CMSC 754: Lecture 10
Voronoi Diagrams and Fortune's Algorithm

Reading: Chapter 7 in the 4M's. A nice applet illustrating the execution of this algorithm can be found at http://www.raymondhill.net/voronoi/rhill-voronoi.html.

Metric Spaces: In this lecture, we make a significant transition in computational geometry by introducing the notion of distance. Distances in geometry are modeled formally by through the concept of a *metric*. This is a function f on pairs of points that satisfies the following properties for any points p, q, and r:

Symmetry: $f(p,q) = f(q,p)$

Positivity: $f(p,q) \geq 0$, and further $f(p,q) = 0$ if and only if $p = q$

Triangle inequality: $f(p,q) \leq f(p,r) + f(r,q)$.

There are many natural metrics over $\mathbb{R}^d$, but the most common by far is *Euclidean distance*, which for any $p, q \in \mathbb{R}^d$ is defined as

$$\|p - q\| \;=\; \left(\sum_{i=1}^{d} (p_i - q_i)^2\right)^{1/2}.$$

Given any set of points P in $\mathbb{R}^d$, there are a number of natural structures whose definition involves the distances between points. Today, we will study one of the most important of these, called the Voronoi diagram.

Voronoi Diagrams: Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points in $\mathbb{R}^d$, which we call *sites*. We want to subdivide $\mathbb{R}^d$ into regions according to which site is most "influential". To make this formal, for any site p_i, define its *Voronoi cell*, denoted $\mathcal{V}_P(p_i)$, to be the set of points $q \in \mathbb{R}^d$ that are closer to p_i than to any other site, that is,

$$\mathcal{V}_P(p_i) \;=\; \{q \in \mathbb{R}^d : \|p_i - q\| < \|p_j - q\|, \forall j \neq i\},$$

When P is clear from context, we will omit it and refer to this simply as $\mathcal{V}(p_i)$. Clearly, the Voronoi cells of two distinct sites of P are disjoint. The union of the closures of the Voronoi cells defines a cell complex, which is called the *Voronoi diagram* of P, and is denoted $\mathrm{Vor}(P)$ (see Fig. 1(a)).

The cells of the Voronoi diagram are (possibly unbounded) convex polyhedra. To see this, observe that the set of points that are strictly closer to one site p_i than to another site p_j is equal to the *open halfspace* whose bounding hyperplane is the perpendicular bisector between p_i and p_j. Denote this halfspace $h(p_i, p_j)$. A point q lies in $\mathcal{V}(p_i)$ if and only if q lies in the intersection of $h(p_i, p_j)$ for all $j \neq i$. In other words,

$$\mathcal{V}(p_i) = \bigcap_{j \neq i} h(p_i, p_j)$$

(see Fig. 1(b)). Since the intersection of convex objects is convex, $\mathcal{V}(p_i)$ is a (possibly unbounded) convex polyhedron.

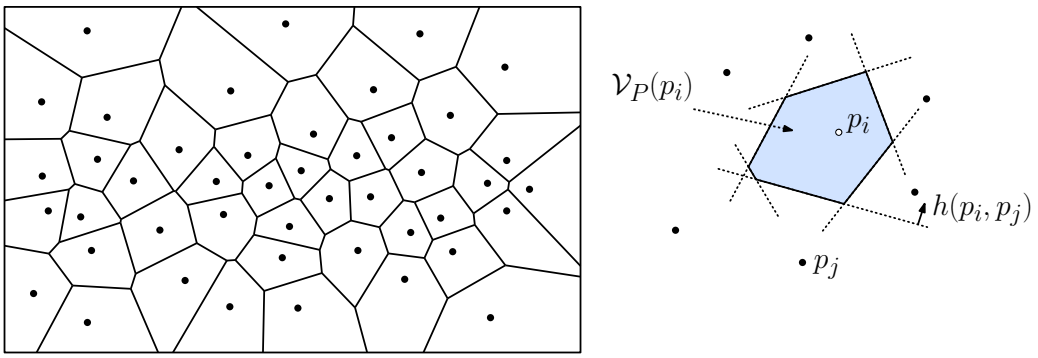


Fig. 1: Voronoi diagram $\mathrm{Vor}(P)$ of a set of sites.

Voronoi diagrams have a huge number of important applications in science and engineering. These include answering nearest neighbor queries, computational morphology and shape analysis, clustering and data mining, facility location, multi-dimensional interpolation.

Nearest neighbor queries: Given a set P of point sites, we wish to preprocess P so that, given a query point q, it is possible to quickly determine the closest site of P to q. We can do this by first computing a Voronoi diagram and then locating the cell that contains q. (In the plane, this can be done by building the trapezoidal map of the edges of the Voronoi diagram. Each trapezoid lies within a single Voronoi cell, and can be labeled with the generating site.)

Computational morphology and shape analysis: A useful structure in shape analysis is called the medial axis. The *medial axis* of a shape (e.g., a simple polygon) is defined to be the union of the center points of all locally maximal disks that are contained within the shape (see Fig. 2). If we generalize the notion of a Voronoi diagram to allow sites that are both points and line segments, then we can extract the medial axis of a simple polygon easily from the Voronoi diagram of these generalized sites.

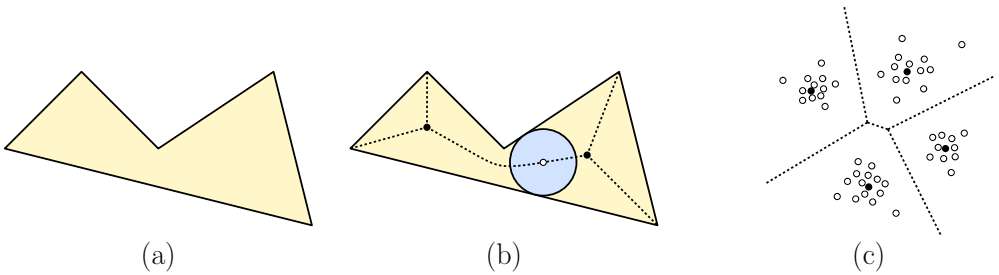


Fig. 2: (a) A simple polygon, (b) its medial axis and a sample maximal disk, and (c) center-based clustering (with cluster centers shown as black points).

Center-based Clustering: Given a set P of points, it is often desirable to represent the union of a significantly smaller set of clusters. In center-based clustering, the clusters

are defined by a set C of *cluster centers* (which may or may not be required to be chosen from P). The *cluster* associated with a given center point $q \in C$ is just the subset of points of P that are closer to q than any other center, that is, the subset of P that lies within q's Voronoi cell (see Fig. 2(c)). (How the center points are selected is another question.)

Neighbors and Interpolation: Given a set of measured height values over some geometric terrain. Each point has (x, y) coordinates and a height value. We want to interpolate the height at a query point that is not one of our measured points. To do so, we would like to interpolate its value from neighboring measured points. One way to do this, called *natural neighbor interpolation*, is to compute the Voronoi neighbors of the query point, assuming it has one of the original measured points.

Properties of the Voronoi diagram: Here are some properties of the Voronoi diagrams in the plane. These all have natural generalizations to higher dimensions.

Empty circle properties: Each point on an edge of the Voronoi diagram is equidistant from its two nearest neighbors p_i and p_j. Thus, there is a circle centered at any such point where p_i and p_j lie on this circle, and no other site is interior to the circle (see Fig. 3(a)).

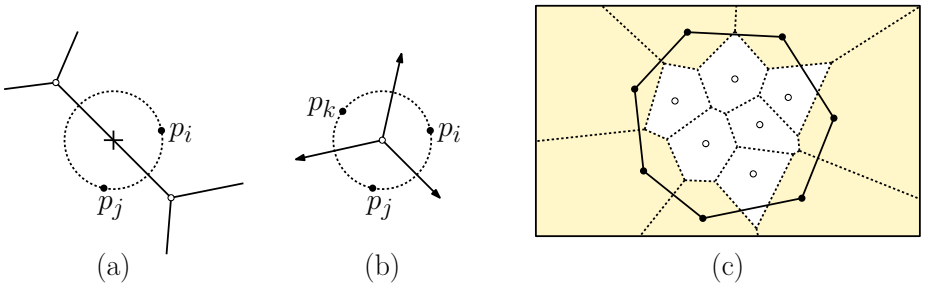


Fig. 3: Properties of the Voronoi diagram.

Voronoi vertices: It follows that the vertex at which three Voronoi cells $\mathcal{V}(p_i)$, $\mathcal{V}(p_j)$, and $\mathcal{V}(p_k)$ intersect, called a *Voronoi vertex* is equidistant from all sites (see Fig. 3(b)). Thus it is the center of the circle passing through these sites, and this circle contains no other sites in its interior. (In $\mathbb{R}^d$, the vertex is defined by $d + 1$ sites and the hypersphere centered at the vertex passing through these sites is empty.)

Degree: Generally, three points in the plane define a unique circle (generally, $d+1$ points in $\mathbb{R}^d$). If we make the general position assumption that no four sites are cocircular, then each vertex of the Voronoi diagram is incident to three edges (generally, $d + 1$ facets).

Convex hull: A cell of the Voronoi diagram is unbounded if and only if the corresponding site lies on the convex hull (see Fig. 3(c)). (Observe that a site is on the convex hull if and only if it is the closest site from some point at infinity, namely the point infinitely far along a vector orthogonal to the supporting line through this vertex.) Thus, given a Voronoi diagram, we can extract the vertices of the convex hull in linear time.

Size: Letting n denote the number of sites, the Voronoi diagram has exactly n faces. It follows from *Euler's formula*[1] that the number of Voronoi vertices is roughly $2n$ and the number of edges is roughly $3n$. (See the text for details. In higher dimensions the diagram's combinatorial complexity ranges from $O(n)$ up to $O(n^{\lceil d/2 \rceil})$.)

Computing Voronoi Diagrams: There are several algorithms for computing the Voronoi diagram of a set of n sites in the plane. Of course, there is a naive $O(n^2 \log n)$ time algorithm, which operates by computing $\mathcal{V}(p_i)$ by intersecting the $n-1$ bisector halfplanes $h(p_i, p_j)$, for $j \neq i$. However, much more efficient methods run in $O(n \log n)$ time. Since the convex hull can be extracted from the Voronoi diagram in $O(n)$ time, this is asymptotically optimal in the worst case.

Historically, $O(n^2)$ algorithms for computing Voronoi diagrams were known for many years (based on incremental constructions). When computational geometry emerged, researchers discovered a more complex but asymptotically superior $O(n \log n)$ algorithm. This algorithm was based on divide-and-conquer. But it was rather complex and somewhat difficult to understand. Later, Steven Fortune discovered a plane-sweep algorithm for the problem, which provided a simpler $O(n \log n)$ solution. We will discuss his algorithm. Later still, it was discovered that the incremental algorithm is quite efficient when run as a randomized incremental algorithm. We will discuss a variant of this algorithm later when we talk about the dual structure, called the Delaunay triangulation.

Sweep-Line Approach: Before discussing Fortune's algorithm, it is worth considering why this algorithm was not invented much earlier. In fact, it is much trickier than any plane sweep algorithm we have seen so far. The key to any plane sweep algorithm is the ability to discover all upcoming events in an efficient manner. (For example, in the line segment intersection algorithm, we considered all pairs of line segments adjacent in the sweep-line status and inserted their intersection point into the queue of upcoming events.) The challenge posed by Voronoi diagrams is predicting when and where upcoming events will occur.

To better understand the difficulty, suppose that you are designing a plane sweep algorithm. Behind the sweep line, you have constructed the Voronoi diagram based on the sites encountered so far. Now a site that lies *beyond* the sweep line (that is, in the "future") may generate a Voronoi vertex that lies *behind* the sweep line (in the past). How can any reasonable sweep-line algorithm know of the existence events generated by sites it has yet to encounter? It is these *unanticipated events* that make the design of a plane sweep algorithm challenging (see Fig. 4).

The Beach Line: To handle unanticipated events, our sweeping process maintains two objects. First, a horizontal sweep line moves from top to bottom. We will also maintain an x-monotonic curve called a *beach line*. (It is so named because it looks like waves rolling up on a beach.) The beach line lags behind the sweep line so that it is unaffected by sites beyond the current sweep line. Thus, no unanticipated events can occur. The sweep-line status depends on how the Voronoi edges intersect the beach line, not on the sweep line itself.

[1] Euler's formula for planar graphs states that a planar graph with v vertices, e edges, and f faces satisfies $v - e + f = 2$. There are n faces, and since each vertex (including a vertex at infinity that joins all the unbounded edges) is of degree at least three, we have $3v \leq 2e$, from which we infer that $v - (3/2)v + n \geq 2$, implying that $v \leq 2n - 4$. A similar argument bounds the number of edges.

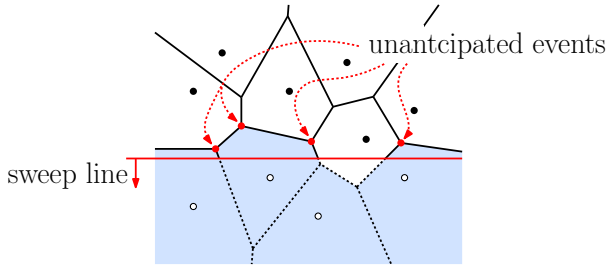


Fig. 4: Unanticipated events generated by sites that lie beyond the current sweep line.

Let's make this intuition more concrete. Given a point $q \in \mathbb{R}^2$ and a nonempty set R of points (which may be finite or infinite) define $\text{dist}(q, R)$ to be the closest Euclidean distance from q to any point in R, that is, $\text{dist}(q, R) = \min_{p \in R} \|q - p\|$. Given the current location of the sweep ℓ (which we assume to be horizontal and sweeping downwards) let $P^+(\ell) \subseteq P$ denote the subset of sites of P that lie strictly above ℓ. Assume $P^+(\ell)$ contains at least one site. Define the *beach line* (for ℓ) to be the set of points $q \in \mathbb{R}^2$ (all above the sweep line) whose distance to its nearest site in P' is equal to its distance from the sweep line (see the blue curve in Fig. 5(a)). That is

$$\text{beach}(\ell) \;=\; \{q \in \mathbb{R}^2 \mid \text{dist}(q, P^+(\ell)) = \text{dist}(q, \ell)\}.$$

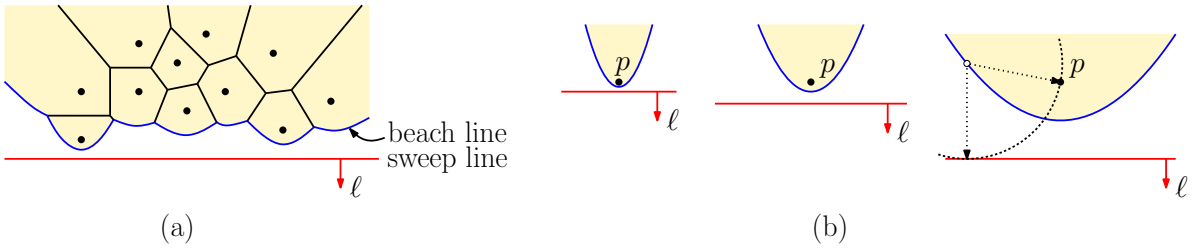


Fig. 5: (a) The beach line and (b) the evolution of a parabolic arc as the sweep line moves.

For all points on or above the beach line, the closest site in P must lie above the sweep line; that is, it must lie in $P^+(\ell)$. This means that it cannot be affected by any site below the sweep line. This further implies that no part of any Voronoi cell of a point lying below the sweep line can affect a point on or above the beach line. Thus, the region above the beach line is "safe" in the sense that there can be no unanticipated events.

What does the beach line look like? Recall from your high-school geometry that the set of points that are equidistant from a point (in this case a site p) and a line (in this case the sweep line ℓ) is a *parabola* separating p from ℓ. The parabola's shape depends on the relative distance between p and ℓ, and in particular, as ℓ moves further away from p, the parabola's shape becomes progressively "flatter" (see Fig. 5(b)). Note that in the special case when p lies on ℓ, the parabola degenerates into a vertical ray shooting up from p.

Thus, the beach line consists of the *lower envelope* of these parabolas, one for each site (see Fig. 5(c)). Note that the parabola associated with some sites may be *redundant* in the sense

that it will not contribute to the beach line. Also, some sites can contribute multiple arcs, but the beach line's total complexity cannot exceed $O(|P^+(\ell)|) = O(n)$. Because the parabolas are x-monotone, the beach line is as well. The beach-line point q where two beach-line arcs intersect is called a *breakpoint*. If these arcs are associated with sites p_i and p_j, then q is equidistant to these sites and they are the closest sites to q. Therefore, q lies on an edge of the final Voronoi diagram.

Fortune's Algorithm: Fortune's algorithm (at least the variant described in our textbook) simulates the growth of the beach line as the sweep line moves downward, tracing the paths of the breakpoints as they travel along the edges of the Voronoi diagram. As the sweep line moves, the parabolas forming the beach line change shape continuously. (For help visualizing this, run the applet described at the top of the lecture.) As with all plane-sweep algorithms, we will maintain a sweep-line status, and we are interested in simulating the discrete event points where there is a "significant event", that is, any event that changes the topological structure of the Voronoi diagram or the beach line.

Sweep-Line Status: The algorithm maintains the current location (y-coordinate) of the sweep line. It stores, in left-to-right order, the sequence of sites that define the beach line. (We will say more about this later.) **Important:** The algorithm does *not* store the parabolic arcs of the beach line. They are shown solely for conceptual purposes.

Events: There are two types of events:

Site events: When the sweep line passes over a new site, insert a new parabolic arc into the beach line.

Voronoi vertex events: (What our text calls *circle events*.) When an arc of the beach line shrinks to zero length, it disappears, and a new Voronoi vertex is created at that point.

The algorithm processes these two event types. As vertex events discover Voronoi vertices, it is easy to update the diagram as we go (assuming a reasonable representation of this planar cell complex) and link the entire diagram together. Let us consider the two types of events that are encountered.

Site events: A site event is generated whenever the horizontal sweep line passes over a site p_i. As we mentioned before, when the sweep line touches the point, its associated parabolic arc degenerates into a vertical ray shooting up from the point to the current beach line. As the sweep line proceeds downward, this ray widens into an arc along the beach line. To process a site event, we determine the sweep-line arc that lies directly above the new site. (We make the general-position assumption that it does not fall immediately below a vertex of the beach line.) Let p_j denote the site generating this arc. We then split this arc in two by inserting a new entry at this point in the sweep-line status. (Initially this corresponds to an infinitesimally small arc along the beach line, but as the sweep line sweeps on, this arc will grow wider. Thus, the entry for $\langle \ldots, p_j, \ldots \rangle$ on the sweep-line status is replaced by the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$ (see Fig. 6).

It is important to consider whether this is the only way that new arcs can be introduced into the sweep line. In fact, it is. We will not prove it here, but a careful proof appears in the

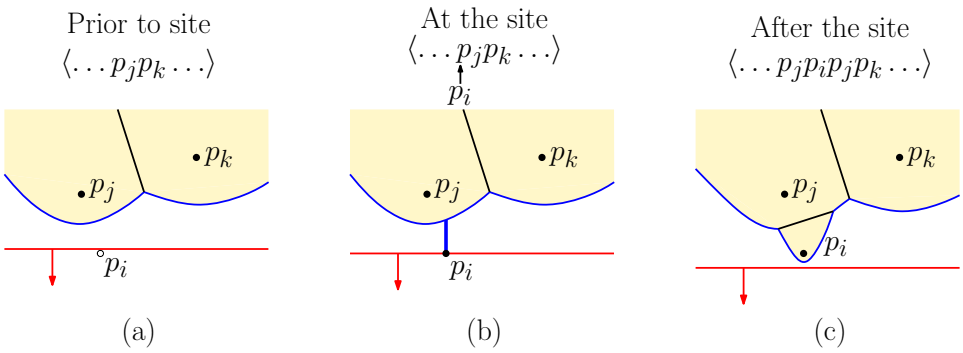


Fig. 6: Site event.

text. As a consequence, the maximum number of arcs on the beach line is at most $2n - 1$, since each new site can create one new arc and split an existing arc, for a net increase of two arcs per site (except the first). Note that a site may generally contribute more than one arc to the beach line. (As an exercise, you might consider what is the maximum number of arcs a single site can contribute.)

The nice thing about site events is that they are all known in advance. Thus, the sites can be presorted by the y-coordinates and inserted as a batch into the event priority queue.

Voronoi vertex events: In contrast to site events, Voronoi vertex events are generated dynamically as the algorithm runs. As with the line segment intersection algorithm, the important idea is that each such event is generated by objects that are *adjacent* on the beach line (and thus, can be found efficiently). However, unlike the segment intersection case, where pairs of consecutive segments generate events, here triples of points generate events.

In particular, consider any three consecutive sites p_i, p_j, and p_k whose arcs appear consecutively on the beach line from left to right (see Fig. 7(a). Further, suppose that the circumcircle for these three sites lies at least partially below the current sweep line (meaning that the Voronoi vertex has not yet been generated), and that this circumcircle contains no sites lying below the sweep line (meaning that no future site will block the creation of the vertex).

Consider the moment when the sweep line reaches the point where it is tangent to the lowest point of this circle. At this instant, the circle's circumcenter is equidistant from all three sites and from the sweep line. Thus all three parabolic arcs pass through this center point, implying that the contribution of the arc from p_j has disappeared from the beach line. In terms of the Voronoi diagram, the bisectors (p_i, p_j) and (p_j, p_k) meet at the Voronoi vertex, and a single bisector (p_i, p_k) remains. Thus, the triple of consecutive sites p_i, p_j, p_k on the sweep-line status is replaced with p_i, p_k (see Fig. 7).

Sweep-line algorithm: We can now present the algorithm in greater detail. The main structures that we will maintain are the following:

(Partial) Voronoi diagram: We store the partial Voronoi diagram constructed so far in a suitable data structure for planar subdivisions, for example, a doubly connected edge

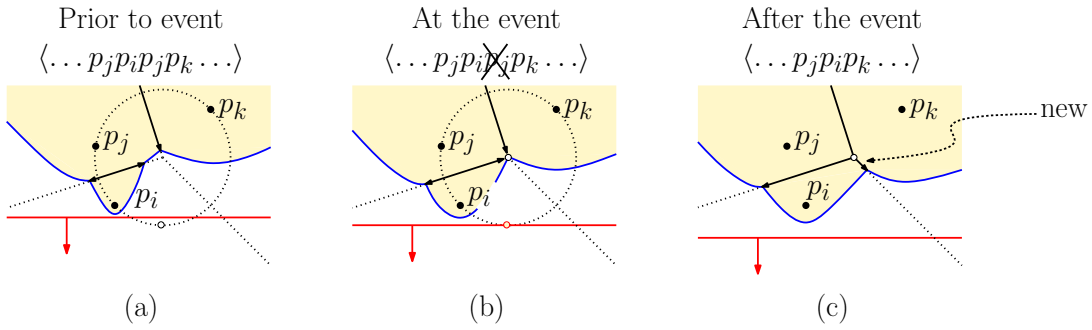


Fig. 7: Voronoi vertex event.

list. One technical difficulty arises because the diagram contains unbounded edges. We can handle this by enclosing everything within a sufficiently large bounding box. (It should be large enough to contain all the Voronoi vertices, but this is not that easy to compute in advance.) An alternative is to create an imaginary Voronoi vertex "at infinity" and connect all the unbounded edges to this imaginary vertex.

Beach line: The beach line consists of the sorted sequence of sites whose arcs form it. It is represented using a dictionary (e.g., a balanced binary tree or skip list). As mentioned above, we *do not* explicitly store the parabolic arcs. They are there only to derive the algorithm. Instead, for each parabolic arc on the current beach line, we store the site that generates it.

The key search operation is that of locating the arc of the beach line that lies directly above a newly discovered site. (As an exercise, before reading the next paragraph you might think about how you would design a binary search to locate this arc, given that you only have the sites, not the actual arcs.)

Between each consecutive pair of sites p_i and p_j, there is a breakpoint. Although the breakpoint moves as a function of the sweep line, you can compute its exact location as a function of p_i, p_j, and the current y-coordinate of the sweep line. In particular, the breakpoint is the center of a circle that passes through p_i, p_j, and is tangent to the sweep line. (Thus, as with beach lines, *we do not explicitly store breakpoints*. Rather, we compute them only when we need them.) Once we compute the breakpoint, we can determine whether a newly added site is to its left or right. Using the sorted ordering of the sites, we use this primitive comparison to drive a binary search for the arc lying above the new site.

The important operations that we will have to support on the beach line are:

Search: Given the current y-coordinate of the sweep line and a new site p_i, determine the arc of the beach line lies immediately above p_i. Let p_j denote the site that contributes this arc. Return a reference to this beach line entry.

Insert and split: Insert a new entry for p_i within a given arc p_j of the beach line (thus effectively replacing the single arc $\langle \ldots, p_j, \ldots \rangle$ with the triple $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$. Return a reference to the newly added beach line entry (for future use).

Delete: Given a reference to an entry p_j on the beach line, delete this entry. This replaces a triple $\langle \ldots, p_i, p_j, p_k, \ldots \rangle$ with the pair $\langle \ldots, p_i, p_k, \ldots \rangle$.

It is not difficult to modify a standard dictionary data structure to perform these operations in $O(\log n)$ time each.

Event queue: The event queue is a priority queue that can both insert and delete new events. It can also extract the event with the largest y-coordinate. For each site, we store its y-coordinate in the queue. All operations can be implemented in $O(\log n)$ time assuming that the priority queue is stored as an ordered dictionary.

For each consecutive triple p_i, p_j, p_k on the beach line, we compute their circumcircle. (We'll leave the messy algebraic details as an exercise, but this can be done in $O(1)$ time.) If the lower endpoint of the circle (the minimum y-coordinate on the circle) lies below the sweep line, then we create a Voronoi vertex event whose y-coordinate is the y-coordinate of the bottom endpoint of the circumcircle. We store this in the priority queue. Each such event in the priority queue has a cross link back to the triple of sites that generated it, and each consecutive triple of sites has a cross link to the event that it generated in the priority queue.

The algorithm proceeds like any plane sweep algorithm. The algorithm starts by inserting the topmost vertex into the sweep-line status. We extract an event, process it, and go on to the next event. Each event may result in a modification of the Voronoi diagram and the beach line, and may result in the creation or deletion of existing events.

Here is how the two event types are handled in more detail.

Site event: Let p_i be the new site (see Fig. 6 above).

(1) Advance the sweep line so that it passes through p_i. Apply the search operation above to determine the beach line arc immediately above p_i. Let p_j be the corresponding site.

(2) Applying the above insert-and-split operation, inserting a new entry for p_i, thus replacing $\langle \ldots, p_j, \ldots \rangle$ with $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$.

(3) Create a new (dangling) edge in the Voronoi diagram on the bisector between p_i and p_j.

(4) Some old triples involving p_j may need to be deleted, and some new triples involving p_i will be inserted, based on the change in neighbors on the beach line. (The straightforward details are omitted.)

Note that the newly created beach-line triple p_j, p_i, p_j does not generate an event because it only involves two distinct sites.

Voronoi vertex event: Let p_i, p_j, and p_k be the three sites that generated this event, from left to right (see Fig. 7 above).

(1) Delete the entry for p_j from the beach line status. (Thus eliminating its associated arc.)

(2) Create a new vertex in the Voronoi diagram (at the circumcenter of $\{p_i, p_j, p_k\}$) and join the two Voronoi edges for the bisectors (p_i, p_j), (p_j, p_k) to this vertex.

(3) Create a new (dangling) edge for the bisector between p_i and p_k.

(4) Delete any events that arose from triples involving the arc of p_j, and generate new events corresponding to consecutive triples involving p_i and p_k. (There are two of them. The straightforward details are omitted.)

The analysis follows a typical plane-sweep analysis. Each event involves $O(1)$ processing time plus a constant number of operations on the various data structures (the sweep line status and the event queue). The data structures have size $O(n)$, and each operation takes $O(\log n)$ time. Thus the total time is $O(n \log n)$, and the total space is $O(n)$.