Tiling the Heavens
In special cases where “the Heavens” is $\mathbb{R}^2$

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Isabella F. Jorissen

May 2016

Approved for the Division
(Mathematics)

James Fix

Acknowledgements

Without Jim's advice to me as a prospie, I would not be here today, nearing the end of my final semester at Reed, writing the acknowledgments section of my Math-CS thesis. Jim, thank you for learning with me throughout this process. Thank you for your encouragement when my thesis felt unoriginal and unambitious; moreover, thank you for reminding me that learning about something "old" is a worthwhile adventure in and of itself.

I feel humbled by my friends, family, and mentors who have supported me as I've navigated the Reed experience. Brooke, you have pushed me to grow as a person and community member— often faster than I thought I could. I have been baffled, at times, by your strong opinion that my opinions are worth voicing. I cannot express how instrumental you have been in helping me learn to stand my ground.

To Alessandra, Elise, Maddie, and the rest of the gang, thank you for patiently waiting through my asocial phases, for educating me, and for the parties. As I'm writing this, I'm reminded of the gratitude and respect I have for each of you. I love you all dearly; thank you for your friendship.

To Elaine, you have been a part of my life at Reed since Day One, and I wouldn't have it any other way.

To Mom and Dad, it will always be wonderful to return home to you two. You have always encouraged me to push myself further, and have always welcomed me back with open arms. If I have developed the ability to commit to a task, or the grit to muscle through something that feels way over my head, it is because of you.

Thank you all for your faith in me, your commitment, and your love.

Table of Contents

List of Algorithms

List of Figures

Abstract

This thesis gives a gentle introduction to the Voronoi diagram. We begin with a summary of Descartes' cosmology, finishing with an examination of his graphical representation of the heavens. We linger on the modern mathematical diagram his drawing evokes: the Voronoi diagram. We define the Voronoi diagram and its straight line dual, the Delaunay Triangulation, in $\mathbb{R}^2$. We present two algorithms to construct the Voronoi diagram: a brute-force algorithm relying on the intersection of half-planes, and Steven Fortune's classic sweep line algorithm. We examine Guibas and Stolfi's divide-and-conquer algorithm for the Delaunay triangulation. With an eye on the benefits of GPGPU programming, we conclude with a survey of previous efforts to parallelize the algorithms for the construction of the Voronoi and Delaunay diagrams.

Chapter 1

The Voronoi Diagram

"... To make this long discourse less boring for you, I want to wrap up part of it in the guise of a fable, in the course of which I hope the truth will not fail to manifest itself sufficiently clearly, and that this will be no less pleasing to you than if I were to set it forth wholly naked," Descartes, p. 21.

1.1 Descartes' Cosmological Picture

René Descartes spent four years writing *The World*, and abandoned it in 1633 after the Roman Inquisition condemned Galileo and his heliocentric theory. It remained largely unpublished for thirty-one years after its abandonment, save for some fragments which appeared in *Principia philosophiae. Treatise on Light*, first published as *The World*, was not published until fifteen years after Descartes' death, on a leap year almost four hundred years ago.

Stepping into the world which Descartes presents in *Treatise on Light* is astonishing in its own right, as his theory culminates in a cosmology where heavenly bodies lie unperturbed at the centers of endlessly swirling corpuscles. For our purposes, the salient portion of Descartes' theory lies in this vortex theory; his illustrations of the movement of bodies within the heavens are often touted as the first, however informal, use of the Voronoi diagram.

Under Descartes' view– which builds off of his contemporaries– there exists a trinity of elements. He writes, "the Philosophers maintain that above the clouds there is a kind of air much subtler than ours, ... They say too that above this air there is yet another body, more subtle still, which they call the element of fire,"[11, p. 17]. The third and final element is the element of earth, and it is the least "fine" of the three elements. These three elements may combine to create mixed or composite elements. Descartes believes that the elements of a higher order, that is to say, of a subtler sort, fill the gaps between the elements of a lower order so that they might constitute a perfectly solid body. "I say that this subtler air and this element of fire fill the gaps between the parts of the gross air that we breathe, so that these bodies, interlaced with one another, make up a mass as solid as any body can be," [11, p. 17].

In the section *How, in the world as described, the heavens, the sun and the stars* *are formed*, Descartes asks us to imagine a new world in which we might have a number of bodies made up of the same matter. For Descartes, any given body of matter is surrounded on all sides by other bodies, arranged in such a way that there is no void between any two of them [11, p. 25]. God agitates each body, giving it direction and motion; under this model, bodies are colliding constantly. The effects of a collision might be the breaking up or change in direction of one or more bodies. The smallest fragments, the products of many collisions, take the form of the finest first element which Descartes describes. The largest bodies do not break apart. Rather, they join together and form the third element.

The inclination of these "agitated" bodies is to move in a straight line, but the bodies all have a variety of masses and levels of agitation associated with them, so there emerges a rough ordering of the bodies about the center. Broadly, the smallest and least agitated bodies turn about the area closest to the center, while the larger or more agitated bodies trace out the larger circles around the center; at first blush, their movement might even resemble a straight line. Descartes superimposes this trinity of elements onto the principal parts of the universe: "the Sun and the fixed stars as the first kind, the heavens as the second, and the Earth with the planets and comets as the third,"[11, p. 20]. Then, the center around which the bodies turn is a star, composed entirely of the first element; the elements which turn around it are the heavens; and the other earthly bodies are composed entirely of the third element.

Descartes' depicted his cosmological understanding with a diagram, one most directly likened to a Voronoi diagram through his description of the heavens as a whole. Shown in figure 1.1, there is a heaven (we might call it a heavenly region) for each star. He writes: "So there are as many different heavens as there are stars, and since the number of stars is indefinite so too is the number of heavens. And the firmament is just a surface without thickness separating all the heavens from one another,"[11, p. 35]. The heavens, then, is a system of vortices whose centers are these heavenly bodies.

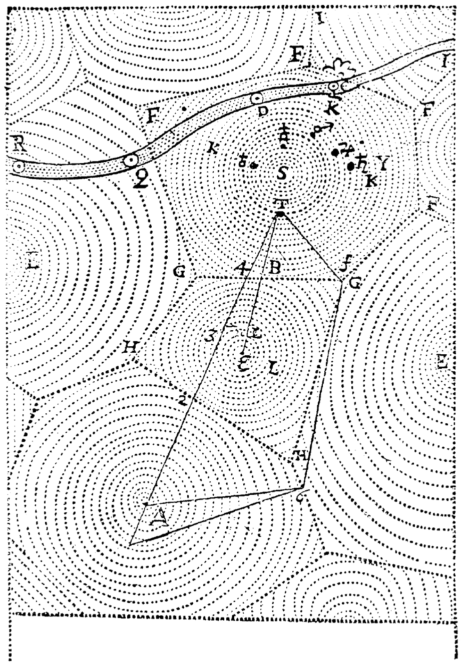


Figure 1.1: Descartes' Heavenly Regions

With Descartes' cosmology in the back of our minds, we look to figure 1.1. What Descartes has drawn around each star are (roughly) concentric circles representing the matter turning about the star. Examining the region near S more closely, we can see that the largest circles intersect with the circles emanating out of the other stars. In Descartes' terms, when circles surrounding a given star intersect with another set of circles, they do so at the *firmament*, which separates the heavenly regions from another. Visually, the firmament between any two heavenly regions— take, for example, S and ϵ — occurs along the set of points equidistant from both S and ϵ. The "corners" of the firmament, as Descartes calls them, are points where three circles— each originating from a distinct heavenly body— intersect. Many believe Descartes to be the first to depict our own object of study, the Voronoi diagram.

1.2 Definition of the Voronoi Tessellation in $\mathbb{R}^d$

Rather than dwelling on Descartes' understanding of the cosmos, we focus on the mathematical diagram that his picture evokes. To that end, let $P := \{p_1, p_2, \ldots, p_n\}$ be a finite set of n distinct points in $\mathbb{R}^d$. The Voronoi diagram of P, Vor(P), is a division of $\mathbb{R}^d$ into n regions. Specifically,

$$\text{Vor}(P) := \{V_1, V_2, \ldots, V_n\}$$

where each region V_i is defined by

$$V_i := \{\, q \in \mathbb{R}^d \mid \delta(q, p_i) \leq \delta(q, p_j) \text{ for all } \ p_j \in P \ \text{ where } j \neq i\}.$$

We take the $\delta(q, p)$ to be $\|p - q\|$, the usual measure of distance in $\mathbb{R}^d$. Figure 1.2 shows a Voronoi diagram of nine sites in the plane $\mathbb{R}^2$.

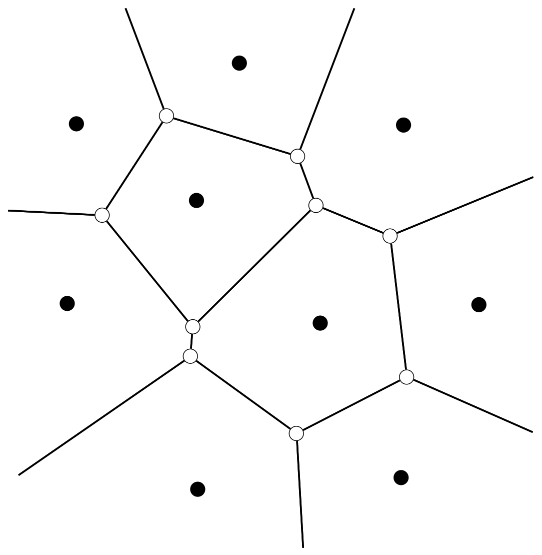


Figure 1.2: A Voronoi Diagram on nine sites

The Voronoi diagram in $\mathbb{R}^2$ consists of Voronoi vertices, Voronoi regions, and Voronoi edges. For the purposes of this thesis, we will be working with diagrams in $\mathbb{R}^2$. We compute the diagram on a set of sites, P, which are points in $\mathbb{R}^2$. A Voronoi region, or cell, is bounded by Voronoi edges; every point contained within the region is closer to the site associated with that region than it is to any other site.

Voronoi edges, represented by solid lines in figure 1.2, are the set of points which are equidistant from the sites on either side of the edge. These edges intersect at Voronoi vertices; every vertex is equidistant to three sites.

As we have just seen, the Voronoi diagram is a tiling of the plane $\mathbb{R}^2$ according to a proximity relationship between the point-set P and the points in the plane. In the next sections, we pick apart the Voronoi diagram and related structures. In deepening our understanding of the diagram, we will see that the information about proximity that the Voronoi diagram encodes can be applied to an array of problems in both "real life" and computational geometry, among many other areas of interest.

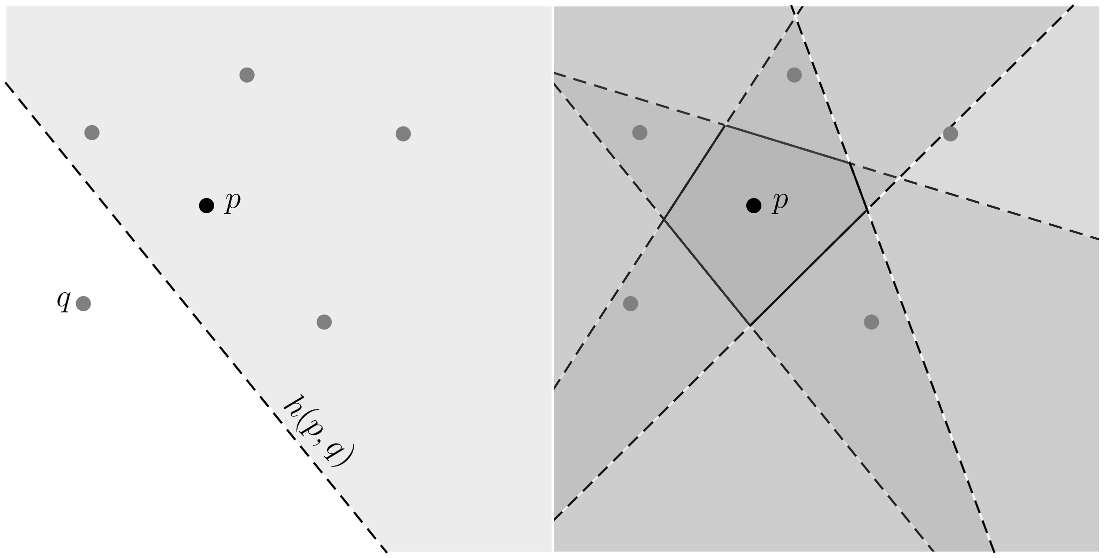
1.3 A Half-Plane Characterization

Every Voronoi region can be thought of as a (possibly unbounded) convex polygon whose edges are the bounding lines of the convex regions and whose vertices are the points of intersection between the bounding lines. Let's unpack that a little bit. Given two sites p_i and p_j, imagine the line, ℓ connecting them. Let b be the perpendicular bisector of ℓ, which splits the plane into two half-planes. Define the half-plane along b containing p_i with $h(p_i, p_j)$; similarly, define the half-plane containing p_j with $h(p_j, p_i)$. We can connect this notion of half-planes to our original definition of a Voronoi region: note that with the exception of the set of points lying on b, all of the points in $h(p_i, p_j)$ will *not* be in p_j's region, since every point in $h(p_i, p_j)$ is closer to p_i than it is to p_j. The points lying along b will be included in both regions, since they are equidistant from p_i and p_j. The dashed line in figure 1.3a is the perpendicular bisector of p and q. The shaded region represents the half-plane $h(p, q)$.

Define set of half-planes containing p_i as $H_i := \{\, h(p_i, p_j) \mid j \neq i\}$. There are $n-1$ half-planes in H_i. Consider the intersection of any two of these half-planes, $h(p_i, p_j)$ and $h(p_i, p_k)$. The set of points lying in the intersection of $h(p_i, p_j)$ and $h(p_i, p_k)$ are all of the points which are at least as close or closer to p_i as they are to p_j or p_k. Extrapolating a bit further, if we performed this intersection over all $n-1$ half-planes, we would have the necessary information to determine the set of points which are at least as close or closer to p_i as they are to all other sites. Which is, of course, the description of the V_i exactly. So we can also describe the region as the intersection of a set of half-planes:

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

Figure 1.3b illustrates how V_p is formed by the intersection of half-planes, it is the darkest shaded region bounded by a solid border.



(a) One half-plane, $h(p, q)$ (b) V_p from half-plane intersections

Figure 1.3: Half-Plane Intersections

Recall that a set is convex if the line segment connecting any two points in the set is also in the region. Because the half-plane is a convex set and the intersection of convex regions is also convex, the Voronoi region is convex. In section 2.1 we describe an algorithm to construct the Voronoi diagram using an approach which relies on half-plane intersections.

Note that some of the edges will be line segments, while others will extend out towards infinity. We will refer to the latter type of edge as a *half infinite edge*. Since each Voronoi region was the result of $n - 1$ half-plane intersections, we know that each such region has, at most, $n - 1$ edges.

1.4 The Dual

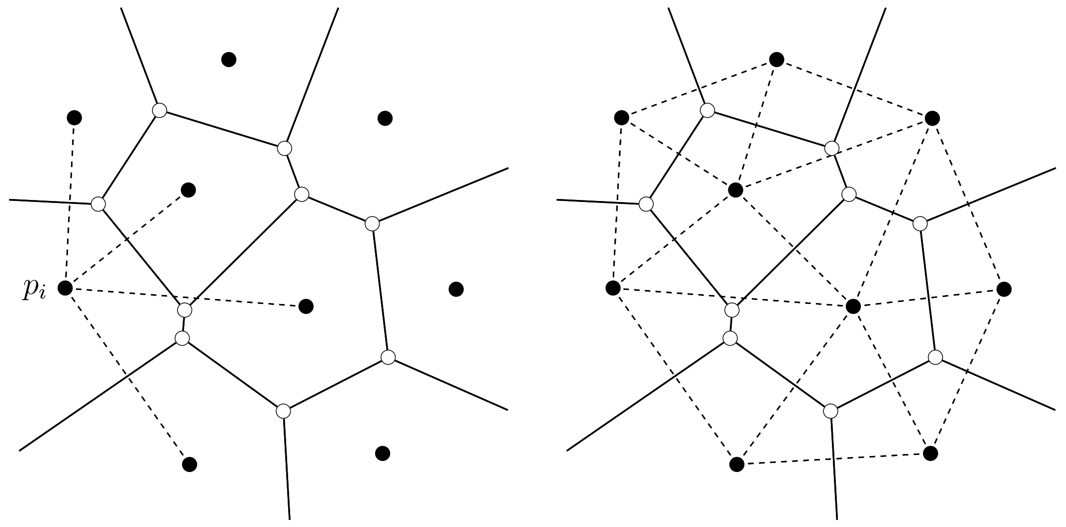
While the Voronoi diagram can be computed directly with relative— though certainly not always computational— ease, it can also be computed somewhat indirectly, that is, through the construction of its dual, the *Delaunay subdivision.* For those not already familiar with the idea of duality, choose a site $p_i \in P$. Using Vor(P) as a reference, draw a line connecting p_i to the sites of the regions adjacent to it, as in figure 1.4a. Do this for all the sites. The resulting diagram is the unique Delaunay triangulation of P, shown in figure 1.4b.

If we assume that the set of sites P is "well-behaved,"[1] then the resulting diagram is the unique Delaunay triangulation of those sites, Del(P). In particular, we'll assume that exactly three Voronoi regions meet at every Voronoi vertex for the remainder of this discussion. One of the many advantages of working with this dual is the ability

[1] We'll wave our hands at this for now, and in section 1.5 we'll define more precisely what it means for the set of sites to be "well-behaved."

to work with triangular faces instead of with polygonal regions with an unknown number of sides.

To get an idea for this entity before its definition, choose a site $p_i \in P$. Using Vor(P) as a reference, draw a line connecting p_i to the sites of the regions adjacent to it, as in figure 1.4a. Do this for all the sites. The resulting diagram is the unique Delaunay Triangulation of P, as shown in figure 1.4b.



(a) A site connected to its adjacent sites (b) All sites connected to neighboring sites

Figure 1.4: Voronoi Diagram (Solid) and Delaunay Triangulation (Dashed)

Let's take a couple steps back and work to define the Delaunay triangulation a bit more rigorously, so we can better examine its properties and relation to the Voronoi diagram. It's useful, for our purposes, to consider the *convex hull.* The convex hull of a set of points P is the smallest convex region containing P. Roughly, it is the intersection of all of the convex sets containing P. Precisely: given points q_1,... q_k, a *convex combination* of those points is defined as $\Sigma\alpha_i q_i$, where the α_i's are non-negative and $\Sigma\alpha_i = 1$. So, for our point set P, we are interested in the convex hull, denoted conv(P), defined as the set:

$$\text{conv}(P) = \{\, \alpha_1 p_1 + ... + \alpha_n p_n \;\mid\; \text{for}\, \alpha_i \in \Re,\ \alpha_i \geq 0 \text{ and } \Sigma\alpha_i = 1\,\}$$

Figure 1.5 illustrates the convex hull of the same point set used in figures 1.4a and 1.4b. This region of the plane is best understood in terms of a complete description of its boundary, as shown in figure 1.5.

A triangulation is a subdivision of the plane into triangular faces. More specifically, the triangulation of a point-set is a set of non-overlapping triangles where the union of the regions bounded by the triangles is the convex hull of P. The vertices of the triangles are taken from the point set, and each point in the set must be the vertex of at least one triangle. The boundary of the Delaunay triangulation of P is the convex hull of P. It's important to note here that every edge in the convex hull of a set of sites is an edge in the Delaunay triangulation.

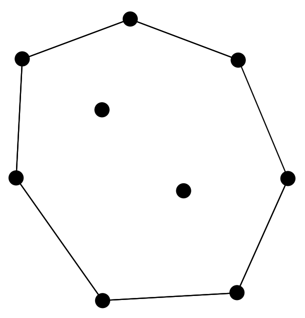


Figure 1.5: Convex hull of a set of points

We can confirm the Delaunayhood of a triangulation in two ways. For two sites $a, b \in P$, we say that that the edge $\overline{ab}$ connecting them is Delaunay if and only if there is a circle passing through them that contains no other sites in P. When this is true, we call the edge *locally Delaunay*. When every edge in the triangulation is locally Delaunay, we may say that the triangulation is *globally Delaunay*. An equivalent understanding of the Delaunay property operates on the faces of the triangulation instead of its edges. It is often referred to as the *empty circle condition*. The *empty circle condition* stipulates that for a triangle defined by three sites $a, b, c \in P$, the circumcircle of that triangle contains no other sites in P. These tests correspond to the geometric properties of the corresponding Voronoi diagram. Asking whether or not a face of the triangulation satisfies the empty circle condition is equivalent to checking whether or not a circle centered at a Voronoi vertex includes no sites and intersects with three sites.

So, given a triangulation, as in figure 1.6, one way to confirm its Delaunayhood is as follows: for each edge e in the triangulation, we consider the two triangles on either side of the edge. In the case of figure 1.6, the triangles which share edge e are Δdab and Δbcd. Because a is not contained in the circumcircle of Δbcd, shown as a dashed line in figure 1.6, we say that e passes the circle test. Specifically, we have shown that there is a circle namely, the circle passing through b, c, and d, that does not contain a or c. Symmetrically, we could ask whether or not c is contained in the circumcircle of Δdab, another possible circle.

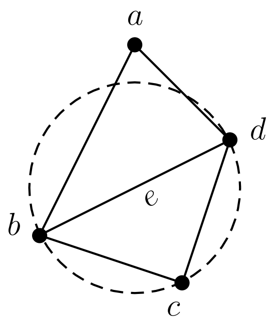


Figure 1.6: Testing the Delaunayhood of a small triangulation

Note that this formulation only confirms the existence of a circle passing through b and d that contains neither a nor c. It tell us nothing of the remaining sites in P. However, an important result was proven in Lee [20]: a triangulation T of the point set P is Delaunay if and only if every edge passes the circle test.

1.5 The Fine Print

In the previous section, we waved our hands at the constraint that the set of sites, P, be "well behaved." Here, we pause to introduce the notion of *general position*, and to make note of a special case. Later, when we examine more sophisticated algorithms to compute the Voronoi Diagram and Delaunay Triangulation, we will assume that the set of sites is in general position and that the special case is not the case.

1.5.1 General Position

When the set of sites we'd like to operate on is in *general position*, we can rest assured that we will not be tasked with handling any degenerate cases. These degenerate cases result from the geometric constraints which characterize the Delaunay and Voronoi diagrams. We might also note that since it is the enforcement of these properties which causes these special cases, that the notion of general position is particular to the metric we use.

When constructing Delaunay or Voronoi diagrams in Euclidean space we say that a set of sites, P, is in *general position* when no four sites in P are co-circular.

Though it isn't a particularly daunting task to understand why four co-circular sites would result in a degenerate case, it's not an exercise I'll leave to you, dear Reader. It is easiest to see the effects of such a scenario play out in the Delaunay triangulation, but I'll also note how it affects the Voronoi diagram.

Four co-circular sites in P results in a degenerate case because the Delaunay triangulation of P will not be unique. In particular, consider the simplest case of this, as illustrated in figure 1.7. When we attempt to triangulate $abcd$, we have a choice between inserting an edge bd and an edge ac. Since both of these edges satisfy the Delaunay condition, we have two valid triangulations of P. If we didn't try to triangulate $abcd$ at all, then our Delaunay triangulation would be a Delaunay subdivision, since one of its facets has four sides. The corresponding effect in the Voronoi diagram would be a Voronoi vertex with out-degree four instead of three.

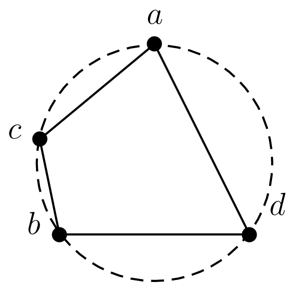


Figure 1.7: Four Co-circular sites

As we will discover in section 2.2, a Voronoi vertex exists at the center of the circumcircle of the corresponding facet of the Delaunay triangulation. In figure 1.7 for example, suppose we inserted edge ab. Then we would have two faces, namely acb and bda. Since these four sites are co-circular, the circumcircle of the two faces is the same, leading us to insert two Voronoi vertices in the same location (that is to say, the at the center of the circle). Note that we would have had inserted the Voronoi vertex at the same location even if we had inserted edge bd. In practice, we will often ignore this degeneracy by allowing two Voronoi vertices in the same location.

1.5.2 The Special Case

When all of the sites in P are collinear, we find ourselves with a special case. Consider one of the simplest scenarios, where P is a set of three collinear sites: a, b, and c, as shown in figure 1.8. This time, we'll focus on what the effects would be like in the Voronoi Diagram. The dashed lines in the figure denote the perpendicular bisectors of the line segments connecting adjacent sites.

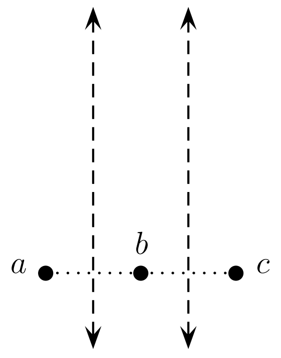


Figure 1.8: Voronoi Diagram of Three Collinear Sites

More generally, note that if all n sites in P are collinear, then there cannot be a triangulation of these sites. Further, the Voronoi Diagram of P is a set of regions whose boundaries are the $n-1$ parallel lines bisecting adjacent sites.

1.6 A Graph Representation

When the dimension of the space in which we compute the Voronoi Diagram or Delaunay triangulation is low (i.e $d = 2,\ d = 3$), it is difficult to divorce the geometric realities of the diagram as it is represented in $\mathbb{R}^d$ from the more subtle topological and combinatorial aspects of the diagrams. As the dimension increases, our ability to reason about and visualize the Voronoi diagram and Delaunay Triangulations by relying on its geometric properties diminishes. In this section, we introduce the notion of the Voronoi and, implicitly, Delaunay *graphs*— where $d = 2$— and shift our focus onto the relationships between the disparate components of the diagrams.

1.6.1 Obtaining the Voronoi Graph

The various components of a Voronoi diagram— its edges, vertices, and regions— relate to each other in a non-geometric way. Namely, there is an underlying combinatorial structure associated with a diagram. Consider the set of edges of a Voronoi diagram. An edge connects two Voronoi vertices. We can leverage this fact to describe the Voronoi diagram in a combinatorial way: as a graph.

Recall that a graph is a collection of vertices and edges. Given $\mathrm{Vor}(P)$, let C be the set of Voronoi vertices and let E be the set of Voronoi edges. Every Voronoi edge e in E is a pair (c_i, c_j) for two distinct c_i, c_j in C. Through describing $\mathrm{Vor}(P)$ in this way, we recognize it as a graph whose embedding in $\mathbb{R}^2$ satisfies the geometric characteristics of the Voronoi diagram.

Our attempts to consider $\mathrm{Vor}(P)$ as a graph will be thwarted if we don't address the edges which do not necessarily connect two Voronoi vertices. These are the *half-infinite edges*, discussed in section 1.3. To account for this, we can introduce a vertex "at infinity". Then, we assign the vertex at infinity to be second endpoint of each half infinite edge in the diagram. Figure 1.9 illustrates how the addition of a vertex at infinity changes the Voronoi diagram of nine sites we saw in figure 1.2.

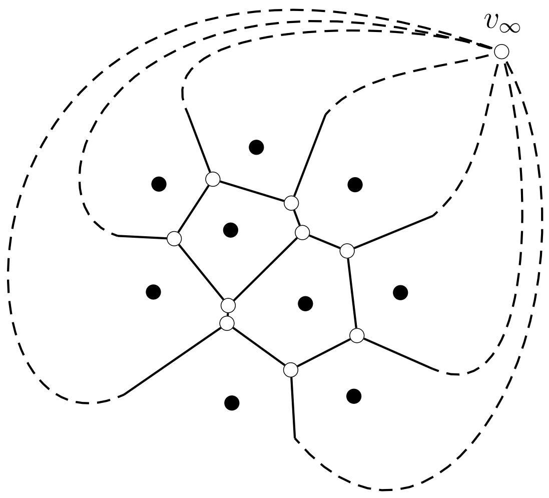


Figure 1.9: Connecting Half-Infinite Edges to v_∞

1.6.2 Orienting the Voronoi Graph

Not much has been said about how the faces (regions) of the Voronoi diagram are represented combinatorially. To that end, we may think of (the boundary of) a Voronoi face as an ordered collection of edges. Consider a Voronoi face V_i in $\mathrm{Vor}(P)$ whose region is bounded by a closed k-gon. Note that the k vertices of V_i are some collection $v_1, \dots, v_k$ such that each $v \in C$. Imagine you are standing atop v_1 in such a way that the site associated with that region, p_i, is to your left. Walk along the boundary of the region in a counterclockwise direction to the next vertex, call it v_2. We've just walked along the edge connecting v_1 to v_2. Often, we will think of v_1 as the edge's origin, and of v_2 as its destination.

We can continue in this fashion until we've walked the entire perimeter of the region V_i. At the end of such an excursion, we will have curated a list of edges, namely, $(v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, v_1)$. Note that each edge in this ordered set is such that the destination of a given edge is the source of the next edge in the set.

When describing edges and faces in this manner, we impose a counterclockwise orientation on the Voronoi graph. What we've done, somewhat implicitly, is convert the undirected Voronoi graph to a directed one. More precisely, for every undirected edge $(c_i, c_j) \in C$ we create two directed edges, namely, (c_i, c_j) and (c_j, c_i).

This has an effect on the dual diagram as well. Specifically, recall that there is a vertex in $\mathrm{Del}(P)$ located at each site in P. In particular, consider the vertex located at p_i, i.e the Delaunay vertex corresponding with the region V_i. We know that the directed Voronoi edge (v_1, v_2) borders p_i on the left. Let its *twin*, the oppositely directed edge with the same endpoints, (v_2, v_1), border some site p_j on its left.[2] This implies that there is an directed edge in the Delaunay diagram (p_i, p_j). More generally, for a given Voronoi edge, e, let a be the face to the left of e and let and b be the face to the left of $e.twin$. Then there is an edge (a, b) in $\mathrm{Del}(P)$. Connecting this back to our previous example, we see that the Delaunay vertex at p_i has out-degree k. In summary, the effects of imposing a counterclockwise orientation on the description of the Voronoi region V_i propagate to the Delaunay vertex at p_i: namely, we can consider the fan of edges originating at p_i to have the same counterclockwise ordering.

1.6.3 An Upper Bound

We can leverage the properties of a planar[3] graph further. Specifically, we can apply *Euler's Formula* to show upper bounds on the number of vertices, edges, and faces in $\mathrm{Vor}(P)$.

Euler's Formula states that any planar graph with v vertices, e edges, and f faces (regions, for our purposes) the following is true: $v - e + f = 2$. Furthermore, if the graph is *simple,*[4] that $e \leq 3n - 6$.

We apply Euler's Formula to slightly modified $\mathrm{Vor}(P)$. Let $n = |V|$, the number of Voronoi regions; $k = |C|$, the number of vertices not including the vertex at infinity, and $m = |E|$, the number of edges.

$$(k + 1) - m + n = 2.$$

Since every edge has exactly two endpoints and every vertex is of degree three,[5] we can say that $2m \geq 3(k + 1)$. Together with Euler's formula, this implies that when

[2]What this means is that the undirected edge $v_1\ v_2$ is a shared edge of the regions V_i and V_j.

[3]Recall that when a graph is planar, it can be embedded in the plane in a way that no edges in the graph intersect unless they have the same endpoints. This is clearly the case for a Voronoi diagram embedded in $\mathbb{R}^2$

[4]A simple graph is one which does not contain multiple edges with the same endpoints or edges whose two endpoints are the same, often called a "loop".

[5]This doesn't include the vertex at infinity, which will have out-degree *at least* three.

$n \geq 3$, the number of vertices is at most $2n - 5$ and the number of edges is at most $3n - 6$. Through recognizing that the number of regions is equal to the number of sites and that the upper bounds on k and m correspond linearly with n, we can say that the Voronoi diagram has linear complexity.

1.7 Applications

Beyond the intrinsic allure of the Voronoi diagram and its straight line dual, the Delaunay triangulation, lies a fascinating set of linear time reductions to some central proximity problems. This means that if we take the time to pre-process a set of sites into a Voronoi diagram, then we can answer many different questions in linear (or better!) time. This is especially significant when the time complexity of answering that question is greater than $\mathcal{O}(n)$.

The Voronoi diagram is often motivated by the classic Post Office problem. Though perhaps slightly dated, the Post Office problem captures one of the central use cases for the Voronoi diagram, which is its ability to answer the *nearest neighbor problem*. Briefly, the Post Office problem supposes that you are at a point q, perhaps in a suburb or city that is unfamiliar to you, and that you'd like to send a piece of mail. Given that there are a number of post offices in the area near you, the Post Office problem asks: "Which post office has the most convenient location?", that is to say, which office is closest to our query point q? If we wanted to answer lots of questions like this one, it would be a boon to devote time beforehand to build a Voronoi diagram using the point-set determined by the locations of the offices. Then, the question becomes: "Which Voronoi region contains q?". This is a familiar question, and one can answer in $\mathcal{O}(\log n)$ time with linear auxiliary storage [21, p. 214].

However, a more pressing question is posed by what we might call the "Starbucks problem." Given the locations of every Starbucks in a given area, which point is furthest from every Starbucks? Consider the point-set P composed of the location of each Starbucks. To answer this question, we begin by just considering the closed regions of the Voronoi diagram on P, $\text{Vor}(P)$. We recall that points located on Voronoi edges are equidistant from the sites on either side of the edge. Voronoi vertices, further, are equidistant from three sites. Our quest is to find the point q[6] that maximizes the distance to the site nearest to it. Clearly this must occur at a Voronoi vertex. At this point, this problem is precisely the issue of finding the *largest empty circle* (LEC) in the Voronoi diagram. More precisely, the LEC of a point-set is the largest circle whose center is contained in the convex hull of the point-set and also includes no points in the point-set. Applying this to the Starbucks problem, we consider a circle centered at each Voronoi vertex, whose radius is determined by the distance to the nearest site in P. The Voronoi vertex with the largest circle, then, is the location of our caffeine-deprived friend.

In our treatment of the Starbucks problem, however, we didn't consider the unbounded Voronoi regions. This can be addressed easily. In fact, Shamos and Hoey

[6] We assume that q is within the boundary of the diagram, by which we mean the union of all of the closed Voronoi regions in $\text{Vor}(P)$.

[22] showed in their 1975 paper that the center of the largest empty circle is either located at a Voronoi vertex in Vor(P), or it is a circle whose center is located at one of the points where the boundary of Vor(P) intersects with conv(P).

Related to the nearest-neighbor problem are the *closest pair* and *all nearest neighbors* problems. The closest pair problem asks, given a set of sites P, which pair of points p_i and p_j in P are closest? The all nearest neighbors problem can be framed similarly: given a set of sites P, return a collection of pairs (p_i, p_j) such that p_j is the nearest neighbor of p_i.[7] When given a Voronoi diagram on those sites, the all nearest neighbors problem can be answered in linear time by recognizing that the nearest neighbor of a single site must be one of the sites with which it shares an edge. Equivalently, we can consider the Delaunay triangulation on the same set of sites, Del(P). Recall that each vertex on the Delaunay triangulation is a site of Vor(P). Furthermore, note that the line segment connecting a vertex p_i of Del(P) to its nearest neighbor is, and must be, one of the edges in Del(P). So, if we spend time on the order of $\mathcal{O}(n)$ computing the result of the all nearest neighbors problem on Vor(P), we will, by definition, have the answer to the closest pair problem for each site in P.

We conclude this section by noting that the Voronoi diagram and Delaunay triangulations are not just used as stepping stones to efficiently answer other proximity problems. In two dimensions, Delaunay diagrams are optimal with respect to several geometric criterion, including the property that the Delaunay triangulation maximizes the minimum angle.[8] Moreover, a novel application of Voronoi diagrams was proposed in a paper co-authored by Andrew Bray, "Voronoi residual analysis of spatial point process models with applications to California earthquake forecasts" [7]. Bray et al. proposed a novel form of residual diagram where the partitions are determined by the Voronoi cells generated from an observed point pattern. They applied the Voronoi residual analysis to a model widely used to describe earthquake catalogs, Epidemic-Type Aftershock Sequence model. They compared their results to the Hector Mine earthquake catalog, and found that the Voronoi residual analysis suggested the model over-predicted seismicity on the periphery of the fault and under-predicted seismicity along the fault.

[7]Note here that the closest pair problem is really a specification of the nearest neighbors problem where the query point q is some p_i in P and not arbitrary q in $\mathbb{R}^2$.

[8]For more reading on the optimality of the Delaunay triangulation, and contrained Delaunay triangulations, I refer the interested reader to Cheng, Dey, and Shewchuk [8].

Chapter 2

Constructing the Voronoi Diagram

In this chapter, we will first pause to consider a brute-force half-plane intersection algorithm. It is a natural extension of the discussion in section 1.3 and runs in $\mathcal{O}(n^3)$ time. However, there are far more optimal approaches to the construction of the Voronoi diagram. They fall into three rough categories: incremental approaches, divide and conquer approaches, and sweep line approaches. The worst case time complexity for each of these is $\mathcal{O}(n^2)$, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log n)$ respectively. In section 2.2, we will examine a classic algorithm to compute the Voronoi diagram: Steven Fortune's sweep line algorithm.

Before we move on, we might note that we can't do better than $\mathcal{O}(n \log n)$. Curious reader that you are, you might wonder: "Why?" And the answer is this: we can show, quite easily, in fact, that the problem of sorting numbers is *reducible* to the problem of computing the Voronoi diagram. This means that given a set of integers, we can use an algorithm to construct of Voronoi diagram as a subroutine to compute the sorted set. Furthermore, it implies that we will never be able to construct the Voronoi diagram faster (in terms of complexity) than we will be able to sort integers. Thus, an immediate consequence of this reduction is a tight bound on the Voronoi diagram: it can be constructed in optimal time $\theta(n \log n)$.

2.1 A Naïve Approach

In section 1.3, we described a Voronoi cell as the intersection of half-planes containing the site associated with that region. In the same section, we saw that the Voronoi region is a convex region, which may or may not be unbounded. Finally, we noted that if $|P| = n$, then no region V_i in $\text{Vor}(P)$ can have more than $n-1$ edges.

From these observations, we can cobble together an algorithm for constructing $\text{Vor}(P)$, though the strategy is a brutal one. For each site p_i in P, we compute the set of $n-1$ half-planes containing p_i; we refer to this set as H_i. Then, we find the point of intersection (if there is one) of every unique pair of half-planes. Since $|H_i| = n-1$, note that there are $\frac{(n-1)(n-2)}{2}$ pairs of half planes.

Note that a half-plane is a region bounded by some line $ax + by + c = 0$.[1] Cor-

[1]For visual clarification, refer to figure 1.3a in section 1.3.

respondingly, it can be described with an inequality of the form $ax + by + c \leq 0$. Knowing that the region V_i is bounded by the Voronoi edges which are segments of the boundaries of some H_i, our goal is to determine where the endpoints of those edges are, namely, where they intersect to meet at a Voronoi vertex.

For example, given two half-planes $h(p_i, p_j)$ and $h(p_i, p_k)$, we compute the point of intersection, q. In order for q to be a Voronoi vertex, it must be the case that it satisfies: $q \leq ax + by + c$ for each of the half-planes in H_i. That is to say, that q must lie within (or on the boundary of) the common intersection of all half-planes in H_i. If q satisfies this property, then it will be a vertex of the convex region, V_i, we are describing.

When we have performed this step for each pair of half-planes, we will have a convex set, which we refer to as *intersections* in algorithm 1, consisting of at most $n-1$ points. To describe the region fully, we find the Voronoi edges of this region through computing the convex hull of this set. We repeat this process for each site in P. VoronoiFromHalfPlanes provides a pseudocode implementation of this approach.

Algorithm 1 VoronoiFromHalfPlanes

```
Require: Set of sites, $P$, which are in general position
 1: procedure VoronoiFromHalfPlanes($P$)                      ▷ Construct Vor($P$)
 2:     Vor($P$) ← {}                                          ▷ Initialize empty diagram
 3:     for $p_i$ in $P$ do
 4:         half_planes ← $H_i$                                ▷ Set of half planes containing $p$
 5:         intersections ← {}                                 ▷ Set of half plane intersections
 6:         for each pair of half planes, ($h_j$, $h_k$) in half_planes do
 7:             $q$ ← HalfPlaneIntersection($h_j$, $h_k$)
 8:             is_in ←HalfPlaneSetContains($H_i$, $q$)
 9:             if is_in then
10:                 AddIntersection(intersections, $q$)
11:             end if
12:         end for
13:         $V_p$ ←ConvexHull(intersections)
14:         AddRegion(Vor($P$), $V_p$)
15:     end for
16:     return Vor($P$)
17: end procedure
```

We use helper functions to abstract away the cumbersome math associated with computing many straight line intersections. Namely, HalfPlaneIntersection in line 7 takes two half planes $h(p_i, p_j)$ and $h(p_i, p_k)$, referred to as h_j, h_k, and returns the point of intersection, if there is one. HalfPlaneSetContains, used on line 8, takes the set of half-planes, H_i, associated with the site p_i along with q, and returns True if the point is contained in every half-plane in the set. By the time we reach line 13, we have a distinct set of points, *intersections*, which are completely contained in the common intersection of H_i. These points are the Voronoi vertices of this region. To determine the Voronoi edges, and describe V_i in terms of its boundary, we compute the convex hull of *intersections* function ConvexHull.

2.2 A Sweep Line Approach

In the 1980's Steven Fortune proposed an algorithm with asymptotic running time $\mathcal{O}(n \log n)$ and using $\mathcal{O}(n)$ space. Fortune's algorithm makes use of a "sweep line"; this approach conceptually works by imagining a horizontal line, one that starts above the sites and then descends past the sites, ending below them [12].

2.2.1 The Actors in Our Play

Events

Very generally, the points of interest in any sweep line algorithm denote moments where we must evaluate whether or not we need to do anything else before continuing to scan the plane. Within the context of Fortune's algorithm, encountering a point of interest signifies that we've learned something new about the Voronoi diagram, such as a Voronoi edge or Voronoi vertex.

We call the points of interest *events*, and there are two types of events we concern ourselves with. The first type, the *site event*, is the simplest. When the sweep line encounters a site, we learn that the Voronoi diagram will have one more region, and we'll also discover two of its bounding edges. In section 2.2.2, we will take a closer look at how to handle a site event. The other type of event is called a *circle event*. When the sweep line intersects with the location associated with the circle event, it signals the addition of a new Voronoi vertex. In section 2.2.2, we'll see how those come about and what it means to handle a circle event.

Finally, to store these upcoming events we might keep them in a *priority queue*, which keeps the events sorted by descending y-coordinate (since the sweep line is traveling "down" the plane). Note that we know about all of site events in advance, but we may need to add and remove circle events as necessary. A priority queue will allow us to maintain the sorted order and perform these kinds of operations in optimal time.

The Beachfront

Now, an astute reader might wonder how exactly we construct the Voronoi diagram with nothing but a priority queue and a vague notion of a line that falls down the plane. After all, there's no obvious method by which the intersection of the sweep line with a point of interest might yield a Voronoi vertex or edge. Further, it would seem that even if we could provide such a relationship, it would be subject to change as soon as the sweep line encountered the next point of interest.

Enter the *beachfront.*[2] As we'll soon see, the beachfront is a way for us to monitor the "in-progress" portion of the Voronoi diagram. It is a frontier of sorts: every point on or above the beachfront has already been associated with a region, while the points below have not. As the sweep line descends, the parabolic arcs which make up the beachfront shrink and swell. The intersection of adjacent arcs in the sequence trace out Voronoi edges. Arcs are added when site events occur and removed when circle events are handled.

Why parabolas? To answer this question, consider the portion of the plane that the sweep line has already encountered: this portion constitutes a closed half-plane, s^+. In figure 2.1, s^+ is the shaded gray area. If a site exists in s^+,—as p_j does—then any point q in s^+ which is closer to that site than it is to s will not be assigned to a site below s. When the sweep line s encounters p_j we handle the site event: we consider the locus of points which are closer to p_j than to s. The set of points equidistant to s and p_j form a parabolic arc. This arc, denoted α, in figure 2.1, is added to a a sequence of parabolic arcs called the *beachfront.* In figure 2.1, α is the first arc in the beachfront. We will refer to the beachfront as $\beta(P)$.

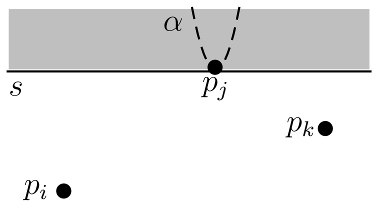


Figure 2.1: Adding the first arc, α, to the beachfront

Reader, it is important to make a distinction between the parabola defined by the site's location and the sweep line, and the arc inserted into the beachfront, which is a segment of that parabola. When subsequent site events occur, as in figure 2.3b, the inserted arc ω, intersects with the arc directly above it, effectively splitting into two pieces. At the moment where the new arc, ω, is inserted into the beachfront, it is a simple vertical line (or a parabola with no width). To insert ω into the beachfront, we split the arc above it, α into two arcs and insert ω in between the split arcs. We record the points where ω intersects with the arcs adjacent to it, and we refer to these points as *breakpoints*[3] As s moves downward, the breakpoints of each arc in the beachfront $\beta(P)$ change since the parabola that governs the arc widens. Another way to understand this is to recognize that that parabola whose corresponding arc is ω has as its focus the site p_i and s as its directrix. If we were to draw lines that tracked the movement of the breakpoints for the beachfront as the sweep line descended, they

[2]The notion of a beachfront is slightly misleading, as it's not warm and sunny and there are no palm trees swaying in the distance here.

[3]Note that this means that the right breakpoint of ω is the same as the left breakpoint of the arc which is adjacent on the right. Similarly, the right breakpoint of the arc adjacent to ω on the left is equal to ω's left breakpoint.

would each trace out the Voronoi edges.

The second way the beachfront's structure changes is when an arc disappears. Circle events trigger the removal of an arc from the beachfront. We'll define the notion of a circle event by studying a small portion of the beachfront. Consider the arcs γ, α, and ω, which are segments of the parabolas defined by three sites p_i, p_j, and p_k, as in figures 2.3c and 2.3d. Define a circle, C, with the three sites p_i, p_j, p_k. Though the reasons aren't immediately obvious, we want to keep track of two points: the center of the circle, q, and the lowest point of the circle, ℓ.

If α is the arc that is shrinking, when does it disappear? Recall that the center of the circle, q, is equidistant from p_i, p_j, p_k and l. When the sweep line reaches ℓ, the arcs corresponding to p_i and p_k, γ and ω, respectively, intersect at q. At this point, the right breakpoint of γ and the left breakpoint of ω become equal to the left and right breakpoints of α, and α disappears. So, we handle the circle event C when the sweep line reaches ℓ. The effect of this on the beachfront is the removal of arc α. Outside the context of the beachfront, the circle event signals the introduction of a Voronoi vertex at q, the point where the two edges traced out by the intersections of γ, α, and ω meet.

To summarize, the beachfront is a sequence of parabolic arc segments. The size of the beachfront, that is, the number of arcs composing the beachfront, must be less than or equal to $2n-1$, where n is the number of sites. Each arc is associated with a site, and there may be multiple arcs associated with the same site. As the sweep line moves downward, the equations of the parabolas governing the arcs change, triggering a shift in the breakpoints of those arcs as well. The structure of the beachfront is a function of site events and circle events. A site event occurs when the sweep line intersects with a site (i.e they both have the same y-coordinate) and a circle event occurs when the sweep line intersects with the lowest point (that is to say, the point with the minimum y-coordinate) of a circle. Arcs are added to the beachfront when site events occur, and removed when circle events occur. Looking past the horizon of the beachfront, the effect of an adding an arc is the growth of a new edge, and the effect of removing an arc is the addition of a vertex at the intersection of two edges.

2.2.2 Putting It All Together

Armed with an understanding of how site events and circle events effect the structure of the beachfront, we might consider ourselves sufficiently motivated to understand the particulars of how each of these parts yields information about the Voronoi diagram. We know that the sweep line travels downward, handling site and circle events as it encounters them. In fact, we know a lot about the site events; since we know all the sites before the sweep line begins, we can simply order the sites by decreasing y-coordinate and initialize the event queue with that information. Unlike site events, we don't know exactly when or how many circle events will occur.[4] So, circle events

[4]Some circle events will be *false alarms*, a highly technical term we'll be defining soon. Aside from these false alarms, recall that handling a circle event adds a vertex to the finished diagram, and we know from section 1.6.1 that an upper bound for the number of vertices is $2n-5$ for a Voronoi diagram on n sites.

complicate the problem insofar as they punctuate an otherwise predictable order of site events. What we do know is that we'd like to maintain an event queue, which consists of—and is ordered by—all of the site events and the known circle events.

Circle events must be sought out. Recall that a unique circle is defined by three non-collinear points. Circle events trigger the removal of an arc from the beachfront and the addition of a vertex at the center of the circle, which is also the point where the edges traced out by the breakpoints of the removed arc meet. We define a circle event for every triple of consecutive arcs on the beachfront. This means that we need to check for new circle events whenever the sequence of arcs in the beachfront changes.

So when we handle a site event and add it to the beachfront, there are three possibilities:

- The inserted arc is the first in the beachfront
- The inserted arc breaks up the segment above it, as in figure 2.2a
- The inserted arc intersects the beachfront at the intersection of two arcs, as in figure 2.2b

In the first case, we clearly don't need to add a circle event, since there are no other arcs in the beachfront. In the second case, we need to consider the three new triples of arcs. Note that there is at least one circle event in the queue: $C(p_i, p_j, p_k)$, with corresponding arcs γ, α, ω. Let χ be the inserted arc corresponding to p_m, so arc α is split into α' and α'', as in figure 2.2a. The new triples are, from left to right: γ, α', χ; α', χ, α''; and χ, α'', ω. Since the second triple has both α' and α'' in it, we know that we do not need to add a circle event here. However, we need to add circles for the other two, these circles are: $C(p_i, p_m, p_j)$ and $C(p_m, p_j, p_k)$.

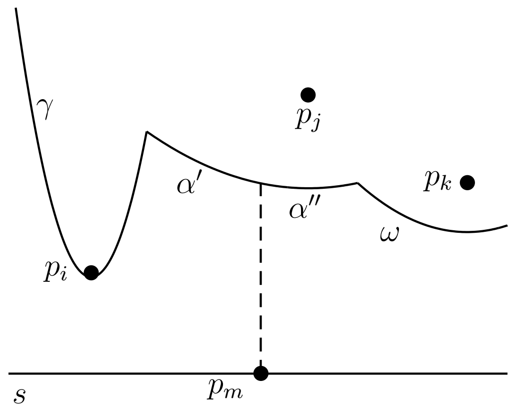
In the third case, we didn't have to split an arc, but we still need to consider new triples. Namely: γ, α, χ; α, χ, ω; and if there is an arc to the right of ω, call it ψ: χ, ω, ψ. We add circle events for each of these three triples.

Note that in both of these cases, the arcs associated with $C(p_i, p_j, p_k)$, γ, α, ω, are no longer sequential. The edges traced out by the intersections of these arcs will no longer meet at $C(p_i, p_j, p_k)$'s center. So we call $C(p_i, p_j, p_k)$ a *false alarm*, and we remove it from the event queue.

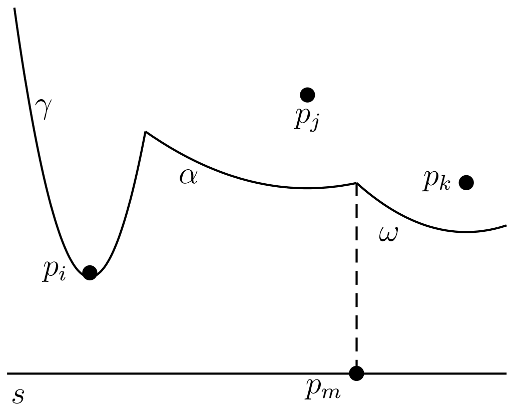
As long as the sites associated with the arcs are not collinear and are distinct, we will be able to describe a circle with a finite radius. Note that a circle event exists in the event queue if and only if it satisfies:

- The property that the circle defined by the sites corresponding to the triple of arcs intersects the sweep line
- The circle event hasn't already been deleted from the event queue

If the circle event was already deleted from the event queue, we've either already handled it, or it was an *false alarm*. As we've seen, it's often the case that handling a site event has a cascading effect wherein the event itself doesn't create a circle



(a) Site event at p_m splits arc α into α' and α''

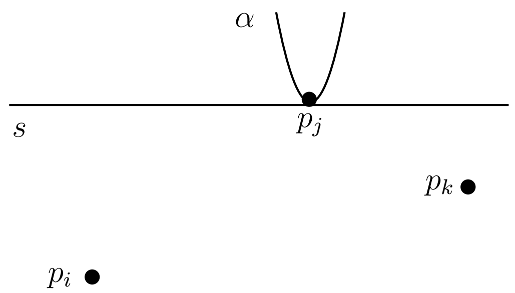


(b) Site event at p_m is directly below the intersection of α and ω

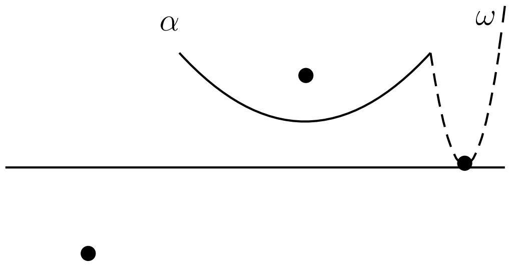
Figure 2.2: Inserting χ: two ways a site event might affect the beachfront

event, but creates a situation where one or more circle events would need to added or removed from the event queue. Similarly, handling a circle event can also lead to the creation or removal of one or more circle events, since the disappearance of an arc can lead to new combinations of triples on the beachfront.

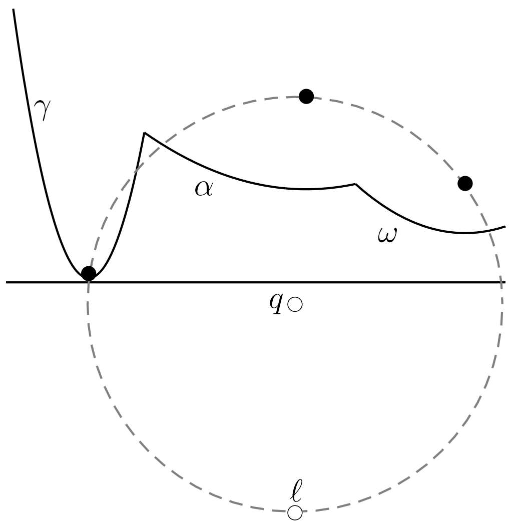
Figure 2.3 illustrates the application of Fortune's algorithm to a small set of three sites: p_i, p_j, and p_k. To understand how to handle a circle event, let's investigate this example. In figures 2.3a to 2.3c we are adding arcs to the beachfront by handling the site events associated with p_j, p_k, and p_i, respectively. In figure 2.3c, we also add a circle event to the event queue, since there are now three consecutive arcs in the beachfront with distinct sites. At this point, the circle event is the only event in the queue left to handle. Figure 2.3d illustrates how the arc α has shrunk as the sweep line has moved downward. The dotted lines reveal how the breakpoints of α have shifted during this time. When the sweep line intersects with ℓ, these edges will meet at q, and we will add a Voronoi vertex at q, as shown in figure 2.3e. At this point the event queue is empty, and the Voronoi diagram of p_i, p_j, and p_k is three open regions. However, note that the beachfront is not empty after handling this circle event. In particular, the arcs γ and ω remain. The edge separating V_i and V_j is a half-infinite edge. Generally, what remains of the beachfront after all of the events have been handled are the remaining half-infinite edges. As a post-processing step of sorts, we will often determine a reasonable bounding box for the diagram, so we can determine the location in $\mathbb{R}^2$ of any (half-infinite) edge endpoints going to the vertex at infinity.



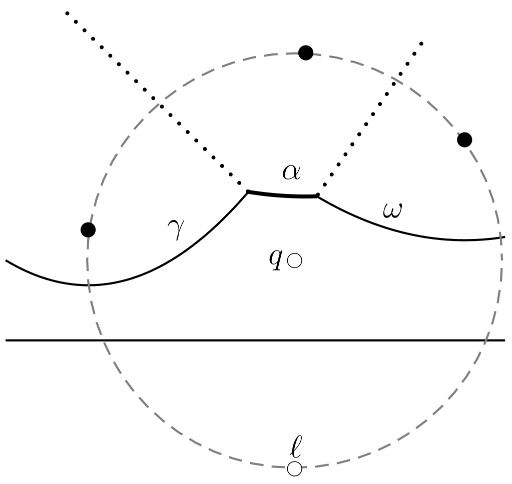
(a) Handling the first site event



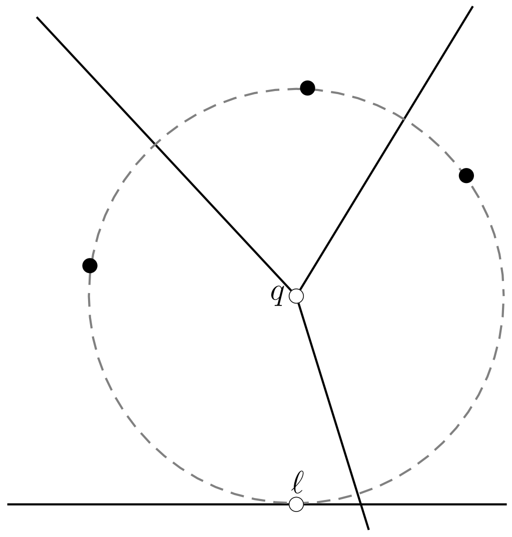
(b) Adding an arc, ω, to the beachfront



(c) Adding a circle event at ℓ defined by circle $C(p_i, p_j, p_k)$, centered at q



(d) Approaching the circle event: α will be removed; dotted edges will meet at q



(e) Handling circle event: finalizing Voronoi edges and adding a Voronoi vertex at q

Figure 2.3: Constructing the Voronoi Diagram of three sites

2.2.3 Algorithm Pseudocode

FORTUNE-VORONOI and its supporting procedures, HANDLESITEEVENT and HANDLECIRCLEEVENT, provide a pseudocode implementation of Fortune's algorithm. Other supporting procedures, such as NEXTEVENT, REMOVECIRCLE, SPLITARCINSERT, GETLEFTARC, GETRIGHTARC, DETECTCIRCLEEVENT, ADDEDGE, among others, will allow us to gloss over the details of the code and to focus on the intention of the procedure at hand. We will use a doubly connected edge list to store information about the Voronoi vertices and edges. A common structure used to maintain the beachfront, $\beta(P)$, is some flavor of a balanced binary search tree; Berg et al. [3] uses a Red-Black Tree, and we assume the same. As we saw in section 2.2.1, the event queue, Q, will be a priority queue, which can be implemented in a multitude of ways.

Algorithm 2 FORTUNE-VORONOI

```
Require: Set of sites, $P$, which are in general position
 1: procedure FORTUNE-VORONOI($P$)                              ▷ Construct Vor($P$)
 2:     $Q \leftarrow P$                 ▷ Initialize event queue, $Q$, with set of sites, $P$
 3:     $E \leftarrow \{\}$                                      ▷ Initialize (empty) DCEL
 4:     $\beta \leftarrow \{\}$                                  ▷ Initialize (empty) beachfront
 5:     while $Q$ is not empty do
 6:         $e \leftarrow$ NEXTEVENT($Q$)
 7:         if $e$ is a site event then
 8:             HANDLESITEEVENT($e$)                              ▷ e is a site $p$
 9:         else
10:             HANDLECIRCLEEVENT($e$)                            ▷ $e$ is a circle, $c$
11:         end if
12:     end while
13: end procedure
```

FORTUNE-VORONOI is the conductor of this symphony. In lines 2, 3, and 4, we initialize our major structures. β is an empty RB-Tree, and E is an empty list of directed half edges. After this step, Q is a priority queue populated with the site events. The main goal of the FORTUNE-VORONOI procedure is to handle all of the events in the event queue. As we will see in HANDLESITEEVENT and HANDLECIRCLEEVENT, calls to DETECTCIRCLEEVENT, REMOVECIRCLEFOR and REMOVECIRCLE will add and remove circle events from the event queue as necessary. As long as there are events in Q, we assign the variable e to the result of calling NEXTEVENT with Q as an argument. The NEXTEVENT function returns the event with the highest priority– that is to say, with the maximum y-coordinate– from Q.

If e is a site event, we enter the HANDLESITEEVENT procedure, which takes site event, p, as its only argument; though we retain access to the major structures of Vor(P), namely: Q, β, and E. In line 2, we check to see whether or not this is the first arc in the beachfront. If it is, we call the INSERT function with the beachfront and the site, which inserts an arc associated with p into β. Otherwise, let α be the arc in the beachfront directly above p. If there was a circle event associated with

Algorithm 3 HandleSiteEvent

```
 1: procedure HandleSiteEvent($p$)
 2:     if $\beta$ is empty then
 3:         Insert($\beta$, $p$)
 4:         return
 5:     else
 6:         $\alpha \leftarrow$ GetArcAbove($\beta$, $p$)            ▷ Get the arc above this site
 7:         if HasCircle($\alpha$) then            ▷ Remove false alarm if necessary
 8:             RemoveCircle($Q$, $\alpha$)
 9:         end if
10:         new_arc $\leftarrow$ SplitArcInsert($\beta$, $\alpha$, $p$)
11:         left_arc $\leftarrow$GetLeftArc($\beta$, new_arc)
12:         right_arc $\leftarrow$GetRightArc($\beta$, new_arc)
13:         AddEdge($E$, left_arc, new_arc)            ▷ Add new edges
14:         AddEdge($E$, new_arc, right_arc)
15:         DetectCircleEvent($Q$, $\beta$, left_arc)            ▷ Add new circle events
16:         DetectCircleEvent($Q$, $\beta$, right_arc)
17:     end if
18:     TidyUp($E$, $\beta$)      ▷ Create cells and handle what remains of the beachfront
19:     return $E$
20: end procedure
```

this arc already, then we need to remove it, since this circle event has become a false alarm.

Line 10 is a dense one. In this line, we make a call to SplitArcInsert, which creates and returns an arc, *new_arc*, associated with p and also splits the arc above p, α, into two arcs (as in figure 2.2a where α splits into α' and α''). In the RB-Tree, the effect of this is the replacement of a leaf node (representing the arc α) with a subtree whose three leaves store α', *new_arc*, and α''. The internal (non-leaf) nodes of this subtree stores the breakpoints of the left and right child arcs. [probably should make a figure of this]. Since we are using an RB-tree, some re-balancing might be necessary (though outside the scope of this thesis), but we will assume that SplitArcInsert takes care of that.

The remainder of HandleSiteEvent involves making the proper updates to E and Q. In lines 11 and 12 we get the arcs to the left and right of *new_arc*, which constitutes searches for the predecessor and successor leaf nodes with respect to *new_arc* in β. Note that these arcs will be α' and α'', respectively. As we saw in the previous sections, the addition of an arc to the beachfront provokes the beginning of two new edges, specifically, the left edge traced out by the intersection of *left_arc* and *new_arc*; and the right edge traced out by the intersection of *new_arc* and *right_arc*. In lines 13 and 14 we add these edges to E. Finally, we evaluate whether or not there are new circle events which need to be added to Q. A call to DetectCircleEvent with arguments *queue*, $beachfront$, and *arc* determines whether or not there is a circle event with the triple of arcs (GetLeftArc(*arc*), *arc*, GetRightArc(*arc*)). If there is, it creates the event, adds it to *queue*, and associates it with *arc*. Recall that there isn't a valid circle event associated with *new_arc*, since the arcs to its left and right are associated with the same site. So we only need to check for two possible circle events: the event where the middle arc is *left_arc*, and the event where the middle arc is *right_arc*.

Algorithm 4 HANDLECIRCLEEVENT

```
 1: procedure HandleCircleEvent($c$)
 2:     $\alpha$ ←GetArcAbove($\beta$, $c$)
 3:     left_arc ←GetLeftArc($\beta$, $\alpha$)
 4:     right_arc ←GetRightArc($\beta$, $\alpha$)
 5:     left_edge ← GetEdge($E$, left_arc, $\alpha$)
 6:     right_edge ← GetEdge($E$, $\alpha$, right_arc)
 7:     if HasCircleFor($left$, $\alpha$) then
 8:         RemoveCircle($Q$, $left$)
 9:     end if
10:     if HasCircleFor($right$, $\alpha$) then
11:         RemoveCircle($Q$, $right$)
12:     end if
13:     RemoveArc($\beta$, $\alpha$)
14:     new_vert ← AddVertex($E$, c.center)          ▷ Insert a new vertex into $E$
15:     new_edge ← AddEdge($E$, left_arc, right_arc)
16:     SetEdgeSrc(new_edge, new_vert)
17:     SetEdgeDest(left_edge, new_vert)             ▷ Update edges of removed arc
18:     SetEdgeDest(right_edge, new_vert)
19:     DetectCircleEvent($Q$, $\beta$, left_arc)
20:     DetectCircleEvent($Q$, $\beta$, right_arc)
21: end procedure
```

If e is a circle event, we handle it by calling HANDLECIRCLEEVENT, which takes a circle, c, as its argument. There is an arc associated with this circle event: it is the arc which will be removed. In line 2 we retrieve this arc (which is the arc located directly above c's lowest point ℓ) and assign it to a variable α. Since we are removing α from the beachfront, the circle events on associated with its adjacent arcs (*left_arc* and *right_arc*) are invalid if they involve α. In lines 7 through 12 we perform this check and remove the offending circle events from Q. We also make the crucial change to the beachfront in the next line: removing α from β.

Similar to the HANDLESITEEVENT procedure, the rest of HANDLECIRCLEEVENT involves updating Q and E to reflect the circle event and the corresponding change in the beachfront. We know that a handled circle event signifies the discovery of a Voronoi vertex at the center of that circle. In line 14, we make a call to the helper function ADDVERTEX save the inserted vertex as *new_vert*. ADDVERTEX takes as its arguments an edge list and a location; it creates a new vertex associated with that location, adds it to the edge list, and returns that vertex. Also note that the removal of α from β yields a new pair of breakpoints: *left_arc* and *right_arc* are now adjacent to each other, and as such their intersection represents the formation of a new edge. We add this edge in line 15. Further, we know that *new_vert* is the source of *new_edge*, so we update *new_edge*'s source vertex in line 16. Recall that *new_vert* is also the point where the left and right edges traced out by the breakpoints of the removed arc meet. In lines 17 and 18, we update these edges with this new information by setting the destinations of *left_edge* and *right_edge* to *new_arc*.Finally, we need to see if the removal of α triggered any new circle events, so in the final two lines of the procedure we call DetectCircleEvent on both of the arcs previously adjacent to α.

Though there are no more events left in Q, we might not have removed all of the arcs in β. The remaining leaves in β are the arcs which were not removed by any circle event. The internal nodes of β represent the breakpoints of its corresponding child arcs, which we know to be Voronoi edges. However, since these arcs were never removed from the beachfront via a circle event, they might not have their source or destination vertices set. We take the edges defined by the internal nodes of the beachfront to be the half infinite edges in the diagram. A call to TidyUp assigns the endpoints of these edges to be the vertex at infinity. It also "audits" E, connecting half-edges into cells describing a Voronoi region by setting the *next* attribute of a half edge. Often, TidyUp will also calculate a bounding box for Vor(P) and find the intersections of the half-infinite edges with the bounding box.

To summarize, Fortune's algorithm imagines a sweep line which scans the plane, updating information about the Voronoi diagram Vor(P) whenever the line intersects with a point of interest. In particular, these points of interest are site events (i.e points in P) and circle events, which emerge during the algorithms progression. We maintain an ordering of these points of interest in a priority queue. When we handle a site event or a circle event, we update three structures: the doubly-connected edge list storing information about the edges and vertices of Vor(P), the priority queue itself, in cases where events need to be added, and the beachfront. The beachfront is a sequence of parabolic segments with information about the in-progress Voronoi regions. It is an ordered dictionary of sorts; Berg et al., among many others, uses a Red-Black tree to represent this structure. The algorithm terminates when there are no more events in the priority queue. At this point, a common step is to find an appropriate bounding box for the diagram and to tidy up the half-infinite edges[5] in Vor(P) so that the diagram has an embedding in $\mathbb{R}^2$.[6]

[5]By "tidy up" we propose calculating the intersection of the half infinite edges with the bounding box.

[6]As a side note, this was an important consideration in my own implementation of Fortune's algorithm [17]. The results after the tidying up process consisted of sets of vertices and edges (each associated with locations in the plane), which were then used to produce all of the diagrams you see in this thesis.

Chapter 3

Constructing the Delaunay Triangulation

Guibas and Stolfi's "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams" [15], presents a detailed and comprehensive presentation of the Voronoi and Delaunay diagrams. They offer a divide-and-conquer approach to the construction of Voronoi or Delaunay diagrams whose time and space complexity is consistent with the bounds known to be optimal in the worst case.[1] For the purposes of computing the Delaunay triangulation (and implicitly, the Voronoi diagram) Guibas and Stolfi introduce two geometric predicates which enforce the "interesting" properties; these primitives ensure that the Delaunayhood of the graph.

A significant byproduct of Guibas and Stolfi's contribution is their *quad-edge* data structure. It is praised for its its elegance, since it represents both the primal and the dual (for our purposes, the Delaunay Triangulation and the Voronoi diagram) simultaneously. This allows us to hop from one to another in constant time by reversing the conventions around the representation of faces and vertices.

3.1 Overview: A Divide-and-Conquer Approach

Like other divide-and-conquer algorithms, Guibas and Stolfi's approach to constructing a Delaunay Triangulation relies on repeated partitioning of the problem's data until each partition conforms to a trivial (or base) case.

Given a set of sites and a desire to compute the Delaunay triangulation of that set, we begin. Though "divide-and-conquer" might lead you to believe that this approach consists of two stages, Guibas and Stolfi proceeds in three stages. First, we have the Split stage, in which we divide the large problem of computing the Delaunay of P into two smaller subproblems. The algorithm is implemented recursively, and we will discuss how the algorithm leverages recursion and handles these base cases shortly. The second stage involves two recursive steps: we compute the Delaunay triangulation of each subproblem. Finally, we proceed to the Merge phase, which

[1]Recall that the Voronoi diagram of n sites can be computed in $\mathcal{O}(n \log n)$ time and occupying $\mathcal{O}(n)$ space, and these bounds are TIGHT!

carefully combines the triangulations of the partitions into a final triangulation on all n sites.

GS-Delaunay provides a pseudocode implementation of the divide-and-conquer algorithm presented in Guibas and Stolfi's paper. The corresponding procedure for the dual is described in Shamos and Preparata's text, [21, p. 206]. Because we are using Guibas and Stolfis quad-edge data structure, the operations around creating and manipulating the edge list are more complex than they would be if we were using a doubly connected edge list. For the most part, this added complexity will be beyond the scope of our discussion. Finally, a note: GS-Delaunay computes the Delaunay triangulation on a set of sites; however, it doesn't return the entire edge structure. When we return D at any point in GS-Delaunay, we are returning two "handles" into the mesh. More precisely, we return the convex hull edge in the clockwise direction whose source is the rightmost vertex and the convex hull edge in the counterclockwise direction whose source is the leftmost vertex.

Algorithm 5 GS-Delaunay

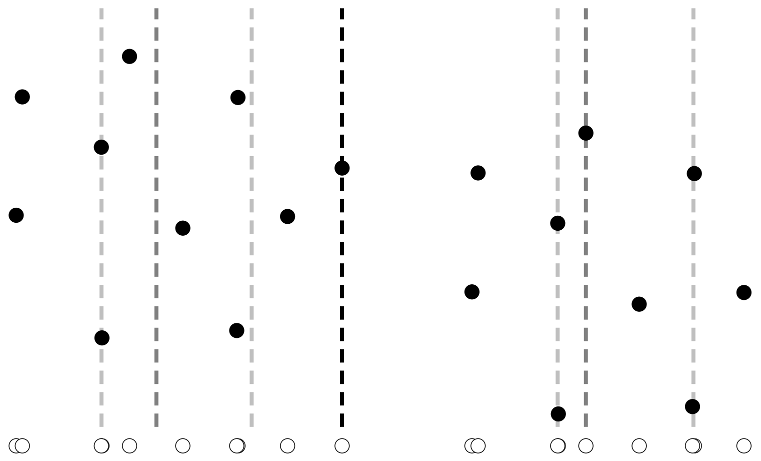
```
Require: Set of sites, $P$, which are in *general position*
 1: procedure GS-Delaunay($P$)                          ▷ Construct $D_P$
 2:     if $|P|$ is 2 then                              ▷ First Base Case
 3:         return Delaunay-2($P$)
 4:     else if $|P|$ is 3 then                         ▷ Second Base Case
 5:         return Delaunay-3($P$)
 6:     else                                            ▷ Recursive Case
 7:         $q \leftarrow$Median-Lex($P$)
 8:         $L, R, \leftarrow$ Split($P$, $q.x$)        ▷ Partition $P$ according to $q$
 9:         $D_L \leftarrow$ GS-Delaunay($L$)
            $D_R \leftarrow$ GS-Delaunay($R$)
10:         $D_P \leftarrow$ Merge($D_L$, $D_R$)        ▷ Merge sub-diagrams
11:         return $D_P$
12:     end if
13: end procedure
```

We assume three things of P: that the x-coordinate of each site in P is unique; that the sites are in *general position* and that not all sites are collinear, a special case we covered in section 1.5.2. Given P, GS-Delaunay computes Del(P) by splitting the problem into subproblems and then merging the results pairwise. Lines 3 and 5 handle the base cases of the algorithm, which we will discuss in-depth in section 3.2.

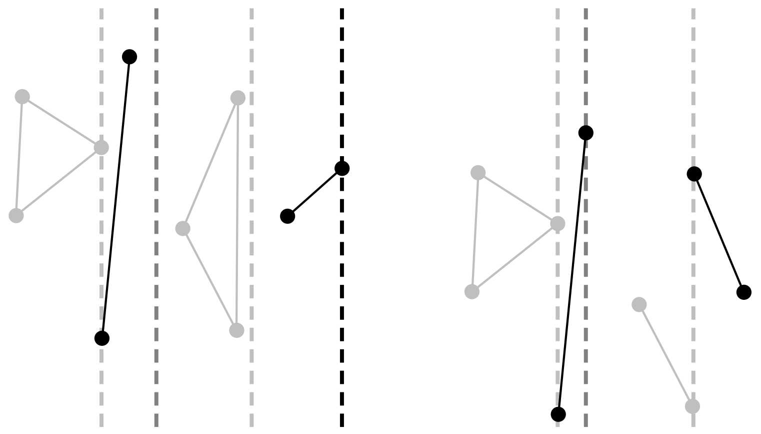
The final chunk of pseudocode, lines 7 to 11, handle the recursive case. When there are more than three sites in P, we need to break the problem down further. We call Median-Lex, which returns the point $q \in P$ whose x-value is the median. In line 8, we pass this value— along with P— to another helper function, Split. Split returns two sets of sites, L and R (where $L = \{p \mid p.x \leq q.x\}$). Then, we make the calls to GS-Delaunay on both L and R. Finally, in line 10, we merge the results with a call to Merge and return the result.

3.1.1 Split!

We seek a vertical line by which we will recursively partition the set of sites. This yields a left site set and a right site set, L and R, respectively. We sort the sites by x-coordinate and choose the median value. Figure 3.1a illustrates how the recursive splitting of P plays out. Note that in this example, some of the sites have very similar x-coordinates. In cases where we can't assume x-coordinates are unique, we may break the "tie", so to speak, by comparing the y-coordinates of the sites. This is a called a lexicographic ordering of the coordinates. Figure 3.1b illustrates how subproblems of size two or three are triangulated.



(a) Choosing the partitions of P. The filled circles are the sites of P, and the unfilled circles are the sites projected onto the x-axis. The darkest vertical line is the first partition, separating P into L and R. The two gray lines are the partitions of the resulting subproblems, and the four lightest lines the medians separating the four subproblems generated from the gray lines.



(b) Solving (triangulating) the subproblems.

Figure 3.1: Dividing P into subproblems we know how to solve

Note that once the size of the set of sites we are operating on reaches two or three, we do not partition further. At this stage, we consider sets of sites with size two or three to be subproblems which conform to one of two base cases. When either of these is true, we immediately recognize that the Delaunay triangulation is trivial: in the case of two sites, it is a line segment connecting the two sites; in the case of three sites, it is the triple of segments connecting the sites into a triangle.

To summarize, we begin with a set of sites, P, where $|P| = n$. We sort the sites by x-coordinate, and split the set of sites into left and right halves according to the median x-value. We continue until the subproblem sizes are trivial to solve, at which point we triangulate each subproblem. At this point, we have a collection of valid Delaunay diagrams on small partitions of P; our next task will be to understand how they can be combined into the final triangulation $\text{Del}(P)$.

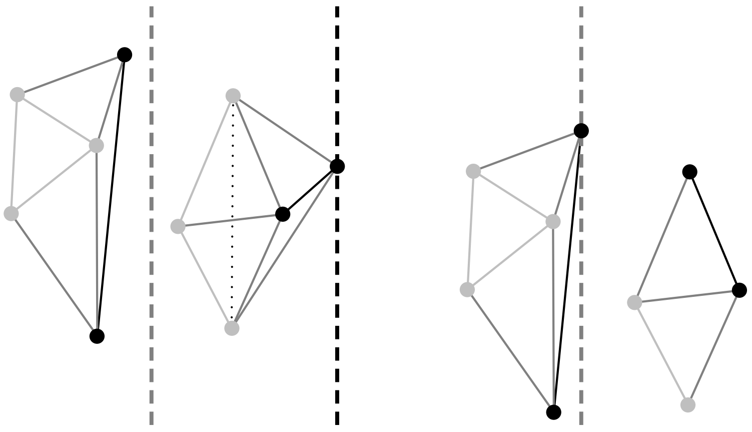
3.1.2 Merge!

The merge step of Guibas and Stolfi's algorithm is one of the more computationally expensive portions of their approach, and we'll soon understand why. This stage knits two triangulations together to compute the Delaunay triangulation of their combined point sets.

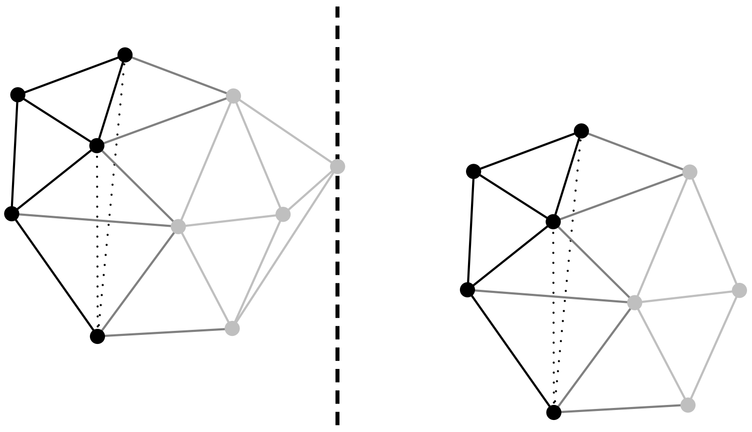
Let L and R be partitions of P, as described in section 3.1.1. We are given $\text{Del}(L)$ and $\text{Del}(R)$, Delaunay triangulations of the point-sets L and R, respectively. We wish to compute $\text{Del}(L \cup R)$. For brevity, we refer to $\text{Del}(L)$ and $\text{Del}(R)$ equivalently as D_L and D_R.

To provide you, dear Reader, with a visual intuition of what the merge step entails, I have included figure 3.2. Figure 3.2a illustrates the first iteration of the merge step; it knits together the eight triangulations in figure 3.1b into four triangulations. Figure 3.2b shows results of the second pass: it combines the diagrams to the left of the dashed vertical line into a single diagram, and combines the pair to the right of the line in a similar fashion. The dotted lines in figures 3.1a and 3.1b are edges which had to be removed in order to maintain the Delaunayhood of the merged diagram.

In section 3.3, we walk through the particulars of constructing the merged diagram; we discuss MERGE, a pseudocode implementation of the process. As an example, we step through the process of knitting together the last pair of triangulations D_L and D_R, shown in figure 3.2b. When we are finished, we will have formed D_P, the Delaunay triangulation on the original point-set, P.



(a) Merging the adjacent subproblems. Note the gray dotted line segment in the second pair from the left. We needed to remove that edge in order to maintain the Delaunay property of the merged result of the pair.



(b) Merging the adjacent pairs from figure 3.2a. Again, the dotted line segments are removed to maintain the Delaunay propertry.

Figure 3.2: Merging subproblems pairwise.

3.1.3 Geometric Predicates

Before we dive into the details of the divide-and-conquer approach, we introduce a couple geometric primitives. When we introduced the Delaunay triangulation in section 1.4 we defined the *empty circle property*, or condition. We can systematically determine whether or not the embedding of a graph in $\mathbb{R}^2$ is *globally Delaunay* by checking that each face is *locally Delaunay*, meaning that the circumcircle passing through the vertices of the face is empty. Guibas and Stolfi refer to this as the InCircle test.

The InCircle test takes as its arguments four sites, A, B, C, and D, and envisions a counterclockwise oriented circle ABC and a lonely point D. It is defined by Guibas and Stolfi as follows: "InCircle(A, B, C, D) is defined to be true if and only if point D is interior to the region of the plane that is bounded by the oriented circle ABC and lies to the left of it," [15, p. 106].

If we wish to think about it in terms of the coordinates of each site A, B, C, and D, the InCircle predicate is equivalent to:

$$\textsc{InCircle}(A,B,C,D) = \begin{vmatrix} x_A & y_A & x_A^2 + y_A^2 & 1 \\ x_B & y_B & x_B^2 + y_B^2 & 1 \\ x_C & y_C & x_C^2 + y_C^2 & 1 \\ x_D & y_D & x_D^2 + y_D^2 & 1 \end{vmatrix} > 0.$$

Recognize that InCircle is precisely the circle test we performed in section 1.4 and saw in figure 1.6. We rely on the InCircle test heavily during the merge procedure in two ways. When the InCircle test is applied to an existing face ΔABC in the graph, and an appropriate vertex D, passing the test indicates that the embedding of that particular facet will not be *locally Delaunay*, so we'll need to change something before proceeding. When we attempt to triangulate the region between two adjacent, non-overlapping triangulations, the InCircle test helps us compose the triangles which are added to the result.

The second geometric predicate used by Guibas and Stolfi is CCW. It is, in a sense, an orientation test. We say that three vertices A, B, and C, are ordered in a counterclockwise fashion if the result of $\mathrm{CCW}(A,B,C) > 0$. This predicate can be implemented as:

$$\mathrm{CCW}(A,B,C) = \begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} > 0.$$

This test can also be thought of as an indicator of whether or not the point C lies to the left of the line obtained from the edge going from A to B. If $\mathrm{CCW}(A,B,C) > 0$, it does, if $\mathrm{CCW}(A,B,C) = 0$, the three points are collinear. To be exhaustive, if $\mathrm{CCW}(A,B,C) < 0$, then C lies on the right.

3.2 Handling the Base Cases

Like all recursive procedures which intend to end, GS-Delaunay has base cases. Delaunay-2 and Delaunay-3 provide the pseudocode for each case. As we saw in section 3.1.1, the triangulation of two sites is a line segment, and the triangulation of three sites is a triangle.

Algorithm 6 Delaunay-2

```
Require: Set of sites, $P$, which are in general position
 1: procedure Delaunay-2($P$)                                ▷ Construct $D_P$
 2:     $p_1$, $p_2$ ← Sort-Lex(P)
 3:     $e$ ←MakeEdge()
 4:     $e.org$, $e.dest$ ← $p_1$, $p_2$
 5:     return $e$, $e.twin$
 6: end procedure
```

To determine the direction and orientation of the directed edge corresponding to this line segment, we first make a call to Sort-Lex(P) , which returns the set of sites sorted according to x-coordinate. In line 3, assign e to the result of MakeEdge. MakeEdge creates a generic, unconnected quad-edge. Now that we have made a single edge, e we return the pair e, *e.twin*. Since the diagram consists of just this single edge, e, *e.twin* are the oppositely oriented leftmost and rightmost edges we should return.

Algorithm 7 Delaunay-3

```
Require: Set of sites, $P$, which are in general position
 1: procedure Delaunay-3($P$)                                ▷ Construct $D_P$
 2:     $p_1$, $p_2$, $p_3$ ← Sort-Lex(P)
 3:     $e1$ ←MakeEdge()
 4:     $e2$ ←MakeEdge()
 5:     Splice($e1.twin$, $e2$)
 6:     $e1.org$ ← $p_1$
 7:     $e1.dest$ ← $p_2$; $b1.org$← $p_2$
 8:     $e2.dest$ ← $p_3$
 9:     $e3$ ←Connect(e2, e1)
10:     if CCW($p_1$, $p_2$, $p_3$) then                       ▷ Make a triangle
11:         return $e1$, $e2.twin$
12:     else if CCW($p_1$, $p_3$, $p_2$) then
13:         return $e3$, $e3.twin$
14:     else                                 ▷ Case where the sites are collinear
15:         return $e1$, $e2.twin$
16:     end if
17: end procedure
```

A slightly more complex base case is outlined in Delaunay-3. Our goal in this case is to try to form a triangle (composed of edges) with the correct orientation. As in the previous case, our first order of business is to sort the sites by x-coordinate and assign them to p_1, p_2, and p_3, respectively. We make two generic edges, $e1$ and $e2$.

The call to Splice on line 5 is something of a mysterious one. Splice takes as its arguments two a and b, and operates on the rings of edges at the origins of a and b. It also operates on rings of edges whose origins are the left faces of a and b. If a and b have different origins, Splice has the effect of combining and ordering them appropriately. If the origins are the same, Splice has the effect of splitting the ring of edges around the origin. It then does the same thing for the rings (of edges) whose origins are *a.left* or *b.left*.

At this point (line 5), $e1$ and $e2$ are totally disconnected from the mesh, so the effect of Splice is that it sets $e1.twin$ to be in the same ring as $e2$. This is a good place to pause and recognize one of the ways in which the quad-edge structure deviates from the doubly connected edge list. In this particular case, if we were working with a doubly connected edge list, we could accomplish line 5 by setting $e1.twin.next$ to $e2$ and then by setting $e2.next$ to $e1.twin$. However, since we are working with a quad-edge structure, a bit more care has to be taken to ensure that these operations propagate appropriately to the dual. This is the last we'll see of Splice, as it will only be used implicitly (via the Connect procedure) for the remainder of our discussion.

After updating the source and destination vertices of the edges, we perform the CCW test to determine the orientation of the triangle. This also determines the correct pair of edges for us to return. We create the final edge of the triangle with the Connect procedure. Connect takes two edges, a and b, and returns an edge, e, whose source is $a.dest$ and whose destination is $b.org$. At this point, the left faces of a, b, and e are the same.

With the base cases handled and a vision of what remains to be accomplished, we consider ourselves equipped to move on to the final task at hand: joining two valid triangulations.

3.3 Merging

We revisit the topic of merging, asking "How are D_L and D_R connected?" We can make significant headway by breaking this question down further: "Will every edge in D_L and D_R be in D_P?" and "How do we start finding the edges which connect D_L and D_R?" In an effort to demystify the merge step somewhat, we apply the process to the particular D_L and D_R[2] we'd have otherwise abandoned in section 3.1.2.

To understand how these questions correspond to the merge process, we break up the process of merging into two steps. Our first order of business is to find an edge which spans D_L and D_R (i.e, has one endpoint in each triangulation) and will certainly be in D_P. We'll refer to edges of this type as *cross-edges.* Then, we will use that cross-edge as a foothold, allowing us to enter the *rising bubble* stage: the

[2]Last seen in figure 3.2.

tasks of pruning out the invalid edges in D_L and D_R, and creating the appropriate cross-edges.

3.3.1 The First Cross-Edge

At the risk of being pedantic, note that deep as we are into the task of merging D_L and D_R to form D_P, we cannot lose sight of the fact that merging D_L and D_R is the last step in the process of triangulating the point set P. And by that, I mean to encourage you to recall that the triangulation of P is bounded by the the convex hull of P *conv*(P).

This gives us insight into the task of determining the "first" cross-edge to span D_L and D_R, accomplished in line 4 of MERGE. One thing we can be sure of is the fact that the edges composing the boundary of *conv*(P) will be in D_P. Consider the *lower common tangent* of D_L and D_R, shown in figure 3.3 as *lct*. The lower common tangent is the lowest line which is both tangent to the polygons formed by the boundary of the convex hull and does not intersect either polygon. Since D_L and D_R are Delaunay triangulations on disjoint subsets of P and since $L \cup R = P$, an edge whose endpoints correspond to the points of tangency of *lct*, also lies on the boundary of *conv*(P). We'll refer to this edge as b.

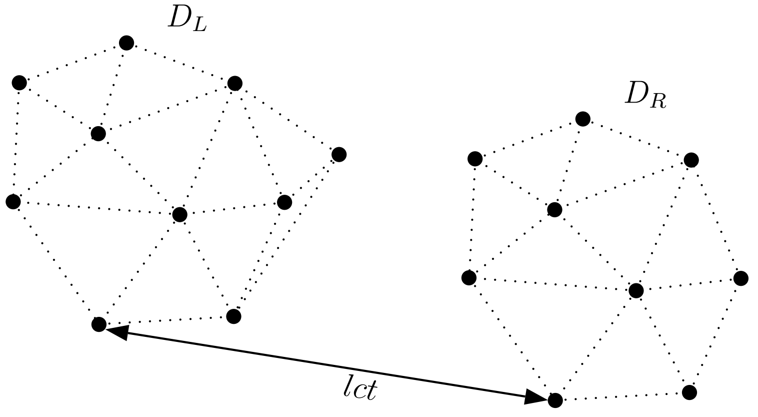


Figure 3.3: Lower Common Tangent, *lct*, of D_L and D_R

LOWER-COMMON-TANGENT provides the pseudocode for finding the lower common tangent of D_L and D_R. We are given *ldi* and *rdi*. Originally, *ldi* is the convex hull edge in the clockwise direction whose origin is the rightmost vertex in D_L. Correspondingly, *rdi* is the convex hull edge in the counterclockwise direction whose source is the leftmost vertex in D_R. Conceptually, our goal is to walk "down" the respective hulls of the adjacent diagrams until we find their lower common tangent. This is done with a series of CCW tests.

The first CCW test on line 3 is equivalent to asking whether or not the source of *rdi* is left of *ldi*. If *rdi.org* is to the left of *ldi*, then we need to take a step (in the clockwise direction) along the hull of L. This is accomplished in line 4 where we update *ldi*'s value to *ldi.l_next*. *ldi.l_next* points to the next edge which has the same

Algorithm 8 Lower-Common-Tangent

```
procedure Lower-Common-Tangent(ldi, rdi)
    loop
        if CCW(rdi.org, ldi.org, ldi.dest) then
            ldi ← ldi.l_next
        else if CCW(ldi.org, rdi.dest, rdi.org) then
            rdi ← rdi.r_next
        else
            b ← Connect(rdi.twin, ldi)
            return b
        end if
    end loop
end procedure
```

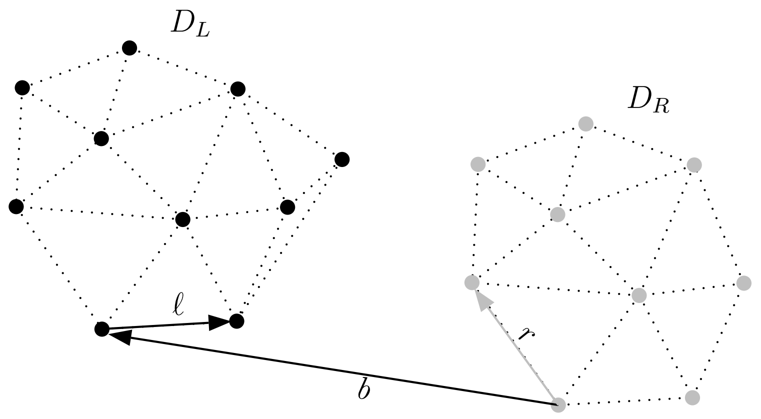
left face as *ldi*. Since the left face of *ldi* is the region exterior to the convex hull, *ldi.l_next* is the next edge on the hull. If the value of this CCW is not positive, we ask whether or not *ldi.org* is right of *rdi*. If it is, then we advance *rdi* to the next edge—in the counterclockwise direction—on the hull of R. Similar to our process with *ldi*, we make the corresponding note that *rdi.r_next* is the next edge with the same right face as *rdi*.

We loop in this fashion, advancing *ldi* and *rdi* in the appropriate manner until both CCW tests fail. When both tests fail, the line along *ldi* will not intersect with D_R and the corresponding line for *rdi* will not intersect with D_L. So we know that the line whose endpoints are *rdi.org* and *ldi.org* is tangent to both triangulations. We create this edge, b, and return it in line 9. As evidenced by the pseudocode above, we also note that the lower common tangent can be found in linear time in the combined sizes of the hulls of L and R.

3.3.2 The Rising Bubble

Satisfied that we have found b, the first cross-edge, we enter the *rising bubble stage.* The name reflects the concepts behind the actions we're about to perform: the rising bubble exists as the series of circle tests performed (roughly) along the vertical bisector separating D_L and D_R.

We introduce two more edges which we will use throughout this portion of the algorithm. ℓ is the next edge in D_L to be "encountered" by the rising bubble. Similarly, r is the next edge in D_R to be encountered by the bubble. As illustrated in figure 3.4a, at the outset of this procedure, we set ℓ to be *b.twin.ccw()* where *ccw()* is conveniently defined to be a function which returns the next edge in the counterclockwise direction out of the vertex *b.dest*. Our preliminary goal is to update ℓ and r (removing edges from either diagram if necessary) until the quadrilateral formed by *b.dest, b.org, r.dest,* ℓ*.dest* is one that we can triangulate without disrupting either D_L or D_R.

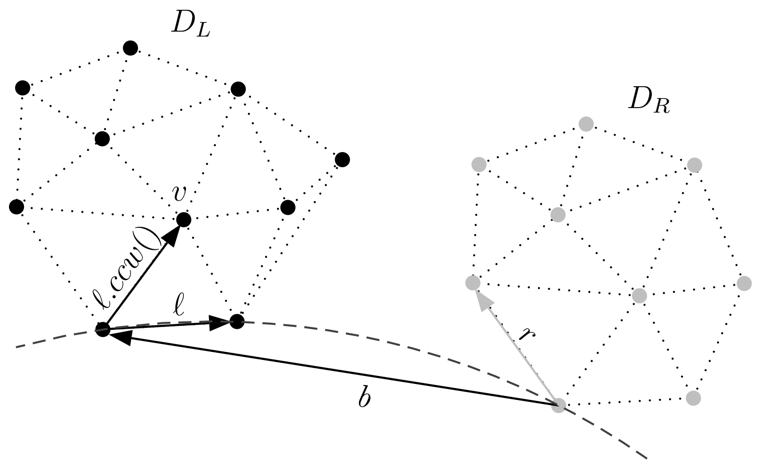


(a) First edge, b, of D_P and ℓ, candidate edge in D_L. Note that ℓ is considered to be valid because CCW(*ℓ.dest, b.dest, b.org*) is true. That is to say, *ℓ.dest, b.dest, b.org* is a counterclockwise ordering.

Dear Reader, if you've developed an intuition about the Delaunay triangulation (or, perhaps you've peeked at the final figure), you might harbor a hunch that the triangle formed by *b.org, b.dest, ℓ.dest* should be in the final triangulation D_P. And while this is the case here, it won't always be this easy: as we hinted at in section 3.1.2, there will be occasions in which the edge encountered by the bubble needs to be removed. To that end, we perform some preliminary checks on the edge be encountered by the rising bubble, ℓ, in our current example. The first thing we check is whether or not ℓ is a valid edge.

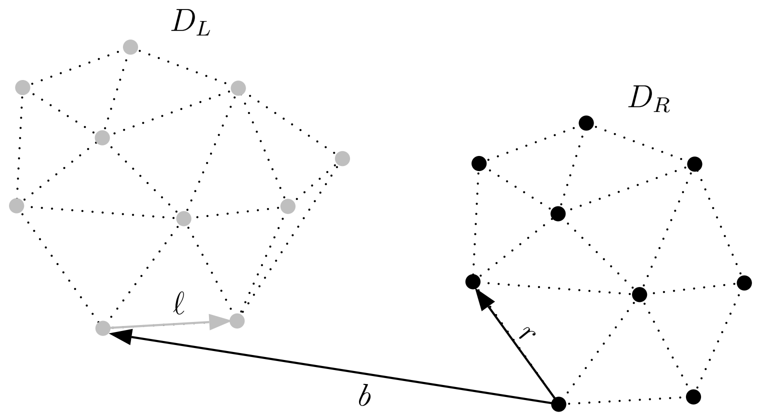
What does it mean for an edge to be *valid*? Generally, an edge is valid if the edge is located "above" b. That is to say, if we took the destination vertex of a given edge e along with the source and destination of b, the three vertices— *e.dest, b.dest, b.org*— need to be ordered in a counterclockwise orientation. Figure 3.4b shows that ℓ is a valid edge. To be precise, we check ℓ's validity by ensuring that CCW(*ℓ.dest, b.dest, b.org*) > 0 holds true, and it does.

Since ℓ is valid, we continue to the next step, which involves determining if any edges incident to *ℓ.org* need to be deleted. During this test, we determine whether or not ℓ passes the circle test. Recall from section 1.4 that an edge e passes the circle test if it is the shared border of two triangles and if there is a circle passing through the endpoints of e that doesn't contain any of the vertices of the two neighboring triangles. The non-existence of such a circle precludes e from being in the merged result. This is implemented as an InCircle test. In our case, we consider the two triangles bordering ℓ, namely (*ℓ.org, ℓ.dest, v*) and (*b.org, ℓ.dest, ℓ.org*). Our InCircle test is: InCircle(*b.org, ℓ.dest, ℓ.org, v*). The oriented circle *b.org, ℓ.dest, ℓ.org*, shown as a dashed arc in figure 3.4b, is the "proof" that there is a point-free circle passing through both endpoints of ℓ.



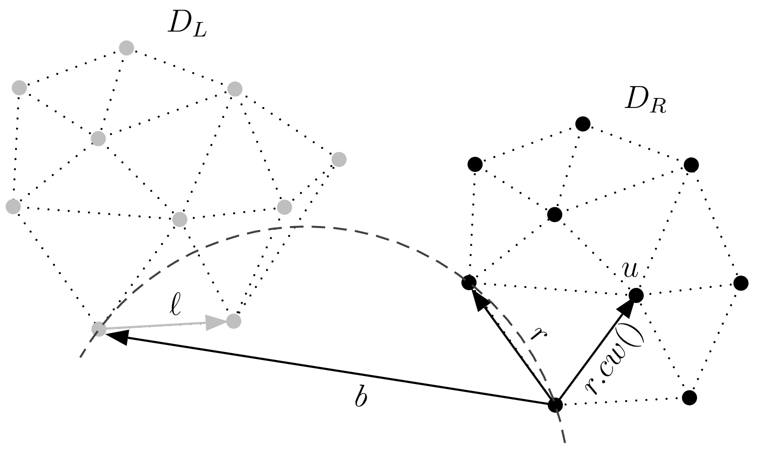
(b) Evaluating whether or not ℓ will be in D_P. The dashed arc is a portion of the *rising bubble* which we say has "witnessed" that v, the destination of *ℓ.ccw()*, is not in the circle defined by *b.org, ℓ.dest, ℓ.org.*

Figures 3.4c and 3.4d illustrate how the symmetric procedure plays out when handling r, the first edge to be encountered by the rising bubble in D_R. Our goal is to make sure that r is valid, and subsequently, to show that u, the destination of the $r.cw()$,[3] *r.dest, b.dest*, and *b.org*.



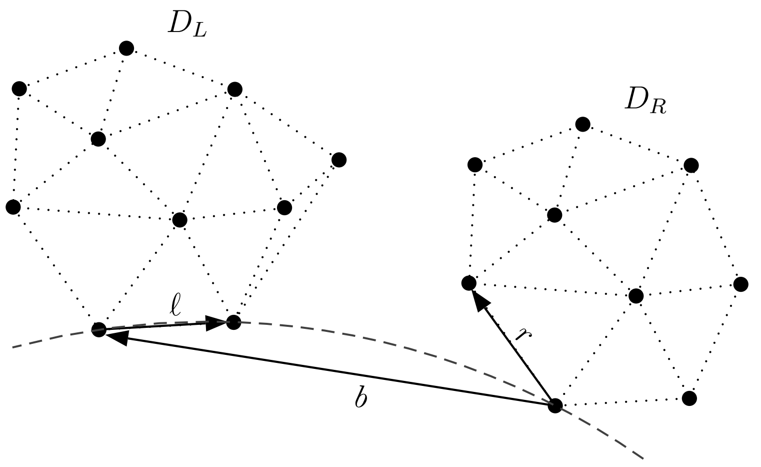
(c) Similarly, now we evaluate whether or not r will be in D_P. Note that it is a valid edge, since *r.dest, b.dest, b.org* is a counterclockwise ordering.

[3]For a given edge, e, recall that *e.ccw()* returns the next edge in the counterclockwise direction going out of *e.org*. Similarly, *e.cw()* returns the next edge in the clockwise direction going out of *e.org*.

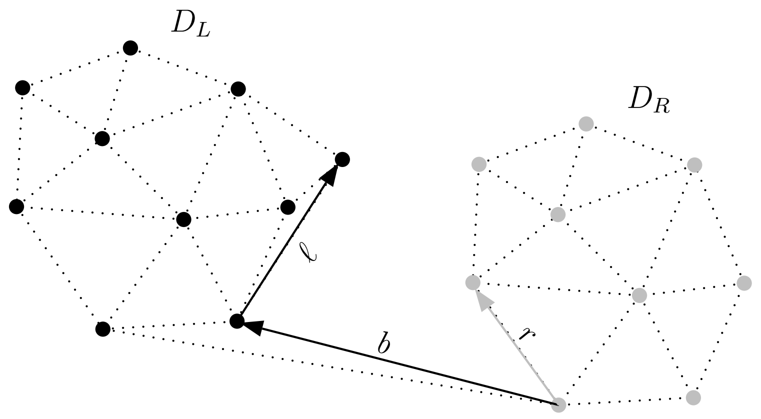


(d) Evaluating whether or not r will be in D_P. The dashed arc is a portion of the "rising bubble" which we say has *witnessed* that u in not in the circle defined by *r.dest, b.dest*, and *b.org*.

At this point, we have four points of interest, namely, the endpoints of b and the destinations of both ℓ and r. We know that ℓ, r, and b will be in D_P, so our goal at this point is to triangulate the quadrilateral formed by these four points. In figure 3.4e, we determine the appropriate cross-edge to insert with the test InCircle(*b.dest, b.org, ℓ.dest, r.dest*), and note that it returns false. This means that the edge connecting *b.org* and *ℓ.dest* passes the circle test, so we insert it, and update b to be that edge precisely. We also update ℓ to be the next edge (in the counterclockwise direction) whose origin is *b.dest* as in figure 3.4f.

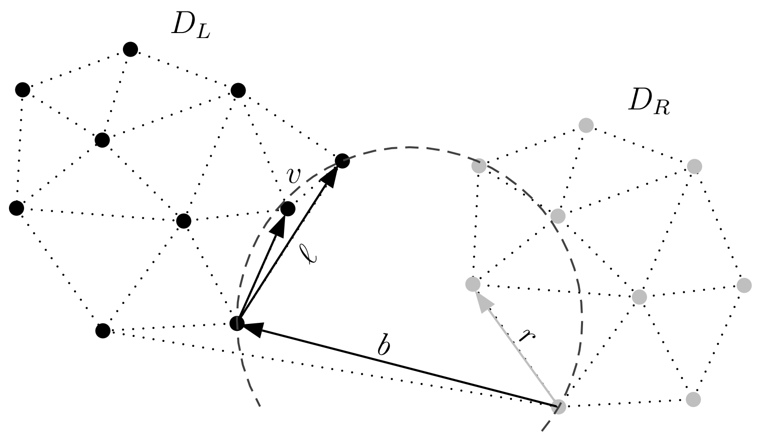


(e) Since ℓ and r are both valid we perform InCircle(*b.dest, b.org, ℓ.dest, r.dest*). Note that *r.dest* is not in the circle formed by (*b.dest, b.org, ℓ.dest*), so InCircle returns false.

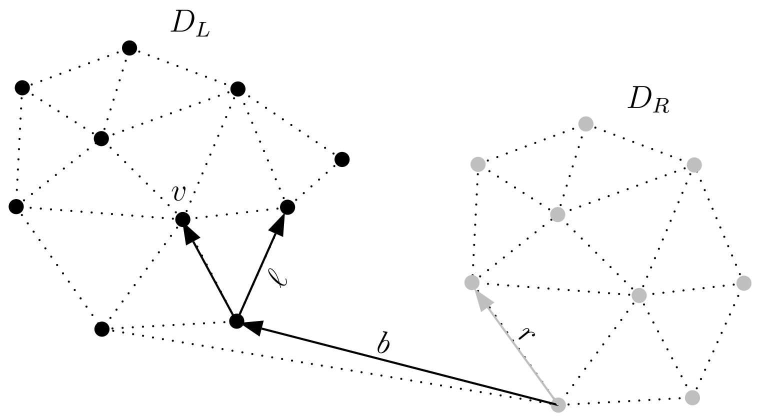


(f) Due to the result of the InCircle test, we update b's destination to *ℓ.dest*. We also update ℓ to *b.ccw()*.

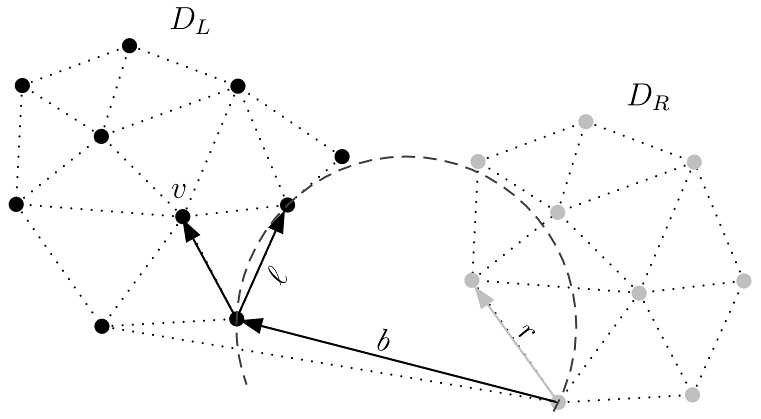
We have now completed all of the steps necessary to add one cross-edge to the merged result. The only thing we've yet to encounter is the case where we need to remove the convex hull edge the bubble is encountering. Luckily for us, the next edge to be encountered by the rising bubble, ℓ, is one such offending edge. Figures 3.4g and 3.4h illustrate the removal, and subsequent updating, of ℓ after we discover that InCircle(*b.org, ℓ.dest, ℓ.org, v*) returns true. We continue to do this— performing the InCircle(*b.org, ℓ.dest, ℓ.org, v*) — and removing ℓ if necessary, until the test returns false. Figure 3.4i demonstrates that the new value of ℓ passes the circle test; we only had to remove one edge this time.



(g) We begin our next assessment, noting that ℓ is a valid edge. Because this is the case, we check to see if v, that is, *ℓ.ccw().dest*, is contained in the circle represented by the dashed lines. It is, so we've shown that ℓ is not Delaunay.

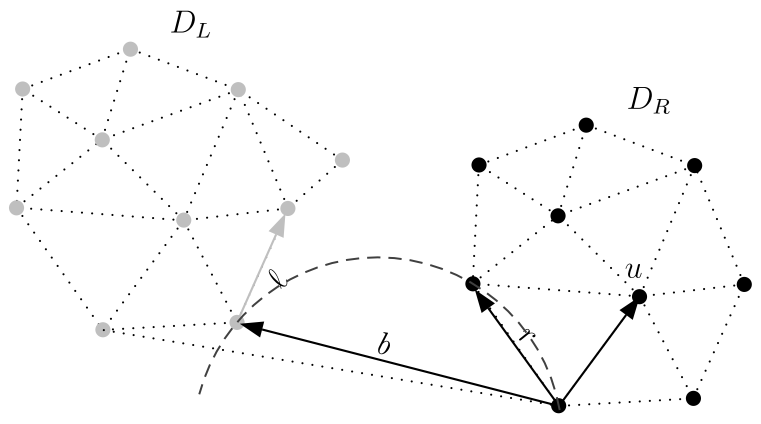


(h) Because v is contained in the circle, we need to remove ℓ and set the new value of ℓ to *ℓ.ccw()*. Note *ℓ.ccw()* is also shown as a directed edge, its destination is v.

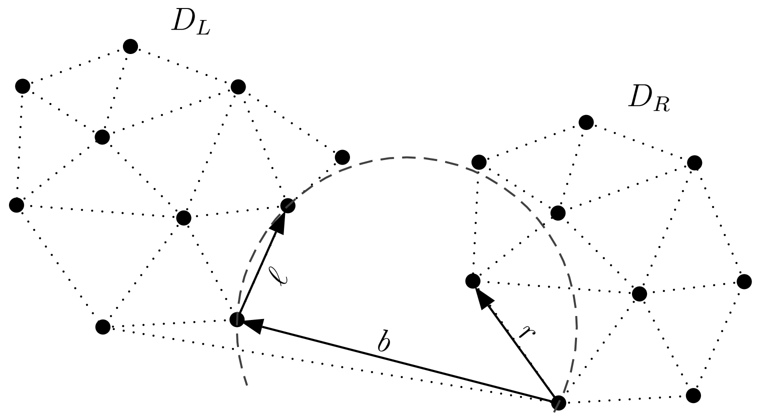


(i) ℓ is still valid, so we perform the same test again, checking to see if v is contained in the dashed circle. This time, it is not, so we know that ℓ will be in D_P.

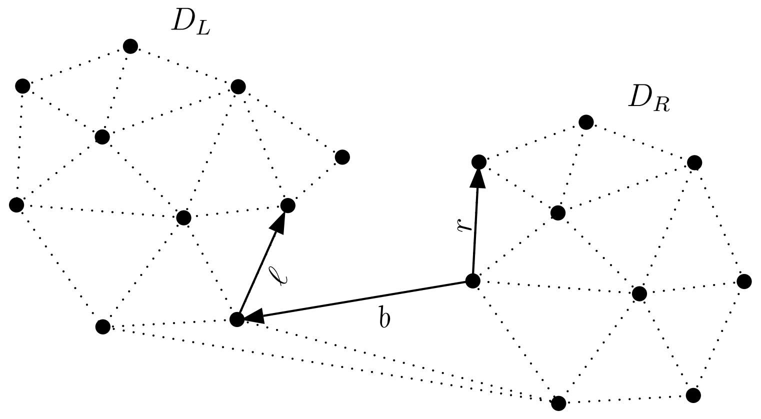
For completeness, figures 3.4j to 3.4l demonstrate the subsequent assessment of r and the addition of the cross-edge connecting *r.dest* and *b.dest*. Figure 3.4j illustrates that r does not require removal. This time, when adding the cross-edge, InCircle(*b.dest, b.org, ℓ.dest, r.dest*) returns true, indicating that an edge connecting *b.org* and *ℓ.dest* would be non-Delaunay. So we add the other cross-edge, the one connecting *r.dest* and *b.dest*, as in figure 3.4l. Figure 3.4m skips over the intermediate cross-edges, and show the final states of ℓ and r when we reach the upper common tangent, *uct* of D_L and D_R. Finally, we reach figure 3.4, the finished Delaunay triangulation D_P.



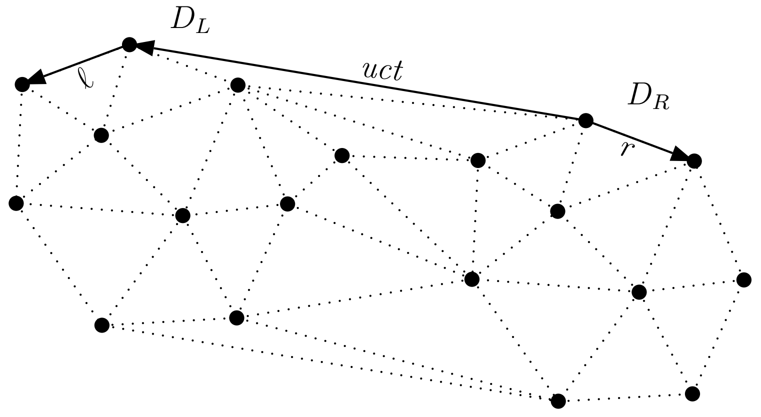
(j) We consider the right edge, r; it is still valid. Since *b.dest* has changed, we check that u is not in the circle formed by *r.dest, b.dest, and b.org*.



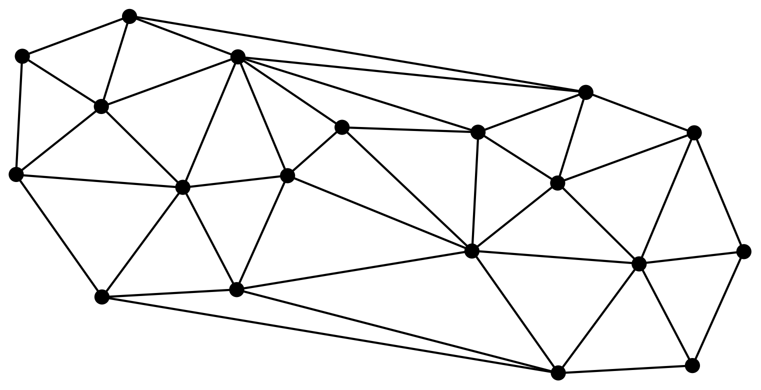
(k) Again, since ℓ and r are both valid we perform the InCircle test. Note that *r.dest* is in this circle, so the the edge connecting *b.org* and *ℓ.dest* is not Delaunay. Finally, we add b to D_P.



(l) Because the InCircle test returned true, we update b so that its source is now *r.dest*. We also update r to *b.org.cw()*.



(m) We continue in this manner until we reach the "top," that is, until the point when we reach the upper common tangent, shown here as *uct*. This will occur when both l and r are invalid.



(n) The finished diagram, D_P

Figure 3.4: Knitting together D_L and D_R to form D_P

3.3.3 Algorithm Pseudocode

MERGE takes two valid Delaunay triangulations, D_L and D_R, and knits them together to form $D_{L\cup R}$, the Delaunay triangulation on their combined point set. In particular, keep in mind that *ldi* is the convex hull edge coming out of the rightmost vertex in D_L, going in the clockwise direction. In addition, *rdi* is the convex hull edge coming out of the leftmost vertex in D_R, going in the counterclockwise direction.

Algorithm 9 MERGE

```
 1: procedure MERGE($D_L$, $D_R$)
 2:     $ldo$, $ldi \leftarrow D_L$
 3:     $rdi$, $rdo \leftarrow D_R$
 4:     $b \leftarrow$ LOWER-COMMON-TANGENT($ldi$, $rdi$)
 5:     $ldo$, $rdo \leftarrow$ OUTEREDGEFIXUP($ldi$, $ldo$, $rdi$, $rdo$, $b$)
 6:     loop                                              ▷ Rising bubble portion
 7:         $\ell \leftarrow b.twin.ccw()$
 8:         if VALID($\ell$) then
 9:             while INCIRCLE($b.dest$, $b.org$, $\ell.dest$, $\ell.ccw().dest$) do
10:                 $\ell' \leftarrow \ell.ccw()$
11:                 DELETEEDGE($\ell$)
12:                 $\ell \leftarrow \ell'$
13:             end while
14:         end if
15:         $r \leftarrow b.oprev$
16:         if VALID($\ell$) then
17:             while INCIRCLE($b.dest$, $b.org$, $r.dest$, $r.cw().dest$) do
18:                 $r' \leftarrow r.cw()$
19:                 DELETEEDGE($r$)
20:                 $r \leftarrow r'$
21:             end while
22:         end if
23:         if not VALID($\ell$) and not VALID($r$) then      ▷ $b$ is upper common tangent
24:             exit loop
25:         end if
26:         if not VALID($\ell$) or
                 (VALID($r$) and INCIRCLE($\ell.dest$, $\ell.org$, $r.org$, $r.dest$)) then
27:             $b \leftarrow$ CONNECT($r$, $b.twin$)
28:         else
29:             $b \leftarrow$ CONNECT($b.twin$, $\ell.twin$)
30:         end if
31:     end loop
32:     return $ldo$, $rdo$
33: end procedure
```

Algorithm 10 OuterEdgeFixUp

```
1: procedure OuterEdgeFixUp($ldi$, $ldo$, $rdi$, $rdo$, $b$)
2:     if $ldi.org$ is $ldo.org$ then
3:         $ldo \leftarrow b.twin$
4:     end if
5:     if $rdi.org$ is $rdo.org$ then
6:         $rdo \leftarrow b$
7:     end if
8:     return $ldo$, $rdo$
9: end procedure
```

We begin by calling Lower-Common-Tangent in line 4, which returns b, the appropriately directed and oriented quad-edge connecting the vertices of the lower common tangent. Stepping back into Merge, we take one more quick detour– we call OuterEdgeFixUp with the handles into D_L, D_R and the edge b. This helper function does a quick check to make sure we're returning the correct handles into the resulting diagram.

We can now begin the journey upward: we're at the "rising bubble" stage, lines 6 through 31. Figure 3.4 on page 43 illustrates this process. The idea behind this loop is to begin with ℓ and r, which are edges on hulls of their respective triangulations. At the first iteration of the loop, we set ℓ to the next outgoing edge whose source is $b.dest$, meaning that ℓ is the edge on the convex hull of D_L who shares its source vertex with the quad-edge representing the lower common tangent. We do a an analogous thing with r. As we iterate through the loop, ℓ and r are always set to the convex hull edge whose respective sources are shared with the "uppermost" known cross-edge that the time, b. We need to determine whether or not these edges will remain in the final diagram, and subsequently, the appropriate edge which will span D_L and D_R. In line 8 we check to see whether or not ℓ is valid with Valid. Valid is a helper function built on top of CCW that tells us whether or not an edge is above b.

If ℓ is a valid edge, we enter a loop in line 9 which we assess whether or not ℓ can remain in the merged diagram. This is accomplished with an InCircle test. Recall that *ldi.ccw()* points to the next edge (in counterclockwise direction) whose source is *ldi.org*. If *ldi.ccw().dest* is contained in the circle defined by the endpoints of b and the destination of ℓ, then we know that creating a face with these vertices would result in a face which is not locally Delaunay. So we remove the offending edge, and update ℓ to point to the next edge with the same source.

When we exit this loop, we know that ℓ will be in the merged result. We wish to do the same for r; this is accomplished in the loop beginning at line 17. The only change is that if we fail the InCircle test, we remove the edge r and update r to *r.cw()*, where *r.cw()*is the next edge in the clockwise direction with the same source as r.

Finally, we have reached line 23, where we can say that we have a grasp on two edges, ℓ and r, which will be in the merged result. Note that since we might have deleted some edges, ℓ and r might not be edges on the convex hull of their respective triangulations. We know that one of the endpoints of new spanning edge will be either *b.org* or *b.dest*. We also know that the other endpoint of the new edge will be either *r.dest* or *ℓ.dest*. There are two possible triangles: *b.org*, *b.dest*, *ℓ.dest*; and *b.org*, *b.dest*, *r.dest*. The if statements following line 23 determine and create the appropriate cross edge. Precisely, if ℓ and r are not valid, then neither of them are above b, so we must be at the upper common tangent, and thus have no need to create a cross edge. In 26 we check to see if one or both of the following is true: ℓ is not valid[4]; or r is valid, and the triangle formed by *ℓ.dest*, *ℓ.org* and *r.dest* contains a site (namely, *r.cw().dest*). Since this triangle would not be locally Delaunay, we create the triangle *r.dest*, *b.dest*, *b.org* by adding an edge with CONNECT in line 27. We update b to be this edge, which is the "uppermost" edge spanning D_L and D_R. If neither of these cases holds, then we know *b.org*, *b.dest*, *ℓ.dest* to be a triangle which is locally Delaunay, and we return the edge reflecting that in line 29. Whew.

To summarize, Guibas and Stolfi's divide-and-conquer approach is, in many ways, an extension of the one presented by Lee and Schachter [19] in 1980. Initially, we sort a point-set lexicographically and then use recursive partitioning to triangulate that point-set. Guibas and Stolfi's approach diverges from the approach of Lee and Schachter in their development and usage of the quad-edge data structure to maintain the diagram. Though their quad-edge data structure has been shown to make a small sacrifice with respect to its memory requirements and overall performance [18, 23], it simultaneously maintains the dual of the triangulation, which is a boon to those interested in both diagrams. Moreover, this approach does not make use of any particular auxiliary data structures; our only responsibility is to maintain the edge structure.

[4]Meaning that ℓ's destination is an endpoint of b, so we can't create a triangle with *b.org*, *b.dest*, *ℓ.dest*.

Chapter 4

Towards Parallelism

One of the earliest goals of this thesis was to survey the state of high performance implementations which could handle the task of computing the Voronoi diagram on millions of points. We sought a foundation, and perhaps, inspiration, for an algorithm which could be adapted to parallelization on the GPU. Parallelism aims to embody the adage that many hands make light work. An algorithm with a high degree of parallelism prides itself on its ability to execute multiple tasks simultaneously. We think of a task as a collection of statements which need to be executed; usually this is done by the same thread which is executing the main program, but sometimes it's appropriate to delegate this bit of work to another thread, or even to another computer. We'll gloss over the particulars of various ways parallelism can be incorporated. Here, we present a brief survey of past material on the subject for the interested reader.

Thinking back to the brute force approach to constructing the Voronoi diagram presented in section 2.1, we see opportunities to introduce parallelism. As it currently stands, the approach proposes to construct the Voronoi region for each site, one after the other. This is in line with any algorithm not concerned with parallelism or concurrency: we have one worker executing all of the tasks sequentially. However, if we recognize that the tasks of computing Voronoi regions can be executed independently of each other— as they clearly are in our brute force approach— then we might wonder: "Why not have n workers computing just one region all at the same time?"

Recall that the runtime of VoronoiFromHalfPlanes is $\mathcal{O}(n^3)$. Note that by assigning n workers to compute each of the n regions in parallel we don't reduce the overall work done; it is still $\mathcal{O}(n^3)$. However, we have reduced the overall running time to the time required to compute just one region: $\mathcal{O}(n^2)$, which we might consider a success in and of itself.

That being said, we find ourselves motivated to do better, since we know that the Voronoi diagram has linear complexity and the optimal time for construction is $\mathcal{O}(n \log n)$. In short, we *must* be doing extra work. We are interested in opportunities to introduce a degree of parallelism to approaches which are already running in optimal time, or, at least closer to optimal than $\mathcal{O}(n^3)$. Namely, if the approach takes $\mathcal{O}(n \log^k n)$ work, then a parallelization of that approach that uses a linear number of processing elements might run as fast as $\mathcal{O}(\log^k n)$ time, that is to say, polylogarithmic time, and would perform roughly the same amount of work as Fortune's and Guibas & Stolfi's sequential algorithms.

Before moving forward, we would be remiss in neglecting to acknowledge the incredibly fast, well documented, and highly regarded implementations to compute the Voronoi and Delaunay diagrams in use today. Shewchuk's project *Triangle*, which provides finely-tuned implementations of sweepline, divide-and-conquer, and incremental algorithms, deserves a specific mention. The CGAL Project [25] and Qhull [2] are two others; though the former is a much more general library for computational geometry algorithms. Finally, we point the interested reader to Su and Drysdale [24] for a comparison of sequential algorithms for computing the Delaunay Triangulation.

There are three approaches which are often used to construct the Voronoi diagram. Further, each of these strategies can be (and has been) modified to find the Delaunay triangulation. In particular, we discussed Fortune's algorithm, an example of a sweep line approach. Then, we dove into a divide-and-conquer strategy, applied to the dual diagram. We didn't cover the incremental approach for either diagram; suffice it to say that the incremental approach to computing $\text{Vor}(P)$ entails computing $\text{Vor}(P \setminus p)$ and then inserting the site p into the point-set and computing the new diagram. Guibas, Knuth, and Sharir give insight into the particulars of this approach [16].

Often, approaches that leverage the divide-and-conquer paradigm are good candidates for parallelization. In their 1975 paper "Closest-point Problems", Shamos and Hoey [22] presented the first approach to constructing the Voronoi diagram using a divide-and-conquer strategy. Chow's PhD thesis, "Parallel Algorithms for Geometric Problems", is the first to provide a parallel approach based on Shamos and Hoey's strategy. Given n processors, hers takes $\mathcal{O}(\log^3 n)$ time [9]. Aggarwal, Chazelle, Guibas, Ó'Dúnlaing, and Yap's paper "Parallel computational geometry"[1] expands on Chow's work, and is improved upon further by Goodrich, O'Dunlaing, and Yap [13]. In their paper "Fast Parallel Algorithms for Voronoi Diagrams" they propose an $\mathcal{O}(\log^2 n)$ time parallel algorithm on n processors.

Goodrich, O'Dunlaing, and Yap takes advantage of an observation made by Aggarwal et al., which concerns the intersection of Voronoi edges and the convex hull of the point-set. This insight allows them to move away from the contour-tracing step of Shamos and Hoey, the (sequential) process by which the jagged dividing line between two adjacent Voronoi diagrams is determined.[1] In particular, they note that the contour tracing problem can be reduced to a point location problem if the subdivision of $\mathbb{R}^2$ is determined by a planar graph whose form is a complete binary tree.

In 1993, Goodrich et al. published "External-Memory Computational Geometry," which includes a parallel algorithm for computing the convex hull of a point-set in a three dimensional space. It has been shown that there is an interesting relationship between Delaunay diagrams in a dimension d and convex hulls in dimension $d + 1$. Specifically, if we project a point-set from $\mathbb{R}^2$ onto a paraboloid in $\mathbb{R}^3$, and subsequently, compute the convex hull of the projected point-set, the embedding of the faces of the lower convex hull back into $\mathbb{R}^2$. For more detail on this relationship, con-

[1] Though we did not discuss it directly, the analogous step in the divide-and-conquer approach for the dual is the "rising bubble" stage.

sult Cheng, Dey, and Shewchuk [8]. In 1996 Blelloch, Miller, and Talmor published "Developing a Practical Projection-Based Parallel Delaunay Algorithm", makes use of the relationship between the convex hulls and Delaunay triangulations a bit differently, using a divide-and-conquer approach which aims to put the brunt of the work on the "split" step rather than the "merge" step. This move allows them to provide a parallel algorithm which performs and optimal amount of work and runs in $\mathcal{O}(\log^3 n)$ on n processing elements.

Almost ten years later, Blandford, Blelloch, and Kadow published "Engineering a Compact Parallel Delaunay Algorithm in 3D", which is a parallelized version of a randomized incremental approach to computing the Delaunay tetrahedralization, that is to say, they Delaunay triangulation in $\mathbb{R}^3$. They rely on the strategies outlined by Bowyer [6] and Watson [26], which they, and many others, refer to as the Bowyer-Watson kernel. Their approach is concurrent and thread-safe, meaning, among other things, that there is shared memory among all the workers.

Though Fortune's sweep line algorithm seems inherently sequential, a sequential variation of it involving a "sweep circle" presented by Dehne and Klein [10] in 1988 sparked recent the work of Xin, Wang, Xia, Mueller-Wittig, Wang, and He in 2013 [27]. In their paper, the sweep circle begins as a circle centered somewhere in the plane $\mathbb{R}^2$ whose radius is 0. The approach involves expanding the radius of the sweep circle, taking note of any intersections with sites. When a site intersects with the sweep circle, it triggers the creation of an ellipse. The ellipse has the sweep circle's center and the site's location as its focii; the boundary of the ellipse is locus of points which are equidistant to both the site and the circle. The outer boundary of the intersection of the elliptic segments constitutes the *beach curve*, the extension of the beach line.

They show that the sweep line approach is a degenerate case of the sweep circle; one where the circle is centered at infinity, this result also implies the same running time complexity for a sequential version of their approach. The facts that the sweep circle can be centered anywhere on the plane and that the diagram within the beach curve is closed and completely determined hints at the potential for a parallel algorithm. Xin et al. propose to partition the domain into disjoint regions and, in parallel, create a sweep circle at the barycenter of each region. Though their experimental results indicate that the sweep circle approach is slightly slower when implemented sequentially on a single-core machine, they demonstrate that when implemented on the GPU there is a significant speedup.

The project of devising parallel algorithms for Delaunay triangulations and Voronoi diagrams is an ongoing one. In this brief review, we mentioned just a few of the many papers published on the subject. The Voronoi and Delaunay diagrams remain significant objects of study due to their many uses in and of themselves; they can also be used to answer many other problems in related proximity and path-finding. The Voronoi diagram has also emerged independently in disciplines beyond mathematics; as such, it has many other aliases, among them: *Thiessen polygons* and *Wigner-Seitz cells*. As parallel programming, and, in particular, GPGPU programming, becomes increasingly accessible, the potential for developing efficient algorithms grows.

And with that, dear Reader, our journey draws to an end.

Bibliography

[1] A. Aggarwal et al. “Parallel computational geometry”. In: *Algorithmica* 3.1 (1988), pp. 293–327. ISSN: 1432-0541.

[2] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. “The Quickhull algorithm for convex hulls”. In: *ACM TRANSACTIONS ON MATHEMATICAL SOFTWARE* 22.4 (1996), pp. 469–483. URL: `http://www.qhull.org`.

[3] Marc de Berg et al. *Computational Geometry: Algorithms and Applications.* English. Germany: Springer-Verlag, 1997. ISBN: 30540-61270-X.

[4] Daniel K. Blandford, Guy E. Blelloch, and Clemens Kadow. “Engineering a Compact Parallel Delaunay Algorithm in 3D”. In: *Proceedings of the Twenty-second Annual Symposium on Computational Geometry.* SCG ’06. Sedona, Arizona, USA: ACM, 2006, pp. 292–300. ISBN: 1-59593-340-9. DOI: `10.1145/1137856.1137900`. URL: `http://doi.acm.org/10.1145/1137856.1137900`.

[5] G. Blelloch, G. L. Miller, and D. Talmor. “Developing a Practical Projection-Based Parallel Delaunay Algorithm”. In: *Proceedings ACM Symposium on Computational Geometry.* May 1996.

[6] A. Bowyer. “Computing Dirichlet tessellations”. In: *The Computer Journal* 24.2 (1981), pp. 162–166. DOI: `10.1093/comjnl/24.2.162`. eprint: `http://comjnl.oxfordjournals.org/content/24/2/162.full.pdf+html`. URL: `http://comjnl.oxfordjournals.org/content/24/2/162.abstract`.

[7] A. Bray et al. “Voronoi residual analysis of spatial point process models with applications to California earthquake forecasts”. In: *ArXiv e-prints* (Jan. 2015). arXiv: `1501.06387 [stat.AP]`.

[8] Siu-Wing Cheng, Tamal K. Dey, and Jonathan Shewchuk. *Delaunay Mesh Generation.* 1st. Chapman & Hall/CRC, 2012, pp. 31–53. ISBN: 1584887303, 9781584887300.

[9] Anita Liu Chow. “Parallel Algorithms for Geometric Problems”. AAI8108465. PhD thesis. Champaign, IL, USA, 1980.

[10] Frank K. H. A. Dehne and Rolf Klein. “A Sweepcircle Algorithm for Voronoi Diagrams”. In: *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science.* WG ’87. London, UK, UK: Springer-Verlag, 1988, pp. 59–83. ISBN: 3-540-19422-3. URL: `http://dl.acm.org/citation.cfm?id=647668.730977`.

[11] Rene Descartes. "Descartes: The World and Other Writings". English. In: ed. by Stephen Gaukroger. Cambridge, United Kingdom: Cambridge University Press, 2004. ISBN: 0-511-03579-9, 0-521-63158-0, 0-521-63646-9.

[12] S Fortune. "A Sweepline Algorithm for Voronoi Diagrams". In: *Proceedings of the Second Annual Symposium on Computational Geometry.* SCG '86. Yorktown Heights, New York, USA: ACM, 1986, pp. 313–322. ISBN: 0-89791-194-6. DOI: 10.1145/10515.10549. URL: http://doi.acm.org/10.1145/10515.10549.

[13] Micahel T Goodrich, Colm O'Dunlaing, and Chee Yap. "Fast Parallel Algorithms for Voronoi Diagrams". In: (1985).

[14] Michael T. Goodrich et al. "External-Memory Computational Geometry (Preliminary Version)". In: *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993.* 1993, pp. 714–723. DOI: 10.1109/SFCS.1993.366816. URL: http://dx.doi.org/10.1109/SFCS.1993.366816.

[15] Leonidas Guibas and Jorge Stolfi. "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams". In: *ACM Transactions on Graphics* 4.2 (1985), pp. 74–123.

[16] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. "Randomized incremental construction of Delaunay and Voronoi diagrams". In: *Algorithmica* 7.1 (1992), pp. 381–413. ISSN: 1432-0541. DOI: 10.1007/BF01758770. URL: http://dx.doi.org/10.1007/BF0175877.

[17] Isabella Jorissen. *Visualizing Fortune's Algorithm with PyOpenGL.* 2015. URL: https://github.com/ifjorissen/fortune_voronoi.git.

[18] Geoff Leach. "Improving Worst-Case Optimal Delaunay Triangulation Algorithms". In: *In 4th Canadian Conference on Computational Geometry.* 1992, p. 15.

[19] D. T. Lee and B. J. Schachter. "Two algorithms for constructing a Delaunay triangulation". In: *International Journal of Computer & Information Sciences* 9.3 (1980), pp. 219–242. ISSN: 1573-7640. DOI: 10.1007/BF00977785. URL: http://dx.doi.org/10.1007/BF00977785.

[20] Der-Tsai Lee. "Proximity and Reachability in the Plane." AAI7913526. PhD thesis. Champaign, IL, USA, 1978.

[21] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* English. New York: Springer-Verlag, 1985. ISBN: 0-387-96131-3 3-540-96131-3.

[22] Michael Ian Shamos and Dan Hoey. "Closest-point Problems". In: *Proceedings of the 16th Annual Symposium on Foundations of Computer Science.* SFCS '75. Washington, DC, USA: IEEE Computer Society, 1975, pp. 151–162. DOI: 10.1109/SFCS.1975.8. URL: http://dx.doi.org/10.1109/SFCS.1975.8.

[23] Jonathan Richard Shewchuk. "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator". In: *Applied Computational Geometry: Towards Geometric Engineering*. Ed. by Ming C. Lin and Dinesh Manocha. Vol. 1148. Lecture Notes in Computer Science. From the First ACM Workshop on Applied Computational Geometry. Springer-Verlag, May 1996, pp. 203–222. URL: `http://www.cs.cmu.edu/~quake/triangle.html`.

[24] Peter Su and Robert L. Scot Drysdale. "A comparison of sequential Delaunay triangulation algorithms". In: *Computational Geometry* 7.56 (1997). 11th {ACM} Symposium on Computational Geometry, pp. 361 –385. ISSN: 0925-7721. DOI: `http://dx.doi.org/10.1016/S0925-7721(96)00025-9`. URL: `http://www.sciencedirect.com/science/article/pii/S0925772196000259`.

[25] The CGAL Project. *CGAL User and Reference Manual*. 4.8. CGAL Editorial Board, 2016. URL: `http://doc.cgal.org/4.8/Manual/packages.html`.

[26] D. F. Watson. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". In: *The Computer Journal* 24.2 (1981), pp. 167–172. DOI: `10.1093/comjnl/24.2.167`. eprint: `http://comjnl.oxfordjournals.org/content/24/2/167.full.pdf+html`. URL: `http://comjnl.oxfordjournals.org/content/24/2/167.abstract`.

[27] Shi-Qing Xin et al. "Parallel computing 2D Voronoi diagrams using untransformed sweepcircles". In: *Computer-Aided Design* 45.2 (2013). Solid and Physical Modeling 2012, pp. 483 –493. ISSN: 0010-4485. DOI: `http://dx.doi.org/10.1016/j.cad.2012.10.031`. URL: `http://www.sciencedirect.com/science/article/pii/S0010448512002369`.