for neighbors, and the operation can be repeated until a target number of vertices or a target priority is reached.

In the definitions of Section 2 we generalize this question to simplifying planar subdivisions under the operations of shortcutting vertices of degree 2 and of removing edges. In Section 3 we describe a data structure that can perform each triangle test and valid shortcutting operation in $O(\log^2 n)$ time, after $O(n \log n)$ preprocessing using linear space. In Section **??** we integrate this with line simplification, and in Section 4 we report on preliminary experimental results.

It is not surprising that our data structure shares some common elements with subdivision maintenance structures that have been developed for dynamic planar point location. In particular, we maintain a decomposition into x-monotone polygons, much like (a 90° rotation of) Preparata and Tamassia's dynamic point location [**?**] structure. In fact, one might wonder whether the structures used for dynamic point location [**?**, **?**, **?**] do not already solve the problem of maintaining simplicity during simplification. Unfortunately, these structures pass the tasks of avoiding intersections and preserving topology to the next level, and assume that all subdivision maintenance operations preserve simplicity, and even connectedness. They do not check that a sequence of operations are valid, but guarantee their results only then they are.

2 Definitions and preliminaries

Although we are primarily concerned with line simplification, our solution will handle subdivisions with high degree vertices and isolated points. Thus, we define the problem generally, using the terminology of embedded graphs. Suppose that we are given a graph G with n vertices that is embedded in the plane with straight line edges that do not cross. To make description easier, we assume that vertices have distinct x coordinates – this can be simulated by breaking ties by y coordinates. Each vertex p has a set of *neighbor vertices*, $N(p)$, that are connected to p by an edge, and the *degree of* p, denoted $d(p) = |N(p)|$, the number of neighbors. We say that vertex p *can see* q if $q \in N(p)$ or if the interior of the line segment $\overline{pq}$ does not intersect a vertex or edge of G.

A *polygonal line* is a sequence of vertices $P = \{p_1, p_2, \ldots, p_k\}$ in which adjacent vertices are joined by an edge and all vertices except the first and last have degree exactly 2. A polygonal line P defines a homotopy class of paths that are deformable to P without intersecting other vertices or edges of the embedded graph G. When we perform line simplification, it is desirable to choose a representation in the same homotopy class to preserve the topological relationships with nearby features. Choosing representation from outside the homotopy class can move a house to the wrong side of a road, or a city to the wet side of a coastline.

If we use the above-defined shortcutting operation on a degree 2 vertex p_i only when p_i passes a *triangle test* – i.e. the triangle $\triangle p_{i-1}p_ip_{i+1}$ contains no vertex of G – then no intersections can be added or homotopy class changed. A linear-time triangle test is trivial – simply test whether any vertex of G lies inside the given triangle. This can be accelerated in practice by sorting by x coordinate or using buckets to test only those features of G that overlap the triangle in x (and y) coordinates. We will present a data structure that supports triangle tests in $\log^2 n$ time apiece.

Note that we apply shortcutting only to vertices of degree 2. Vertices of degree one or zero can always be removed, and we support such removals in our structure. Vertices of higher degree (e.g., *nodes* in an ARC/INFO planar map structure) must first have their degrees reduced by splitting or deleting edges before they can be removed. This reflects the way that such nodes are used in digital maps.

3 Data structure

In this section we detail a data structure that stores a planar subdivision so that edges and vertices of degree at most one can be removed, and vertices of degree 2 can be removed and shortcut, provided the triangle they define is empty.

The primary consideration in looking for an efficient data structure is that no vertex or edge should participate in too many auxiliary segments or pointers, or else the removal of such a point will cause too much data structure rebuilding. This cannot be guaranteed by a constrained triangulation or a trapezoidation of G. We show that it can be by a canonical decomposition of the faces of G into monotone polygons, storing the chains that bound these polygons in an Overmars/van Leeuwen convex hull structure to answer queries for tangents in given intervals. In the next subsections, we describe a canonical decomposition into monotone polygons, the hull structure, and its application to triangle tests.

3.1 Canonical decomposition into monotone polygons

A polygon Q is *monotone* (also known as x-monotone) if any vertical line intersects it in a single interval. Equivalently, we can split the boundary of Q into *upper* and *lower* monotone polygonal lines, which any vertical line intersects in at most one point. We maintain the decomposition into monotone polygons that is the first step of triangulation [**?**, sec 3.2] from Lee and Preparata's point location algorithm [**?**].

We say that a vertex p is a *left extreme* of a polygon if it is to the left of its neighbors – if $p.x < q.x$ for all $q \in N(p)$. Similarly, p_i is a *right extreme* if $p.x > q.x$ for all $q \in N(p)$. (Recall that we assume that points have distinct x-coordinates.) For a graph G, note that any vertex with degree greater than 2 is necessarily extreme for at least one of its incident polygons.

Missing figure monotone.ps

Figure 1: Decomposition into monotone polygons

We maintain a decomposition of the plane into x-monotone polygons according to the following rule: add a *decomposition edge* to connect every left extreme to the rightmost point that it can see to the left, and every right extreme to the leftomost point that it can see to the right. We include two points at infinity $(\pm\infty, 0)$ as the leftmost and rightmost points to avoid some special cases. Decomposition edges are drawn gray in Figure 1; their number at any vertex is limited.

Lemma 1 *A degree d vertex is incident on at most $d+2$ decomposition edges.*

Proof: This is most easily observed from a trapezoidation of the input graph by making vertical cuts from each vertex until we hit segments, as illustrated in Figure **??**. Each decomposition edge added is completely contained in one trapezoid and $d+2$ trapezoids (or triangles, which are degenerate trapezoids for this proof) touch any vertex of degree d. ∎

We need to give a more specific detail about what is in the data structure for a monotone subdivision. First, we store an array of vertices, each with its degree and pointers to two of the polygons on which it appears. This allows us toe access polygons by their removable, degree 2 vertices, at least. Second, we store each monotone chain of a polygon in a data structure that supports search by x-coordinate, split, merge, and search in a given x-interval for an extreme point in a given direction or tangent from a given point. The first three operations can be supported by any balanced binary search tree. In fact, since we know all n points in advance and only delete, we need not perform rotations or dynamic rebalancing to guarantee logarithmic search tree height. For the last operation, we use Overmars/van Leewen dynamic convex hulls.

Overmars and van Leeuwen showed how to maintain the upper convex hull under insertion and deletion of points in $O(\log^2 n)$ time per operation [**?**]. They do this by keeping track of the merge of convex hull in x-coordinate order. Thus, we can use their structure to keep track of convex hulls of our monotone chains, and can also support splits and merges of monotone chains in the same time bound.

The basic query on this structure will be the following: for the points whose x coordinates lie in a query range $[a, b]$, report the convex hull point with a tangent of slope s. The answer can be obtained in $O(\log^2 n)$ time by finding at most $2\log_2 n$ subtrees that cover the points in $[a, b]$, and then using binary search on the tangent slope s in each subtree. When we merge or split monotone chains we need to repair the tangents at $\log n$ levels of OvL structures, which can be done in $O(\log^2 n)$ time.

3.2 Triangle test

Finally, we observe how monotonicity and convex hulls can reduce the triangle test – whether triangle $\tau = \triangle p_{i-1}p_ip_{i+1}$ is free of points – to two cases of testing a single point, which are illustrated in Figure 2. Let us assume that the acute angle $\angle p_{i-1}p_ip_{i+1}$ is below the edge $\overline{p_ip_{i+1}}$ with $p_{i-1}.x < p_{i+1}.x$, and either p_i is a left extreme or lies between p_{i-1} and p_{i+1}.

In both cases, we use the hull structure to test if there is a point in the triangle with x coordinate in the interval $[p_{i-1}.x, p_{i+1}.x]$ by querying the lower chain in this interval for the hull vertex that has a tangent with the slope of $\overline{p_{i-1}p_{i+1}}$, and testing if the resulting vertex lies inside the triangle τ.

Missing figure triangles.ps

Figure 2: Cases for triangle test

When p_i is a left extreme, there is a point in τ with x coordinate in $[p_i.x, p_{i-1}.x]$ if and only if there is another extreme point in this region of τ. When this occurs, p_i necessarily has a decomposition edge to an extreme vertex that lies in this region, which can be determined from the monotone decomposition data structure.

3.3 Maintaining the data structure

If we would maintain the trapezoidation as we shortcut and remove vertices of degree at most 2, then it would be relatively easy to update in time proportional to the number of changes. Unfortunately, since an edge can be the base of many trapezoids, the number of changes could be $\Omega(n)$ for each deletion. Instead we will find trapezoids when we need them.

To update the structures as we delete a vertex p, we need to do a detailed case analysis, but the basic idea is: for the at most four points connected to p_i by new edges, shoot up and down to find the segments that you will hit. The endpoints of these segments (plus the four points) are the candidates for any new connections that have to replace the old connections to p_i. I'll have to draw more pictures to illustrate how that all goes.

4 Experimental results

We tested our algorithm on coastline data from the British Columbia watershed atlas `http://www.bcfisheries.gov.bc.ca/fishinv/basemaps-watershed.html`, which was scanned from 1:50K NTS mapsheets from Geomatics Canada, the highest resolution data in the Global Self-Consistent Hierarchical High-Resolution Shoreline [8] `http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html`, Contours derived from bathymetry in the Gulf of Maine `http://pubs.usgs.gov/of/of98-801/bathy/data.htm`.

One natural question is whether there is any practical advantage to the use of the Overmars/van Leeuwen convex hall structure, or whether we should perform a linear search of the points of the monotone chain in the interval $[p_{i-1}.x, p_{i+1}.x]$. Figure **??** shows the distribution the number of points that fall in this range.

((other tests?)) ((what simplification criteria work well?)) Candidate simplification criteria: importance could be angle deviation from 180°, triangle area, edge length, douglas-peucker recursion depth...

Acknowledgments

We thank Lars Kulik for posing this question. This research has been partially supported by NSF ITR grant 0086013, and **((Souvaine))**.

References

[1] Barbara Buttenfield and Robert McMaster, editors. *Map Generalization: Making Rules for Knowledge Representation.* Wiley, 1991.

[2] Regina Estkowski and Joseph S. B. Mitchell. Simplifying a polygonal subdivision while keeping it simple. In *Proc. 17th ACM SCG*, pages 40–49, 2001.

[3] Leonidas J. Guibas, John E. Hershberger, Joseph S. B. Mitchell, and Jack Scott Snoeyink. Approximating polygons and subdivisions with minimum-link paths. *Int. J. Comp. Geom. Appl.*, 3(4):383–415, 1993.

[4] L. Lam, S.-W. Lee, and C. Y. Suen. Thinning methodologies – a comprehensive survey. *IEEE Pat. Anal. Mach. Int.*, 14(9):869–885, 1992.

[5] Andrea Mantler and Jack Snoeyink. Safe sets for line simplification. Presented at CGC Workshop, 2000.

[6] Robert Brainerd McMaster. Automated line generalization. *Cartographica*, 24(2):74–111, 1987.

[7] Nabil H. Mustafa, Eleftherios Koutsofios, Shankar Krishnan, and Suresh Venkatasubramanian. Hardware-assisted view-dependent map simplification. In *Proc. 17th ACM SCG*, pages 50–59, 2001.

[8] P. Wessel and W. H. F. Smith. A global self-consistent, hierarchical, high-resolution shoreline database. *J. Geophys. Res.*, 100(B4):8741–8743, 1996. `http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html`.

A Ideas for improvement

Since we never add points, another way to store monotone chains is in an interval tree, as

Guibas et al. [**?**] developed compact interval trees to store convex hulls and support queries on the hulls of subpaths.

((There is a chance that compact interval trees will work better. Here is a brief intro to them.)) Note that all our convex polygon vertices are drawn from a fixed set of n points. The sorted list of x coordinates defines $n-1$ adjacent intervals on the x axis. We recursively build a complete binary tree on these intervals: at the root of the tree we store the median interval, and build subtrees for using the midpoint of each as the location for a node. The *interval tree* is a complete binary tree on these intervals. An edge is stored at a node in the interval tree if its endpoints the node's interval lies between the y coordinates of the edge's endpoints; an interval tree stores an edge at the the highest node that that edge spans. For storing the edges of a (left or right) convex hull, we can use a *compact interval tree* which is a tree obtained from an interval tree by deleting empty nodes, making non-empty nodes be children of their nearest non-empty ancestor. Note that any node stores at most one hull edge, and that the lowest common ancestor of two nodes that store an edge must also store an edge. Thus, the compact interval tree is still binary. If each node also stores the level of the underlying interval tree from which it came, then we can emulate a top-down search in the interval tree. Thus,

Since the searches for intersections and common tangents walk down interval trees, we can simulate them on the compact interval tree by remembering edges stored at ancestors.

Building the compact interval tree for a convex hull with h edges takes $O(()$h) time. Given the list of nodes in x coordinate order, make lower-level nodes the children of higher-level nodes. Merging two compact interval trees: Given compact interval trees for the red and blue convex polygons, build the compact interval tree for their convex hull in $O(\log n)$ time. Non-destructive merging: copy the nodes to be altered in the merge algorithm and modify the copies.

Compact interval trees support:

Time	Operation
$\min(\log n, h)$	extreme vertex in given direction
$\min(\log n, h)$	tangent from a point
h	constructing a compact interval tree
$\min(\log n, h)$	intersection searches

$\min(\log n, h)$	finding single common tangent
$\min(\log n, h)$	merging two compact interval trees