DIMACS
Center for Discrete Mathematics &
Theoretical Computer Science

DIMACS EDUCATIONAL MODULE SERIES

MODULE 08-5

Reconstructing Curves from Sample Data: Implementing Algorithms using Voronoi Diagrams and Delaunay Triangulations

Date Prepared: August, 2008

Pallavi Jayawant
Bates College, Lewiston, ME 04240
pjayawan@bates.edu

Martha Kosa
Tennessee Tech University, Cookeville, TN 38505
mjkosa@csc.tntech.edu

Christine Shannon
Centre College, Danville, KY 40422
shannon@centre.edu

DIMACS Center, CoRE Bldg., Rutgers University, 96 Frelinghuysen Road, Piscataway, NJ 08854-8018
TEL: 732-445-5928 • FAX: 732-445-5932 • EMAIL: center@dimacs.rutgers.edu Web: http://dimacs.rutgers.edu/

Module Description Information

- **Title:**

  Reconstructing Curves from Sample Data: Implementing Algorithms using Voronoi Diagrams and Delaunay Triangulations

- **Author(s):**

  1. Pallavi Jayawant, Bates College, Lewiston, ME 04240, pjayawan@bates.edu

  2. Martha Kosa, Tennessee Tech University, Cookeville, TN 38505, mjkosa@csc.tntech.edu

  3. Christine Shannon, Centre College, Danville, KY 40422, shannon@centre.edu

- **Abstract:**

  A polygonal reconstruction of a curve *C* from a set of sample points *S* is a graph which connects every pair of sample points that are adjacent along the original curve *C* and no others. Curve reconstruction has a wide variety of applications in areas such as image processing, geographic information systems, and curve fitting. This module describes a simple algorithm for curve reconstruction and gives a method to implement it. The description of the algorithm is preceded by a discussion of the topics of Voronoi diagrams and Delaunay triangulation. The implementation includes a detailed description of the necessary data structures.

- **Informal Description:**
  In this module we describe an algorithm to reconstruct a curve from a set of sample points on the curve and then give a method to implement it. To understand the algorithm, we need to know about Voronoi diagrams and Delaunay triangulations. In Sections 2 and 3, we introduce these topics and then in Section 4 we go on to describe the algorithm. Section 5 gives resources on the World Wide Web that are relevant to the topics covered in the module. The details of the implementation of the algorithm are given in Section 6. Sections 7 and 8 describe in detail how the module can be used in a computer science or mathematics classroom.

.

- **Target Audience:**
  Sophomore/junior computer science students or mathematics students with sufficient programming background (only necessary for implementing the algorithms)

- **Prerequisites:**
  The discrete mathematics topics are all defined. If students are expected to implement the algorithms they will need sufficient experience with an object oriented language like Java to write the necessary code. However, all code is included for those who would merely like to study it or to experiment with it. Instructors may elect to supply a portion of the code and ask students to implement one or more classes.

- **Mathematical Field:**
  Discrete mathematics, Computational Geometry

- **Application Areas:**
  Computer Graphics

- **Mathematics Subject Classification:**
  MSC (2000):   05C85, 05C90, 65D18, 68P05, 68R10, 68U05

- **Contact Information:**
  Christine Shannon, Centre College, Danville, KY 40422, shannon@centre.edu

- **Other DIMACS modules related to this module:**
  None

Table of Contents:

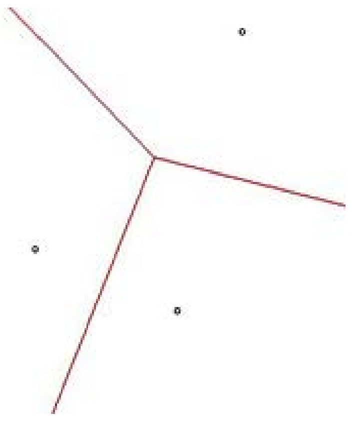
1. Overview

In this module we describe an algorithm to reconstruct a curve from a set of sample points on the curve and then give a method to implement it. To understand the algorithm, we need to know about Voronoi diagrams and Delaunay triangulations. In Sections 2 and 3, we introduce these topics and then in Section 4 we go on to describe the algorithm. Section 5 gives resources on the World Wide Web that are relevant to the topics covered in the module. The details of the implementation of the algorithm are given in Section 6. Sections 7 and 8 describe in detail how the module can be used in a computer science or mathematics classroom.

2. Voronoi Diagrams

2.1 Introduction

A town has three sites where the town authorities have put large recycling containers. Each recycling site has its service area consisting of the points in the town that are nearest to it. Thus all the points in the town are divided among the three service areas. However, some of the points in the town belong to more than one service area – these are the points that are equidistant to two or more recycling sites.



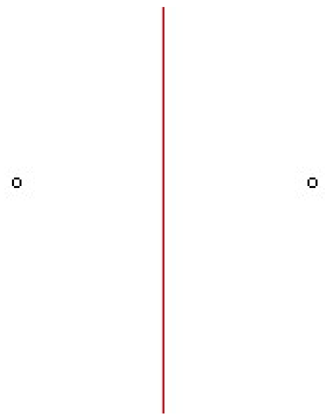
As shown in the figure, the points on the three line segments are in more than one service area. The set of these points is called the Voronoi diagram for the three sites. The service area of each site is called its Voronoi region. The point that is equidistant to all the three sites is called the Voronoi vertex.
We can use this example to define the Voronoi diagram of any set of points in the two-dimensional Euclidean plane.

2.2 Definitions

Let $P = \{p_1, p_2, ..., p_n\}$ be a set of *n* points in the plane. These points are called the sites. Each point in the plane is assigned to its nearest site. However, some points may not have a unique nearest site. The Voronoi diagram for the set of sites, *V(P)* is the set of all points in the plane that do not have a unique nearest site. For each site p_i, its Voronoi region $V(p_i)$, is the set of all points in the plane that are at least as close to p_i as to any other site, i.e.,

$$V(p_i) = \{ x : | x - p_i | \le | x - p_j | \ \forall \ j \ne i \}.$$

Using this definition let us construct the Voronoi diagram for two sites p_1 and p_2. The perpendicular bisector B_{12} of the line segment joining p_1 and p_2 is the Voronoi diagram for the two sites. The diagram is shown in the figure below. The Voronoi region of each site is the closed halfplane that contains the site and has boundary B_{12}. Later in the section we obtain a description of the Voronoi regions in terms of halfplanes.



Voronoi diagram for two sites

To see that any point *x* on the perpendicular bisector is equidistant from the two sites, draw the triangle $p_1\ p_2\ x$. Let *M* be the point where the perpendicular bisector meets the segment joining p_1 and p_2; i.e., *M* is the midpoint of the segment joining p_1 and p_2. Then the two triangles $x\ p_1 M$ and $x\ p_2 M$ are congruent by the side-angle-side test for congruence of two triangles and hence the distance from *x* to p_1 is equal to the distance from *x* to p_2.

In the case of two sites, both the Voronoi regions are unbounded. This is clear from the figure of the Voronoi diagram of two sites. But when we have more sites, some of the regions are bounded. A necessary and sufficient condition for a region to be bounded is left as an exercise. When the regions are bounded, they are convex polygons. The edges of the Voronoi regions are called Voronoi edges and the vertices are called Voronoi vertices. Thus a Voronoi vertex has at least three nearest sites.

The construction of the Voronoi diagram for three sites is left as an exercise. As the number of sites increases, it is not easy to construct the Voronoi diagram by hand. There are several algorithms to construct Voronoi diagrams. Instead of going into the description of these algorithms, at the end of this section we give you a resource on the World Wide Web that constructs the Voronoi diagrams.

Now we study another way to characterize Voronoi regions that is useful to construct Voronoi diagrams and to obtain some properties of the Voronoi regions. Let B_{ij} be the perpendicular bisector of the line segment joining sites p_i and p_j. Let $H(p_i, p_j)$ be the closed halfplane with boundary B_{ij} and containing p_i. Thus $H(p_i, p_j)$ is the set of all points that are closer to p_i than to p_j and all points that are equidistant from p_i and p_j (since we include the boundary in the halfplane). Note that B_{ij} and B_{ji} refer to the same perpendicular bisector but $H(p_i, p_j)$ and $H(p_j, p_i)$ are different closed halfplanes with the same boundary whose union is the whole plane. In the case of only two sites, as seen before $H(p_1, p_2)$ is the Voronoi region of p_1 and $H(p_2, p_1)$ is the Voronoi region of p_2. In the case of three sites, it follows from the definition of the Voronoi region that $V(p_1)$ is the intersection of the two closed halfplanes $H(p_1, p_2)$ and $H(p_1, p_3)$. Similarly $V(p_2)$ is the intersection of the two closed halfplanes $H(p_2, p_1)$ and $H(p_2, p_3)$. This description should help in the construction of the Voronoi diagram for three sites. The description of the Voronoi region in terms of halfplanes in the case of *n* sites is left as an exercise.

2.3 Properties

Each Voronoi region is convex. This property follows from the description of the regions in terms of halfplanes.

If *v* is a Voronoi vertex at the junction of three Voronoi regions corresponding to three sites, then *v* is the center of the circle *C(v)* determined by the three sites. In other words, *v* is the circumcenter of the triangle determined by the three sites. This property follows from the fact that *v* is equidistant from the three sites and is used in the crust algorithm in section 4.

A couple of more properties are mentioned in the exercises.

2.4 Exercises

1. Suppose there are *n* sites. Give a description of the Voronoi region of a site p_i in terms of the halfplanes $H(p_i, p_j)$. (Your answer should involve taking the intersection of certain halfplanes.) Use this description to show that the Voronoi region is convex.

2. Construct the Voronoi diagram for three sites that are not on one line. The diagram should look similar to the diagram in the recycling sites example.

3. Construct a Voronoi diagram in which at least one of the regions is bounded. Use as many sites as you want.

4. Constructing a Voronoi diagram by hand when there are more than just a few points can be a formidable task. Fortunately there are computer programs which will do the hard work for you. You can experiment with a Java applet which allows you to see how the Voronoi diagram changes as you add points by going to the URL http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide/index.html.en .

   For the remaining exercises, assume that no four sites lie on a circle.

5. Prove that every Voronoi vertex is at the intersection of exactly three Voronoi edges.

6. Prove that for every Voronoi vertex *v*, the circle *C(v)* contains no other sites.

7. Prove that for a site p_i, its Voronoi region is bounded if and only if p_i is not on the boundary of the convex hull of the sites. (The convex hull of the sites is the smallest convex set containing the sites.)

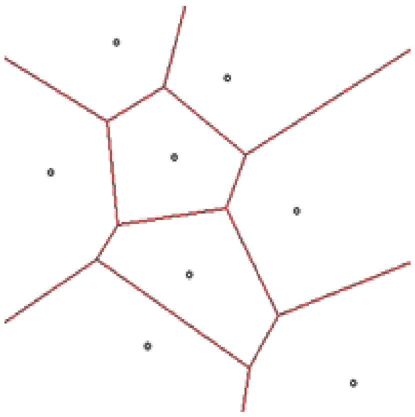
i

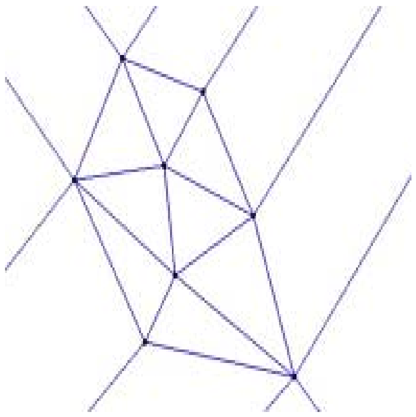
3. The Delaunay Triangulation

3.1 Definitions

Given a set $P = \{p_1, p_2, ..., p_n\}$ of points in the plane, we now know how to compute the Voronoi diagram *V*(*P*) associated with it. Each of these points p_i is located in its own region which we denote as $V(p_i)$. Recall that for each point *q* in $V(p_i)$, p_i is the closest point in *P* to *q*. Points which lie equidistant to two points p_j and p_k in P lie on the boundary of $V(p_j)$ and $V(p_k)$.

To construct the Delaunay graph for *P* (often called a *dual* graph) draw a line segment from p_j to p_k whenever $V(p_j)$ and $V(p_k)$ share an edge. Observe that each vertex of the Voronoi diagram corresponds to a face of the Delaunay graph.





Voronoi diagram **Delaunay Triangulation**

3.2 Properties

By definition, a planar graph is one that can be drawn in the plane in such a way that its edges intersect only at a common vertex of two edges. The Delaunay graph as described above is a planar graph.

In the example in the previous section, the Delaunay *graph* is actually a *triangulation* and in most cases that is exactly what happens. A triangulation *T* of a set of points *P* is a set of segments that satisfies the following conditions:

- The endpoints of the segments are in *P*.
- The segments intersect only at their endpoints.
- The segments partition the convex hull of *P* into triangles. (The convex hull of *P* is the smallest convex set containing P).

A set of points can have many triangulations. But when the Delaunay graph is a triangulation, it is a special triangulation with certain properties that we will see soon.

When is the Delaunay graph not a triangulation? If a Voronoi vertex **v** results from a meeting of k Voronoi regions where k is at least four then the face in the Delaunay graph corresponding to that vertex will actually have k edges instead of three. It turns out that this only happens if the original points lie on a circle around v. We say that a set of points is in ***general position*** if no subset of four points lies on a circle. For random points, this condition is generally satisfied. However, if it is not, additional edges will have to be added to create the triangulation and in this case the triangulation is not unique. We will assume our point set is in general position and henceforth we will refer to the Delaunay graph as the Delaunay triangulation. This also means that each Voronoi vertex in the corresponding Voronoi diagram results from a meeting of exactly three Voronoi regions.

From the construction, it is clear that each edge of the Delaunay triangulation corresponds to an edge of the Voronoi diagram.

One interesting property of the Delaunay triangulation of a set of points P is that the interior of the circumcircle of the vertices in each of the triangles of the Delaunay triangulation contains no other points of *P*. In fact, this property defines a Delaunay triangulation.

Theorem 3.2: Let *P* be a set of points and *T* be a triangulation of *P*. Then *T* is a Delaunay triangulation of *P* if and only if the circumcircle of any triangle in *T* does not contain a point of P in its interior.

We will use this property in our implementation of the Delaunay triangulation algorithm in Section 6.

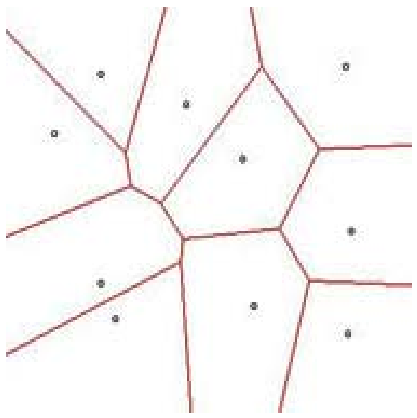
Another property of Delaunay triangulations turns out to be useful for some applications. Suppose we wish to construct a topographical map of a region and we have measured the height (above sea level) at a set of points. We can think of this as a function from a two dimensional domain into the real numbers. Now, how can we construct an approximation to the entire surface just from the values of the function on a finite set of points? One way to do this is to triangulate the domain using the points at which the elevations are known and then for each triangle in the domain we construct a triangle in three space with vertices whose x and y coordinates match those of the point in the domain and whose z coordinate is equal to the elevation at that point. Because the value at each point in the interior of the triangle will depend on the values at the vertices, it is more appropriate to have triangles which are not long and skinny. In other words we prefer a triangulation of the domain which creates triangles which have angles that are as large as possible. This is exactly what the Delaunay triangulation does.

Suppose a triangulation has k triangles and hence 3k angles. Let $\alpha_1, \alpha_2, ...\alpha_{3k}$ be the sequence of angles sorted by increasing size. Thus $\alpha_i \le \alpha_j$ if i < j. Given two such angle sequences resulting from two triangulations we say $\alpha_1, \alpha_2, ..., \alpha_{3k} \langle \beta_1, \beta_2, ..., \beta_{3k}$ if there exists an m with $1 \le m \le 3k$ and $a_i = \beta_i$ for $i < m$ and $\alpha_{\mathrm{m}} < \beta_m$. It can be shown that when the set of points is in general position, the angle sequence from the Delaunay triangulation will be greater than or equal to the angle sequence for any other triangulation.

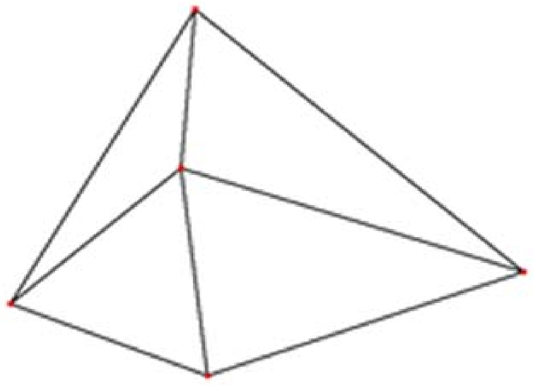
The last two properties (the property of the circumcircles of the triangles and the property of the angle sequence) make the Delaunay triangulation special among all triangulations.

3.3 Exercises

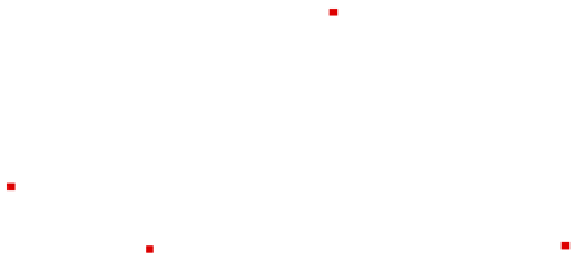
1. Draw the Delaunay triangulation for the Voronoi diagram below.



2. Verify that the following triangulation is a Delaunay triangulation. (Use the theorem in this section to verify this.) Draw the corresponding Voronoi diagram.



3. Draw two different triangulations of the following set of points. Is any of the triangulations you have drawn the Delaunay triangulation?



In order to compute the circumcenter of each triangle you have drawn and then test the condition in the theorem, take the coordinates of the above points to be (-1, 1), (0, 0), (3, 5), and (7, 0).

4. Prove theorem 3.2

5. Use the fact that the Delaunay graph is a planar graph to show that a Voronoi diagram on *n* points has at most *2n-5* Voronoi vertices and *3n-6* Voronoi edges.

4. Crust Algorithm

4.1 Curve Reconstruction Problem

We now apply the concepts we have learnt so far to the curve reconstruction problem. We motivate the need for curve reconstruction with the following situations.

A theft occurs at night in a big department store. The police look at the images recorded by the surveillance camera and obtain a blurred image of the suspect. How can they get a better image from this blurred image?

A group of archeologists finds an old handwritten document at its exploration site. It is difficult to read the document as the words can hardly be seen. Can the group make a readable document with what it has?

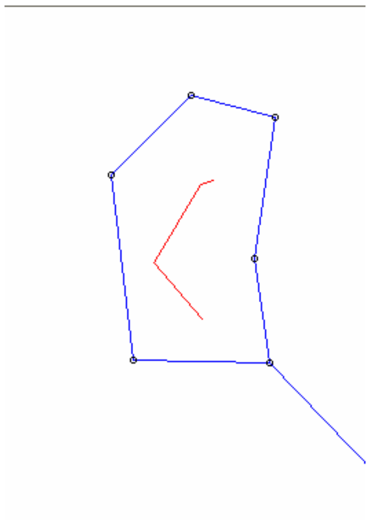
In each of the above situations, the basic problem is the same. Think about the outlines of the objects in the blurred image as curves. Each letter in the handwritten document can be thought of as a curve. Then the problem can be stated as: we have a finite set of sample points *S* from a curve *C* in the plane and we need to reconstruct the original curve from these sample points. More precisely, a **polygonal reconstruction** of a curve *C* from a set of sample points *S* is a graph which connects every pair of sample points that are adjacent along the original curve *C* and no others. Such a polygonal reconstruction gives an approximation of the curve. Obviously, if the sample points are not dense enough, then the approximation to the original curve may not be very good. Hence the choice of conditions on the density of the sample points is an important one.

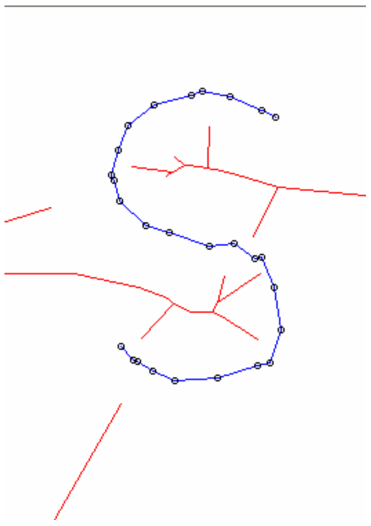
4.2 Crust Algorithm

Over the past decade, a number of algorithms have been developed that impose different sampling density conditions on the input points, but guarantee a good approximation of the original curve once the conditions are met. One of these algorithms is the crust algorithm which is a simple and beautiful application of the Delaunay triangulation. The algorithm not only gives a reconstruction of the original curve, but it also produces a reconstruction of the medial axis of the curve. The medial axis of a curve is the closure of the set of points in the plane that have at least two closest points on the curve and is useful in certain applications.

The crust algorithm guarantees a correct reconstruction only for smooth, closed curves and for certain sampling density. The following illustrations of the crust algorithm were made using the applet at http://www.cs.unc.edu/~snoeyink/demos/crust/home.html

At the left in the figures below, we see the problem that can occur when a reconstruction is attempted from an inadequate set of points sampled from the letter "S". On the right we see much better results for a larger set of points. In addition to the crust, the figure also displays the medial axis.

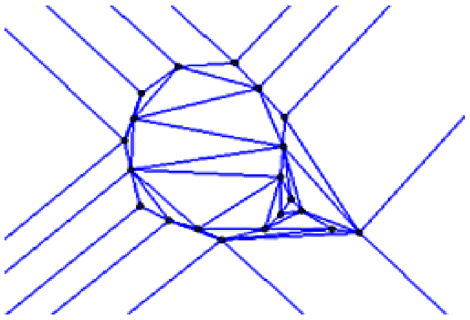




The local feature size at a point on the curve is the distance of the point from the closest point on the medial axis. The sampling density is defined in terms of the local feature size and a parameter ε. A set of sample points *S* is an ε - sample from the curve *C* if each point *c* on the curve has a sample point within distance $\varepsilon \times$ the local feature size at *c*. The crust algorithm works for smooth, closed curves that are sampled with $\varepsilon \leq 0.252$. This means each point *c* on the curve must have a sample point within distance $0.252 \times$ the local feature size at *c*. Thus if there are not enough sample points to satisfy this condition, then the algorithm may not give a correct reconstruction.

We describe the steps of the algorithm here, but for more details on why and when it works see [1].

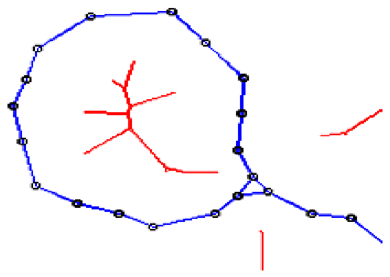
- **Step 1**. Find the Delaunay triangulation of the sample points in *S*. The figure below shows the set of sample points with the Delaunay triangulation.



- **Step 2.** Find the circumcenters of all the triangles in the Delaunay triangulation. As we have seen in the properties in section 2.3, these circumcenters are the Voronoi vertices of the Voronoi diagram for the points in *S*. Let *V* be the set of these Voronoi vertices.

- **Step 3**. Find the Delaunay triangulation of the points in the set $S \cup V$. There are three kinds of edges in this Delaunay triangulation:

a. Edges with both end-points in the set of sample points *S*. The **crust** of *S* is the graph consisting of these edges and their endpoints. The crust is a polygonal reconstruction of the curve.
b. Edges with one end-point in the set *S* and the other end-point in the set *V*.
c. Edges with both end-points in the set of Voronoi vertices *V*. The anti-crust of *S* is the graph consisting of these edges and their endpoints. The anti-crust is a polygonal reconstruction of the medial axis of the curve.



The figure shows the crust in blue and the anti-crust in red for the same set of sample points illustrated in Step 1. The Voronoi vertices are not shown on the anti-crust.

4.3 Other Algorithms

A couple of other algorithms that employ different methods and impose different sampling density conditions are [6], the nearest neighbor algorithm in [3] and the conservative crust algorithm in [4]. However, all these algorithms work only for smooth, closed curves. None of these algorithms, including the crust algorithm, can handle curves with multiple components, sharp corners or boundary points. When presented with such curves, these algorithms miss some of the required edges or put in extra edges. A recent algorithm that can handle curves with sharp corners and multiple components is the gathan algorithm in [5]. This algorithm can also detect boundary points in many cases.

Curve reconstruction algorithms provide insights for designing algorithms in three dimensions that deal with surface reconstruction.

However, instead of going into the description of any of these algorithms, we will take a brief survey of some interesting resources on the World Wide Web and then shift our focus to implementing the Delaunay triangulation and the crust algorithm with the use of appropriate data structures.

5. Web Resources

The World Wide Web has many applets and images which can motivate and illustrate the ideas we have presented. Here is a list of some of them.

- A Delaunay triangulation of points on a human fist http://www.cs.berkeley.edu/~jrs/mesh/
- A human face on a Voronoi diagram http://www.ics.uci.edu/~eppstein/vorpic.html
- Canadian maple leaf from http://www.cs.unc.edu/~snoeyink/demos/crust/Crust.pdf

- Voronoi diagrams for people standing in a gallery http://www.snibbe.com/scott/bf/index.htm

- A very nice applet which lets you enter points and then allows you to see the Voronoi Diagram, the Delaunay Triangulation or the Crust.
  http://www.cs.unc.edu/~snoeyink/demos/crust/home.html

6. Implementing the Delaunay Triangulation and the Crust Algorithm

We describe next how to perform the Delaunay triangulation algorithm, first by describing how triangulations can be stored and then by providing a high level view of the algorithm. Though the Delaunay triangulation was defined in section 3.1 using the Voronoi diagram, we do not construct the Voronoi diagram in this algorithm to obtain the Delaunay triangulation. We use the theorem in section 3.2 to construct the Delaunay triangulation.

6.1 An algorithm for the Delaunay Triangulation

We begin with a very high level version of the Boyer/Watson algorithm. Pseudocode for the algorithm is given after this example.

1. We begin with a set of points S for which we wish to create a triangulation. We draw a triangle large enough to contain all the points of S which will form the initial partial triangulation. As this incremental algorithm proceeds the points in S will be inserted into a partially completed triangulation in such a way as to maintain the properties of a Delaunay triangulation. In particular, at each step adjustments are made in the triangulation in the event that Theorem 3.2 is violated. In the end, all edges connected to the initial triangle are eliminated. The following trivial example illustrates the steps of the algorithm.

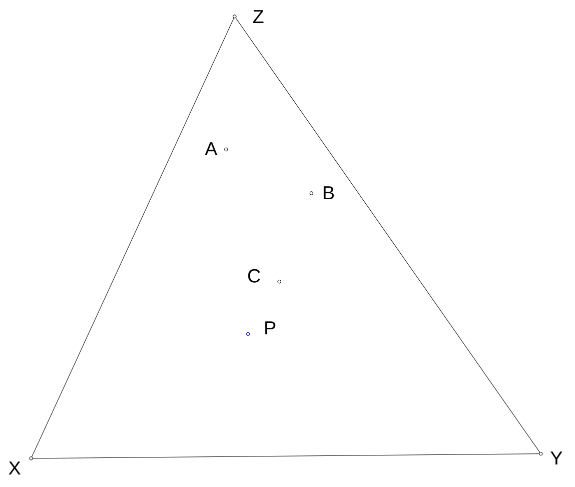


Figure 1. The set of points S = {A, B, C, P} and triangle XYZ is the large surrounding triangle which forms the initial triangulation.

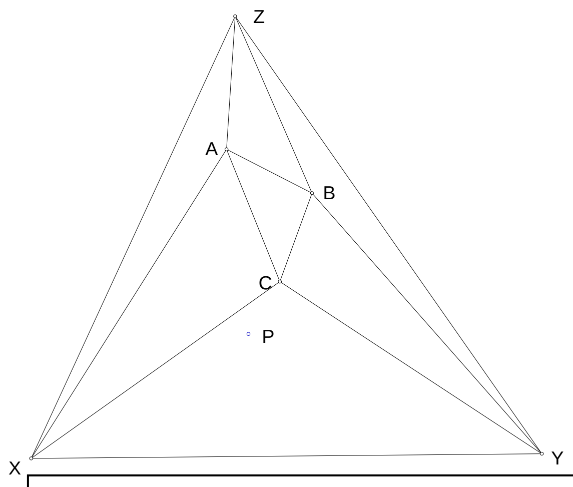


Figure 2 Assume points A, B, and C have been added resulting in the given partial triangulation. Point P will now be added.

2. Find all the existing triangles for which the new point, P would violate the circumcenter condition. That is, if this new point falls inside the circumcircle of a triangle in the triangulation, that triangle cannot be in the Delaunay triangulation for the enlarged set of points. Hence that triangle needs to be removed from the triangulation. We do this by removing edges which belong to two offending triangles.

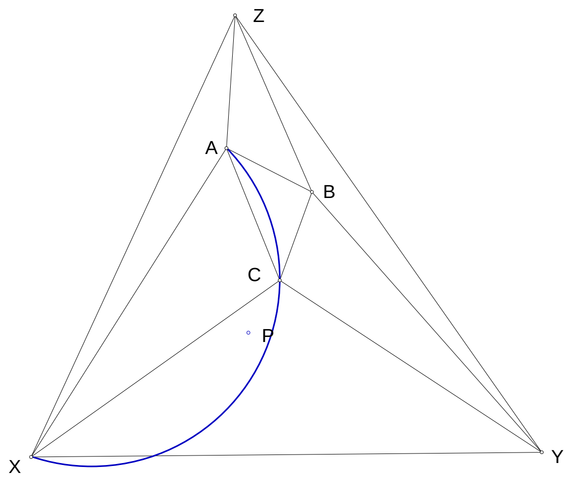


Figure 3 P falls in the circumcircle of triangles ACX and CXY.

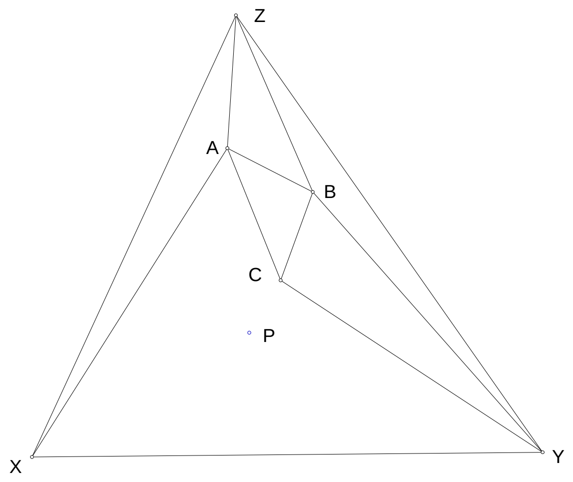


Figure 4 Remove segment XC thus removing triangles ACX and CXY.

3. Once the triangles are removed from the triangulation, P will lie in a polygonal region. To restore the triangulation, we simply add the triangles that are formed by adding an edge from P to each point on the polygonal boundary. Once all points are processed, the segments involving the large initial triangle are eliminated.

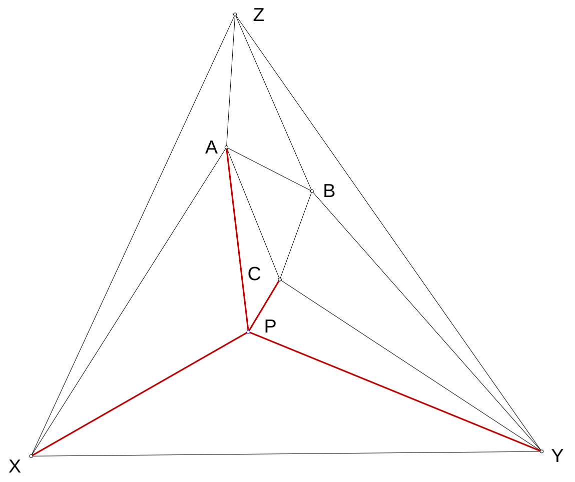


Figure 5 The new edges are added to the triangulation

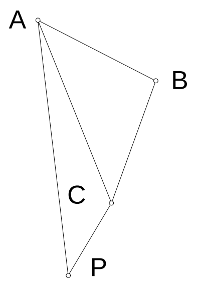


Figure 6 The desired triangulation.

The pseudocode for the algorithm is given below.

```
DT (P)

//P is a set of points in the plane.  The algorithm returns the set D of
//triangles in a Delaunay triangulation.

1. Choose a triangle T($p_{-1}$,$p_{-2}$,$p_{-3}$) containing the set of points P in its
   interior and initialize D = { T($p_{-1}$,$p_{-2}$,$p_{-3}$) }

2.  For each p in P :
      // delete triangles and add new ones so that p will be in the
      //triangulation
      Initialize a list L to hold the edges of triangles which are deleted
      For each T in D
            If p is in the interior of the circumcircle of T
                  Delete T from D.
                  Add its edges to L
      Delete the edges from L that belong to two different deleted triangles.
      The edges that remain will be the boundary of the enclosing polygon
      Add the triangles that are formed by joining p to each of the vertices
      of the enclosing polygon to D

3.  Delete any triangles from D which contain one or more of the vertices
    $p_{-1}$,$p_{-2}$,or $p_{-3}$.

4.  The remaining triangles in D form the Delaunay Triangulation.  Return D.
```

In step 1 we begin with a triangle large enough to contain all the points in their interior. Eventually these points and all the edges connected to these points will be removed leaving the triangulation of the original point set.

Here we describe how to build this giant triangle. If we have all the points at the beginning we can use linear search to compute the maximum absolute value of any coordinate of a point of the set. Call this value M. We now can specify the coordinates of the three endpoints of the large triangle: p_{-1} has coordinates (3*M, 0), p_{-2} has coordinates (0, 3*M), and p_{-3} has coordinates (-3*M, -3*M). These values are chosen so that the points do not interfere with the true triangulation. If we do not have all the points at the beginning, perhaps because the points are being entered interactively, we can replace the 3*M with something like the maximum integer in the system.

In Step 2, one will need a formula for computing the circumcenter for each triangle. Here is the formula that is needed [9]

Given three non-collinear points $\left(a_x, a_y\right), \left(b_x, b_y\right)$, and $\left(c_x, c_y\right)$ let $k = 2(a_y c_x + b_y a_x - b_y c_x - a_y b_x - c_y a_x + c_y b_x)$. Then the circumcenter (cc_x, cc_y) can be computed by the formulas

$$cc_x = (b_y a_x{}^2 - c_y a_x{}^2 - b_y{}^2 a_y + c_y{}^2 a_y + c_y b_x{}^2 + a_y{}^2 b_y + a_y c_x{}^2 - c_y{}^2 b_y - \\ b_y c_x{}^2 - b_x{}^2 a_y + c_y b_y{}^2 - a_y{}^2 c_y) / k$$

$$cc_y = (c_x a_x{}^2 + c_x a_y{}^2 + b_x{}^2 a_x - b_x{}^2 c_x + a_x b_y{}^2 - b_y{}^2 c_x - b_x a_x{}^2 - a_y{}^2 b_x - \\ a_x c_x{}^2 + c_x{}^2 b_x - a_x c_y{}^2 + c_y{}^2 b_x) / k$$

The radius of the circumcircle can then be computed by finding the distance from the center to one of the original points. It is convenient to store the circumcenter and the radius of the circumcircle with the triangle.

6.2 Implementation of the Crust Algorithm

The crust algorithm requires Voronoi vertices. How are Voronoi vertices computed? We just computed the Delaunay triangulation. Remember the duality relationship between Delaunay triangulations and Voronoi diagrams. A Voronoi region corresponds to an endpoint on a triangle in the Delaunay triangulation. A Voronoi vertex is the meeting point for three Voronoi regions. The vertex is the circumcenter of the circle through the endpoints of the corresponding triangle.

```
GetVoronoiVertices(D)
// D is the set of triangles in the Delaunay Triangulation of a set of points
//returns the set of Voronoi vertices
Initialize L as an empty list of vertices
for each t in D
      let v = the circumcenter of t
      insert v in L
return L
```

We now have the components to implement the Crust Algorithm

```
Crust(P)
// P is a set of points in general position
// returns the set of edges in the crust

Initialize an empty list E of edges
D = DT(P)
V = GetVoronoiVertices(D)
P1 = the union of P and V
D1 = DT(P1)
For each T in D1
      For each edge e in T
            if e joins two points in P, add e to E (without duplication)
return E
```

If the implementation of these algorithms is in a language which supports graphics, it would be desirable to draw the triangulation using one color to draw the crust edges (those which connect two of the original points), another color for the edges of the medial axis (which connect two Voronoi vertices), and a third, complementary color for the remaining edges.

```
Draw(D)
//D is a Delaunay triangulation of a point set P which is in general position

for each T in D
         TriangleDraw(T)
```

```
TriangleDraw(T)
//T is a triangle with three vertices, v1, v2 and v3

for each possible pair of vertices vi and vj
      if vi and vj are both Vornonoi vertices draw a line between them in
            magenta
      else if only one is a Voronoi vertex draw a line between them in green
      else draw a line between them in blue
```

6.3 Data Structures: Lists of Vertices, Edges and Triangles

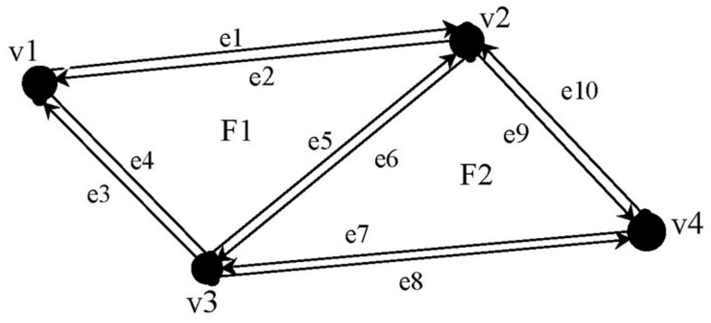
The crust algorithm needs the original vertices, computes the Delaunay triangulation for them, then computes the Voronoi vertices, and finally computes the Delaunay triangulation for all the vertices.

Recall that the Delaunay triangulation groups vertices into a planar graph with triangular faces. A graph is a set of vertices and a set of edges. In object-oriented design, the nouns in a problem description often correspond to classes or instance variables, and the verbs to methods. The nouns we consider are vertex, edge, and triangle. A two-dimensional vertex is defined by its x-coordinate and y-coordinate. For our particular application, we are also interested to know whether a vertex is a specially calculated Voronoi vertex, the circumcenter of one of the triangles in the Delaunay triangulation. An edge connects two vertices one of which is the source and the other is the destination. A triangle will need three vertices and it will be convenient to store the circumcenter and the radius of the circumcircle.

We must be able to retrieve information about every vertex, edge, and triangle in order to display the crust properly. Thus there will have to be methods for accessing the data stored in each of these objects as well as the usual constructors. We will also need to use collections to hold the data. One may envision separate collections for the vertices, edges, and triangles. Fixed-size collections are typically implemented with arrays. If the number of vertices is bounded above by n and the vertices are not all collinear, the number of triangles would be bounded above by 2n - 2 and the number of edges would be bounded above by 3n - 2 [2, p. 185]. The number of triangles produced by the crust algorithm would then be bounded above by 6n - 6 and the number of edges would be bounded above by 9n - 8 because the number of Voronoi vertices would be bounded above by 2n - 2, implying an upper bound of 3n - 2 for the total number of vertices. If the maximum number of vertices is unknown, the number of triangles and edges would also be unknown. Arrays would not be suitable for storing the vertices, edges, and triangles in this case. We need to consider a collection that can grow to an arbitrary size, a generalized list abstract data type with operations to add, remove, and retrieve items.

6.4 Representing Triangulations

Suppose we have a triangulation for a convex region in the plane.



There are a variety of ways in which to represent this triangulation in a computer program. One way is to use the notion of ***half edges*** depicted above by which this graph is made up of four **vertices,** ten **half edges,** and two **faces.**   Half edges are useful because they allow an easy way to give an orientation to a traversal of the edges of the face. Some algorithms for computing a Delaunay triangulation require the traversal of each face in a counterclockwise direction. Notice that each half edge has a corresponding half edge with the opposite orientation which is called its *twin.*

Another technique would simply use **edges** between the vertices. The example above would still have four vertices and two faces, but the pairs of half edges between two vertices would each form one edge giving five edges. To specify a direction along an edge one would also have to identify one end of the edge as the source. For example, if **e** is the undirected edge between **v1** and **v2,** specifying **v1** as the source would give **e** the same direction as the half edge **e1.**

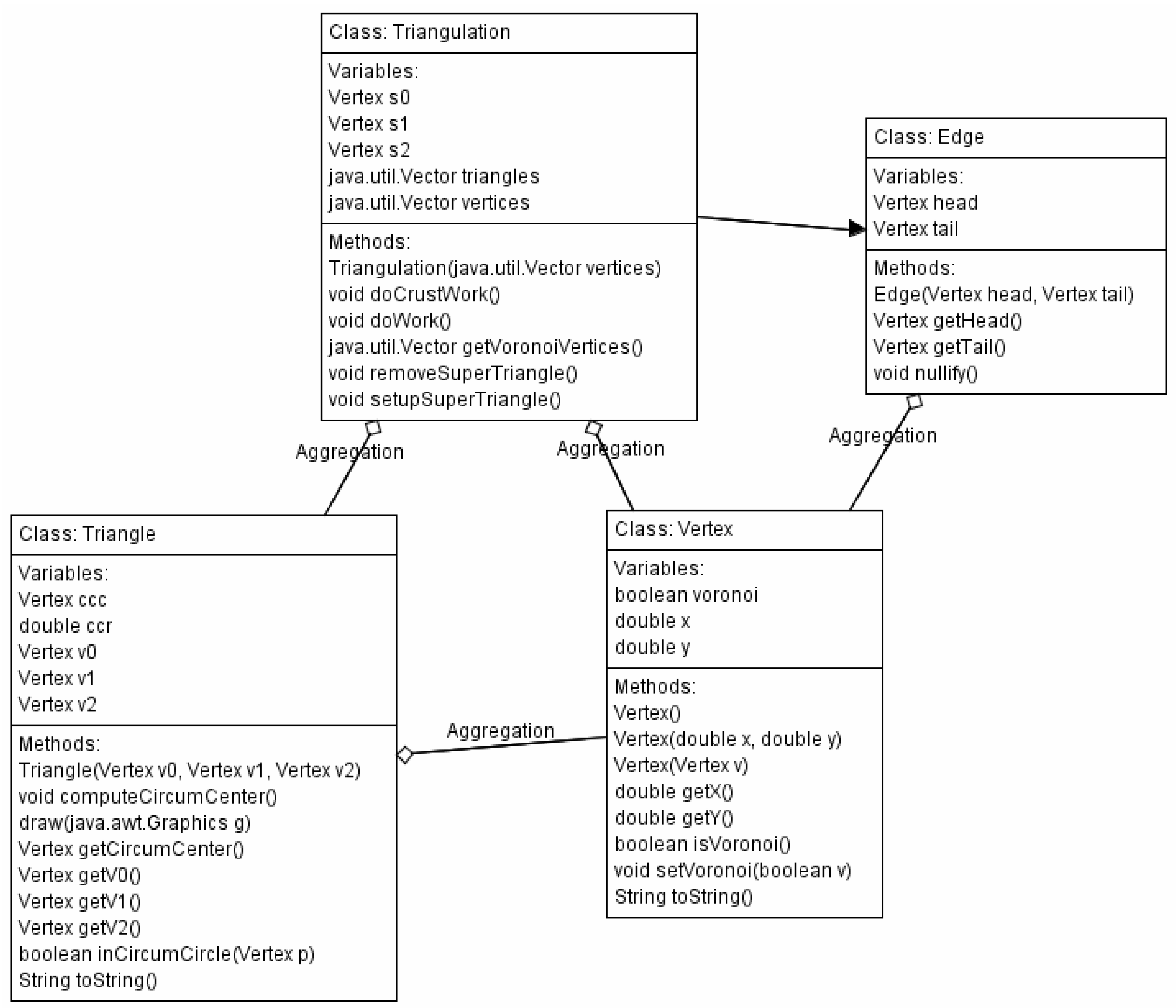
Yet a third way to store the data is the **quad-edge** data structure of Guibas and Stolfi [7]. A quad edge represents each undirected edge, by four directed edges, namely the edge itself in both directions, and its dual edge in both directions. To get the dual edge of an edge, identify the two faces for which it is a boundary and construct a point for each face. The dual edge is the one which connects these two points.

6.5 A Complete Solution in Java for the Boyer/Watson Algorithm

An electronic version of the source code is available from the authors by request.

The SimpleTriangulator class is the entry point for the application. The user can optionally provide a file name as a command-line argument to read in initial points and can use the mouse to add points. A checkbox is selected if the user wants to see the crust of the set of points. The user presses the Triangulate button to begin the computation of the Delaunay triangulation and/or crust. The results of the computation are displayed in the drawing area, with the original vertices drawn in red, the Voronoi vertices drawn in black, the crust edges drawn in blue, the medial axis edges drawn in magenta, and the remaining edges in green. The Triangulation class implements the Bowyer/Watson algorithm. The supporting classes are Vertex, Edge, and Triangle. We use the predefined Vector class for our list storage needs. We have documented all classes in the standard **javadoc** format.

The following UML diagram describes the relations among the classes.



The following is the code for the SimpleTriangulator class.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import java.io.*;

/** This class represents a GUI for displaying Delaunay triangulations and/or
* crusts for 2D vertices.  Vertices can be read in from a file upon initialization,
* and the user can choose to add vertices via mouse clicks.
* @author Martha Kosa
*/
public class SimpleTriangulator extends JFrame
{

/** drawing area for the vertices and edges */

   private TriPanel drawPanel;

/** container to hold crust checkbox and triangulation button */
   private JPanel crustPanel;

/** holds the vertices to be triangulated */
   private Vector points;

/** clicked when the user wants to see the triangulation */
   private JButton triButton;

/** clicked when the user wants to remove all vertices and clear the display */
   private JButton clearButton;

/** container holding all GUI components */
   private Container guiContainer;

/** the triangulation for the set of vertices, calculated on request only */
   private Triangulation tri;

/** selected when the user wants to compute the crust of the vertices */
   private JCheckBox crustCheck;

/** constructor assuming no vertices yet */
   public SimpleTriangulator()
   {
      super("Simple Triangulator");

      points = new Vector();
      guiContainer = getContentPane();
      drawPanel = new TriPanel();
      drawPanel.addMouseListener(new MyMouseListener());
      guiContainer.add(drawPanel, BorderLayout.CENTER);
      crustPanel = new JPanel();
      crustCheck = new JCheckBox("Crust");
      crustPanel.add(crustCheck);
      triButton = new JButton("Triangulate");
      triButton.addActionListener(new MyTriangulateListener());
      crustPanel.add(triButton);
      guiContainer.add(crustPanel, BorderLayout.NORTH);
      clearButton = new JButton("Clear");
      clearButton.addActionListener(new MyClearListener());
      guiContainer.add(clearButton, BorderLayout.SOUTH);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(500,500);
      show();
   }

/** constructor reading vertices from an input file with the given file name */
   public SimpleTriangulator(String fileName)
   {
      this();
      try
      {
         FileReader fr = new FileReader(fileName);
         BufferedReader br = new BufferedReader(fr);
         String line = br.readLine();
         while (line != null)
         {
            StringTokenizer st = new StringTokenizer(line);
            int x = Integer.parseInt(st.nextToken());
            int y = Integer.parseInt(st.nextToken());
            Vertex v = new Vertex(x, y);
            points.add(v);
            line = br.readLine();
         }
      }
      catch (IOException ioe)
      {
         System.out.println("IO problem");
      }
      drawPanel.repaint();
   }

/** inner class to perform event handling for mouse presses */
   public class MyMouseListener extends MouseAdapter
   {
```

```
/** Event handler for mouse presses.  Creates a vertex with coordinates
*  corresponding to the mouse coordinates, adds the vertex to the collection,
*  and displays the new vertex. */
      public void mousePressed(MouseEvent e)
      {
         tri = null;
         Vertex p = new Vertex(e.getX(), e.getY());
         points.add(p);
         drawPanel.repaint();
      }
   }

/** inner class to perform event handling for the triangulation button */
   public class MyTriangulateListener implements ActionListener
   {

/** Event handler for the triangulation button.  Determines if the user wants to see the
* crust for the vertices in addition to the Delaunay triangulation. */
      public void actionPerformed(ActionEvent e)
      {
         tri = new Triangulation(points);
         if (crustCheck.isSelected())
            tri.doCrustWork();
         else
            tri.doWork();
         drawPanel.repaint();
      }
   }

/** inner class to perform event handling for the clear button */
   public class MyClearListener implements ActionListener
   {

/** Event handler for the clear button.  Removes all vertices and clears any previously
*  drawn vertices and/or triangulations. */
      public void actionPerformed(ActionEvent e)
      {
         points.removeAllElements();
         tri = null;
         drawPanel.repaint();
      }
   }

/** inner class for the drawing area to display vertices and triangulations */
   public class TriPanel extends JPanel
   {

/** Displays the vertices in red and the triangulation (if it has been computed). */
      public void paint(Graphics g)
      {
         g.clearRect(0,0,getWidth(),getHeight());
         Color oldColor = g.getColor();
         g.setColor(Color.red);
         for (int i = 0; i < points.size(); i++)
         {
            Vertex curr = (Vertex) points.elementAt(i);
            int x = (int) curr.getX();
            int y = (int) curr.getY();
            g.fillOval(x,y,3,3);
         }
         g.setColor(oldColor);
         if (tri != null)
            tri.draw(g);
      }
   }


/** Entry point for the application.  If no command-line argument is specified, the application
*  begins with no vertices.  Otherwise, the application attempts to read vertices from the file
*  name specified by the first argument and displays the vertices that have been successfully
*  read. */
   public static void main(String[] args)
   {
      SimpleTriangulator st;
      if (args.length == 0)
         st = new SimpleTriangulator();
      else
         st = new SimpleTriangulator(args[0]);
   }
}
```
The following is the code for the Triangulation class.

```
import java.util.Vector;
import java.awt.*;

/**  This class allows the Delaunay triangulation and the crust for a set of vertices to be computed and
* provides the capability for the edges to be drawn.  The triangles are stored in a vector.  The
algorithm
* computes the triangles in the triangulations via the Bowyer/Watson algorithm.
* @author Martha Kosa
*/
public class Triangulation
{

/** holds the vertices to be triangulated */
   private Vector vertices;

/** holds the triangles of the current triangulation */
   private Vector triangles;

/** the first vertex in the supertriangle constructed to contain all vertices to be triangulated */
   private Vertex s0;

/** the second vertex in the supertriangle constructed to contain all vertices to be triangulated */
   private Vertex s1;

/** the third vertex in the supertriangle constructed to contain all vertices to be triangulated */
   private Vertex s2;

/** constructor to initialize the vertices for which the Delaunay
*  triangulation is to be computed.  */
   public Triangulation(Vector vertices)
   {
      this.vertices = new Vector();
      for (int i = 0; i < vertices.size(); i++)
      {
         Vertex v = (Vertex) vertices.elementAt(i);
         this.vertices.add(new Vertex(v));
      }
      triangles = new Vector();
   }

/** Computes the crust of the vertices according to the algorithm of Amenta, Bern, and Eppstein.
* First, the Delaunay triangulation of the original vertices is computed.  Then the
* Voronoi vertices are determined.  Finally, the Delaunay triangulation of all the
* vertices is computed.
*/
   public void doCrustWork()
   {
      doWork();
      Vector vv = getVoronoiVertices();
      triangles.removeAllElements();
      for (int i = 0; i < vv.size(); i++)
      {
         vertices.add(vv.elementAt(i));
      }
      doWork();
   }



/** Does the work of the incremental Bowyer/Watson Delaunay triangulation algorithm.
*  Assumes at least 3 vertices for a proper triangulation.  Then initializes
*  a triangle large enough to contain all input vertices, the "supertriangle".
*  Next, inserts the vertices one at a time.  Finally, deletes the supertriangle.
*  The vertex insertion code is a Java translation of C code by Paul Bourke.
*  @see <a href="http://astronomy.swin.edu.au/~pbourke/terrain/triangulate">Triangulate</a>
*/
   public void doWork()
   {
      if (vertices.size() < 3)
         return;
      setupSuperTriangle();

      Vector edges;

      for (int i = 0; i < vertices.size(); i++)
      {
         Vertex v = (Vertex) vertices.elementAt(i);
         System.out.println("Adding vertex: " + v);
         int j = 0;
         edges = new Vector();
         while (j < triangles.size())
         {
            Triangle t = (Triangle) triangles.elementAt(j);
            System.out.println("Checking " + j + " th triangle: " + t);
            if (t.inCircumCircle(v))
            {
               System.out.println("Vertex in triangle " + j);
               edges.add(new Edge(t.getV0(), t.getV1()));
               edges.add(new Edge(t.getV1(), t.getV2()));
               edges.add(new Edge(t.getV2(), t.getV0()));
               triangles.remove(j);
            }
            else
            {
               System.out.println("Vertex not in triangle " + j);
               j++;
            }
         }
         System.out.println("i: " + i + " " + edges.size());
         for (j = 0; j < edges.size()-1; j++)
         {
            Edge e = (Edge) edges.elementAt(j);
            if (e.getHead() != null && e.getTail() != null)
            {
               for (int k = j+1; k < edges.size(); k++)
               {
                  Edge ek = (Edge) edges.elementAt(k);
                  if (ek.getHead() != null && ek.getTail() != null)
                  {
                     if (e.getHead() == ek.getTail() &&
                         e.getTail() == ek.getHead())
                     {
                        e.nullify();
                        ek.nullify();
                     }
                  }
               }
            }
         }

         for (j = 0; j < edges.size(); j++)
         {
            Edge e = (Edge) edges.elementAt(j);
            if (e.getHead() != null && e.getTail() != null)
            {
               Triangle t = new Triangle(e.getHead(), e.getTail(), v);
               triangles.add(t);
               System.out.println("Adding new triangle: " + t);
            }
         }
         System.out.println("*** end of processing vertex " + i);
      }
      removeSuperTriangle();
   }

/** Computes three vertices far away from the given vertices and forms
* a triangle from them. */
   private void setupSuperTriangle()
   {
      /* set up supertriangle */
      s0 = new Vertex(0, -16384);
      s1 = new Vertex(-32768, 16384);
      s2 = new Vertex(32768, 16384);
      Triangle st = new Triangle(s0, s1, s2);
      triangles.add(st);
   }

/** Removes any triangles having a vertex belonging to the supertriangle. */
   private void removeSuperTriangle()
   {
      int i = 0;
      while (i < triangles.size())
      {
         Triangle t = (Triangle) triangles.elementAt(i);
         if (t.getV0() == s0 || t.getV0() == s1 || t.getV0() == s2 ||
             t.getV1() == s0 || t.getV1() == s1 || t.getV1() == s2 ||
             t.getV2() == s0 || t.getV2() == s1 || t.getV2() == s2)
         {
            triangles.remove(i);
            System.out.println("Removing triangle: " + t);
         }
         else
            i++;
      }
   }
```

```
/** Computes and returns the Voronoi vertices corresponding to the Delaunay triangulation.
*  A Voronoi vertex is the center of the circumcircle formed by the three points of a triangle. */
   private Vector getVoronoiVertices()
   {
      Vector vv = new Vector();
      for (int i = 0; i < triangles.size(); i++)
      {
         Triangle t = (Triangle) triangles.elementAt(i);
         Vertex tcc = t.getCircumCenter();
         tcc.setVoronoi(true);
         vv.add(tcc);
      }
      return vv;
   }

/** Displays the triangles of the triangulation on the associated graphics context. */
   public void draw(Graphics g)
   {
      for (int i = 0; i < triangles.size(); i++)
      {
         Triangle t = (Triangle) triangles.elementAt(i);
         t.draw(g);
      }
   }
}
```

The following is the code for the Vertex class.

```
/** This class represents a 2D vertex that may or may not be Voronoi.
* @author Martha Kosa
*/
public class Vertex
{

/** the x-coordinate for the vertex */
   private double x;

/** the y-coordinate for the vertex */
   private double y;

/** indication of whether the vertex is Voronoi */
   private boolean voronoi;

/** @param x  the x-coordinate for the vertex
*   @param y  the y-coordinate for the vertex
*/
   public Vertex(double x, double y)
   {
      this.x = x;
      this.y = y;
      voronoi = false;
   }

/** copy constructor */
   public Vertex(Vertex v)
   {
      this(v.getX(), v.getY());
   }

/** no-argument constructor */
   public Vertex()
   {
      this(0,0);
   }

/** Updates the Voronoi status of the vertex. */
   public void setVoronoi(boolean v)
   {
      voronoi = v;
   }

/** Returns the Voronoi status of the vertex. */
   public boolean isVoronoi()
   {
      return voronoi;
   }

/** Returns the x-coordinate of the vertex. */
   public double getX()
   {
      return x;
   }

/** Returns the y-coordinate of the vertex. */
   public double getY()
   {
      return y;
   }

/** Returns a string representation of the vertex. */
   public String toString()
   {
      return "(" + x + "," + y + ") voronoi: " + voronoi;
   }
}
```

The following is the code for the Edge class.

```
/** This class represents an edge to be used for 2D Delaunay triangulations.  This implementation is a
*  Java translation of Paul Bourke's C code for the Bowyer/Watson algorithm.
* @see <a href="http://astronomy.swin.edu.au/~pbourke/terrain/triangulate">Triangulate</a>
* @author Martha Kosa */



public class Edge
{
/** the origin of the edge */
   private Vertex head;


/** the destination of the edge */
   private Vertex tail;


/** constructor to create an edge with the given origin and destination */
   public Edge(Vertex head, Vertex tail)
   {
      this.head = head;
      this.tail = tail;
   }

/** Returns the origin of the edge. */
   public Vertex getHead()
   {
      return head;
   }

/** Returns the destination of the edge. */
   public Vertex getTail()
   {
      return tail;
   }

/** Removes the endpoints of the edge. */
   public void nullify()
   {
      this.head = null;
      this.tail = null;
   }
}
```

The following is the code for the Triangle class.

```
import java.awt.*;

/** This class represents a 2D triangle.
*  @author Martha Kosa
*/
public class Triangle
{

/** the first vertex of the triangle */
   private Vertex v0;

/** the second vertex of the triangle */
   private Vertex v1;

/** the third vertex of the triangle */
   private Vertex v2;
```

```
/** the center of the circumcircle formed by the vertices of the triangle */
   private Vertex ccc;

/** the radius of the circumcircle formed by the vertices of the triangle */
   private double ccr;

/** constructor to initialize a triangle from the three given vertices */
   public Triangle(Vertex v0, Vertex v1, Vertex v2)
   {
      this.v0 = v0;
      this.v1 = v1;
      this.v2 = v2;
      computeCircumCenter();
   }


/** Returns the first vertex of the triangle. */
   public Vertex getV0()
   {
      return v0;
   }

/** Returns the second vertex of the triangle. */
   public Vertex getV1()
   {
      return v1;
   }

/** Returns the third vertex of the triangle. */
   public Vertex getV2()
   {
      return v2;
   }

/** Returns a string representation of the triangle. */
   public String toString()
   {
      return "Vertices: " + v0 + "/" + v1 + "/" + v2;
   }

/** Computes the center of the circumcircle formed by the vertices of the triangle.
*  @see <a href="http://mathworld.wolfram.com/Circumcircle.html">Eric W. Weisstein. "Circumcircle."
*  From MathWorld--A Wolfram Web Resource</a> [9] */
   public void computeCircumCenter()
   {
      double ax = v0.getX();
      double ay = v0.getY();
      double bx = v1.getX();
      double by = v1.getY();
      double cx = v2.getX();
      double cy = v2.getY();

      double Dval = 2.0 * (ay*cx + by*ax - by*cx - ay*bx - cy*ax +
                     cy*bx);

      double ccx = (by*ax*ax - cy*ax*ax - by*by*ay + cy*cy*ay +
                     bx*bx*cy + ay*ay*by + cx*cx*ay - cy*cy*by -
                     cx*cx*by - bx*bx*ay + by*by*cy - ay*ay*cy)/Dval;

      double ccy = (cx*ax*ax + cx*ay*ay + bx*bx*ax - bx*bx*cx +
                     by*by*ax - by*by*cx - ax*ax*bx - ay*ay*bx -
                     cx*cx*ax + cx*cx*bx - cy*cy*ax + cy*cy*bx)/Dval;

      ccc = new Vertex(ccx, ccy);
      ccr = Math.sqrt(Math.pow(ax - ccx, 2) + Math.pow(ay - ccy,
                       2));
   }

/** Returns true if and only if the given vertex lies in the circumcircle formed
*  by the triangle's vertices. */
   public boolean inCircumCircle(Vertex p)
   {
      double ccx = ccc.getX();
      double ccy = ccc.getY();
      return Math.sqrt(Math.pow(p.getX() - ccx, 2) +
                        Math.pow(p.getY() - ccy, 2)) <= ccr;
   }

/** Returns the center vertex of the circumcircle formed by the triangle's
* vertices. */
   public Vertex getCircumCenter()
   {
      return ccc;
   }
```

```
/** Displays the edges of the triangle on the associated graphics context.
*  Voronoi endpoints are colored black.  If the edge has
*  two Voronoi endpoints, it is colored magenta.  If it
*  has one Voronoi endpoint, it is colored green.  If it
*  has no Voronoi endpoints, it is colored blue. */
   public void draw(Graphics g)
   {
      int v0x, v0y, v1x, v1y, v2x, v2y;
      v0x = (int) Math.round(v0.getX());
      v0y = (int) Math.round(v0.getY());
      v1x = (int) Math.round(v1.getX());
      v1y = (int) Math.round(v1.getY());
      v2x = (int) Math.round(v2.getX());
      v2y = (int) Math.round(v2.getY());
      Color oldColor = g.getColor();
      if (v0.isVoronoi())
      {
         g.setColor(Color.black);
         g.fillOval(v0x, v0y, 3, 3);
      }
      if (v1.isVoronoi())
      {
         g.setColor(Color.black);
         g.fillOval(v1x, v1y, 3, 3);
      }
      if (v2.isVoronoi())
      {
         g.setColor(Color.black);
         g.fillOval(v2x, v2y, 3, 3);
      }
      if (v0.isVoronoi() && v1.isVoronoi())
         g.setColor(Color.magenta);
      else if (v0.isVoronoi() || v1.isVoronoi())
         g.setColor(Color.green);
      else
         g.setColor(Color.blue);
      g.drawLine(v0x, v0y, v1x, v1y);
      if (v1.isVoronoi() && v2.isVoronoi())
         g.setColor(Color.magenta);
      else if (v1.isVoronoi() || v2.isVoronoi())
         g.setColor(Color.green);
      else
         g.setColor(Color.blue);
      g.drawLine(v1x, v1y, v2x, v2y);
      if (v2.isVoronoi() && v0.isVoronoi())
         g.setColor(Color.magenta);
      else if (v2.isVoronoi() || v0.isVoronoi())
         g.setColor(Color.green);
      else
         g.setColor(Color.blue);
      g.drawLine(v2x, v2y, v0x, v0y);
      g.setColor(oldColor);
   }
}
```

7. Usage in the Computer Science Classroom

Computer science faculty members teaching undergraduate students, especially those faculty at smaller institutions, regularly struggle to keep abreast of rapid changes in the field. They welcome ideas for new classroom activities and assignments that integrate interesting topics with the fundamentals of the field. For example, the “Nifty Assignments” panel at the annual ACM SIGCSE (Special Interest Group on Computer Science Education) conference is popular with attendees. A tutorial titled “Assignments to Use Next Week” was presented at the CCSC (Consortium for Computing Sciences in Colleges) Midwest Conference in 2004.

We believe that Voronoi diagrams, Delaunay triangulations, and associated algorithms, notably the crust algorithm, serve as rich sources of materials for the computer science classroom, and we hope that some computer science faculty members will be encouraged, via our module, to introduce their undergraduate students to the vital field of computational geometry.

In the next three subsections, we outline approaches for integrating these topics into the CS2 course. Astrachan, Wilkes, and Smith (1997 SIGCSE paper on Apprentice Learning in CS2) summarize the CS2 course with the following: "A typical Data Structures CS2 course covers a wide variety of topics: elementary algorithm analysis; data structures including dynamic structures, trees, tables, graphs, *etc.*; large programming projects; and more advanced object-oriented concepts. Integrating these topics into assignments is a challenging task." The instructor can decide whether to cover the topics in a light (in-class examples only), medium (laboratory examples), or heavy fashion (out-of-class programming assignments), depending on the available time in the course. Covering the topics in the module supports the goals of the ACM Curriculum 2001 document (http://www.acm.org/education/education/education/curric_vols/cc2001.pdf) with respect to the body of knowledge in computer science, specifically core areas DS5 (graphs and trees), PF1 (fundamental programming constructs), PF2 (algorithms and problem-solving), and PF3 (fundamental data structures), and elective area AL10 (geometric algorithms). The ACM Curriculum 2001 document also recommends that computer science curricula be mathematically rigorous. The topics in this module support this recommendation because students are exposed to applications of graphs and 2D geometric calculations. All the newest curricular recommendations can be found at http://www.acm.org/education/curricula-recommendations.

7.1 In-Class Examples

For a light coverage of the topics, a discovery-based learning approach can be used. The instructor can guide the students to discover the Bowyer/Watson algorithm through a series of questions. The instructor and students can assume the existence of the supertriangle at the beginning. The instructor can ask the students about what information needs to be maintained in the computation of the Delaunay triangulation, leading to the design of the Vertex, Edge, and Triangle classes. They can then realize the proverb "A journey of one thousand miles begins with the first step" to work in an incremental manner. The first vertex goes into the supertriangle with no problem. Any subsequent vertex that is added will be within an existing triangle or on two neighboring triangles. What happens if edges from the new vertex to the triangle vertices are added naively? Is the triangulation Delaunay? What must be done to ensure the Delaunay property? Triangles need to be deleted, and the distinct vertices of the deleted triangles need to be collected so that proper new triangles can be created. Is it necessary to store all edges in the triangulation? How can all necessary vertices, edges, and triangles be stored for efficient and flexible access? If students are familiar with parameterized data types (generics), they can see a practical application for generics with this example. What needs to be done when the last vertex is added? What additional information is needed so that the crust of the vertices can be computed? Discussing the crust algorithm demonstrates algorithm reuse because the Delaunay Triangulation algorithm is called twice. By demonstration of the application program discussed in Section 6.5 or Snoeyink's crust applet mentioned in Section 5, students can see that graphs can be used to represent pictorial information in addition to more abstract concepts such as equivalence relations. The crust algorithm solves children's "connect-the-dots" puzzles automatically. The students can see an exciting, yet accessible, application of graphs.

7.2 Laboratory Activities

Many CS2 courses have a closed laboratory component, in which the enrolled students work on small programming problems to reinforce the concepts covered in class. The closed laboratory forces the students to think about the course material and apply it, following the "learning by doing" approach.

The instructor can choose to give the students portions of the application program from Section 6.5 to complete by removing certain classes and/or methods and giving the students instructions for completing them as comments or worksheets. We list several possible laboratory activities below:

- file I/O to read vertex information from a text file, where each line of the file contains the x and y coordinates of a vertex
- implementing the Vertex, Edge, and Triangle classes
- implementing the generic linked list data structure and using it to store the vertices, edges, and triangles in separate lists
- using a suitable API for maintaining the lists of vertices, edges, and triangles
- drawing the triangulation and/or crust
- drawing circumcircles for each triangle to verify the Delaunay property visually

We have chosen Java as the implementation language for our application program because of its wide usage in CS2 courses. There has been much debate in the computer science education community about whether and when to teach GUI programming in the introductory computer science sequence. If the instructor does not cover any GUI programming in the introductory computer science sequence, the instructor could assign the students the task of implementing the file I/O for writing the triangulation and/or crust information to a file. The instructor could then supply a test program for reading and displaying triangulation files of a suitable format.

7.3 Programming Assignments

Every CS2 course needs programming assignments. We believe that implementing the Bowyer/Watson algorithm and the crust algorithm from scratch would be a non-trivial, yet accessible, assignment for CS2 students, given suitable guidance from the instructor.

8. Usage in the Mathematics Classroom

We believe that the module can be used in the mathematics classroom to introduce the topics in the first four sections. A good place for the module would be a geometry course or a graph theory course. If used in a graph theory course, the instructor can emphasize the Delaunay triangulation more than the Voronoi diagrams. If the students have enough computer science background, then the suggestions in Section 7 can be used after the introduction of the topics.

Exercises 5, 6, 7 in Section 2 and exercises 4, 5 in Section 3 are simple proof writing exercises. Exercise 7 in Section 2 needs more work than the other proofs. The students would need to know Euler's formula for planar graphs to solve exercise 5 in Section 3. Depending on the students' proof writing skills, the instructor can decide which exercises to assign.

A mathematics student with a good computer science background can work through the module as an independent study project. The advisor could decide how much theory the student should pursue before going into the implementation or the implementation could be left out. The advisor could ask the student to read the paper by Amenta, Bern and Eppstein to understand why the algorithm works.

9. References

[1] N. Amenta, M. Bern, and D. Eppstein. The crust and the β-skeleton: combinatorial curve reconstruction. *Graphical Models and Image Processing*, **60** (1998), 125-135.

[2] M. de Berg, M. van Krevald, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, 2nd ed., Springer, 2000.

[3] T. K. Dey and P. Kumar. A simple provable algorithm for curve reconstruction. *Proc. ACM-SIAM Sympos. Discr. Algorithms*, (1999), 893-894.

[4] T. K. Dey, K. Mehlhorn, and E. A. Ramos. Curve reconstruction: connecting dots with good reason. *Proc. 15th ACM Sympos. Comput. Geom.*, (1999), 197-206.

[5] T. K. Dey and R. Wenger. Reconstructing curves with sharp corners. *Proc. 16th ACM Sympos. Comput. Geom.*, (2000), 233-241.

[6] F. Preparata and M. Shamos. *Computational Geometry: An Introduction*, Springer, 1985.

[7] C. Gold and J. Snoeyink. Crust and anti-crust: a one-step boundary and skeleton extraction algorithm. *Proc. 15th ACM Sympos. Comput. Geom.*, (1999), 189-196.

[8] L. Guibas and J. Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. ACM *Transactions on Graphics*, **4** (1985), 74-123.

[9] J. O'Rourke. *Computational Geometry in C*. 2nd ed., Cambridge University Press, 1998.

[10] E. Weisstein. *World of Mathematics*. http://mathworld.wolfram.com/Circumcircle.html