Computing Voronoi Diagrams

Sam Lichtenberg

January 14, 2014

1 Project Overview

We implement Steven Fortune's $O(nlogn)$ sweepline algorithm for computing the Voronoi diagram of n points in the plane. The implementation, based off of that in deBerg et al., was written in Javascript. Visualization was emphasized over performance and robustness.

The project can be found at `www.princeton.edu/~splichte/fortune`.

2 Background

2.1 Voronoi Diagrams

Let $dist(p, q)$ be the Euclidean distance between two points in the plane p and q. Let $P = \{p_1, p_2, ..., p_n\}$ be a set of n sites in the plane. The Voronoi diagram of P is then defined as the subdivision of the plane into n cells such that a point q lies in site p_i's cell if an only if $dist(q, p_i) < dist(q, p_j)$ for each $p_j \in P$, $j \neq i$.

The above is taken from deBerg et al., page 148.

2.2 Fortune's Algorithm

The problem is to efficiently compute such a subdivision of the plane, given n points. Fortune's sweepline algorithm does so in $O(nlogn)$ time, which is asymptotically optimal. Because the emphasis of this project is on the implementation, we refer the reader to deBerg et al. or Fortune's original paper for the correctness proof.

The idea behind the algorithm is to maintain a region, represented by an x-monotone beach line of parabolas, wherein the current voronoi diagram cannot be changed by any events beyond the beach line. Given a site p_j and l_y, the y-coordinate of the sweepline, the parabola β_j for each site is represented by the

equation

$$y = \frac{1}{2(p_{j,y} - l_y}(x^2 - 2p_{j,x}x + p_{j,x}^2 + p_{j,y}^2 - l_y^2)$$

The beach line consists of at most $2n - 1$ parabolic arcs. These parabolas are not maintained explicitly by the algorithm, however. The basic method of the algorithm is to maintain the beach line, adding new sites from the sorted list of input sites one at a time, updating the sweepline, and checking for "circle events" between triples of adjacent points. A "circle event" occurs when the bisectors of sites p_i, p_j and p_j, p_k (assuming without loss of generality that p_j has the middle x-coordinate) meet at some point below the current beachline, represented by the tree and the sweepline position. This is the moment when an arc disappears from the beach line, and is also how Voronoi vertices are detected.

The data structures used by the algorithm are as follows.

The beach line is represented by a balanced binary search tree. Leaf nodes represent the sites that have been processed by the algorithm so far, and that whose Voronoi region (for this position of the sweepline) is not yet fully determined. Internal nodes in the tree store a tuple of sites p_i and p_j and represent "breakpoints", or the current intersection of the two parabolas defined by those sites. All nodes in the tree, both internal and leaf, are ordered by x-coordinate at any given position of the sweepline. This is how the breakpoints do not have to maintain the parabola; they simply maintain the tuple of sites and use the parabolic equation, along with the sweepline position, to calculate their current x-coordinate and direct the search.

Events can either be sites from the list of n points taken as input, or "circle events" that occur during the running of the algorithm. Both of these events are stored in a generic priority queue, ordered by y-coordinate of the event. A "circle event" is stored by the bottommost y-coordinate of that circle, rather than the actual Voronoi vertex it represents.

3 Implementation

3.1 Overview

The files used by the algorithm are index.html, fortune.js, avl.js, structs.js, along with a priority queue implementation from the Google Closure library. These files maintain global values for the current sweepline position, the data structures (the list of input points, the priority queue, the binary search tree), an array of timeouts (so that the timeouts can be cancelled if the user wishes to re-animate), and start and end times, used for timing. structs.js keeps some data structures useful for storing output. A DCEL object is used to store the output edges and vertices,

but it is not a proper DCEL structure in this context, merely a container for the Voronoi edges and vertices.

The implementation closely follows that of deBerg et al., page 157. Therefore, this section will deal first with the departures from the algorithm they describe, and second with specific implementation details.

Instead of a balanced binary search tree, such as an AVL or Red-Black Tree, a regular Binary Search Tree was used. This makes the algorithm, in the worst case, unoptimal (becoming $O(n^2)$ for malicious inputs), but in practice this did not seem to significantly impact the running time for inputs on the order of 100000 points.

Another change was the outputted data structure. Whereas deBerg et al. output a doubly-connected edge list, our algorithm instead outputs a list of Voronoi points and edges (contained within a bounding box). This list can be converted to a doubly-connected edge list structure in linear time, if need be–during computation, simply attach each edge to both of the sites it borders, as well as the other two edges that intersect the voronoi point it leads into (if it does lead into one). The list of half-edges and faces and their relationships can thus be constructed by traversing the edge list, changing all edges to half-edges, and updating the *next* and *prev* pointers appropriately.

The implementation does not guarantee that degeneracies will be handled correctly. Specifically, there is no extra code to ensure that, in the event that the first two sites encountered have the same y-coordinate, the algorithm will handle it properly.

The basic data types are implemented as Javascript classes: Site, Event, Node, and AVLTree. The latter was named so because it was originally intended to be an AVLTree, and contains (currently non-working) functions that can be used to implement rebalancing. The Site object is just a wrapper class for a point, and has x and y parameters. The Event class is for use in the priority queue, and has a type parameter to indicate whether it is a Circle or Site event, a Site object parameter (which has the y-value that will be used in adding to the priority queue), a center parameter (for Circle events–the Voronoi vertex, for easy access), and a "deleted" parameter: if this parameter is set to true, the algorithm will not consider this event. This form of lazy deletion was implemented in order to handle Circle events that may disappear as more sites are considered.

The AVLTree and Node classes contain a variety of functions that are generally useful in Voronoi computation. The Node class contains some functions and parameters that were to be used for the AVLTree but are not currently used in this implementation: height, left rotate, right rotate. It has a value parameter, which is either of class Internal or Leaf. It has pointers to its parents and its children, as well as a pointer to its corresponding circle event, so circle events can be deleted

in constant time in HandleCircleEvent. It contains a method for calculating its current x value, given a sweepline position; an isLeaf function; a Handle HE function, which handles the half-edges at the end of processing; an Inside Box function, used in animation; a printTree function useful for debugging the tree; a Fill List function, used in the animation function drawtree.

The AVLTree class contains a root parameter, insert and remove methods (typical tree operations), and two sets of neighbor methods; the get left neighbor and get right neighbor methods find a query x value given a y-value (sweepline position), while the gln node and grn node methods find the neighbors given a query node already in the tree. The former set is useful for finding where to insert, while the latter set is useful for finding a leaf's neighbors in HandleCircleEvent and elsewhere. It contains a verticallyAbove, which finds the arc vertically above a query value; helper functions go left and go right (which just traverse the tree), update lr (left-right) and rl (which update the internal node values), parabolic (which implements the parabolic formula used in Fortune's algorithm); a drawtree function, which handles animation; replace, which replaces a leaf node with a subtree (as occurs when a new site event is added); get middle, which finds the internal node between two nodes; checkTriples and findCircleEvent, which perform the operations described in de Berg et al., and converging, which tests if two breakpoints will move closer to or pass their intersection point with future movement of the sweepline. This is how findCircleEvent is implemented. It does this by computing the beachline values (the internal nodes' x-coordinates) at both the current sweepline position l_y and $l'_y = l_y + \epsilon$, and testing if the x-coordinates of the internal nodes have gotten closer to the intersection point or have passed the intersection point entirely. If so, it returns "true"; if "not" false.

Then, there is the generic Google Closure library Priority Queue, which handles the events. The priority queue implements lazy deletion, so the algorithm will check if the item has been deleted, and continue to dequeue the priority queue until a non-deleted Event (if there is one) has been found. The deletions take total time equal to the deletion in de Berg et al., where the false alarms are deleted as they are found, over the course of the total runtime; it is an amortized cost.

In the actual running of the algorithm, the file fortune.js contains two methods HandleCircleEvent and HandleSiteEvent, implemented almost exactly as in deBerg et al. The function VoronoiDiagram repeatedly calls a function step (implemented for later ease of visualization), which dequeues the priority queue, updates the sweepline and the bounding box, and calls the appropriate method of the above two. It repeats this until the priority queue is empty, at which point it transitions to functions that handle the half-edges still remaining in the graph, setting their sites to be on the bounding box and adding them to the list of Voronoi edges.

3.2 Input and Output

The algorithm as implemented runs in a web browser and takes as input either a file of points with format ("x y newline"), with no trailing newline, or randomly generates a specified number of points. It produces as output in the web browser a list of voronoi vertices and voronoi edges, as well as a visual of the resulting subdivision of the plane, if visualization was specified. The output list of edges actually splits the edge between two sites into two edges, which share a vertex. This was designed to aid animation, and a list of objects, each of which contains two matching edges, can be created, to merge all these edges at the end of processing. This takes time linear in the number of Voronoi edges.

3.3 Visualization

A browser running this algorithm requires HTML5 canvas; a modern version of Firefox, Chrome, Safari, or IE should work fine, although only Firefox and Chrome have been tested.

The algorithm can produce either an animation of the sweepline as it is computed, a diagram of the Voronoi subdivision after it is computed, or neither. Some specific details are as follows.

We mention that the algorithm computes a bounding box so that there are no half-infinite edges. To make the graphic fill the entire HTML5 canvas box, the bounding box has a minimum size of $(800, 500)$, which is the size of the canvas element.

If animation is not specified, the function VoronoiDiagram calls a function "step" that dequeues the priority queue. If animation is specified, the algorithm first clears any timeouts or intervals that have been set by the previous running of the algorithm. Then, it calls the function $step_anim$, which calls the step function, then the drawer function, and then sets a number of timeouts between the current event and the next event (found by peeking at the head of the priority queue). It computes the number of steps of a certain length between each points, to help preserve the smoothness of the animation.

The drawer function fills in the Voronoi vertices and edges already found (and ended, in the case of the edges–i.e. hit a Voronoi vertex) and draws the sweepline; it then calls a drawer function AVLTree itself, which draws the parabolas based on the x-values of the internal nodes and leaf nodes at the sweepline position.

When the algorithm is finished running, the algorithm then sets an interval to continue the animation outside of the bounding box. There is a minor bug where the final (half-infinite) edges are drawn to the canvas before the algorithm has finished animating them; this is a simple fix, by delaying the addition of these edges to the list of edges until after the algorithm has finished animating.

4 Efficiency

To test efficency and verify that the algorithm implemented has $O(nlogn)$ running time, we timed the runtime of the algorithm as follows. The algorithm calls the Javascript time function $Date.now()$ right before VoronoiDiagram is called; then, it stores $Date.now()$ again when the priority queue is empty; this means that all site events and circle events have been processed. We repeated the timing five times for each input size, producing the following averages:

Table 1: Running times (ms) for n points

n	Time
100	22.0
500	65.8
1000	42.8
2000	156.6
5000	458.6
10000	866.4
20000	1967.2
50000	7559.0
100000	12882.4
200000	22081.0

Using the NumPy library, other points were interpolated and the following plot was produced (figure 3). We can see that the running time follows roughly a linearithmic order of growth, and is certainly not exponential. Running a curve-fit on this data, we get that the running time with form $T = a * x^b$ has constants $a \approx 0.000187, b \approx 1.5800$. So this is indeed roughly linearithmic growth.

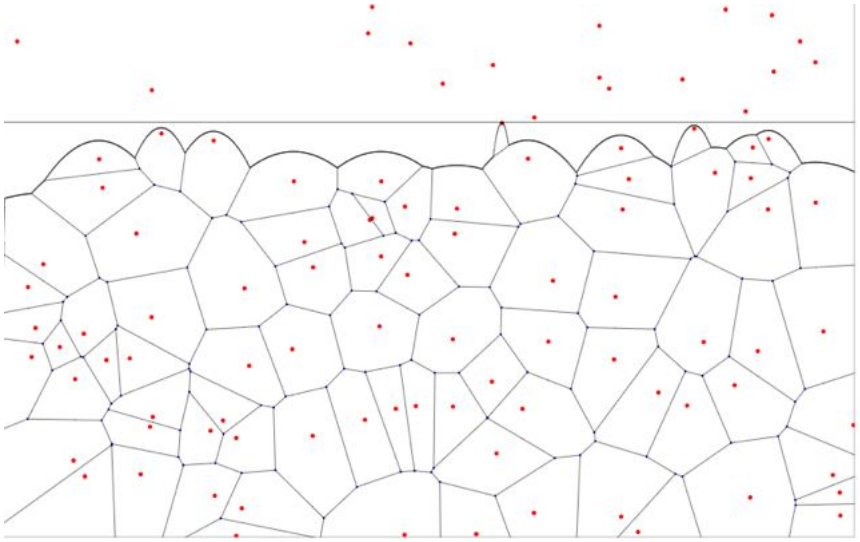

Figure 1: Voronoi Diagram animation, n=100

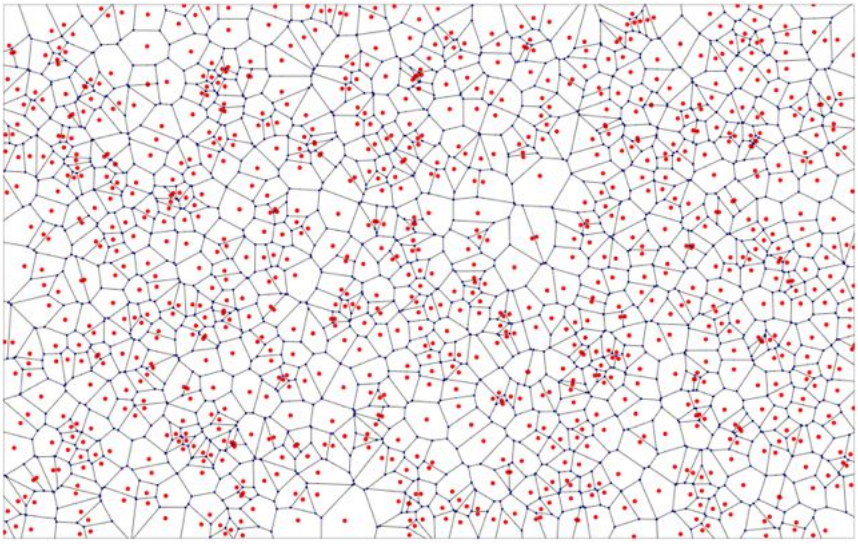

Figure 2: Voronoi Diagram, n=1000

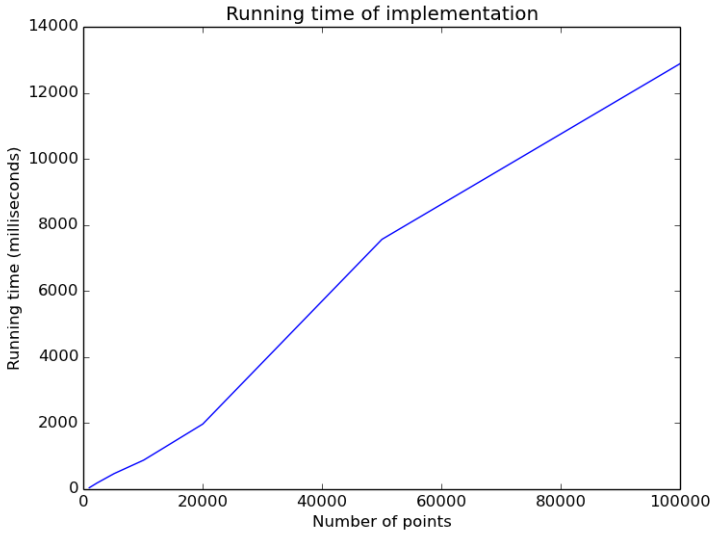

Figure 3: Input size vs. time (ms)