Computer Science Department
CPSC 097

Class of 2008
Senior Conference on
Computational Geometry

Proceedings of the Conference

Order copies of this proceedings from:

Computer Science Department
Swarthmore College
500 College Avenue
Swarthmore, PA 19081
USA
Tel: +1-610-328-8272
Fax: +1-610-328-8606
adanner@cs.swarthmore.edu

Introduction

About CPSC 097: Senior Conference

This course provides honors and course majors an opportunity to delve more deeply into a particular topic in computer science, synthesizing material from previous courses. Topics have included advanced algorithms, networking, evolutionary computation, complexity, encryption and compression, and parallel processing. CPSC 097 is the usual method used to satisfy the comprehensive requirement for a computer science major.

During the 2007-2008 academic year, the Senior Conference was led by Andrew Danner in the area of Computational Geometry.

Computer Science Department

Charles Kelemen, Edward Hicks Magill Professor and Chair
Lisa Meeden, Associate Professor
Tia Newhall, Associate Professor
Richard Wicentowski, Assistant Professor
Andrew Danner, Visiting Assistant Professor

Program Committee Members

Allison Barlow
Alex Benn
George Dahl
Scott Dalane
Kit La Touche
Andrew Frampton
David German
Michael Gorbach
Chris Harman
Mike Johns
Jeff Kaufman
Drew Perkins
David Rosen
Lucas Sanders
Megan Schuster
Bryce Wiedenbeck
Mary Wootters

Conference Website

`http://www.cs.swarthmore.edu/~adanner/cs97/s08/`

Conference Program

Thursday 24 April 2008

9:55–10:10 *Seamless Intersection Between Triangle Meshes*
David Rosen

10:14–10:29 *Approximate K Nearest Neighbors in High Dimensions*
George Dahl and Mary Wooters

10:33–10:48 *The Road Not Taken: Creating a Path-Finding Tool Using Consumer-Grade GPS Equipment*
Allison Barlow and Lucas Sanders

10:52–11:07 *Drawing Isoglosses Algorithmically*
Kit La Touche and Bryce Wiedenbeck

Tuesday 29 April 2008

9:55–10:10 *The Largest Empty Circle Problem*
Megan Schuster

10:14–10:29 *Unbiased Congressional Districts*
Alex Benn and David German

Thursday 1 May 2008

9:55–10:10 *Optimal Double Coverage in the Art Gallery*
Scott Dalane and Andrew Frampton

10:14–10:29 *Voronoi Natural Neighbors Interpolation*
Chris Harman and Mike Johns

10:33–10:48 *Bridge Detection by Road Detection*
Jeff Kaufman

10:52–11:07 *Image Stained Glass using Voronoi Diagrams*
Michael Gorbach

Table of Contents

Seamless Intersection Between Triangle Meshes

David Rosen
drosen2@swarthmore.edu

Abstract

We present an algorithm that provides artistic control of the rendering of intersections between two triangle meshes. We detect the intersection edges using an octree, and then seamlessly subdivide both meshes at and around the intersection edges to allow local geometry morphing and creation of transition texture bands.

1 Introduction

In computer graphics there has been a lot of progress on rendering techniques that use diffuse, height, and normal maps to make rendered surfaces look much more detailed than their underlying geometry. However, these techniques do nothing to break up the linear silhouette of each polygon (Figure 1). This is a very difficult problem to solve in general, because the silhouette of an object can change significantly every frame, so any algorithm that appropriately changes the silhouette would have to be extremely fast for real-time applications. There has been some progress in this area, but nothing that is really practical yet (Oliveira and Policarpo, 2005). However, this problem can be solved more easily where two static objects intersect because the intersection line is independent of camera position.

There are two obvious problems with the naive intersection. First, there is an immediate discontinuity in the lighting and texturing that is unlike what you see in such intersections in real-life (Figure 2). Second, this discontinuity occurs along perfectly straight lines, destroying the illusion of depth that the texture mapping is supposed to create.

Figure 1: Rocks on the beach in Crysis, with shadows disabled for clarity. Despite the detailed textures, there is an unrealistically sharp line between the big rocks and the terrain.

Figure 2: A photograph of a post intersecting the ground.

2 Seamless Intersection Algorithm

To make intersections more realistic, we must address both of these problems. First, we need to find

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 1–8,
Computer Science Department, Swarthmore College

the intersection between the objects and create vertices at each point of intersection. We can then find auxiliary intersections which we will discuss auxiliary intersections in more detail in section 2.2, and explain more clearly why they are important. Using our new intersection and auxiliary intersection vertices, we can apply geometric deformations around the intersections, and create texture bands to make them look more natural. None of these steps are trivial, so we will explore them one at a time.

2.1 Data Structures

First, we should look at the data structures we are using to represent the mesh. The mesh object contains an array of vertices and an array of triangles. Each vertex object contains several vectors: a 3-part vector storing its local coordinates, a 3-part vector storing its surface normal, and a 2-part vector storing its texture coordinates. Each triangle object contains three pointers, one to each of its vertices.

2.2 Finding Intersection Segments

To find the intersection of two meshes, we could check every triangle against every other triangle to see if there is an intersection. However, this would require $O(n^2)$ comparisons, which is not acceptable when working with detailed meshes. Fortunately, we can do much better using what I call an intersection octree (Figure 3), which indexes pairs of triangles that might intersect. Each node in the octree contains the coordinates of its bounding box and two lists of triangles (one for each mesh). We create the root of the octree by taking the intersection of the bounding boxes of each mesh, and adding all of the triangles from each mesh that at least partially lie within this shared bounding box. We then recursively subdivide each node until it reaches a maximum depth or no longer contains at least one triangle from each mesh. Finally, we walk through all of the child nodes and report each pair of triangles, and check each pair for intersection using well-known techniques (Moller, 1997), returning the intersection as a line segment. We have now efficiently found the intersection between two meshes as a set of line segments (Figure 4), which will form a closed loop if both meshes are closed.

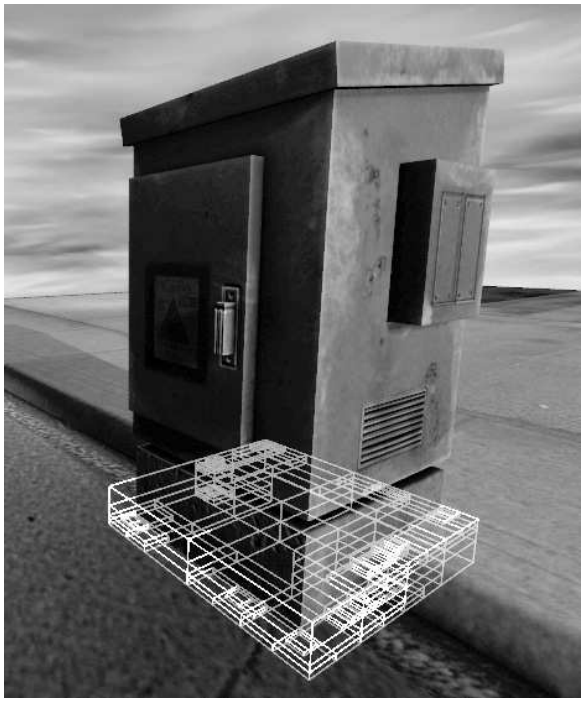

Figure 3: The intersection octree generated by a power transformer and a detailed sidewalk. The octree returned 70 pairs of triangles that might intersect, out of 846,000 possible pairs.

Figure 4: The intersection segments between a power transformer and a sidewalk.

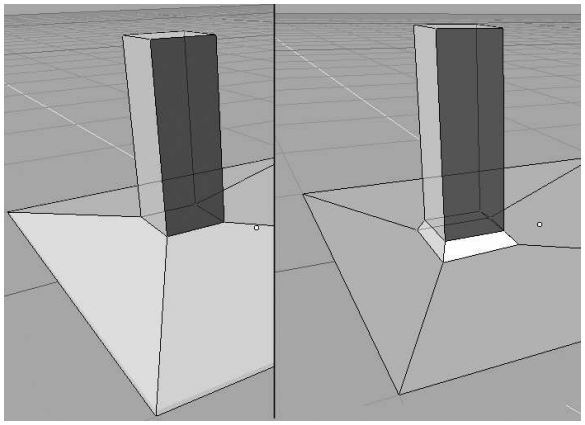

Figure 5: Without an auxiliary intersection (left), the deformation propagates in an uncontrolled way. With the auxiliary intersection (right), we have precise control over the shape of the deformation.

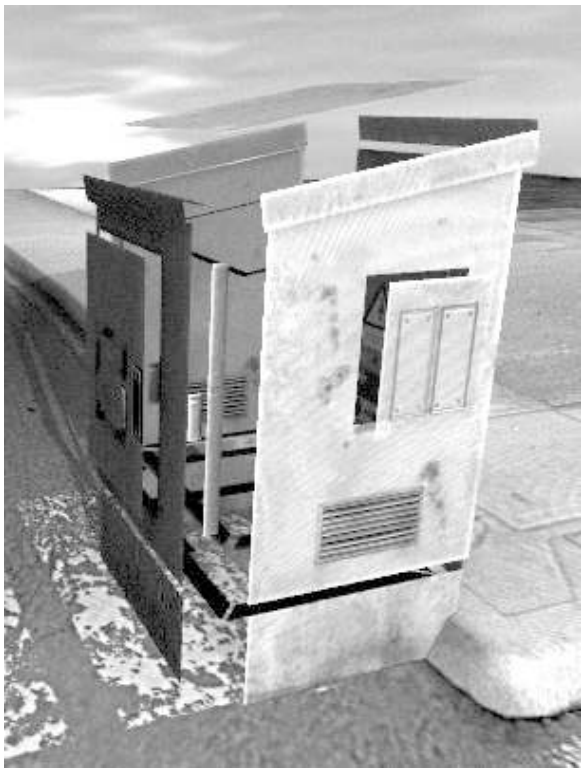

Figure 6: Scaling a model by moving vertices along their normals, resulting in discontinuities at sharp edges.

2.3 Finding Auxiliary Intersections

If we would like to change the geometry around the intersecting object, we will need to find auxiliary intersections to protect the environment geometry from these changes(Figure 5). For example, if a tree is intersecting the ground on a large, flat field represented by a single triangle, then altering the intersection points will alter the geometry of the entire field. If we want to create a small slope in the ground around the tree of some controlled size, say two inches, then we can find the auxiliary points by translating each vertex in the object out by two inches in the direction of the surface normal. However, for meshes with sharp edges, the surface normals of overlapping vertices can point in different directions, and moving each vertex along its surface normal could result in a discontinuous mesh (Figure 6). We can solve this by temporarily averaging together all normals belonging to overlapping vertices (Figure 7).

Figure 7: Scaling a model by moving vertices along their corrected normals. This results in some distortion, but no discontinuities

This can be done in $O(n \log n)$ time using a kd-tree (Bentley, 1975). We loop through each vertex in the model, and check if the tree is storing a vertex that has the same coordinates. If not, we add the vertex to the tree. If so, we store a pointer to the vertex that is already in the tree. Now for each group of overlapping vertices, we have a representative vertex in the tree, and every other overlapping vertex has a pointer to its representative. We find the average normal for each representative vertex by looping through the vertices again, adding each vertex's normal to its representative's normal, and then normalizing it by dividing it by its length. Finally, we can translate each vertex by some multiple of the normal of its representative vertex.

Figure 8 is an example of typical auxiliary intersections that we can use to isolate the effects of manipulation of intersection points. These will also be useful later to isolate texture transition effects, so we do not have to apply them to any unnecessarily large triangles (so we can avoid drawing too many unnecessary pixels). It is essential to find and store all of the lines of intersection, including auxiliary intersection, before the final retriangulation. Otherwise we will have to find intersections with the triangles that are already subdivided, and the algorithm will take longer and end up with many unnecessary subdivisions.

2.4 Dividing Polygons Along Intersection Lines

Now that we have our intersection segments, what do we do with them? We have to retriangulate our mesh to include these segments as edges, and we have to do it without any cosmetic changes. That is, the retriangulation itself should have no visible effect on the scene, but we can use our new vertices later to change the intersection however we like. This problem can be divided further into two subproblems: creating the new vertices, and retriangulating the mesh to include the new edges.

2.4.1 Seamlessly Adding Vertices

Suppose we have an intersection point on the edge of a triangle. How do we incorporate it into the mesh? We know the position component already, but our mesh vertices contain other auxiliary information, such as texture coordinates and surface normals. Since our vertex is on an edge, we can just

Figure 8: Some examples of auxiliary intersections

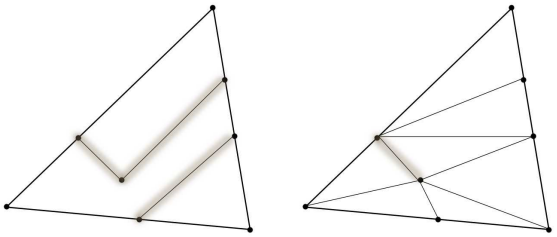

Figure 9: The Delaunay triangulation (right) does not necessarily preserve the lines of intersection (left).

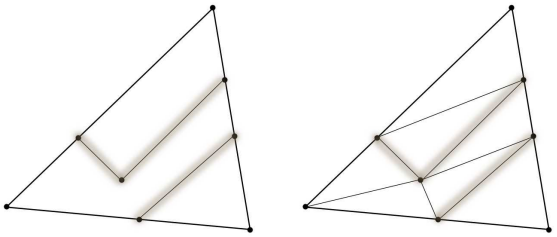

Figure 10: The constrained Delaunay triangulation (right) is guaranteed to preserve the lines of intersection (left).

interpolate between the auxiliary information stored in the two endpoint vertices based on their relative distance from the new point. But what if our intersection point is not on an edge? Suddenly the problem is much more complicated. We still have to interpolate between the three vertices, but it is not as obvious what weights to assign to each vertex in the interpolation.

The solution here is to find the position of the intersection point in barycentric coordinates. That is, find its position as a weighted sum of the positions of the three triangle vertices. So far this sounds somewhat circular: how do we find the barycentric coordinates? We know that the weighted sum of the triangle vertices in each axis adds up to our new vertex, and we know that the weights have to add up to 1. We have a system of four independent equations and three unknowns, so we can just solve it using linear algebra. Once we have the weights of each triangle vertex, we can use them to interpolate all of the auxiliary values of our new vertex.

2.4.2 Retriangulation

Now that we have our seamless vertices, how do we incorporate them into the triangulation? At first we expected that we could just use Delaunay triangulation on the set of intersection points and triangle vertices, but that failed because it did not preserve our lines of intersection (Figure 9). To achieve the desired results, we had to use the constrained Delaunay triangulation, which takes as input a set of points as well as a set of line segments to preserve(Figure 10). Implementing the constrained Delaunay triangulation efficiently could easily be a final project in itself, so we used the free "Triangle" library (Shewchuk, 2002).

2.5 Geometry Morphing

To deform the geometry, we can now safely translate any of intersection vertices that do not lie on the outermost auxiliary intersection ring. After moving any of the vertices, we have to recalculate the normals of all of the adjacent triangles so that the lighting will be correct for their new orientation. With one intersection ring and two auxiliary intersection rings,

Figure 11: Translating upwards the intersection ring and inner auxiliary intersection ring.

Figure 12: Translating upwards only the inner auxiliary intersection ring.

we can create a number of interesting effects. We can raise the intersection ring and the inner auxiliary ring to slope the ground around the object (Figure 11), or we can only raise the inner auxiliary ring to create a crater effect (Figure 12).

2.6 Texture Band

Our final step is to create a texture band around the intersection. First we will have to create a new mesh that contains a copy of all the triangles and vertices that we will need, and then we will have to assign texture coordinates to each vertex in the mesh. To create the new mesh, we can make a copy of each triangle that contains at least one of the intersection points, and make a copy of each of its vertices. If it fits the effect we are going for, we can now recalculate the normals to smooth the lighting transition at the intersection. Alternately, we can leave them how they are to keep the sharp edge.

Finally we have to calculate the texture coordinates. We would like to map the texture in Figure 13 to achieve the effect shown in Figure 14 and Figure 15. The vertical texture coordinate can be assigned quite simply. Vertices on the intersection ring receive the vertical texture coordinate of 0.5, so that it is lined up exactly with the center of the texture. Vertices on the innermost auxiliary intersection ring of one mesh receive the vertical texture coordinate of 1.0, and of the other mesh, 0.0. This stretches the texture over the intersection such that the middle of the texture corresponds to the intersection ring, and the top and bottom stretch out to the innermost auxiliary ring of each mesh.

The horizontal texture coordinates are a bit more tricky. We have two priorities here: we would like the texture to be mapped with uniform density, and we would like it to wrap around seamlessly. Mapping it with uniform density means that it is never stretched or compressed. That is, the difference in texture coordinate between two connected vertices is proportional to the physical distance between them. To satisfy this constraint, we can start at any intersection point and assign it a horizontal texture coordinate of 0.0, and walk around the intersection, and assign to each point a texture coordinate equal to the total distance walked so far.

To satisfy the second constraint (seamless texture wrapping), we have to round up to the nearest integer the total distance. That is, if the total distance walked is 1.58 units, we have to round it to 2.0 units, and scale all of the other texture coordinates similarly. We do this because the right edge of the texture image lines up seamlessly with the left edge of the texture image, and the only way that these can line up on the texture band is if the image is repeated an integral number of times. If the texture is repeated 1.58 times, then there will be a visible seam where the texture coordinate wraps from 1.58 back to 0.0.

Now that we have the horizontal texture coordinates for the intersection points, we need horizontal texture coordinates for the remaining points. To find these we simply loop through all of the remaining points and find the nearest intersection point, and copy its horizontal texture coordinate. We can do

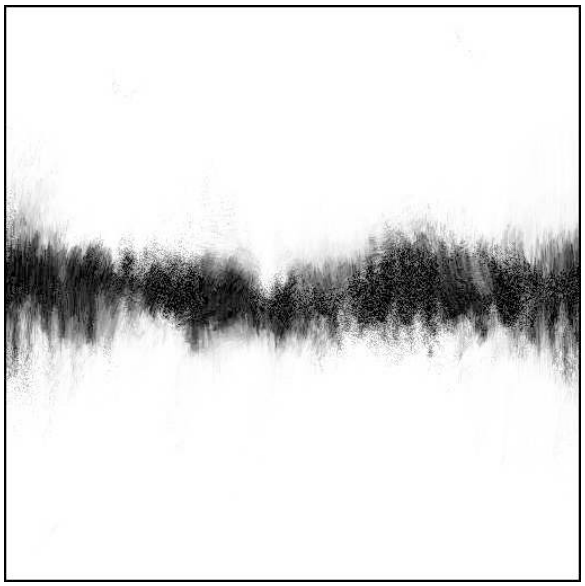

Figure 13: A simple dirt texture that can be used for texture bands.

Figure 14: A subtle dirt texture (right) around the base of an object.

this in $O(n)$ time instead of $O(n^2)$ by only looking at points that are connected by a triangle edge.

3 Results

This algorithm allows for artistic control of mesh intersections, from subtle weathering effects (Figure 14) to dramatic impact deformations (Figure 12). It accomplishes this in $O(n \log n)$ time. In this prototype implementation it takes about 0.2 seconds to create the intersections for the simple power transformers (400 triangles), and about 1.6 seconds to create the intersections for the detailed boulder (2500 triangles).

The intersection rings and retriangulation are robust to degenerate cases, but the deformation and texture band creation can give unexpected results in some cases. For example, the current deformation algorithm will only affect vertices that are on one of the intersection rings, so if there are other vertices within the deformed area, there can be unexpected indentations. Similarly, the texture bands stretch out to the nearest vertices connected to the intersection rings, which may not be the auxiliary intersection rings if there are already vertices nearby.

Figure 15: A large dirt texture around the base of an object.

4 Discussion and Future Work

There are many applications for this kind of technology. Two of the most significant applications are aging effects (e.g. rust accumulation around the base of nails) or ambient occlusion (obstruction of ambient light when two surfaces are close together). The algorithm takes a fraction of a second to run, so with some optimization it will be useful for dynamic heightmaps, such as water surfaces, and dynamic environmental damage from explosions and heavy impacts.

In the future we would like to make the deformation and texture bands more predictable in degenerate cases, and optimize the algorithm to run faster. We would also like to create a user-friendly interface that allows artists to easily create and test new kinds of intersection effects.

5 Conclusion

Intersections between detailed meshes have been a serious problem in modern 3D graphics. We have created an algorithm that can automatically create deformations or texture bands that can greatly improve the appearance of these intersections. As far as we know, there has been no other work addressing this issue, so this is an important new step towards efficiently creating realistic 3D environments.

References

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517.

Tomas Moller. 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30.

Manuel M. Oliveira and Fabio Policarpo. 2005. An efficient representation for surface details. *UFRGS Technical Report*, RP-351.

Jonathan Richard Shewchuk. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational Geometry: Theory and Applications*, 22(1-3):21–74.

Approximate K Nearest Neighbors in High Dimensions

George Dahl
gdahl@cs.swarthmore.edu

Mary Wootters
mwootte1@cs.swarthmore.edu

Abstract

Given a set P of N points in a d-dimensional space, along with a query point q, it is often desirable to find k points of P that are with high probability close to q. This is the Approximate k-Nearest-Neighbors problem. We present two algorithms for AkNN. Both require $O(N^2 d)$ preprocessing time. The first algorithm has a query time cost that is $O(d+\log N)$, while the second has a query time cost that is $O(d)$. Both algorithms create an undirected graph on the points of P by adding edges to a linked list storing P in Hilbert order. To find approximate nearest neighbors of a query point, both algorithms perform best-first search on this graph. The first algorithm uses standard one dimensional indexing structures to find starting points on the graph for this search, whereas the second algorithm using random starting points. Despite the quadratic preprocessing time, our algorithms have the potential to be useful in machine learning applications where the number of query points that need to be processed is large compared to the number of points in P. The linear dependence in d of the preprocessing and query time costs of our algorithms allows them to remain effective even when dealing with high-dimensional data.

1 The Problem

The K-Nearest Neighbors problem is the following: given a set P of N points in a $d-$dimensional space and a query point q, return the k points in P that are closest to q.

However, solving K-Nearest-Neighbors in high dimensions (say, more than 10) has proved computationally infeasible - most solutions are not much better than the naïve method. Thus, we consider the Approximate K-Nearest Neighbors problem: given a set P of N points in a $d-$dimensional space, a query point q, and parameters ϵ and δ between 0 and 1, return, with probability greater than $1-\delta$, k points of P such that the i^{th} point is at most $(1+\epsilon)$ times farther from q than the true i^{th} nearest neighbor of q (Arya et al., 1994).

Approximate K-Nearest Neighbors is widely applicable, but we are motivated by its application to supervised machine learning. Machine learning applications are often characterized by a data set of relatively high dimensionality, so we are interested in solutions that scale well with d. In a typical supervised learning scenario, a training set is processed offline, and later the system must be able to quickly answer a stream of previously unknown queries. Our assumption is that the number of queries will be large compared to N, which is why we are more concerned with query time than preprocessing time. Many supervised machine learning techniques that could be alternatives to K-Nearest Neighbors have quadratic or cubic (in N) training time. To this end, our goal is to make query time as fast as possible, and accept almost any reasonable preprocessing cost (quadratic in N or better). Since the naïve algorithm has a query time complexity of $O(Nd)$, we demand a solution that provides query times sublinear in N and linear in d.

2 Related Work

Recent solutions to K-Nearest Neighbors that we have found tend to fall into two categories: ones that employ locality sensitive hashing (LSH) (Andoni and Indyk, 2008; Gionis et al., 1999) or ones that use sophisticated tree-like structures to do spatial partitioning (Liu et al., 2004; Arya and Mount, 1993; Arya et al., 1994; Beckmann et al., 1990; Berchtold et al., 1996). LSH defines a hash function on the query

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 9–17,
Computer Science Department, Swarthmore College

space which has a collision probability that increases as the distance between two points decreases. In general, LSH approaches scale reasonably well with d, while the tree-based algorithms tend not to scale as well. Most notably, Arya et al. (1994) present an algorithm which, for fixed d, has a preprocessing cost of $O(N \log N)$ and a query time of $O(\log N)$, but is exponential in d. There are results which scale well with d and have fast query time. In particular, Kleinberg (1997) presents an algorithm with query time $O(N + d \log^3 N)$, and preprocessing cost quadratic in d, linear in N, and is $O(1/\log(\delta))$ in δ. Andoni and Indyk (2008) use LSH to achieve a query time of $O(N^{1/c^2+o(1)}d)$, and pre-processing cost of $O(N^{1+1/c^2+o(1)}d)$, where $c = (1+\epsilon)$.

3 Algorithm 1

Our two algorithms are similar. We will describe the first in its entirety, and then describe the changes we make to produce the second one.

3.1 Overview

`Algorithm 1` creates several auxiliary data structures to speed up query processing. The most important of these index structures is a graph, G, that holds all the N points in P. To create G, we first create a linked list containing all the points in P that is sorted in the Hilbert order. Then we add edges to the graph by linking points that are close together. The goal is to create a connected graph (starting with a linked list ensures connectedness) in which two points that are close in space will be very likely to be close on the graph. We also construct a small, constant number of one-dimensional search structures, specifically red-black trees, that order points based on their projections onto one-dimensional subspaces of our space. Given a query point q, the one dimensional search structures are used to obtain a small set of initial guess nodes in G. These are the nodes corresponding to points whose projections are close to the projection of q in the one-dimensional subspaces. Starting at these guess points, our algorithm searches G until $k+m$ nodes have been touched, for some constant m (assuming $k+m$ is greater than the number of guess points, otherwise we touch each guess point once). The nodes are sorted by their distance to q, and the first k are returned.

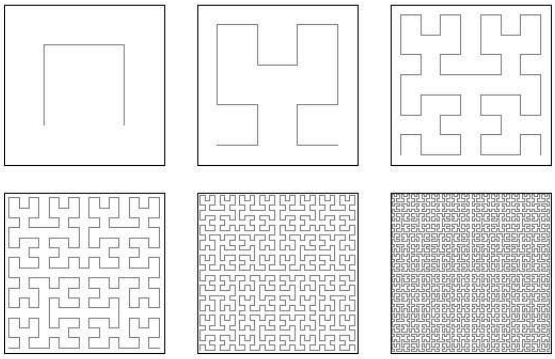

Figure 1: The first six curves in the sequence limiting to the Hilbert curve. Picture courtesy of Wikimedia.

3.2 Preprocessing and Auxiliary Data Structures

`Algorithm 1` requires several auxialiary data structures, as mentioned above. It uses a graph G on nodes corresponding to points in P, and several red-black trees. In order to create G, we first compute the Hilbert order of the points in P.

3.2.1 Snapping Points to an Integer Lattice and Sorting by Hilbert Order

The Hilbert curve is a space-filling curve defined as the limit of self-similar curves in a d-dimensional space. The first few curves in the 2-dimensional sequence are shown in Figure 1. One of the more celebrated properties of the Hilbert curve (Jagadish, 1990) is that it preserves locality well. That is, if two points are close in the d-dimensional space, their preimages on the unit interval will likely be close. Each curve in the sequence touches every point of a d-dimensional lattice with 2^n points on each side for some n. The Hilbert order of a set of points P on such a lattice is the order of the preimages of the points in the unit interval. We impose a Hilbert order on our set of points P by snapping them to such a lattice first. We compute the location in the lattice for a point by applying the following function to it:

$$f(\vec{x}) = \lceil a\vec{x} \rceil + \vec{b},$$

where

$$\frac{1}{a} = \min_{\vec{p},\vec{q} \in P} \left(\min_{i \leq d} |p_i - q_i| \right)$$

and

$$b_i = -\min_{\vec{p} \in P} \lceil p_i \rceil,$$

where x_i denotes the i^{th} component of $\vec{x}$. That is, the smallest distance along any axis between any two

points becomes the lattice spacing. Such a potentially small lattice spacing could be undesirable because computing the Hilbert order might take too long. In practice, we have not found this to be a problem, but if it were, a coarser lattice could be used. Once the points are on a lattice, we compute the Hilbert order using an algorithm developed by Butz (1971), and explained to us by Jaffer (2008b). Our implementation is based on the implementation of `hilbert->int` in the SLIB library (Jaffer, 2008a).

3.2.2 Additional Graph Edges

The graph G begins as a linked list of the points of P in the Hilbert order as described above. For clarity, we will refer to nodes in G by the point they correspond to in P. Edges are strategically added as follows: each node p in G is linked to the b nodes closest to p in space, for some constant b. If p's b nearest neighbors are already adjacent to it (perhaps they were in the original linked list or they themselves have already been processed), these edges are not added again. This guarantees that each node of G will be adjacent to its b nearest neighbors. These nearest neighbors are computed using the naïve method, i.e., simply scanning all the points in P.

3.2.3 One-Dimensional Search Structures

In order to keep preprocessing costs low, we choose a subset P' of P consisting of $N^{2/3}$ points randomly selected from P. For some constant c, suppose the first c principal components of P' are $\{a_1, \ldots, a_c\}$. For each principal component a_i, we create a red-black tree T_i holding the elements of P ordered by where they fall along a_i.

3.3 Handling Queries

Given a query point q, we search each one-dimensional search structure T_i for the point p_i whose projection onto a_i is the closest to the projection of q onto a_i. These p_i are the c initial nodes for the search of G.

The search proceeds in a best-first manner by preferentially expanding nodes closer to q. If n is a node in G, let $d(n)$ denote the distance from n to q. Two priority queues are maintained during the search, *ToExpand* and *BestK*. We initialize *ToExpand* to contain the nodes p_i. *ToExpand* is a minheap with the priority of node n being $d(n)$. We initialize *BestK* to be empty. *BestK* is a maxheap, such that the highest priority node l maximizes $d(l)$. For $m + k$ steps, a node n is removed from *ToExpand*. If $d(n) > d(l)$ for the node l with the highest priority in *BestK,* then n is added onto *BestK* and l is removed (assuming *BestK* contains k items). Then all of the nodes adjacent to n are added to *ToExpand*. After $m + k$ steps, the k nodes in *BestK* are returned.

3.4 Cost Analysis

In order to compute the Hilbert order, we map each point in P to its distance from the origin along the appropriate Hilbert curve. This computation is $O(d + \log s)$, where s is a times the maximum coordinate of p, where a is the factor from our lattice snapping. Although we cannot control the maximum coordinate or a, we find that in practice, at least, we can compute the Hilbert order very quickly in hundreds of dimensions. We could theoretically control these variables by creating a coarser lattice, which might result in an approximation of the Hilbert order. We are convinced that this is not a problem. We assume that either s is reasonable or we force it to be so by adjusting the lattice, so this step should take time approximately linear in Nd.

We can complete the preprocessing phase of our algorithm in $O(N^2 d)$ time. Once points are in a linked list, for each point in P we add at most b additional edges. The new edges can be computed for a given point with a single scan of the points in P which will require $O(N)$ distance computations which each take time linear in d. Therefore we can construct the graph in $O(N^2 d)$ time. Computing the principal components of a subset of P of cardinality $N^{2/3}$ can be done in time quadratic in N since Principal Component Analysis can be performed in time cubic in the size of the dataset. The search trees can easily be constructed in $O(N \log N)$ time, so our preprocessing phase can be completed in $O(N^2 d)$ time. The space requirements of our auxiliary data structures are clearly linear in N.

To evaluate a query, we need to do a constant number of searches of red-black trees on N nodes which will have a cost logarithmic in N. We also have to project the query point into one dimension which adds a term proportional to d to our cost. In the best-first search of the graph we search a constant number of nodes and do a constant number of distance computations in d dimensions. Thus our query time is $O(d + \log N)$.

4 Algorithm 2

`Algorithm 2` is a simplification of `Algorithm 1`. Initial experiments suggested that the one-

dimensional search structures used in `Algorithm 1` were not that important to performance. Since the 1D search structures added a term proportional to $\log N$ to our query time, we designed a new algorithm that does not use them. In `Algorithm 1`, the red-black trees were used to get starting points for the best-first search. In `Algorithm 2`, we simply pick c random start points for this search. However, these start points will almost certainly be worse guesses than those produced by the red-black trees. Since we only expand a small constant number of nodes in our best-first search and since all of the edges in G connect points that are close in space, the search will expand many nodes that are far from the query point. Our solution is to add some longer edges to G. In the preprocessing phase of `Algorithm 2`, for each node in G, we add an edge from that node to one other random node (if the edge does not already exist). On average, these edges will be much longer than the nearest-neighbor edges.

4.1 Cost Analysis

The preprocessing phase of `Algorithm 2` is identical to the preprocessing phase of `Algorithm 1`, except

1. PCA is not performed.
2. Red-black trees are not constructed.
3. We add up to N random edges.

Adding the N random edges requires a single scan of the nodes in G, therefore our preprocessing time is still $O(N^2 d)$.

The term proportional to $\log N$ in the query time of `Algorithm 1` resulted from querying the red-black trees. `Algorithm 2` does not do this, so the query time for `Algorithm 2` is $O(d)$.

5 Experiments

Because we do not have proofs about the relationships between b, c, m, ϵ, and δ, we ran extensive empirical tests of our algorithms.

5.1 Datasets

We tested our algorithms on several large high dimensional data sets, both synthetic and real. Our synthetic data consists of mixtures of multivariate Gaussian distributions. Covariance matrices and means were generated randomly to produce these distributions. In particular, we considered unimodal, bimodal, pentamodal, and dodecamodal synthetic distributions. Gaussian distributions were chosen both because they often approximate distributions found in the wild, and because given a finite variance, the Gaussian distribution achieves the maximum entropy. We predicted that our algorithms would perform best on data with fewer modes, so the higher modal[1] distributions were selected to challenge our algorithms. In the case of the synthetic data, query points were drawn from the same distribution used to create the data. All of the synthetic data sets were 50-dimensional and contained 3000 points.

Real-world data was obtained from the UCI Machine Learning Repository (Asuncion and Newman, 2007). We used the ISOLET data set, which is audio data of spoken letters of the alphebet, and the Waveform Database Generator dataset. The ISOLET data set has 617 dimensions and more than 5000 points. The waveform data set has 22 dimensions and also more than 5000 points. In the case of the real data, the data sets were split into two parts, one for the initial set of points, and one from which to draw queries.

5.2 Parameters and Measurements

For each data set tested, the independent variables were:

- b: The number of nearest neighbors each point in P is connected to in G.
- c: The number of one-dimensional search structures created in the case of `Algorithm 1`, or the number of guess points in the case of `Algorithm 2`.
- m: The number of nodes (beyond k) in the graph that are expanded.
- k: The number of nearest neighbors requested.

The variables measured were:

- Percent Correct: The percent of the points returned which are actually among the k nearest neighbors.
- Excess Rank: The actual rank (that is, the j of "j^{th}-nearest neighbor") of the worst point returned, minus k.
- Maximum Epsilon: If the actual i^{th} nearest neighbor of a query point q has distance d_i from q, and the i^{th} approximate nearest neighbor has distance d'_i from q, then Max Epsilon=

[1] With more extreme modality/modacitude, as it were.

$\max_i (d'_i/d_i - 1)$. Note that this is an upper bound on the ϵ from the definition of AkNN, if $\delta = 1$.[2]

Based on preliminary experiments, the parameters b, c, m, and k were allowed to vary in the ranges below.

- $b \in \{0, 1, \ldots, 10\}$
- $c \in \{1, 4, 16\}$
- $m \in \{0, 1, 10, 30, 60, 100, 150, 200, 250\}$ in `Algorithm 2`. For `Algorithm 1`, we ommitted 200 and 250 because m had less of an impact.
- $k \in \{100\}$

We ran preliminary experiments using a variety of values for k, but settled on $k = 100$. The relationships between parameters are easier to determine for larger k, since the algorithms are identifying more neighbors, and so setting $k = 100$ is more enlightening than looking at smaller values of k. For each combination of parameters, 50 queries were made, and the average value for each dependent variable was recorded. An implementation of the naïve exact kNN algorithm determined the accuracy of the approximate nearest neighbors. We ran experiments using the above parameters on all of our synthetic data sets.

We tested how each parameter affected our performance metrics (Percent Correct, Maximum Epsilon, Excess Rank) when the other parameters were small and fixed. We fixed two of $c = 4$, $m = 10$, $b = 4$, varying the third.

On the real-world data, we picked reasonable values for all parameters and did not vary them. The chosen parameter values were $c = 4$, $b = 4$, $m = 100$, $k = 100$.

In the results presented below, the graphs for Excess Rank exactly followed the graphs for Maximum Epsilon, so we omit them.

6 Results

Unsurprisingly, b is an important parameter for both algorithms. Figure 2 shows that for `Algorithm 1`, if c and m are small, b can easily pick up the slack. For $b > 4$, nearly all of the nearest neighbors are guessed correctly. As shown in Figure 3, for distributions with few modes, using a large enough b ensures

[2]Further note that this is not generally how the experimental ϵ is computed for AkNN.

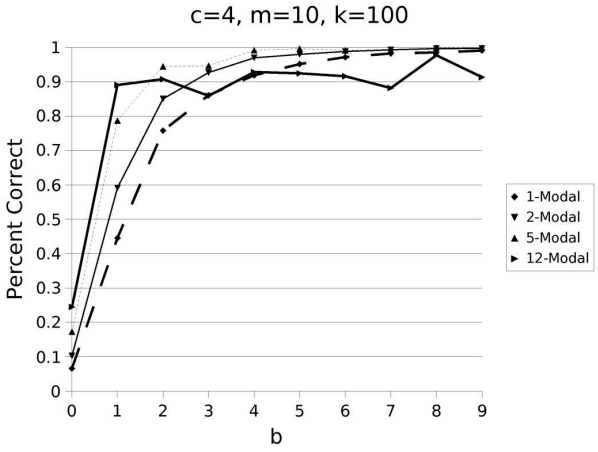

Figure 2: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$, nearly 100% of the nearest neighbors are correctly identified.

that Maximum Epsilon is close to zero. Unfortunately, when we run `Algorithm 1` on distributions with more modes, Maximum Epsilon is not as close to zero.

As can be seen in Figure 4, `Algorithm 2` scales with b in the same way as `Algorithm 1` does. A high b guarantees a good maxEpsilon for the unimodal and bimodal cases, but the situation is worse for the 5 and 12−modal cases. This is because incorrect neighbors were sometimes drawn from a Gaussian in the mixture that was far away.

However, there is a difference in the Percent Correct achieved by our two algorithms as a function of b. As can be seen in Figure 5, while the general relationship between Percent Correct and b on a single distribution is the same for both algorithms, the distributions that are easier for `Algorithm 1` to handle are not the distributions that are easier for `Algorithm 2` to handle. While `Algorithm 1` had a lower Percent Correct on the 12-modal distribution, even for larger b, `Algorithm 2` appears to behave in the opposite way. For `Algorithm 2`, Percent Correct is highest for the 12-modal distribution for pretty much all b. In all cases, a choice of $b > 4$ still guaranteed a high Percent Correct.

Increasing m improves Percent Correct and Max Epsilon. However, over the ranges we tested, b has more of an impact than m on the performance of `Algorithm 1`. Figures 6 and 7 demonstrate this effect nicely. As we have come to expect for `Algorithm 1`, the 12-modal distribution produces the worst performance. `Algorithm 2` benefits even more than `Algorithm 1` from increased m. Figures 8 and 9

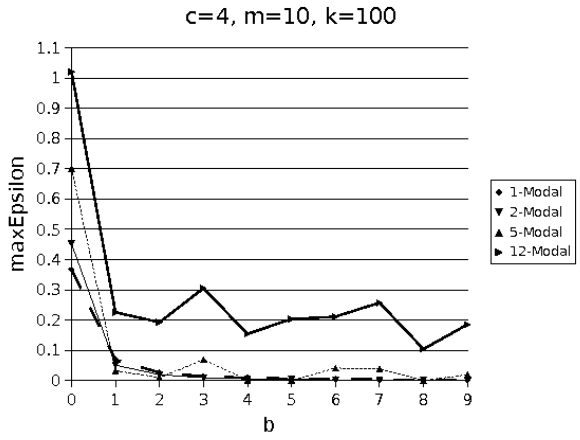

Figure 3: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$ and for data sets with few modes, the points which are not correct are not far from the points which are correct. For data sets with many modes, this is less true.

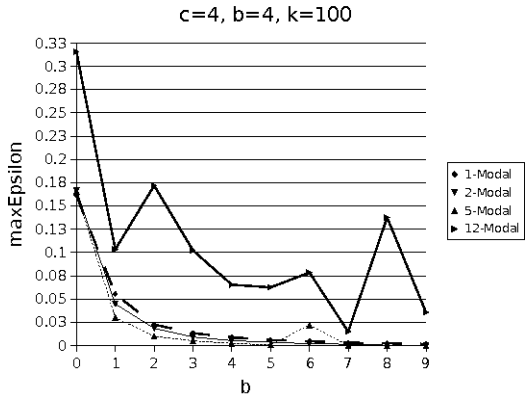

Figure 4: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$ and for data sets with few modes, the points that are not correct are not far from the correct points. For data sets with more modes, this is less ture.

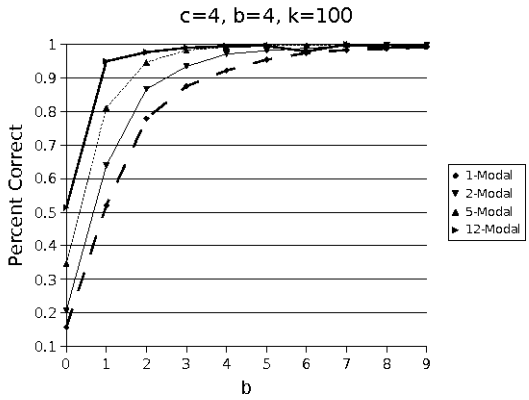

Figure 5: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. For $b > 4$, nearly 100% of the nearest neighbors are correctly identified. Surprisingly, `Algorithm 2` does better on more complicated distributions.

demonstrate that when we run `Algorithm 2` with increasing m on any of our synthetic data sets, Percent Correct rapidly approaches 1 and Max Epsilon rapidly approaches 0. The increased impact of m of `Algorithm 2` makes sense because `Algorithm 2` partly depends on a more extensive search to make up for its random starting points.

For both algorithms, over the ranges that we tested, c had less of an impact on performance than m or b. In particular, the Percent Correct for `Algorithm 2` was almost independent of c. It should be noted that while c represents the number of initial points for the search in both algorithms, these points are obtained in completely different ways. Thus, we do not gain much insight by comparing the effects on `Algorithm 1` and `Algorithm 2` of varying c. At some point, changing c would impact performance, but we are content to find a single reasonable value for c and focus on the other more important parameters.

Due to time constraints, we did not test `Algorithm 2` on our real world data sets. However, `Algorithm 1` performed admirably, especially considering the large number of dimensions (617) in the ISOLET data set. We have no reason to believe that `Algorithm 2` would not be as good or better than `Algorithm 1` on the real world data. Our experiments on synthetic data sets suggested that $m = 100$, $b = 4$ and $c = 4$ would be reasonable parameter settings that should work on any task. The results of running `Algorithm 1` with these parameters on all of our data sets are shown in Table 1. `Algorithm 1` returned more than 90% of the correct k nearest

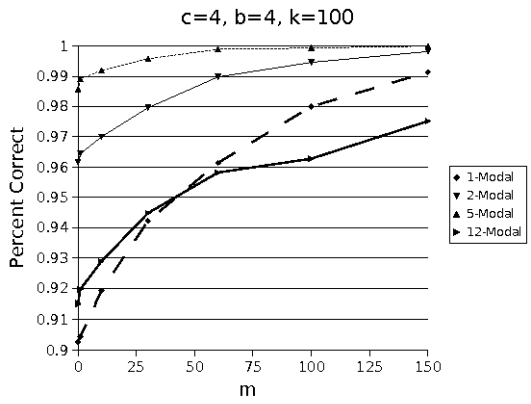

Figure 6: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. As we would hope, Percent Correct increases with m.

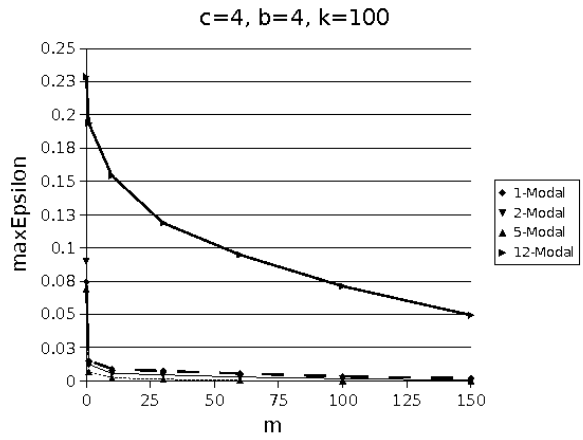

Figure 7: `Algorithm 1` on synthetic data sets with 3000 points in 50 dimensions. m seems relatively effective for data sets with few modes. For data sets with many modes, this is less true.

	PC	ME	ER
1-Modal	0.919	0.009	9.83
2-Modal	0.970	0.005	3.22
5-Modal	0.992	0.002	0.82
12-Modal	0.929	0.154	58.86
ISOLET	0.922	0.042	33.93
Wave	0.952	0.009	5.55

Table 1: Average Percent Correct (PC), Max Epsilon (ME), and Excess Rank (ER) over 100 queries on real world data (ISOLET and Wave) and over 50 queries on the synthetic data

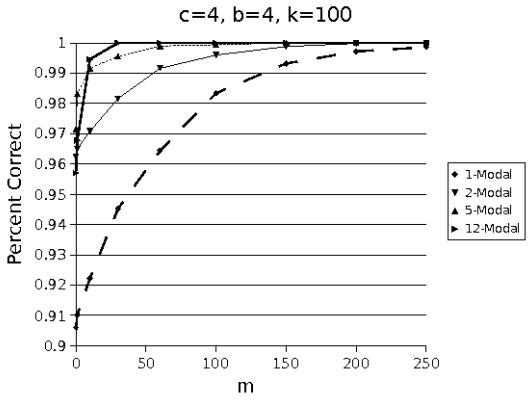

Figure 8: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. As we would hope, Percent Correct increases with m, and more so than in Figure 6.

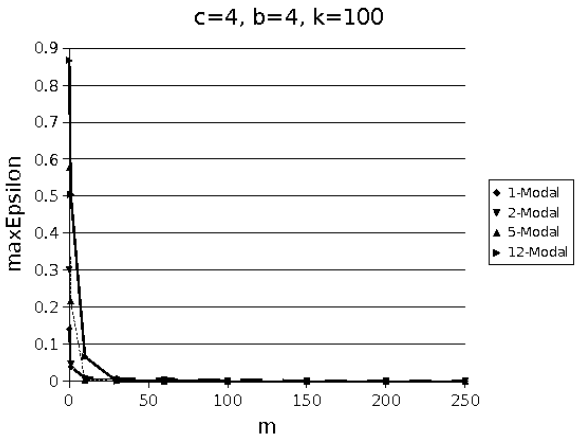

Figure 9: `Algorithm 2` on synthetic data sets with 3000 points in 50 dimensions. m seems relatively effective for data sets with any number of modes.

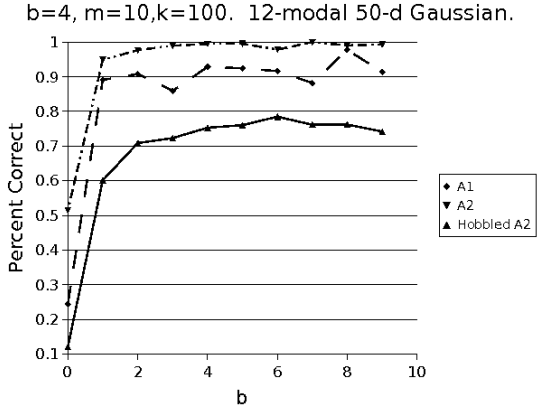

Figure 10: Both algorithms on the 12-modal synthetic data sets with 3000 points in 50 dimensions, along with a new algorithm `Hobbled Algorithm 2`. This algorithm is the same as `Algorithm 2`, except no random edges are added.

neighbors of the query points on each data set. As we have come to expect, `Algorithm 1` performed worse on the 12-modal data set. The ISOLET data set was also difficult, as presumably its intrinsic dimensionality was much larger that the dimensionality of the synthetic data.

6.1 Comparison of Algorithms

While `Algorithm 2` may sacrifice quality in its initial guesses when compared to `Algorithm 1`, it also has a more complete graph to search. Comparing the performance of both algorithms on our synthetic data sets, we found that `Algorithm 2` tended to outperform `Algorithm 1` on the 12-modal set, and that there was no discernable pattern on the other sets. One might wonder whether or not using the one dimensional search structures helped `Algorithm 1` at all, or whether the additional random edges helped `Algorithm 2`. Figure 10 demonstrates that the answer to both questions is yes. While `Algorithm 2` does slightly better than `Algorithm 1`, both do much better than `Hobbled Algorithm 2`, which is the same as `Algorithm 2`, except no random edges are added.

7 Conclusions and Future Work

The adjustments we made to `Algorithm 1` to obtain `Algorithm 2` suggest a third algorithm with $O(Nd)$ preprocessing time and $O(d)$ query time that might be interesting to try in future work. This algorithm, tentatively called `Algorithm 3`, would start with the same linked list as `Algorithms 1` and `2`, add b random edges to each node, and process queries as in `Algorithm 2`. Dispensing with the nearest neighbor edges would give us the faster preprocessing time. It would be interesting to see how accurate `Algorithm 3` would be.

We have presented two algorithms which, though $O(N^2d)$ in preprocessing cost, handle queries in $O(d + logN)$ and $O(d)$ time, respectively, with experimentally good accuracy. As they are only linear in d, our algorithms scale well to the high-dimensional problems common in machine learning. Furthermore, our algorithms do not appear to be overly sensitive to parameter settings - choices of, say, $m = 100$, $b = 4$, and $c = 4$, seem to be sufficient to get good accuracy on all the data sets we tried. Our second, faster algorithm seems to do as well or better than our first algorithm, and its performance seems depend even less on the data distribution. Since `Algorithm 2` is faster and seems to be more robust, it should be preferred in general. Ultimately, our second algorithm is an attractive choice for solving Approximate K Nearest Neighbors in high dimensions.

References

Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122.

Arya and Mount. 1993. Approximate nearest neighbor queries in fixed dimensions. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*.

Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Wu. 1994. An optimal algorithm for approximate nearest neighbor searching. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 573–582, Philadelphia, PA, USA.

A. Asuncion and D.J. Newman. 2007. UCI machine learning repository.

Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331.

Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. 1996. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman,

Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A. Morgan Kaufmann Publishers.

A. R. Butz. 1971. Alternative algorithm for hilbert's space-filling curve. *IEEE Trans. Computers*, C-20:424–426.

Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 1999. Similarity search in high dimensions via hashing. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Jaffer. 2008a. The SLIB Portable SCHEME Library, available at http://swissnet.ai.mit.edu/ jaffer/SLIB.html.

Aubrey Jaffer. 2008b. Personal Communication.

H. V. Jagadish. 1990. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2):332–342.

Jon M. Kleinberg. 1997. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 599–608.

T. Liu, A. Moore, A. Gray, and K. Yang. 2004. An investigation of practical approximate nearest neighbor algorithms.

The Road Not Taken: Creating a Path-Finding Tool Using Consumer-Grade GPS Equipment

Allison Barlow
ajb@sccs.swarthmore.edu

Lucas Sanders
lsanders@sccs.swarthmore.edu

Abstract

We automate the organization of GPS data to create a visibility map with correctly connected intersections. For our example case study, we use GPS data collected on the Swarthmore College campus. After the data is automatically organized and cleaned, a user is able to select points on a visual map to search for an optimal path between those points.

1 The Problem of Navigation

Swarthmore College, set in the beautiful Scott Arboretum, has many scenic routes and winding paths between the lovely stone buildings. This serene setting is wonderful for long walks, but can be troublesome in trying to navigate from one's dorm to a classroom when one is half-awake and running behind schedule. In order to assist the poor, sleep-deprived students of Swarthmore College, we create an interactive map of the campus to help users discover the appropriate paths between the various buildings on campus. In doing so, we develop a set of tools that can easily create similar systems for data collected in other locations.

1.1 Previous Work

Another group worked on a GPS path-planning project for last year's Senior Conference (Singleton and Woods, 2007). Because their data required quite a bit of manual editing, their system would not easily scale to larger data sets. We also noticed that their interface was difficult to understand immediately. Taking these challenges into account, we create a scalable, user-friendly path planning tool.

We used a large amount of previous research to build our tools. Most notably, Dijkstra's algorithm (Dijkstra, 1959; de Berg et al., 1997) and the more general A* search algorithm (Hart et al., 1968) have been developed to find appropriate paths through our spatial graph. We also implement an improved version of the Douglas-Peucker line simplification algorithm (Hart et al., 1968).

1.2 Approaches to Path Planning

Path planning is essentially a least-cost graph searching problem, a task for which several algorithmic variations have been developed. In the past, Dijkstra's algorithm has been widely used for this task, but we implement the A* search algorithm, which is a generalization of Dijkstra's approach that is both complete and optimally efficient. In typical situations, A* performs slightly faster than Dijkstra's algorithm because it uses a heuristic to help decide which search paths are most promising. This process minimizes the size of the subgraph to be explored.

2 Implementation

This project consists of three main parts: data collection and processing, path planning, and UI development. The first two parts center on problem solving using computational geometry while the third provides easy access to the results.

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 18–21,
Computer Science Department, Swarthmore College

2.1 Data Collection and Processing

We use self-collected GPS data for our finished mapping system. Swarthmore College kindly provided their survey plans for use with this project, but due to a lack of time, we were not able to complete an integration of this data into our user interface. We prioritized our work with the GPS data, even though that data is not as precise, because we wanted our system to be useful in creating mapping systems in other situations where no such detailed survey plans have been prepared.

2.1.1 GPS Data Collection

Our main dataset is GPS tracking data recorded by walking the paths of Swarthmore's campus with a consumer-grade GPS receiver, the Garmin GPSmap 60CSx. For consistency, we carried the GPS receiver at about waist-height and walked on the center of each footpath, recording tracks of where we had walked and marking a waypoint each time we encountered a path intersection. We used GPSBabel to transfer data from the Garmin unit to the GPX format, an XML-based file format that includes latitude, longitude, elevation, and timestamp data. We collected GPS data for virtually all exterior paved footpaths on the Swarthmore campus; in contrast to last year's project, we did not record indoor footpaths because of the inability to collect GPS data indoors with inexpensive equipment and our desire to avoid manual editing of the collected data (Singleton and Woods, 2007).

2.1.2 GPS Data Processing

Our pre-processing algorithms find clusters of paths with geographically nearby endpoints. We assume that these paths intersect at a common point, so we reassign the endpoint coordinates of all paths in a cluster to the arithmetic mean of the endpoint coordinates in that cluster. While this approach is fairly naive, we found that it works quite well in practice with the relatively coarse resolution captured by consumer-grade GPS equipment.

We experimented with simplifying the data as a final processing step, hoping to reduce the computational power needed for both the user interface display and our path-finding algorithms. We use the Douglas-Peucker line simplification algorithm to do so. This, however, was not particularly helpful for this particular application in part because of the relatively infrequent position sampling provided by our equipment. Also, we constructed our search graph on adjacent path intersections, ignoring the complexity of the path segment between those intersections, which is an even more efficient simplification for searching purposes. It is important, however, to explore the practicability of such simplification techniques for use with larger, higher-precision datasets.

At the end of these pre-processing steps, the resulting data is saved as latitude, longitude, and elevation tuples for each meaningful point of each path segment. Figure 1 shows a clear difference between our raw data and the same data after being processed with our programs.

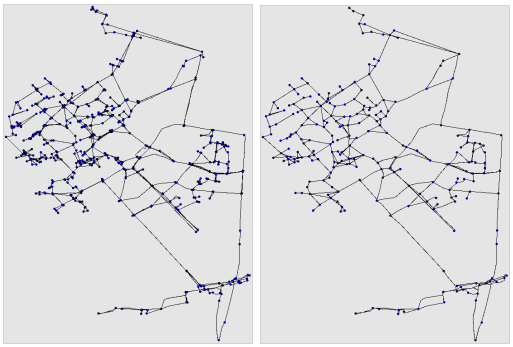

Figure 1: Before and after data processing cleanup on a subset of our GPS data

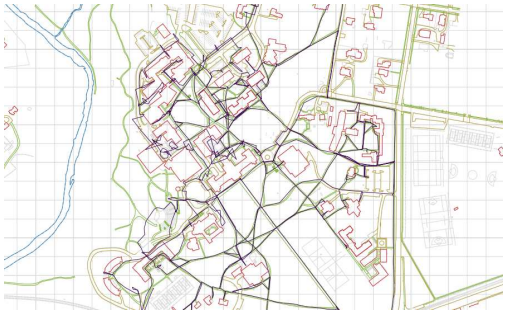

Figure 2: GIS overlay of our GPS data and the survey drawings

2.1.3 CAD Survey Drawings

Thanks to the maintenance staff at Swarthmore College, we also acquired survey-grade mapping data for the Swarthmore campus in AutoCAD format. We convert the several layers of data from these

drawings to the Shapefile format and simplify the data using the Douglas-Peucker algorithm. The final output data uses the same simplified data format described at the end of our GPS data pre-processing algorithms. We had hoped to use information from these drawings as a base layer to help orient users to the paths as displayed in our user interface, but ran out of time before we could complete that effort. Figure 2, however, shows an overlay of these datasets that has been prepared with a GIS package.

2.2 Path Planning

Path planning using the A* search algorithm is fairly straightforward. We create a graph of the connections between path intersections; each edge in the search graph is weighted with the path distance between its endpoint vertices. Then, we apply the search algorithm to this graph, finding the shortest path between the two selected nodes.

A* uses a heuristic function: the sum of distance to get to the current node plus the cost to get to the next node. Using this function, A* chooses which nodes to visit based on the routes that appear to be shortest. The partial paths it has explored are stored in a priority queue, and when its cheapest path becomes relatively expensive, other potentially shorter paths are explored. When a path is found to reach the goal with a lower heuristic value than any other (partial) path in the queue, A* has found the shortest path. A* is both complete and optimal, making it an obvious choice for path finding.

We choose to implement the A* search algorithm because its implementation is not significantly more difficult than implementing Dijkstra's algorithm. Given this, the heuristic in A* provides a slight performance improvement in typical situations.

2.3 UI Development

We provide a python program for visualizing the paths in our search space and the optimized paths between selected points. Search endpoints are selected simply by clicking on intersections in our search graph, and the optimal path between those points is then highlighted on the map. A screenshot of this interface is provided in Figure 3, where the first point selected is green, the second point selected is red, and the shortest path between them is shown in blue.

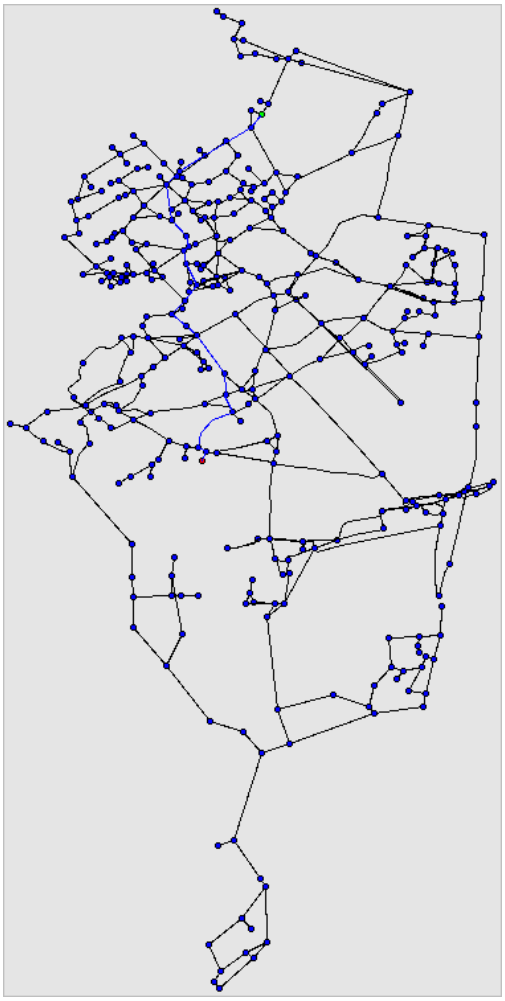

Figure 3: Screenshot of our UI with a path selected

We did not have time to complete further UI development work, although this was not the largest focus of our project. The UI would be even more intuitive, however, if the paths were overlaid on a map that shows other landmarks, such as buildings.

3 Results & Conclusions

Our system appears to be robust and to provide accurate results for a user's queries. The interface can be quickly and simply explained, but is not yet intuitive enough to be used by most individuals without a brief explanation.

We are quite happy with the way that our system meets the goals set out at the beginning, especially with regards to scalability. For example, we developed our processing routines with a subset of our collected GPS data, then were able to add the remainder of our data with less than five minutes of additional work. We are confident that our system would scale well even to much larger datasets than the ones used in this project, and that this represents a good approach to developing such path-finding systems much more quickly and cheaply than has been previously possible.

References

M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. 1997. *Computational Geometry: Algorithms and Applications*. Springer.

E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs.

P. E. Hart, N. J. Nilsson, and B. Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths.

Matt Singleton and Bronwyn Woods. 2007. Finding Your Inner Blaha: GPS Mapping of the Swarthmore Campus.

Drawing Isoglosses Algorithmically

Bryce Wiedenbeck
bryce@cs.swarthmore.edu

Kit La Touche
kit@cs.swarthmore.edu

Abstract

In this paper, we apply algorithms for defining regions from sets of points to the problem of drawing *isoglosses*, the boundaries between dialect regions. We discuss the justifications for our method, and alternative models that could be constructed from this data. We evaluate the resultant model by comparison to the traditional method of drawing isoglosses, by hand.

1 Introduction

In the linguistic subfield of dialectology, an important activity is the drawing of isoglosses, or boundaries between dialect areas. It is often difficult to pin down the meaning of these terms, as within a region people come and go, and bring their speech with them, but broadly speaking, an isogloss is a boundary between where people speak like *this* and where people speak like *that*. Typically, an isogloss will not be a sharp line, but will be an area of overlap between speakers of one type and the other.

Unfortunately, isoglosses are typically drawn by hand, as an approximate dividing line. This leads to two problems: one, they should not be thought of or represented as lines, but rather as approximate transition zones, and two, they could be better drawn, we think, by algorithm than by eyeballing. As the noted sociolinguist William Labov writes,

> Every dialect geographer yearns for an automatic method for drawing dialect boundaries which would insulate this procedure from the preconceived notions of the analyst. No satisfactory program has yet been written. (Labov et al., 2005)

We have therefore attempted, as something like a proof-of-concept, to redraw the isoglosses for certain dialect differences in American English. We have taken as our data the results of the telephone survey of speakers across the contiguous USA done for the Atlas of North American English (Labov et al., 2005). To this data, we have applied a number of algorithms from the field of computational geometry, and hope that the result will better represent the boundaries between dialect regions.

One way we expect that our method will improve on a hand-drawn line is by clearly showing areas that are thoroughly mixed. It can be tempting, when drawing by hand, to mark an area as primarily speaking one way, and drawing your line as though that is the case. It may well be the case, however, that in such situations, the region is much more evenly mixed than it appears to the eye, and should probably not be marked decidedly in either direction. We suspect that an algorithmic approach will see these cases more clearly.

The clearest way to test whether our method improves on a hand-drawn line would be to see if this model has greater predictive power, such that if we were to randomly make telephone calls to people, their proximity to our lines would be a better indicator of the features of their dialect. However, doing so is outside the scope of the project. Other changes to our method that would make it more explicitly a predictive model, such as predicting the value for a point based on the inverse-distance weighted average of the n-nearest neighbors,[1] while interesting, would also be outside the scope of this project. Such an approach would really be a machine learning task, and not a cartographic task.

As such, our evaluation is limited to observing in a qualitative way the differences between our method and the hand-drawn method. We expect that if there is no significant difference, this will at least provide a way of automating the creation of a machine-readable form, assuming data points are available. If there is a significant difference, then

[1] As suggested by George Dahl.

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 22–27,
Computer Science Department, Swarthmore College

perhaps more investigation into the predictive powers of the two models is warranted.

2 The data

The data from the Atlas of North American English (Labov et al., 2005) is in the form of approximately 600 points, identified geographically by ZIP code, which we then converted to latitude and longitude coordinates. At each point, there was an indication of whether the speaker at that point makes or does not make each of a set of possible linguistic distinctions. The data is generally denser around the north east and Great Lakes regions, but this is in part due to greater population density in these areas. Of course, more datapoints would always be desirable, but these data are still useful.

A dialect area can be considered an area of overlap between polygons from different feature sets, where the area of overlap consists of most of the area of the parent polygons. Thus, it is an area where, at least with respect to the features under consideration, people speak the same way, and that way is distinct from the surrounding area. The scale of this can vary, of course: a city may form a distinct dialect area within a state, if it has sufficiently different speech, but there may also be dialect areas within that city, which are each more like each other than they are like the speech outside the city, but still differ from each other.

The specific features which we mapped were the following: whether the speaker distinguishes ɑ and ɔ, whether the speaker distinguishes ʍ and w, and whether the speaker distinguishes ɪ and ɛ before a nasal, such as n or m.[2] These are a set of easily explicable features, with the first and third being generally considered to be characteristic of large US dialects. For each feature, following the format in the Atlas, a speaker could make the distinction, not make the distinction, or be unclear — that is, the interviewer was unable to tell whether they reliably made or did not make the distinction. In all cases, we used the interviewer's judgment, rather than the speaker's self-reporting.

The scale of our project is across the contiguous USA, so many smaller-scale regional variations don't show up. This is dependent on a number of parameters in the algorithm, and the size and granularity of the dataset. For example the data set and our techniques might allow us to note the speech of western Pennsylvania and West Virginia as distinct from the area around it, but would not create a distinct region for the dialect of San Francisco and its countryside.

[2]These symbols are explained in Section 7.

3 Methods

Our goal of identifying dialect regions based on individual phoneme information required us to first locate areas where speakers pronounced a phoneme similarly. For most phonemic variables in the Atlas of North American English, the speakers with similar pronounciation are not located all in one area. This meant that for each feature setting, we had to first break the data points up into several regional groupings, which we did using k-means clustering. Once we had clustered the data for a particular feature setting we found boundaries for the resulting regions by computing the convex hulls of the data points in each cluster. This gave us several polygons describing regions in which a particular feature had the same setting, so to find dialect regions we overlaid these polygon sets and found regions of intersection. Each of these three steps is described in detail below.

3.1 Clustering

To break the data points from a data set for a particular feature setting down into smaller groups, we used k-means, a clustering algorithm proposed by MacQueen (1967). The algorithm starts with k centroids that can be specified as input or randomly selected from the data points. Each data point is then assigned to the nearest centroid, and each centroid is adjusted to minimize the distance to all of its points. Then the data points are again assigned to the closest centroid, and the centroids are recomputed, and so on. The algorithm terminates when an iteration occurs in which no data points switched centroids. The theoretical worst-case time complexity of the k-means algorithm is superpolynomial (Arthur and Vassilvitskii, 2006), but in practice it runs quite quickly.

We considered a number of alternative clustering algorithms including faster heuristic-based or

approximate k-means implementations. We also considered other more adaptive clustering algorithms, like growing k-means or growing neural gas (Daszykowski et al., 2002). The main advantage to such techniques is that they would not require the number of clusters to be specified in advance, but instead start with very few centroids and add more as needed. We decided against such methods after considering the potential applications of our work. First, if a user of our algorithm, say a linguist creating dialect maps, is unfamiliar with the details of the implementation, it might be easier for him to specify starting-point centroids than to tune the parameters that control the termination point of growing k-means or growing neural gas. Additionally, because our data come from an atlas with hand-drawn isoglosses, we have available to us reasonably good starting-point centroids to use in our testing.

3.2 Convex Hulls

After forming clusters, we computed the convex hull of each set of clustered points, using a simple convex hull algorithm, the Jarvis March. The algorithm uses a "gift-wrapping" technique, starting from the left-most point in the data set, and at each subusequent step, scanning through all the remaining points to find the one that makes the widest angle with the previous hull boundary-segment. This runs in in O(nh) time, because each of h loops scans all of the input points (de Berg et al., 1997). Non output-sensitive, O($n \log n$), convex hull algorithms also exist and are useful when h is close to n, resulting in n^2 complexity for Jarvis, but in our application, h tends to be quite small relative to h, so the Jarvis March performs marginally better.

When inspecting the first applications of our convex-hull implementation to real data, we noticed that the hulls were often significantly affected by a few outlier points that weren't particularly close to any of the cluster centroids but our implementation of k-means required that they be assigned to some cluster. For example one data set had three clear clusters in the northeast, the midwest, and the deep south, plus a few extraneous data points in California. These California points caused the midwest-hull to stretch halfway across the country (see figure 1).

To fix this, we tried two different methods. The

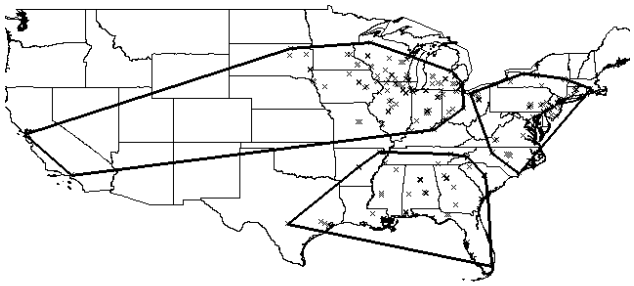

Figure 1: Convex hulls of point clusters for regions where ɑ and ɔ are distinguished, without oulier removal.

first was onion-peeling of the convex hulls, [3] which would involve throwing out the points on the convex hull and recomputing the convex hull of the remaining points (perhaps more than once). We found this method ineffective because removing all outliers would sometimes require more than one onion-peeling step, in which case some more-compact clusters would be whittled away and no longer accurately represent the data. We then tried, and were satisfied with, calculating the outliers more explicitly. For each cluster, we computed the standard deviation of the distance from each point to the centroid and removed all points more than some number of standard deviations[4] away before computing the convex hull.

3.3 Overlay

Finally, given the hulls for each cluster in a particular map, we overlaid them to find the common area between, for example, the speakers who both make the ɑ/ɔ distinction, and drop syllable-final r. By overlaying hulls from different features, we could identify regions of significant overlap, indicating dialect regions, and by overlaying hulls from different settings of the same feature, we can determine border areas where common sets of linguistic features cannot be readily described.

[3] Suggested by Andrew Danner.

[4] We found that 3 standard deviations produced good results.

Overlaying these regions requires first detecting whether two polygons intersect. A method to do this in O($\log n$) time was proposed by Chazelle and Dobkin (1980), but is exceedingly difficult to implement, so we opted instead for an O(n^2) algorithm based on the leftTurn and rightTurn primitives we had already implemented for computing the intersection. The idea is that if two polygons don't intersect, then there exists a dividing line between them, and one such line must be a side of one of the polygons. Therefore we can walk around the outside of one polygon and determine for each side, whether all points in the other polygon lie on the opposite side of that line from the rest of its polygon. If so, the polyogns don't intersect; if no such line is found after walking around both polygons, then the polygons intersect.

Once we know that two polygons intersect, we use a rotating calipers implementation provided by Mary Wooters and George Dahl to compute bridge points between them (Toussaint, 1983). Each of these bridge points corresponds to a point of intersection between the two hulls at the other end of the "sail polygon" (see figure 2), which we locate using the stepDown procedure outlined in Toussaint (1985). Once we have located these intersection points, finding the convex intersection of the two polygons is simply a matter of walking around one polygon and then the other, switching at each intersection point. Both rotating calipers and stepDown run in time linear in the size of the polygons, as does the final output step, so the whole overlay step runs in worst-case O($k^2 * h$) time, where k is the number of k-means centroids (and therefore the number of polygons), and h is the number of points on the polyogns being combined. In practice, k and h are well under $n^{\frac{1}{3}}$, so this works out to be no worse than linear in the size of the overall data set.

To visualize the data, we used GRASS,[5] mapping our datapoints to latitude and longitude coordinates.

4 Results

The regions drawn by this method seem plausible given our knowledge of the data sets we used. However, a rigorous test of the predictive power of this model is not particularly feasible without more data.

[5]GRASS is an Open Source geographic information system, available at `http://grass.osgeo.org/`.

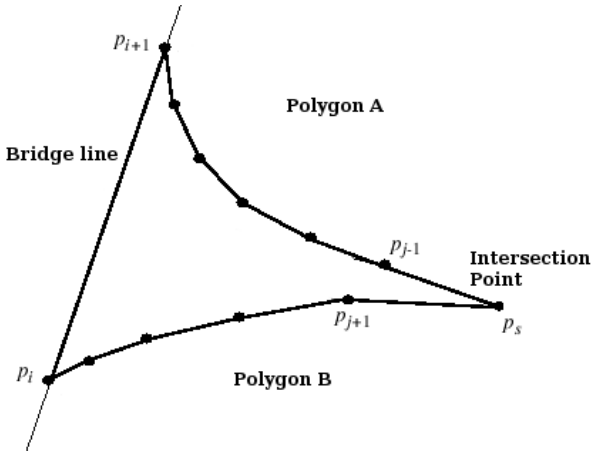

Figure 2: The stepDown function iterates over a "sail polygon" to find the intersection between two polygons given a bridge-line between them.

One major difference is that this method produces only convex polygons as dialect regions, and also does not cover all the inhabited territory on the map. Both of these facts imply something about how our regions are to be interpreted: for a given feature, the regions of no overlap with other settings for that feature are areas of high-confidence that there is one dialect, areas of overlap are areas of confidence that the speech is mixed there, and areas outside of polygons are areas of insufficient data.

There were some moments when it was clear how to hand-evaluate the algorithm. For example, before switching to standard-deviation hulls, we attempted onion-skin hulls. This resulted in a hull that stretched across the US from Wisconsin to southern California, even after peeling off the first layer. This was clearly an artifact of the processing, and not reflective of a here-to-fore unobserved swath of speakers who distinguish ɑ and ɔ; the greater part of the polygon, as it stretched across the country, was devoid of datapoints, having them all clustered at the Wisconsin end.

5 Future Work

The clearest route for future work would be to develop a metric with which to test the quality of a dialect map. The purpose of such a map is presumably as a predictive tool; as such, gathering larger amounts of data, for distinct training and testing,

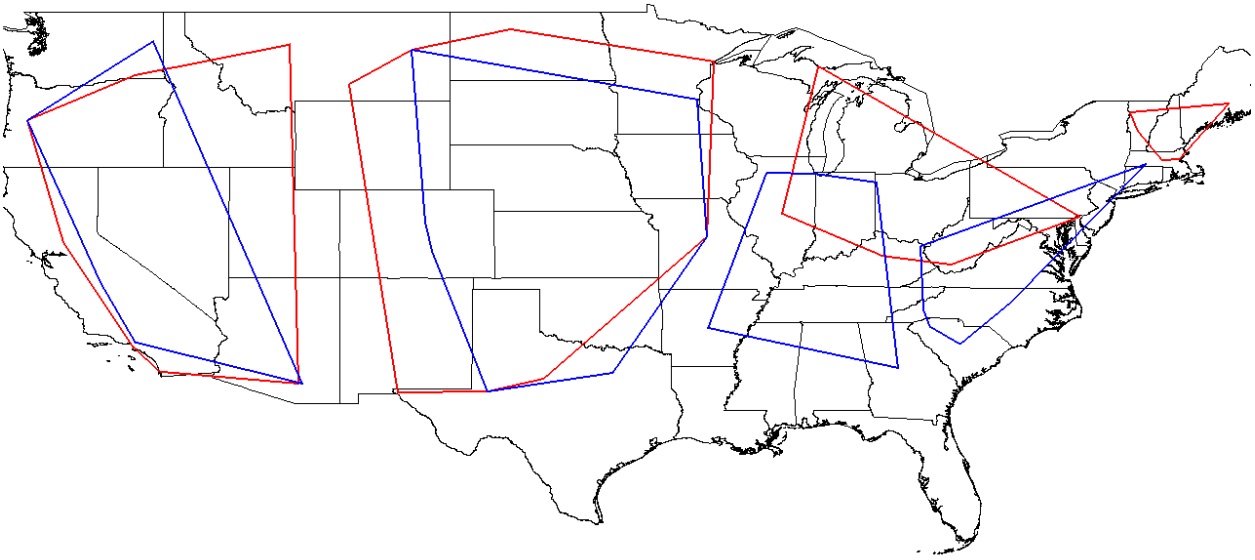

Figure 3: Overlaid convex hulls of clustered data. The red are areas where ɑ and ɔ are merged. The blue are areas where ɪ and ɛ are merged before nasals.

would be ideal. However, collecting such data remains time-consuming and difficult.

It has been suggested that an alternative approach to this problem would be to treat it as a machine learning task. One could imagine easily a learning system that would, given training data of geographic points classed by setting of linguistic features, would produce the likeliest settings for linguistic features, given testing data of just geographic points. While such a system would be interesting, and possibly very informative, it would not lend itself immediately to map-making, and was outside the scope of this project.

6 Conclusions

It is difficult to evaluate the success of our algorithm both because the Atlas is not conducive to a quantitative comparison, and because testing our algorithm against real-world data is impractical. The algorithm seems to produce reasonable results, but a large part of our ability to evaluate this is based, circularly, on the maps available in the Atlas. Clearly, both more data and a better means of evaluation would be desirable.

7 Linguistic Appendix

A number of phonetic symbols have been used in this paper. As familiarity with them is not assumed, we will explain them here.

- ɑ: for those that distinguish this from the following vowel, it is the vowel in t**o**t.
- ɔ: for those that distinguish this from the preceding vowel, it is the vowel in t**augh**t.

- w: this is the first consonant of **w**itch.
- ʍ: this is the first consonant of **wh**ich, though many speakers do not use this, instead merging with w, above.
- ɪ: as in p**i**n.
- ɛ: as in p**e**n. Some US dialects do not distinguish ɪ and ɛ before nasal consonants, such as n or m.

These features were selected because they were available in the Atlas, and easily explicable. A thorough effort to algorithmically create dialect maps would use many more features.

References

David Arthur and Sergei Vassilvitskii. 2006. How slow is the k-means method? In *SCG '06: Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, New York, NY, USA. ACM.

Bernard Chazelle and David P. Dobkin. 1980. Detection is easier than computation (extended abstract). In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 146–153, New York, NY, USA. ACM.

M. Daszykowski, B. Walczak, and D. L. Massart. 2002. On the optimal partioning of data with k-means, growing k-means, neural gas and growing neural gas. *Journal of Chemical Information and Modelling*, 42(6):1378–1389.

Mark de Berg, Marc van Kreveld, Mark Overmas, and Otfried Schwartzkopf. 1997. *Computational Geometry*. Springer.

William Labov, Sharon Ash, and Charles Boberg. 2005. *The Atlas of North American English*. Mouton de Gruyter.

J. B. MacQueen. 1967. Some methods for classification and analysis of multivariate observations. In L. M. Le Cam and J. Neyman, editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

Godfried Toussaint. 1983. Solving geometric problems with the rotating calipers.

Godfried T. Toussaint. 1985. A simple linear algorithm for intersecting convex polygons. *The Visual Computer*, 1(2):118–123.

The Largest Empty Circle Problem

Megan Schuster
megan@cs.swarthmore.edu

Abstract

The largest empty circle (LEC) problem is defined on a set P and consists of finding the largest circle that contains no points in P and is also centered inside the convex hull of P. The LEC is always centered at either a vertex on the Voronoi diagram for P or on an intersection between a Voronoi edge and the convex hull of P. Thus, finding the LEC consists of constructing a Voronoi diagram and convex hull for P, then searching the Voronoi vertices and intersections between Voronoi edges and convex hull edges to see where the LEC lies. This paper presents a simple $O(n[h+\log n])$ solution to the largest empty circle problem. Though previous work on this problem has found $O(n \log n)$ solutions, we find that for data sets which are somewhat normally distributed, h is small and our simple algorithm performs well.

1 Introduction

The Largest Empty Circle (LEC) problem is defined on a set of points P and consists of locating the largest circle that contains no points of P and is centered inside the convex hull of P. Less formally, this problem finds the point q bounded by the convex hull which maximizes the distance to its nearest neighbor $p \in P$; this point is the center of the largest empty circle. We constrain q to lie within the convex hull of P because otherwise we would simply choose the point at infinity as the center of the LEC.

This problem is sometimes referred to as the Toxic Waste Dump problem, because given the coordinates for a set of cities, the LEC problem would allow you to find the best site for a toxic waste dump by finding the location which is maximally far from every city. This might also be useful for planning locations for new stores. For example, imagine you would like to build a new McDonald's in a metropolitan ares that already has several dozen McDonald's stores. By computing the LEC on the set of existing McDonald's restaurants, you could select a site for a new store which is maximally far from all existing stores to minimize competition with other McDonald's restaurants and situate yourself near people who previously did not have a McDonald's nearby.

The Voronoi diagram for the set P is a useful tool for solving the LEC problem. The Voronoi diagram is a partition of the plane into convex faces such that given a set of points P, each face (hereafter, Voronoi cell) contains exactly one point $p \in P$ and all points in the plane which are closer to p than to any other point in P. Points lying on the edges between Voronoi cells are equidistant between the two points contained in the cells lying to either side of the edge. Given these properties, it seems intuitive that the largest empty circle should be centered on a Voronoi vertex. Since the edges represent all points which are equidistant to the two points they divide, it follows that a vertex, which is simply an intersection between multiple Voronoi edges, would maximizes the minimum distance to all nearby points. Any point that is not on a Voronoi vertex must be closer to one point than any other and as such any empty circle we can draw around it will not be of maximal size.

Based on the above principles, it seems that we might simply be able to draw empty circles around all Voronoi vertices and see which is the largest. However, since we constrain our circle to be centered inside the convex hull of P, we must be slightly more careful in our search for the LEC. First, we must consider only Voronoi vertices which lie inside the convex hull of P. Second, we must consider what happens on the edges of the convex

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 28–37,
Computer Science Department, Swarthmore College

hull itself. At points where a Voronoi edge intersects a convex hull edge, the distances between each of the two nearest points is maximized, since we are on a Voronoi edge. Such points must also be considered as candidates for the center of the LEC.

2 Related Work

The earliest solution to the LEC problem was presented by Shamos in his Ph.D. thesis (1978). Shamos presented an algorithm that, given the Voronoi diagram and the convex hull, could find the largest empty circle in $O(n)$ time. Unfortunately, this algorithm was based on the assumption that every convex hull edge is intersected by at most two Voronoi edges, which is not always true, as later shown by Toussaint (1983). Because the original Shamos algorithm incorrectly assumes a maximum of two Voronoi edge intersections at each convex hull edge, it can miss intersections at edges with more than two intersections with Voronoi edges and, as such, can fail to recognize the true LEC.

Toussaint (1983) went on to present an algorithm which correctly finds the LEC in all cases. However, the Toussaint algorithm requires $O(n \log n)$ running time when given the convex hull and Voronoi diagram. The algorithm first computes the largest empty circle about each Voronoi vertex which lies in the interior of the convex hull. This step requires $O(n \log h)$ operations; there are $O(n)$ Voronoi vertices for which we must do an $O(\log h)$ point location step to check for interiority to the convex hull (where h is the number of convex hull edges), and computing the largest empty circle about a point can be done in constant time. Once all interior points have been considered, the algorithm computes all intersections between Voronoi edges and convex hull edges. Toussaint uses an $O(\log n)$ algorithm taken from Chazelle (1980) to find the intersections between a line segment and a convex n-gon. Since there are $O(n)$ Voronoi edges (de Berg et al., 2000), this step requires $O(n \log h)$ time overall. By checking all points interior to the convex hull and all intersection points between the Voronoi diagram and the convex hull, all possible sites for the center of the LEC have been considered. All that remains is to report the point about which the largest circle was drawn. Toussaint thus finds the LEC in $O(n \log n)$ time.

Later, Preparata and Shamos (1985) offer an improved version of Shamos's original methods (1978) which no longer relies on the assumption that each convex hull edge is intersected by at most two Voronoi edges. Preparata and Shamos describe an $O(n)$ marching method for finding all intersections between the Voronoi edges and the convex hull, which is an improvement on Toussaint's $O(n \log h)$ method. Preparata and Shamos do not go into detail about how to check Voronoi vertices on the interior of the convex hull to see if they might be the center of the largest empty circle. We can only assume that they, like Toussaint, also require an $O(n \log h)$ technique for checking interior points. This solution is then slightly faster than Toussaint's, thanks to the $O(n)$ intersection location step.

Still, all of the LEC-finding algorithms discussed above require use of the Voronoi diagram and the convex hull. Both of these structures require $O(n \log n)$ steps to compute. Construction of these structures dominates the computation time when finding the LEC, so while there may be some quibbling about the fastest methods for finding intersections between Voronoi diagrams and convex hulls, overall the solution to the LEC problem is at best $O(n \log n)$ regardless of the steps required to actually find the largest empty circle.

In this paper, we present a complete method for finding the LEC, borrowing our approach to the problem from the previous work of Toussaint. The algorithm presented includes a computation of the Voronoi diagram and the convex hull and requires $O(n[h + \log n])$ running time.

3 Methods

3.1 Computing the Voronoi Diagram and Convex Hull

Before we can find the largest empty circle for a set of points P, we must first construct the Voronoi diagram ($Vor(P)$) and convex hull ($CH(P)$) for the set of points. Here, $Vor(P)$ is computed by first finding the Delaunay triangulation, $DT(P)$, which is the dual of $Vor(P)$. Common algorithms for computing $Vor(P)$ involve an $O(n \log n)$ plane sweep and are not dynamically updateable. However, de Berg et al. (2000) describe an incremen-

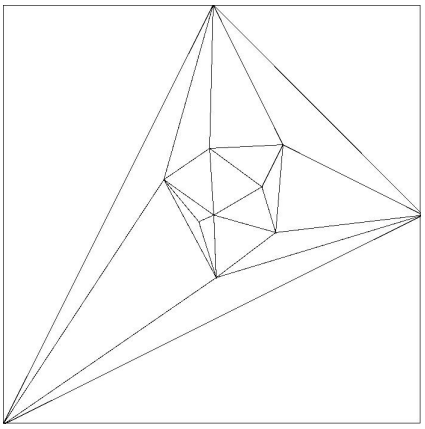

Figure 1: A simple example of the Delaunay triangulation computed on a small set of points.

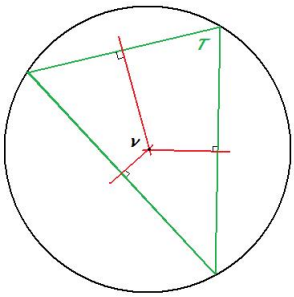

Figure 2: The circumcircle for a triangle T, whose center v is found by perpendicular bisector construction. If T is a triangle in a Delaunay triangulation, the v is the Voronoi vertex obtained when dualizing the Delaunay triangulation to the Voronoi diagram.

tal algorithm for computing $DT(P)$. Because we would like to support dynamic updates to $Vor(P)$, we choose to compute $Vor(P)$ by dualization of a dynamically updateable implementation of $DT(P)$.

3.1.1 The Delaunay Triangulation

de Berg et. al (2000) describe an incremental algorithm for computing Delaunay triangulations, which we employ here to compute $DT(P)$. The Delaunay triangulation is a special type of triangulation which maximizes the minimum angle found in any triangle in the triangulation. An example is shown in Figure 1.

We begin with a very large bounding triangle and add points from P to it one at a time. When adding a point p, we check the current triangulation and locate the triangle T which contains p. We then draw edges between p and each of the vertices of T to re-triangulate the set P. In doing so, however, we may have introduced new triangles with small angles such that we no longer have a Delaunay triangulation. Thus, as we re-triangulate with every added point, we must check the edges of new triangles introduced to see whether they form any small angles. We flip such edges as needed to the opposite corners of the quadrilateral which contains them, recursing on triangles in the neighborhood of newly flipped edges until we have ensured again that the smallest angle in the triangulation is as large as possible.

We represent triangles as nodes in a directed acyclic graph. Whenever a triangle is divided by insertion of a new point or changed due to edge-flipping, it sets pointers to the new triangles which result from these changes. Each triangle node also maintains a pointer for each of its three edges indicating which other triangle neighbors it along that edge, with care being taken to update these neighbor pointers as new triangles are introduced and edge flips are performed. These neighbor pointers are crucial to our ability to dualize $DT(P)$ to $Vor(P)$.

For full details on the computation of Delaunay triangulations, see see de Berg et al. (2000). Once we have a Delaunay triangulation in place, we can dualize it to give the desired Voronoi diagram.

3.1.2 The Voronoi Diagram

Once $DT(P)$ has been computed, it is fairly straightforward to dualize it to $Vor(P)$. The duality between the Voronoi diagram and the Delaunay triangulation is such that every triangle in $DT(P)$ corresponds to a vertex in $Vor(P)$. Triangles which are neighbors in $DT(P)$ have their vertices connected by an edge in the $Vor(P)$ dual space (de Berg et al., 2000).

The coordinates of the Voronoi vertex which correspond to a triangle $T \in DT(P)$ can be found by locating the center of the circle which circumscribes T (Okabe et al., 2000). To compute the circumcircle, we use the perpendicular bisector construction as in Figure 2; the circumcircle for a triangle is centered at the point at which the perpendicular bisectors for the triangle's edges all intersect. To compute $Vor(P)$ from $DT(P)$, then, we first iterate over all triangles in $DT(P)$, computing the circumcircle to find the dual vertex in $Vor(P)$.

Once all Voronoi vertices have been found, we

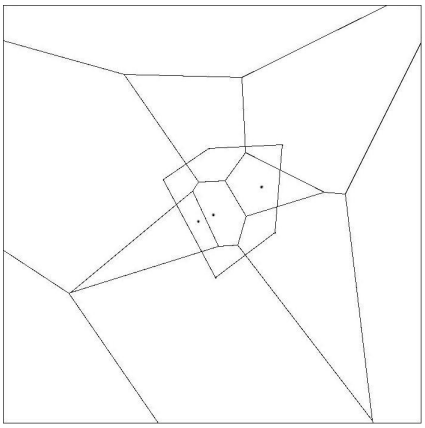

Figure 3: A simple example of the Voronoi diagram and convex hull computed on the set of points shown in Figure 1.

connect them to one another by again iterating over the triangles of $DT(P)$. Here, we make use of the neighbor pointers stored in our Delaunay structure. For each triangle $T \in DT(P)$ and its corresponding point $p \in Vor(P)$, we retrieve all of T's neighbors, T'. For each $t \in T'$, we take its corresponding point $p' \in Vor(P)$ and draw an edge between p and p'. The resulting set of edges describes exactly the Voronoi planar partition.

The dualization from $DT(P)$ to $Vor(P)$ described here can be computed in $O(n)$ time. We must iterate over all triangles in $DT(P)$, of which there are $O(n)$ (de Berg et al., 2000). For each triangle, we must compute the circumcircle to locate the corresponding Voronoi vertex and check three neighbor pointers to draw the appropriate Voronoi edges. These two operations can be done in constant time, giving an overall $O(n)$ runtime for the dualization step.

3.1.3 The Convex Hull

Finally, we must compute $CH(P)$ before we can go on to find the LEC. Here, we use the simple $O(nh)$ Jarvis march algorithm to compute the convex hull. The interested reader should refer to de Berg et al. (2000) for further details on this algorithm.

At this point, we have found the Voronoi diagram and convex hull for our data set (Figure 3). With all necessary data structures in place, we now proceed to find the largest empty circle.

3.2 Finding the Largest Empty Circle

To find the largest empty circle, we first locate all potential centers for that circle, which involves identifying all Voronoi vertices which are interior to $CH(P)$ and finding all intersections between Voronoi edges and convex hull edges. Once all candidate centers have been located, we draw the largest possible empty circle around each and report which was the largest of all.

3.2.1 Checking Interior Voronoi Vertices

We use a naive approach for finding all Voronoi vertices which are interior to $CH(P)$. For each Voronoi vertex, we march counter-clockwise around $CH(P)$, checking whether the vertex lies to the left of the edge. If the vertex lies to the left of all edges in $CH(P)$, we know that it is interior and add it to the list of candidate LEC centers. This requires $O(nh)$ steps, since we must check all h convex hull edges for all $O(n)$ Voronoi vertices.

3.2.2 Finding Convex Hull and Voronoi Edge Intersections

We employ another naive approach for finding all intersections between Voronoi edges and convex hull edges. For every Voronoi edge, we check both endpoints for interiority to $CH(P)$. If one is interior and one is exterior, we know that this edge must intersect $CH(P)$ at some point. We then iterate over all convex hull edges and check for intersection with the Voronoi edge in question. By repeating this process for every Voronoi edge, we are guaranteed to find all intersections between $Vor(P)$ and $CH(P)$.

This step requires $O(nh)$ runtime. We must check all $O(n)$ Voronoi edges, and for any which intersect the convex hull, we must iterate over all h convex hull edges to find the intersection point.

3.2.3 Locating the Largest Empty Circle

Now that we have found all possible points at which the LEC can be centered (as in Figure 4), we have to decide which of these candidates is the actual center of the LEC. To find the largest empty circle that can be drawn around any given candidate center, we exploit the duality between $DT(P)$ and $Vor(P)$. Candidate centers are vertices in Vor space; however, when drawing a circle around a candidate point we want to ensure that this circle does

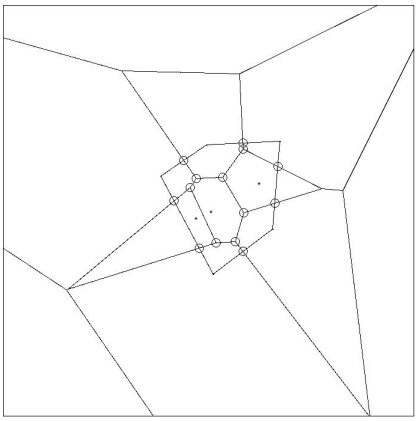

Figure 4: Candidate centers for the largest empty circle for the set of points shown in Figures 3 are outlined here. All candidate centers lie on Voronoi vertices or on intersections between Voronoi and convex hull edges.

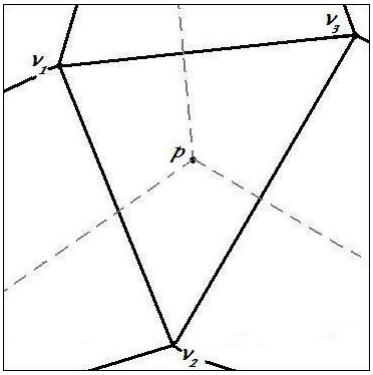

Figure 5: Solid, black lines indicate edges in $DT(P)$. Dashed, grey lines indicate edges in $Vor(P)$. The point p is a Voronoi vertex and is thus a candidate LEC center. Here, p's three nearest neighbors are the three points contained in the Voronoi cells adjacent to p, which are the vertices of p's dual triangle in DT space, v_1, v_2, and v_3.

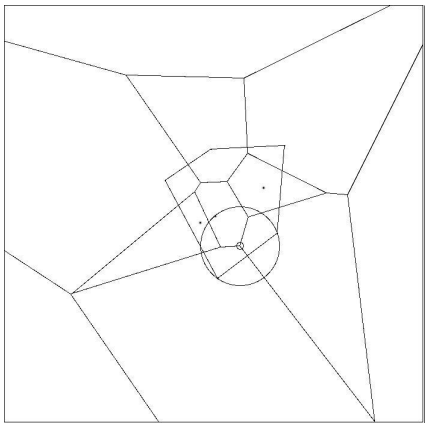

Figure 6: The LEC for our simple set of points.

not contain any points in P, which are vertices in DT space.

Any Voronoi vertex is the intersection of at most three Voronoi edges. This follows directly from the dual relationship between $Vor(P)$ and $CH(P)$; each Voronoi vertex corresponds to one Delaunay triangle and is connected to the vertices which correspond to that triangle's neighbors, of which there are exactly three (except near the edges of the space, where there may be only two neighbors). Since each Voronoi vertex p is incident to at most three Voronoi cells, the three points in P closest to p are those three points which lie in the Voronoi cells to which p is adjacent. Those three points are the vertices of p's dual triangle in $DT(P)$ (see Figure 5). Thus, by dualizing $p \in Vor(P)$ back to $T \in DT(P)$, we can find p's three nearest neighbors. We then choose the closest neighbor and draw a circle about p whose radius equals the distance between that neighbor and p.

We accomplish the dualization of a Voronoi vertex to a Delaunay triangle very simply by building a dictionary of Voronoi vertex-Delaunay triangle pairs as we are doing our initial construction of $Vor(P)$ from $DT(P)$. We then look up candidate vertices in this dictionary to find their nearest neighbors and draw empty circles about them. By drawing empty circles around each candidate vertex, we can locate and report the LEC.

This final step of the algorithm requires $O(n)$ time. Finding the closest point P to any candidate center can be done in constant time by simple dictionary lookup, and we must repeat this process for

Computational Step	Runtime
Compute *DT(P)*	*O(n log n)*
Dualize *DT(P)* to *Vor(P)*	*O(n)*
Compute *CH(P)*	*O(nh)*
Locate Interior Voronoi Vertices	*O(nh)*
Locate *Vor(P)/CH(P)* Edge Intersections	*O(nh)*
Locate LEC from Amongst Candidate Points	*O(n)*

Figure 7: A summary of the steps involved in computing the LEC.

all $O(n)$ candidate centers.

At this point, we need only to report the largest empty circle we have found, as in Figure 6. Figure 7 provides a summary of all steps taken to compute the LEC and their associated runtimes. The overall runtime of our algorithm is thus $O(n[h + \log n])$.

4 Results

We ran the algorithm described above on a set of points corresponding to the latitude and longitude coordinates of all U.S. cities in the 48 contiguous states of population 100,000 or greater (there are 251 such cities). We found the LEC to be centered at -108.2659° latitude, 46.7316° longitude, near Winnette, MT. $Vor(P)$ and $CH(P)$ for this data set are displayed in Figure 8; the resulting LEC is displayed in Figure 9.

The algorithm runs on this dataset of U.S. cities in less than four seconds. In order to analyze the algorithm's overall performance and the performance of intermediate steps within the algorithm, we ran it on a number of data sets ranging in size from 100 points to 10,000 points and recorded the amount of processor time required to compute each step of the algorithm. For these timed tests, we used randomly generated, normally distributed sets of points. The results of these tests are shown in Figure 10.

5 Discussion

Our algorithm was tested on and successfully computed the LEC for the U.S. cities data set as well as randomly generated, normally distributed data sets of up to 10,000 points. It is clear from Figure 10 that the time required to compute the LEC is dominated by the computation of $DT(P)$; all other steps of our algorithm proceed quickly in comparison. For the 10,000 point data set, for example, it took about 33 minutes to compute $DT(P)$. The next slowest step of the algorithm was locating the intersections between $Vor(P)$ and $CH(P)$, which took only 26 seconds.

As mentioned earlier, we used naive approaches to several of the intermediate steps of this algorithm, including the simple Jarvis march for convex hull computation, the point location step for finding all Voronoi vertices interior to the convex hull, and the step for identifying intersections between Voronoi and convex hull edges. Each of these naive steps was $O(nh)$, and for each of these steps we might have used a more sophisticated algorithm in hopes of achieving better runtime for the overall LEC algorithm. For the convex hull, we might have chosen from a variety of $O(n \log n)$ algorithms (see Preparata and Hong (1985), de Berg et al. (2000), for examples). This only improves the speed of computing the convex hull if the number of convex hull edges is somewhat large. For the identification of interior Voronoi vertices, Toussaint (1983) describes an $O(n \log h)$ technique, which is a guaranteed improvement over the $O(nh)$ technique we use, regardless of the distribution of our set of points. For the Voronoi/convex hull intersection location step, Chazelle (1980) describes an $O(n \log h)$ technique for computing the intersections between a set of segments and a convex polygon, and better still, Preparata and Shamos (1985) describe an $O(n)$ march around all Voronoi cells that discovers all intersections with the convex hull.

While the naive approaches to these intermediate steps used in our algorithm could be improved by implementing any of the known faster algorithms mentioned above, the results of Figure 10 suggest that this would provide very little improvement to the running time of our overall LEC algorithm. It is clear that the $O(n \log n)$ computation of $DT(P)$ dominates the computation time for finding the LEC, and thus minor speed-ups to intermediate steps would be of limited value.

It should be noted that the data sets on which we tested our algorithm's running time were all approximately normally distributed. As such, h, the number of convex hull edges, was quite small compared to n, the number of data points (see, for example, Figure 8). Considering the possible applications of the LEC problem, such as toxic waste dump site selection or business location planning, we expect that the data

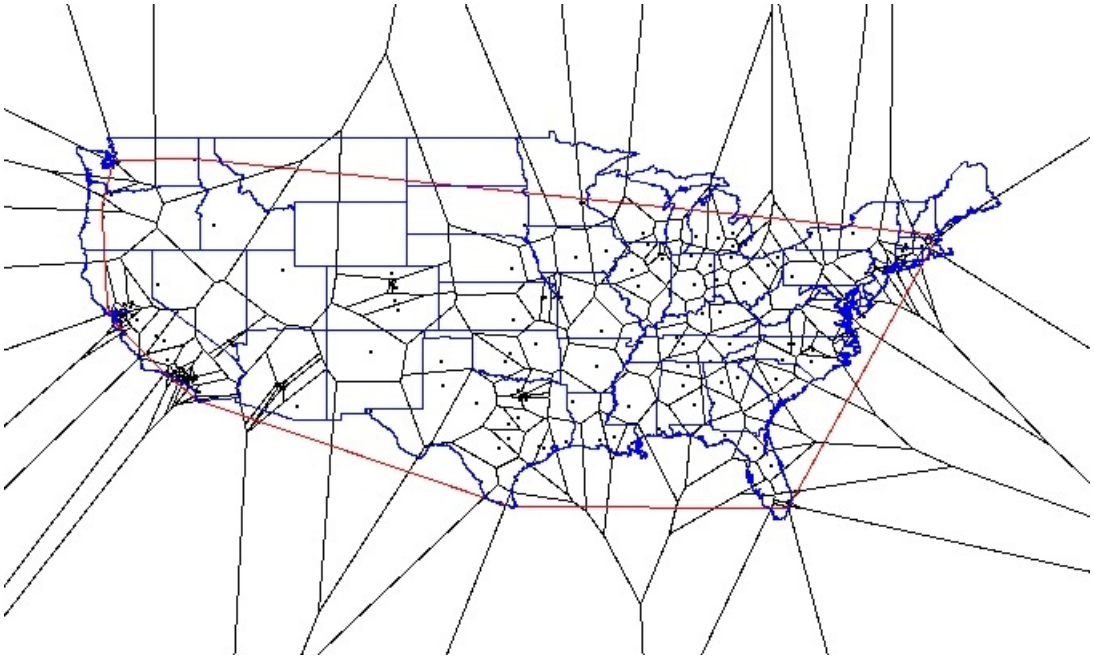

Figure 8: The Voronoi diagram and convex hull for the data set of all US cities of population 100,000 or greater.

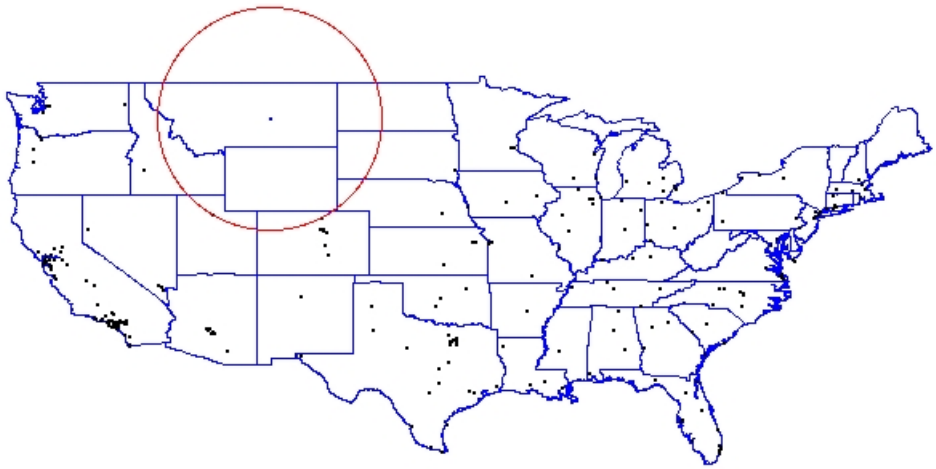

Figure 9: The largest circle which contains no U.S. cities of population 100,000 or greater and is centered within the convex hull of these cities. The center lies at -108.2659° latitude, 46.7316° longitude, near Winnette, MT.

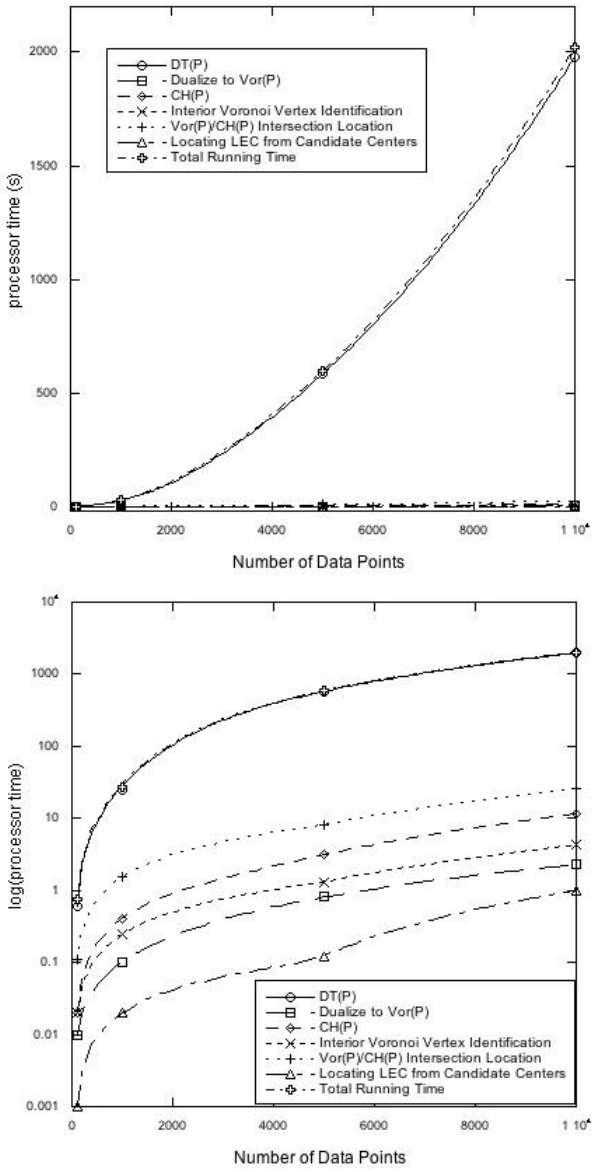

Figure 10: At top, a plot of processor time versus number of data points. The time required for the $DT(P)$ computation is barely distinguishable from the total running time of the entire algorithm. All other algorithmic steps have running times clustered very near to zero for data sets of all sizes and are not distinguishable on this plot. For this reason, a plot of log(processor time) versus number of data points is shown at bottom. The time for the $DT(P)$ computation is again quite similar to the time for the overall algorithm. The next most time consuming step is the identification of $Vor(P)/CH(P)$ edge intersections, but this step is far faster than the computation of $DT(P)$.

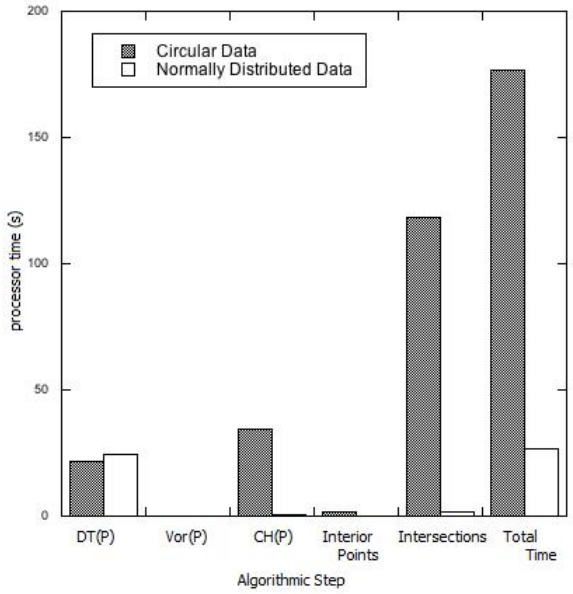

Figure 11: Processor time used by our algorithm on 1000 data points lying on a circle as compared to 1000 normally distributed, randomly generated data points.

sets our algorithm will most commonly encounter will be similar to the data sets on which we have already tested it–somewhat normally distributed, with h small compared to n. However, should we need to compute the LEC on some set of points which is distributed differently (say, in a ring-like shape, where h and n would be similar), we expect our $O(nh)$ intermediate steps to begin to contribute to significantly slower runtimes for our algorithm overall.

To test this idea, we ran our algorithm on a set of 1000 points distributed on a circle so that $h = n$ and compared the resulting performance to the performance on a normally distributed set of the same size. The resulting running times are displayed in Figure 11. Note that the $O(nh)$ check for interior Voronoi vertices is still relatively quick. However, we see that our convex hull algorithm and our technique for locating Voronoi/convex hull intersections take much longer on the circular data than on the normally distributed data, leading to about a sevenfold increase in the overall running time of our algorithm. Thus, if we intend to run our algorithm on non-normally distributed data where h may become large, we would almost certainly benefit from switching to an $O(n \log n)$ convex hull algorithm and an $O(n)$ technique for finding Voronoi/convex

hull intersections. Still, because our algorithm is primarily applicable in domains where data is likely to be normally distributed, such as sets of cities or business locations, we continue to use the naive $O(nh)$ intermediate steps, since they seem to work well for this type of data.

6 Conclusion and Future Work

In this paper, we have describe an algorithm which computes the largest empty circle on a set of points P. The algorithm is $O(n[h + \log n])$ and supports dynamic updates to the set P without heavy re-computation of underlying structures. The algorithm requires the use of the Voronoi diagram, which we compute by dualizing the Delaunay triangulation of P. The convex hull of P must also be computed. We then check all Voronoi vertices and intersections between Voronoi and convex hull edges to see which is the center of the largest empty circle.

Our algorithm is $O(n[h + \log n])$, which is asymptotically worse than the $O(n \log n)$ solutions published by Toussaint (1983) and Preparata and Shamos (1985). This extra h term results from taking naive $O(nh)$ approaches to a few intermediate steps in computing the LEC. However, we have shown that our algorithm's overall running time is not much affected by these $O(nh)$ intermediate steps when our data is more or less normally distributed. Thus, we find that our use of simple, naive techniques at some steps of our algorithm is justified; while asymptotically faster techniques do exist for these steps, these faster techniques are considerably more complicated than the simple approach we take. Because our $O(nh)$ methods contribute very little to the total runtime of our algorithm (Figure 10), we find that we can use simple, straightforward techniques at a negligible cost to the running time of our algorithm.

For future work, it may be useful to implement support for further location constraints on the center of the LEC, such as described in Chew and Drysdale (1986) or Toussaint (1983). For example, we might like to restrict the LEC to be centered in some simple, though not necessarily convex, polygon or set of polygons other than the convex hull. This would have been useful when working with the data set comprised of US cities. Suppose we are using our algorithm to find a toxic waste dump site. Rather than using the convex hull of this data set, which contains parts of Mexico, the Gulf of Mexico, and the Atlantic and Pacific oceans, and does not include all of the land comprising the 48 contiguous states (Figure 8), we might have preferred to constrain the LEC to be centered anywhere on US mainland territory. This would both ensure that the selected site were actually a United States holding, and would also provide a wider selection of possible LEC centers by including more area in the northern United States. Thus, using some sort of simple, polygonal approximation of the 48 contiguous states would be an improvement.

To do this, we would have to change our point location strategy for testing whether a Voronoi vertex is interior to the bounding region, since that region would no longer be convex. The current naive $O(nh)$ technique for finding intersections between the Voronoi diagram and the bounding region would still be effective, but if we were to update to the $O(n)$ marching technique described by Preparata and Shamos (1985), we would have to modify it slightly to deal with non-convex bounding regions.

Our current algorithm avoids complicated intermediate steps and successfully computes the LEC on data sets of varying sizes and distributions. It is quite fast on normally distributed sets. While we could improve its performance on non-normal data sets and support further location constraints on the center of the LEC by using more complicated intermediate steps in our algorithm, for now we stick with the simple solution to the LEC problem and assume it will most often be used on normally distributed data sets.

7 Acknowledgements

I am grateful to Professor Andy Danner for advising this project and for providing a script for graphing the 48 contiguous states. I also thank my Swarthmore College Computer Science Senior Conference classmates for reviewing this paper and providing suggestions for its improvement.

References

B.M. Chazelle, 1980. Computational Geometry and Convexity, Ph.D. thesis, Carnegie-Mellon University.

L.P. Chew and R.L. Drysdale, 1986. Finding Largest Empty Circles with Location Constraints *Dartmouth Computer Science Technical Report PCS-TR86-130*

M. de Berg et al. *Computational Geometry: Algorithms and Applications (2ed)*. Berlin: Springer, 2000. pp 185-197.

A. Okabe et al. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams (2ed)*. Chichester: John Wiley and Sons, Ltd, 2000. pp 43-57.

F.P. Preparata and S.J. Hong. 1977. Convex Hulls of Finite Sets of Points in Two and Three Dimensions. *Communications of the ACM*, v.20, n.2, pp 87-93.

F.P. Preparata and M.I. Shamos. *Computational Geometry: An Introduction*. New York, NY: Springer-Verlag, 1985. pp 251-253.

M.I. Shamos, 1978. Computational Geometry, Ph.D. thesis, Yale University.

G.T. Toussaint, 1983. Computing Largest Empty Circles with Location Constraints. *International Journal of Parallel Programming*, v12.5, pp 347-358.

Unbiased Congressional Districts

Alex Benn
abenn1@swarthmore.edu

David German
dgerman1@swarthmore.edu

Abstract

This paper presents a strategy for dividing a state into congressional districts using a modified version of Smith and Ryan's recursive shortest splitline algorithm (2007). Our strategy reduces the cost of the computation by approximating the population of a ZIP Code Tabulation Area (ZCTA) as a point mass at its population centroid. If there are N ZCTAs in the state with E total edges, and k districts to be created, our algorithm runs in expected $O((N^2 + E) \cdot \log k)$ time.

1 Introduction

The determination of congressional district boundaries within a state is often a controversial process. Typically, any solution that gives contiguous, equipopulous districts is legal. Therefore, the majority party in the state legislature has wide latitude to draw districts that bolster its chances in congressional elections. Such gerrymandered districts will typically disperse regions of strong support for the opposition party among the districts so that the majority will control most or all seats.

We adopt the following definitions. An *unbiased districting* is a division of a region into equipopulous districts that is developed without any information about the residents' voting preferences. A *gerrymandered districting* is a division of a region into equipopulous districts such that the region's majority party is the majority party in every district.

2 Related Work

The literature contains several proposals for generating or maintaining unbiased districts that optimize some geometric criterion.

Given an initial districting, Helbig et. al. heuristically maximize district compactness using an iterative linear programming technique (1972). In their method, the region is discretized into m existing political zones, such as counties or census tracts, that are small compared to the whole state and approximately equipopulous. They then proceed as summarized in Algorithm 1. Constraint A is the one-vote constraint: each zone belongs to exactly one district. Constraint B is the equipopulation constraint. It relies on the assumption that zones are equipopulous, but this restriction could easily be lifted by rewriting the constraint as an inequality on the population sum, with some deviation tolerance. The value function to be minimized is simply the summed population-weighted distance of zones from the centroids of their districts. Since finding a minimum by brute force takes $O(2^m)$ time, the transportation algorithm is used; the details are out of the scope of this paper. Note that this method does not guarantee that districts will be contiguous.

Macmillan presents an innovative solution to the contiguity problem (Macmillan, 2001). Macmillan first presents the connectivity method of checking contiguity of a region, which is a streamlined, but still $O(m^3)$, version of connectivity matrix multiplication. Macmillan improves on this algorithm by initially assuming that the regions in the districting are already connected, and what is being examined is the decision to either add a single zone to or remove a single zone from the current region. Since the only way the region can become disconnected is if adding or removing the zone breaks contiguity, we need only examine the single zone and the zones it borders. Using Macmillan's switching point technique, it can be determined quickly whether the operation breaks contiguity. An initial districting can then be updated in response to population movement

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 38–44,
Computer Science Department, Swarthmore College

Algorithm 1 Helbig et. al. linear transport

Let ϵ be an arbitrary stagnation threshold.
Let m be the total number of zones.
Let n be the number of districts.
$g = m/n$
Let (u_i, v_i) be the centroid of zone i.
Let p_i be the population of zone i.
repeat
  for All districts j **do**
    Compute (a_j, b_j), the centroid of district j.
    for All zones i **do**
      $d_{ij} = \sqrt{(a_j - u_i)^2 + (b_j - v_i)^2}$
      if zone i is in district j **then**
        $x_{ij} = 1$
      else
        $x_{ij} = 0$
      end if
    end for
  end for
  constraint A: $\sum_j x_{ij} = 1$ for all i.
  constraint B: $\sum_i x_{ij} p_i = g$.
  Using the transportation algorithm, minimize $\sum_i \sum_j (x_{ij} \cdot d_{ij} \cdot p_i)$ subject to constraints A,B.
  for All districts **do**
    Let $(\bar{a}_j, \bar{b}_j)$ be the centroid of the new district j.
  end for
until $|\bar{a}_j - a_j| < \epsilon$ and $|\bar{b}_j - b_j| < \epsilon$ for all j.

by simulated annealing. In this process, donor districts with excess population and recipient districts with a population deficit are selected at weighted-random by the size of the deviation. A zone to transfer is selected at weighted-random by the size of the resulting deviation. So long as the transfer would not break contiguity, it is accepted if it strictly improves deviation, and probabilistically accepted based on the current annealing temperature otherwise. Macmillan's algorithm does not optimize for compactness or any other geometric property, and therefore may yield results as unattractive as a gerrymander.

The shortest splitline algorithm of Smith and Ryan recursively divides the region to be districted into two regions (Smith and Ryan, 2007). If the region being divided has an even number of congressional seats, the two child regions are equipopulous. If it has an odd number, then one region will be large enough to have one more seat than the other. Smith and Ryan compute splitlines on a grid sampling of population. The details of their implementation are only given as uncommented C source code, and are therefore not summarized here.

Our literature search did not discover any existing work explicitly concerned with optimal gerrymandering. However, the generalized equitable ham sandwich algorithm of Bespamyatnikh et. al. is applicable (1999). This algorithm splits a set of $r \cdot p$ red points and $r \cdot q$ blue points into r regions with p red points and q blue points using a divide-and-conquer strategy. Using voter registration rolls, registered Democrats (blue points) and Republicans (red points) can be identified with geographic locations. Letting r equal the size of the state's congressional delegation, the majority party will control every district in the equitable ham sandwich subdivision.

3 Implementation

We have implemented a shortest splitline algorithm using the basic concepts developed by Smith and Ryan (2007), but with the adapted implementation summarized in Algorithm 2. The algorithm produces districts that are contiguous, and equipopulous to a variable tolerance parameter ϵ. Degeneracy handling is not included in the pseudocode, but has been

implemented and is discussed further in Section 3.3.

Smith and Ryan use a raster model for the region: boundary points are represented by pixels, and a splitline starts and ends at the centers of pixels. This model has the problem of fixed and arbitrary granularity: edges cannot shift by less than one pixel at either end, and the granularity is set by the resolution of the original raster image of the region. If the original image is too small, the resulting splitline may be unsatisfactory in population distribution, but if the original image is too large, the algorithm takes much longer.

Instead of working within the limitations of a raster image, we chose to use the ZIP code as our fundamental element. Populations are stored by ZIP code rather than by pixel. Splitlines start and end on ZIP code population centroids, and the determination of the population of each district is calculated by summing the populations of the ZIP codes whose centroids fall within that region. The outer boundaries of the ZIP code zones comprise the border of each district.

3.1 Data Set

ZIP codes have a number of virtues as fundamental elements for the software. They are generally sized to contain roughly even populations, so areas of high population density are broken down into smaller ZIP codes. However, ZIP code regions are determined arbitrarily by the post office and tend to follow postal routes rather than being shaped according to any metric of geographical cohesiveness.

The US Census has found it useful to map out the regions containing all addresses in each ZIP code, called Zip Code Tabulation Areas (ZCTAs). Map overlays in standard formats are available online from the US Census's website for all fifty states.[1] Our implementation extracts data from the ArcView Shapefile format.

Grubesic and Matisziw's use of census ZCTA maps for epidemiological studies (2006) helpfully clarified that ZCTAs in which three numbers are followed by an `HH` represent water surface, and those with an `XX` represent unpopulated land. For example, any water feature mostly within the `303` ZIP3 ZCTA would be labeled `303HH`.

[1] `http://www.census.gov/geo/www/cob/z52000.html`

Algorithm 2 ZCTASplit

Let ϵ be an arbitrary population deviation tolerance.
Let k be the number of districts.
Let t be the population.
Let R be the set of ZIP codes in the region.
if $k = 1$ **then**
  return R
end if
for Each edge e of a ZIP in R **do**
  If e adjoins only one ZIP, it is in the set of possible boundary edges B.
end for
Find the northwesternmost edge p in B. $\{p$ must be on the outer boundary.$\}$
$Z \leftarrow \emptyset$
Let c be the edge following p on the ZIP z that p adjoins.
while $c \neq p$ **do**
  $Z \leftarrow Z \cup \{z\}$
  while $c \in B$ **do**
    $c_n \leftarrow$ the edge following c on z.
    $c \leftarrow c_n$
  end while
  $z \leftarrow$ the ZIP that c adjoins that is not z.
  $c_n \leftarrow$ the edge on z in B that shares a vertex with c.
  $c \leftarrow c_n$
end while
$g \leftarrow \lfloor k/2 \rfloor \cdot t$ $\{g$ is the goal population.$\}$
$l \leftarrow \infty$
for All $z_1 \in Z$ **do**
  for All $z_2 \in Z$ **do**
    Draw a splitline connecting the centroids of z_1 and z_2.
    $a \leftarrow$ the population above the line.
    $d \leftarrow$ the distance between the centroids.
    if $(1 - \epsilon)g \leq a \leq (1 + \epsilon)g$ and $d < l$ **then**
      $l \leftarrow d$.
      $s_b \leftarrow (z_1, z_2)$
    end if
  end for
end for
if s_b is not defined **then**
  error *no acceptable splitline found*
else
  $U \leftarrow$ ZCTASplit(ϵ, $k = \lfloor k/2 \rfloor$, $t = g$, $R =$ ZIP codes above the splitline)
  $O \leftarrow$ ZCTASplit(ϵ, $k = \lceil k/2 \rceil$, $t = t - g$, $R =$ ZIP codes below the splitline)
end if
return $U \cup O$

Since the ZCTA maps simply delineate each ZCTA by one or more labeled polygons, no neighborhood information can be directly determined from the ZCTA shapefile. This complicates finding the boundary of a region to split, an issue that is dealt with in Section 3.2.

We have been testing our implementation with publicly available voter rolls from the state of Georgia.[2] These voter rolls contain ZIP code, latitude, longitude, and party registration. All voter entries for a single ZIP code are placed within that ZIP code, and the centroid of the population of that ZIP code is found. This population centroid is used to determine the side of a splitline on which each ZIP code falls.

3.2 Strategy

Algorithm 2 can be divided into two phases. First, ZIP codes on the boundary of the region are identified. A ZIP is a candidate to be on the boundary if it has at least one edge that no other ZIP code in the region shares. Since the census ZCTA polygons may not perfectly tessellate the state, a walk is performed to identify the actual edge. This walk begins at the northwesternmost edge in the entire set of candidate boundary edges, which bounds some boundary ZIP z_0. The edges of z_0 are traversed in the cyclical order extracted from the shapefile until an edge e that also bounds some other ZIP z_1 is reached. The walk then moves onto z_1. One edge adjacent to e bounding z_1 is in the set of candidate boundary edges. The walk returns to the boundary by proceeding in the direction of that edge. When the walk revisits the first edge, all visited ZIPs z_n are returned.

Then, for every pair of boundary ZIPs (z_a, z_b), a splitline is drawn between the population centroids. Each ZIP population centroid in the region is classified as above or below the splitline, and the populations in each half are summed. If the ratio of population above the splitline to below the splitline is within some arbitrary tolerance of the equipopulous ratio, the splitline is a legal candidate. The shortest legal candidate splitline is accepted, and the algorithm recurses on the two ZIP sets returned.

[2] `http://grso.uga.edu/voter/`

3.3 Degeneracies

3.3.1 Rivers and Lakes

HH regions representing rivers can wander from the border across large swaths of a state, making it difficult to identify ZIP codes on the boundary. Simply removing the water from the dataset directly places a large number of interior ZIP codes on the apparent boundary, which is no improvement. When a split is made with water preserved, allocating water that crosses the boundary to one side or the other produces similar problems upon recursion.

The solution is to allocate the entire body of water to both regions. The boundary walk then proceeds along the edge of the water until it eventually returns to the proper boundary on the other side. The resulting boundary will thus include protrusions that are not actually part of the intended district. While potentially bizarre in appearance, these protrusions are not disruptive to the algorithm because they contain no population. Since regions will eventually contain water that is entirely disconnected from their actual land area, the boundary edge traversal must begin on the northwesternmost edge in B that adjoins a populated ZIP in the region.

3.3.2 Eccentric Splitlines

Consider the line containing the segment that connects the centroids of two boundary ZIP codes z_1 and z_2. If this line crosses the region boundary at any other ZIP code, the subsets of ZIPs produced by splitting on the line may be misidentified. We call such lines eccentric splitlines.

As a simple example, consider z_1 and z_2 on the boundary of a circular region, with populations distributed such that the line connecting the centroids is perpendicular to the local boundary. A splitline from z_1 to z_2 appears to divide the region in half, when it actually should separate z_1 and z_2 from all other ZIPs.

To work around this degeneracy, we simply reject any eccentric splitline. The check for eccentricity is fast. Recall that the boundary walk produces the boundary ZIP codes in order. After classifying all ZIP codes as above or below the line, we start at an arbitrary ZIP code on the boundary and look up its classification. We then consider each subsequent boundary ZIP in order. If the classifications of ad-

jacent ZIPs differ, but neither is a boundary ZIP, the splitline is eccentric.

3.3.3 Discontiguity

ZIP code boundaries are stored in the shapefile as ordered lists of edges. Since the boundary-walk algorithm must be able to wrap around the edge list from the last edge to the first, each ZIP code cannot store more than one polygon. However, the dataset may include more than one polygon with the same ZIP code label. Examples of situations where this may occur include groups of islands with the same ZIP code, and ZIP codes that have a river cutting through the middle.

The solution to this problem is to symbolically perturb the dataset by re-labeling multiple polygons that have the same ZIP code. Data is read in a linear fashion, so the first polygon with the ZIP code 30304 will keep that label, but subsequent polygons will be re-labelled 30304I, 30304II, etc. Population data is only inserted into the polygon that happens to have been inserted first, and therefore happens to be labeled 30304. Since the population centroid is used to determine splitline information, this has no negative effect on the accuracy of the resulting splitline.

3.3.4 High-Degree Vertices

There are two other degeneracies that result from peculiarities of the ZCTA dataset. These must be dealt with on a case-by-case basis.

The first situation that may arise is a polygon making contact with the boundary at a single vertex. This is a special case of a situation called a high-degree vertex. The implementation is designed to deal with vertices that are incident to at most three edges: we walk the boundary of the current polygon until we run into another polygon on the boundary, then we jump to that polygon and keep walking. If the abutting polygon only touches the boundary at a single point, however, then the boundary continues on some other polygon that also touches that vertex. Since the algorithm does not know how to get to that next polygon, it must search through all edges in the candidate boundary edge list to find the one that continues from this vertex. It is important to look for an incident edge that has not yet been traversed.

Another degeneracy occurs when a polygon has only one edge on the boundary. The implementation determines the direction to walk along each polygon by looking one edge ahead and one edge behind. If the first boundary edge is also the last boundary edge, then the implementation cannot determine this direction. It must then examine the vertices; the direction selected is that which carries the walk away from the last vertex seen.

3.4 Asymptotic Analysis

Say the dataset has N ZCTAs and E edges, and we wish to form k districts. Let us first consider the topmost level of recursion. The algorithm first examines each edge to determine whether it appears in more than one polygon in order to build a list of candidate boundary edges. This takes $O(E)$ time. Next, we traverse the polygons that are incident to the boundary edges to cull for internal water features and polygon edge misalignments. Our algorithm only walks those polygons that sit on the boundary of the region. For the next step, we make two assumptions. First, we assume that approximately $\sqrt{N}$ ZCTAs lie on the boundary, and thus $\sqrt{N}$ polygons. Second, we assume that all polygons in the dataset contain an approximately constant number of edges. walking the boundary is then an $O\left(E + \sqrt{N}\right)$ operation at each level.

The final portion of the algorithm examines all candidate splitlines, that is, the set of lines connecting any pair of boundary ZIP code population centroids. Assuming there are $\sqrt{N}$ centroids, there will be $O\left(\sqrt{N}^2\right) = O(N)$ splitlines. For each splitline, we must calculate the population above and below that splitline, taking $O(N)$ time. Thus the runtime for this portion of the algorithm takes $O\left(N^2\right)$ time.

Each level of recursion must execute both of the above operations. There are $log(k)$ levels of recursion, so the expected runtime for the overall algorithm is $O\left((N^2 + E)\log k\right)$. The worst-case runtime occurs when all ZCTAs lie on the boundary of the region to be split, resulting in a runtime of $O\left((N^3 + E)\log k\right)$.

4 Results

We have tested our software on the state of Georgia. Since our implementation is proof-of-concept, we let the registered voters of Georgia approximate the population. Our software produces lists of ZIP

codes by district, and generates KML output for visualization with Google Earth.

Georgia has 13 congressional districts. Our software successfully computes a shortest splitline districting for population variation tolerances of 3% and higher. Since there are only about 1000 ZIP codes in Georgia, the regions at low levels of recursion can have very few non-eccentric splitline candidates, resulting in failure to find an acceptable splitline at tighter tolerances. If better population equity is required, smaller tabulation regions should be used.

Figure 1 shows the results with a 3% tolerance, and Figure 2 shows the results with a 5% tolerance. As expected, the splitlines are noticeably longer in Figure 1, particularly in the north of the state.

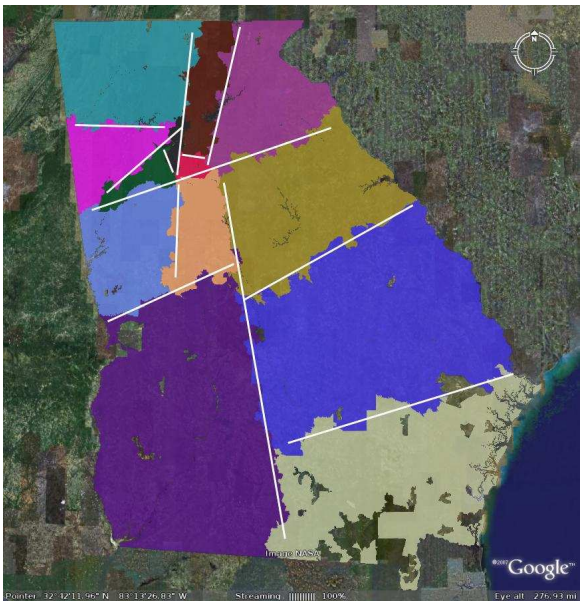

Figure 1: 13 districts attempted, 3% tolerance.

Of the approximately 3.5 million registered voters in the Georgia data, only 102,227 have a declared party affiliation. 63% of those with a declared affiliation are Democrats; consequently, Democrats appear to carry 10 of the 13 districts at 5% population tolerance. Since so few voters have formal affiliation, this result has no value as a prediction of actual outcomes.

5 Conclusion

The software we have developed successfully districts Georgia. Since there are relatively few ZIP

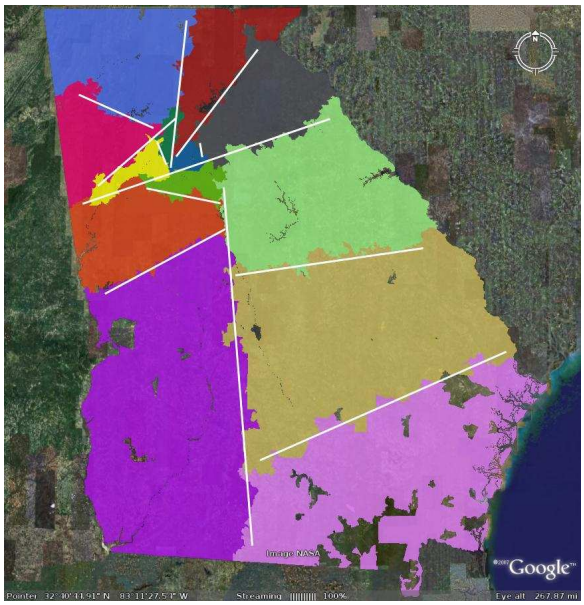

Figure 2: 13 districts, 5% tolerance.

codes per state, it runs quickly in practice: Georgia, with about 1200 ZIPs, is districted in 210 seconds on a modern workstation. Qualitatively, the results are reasonable. The Atlanta area is partitioned into small, compact districts, with larger but still simple polygons outstate.

The districts produced by using ZCTAs as the fundamental zone are certainly preferable to gerrymandering, but have some quality issues. As seen in Georgia, high equipopulation precision is not attainable with such large quanta. Moreover, the degeneracy handling required to cope with idiosyncrasies of the ZCTA data set can distort results. In particular, the eccentric splitline handling will reject most lines that closely parallel a region boundary, even if that line would be a desirable solution. Finally, ZIP boundaries themselves are arbitrary, irregular, and potentially open to political manipulation.

The natural solution is to replace ZCTAs with some other, finer tessellation of convex polygons. In the extreme, this tessellation could be the Voronoi diagram on the set of all voters. However, the expected quadratic and possible cubic running times will penalize performance severely for higher-frequency sampling. It may be possible to remove a factor of N from the runtime by building a quadtree index of voters, with each node storing the entire population descended from it.

6 Acknowledgments

We thank Andy Danner for advising our project.

References

Micah Altman. 1997. Is automation the answer: The computational complexity of automated redistricting. *Rutgers Computer and Law Technology Journal*.

Sergei Bespamyatnikh, David Kirkpatrick, and Jack Snoeyink. 1999. Generalizing ham sandwich cuts to equitable subdivisions. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 49–58, New York, NY, USA. ACM.

Tony H. Grubesic and Timothy C Matisziw. 2006. On the use of zip codes and zip code tabulation areas (zctas) for the spatial analysis of epidemiological data. *Int J Health Geogr.*, 5(58).

Robert E. Helbig, Patrick K. Orr, and Robert R. Roediger. 1972. Political redistricting by computer. *Commun. ACM*, 15(8):735–741.

Jorg Kalcsics, Stefan Nickel, and Michael Schroder. 2005. Towards a unified territorial design approach - applications, algorithms, and gis integration. *Journal of Geographical Systems*, 13(1):1–56.

W. Macmillan. 2001. Redistricting in a gis environment: An optimisation algorithm using switching-points. *Journal of Geographical Systems*, 3(2):167–180.

Warren D. Smith and Ivan Ryan. 2007. Gerrymandering and a cure - shortest splitline algorithm. `http://www.rangevoting.org/GerryExamples.html`.

Optimal Double Coverage In The Art Gallery

Scott Dalane
sdalane1@swarthmore.edu

Andrew Frampton
aframpt1@swarthmore.eu

Abstract

We plan to examine a well known visibility problem termed the art gallery problem. The general idea behind the problem is that a museum wants to minimizethe number of cameras present in an art gallery while still recording everythingthat happens in the room. We take the problem a step further and investigate the optimal arrangement of cameras so that every spot in the room is covered by not just one, but two cameras. This will allow constant surveilance even if any one camera were to fail. We do this by using a bottom-up approach where we use a three coloring algorithm to find the optimal placement for single coverage and a basic dobuble coverage of a polygon representing a floorplan. We then use a optimization algorithm with a line of sight algorithm to check each camera of the double coverage and remove any unnecessary cameras, leaving the optimal double coverage of the polygon.

1 Introduction

The problem is named after art galleries because the art needs to be under surveilance at all times and they tend to be shaped in irregular ways which can make this security difficult. The problem was first proposed by Viktor Klee in 1973 (Klee, 1979). It has since been proven that a maximum of $\lfloor n/3 \rfloor$ cameras will be needed, n being number of vertices in the polygon that is the room, to cover the entire room. This is known as the Art Gallery Theorem and was stated by Vaclav Chvatal (Chvatal, 1973). For us to tackle this problem we designed a program that takes a set of points that make up the polygon that will be guarded, and then returns an image of the floorplan with the cameras placed upon it. By using a three coloring algorithm on each of the points in the triangulation determine the ideal single camera placement and the basic double coverage placement of cameras, which can then be optimized.

2 Triangulation

In order to perform the 3-coloring on the points of the polygon, it is first necessary to break the polygon down into its constituent triangles, so that we can accurately determine whether the 3-coloring is accurate. We do this by finding any splitpoints within the polygon so that it can be seperated into monotone polygons. These monotone polygons are then individually triangulated, then put back together to create an ideal triangulation of $n - 2$ triangles for n points.

2.1 Finding Split Points

The first step in our triangulation algorithm was to locate any split points in the polygon that we can use to break the polygon down into smaller, monotone polygons to triangulate. The split point is a point that is determined to be interior to its neighboring points on the y-axis, in other words, when the point causes the polygon to be convex on the top or bottom. This is done by going through each point and shooting a rays up and down the point's x-axis (Seidel, 1991). We then use helper functions to determine whether or not the rays intersect any other segments of the polygon. If both rays hit a segment of the polygon, they are then tested on the basis of how many times they intersect the polygon to determine whether they are interior to the polygon. If the point and its rays

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 45–48,
Computer Science Department, Swarthmore College

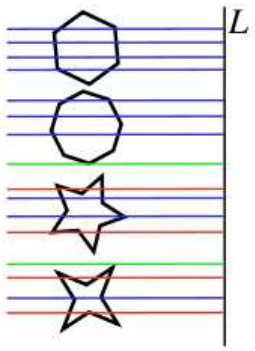

Figure 1: The top two polygons are monotone because every line only intersect the polygon twice.[1]

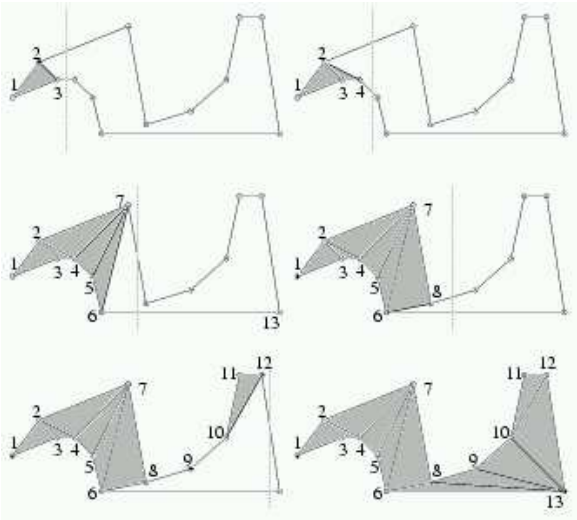

Figure 2: An example of triangulation around start points.[2]

meet these criteria they are then flagged as a split point and the polygon is then split into three polygons, one to the upper right of the point, one to the upper left, and one beneath the point, with the bases of the upper two polygons and the top of the lower polygon formed by the rays along the x-axis of the point. This continues until the original polygon has been completely broken down into monotone polygons (de Berg et al., 2000).

2.2 Monotone Triangulation

Since monotone polygons have no interior points on the y-axis, they get rid of several possible degeneracies when being triangulate as opposed to trying to triangulate the entire poylgon at once. Once the monotone polygons have obtained, each point within them is checked as the start point for the triangulation using a method similar to the one used for finding split points, but this time we check to see if they are interior to their neighboring points in regard to the x-axis. If such a point is found, then it is chosen as a start point, otherwise the split point is chosen. The triangulation algorithm then walks around the polygon's points using a list of connections stored in each point to determine the points it is connected to, and if there is no connection to the start point, then one is added and a line is formed. The shared point between the two previous points is added and then the three are saved as a triangle, and this continues until the entire polygon has been traversed. After each monotone triangle has been triangulated, they are all put back together in the polygon, and the horizontal rays from the split points are removed, creating an optimally triangulated polygon (de Berg et al., 2000).

2.3 Handling Degeneracies

It is worth noting that the algorithm that we have come up with in this is capable of handling just about any degeneracy that we encountered. If the polygon is concave, our algorithm can deal with the problems of possible outside segments as well as any problems that may arise from having a vertex in the middle of the polygon. the outside segment problem is done by our suprisingly simple intersection test which takes the modulus of the number of interstections that the ray encounters with both edges and vertices. If the number of intersections is even then the resulting modulus is zero, and the ray is considered outside and discarded, as was previously explained in section 2.1.

3 Optimizing Camera Placement

3.1 3-coloring and Camera Placement

With the triangulation in place it is now possible to perform an easy 3-coloring of the polygon, so that cameras can be placed. Using the triangulation within the polygon, we can now proceed to use

[1] image from http://en.wikipedia.org/wiki/Monotone_polygon

[2] from http://www.cs.ucsb.edu/s̃uri/cs235/Triangulation.pdf

three coloring to determine where to place the cameras. Using the list of triangles that was created during the monotone triangulation, our algorithm takes the first triangle from the list and colors each point a different color. The algorithm then looks for triangles that share sides with the previously colored triangle and color the remaining point is then colored based on what coloring of the other two points are. This continures until all of the triangles in the list have been colored, with the number of the points with each coloring stored as an integer. The integers are then compared with one another, cameras being placed at the color with the lowest integer value, creating optimal single coverage for the polygon (Urrutia, 1991) . But in order to achieve double coverage another set of cameras have to be placed at the next smallest coloring to provide a basic double coverage which can be optimized.

3.2 Visibility Graph

Now that the cameras have been placed it is time to determine what points they see and and what how many cameras see each point, assuming that if a camera could see a point then it generally sees the area surrounding the point, barring degenerate cases which are taken care of when the cameras are optimalized later on. Using a line of sight function,we created a visibility graph by traversing every point from each camera, drawing a line between the two points, making sure that the line did not intersect a segment of the polygon and remained inside of it. By doing this we then built up a list of each point seen by each camera, with thenumber of cameras seeing each point, allowing our heuristic to optimalize the camera placement (O'Rourke, 1987).

3.3 Camera Optimalization

Now we have our optimalize the camera placement by going through each camera and seeing if it can be removed from the polygon, updating the each point it sees and decrements the number of cameras that can see them accordingly. If one any point's count of cameras that can view it drops below 2, then the camera has to be replaced, otherwise, it remains taken off and the next camera is then checked. After the optimization algorithm is complete, the cameras left are the optimal.

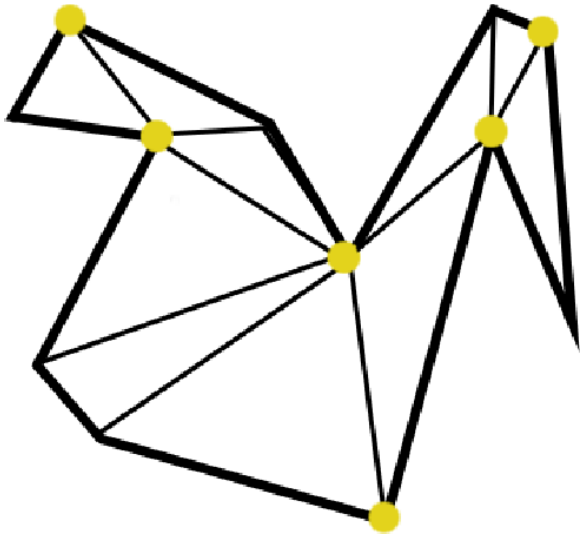

Figure 3: Simple double coverage on a test polygon using only 3-coloring.

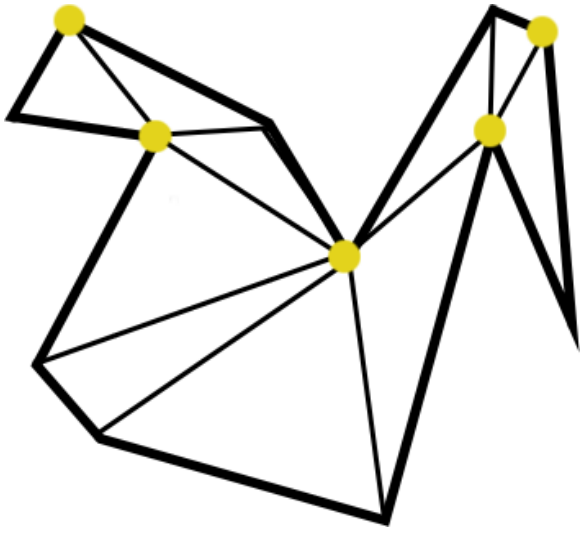

Figure 4: Optimized double coverage using a visibility graph and optimization.

4 Results and Conclusion

After testing our algorithm on progressively more complex polygons, our algorithm easily does better than $\lfloor 2n/3 \rfloor$ camera placement for double coverage of cameras outside of the the worst case, while the entire program runs in n^2 time, due to the fact that the points are looked at multiple times espescially during the optimalization step. It also proves fairly adept at making a strong placement for coverage greater than double, although it needs to be more thoroughly tested to make any conclusions in that regard.

References

V. Chvatal. 1973. A combinatoral theorem in plane geometry. Number 18, pages 39–41.

M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. 2000. Computational geometry: Algorithms and applications (2nd ed.). pages 45–61.

V. Klee. 1979. Some unsolved problems in plane geometry. volume 52, pages 131–145.

Joseph O'Rourke. 1987. Art gallery theorems and algorithms. pages 11–23.

Raimund Seidel. 1991. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. In *Computational Geometry The- ory and Application, vol. 1, no. 1, pp. 51-64.*

Jorge Urrutia. 1991. Art gallery and illumination problems.

Voronoi Natural Neighbors Interpolation

Chris Harman
charman1@cs.swarthmore.edu

Mike Johns
mjohns2@cs.swarthmore.edu

Abstract

Implementing point cloud to grid conversion for digital elevation maps (DEM) presents one with many options for interpolation and we intend to explore algorithms for interpolation during the conversion process with a specific focus on Voronoi natural neighbor interpolation. By partitioning the environment into Voronoi cells and using the information from neighboring cells and their respective generating point's elevation, we can achieve aesthetically pleasing interpolation results with runtimes competitive with lower-order interpolation algorithms. We compare our natural neighbors interpolation method with a linear interpolation method and a regularized spline interpolation method quantitatively using cross-validation and qualitatively by rendering the interpolted meshes using GRASS. [1]

1 Introduction

When dealing with discrete sets of elevation data such as elevation information collected using methods such as laser range finding or LIDAR, it is often desirable to be able to estimate or predict the values of unsampled locations using the available information. One application where interpolation can be useful is visualization of elevation data. Point elevation data obtained through LIDAR or other remote sensing methods is not always uniformly sampled. One way to visualize the data set would be to simply triangulate the point set into a triangular mesh and render that mesh, but the resulting image would not model the underlying data well. By using different interpolation methods on the original elevation data, more accurate rendering can be performed and depending on the method of interpolation, the resulting visualization can change dramatically.

There are many interpolation methods available to estimate these unsampled values, and the different methods usually offer tradeoffs between the accuracy of the prediction and the efficiency of the computation. One of the simplest interpolation methods would be to assign each unsampled location a value based upon which location within the set of known values it lies closest to. Although this simple method is computationally efficient, the resulting function is not continuous everywhere, specifically along the edges where locations are equidistant from the samples. In most cases these discretized estimations will not be as accurate as the results from other more computationally expensive interpolations. The discontinuities in the resulting function also do not model the sampled data in an aesthetically pleasing manner. Methods such as interpolation by regularized spline with tension result in functions which have regular derivatives of all orders everywhere allowing for analysis of surface geometry as well as improved accuracy in estimation (**?**).

Natural neighbors interpolation provides a good tradeoff between computational efficiency and accuracy. In this method the value of an unsampled point is determined through a weighted average of the values of the interpolation points neighbors within the sample set. These natural neighbors are determined by finding which Voronoi regions from the original point set would intersect the Voronoi region of the interpolation point, if it were to be inserted. The resulting function is continuous everywhere within the convex hull of the sample set and mimics a taut rub-

[1] http://grass.itc.it/

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 49–53,
Computer Science Department, Swarthmore College

ber sheet being stretched over the data. Our hope is that Natural Neighbors Interpolation will provide a computationally efficient method with which to accurately visualize elevation data.

2 Related Work

There are an overwhelming number of options when choosing which method to use for interpolation when converting from a point cloud to a grid for a DEM. (**?**) details many low-level routines for interpolation including: the level plane; linear plane; double linear; bilinear; biquadratic; Jancaitis biquadratic polynomial; piecewise cubic; bicubic; and biquintic interpolation method. (**?**) draws the conclusion that higher- order interpolation methods will always outperform those with linear complexity in terms of modeling the terrain accurately. The problem with choosing higher-order interpolation routines is that the computational backlash is not necessarily proportional to the gain in accuracy; this begs the question: How do we balance modeling accuracy and compuational efficiency?

Other interpolation methods such as the regularized spline with tension described in (**?**) attempt to balance computational efficiency and modeling accuracy. The regularized spline with tension deliberately attempts to smooth the surface being iterpolated and tweaks the mesh appropriately. Though a little less straightforward than other lower-level methods of interpolation, this method is a good benchmark for performance and modeling accuracy; as such, we will use it as a comparison for our Voronoi natual neighbors method in this paper. We expect Voronoi natural neighbors interpolation to represent a desirable balance between computational complexity and aesthetics.

3 Methods

3.1 Computing the Delaunay triangulation

The Delaunay triangulation for a set of points P is a triangulation DT(P) such that no point in P is within the circumcircle of any triangle within the triangulation. This is also the triangulation which maximizes the minimum angle within the triangulation. DT(P) is computed using a randomized incremental approach as outlined in (**?**). Each point in P is inserted incrementally and the Delaunay triangulation for that set of points is computed. Initially the triangulation consists of a single triangle which is defined to contain all of the points within P. As each point is inserted, the appropriate edges are added to the current Delaunay triangulation so that it remains a triangulation although it is not necessarily still a Delaunay triangulation. In order to ensure that no point within the current triangulation lies within the circumcircle of any triangle, illegal edges must be flipped. An illegal edge is one where the circumcircle defined by the three vertices of one adjacent triangle contains the outlying point from the other adjacent triangle. Special care must be taken when legalizing edges where one or more of the vertices involved belong to the initial bounding triangle. Only edges whose adjacent triangles have been changed due to the insertion of the new point can be illegal since the triangulation was legal beforehand. As a result when an edge is illegal and must be flipped, the other edges incident to the involved triangles must also be legalized. Since the angle measure of the triangulation increases with each edge flip and there is a maximum angle measure for the given set of points, the edge legalization is guaranteed to terminate. Once all of the points have been inserted into the triangulation, the vertices of the initial bounding triangle and all of the edges incident to these three vertices must be removed from the triangulation.

3.2 Transforming to the Voronoi Diagram

The Voronoi Diagram of a set of points P Vor(P) can be computed easily given its dual DT(P). Each triangle within DT(P) corresponds to a vertex in Vor(P). Each edge in DT(P) corresponds to an edge in Vor(P). For a triangle in DT(P), the location of the vertex in Vor(P) can be calculated by determining the center of the circle circumscribed by the three vertices of the triangle. This point can be found by determining the intersection of the perpindicular bisectors of each edge in the triangle. Each edge in DT(P) is adjacent to two triangles and corresponds to an edge in Vor(P) connecting the two vertices which are the duals of these adjacent triangles. Using this dual transformation it is a simple procedure to compute Vor(P) given DT(P).

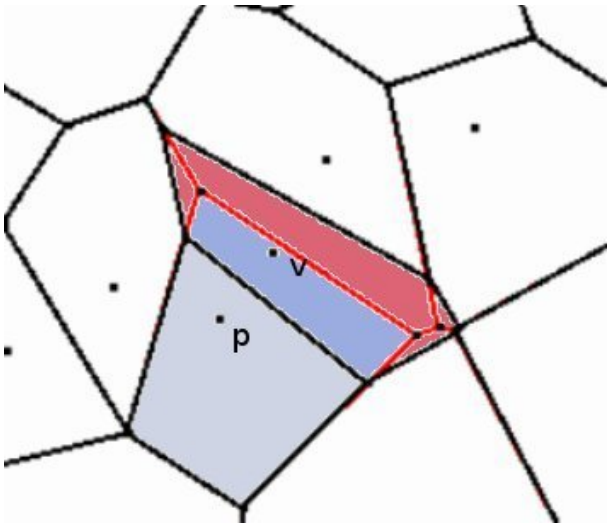

Figure 1: This figureashows the Voronoi cell for the inserted point v overlayed on the Voronoi diagram for the original point set. One of v's natural neighbors, p, is shown with its Voronoi cell shaded. The area stolen by the insertion of v can be seen in the overlap between the two Voronoi cells.

3.3 Computing the Natural Neighbors Interpolant

When computing the natural neighbors interpolant, it is important to intuit the relative 'neighborliness,' as (**?**) calls it, of adjacent Voronoi cells. These neighbors are the points within the original point set whose Voronoi cells intersect the Voronoi cell of the interpolated point, if it were to be added to the point set. The interpolated value is computed as a weighted average of the area stolen from each neighbor by the insertion of this point.

To compute this weighted average, we first compute the Delaunay triangulation of the original set of points P. Given DT(P) we need to calculate the Voronoi cell of the point v, whose value we want to estimate, if it were to be added to P. Since the insertion of v will only alter DT(P) and consequently Vor(P) locally, we do not need to recompute the entire Delaunay triangulation. We only need to compute a Delaunay triangulation for the points which would be neighbors of v. In order to determine this local point set we can use the existing triangulation DT(P)and corresponding Voronoi diagram. The area stolen from each neighbor of v can be computed by finding the difference between the area of Voronoi cell in Vor(P) and the area within the local Voronoi diagram. The interpolated value for v is then calculated as: $\sum_{\text{neighbors of} v} \frac{\text{areastolen}}{\text{area of the Voronoi cell of} v} \alpha$ where alpha is the value, possibly an elevation value, for the specific neighbor.

To compute DT(P) and transform to Vor(P) as described earlier takes O(n logn) time where n is the size of P. Vor(P) must be calculated once for the original point set. The local Voronoi diagram must be calculated for each interpolated point. Since the size of this local subset of P does not depend on n but rather the distribution of points in P and the number of neighbors that the inserted point would have, we can assume that for large point sets, this local set of points will be much smaller than n and will not depend on n. This means calculating the local Voronoi diagram will take a relatively constant amount of time independent of the size of P. In order to determine which points belong in this local point set we determine which triangle in DT(P) the point lies within. We can follow the outgoing edges from the vertices of this triangle to determine the appropriate neighbors to include. This process again depends on the the size of the local point set which does not depend on n. To calculate the weighted average we must locate Voronoi cells a constant number of times. We can use the DAG search structure for DT(P) to locate points and follow the dual pointers into Vor(P). Locating a Voronoi cell will take O(logn) time. For each point we need to interpolate we need to perform a constant number of logn searches plus a relatively constant amount of work to compute the local Voronoi diagram, so to interpolate k points for a point set of size n would take O((n+k)logn) time.

4 Results

A set of 50 points within a 400 by 400 region were chosen. Elevation data for these points was then obtained by sampling an elevation image which can be seen in Figure 2 (far left). Three different interpolation methods were performed on a small subregion of the image using the 50 sampled points. The first method used was our natural neighbor interpolation algorithm. The results of our interpolation were then compared to two of GRASS' built in interpolation routines, regularized spline with tension and inverse distance weighting. The differences between the resulting interpolated images and the original image

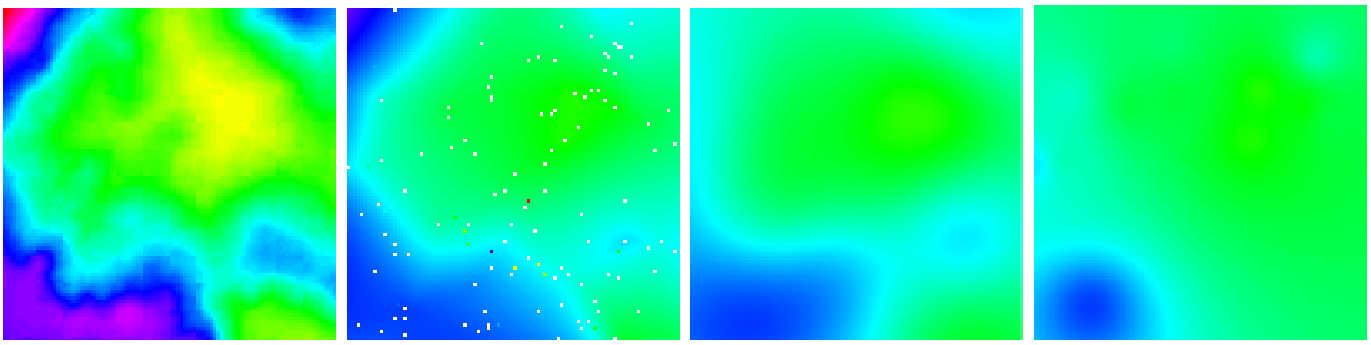

Figure 2: This figure shows the original image, the image generated using our natural neighbors interpolation routine, GRASS' regularized spline with tension and GRASS' inverse distance weighting from left to right, respectively.

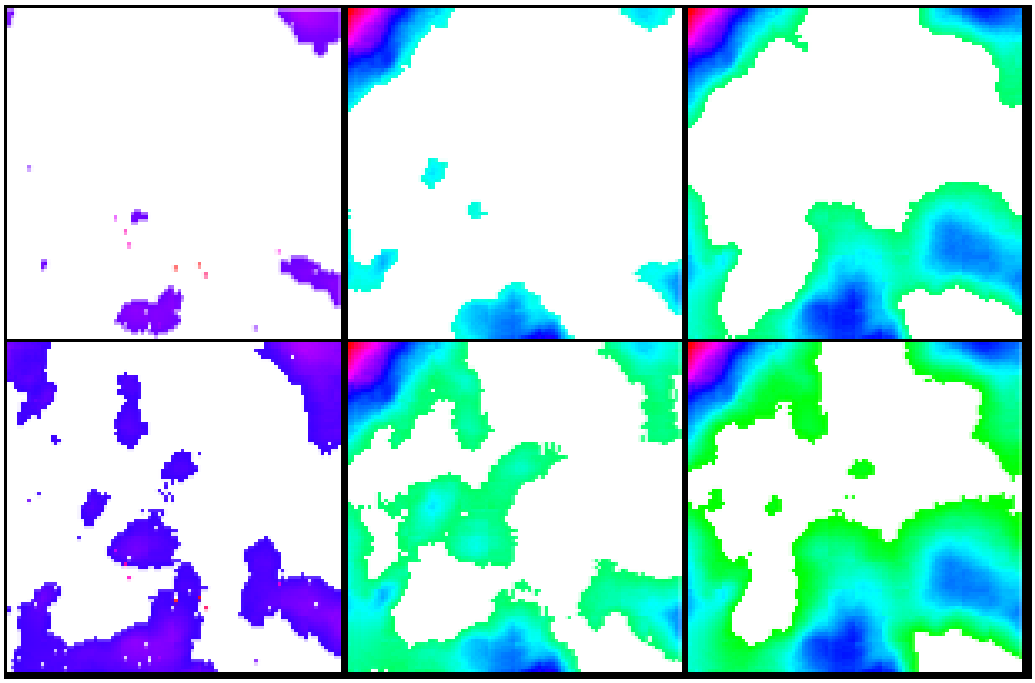

Figure 3: This figure shows the difference images generated by subtracting the interpolated images from the original image. The difference images were then thresholded to only display values above a certain value. Their arrangement is identical to the table below.

Threshold	NNI	RST	IDW
10	5.1%	13.5%	38.9%
2	28.4%	44.4%	57.1%

Table 1: This table displays the percent of the interpolated image that differs from the original image by a defined threshold.

were calculated within the appropriate region. The results were thresholded at varying levels and percentages for estimated values lying outside of the threshold were calculated in table 1.

For the given data set and sampled points, our natural neighbor interpolation routine outperformed the built in regularized spline with tension and inverse distance weighting interpolation routines. At a threshold of 10 units, only 5.1% of the pixels were incorrectly estimated by our interpolation routine compared with 13.5% and 38.9% incorrect respectively for the regularized spline with tension and inverse distance weighting routines. At a lower threshold of 2 units, our routine still dramatically outperformed the other two methods.

5 Discussion

All three methods of interpolation seemed to have trouble estimating values in relatively similiar areas. This could result from the sample points being more sparse in these areas or the frequency of the data in the original image being higher. Looking at the difference images generated for each of the interpolation routines reveals that these trouble zones for interpolation are common across each method.

Due to a lack of robustness in our interpolation routine, values at certain locations could not be estimated which can be seen with the speckling in the difference image. This leads to a greater percent incorrect which leads us to believe that our method would perform even better given a more robust implementation. The relative aesthetic advantage of natural neighbors interpolation over the regularized spline with tension method and the inverse distance weighting method is readily apparent. It seems that the natural neighbors method is much more capable of dealing with a sparse set of points to interpolate. This advantage could be because the tesselation underlying the interpolation routine extends beyond the boundary of the eventual image, allowing for better interpolation around the boundary of the image relative to the other methods.

We chose not to include runtime comparisons because our method took far longer to finish compared to the others. Each time we ran the natural neighbors routine, it required approximately an entire night to complete. When GRASS interpolated using its built-in routines, they both finished in about one second - negligible compared to our runtime. The extreme expedience of GRASS' routine is likely due to the sparse number of points that were interpolated over.

6 Future Work

There are several improvments to our interpolation method that would greatly enhance the usefullness of our implementation. The first necessary improvement centers around the degeneracies hinder the proper triangulation of the environment. Having a more robust Delaunay triangulation would remove the uninterpolated holes in the final image. This could have also been worked around easily by averaging over the gaps in interpolation once the Delaunay triangulation failed. Improving our point insertion method during the interpolation step would help cut down on some of the computational costs as well.

Creating a more extensive set of comparisons by varying the number of points interpolated over, varying the image frequency and complexity and including other interpolation methods would provide a more accurate gauge of our interpolation method's relative performance. A more comprehensive runtime comparison between methods would hopefully help determine the practical usage of our interpolation method too.

Bridge Detection By Road Detection

Jeff Kaufman
cbr@sccs.swarthmore.edu

1 Introduction

It is useful to be able to determine water flow patterns over terrain. The raw data for this task is usually collected via airborne laser range finding, or LIDAR. This yields a point cloud representing the uppermost surface of the terrain. This cloud is interpolated onto a grid where each grid cell represents a square portion of the earth's surface and it's value is the average hight of that portion. Water flow patterns can then be calculated by looking at the direction of greatest descent from each cell.

There is a problem, however, with local minima: cells from which there is no direction of descent. A common cause of this is bridges. The area over which a bridge passes shows up in the digital elevation as being of a height greater than the surrounding terrain, but for the purposes of water flow it is not there. A water flow simulation model will treat those grid cells as indicating a barrier, then, when there is in fact no impediment to water flow. The goal of this project is to identify bridges to aid in the accurate determination of water flow patterns

2 Past Attempts and Related Work

A common method for dealing with local minima is flooding. In this all grid cells, excluding those at the edges of the grid, where there is no lower adjacent cell are raised up to the height of their lowest neighbor. Repeating this will eventually rid the map of local minima. Identical results to this naive method can be achieved efficiently with a plane sweep algorithm using topological persistence. (Edelsbrunner et al., 2000) Unfortunately, as Soille et. al. (Soille et al., 2003) recognize, this loses information. Large flat areas, a common result of flooding, retain no information about their original low points and do not provide information actual flow patterns. One can still determine a possible flow pattern through it, but that pattern may be totally different from the true one.

Soille et. al. were working with a very low resolution (250m) grid elevation model to determine the water flow pattern for Europe and the problems they ran into are somewhat different from those encountered with higher resolution data. When they encountered a local minimum, specifically, it was usually because some small stream or channel went undetected with the coarse sampling. Their replacement of flooding, carving paths from local minima to the nearest lower area, makes sense for dealing with the missing channels but can of course get things wrong. As the local minima in higher resolution data are much more likely to be products of human terrain manipulation, generally in the creation of roads, a system that tags and removes bridges ought to come closer to true water flow paths than either flooding or carving.

For last year's senior conference, Manfredi and Pshenichkin (Manfredi and Pshenichkin, 2006) used a series of classifiers to tag bridges. They had a series of simple criteria that a window had to match to be tagged a bridge. They were able to detect many of the larger bridges but missed some smaller ones, as well as complex structures such as highway interchanges. Their system also had a large number of false positives, detecting vegetation and other small artifacts as bridges. They rightly point out, however, that there is not too much harm in removing them along with bridges as they are also not really there from the perspective of water flow.

One feature that they did not take advantage of is the tendency of bridges to be part of the road network. All of their classification work considered only the window that potentially contained the bridge. There has been some

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 54–58,
Computer Science Department, Swarthmore College

work on detection of of roads from LIDAR data (Clode et al., 2005), and while the final detected roads may not be completely accurate, for this task we don't need perfection. Instead we just need a rough idea of how likely a region is to be part of the road network, which can then be input to the bridge detection system.

3 Bridge Detection via Road Detection

For this project I locate bridges in two stages. In the first I determine an approximate map of the road network, a map that should generally be best in areas where the roads are in high relief. These areas correspond well to those where bridges are likely, so it should be a well suited map for the task. Second I identify local minima that are near the computed roads in order to tag road sections as bridges. Input consists of a digital elevation model in the form of a grid of floats indicating hight. Output consists of five similar models with floats indicating likelihood of being a bridge, with calibration required for the particular data set.

3.1 Road Detection

Roads are places in the terrain that are flat. Any flat area could be part of a road. Areas that are linearly flat, however, are much more likely to be roads. These would be areas where lines in one direction are flat while in other directions are not. Finally, roads tend not to bend sharply, so if there is a road in a direction we treat grid cells in that direction as being more likely to be a road.

This yields four indicators of bridge-likeness. All are computed on a series of cells representing a potential road. For every cell c in the grid we calculate 32 potential roads of a configurable length running through that point. We then find which of those series of cells has the lowest average change in steepness and call that the 'best road' centered on that cell. We also find the set of cells representing a line perpendicular to the best road and call that the 'perpendicular'. Each indicator acts on one of these two roads and yields a value attributed to c.

1. Maximum gradient. For the best road, the likelihood of it being actually a road is inversely proportional to the greatest difference between adjacent cells in the road.

2. Average gradient. Like the previous, except the average absolute difference is calculated instead of the maximum one.

3. Maximum gradient of perpendicular. The likelihood of the best road being a road instead of just a cornfield is indicated by the unroadlikeness of the perpendicular. This is calculated as for the maximum gradient.

4. Standard deviation of gradient. Even when not level, roads tend to be flat. That is, while they might sometimes have high gradients the (root mean square) standard deviation should be low.

3.2 Local Minima

A maximally simple algorithm for determination of local minima turns out to be quite effective as the data is not very noisy. For every grid cell, if no neighbor is smaller, then that cell is a local minimum. With worse data we might have a large number of these places and a small amount of flooding might be worth while. After flooding an amount small enough not to overflow bridges we have not lost much flow information and now should generally have local minima just in places where there are bridges.

4 Results

These four indicators were tested on two different examples of roads. Figures 1 and 2 show the LIDAR-derived input grids. Figures 3 and 4 show the maximum gradient indicator. Figures 5 and 6 show the average gradient indicator. Figures 7 and 8 show the maximum gradient of perpendicular indicator. Figures 9 and 10 show the standard deviation of gradient indicator.

All four indicators appear to capture an element of bridge detection. One important aspect of these indicators is that they most strongly label cells as indicating bridges when those cells are in places where they would be incorrectly

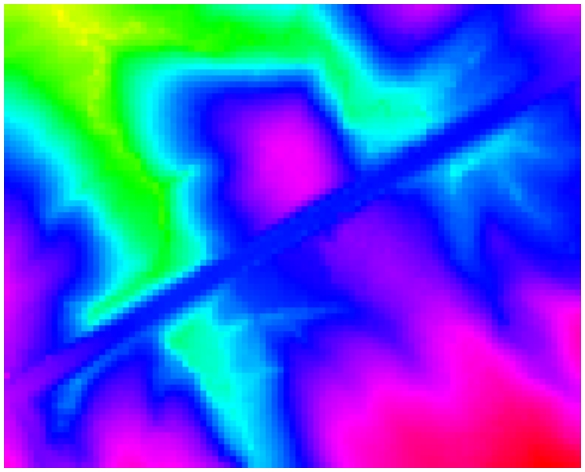

Figure 1: The input digital elevation model for the Bridge test

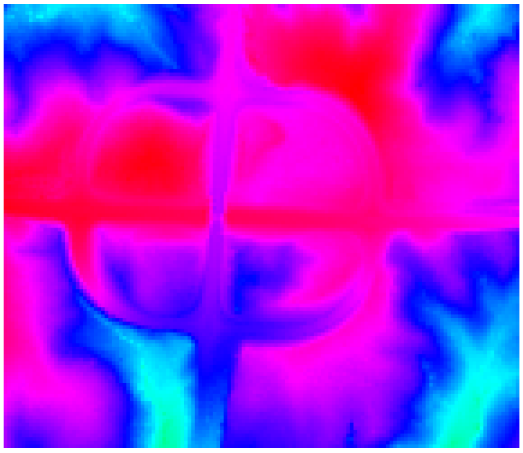

Figure 2: The input digital elevation model for the Interchange test

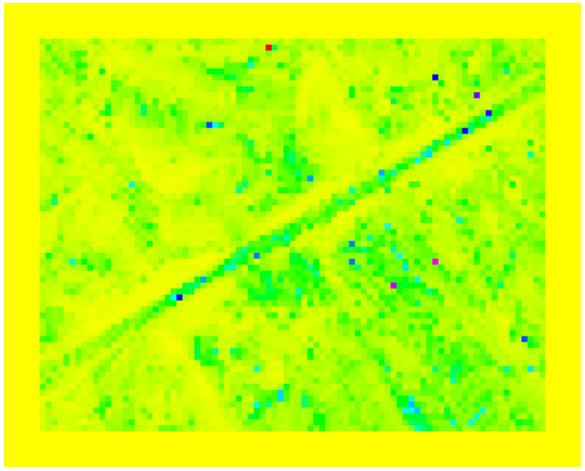

Figure 3: The maximum gradient indicator on the Bridge

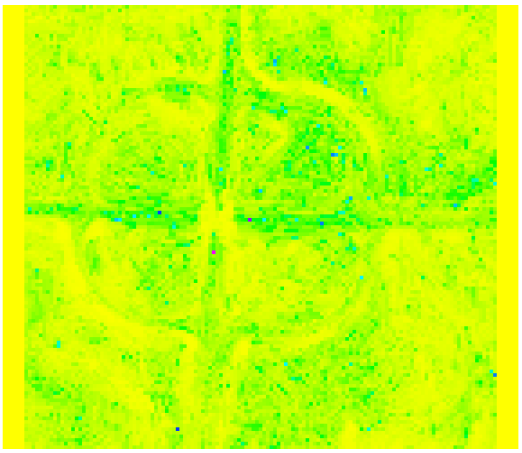

Figure 4: The maximum gradient indicator on the Interchange

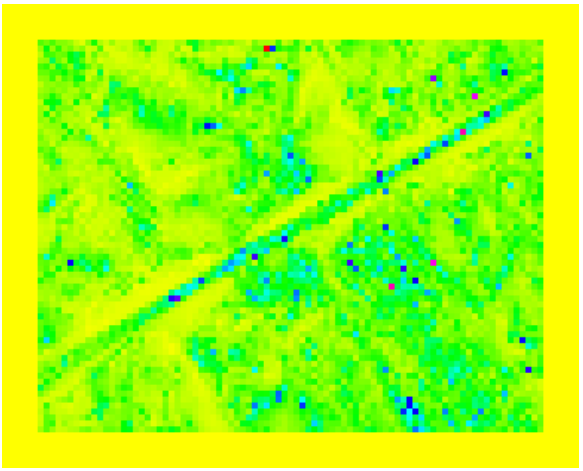

Figure 5: The average gradient indicator on the Bridge

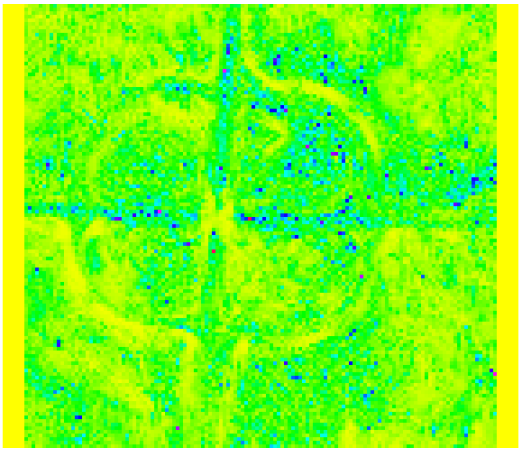

Figure 6: The average gradient indicator on the Interchange

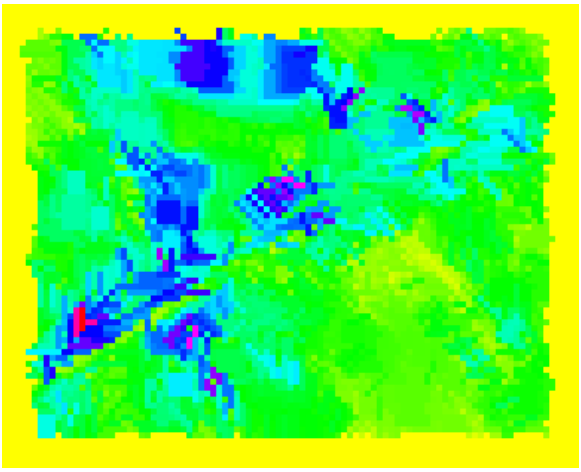

Figure 7: The maximum perpendicular of gradient indicator on the Bridge

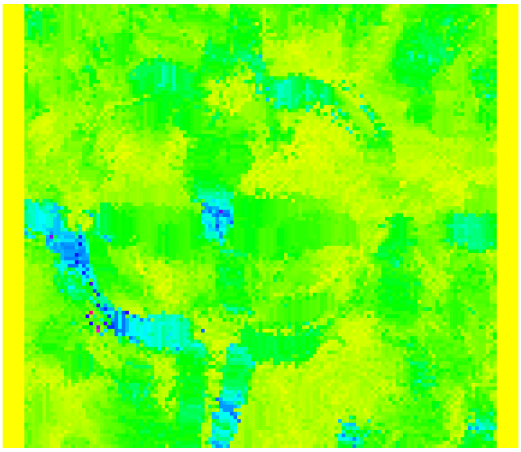

Figure 8: The maximum perpendicular of gradient indicator on the Interchange

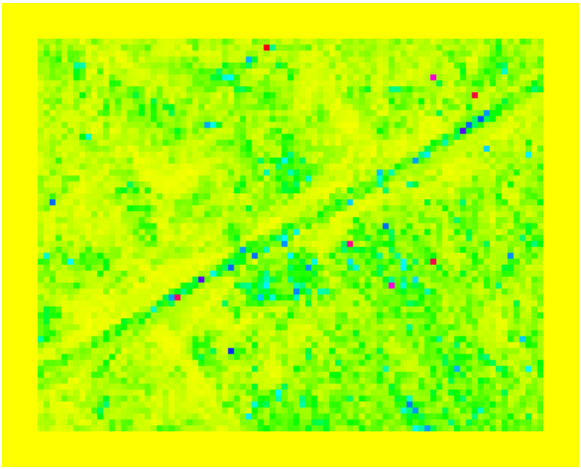

Figure 9: The standard deviation of gradient on the Bridge

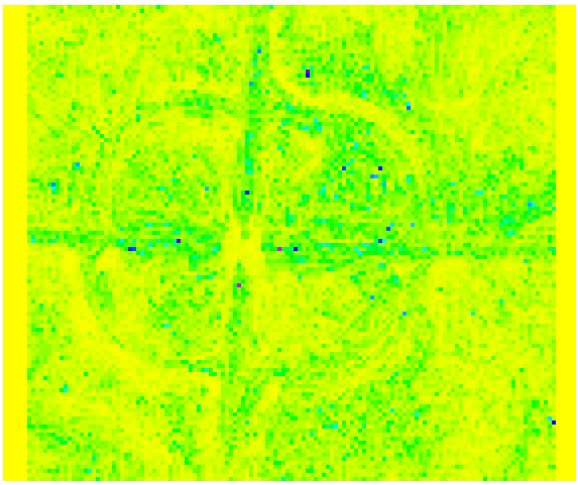

Figure 10: The standard deviation of gradient on the Interchange

impeding water flow. Actual use of these indicators on real data would require calibrating them. This would require hand tagging a small number of bridge examples and computationally determining the combination of these indicators that best fits that data.

5 Conclusions

In this paper we have shown that several relatively simple functions analyzing a digital elevation model can produce good indicators for classification of cells as to their probability of being a road. Further work would include a large scale test with calibration on a large digital elevation model. Implementation of a second pass that combined these local bridge likelihood estimates into a road network could also improve accuracy.

References

S. Clode, F. Rottensteiner, and P. Kootsookos. 2005. Improving city model determination by using road detection from lidar data. *IAPRSSIS*, XXXVI(3/W24):159–164.

H. Edelsbrunner, D. Letscher, and A. Zomorodian. 2000. Topological persistence and simplification. In *41st IEEE Symposium on Foundations Computer Science*, pages 454–463.

A. Manfredi and A. Pshenichkin. 2006. Bridge detection from elevation data using a classi-

fier cascade. http://web.cs.swarthmore.edu/~adanner/cs97/f06/papers/bridge.pdf.

P. Soille, J. Vogt, and R. Colombo. 2003. Carving and adaptive drainage enforcement of grid digital elevation models. *Water Resources Research*, 39(12).

Image Stained Glass using Voronoi Diagrams

Michael Gorbach
mgorbac1@cs.swarthmore.edu

Abstract

The geometrical concept of the Voronoi diagram was used to create an image filter providing a "stained glass" or mosaic effect on an image. The Voronoi diagram was calculated by exploiting its dual relationship with the Delaunay triangulation, which was in turn calculated using a randomized incremental algorithm and stored in a DCEL. Various methods were tried for selecting the points, including sampling from a distribution built using edge detection. Sampling using edge detection distributions was shown to provide results significantly better than uniform random sampling.

1 Introduction

Voronoi diagrams, when calculated on some set of N points in the 2d plane, segment the space into regions surrounding every point. The polygonal regions are such that, within a region surrounding some point p_0, the point p_0 is closer to any point p in that region than any other of the N points included in the Voronoi diagram. The mapping between points in the plane and surrounding regions is one to one.

This information has many uses, but one of the most obvious is processing an image for an artistic effect. The representation created by shading a Voronoi diagram on N points in the image plane with colors from each sample point creates a "stained glass" or mosaic version of the image. One of the key problems here is effective selection of the point set P for the Voronoi diagram.

2 Theory

2.1 Voronoi Diagrams

First, it is appropriate to examine the algorithms involved in the efficient calculation of a Voronoi diagram on a set of N points. The goal is a polygonal map of the plane consisting of a set of polygons surrounding the N points. The polygon surrounding a point p covers the area for which p is the closest of the N points.

Given two points p and p', we can create a Voronoi diagram by drawing a line perpendicular to the line pp', intersecting pp' at its midpoint. A Voronoi diagram with more points includes many such lines, meaning that each polygon has straight edges consisting of line segments which are sections of such perpendiculars.

Voronoi diagrams can be calculated directly, using for example the beach line algorithm from the work (Fortune, 1986). It is often simpler, however, to take advantage of the close relationship that exists between the structure of the Voronoi diagram, and that of the Delaunay triangulation. (Guibas et al., 1990)

2.2 Delaunay Triangulation

A triangulation of some point set P is a planar subdivision such that every polygon is a triangle (except for the unbounded face), and the vertices are points in P. A triangulation exists for every point set P, as any bounded face can be split up into triangles, and the unbounded face is simply the complement of the convex hull for P. There are, of course, many different triangulations on any one set of points P. Given two triangles bordered by a common edge, it is always possible to "flip" this edge such that it connects the remaining two points, assuming the quadrilateral in question is convex. (Berg, 2000)

In a triangulation, it is undesirable to have small (sharp) angles. There is one triangula-

Appeared in: *Proceedings of the Class of 2008 Senior Conference*, pages 59–65,
Computer Science Department, Swarthmore College

tion, called the Delaunay triangulation, which maximizes the minimum angle and therefore is the "best" triangulation. One simple way to find this triangulation is to take an arbitrary triangulation and flip all illegal edges. Here, an illegal edge is defined as an edge for which flipping will improve the triangulation: the ordered set of angles after flipping will be lexicographically greater than the set before flipping. (Berg, 2000) Of course, an edge can only be flipped in the case of a convex quadralateral. An example of such an edge flip is shown on fig. 1.

It is not necessary to calculate all the angles to determine the legality of an edge. Consider two triangles *abc* and *dbc* that share an edge *cb*. Let C be the circle defined by *abc*. The edge ij is illegal if an only if the point d lines inside C. A proof can be found in (Berg, 2000).

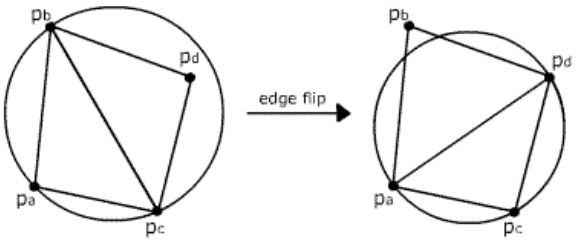

Figure 1: An edge flip during the process of creating a Delaunay triangulation.
http://www.cescg.org/CESCG-2004/web/Domiter-Vid/

A Delaunay triangulation can be constructed using an incremental algorithm based on the above. (Berg, 2000) Randomizing the point set P, add the points sequentially to the triangulation. Each time a point is added, start by triangulating the face containing the new point, and then legalize edges recursively until all edges in the triangulation are legal. Thus, the algorithm maintains a correct Delaunay triangulation of the currently included points as an invariant.

2.3 Dual Transformation

The last step in constructing a Voronoi diagram on P is to convert the Delaunay triangulation on P into a Voronoi diagram. The structures are related through duality.

A face in the Delaunay triangulation corresponds to a vertex of the Voronoi diagram, such that the location of the Voronoi vertex is the center of circumcircle for the (triangular) Delaunay face. A vertex in the Delaunay triangulation corresponds to a face in the Voronoi diagram. This Voronoi face surrounds the Delaunay vertex and represents the Voronoi cell for this vertex. An edge in the Delaunay triangulation corresponds to a perpendicular edge in the Voronoi diagram. Two Voronoi vertices are connected if an only if the corresponding Delaunay faces are adjacent. Figure 2 shows a Delaunay triangulation and the corresponding Voronoi diagram.

2.4 Point Sampling

One of the primary difficulties in using Voronoi diagrams to create stained glass effects is the selection of the point set P on which to build the diagram. Badly chosen points create a result that captures none of the features in the original image. In this project, the implementation of fully automated, intelligent point selection was a key goal. Point selection can be done, or adjusted, manually, however the need for such intervention limits to applicability of the processing, and so was not studied here.

2.4.1 Naive Approaches

The simplest method for point selection uses a random sample of N points, distributed uniformly within the boundaries of the image. Such a method is of course very simple to implement, and also has an advantage that follows from its uniformity. Because the distribution is uniform, the sizes of all the Voronoi cells will be relatively small, and thus a badly-colored Voronoi cell can have only a limited size. The obvious issue with such a method is that it fails to account for the global features of an image. Random point selection, in practice, results in significantly distorted representations, especially in high-detail regions of the image, even at large N.

A grid-based point selection approach is another simple alternative. It has the advantage of highly uniform cell size, similar to that seen in a real mosaic. Like uniform random sampling, it

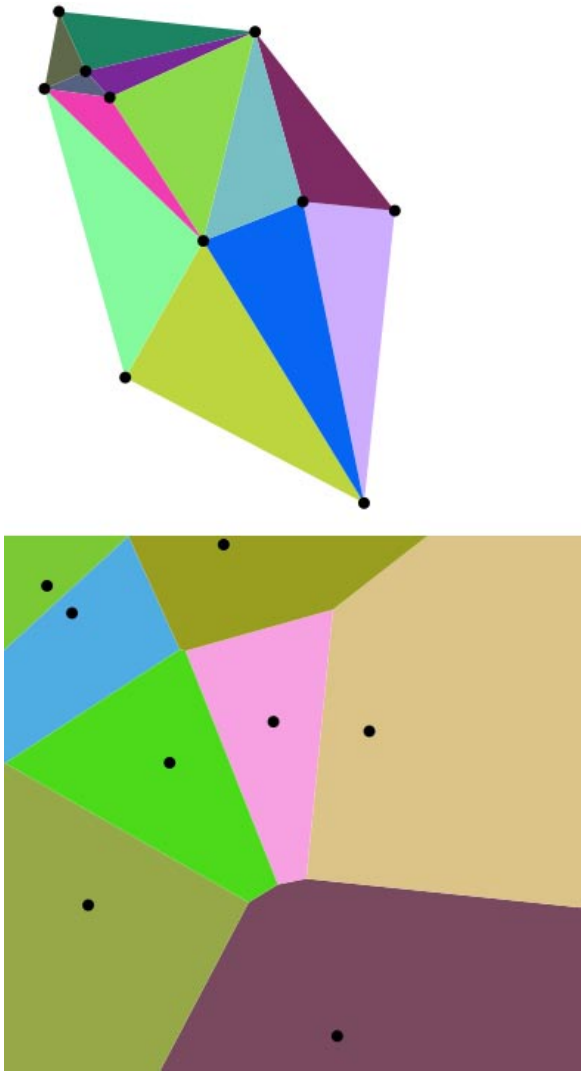

Figure 2: Example of a Delaunay triangulation (top) and corresponding Voronoi diagram (bottom). The colors have no meaning.

suffers from a failure to account for the image's important features. Good representations can only be obtained with fairly large N values.

2.4.2 Distribution Sampling

The earlier discussion of uniform sampling can be generalized to an arbitrary probability distribution on the 2d plane of the image. The question, then, is what distribution $P_s(x, y)$ on the image pixels would, when sampled from for a total of N points, produce the best representation of the image. One quantitative criterion for $P_s(x, y)$ is the error between the colored, N-point Voronoi representation with sampling from $P_s(x, y)$ and the real image. Such a value, however, does not necessarily reflect an aesthetic judgment of the colored Voronoi mosaic.

Edge detection appears as an efficient way to obtain a good $P_s(x, y)$. Good mosaics are created when Voronoi cell edges fall on edges of the image. In order for this to happen, Voronoi points must be located at equal distances from an edge line. It is undesirable for sampled points to fall on image edges themselves, as then the Voronoi cell surrounding that point is likely to be badly colored, in a way that is not expressive of key image features. The goal then, is to receive a distribution that has symmetric and significant values around edges (leading to points likely sampled there), and low values directly on edges. Here, symmetric means that values are equal at equal distances along a perpendicular to the edge.

Such a distribution can be achieved using basic edge detection and blur filters. Specifically, it is effective to use a distribution of the form $P_s(x, y) = P_{blur}(x, y) - P_{sharp}(x, y)$. Here, $P_{sharp}(x, y)$ comes from an sharp, or only very slightly blurred, black and white edge detection image. $P_{blur}(x, y)$ comes from an image processed with the same edge detection filter, followed by a significant (on the order of 5 pixels) gaussian blur. The subtracted distribution has low (dark) values immediately on the edges, and higher (lighter) values farther out from the edges. Due to the Gaussian blur, the lighter values decrease in intensity with distance from the edge. Example distributions are presented on

fig. 3. Note that the distribution along both sides of an edge is symmetric, which is good for Voronoi point selection.

2.4.3 Related Work

In addition to considering the distribution $P_s(x, y)$, it is also possible to look instead at the representation error discussed earlier. This was implemented in the work (Dobashi et al., 2002). The authors started with a simple set of points leading to a hexagonal Voronoi diagram across the image. They then adjust the locations of the Voronoi points to decrease the error (calculated as difference in colors per pixel) between the mosaic and real image representation. In the first phase, the entire set of points is moved in batch, with each point moving somewhere in its surrounding 8 pixels in such a way as to decrease the error. This process continues until changes in error are below a threshold. The second phase implements finer adjustment where each site is moved individually within its 8 pixels and the error is recalculated each time.

While this approach appears to be effective, it requires significant computational power even with approximations, and the inclusion of manual adjustments in the paper makes it difficult to judge the effectiveness of such a method for purely autoamted processing.

3 Methodology and Implementation

3.1 The Doubly-Connected Edge List

The most important component in an implementation of the above algorithms is the data structure used to represent the planar subdivision, whether it be the Delaunay triangulation or the Voronoi diagram. This data structure must support several operations in a performant way. It needs to allow fast adding of points into an existing triangulation structure, flipping of any particular edge, and traversal of a face to find its boundary edges.

The most common data structure used to meet the above requirements is called a doubly connected edge list (DCEL) (Muller and Preparata, 1977). The structure contains several types of records: faces, edges, and vertices.

Figure 3: Examples of sampling distributions created using edge detection: $P_{sharp}(x, y)$ (top), $P_{blur}(x, y)$ (middle), and $P_s(x, y)$ (bottom). Edge detection and blur were implemented using Apple Inc.'s Core Image processing filters.

Edges are represented as half edges, storing a pointer to their twin, adjacent face, and doubly linked list pointers allowing traversal of face boundaries. A face record simply contains a pointer to one half-edge along the face's outer boundary (if it exists), and a set containing one edge along every "hole" inside the face. The Delaunay triangulation was constructed in a DCEL, and then the Delaunay DCEL was transformed into its dual, representing the Voronoi diagram.

The DCEL structure described here meets all the requirements for storing a planar subdivision. However, given a point, the structure does not provide a fast way to locate the face containing a point. For this purpose, an additional DAG (Directed Acyclic Graph) data structure is layered on top of the DCEL.

3.2 DAG For Point Location

A Directed Acyclic Graph was constructed to provide fast point location during triangulation. This point location was used during the first triangulation step, where it is necessary to find the face containing the point being added. The DAG algorithm used was described in (Berg, 2000).

The leaf nodes of the DAG correspond to the current triangulation. The other nodes correspond to previous triangles that existed earlier during the incremental triangulation process. When a point p is added, causing a split of the face f, the leaf node representing f receives 3 children for the newly created triangles. The DAG is also updated on edge flips, leading to situations where a leaf node has more than 1 pointer leading to it.

Using such a DAG structure, a point can be located by starting from the root and navigating down the children, checking for containment in the process.

4 Results and Discussion

The figures below present the results of constructing mosaic representations for a test image.

The butterfly image was chosen as it is similar to example images used in (Dobashi et al., 2002). It is extremely difficult to constructed automated mosaic representations on images with many human faces.

Looking at the images on figs 4 and 5, we can see that the representation unsurprisingly improves with increasing N. It is clear that the images generated using edge-detection distribution based sampling retain significantly more of the key features, and also have a far cleaner appearance. This is due to points being sampled from a distribution symmetric around the edges, leading to Voronoi cell edges matching up with image edges. The improvement using edge-detection distribution based sampling is particularly evident with lower N values.

Significant artifacts still exist with edge-detection based sampling under lower point counts. This is because the correct distribution does not guarantee a good set of sampled points with low N, meaning that there are or empty areas in the points. One approach to remedy this would be to adjust the distribution during the sampling process, subtracting discreet, 2d Gaussians from the distribution around each point as it is sampled. This would help prevent clusters of points and empty areas, improving uniformity with low N.

Ideally, it would be effective to add equidistant points in pairs, one on each side of an edge. Doing this, however, requires knowledge of the edges as vector paths instead of as lighter pixels in an image. Such an approach would not require sampling, and would probably work significantly better than a distribution-based approach. It does, however, require a very different kind of processing.

5 Conclusion

A stained glass / mosaic filter was successfully implemented based on Voronoi diagrams. Several solutions to the problem of sampling points were compared. While the solution by (Dobashi et al., 2002) provides good results, it requires both significant processing power and manual adjustment. A simple, fully automated sampling method was proposed based on subtraction of blurred distributions obtained using edge de-

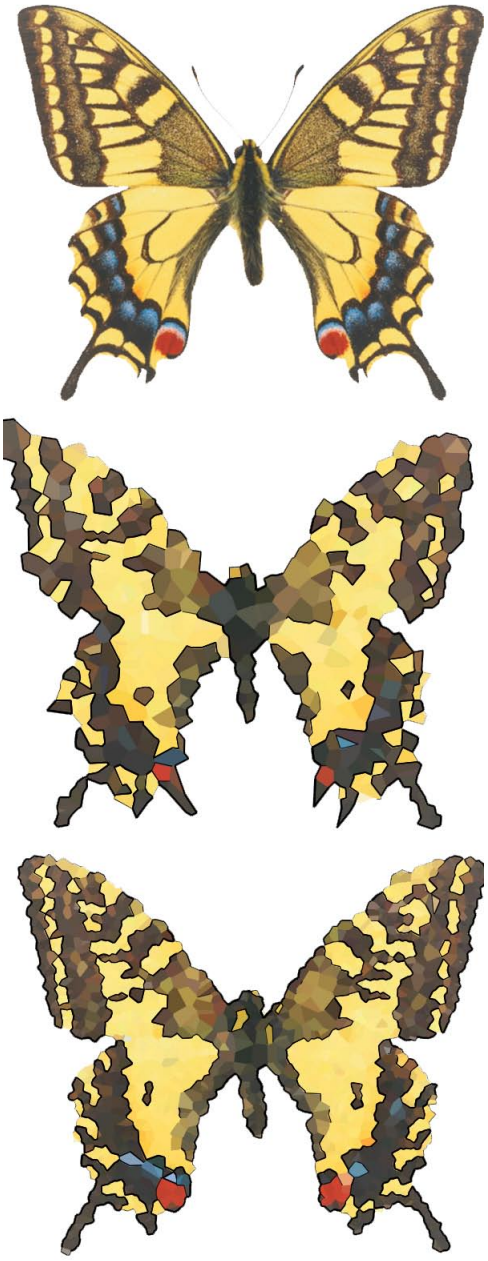

Figure 4: An example image (top), rendered using distribution-based point selection. The distribution was created by subtracting two blurred edge detection distributions, with a blur of 5.0 and a blur of 1.0. The middle image has $N = 1000$ points sampled, and the lower image has $N = 5000$.

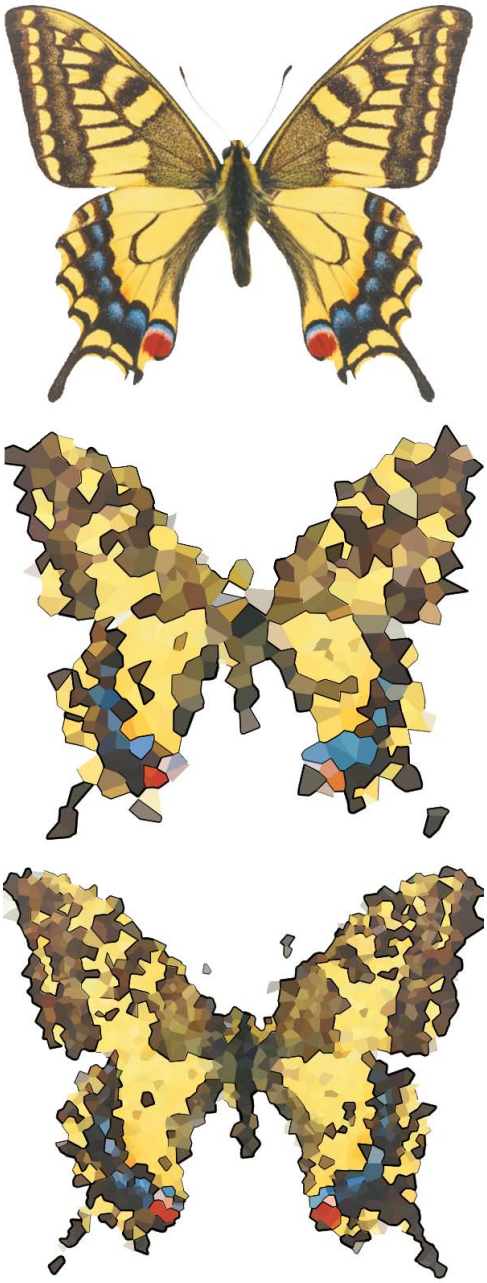

Figure 5: An example image (top), rendered using uniformly random point selection. The middle image has $N = 1000$ points sampled, and the lower image has $N = 5000$.

tection. Results presented from application of this method, shown on figs. 4 and 5, are both significantly cleaner than the mosaics created using random sampling, and more reflective of key image features.

References

Mark de Berg. 2000. *Computational geometry: algorithms and applications*. Springer, Berlin, 2nd rev. ed edition.

Y. Dobashi, T. Haga, H. Johan, and T. Nishita. 2002. A method for creating mosaic images using voronoi diagrams. In *Proc. EUROGRAPHICS 2002 Short Presentations*, pages 341–348.

S Fortune. 1986. A sweepline algorithm for voronoi diagrams. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 313–322, New York, NY, USA. ACM.

Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. 1990. Randomized incremental construction of delaunay and voronoi diagrams. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 414–431, New York, NY, USA. Springer-Verlag New York, Inc.

N. E. Muller and F. P. Preparata. 1977. Finding the intersection of two convex polyhedra. Technical Report ADA056889, U. Illinois at Urbana Champaign, Coordinated Science Lab, October.