Algorithms for GIS csci3225

Laura Toma

Bowdoin College

Simplification

2D: Line simplification

3D: Terrain simplification



https://algorithmist.files.wordpress.com/2011/10/lang.jpg

Why?

- storing polygonal lines at high level of detail consumes lots of memory
- often high level of detail is not necessary

Demos

- https://bost.ocks.org/mike/simplify/
- http://mourner.github.io/simplify-js/
- https://www.jasondavies.com/simplify/

Given:

- A polygonal line $P = p_1, p_2, p_3, \dots p_n$ (assume not self-intersecting)
- An error threshold epsilon

Want a set S of points such that distance(S, P) < epsilon

Given:

- A polygonal line $P = p_1, p_2, p_3, \dots p_n$ (assume not self-intersecting)
- An error threshold epsilon

epsilon: small

Want a set S of points such that distance(S, P) < epsilon

June 1, 2012 / Mike Bostock

Line Simplification

0.00011px² / 75.77%

plification

epsilon: large

63px² / 1.06%

Given:

- A polygonal line $P = p_1, p_2, p_3, \dots p_n$ (assume not self-intersecting)
- An error threshold epsilon

Want a set S of points such that distance(S, P) < epsilon



General approaches

- Greedy insertion
 - Start from a coarse approximation and add one point at a time, until all remaining points are within the tolerance
- Decimation
 - Start with all points in P and eliminate one point at a time

Outline

- Ramer-Douglas-Peucker algorithm (greedy insertion) <------ most popular
- Visalingam algorithm (decimation)
- Imai-Iri algorithm

- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon

How to define the error, i.e. the distance between S and P?



- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon

How to define the error, i.e. the distance between S and P?



For every p in P: dist(p, af) = length of perpendicular from p to af

- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon

How to define the error, i.e. the distance between S and P?



For every p in P: dist(p, af) = length of perpendicular from p to af dist(P, S) = max {p in P, dist(p,S)}

- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon



- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon



- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon



d(S,P) = max distance of a point in P to its segment in S

- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon

```
//return a subset S of pi....pj so that dist(S, P) < eps</pre>
DP(P, i, j, eps)
     //basecase
     if i+1 == j: return [p_i, p_j]
     else
       for k=i+1 to j-1: compute the perpendicular-
       distance(p_k, p_ip_j) and find the point p_k that's
       farthest from p<sub>i</sub>p<sub>i</sub>
       S1 = DP(P, i, k, eps)
       S2 = DP(P, k, j, eps)
       return [S1 + S2] //S1 concatenated with S2
```

























- Input: Parray of n points, epsilon
- Output: subset S of P such that distance(S,P) < epsilon

```
//return a subset S of pi....pj so that dist(S, P) < eps</pre>
DP(P, i, j, eps)
     //basecase
     if i+1 == j: return [p_i, p_j]
     else
       for k=i+1 to j-1: compute the perpendicular-
       distance(p_k, p_ip_j) and find the point p_k that's
       farthest from p<sub>i</sub>p<sub>i</sub>
       S1 = DP(P, i, k, eps)
       S2 = DP(P, k, j, eps)
       return [S1 + S2] //S1 concatenated with S2
```

Analysis

- Self-intersections
 - Is it possible that S self-intersects?

- Optimality
 - S is a subset that is within epsilon of P. Is it the best one? i.e. Is S is the smallest possible subset that approximates P within epsilon?

- Analysis
 - O(n²) worst case, O(n lg n) if good splits
- Self-intersections
 - Is it possible that S self-intersects?
 - YES
- Optimality
 - S is a subset that is within epsilon of P. Is it the best one? i.e. Is S is the smallest possible subset that approximates P within epsilon?
 - NO

S can self-intersect

S is not optimal (wrt size)

- O(n²) worst case
- S may be self-intersecting
- S not optimal
- However, very popular in practice :)
- Improvements
 - Run RDP with eps=0 and keep track of the splitting points and their distances. Then, for a given eps, run the algorithm, but instead of searching for the splitter just use it. RDP in O(n) for any epsilon.
 - Improved to O(n lg n) worst-case using dynamic convex hulls [Hersheberger, Snoeyink]

Line simplification: Imai-Iri algorithm

- Input: Parray of n points, epsilon
- Output: subset S of P of minimum size such that distance(S,P) < epsilon

- Idea: Define a directed graph G=(V,E)
 - **vertices**: $V = \{ p_1, p_2, ..., p_n \}$
 - edges: all (p_i,p_j) such that i<j and p_ip_j is an epsilon-approximation for {p_i.....p_j}
 (an edge is an "admissible shortcut" for p_i.....p_j)

 $d(\{p_i,\ldots,p_j\}, p_ip_j) < epsilon$










Any possible admissible shortcut from p_i to p_j is represented by an edge

Claims:

- There is at least one path in G from p_1 to p_n
- Any path in G from p_1 to p_n is an epsilon-approximation for P



Claims:

- There is at least one path in G from p_1 to p_n
- Any path in G from p_1 to p_n is an epsilon-approximation for P



We want a path from p_1 to p_n with fewest number of edges.

Line simplification: Imai-Iri algorithm

- 1. Construct G=(V,E)
- 2. Compute shortest paths in G from p_1 to p_n



Line simplification: Imai-Iri algorithm





- S has optimal size
- S may self-intersect

Line simplification: decimation

- Decimation approach
 - Each point is assigned an "importance" score
 - Delete a point with low importance
 - Repeat

A point is important if removing it introduces a large error

- \bullet Each point p_i is assigned the area of triangle $p_{i\text{-}1}p_ip_{i\text{+}1}$
- Repeat
 - \bullet Delete point p_i with lowest area
 - Recompute (only the scores of p_{i-i} and p_{i+1} changes)

- Each point p_i is assigned the area of triangle $p_{i-1}p_ip_{i+1}$
- Repeat
 - Delete point p_i with lowest area
 - Recompute (only the scores of p_{i-i} and p_{i+1} changes)



https://bost.ocks.org/mike/simplify/

- Each point p_i is assigned the area of triangle $p_{i-1}p_ip_{i+1}$
- Repeat
 - Delete point p_i with lowest area
 - \bullet Recompute (only the scores of $p_{i\text{-}i}$ and $p_{i\text{+}1}$ changes)

- Analysis?
- Self-intersecting ?
- S optimal (wrt size)?

- Each point p_i is assigned the area of triangle $p_{i-1}p_ip_{i+1}$
- Repeat
 - \bullet Delete point p_i with lowest area
 - \bullet Recompute (only the scores of $p_{i\text{-}i}$ and $p_{i\text{+}1}$ changes)
- Analysis? O(n lg n)
- Self-intersecting ? YES
- S optimal (wrt size) ? NO

- Each point p_i is assigned the area of triangle $p_{i-1}p_ip_{i+1}$
- Repeat
 - Delete point p_i with lowest area
 - Recompute (only the scores of p_{i-i} and p_{i+1} changes)
- Analysis? O(n lg n)
- Self-intersecting ? YES
- S optimal (wrt size) ? NO
- Errors accumulate
 - A point p is removed because it's approximated well by its neighbors, assuming those neighbors are kept
 - But those neighbors may be removed in the future
 - => No guarantee that S is within epsilon of P

Ramer-Douglas-Peucker algorithm

- distance(S,P) < eps
- O(n²) worst case
- S not optimal size
- S may self-intersect

Visvalingam-Whyatt algorithm

- distance(S,P) could be > eps
- O(n lg n) worst case
- S not optimal size
- S may self-intersect

Imai-Iri algorithm

- distance(S,P) < eps
- O(n²) worst case
- S optimal size
- S may self-intersect

Other results

- Improved RDP: O(n Ig n) [HS]
- With some effort can avoid self-intersections
- Simpler/faster algorithms for special cases (like monotone chains)
- Algorithms that maintain topological constraints (like keep left-of relationships)

Mark de Berg

Dept. of Computer Science Utrecht University P.O.Box 80.089 3508 TB Utrecht The Netherlands markdb@cs.ruu.nl

Marc van Kreveld

Dept. of Computer Science Utrecht University P.O.Box 80.089 3508 TB Utrecht The Netherlands marc@cs.ruu.nl

Stefan Schirra Max-Planck-Institut für Informatik Im Stadtwald

D-66123 Saarbrücken

Germany stschirr@mpi-sb.mpg.de



Figure 1: Part of a map of Western Europe, and an inconsistent simplification of the subdivision.

Avoiding self-intersections: some ideas

• Pro-active:

- Before adding a segment, check whether it intersects with existing segments
- Retro-active:
 - run RDP(P, epsilon) [or Imai-Iri(), or..]
 - repeat
 - check if segments in S intersect
 - If YES, let s be a segment in S that causes intersections. Go back and refine it (add a point of maximum distance, even if its closer than epsilon)
 - until S causes no intersections

3D: Terrain simplification

Data sources



grid to TIN

point cloud to TIN

satellite imagery



LIDAR point cloud

Terrain models



Terrain simplification

A terrain consists of:

- A set $P = \{ (x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_n, y_n, z_n) \}$ of terrain elevation samples
- An interpolation method

Denote by surf(P) the surface corresponding to terrain P

P could be a grid or a point cloud

sometimes called a "height field"



n points

Terrain simplification

- Input: A set P and an error threshold epsilon
- Output: A subset S of P such that distance(surf(S), surf(P)) < epsilon



dist(Surf(P}, Surf(S)) < epsilon

Outline

- Grid-to-TIN
- Point-cloud-to-TIN
- Point-cloud-to-grid

- We'll focus on grid-to-TIN simplification
- The method can be extended to deal with arbitrary (non-grid) data

Grid-to-TIN simplification





Grid-to-TIN simplification

- WHY?
 - uniform resolution means a lot of data redundancy
 - grids get very large very fast
 - Example:
 - Area if approx. 800 km x 800 km
 - Sampled at:
 - 100 resolution: 64 million points (128MB)
 - 30m resolution: 640 (1.2GB)
 - 10m resolution: 6400 = 6.4 billion (12GB)
 - 1m resolution: 600.4 billion (1.2TB)

Grid-to-TIN simplification

- Incremental refinement (greedy insertion)
 - Start with an initial approximation and add points one by one [Garland & Heckbert, 1995]

- Decimation methods
 - start with P and discard points (one by one)
 - E.g.: Lee's drop heuristic
- One-pass methods
 - pre-compute importance of points
 - select points that are considered important features and triangulate them
 - based on quad trees or kd-trees

Fast Polygonal Approximation of Terrains and Height Fields

Michael Garland and Paul S. Heckbert September 19, 1995 CMU-CS-95-181

> School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213

email: {garland,ph}@cs.cmu.edu tech report & C++ code: http://www.cs.cmu.edu/~garland/scape

Abstract

Several algorithms for approximating terrains and other height fields using polygonal meshes are described, compared, and optimized. These algorithms take a height field as input, typically a rectangular grid of elevation data H(x, y), and approximate it with a mesh of triangles, also known as a triangulated irregular network, or TIN. The algorithms attempt to minimize both the error and the number of triangles in the approximation. Applications include fast rendering of terrain data for flight simulation and fitting of surfaces to range data in computer vision. The methods can also be used to simplify multi-channel height fields such as textured terrains or planar color images.

The most successful method we examine is the greedy insertion algorithm. It begins with a simple triangulation of the domain and, on each pass, finds the input point with highest error in the current approximation and inserts it as a vertex in the triangulation. The mesh is updated either with Delaunay triangulation or with data-dependent triangulation. Most previously published variants of this algorithm had expected time cost of O(mn) or $O(n \log m + m^2)$, where n is the number of points in the input height field and m is the number of vertices in the triangulation. Our optimized algorithm is faster, with an expected cost of $O((m+n) \log m)$. On current workstations, this allows one million point terrains to be simplified quite accurately in less than a minute. We are releasing a C++ implementation of our algorithm.

Greedy insertion	Notation:
	 P = set of grid points
	 P' = set of points in the TIN
	TIN(P'): the TIN on P'
Algorithm:	
 P = {all grid points}, P' = {4 corner points} 	
• Initialize TIN to two triangles with 4 corners as vertices	
 while not DONE() do 	
 for each point p in P, compute error(p) 	
 select point p with largest error(p) 	
• insert p in P', delete p from P and update TIN(P')	
create 3 new triangles	
find triangle that contains p and compute the between p and its interpolation on the trian	ne vertical difference in height

DONE() :: return (max error below given epsilon) ?TRUE; FALSE;

Greedy insertion

• Assume when inserting a point in a triangle, split the triangle in 3



Come up with a "straightforward" implementation of greedy insertion and analyze it.

Greedy insertion: Analysis

	P	P'
initially	n	4 = O(1)
iteration 1	n-1	1 + O(1)
iteration 2	n-2	2 + O(1)
iteration k	n-k	k
at the end	n-m	m

- Notation:
 - m = nb of vertices in the simplified TIN at the end (when error of P' falls below epsilon) [usually m is a fraction of n (e.g. 5%)]



ANALYSIS: At iteration k: we have O(n-k) points in P, O(k) points in P'

- RE-CALCULATION
 - compute the error of a point: must search through all triangles to see which one contains it ==> worst case O(k)
 - compute errors of all points ==> O(n-k) x O(k)
- **SELECTION**: select point with largest error: O(n-k)
- **INSERTION**: insert p in P', update TIN ==> O(1)
 - unless each point stores the triangle that contains it, need to find the triangle that contains p
 - for a straightforward triangulation: split the triangle that contains p into 3 triangles ==> O(1) time

Analysis worst case:

• iteration k: $O((n-k) \times k) + O(n-k) + O(1)$



- overall: SUM { (n-k) × k } = ...= O(m²n)
- Note: dominant cost is re-calculation of errors (which includes point location)

• More on point location:

- to locate the triangle that contains a given point, we "walk" (traverse) the TIN from triangle to triangle, starting from a triangle on the boundary (aka DFS on the triangle graph).
- we must be very unlucky to always take O(k)
- simple trick: start walking the TIN from the triangle that contained the previous point.
 - because points in the grid are spatially adjacent, most of the time a point will fall in the same triangle as the previous point or in one adjacent to it
- average time for point location will be O(1)



Average case: O(mn)

• trick to speed up point location ==> average time for pt location will be O(1)

• SUM
$$\{O(n-k)\} = O(mn)$$

Observation: Only the points that fall inside triangles that have changed need to re-compute their error.

• Re-compute errors ONLY for points whose errors have changed

- Each point p in P stores its error, error(p)
- Each triangle stores a list of points inside it

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
 - for each point p in P, compute error(p)
 - select point p with largest error(p)
 - insert p in P', delete p from P and update TIN(P')
 - create 3 new triangles
 - for all points in triangle that contains p:
 - find the new triangles where they belong, re-compute their errors

Worst-case: O(mn) • iteration k: - + O(n-k) + O(1) + O(n-k) x O(1) • \mathbf{A} RE-CALC SELECT INSERT + re-calc • overall: SUM {O(n-k) } = O(mn)

Average case: O(mn)

 if points are uniformly distributed in the triangles ==> O((n-k)/k) points per triangle



• Version2, re-calculation goes down and selection becomes dominant



• store a heap of errors of all points in P

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
 - use heap to select point p with largest error(p)
 - insert p in P', delete p from P and update TIN(P')
 - for all points in the triangle that contains p:
 - find the new triangles where they belong, re-compute their errors
 - <u>update new errors in heap</u>
Greedy insertion—VERSION 3

Worst-case: O(mn lg n)

- iteration k: • $+ O(lg (n-k)) + O(1) + O(n-k) \times O(lg (n-k))$ • $+ P(1) + O(n-k) \times O(lg (n-k))$
- overall: SUM $\{(n-k) | g(n-k)\} = O(mn | gn)$

Average case: O((m+n) lg² n)

- if points are uniformly distributed in the triangles ==> O((n-k)/k) points per triangle
- iteration k: • P = CALC• SUM {lg (n-k) + O((n-k)/k} = O((m+n) lg² n)

heap updates will be dominant!

Greedy insertion—VERSION 4

- Version 3: selection is down, but updating the heap is now dominant
- Version 4: store in heap only one point per triangle (point of largest error)

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with 4 corners as vertices
- while not DONE() do
 - use heap to select point p with largest error(p)
 - insert p in P', delete p from P and update TIN(P')
 - for all points in the triangle that contains p:
 - find the new triangles where they belong, re-compute their errors
 - <u>find point with largest error per triangle</u>
 - add these points (one per triangle) to the heap

Greedy insertion—VERSION 4



Average case: O((m+n) lg n)

- if points are uniformly distributed in the triangles ==> O((n-k)/k) points per triangle
- iteration k: + $O(\lg k) + O(1) + O((n-k)/k) \times O(1) + O(1) \times O(\lg k)$ \uparrow RE-CALC SELECT INSERT + re-calc
- SUM { $\lg k + O((n-k)/k$ } = O((m+n) $\lg n$)













- The straightforward way to triangulate when adding new points runs in O(1) time but will create long and skinny triangles
- Small angles are undesirable (numerical instability)
- Good meshes have uniform triangles and angles that are neither too small
 nor too large



http://doc.cgal.org/latest/Surface_mesh_simplification/Illustration-Simplification-ALL.jpg



• A triangulation of a point set P in 2D is a triangulation of the convex hull of P



• A triangulation of a point set P in 2D is a triangulation of the convex hull of P



• A triangulation of a point set P in 2D is a triangulation of the convex hull of P



• Many ways to triangulate a set of points P



- Many ways to triangulate a set of points P
- Different ways to evaluate a triangulation
 - minimum angle
 - maximum degree
 - sum of edge lengths
 - ...
- A triangulation that maximizes the minimum angle across all triangles is called the Delaunay triangulation. It's known how to computed in O(n Ig n) time.
- Algorithms for various other kinds of optimal triangulations are known.

Greedy insertion with Delaunay triangulation

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with corners as vertices
- while not DONE() do
 - for each point p in P, compute error(p)
 - select point p with largest error(p)
 - insert p in P', delete p from P, and update TIN(P')

maintain TIN as a Delaunay triangulation of P'

Point-cloud-to-TIN





TIN

Brainstorming: Point-cloud-to-TIN ?



What needs to change?

Algorithm:

- P = {all grid points}, P' = {4 corner points}
- Initialize TIN to two triangles with corners as vertices
- while not DONE() do
 - for each point p in P, compute error(p)
 - select point p with largest error(p)
 - insert p in P', delete p from P, and update TIN(P')

Point-cloud-to-grid



California Lidar data

http://www.opentopography.org/images/opentopo_images/garlock_slope.jpg

Point-cloud-to-grid



Given a point-cloud P and a desired grid spacing, compute a grid that represents P.

Brainstorming: Point-cloud-to-grid ?



Given a point-cloud P and a desired grid spacing, compute a grid that represents P.

Brainstorming: Point-cloud-to-grid ?



Given a point-cloud P and a desired grid spacing, compute a grid that represents P.

Brainstorming: Point-cloud-to-grid ?



Sketch an algorithm to compute a grid given a point cloud and a desired resolution. Analyze it.