the big paper, if you will

---

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

I want to thank a few (lots, really) people.

# Preface

[I'm going to tell you a story about a math problem. There will be some words (don't worry about their meaning too much), some concepts (the words will help some), and a few pretty diagrams (to pull it all together). This is a story about a lot of things, depending on your perspective. In a way, it's a story about figuring out where you ought to go based on where you are. At least, that one of the ways I like to think about it. I've thrown in a little High Performance Computing for good measure, since - if you're anything like me - you'll want to know where your destination is as quickly as possible.

This math problem has a lot of names, and has been discovered, re-discovered, named, re-named, used, and re-used in more disciplines than I'm willing to count or list. I'd argue that the "conceptual" "mathematical" structure itself "belongs" to the realm of Computational Geometry, where it's referred to as the Voronoi Diagram. It has many aliases: Dirichlet Tesselation, Thiessen Polygons, Plant Polygons, and Wigner-Seitz Cell, among others.

In the names of Tradition and Clarity, I've done my best to use exoteric language whenever possible, and to indulge in mathematical notation where appropriate. You're welcome to treat either as an endless stream of notation. My only advice to you, in the words of a Reed math professor I once had: "remain calm."]

# Table of Contents

# List of Algorithms

# List of Figures

# Abstract

[more specifics][math specifics]

# 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, 2004, 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 It remained largely unpublished for thirty-one years after its abandonment, save for some fragments which appeared in *Principia philosophiae*.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 cosmology; 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,"(Descartes, 2004, 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. The form of the first element is terribly small, and moves incredibly quickly– the status, position, or size at any given time is imperceptible and indeterminate. The form of the third element, however, a large and move with much less urgency, and may resist the forces of other bodies. Further, 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," (Descartes, 2004, 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 (Descartes, 2004, 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,"(Descartes, 2004, 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,"(Descartes, 2004, p. 35).

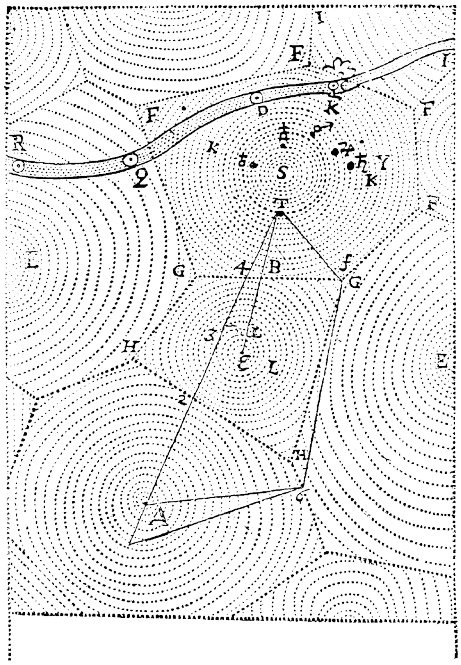


Figure 1.1: Descartes' Heavenly Regions

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 $\epsilon$ — occurs along the set of points equidistant from both $S$ and $\epsilon$. The "corners" of the firmament, as Descartes calls them, are points where three circles— each originating from a distinct heavenly body— intersect. [What was the sentence that Jim said would be good?]

## 1.2 Definition of the Voronoi Diagram 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 $P$, $\text{Vor}(P)$, 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 $\mathbb{R}^2$.

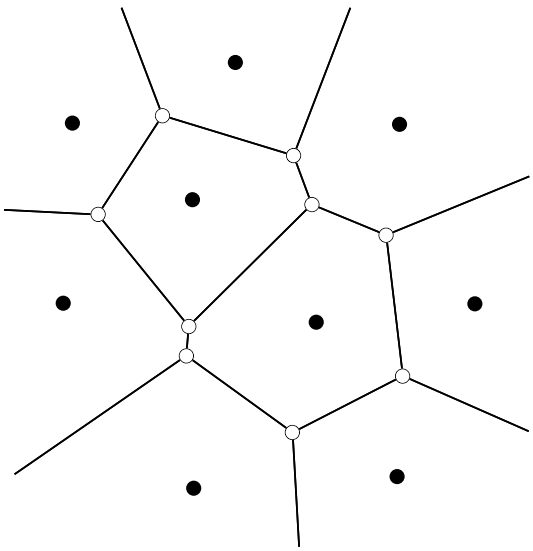


Figure 1.2: A Voronoi Diagram on nine sites

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.

The Voronoi diagram is a tiling of the plane. It is also referred to as the Voronoi tessellation, and occasionally as the Voronoi graph. [finish or cut]

## 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, $\ell$ connecting them. Let $b$ be the perpendicular bisector of $\ell$, 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 $h(p, q)$.

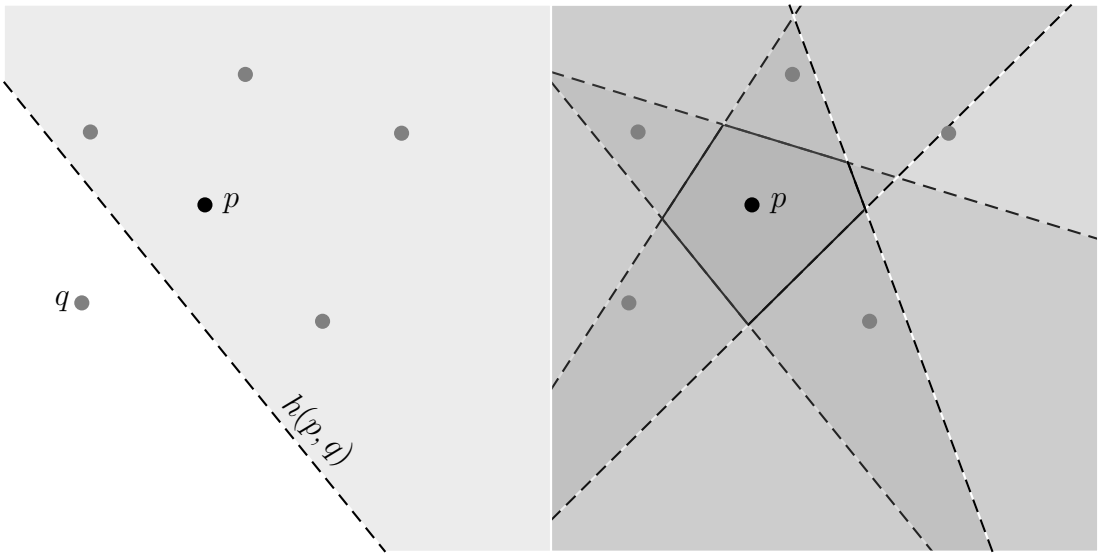
Define set of half-planes containing $p_i$ as $H_i := \{\, h(p_i, p_j) \mid j \neq i \}$. There are $n-1$ half-planes in $H_i$. Consider the intersection of any two of these half-planes, $h(p_i, p_j)$ and $h(p_i, p_k)$. The set of points lying in the intersection of $h(p_i, p_j)$ and $h(p_i, p_k)$ are 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:

$$V_i = \bigcap_{h \in H_i}$$

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.

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 the region was the result of $n - 1$ half-plane intersections, we know that the region has, at most, $n - 1$ edges.




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

Figure 1.3: Half-Plane Intersections

## 1.4 The Voronoi Graph

Note that 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 $\text{Vor}(P)$, 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 $\text{Vor}(P)$ 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 $\text{Vor}(P)$ 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 1.4 illustrates how the addition of a vertex at infinity changes the Voronoi diagram of nine sites we saw in figure 1.2.

We can leverage the properties of a planar[1], graph further. Specifically, we can apply *Euler's Formula* to show upper bounds on the number of vertices, edges, and faces in $\text{Vor}(P)$.

*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

[1]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 $\mathbb{R}^2$

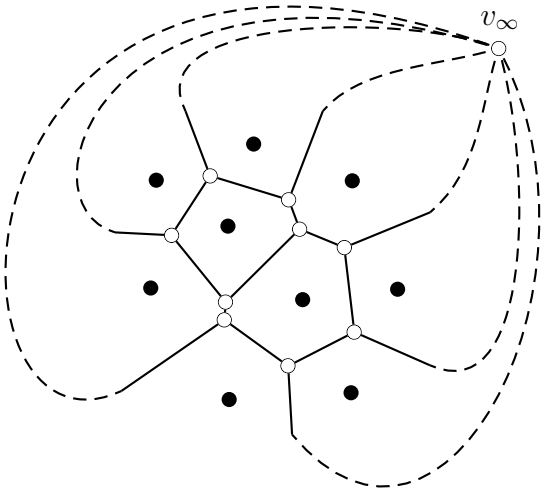



Figure 1.4: Connecting Half-Infinite Edges to $v_\infty$

graph is *simple,*[2] 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,[3] we can say that $2m \geq 3(k + 1)$. Together with Euler's formula, this implies that when $n \geq 3$, 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.

## 1.5 The Voronoi Diagram and its Dual

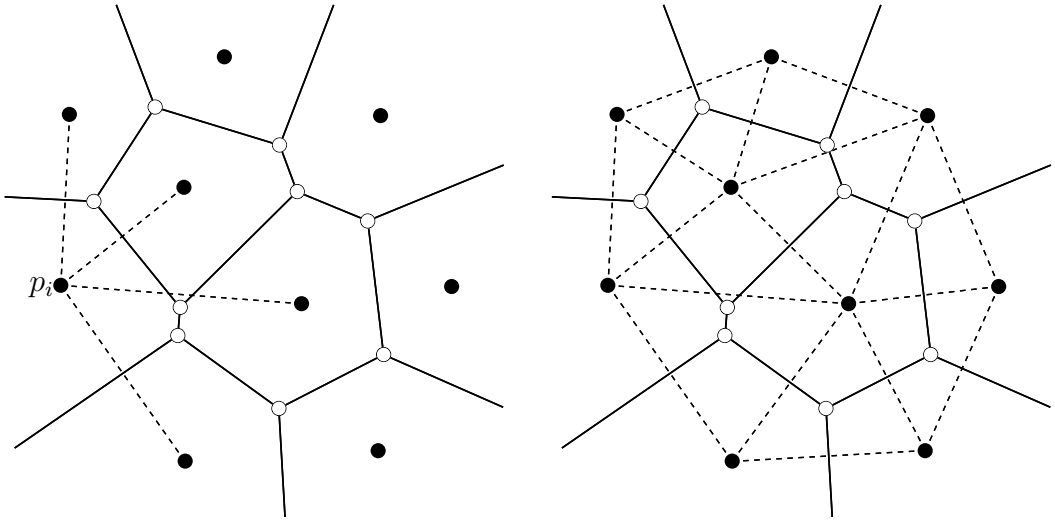
While the Voronoi diagram can be computed directly with relative— though certainly not always computational— ease, it can also be computed somewhat indirectly, that is, through the construction of its dual, the *Delaunay subdivision.* If we assume that the set of sites $P$ is "well-behaved,"[4] then the resulting diagram is the Delaunay triangulation of those sites, $\mathrm{Del}(P)$. Furthermore, the Delaunay triangulation of a Voronoi diagram is unique. One of the many advantages of working with the dual is the ability to work with triangular faces instead of with polygonal regions with an unknown number of sides.

[2] 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".

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

[4] We'll wave our hands at this for now, and in section 1.6 we'll define more precisely what it means for the set of sites to be "well-behaved."

We can develop our intuition for the Delaunay triangulation and the primal-dual relationship by constructing Del($P$) from Vor($P$). 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 1.5a. Do this for all the sites. The resulting diagram is the unique Delaunay Triangulation of $P$, as shown in figure 1.5b.




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

Figure 1.5: Voronoi Diagram (Solid) and Delaunay Triangulation (Dashed)

Let's take a couple steps back and 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 $\alpha_i$'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 1.6 illustrates the convex hull of the same point set used in figures 1.5a and 1.5b. This region of the plane is best understood in terms of a complete description of its boundary, as shown in figure 1.6.

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.

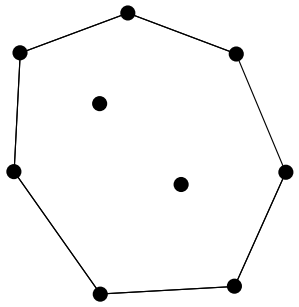


Figure 1.6: Convex hull of a set of points

We can confirm the Delaunayhood of a triangulation in two ways. For two sites $a, b \in P$, we say that that the edge connecting them satisfies the Delaunay property if the circle passing through $a$ and $b$ does not contain any of the other sites in $P$. An equivalent, and perhaps more common, understanding of the Delaunay property operates on the faces (triangles) 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 1.7, 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 1.7, the triangles which share edge $e$ are $\Delta abd$ and $\Delta bcd$. Because $a$ does not lie in the circumcircle of $\Delta bcd$, shown as a dashed line in figure 1.7, we say that $e$ is *locally delaunay*. When every edge in the triangulation is locally Delaunay, we may say that the triangulation is *globally Delaunay*.

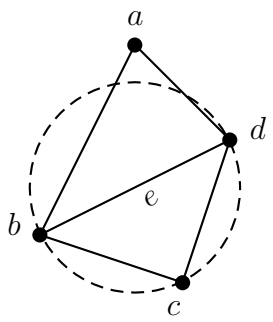



Figure 1.7: Testing the Delaunayhood of a small triangulation

What distinguishes the Delaunay triangulation of a set $P$ from any other triangulation of $P$ are its unique properties, which are directly related to its privileged position as the dual of the Voronoi diagram.

## 1.6 The Fine Print

Here, I'd like to introduce the notion of *general position*, and 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.

### 1.6.1 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 the 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 1.8. Note that 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, in fact, 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.

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 1.8 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.

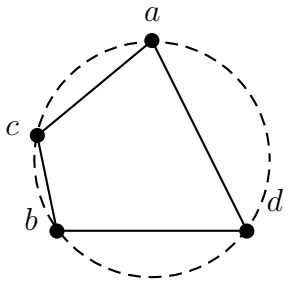



Figure 1.8: Four Co-circular sites

### 1.6.2 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 1.9. 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.

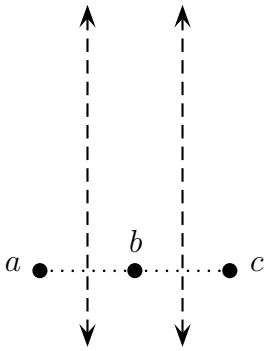



Figure 1.9: Voronoi Diagram of Three Collinear Sites

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.

## 1.7 Applications of the Voronoi and Delaunay Diagram

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 of the 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 $\mathcal{O}(n)$.

From the previous section 1.5, we saw that it is possible to obtain the dual of the Voronoi diagram in linear time. Thus, we can use the Voronoi diagram to solve the problem of triangulating a set of sites $P$, by returning the dual of the diagram.

The Post Office Problem is a classic, though perhaps slightly dated, motivation for the Voronoi diagram. However, it captures one of the central use cases for the Voronoi diagram, which is its ability to answer the *nearest neighbor problem*. The nearest neighbor problem can be stated as such: given a set of sites $P$, and a query point $q$, return the point in $P$ which is closest to $q$. Recall that the Voronoi diagram partitions the plane into a set of $\mid P \mid$ regions which satisfy the stipulation that every point in a given region is closer to the site associated with that region than it is to any other site. Then, given a query point $q$ and a Voronoi diagram of $P$, we can answer the nearest neighbor question by simply locating the region that $q$ is in, a question we can answer in $\mathcal{O}(\log n)$ time with linear auxiliary storage.(Preparata & Shamos, 1985, p. 214)

(Closely) related to this 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 $p_i$.[5] 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, $\mathrm{Del}(P)$. Recall that each vertex on the Delaunay triangulation is a site of $\mathrm{Vor}(P)$. Furthermore, note that the line segment connecting a vertex $p_i$ of $\mathrm{Del}(P)$ to its nearest neighbor is, and must be, one of the edges in $\mathrm{Del}(P)$. So, if we spend time on the order of $\mathcal{O}(n)$ computing the result of the all nearest neighbors problem on $\mathrm{Vor}(P)$, we will, by definition, have the answer to the closest pair problem for each site in $P$.

[5]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 $\mathbb{R}^2$.

# 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 $ax + by + c = 0$[6]. Correspondingly, 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

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

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, *inter*. In order for *inter* to be a Voronoi vertex, it must be the case that it satisfies: $inter \leq ax + by + c$ for each of the half-planes in $H_i$. That is to say, that *inter* must lie within (or on the boundary of) the common intersection of all half-planes in $H_i$. If *inter* 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 *intersection* 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 *intersection*. We repeat this process for each site in $P$. Section 2.1.1 provides a pseudocode implementation of this approach.

### 2.1.1 Algorithm Sketch & Pseudocode

**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:              inter ← HalfPlaneIntersection(h_j, h_k)
8:              is_in ←HalfPlaneSetContains(H_i, inter)
9:              if is_in then
10:                 AddIntersection(intersections, inter)
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
```

## 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.

### 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*.[7] As we'll soon see, the beachfront is a way for us to monitor the "in-progress" portion of the Voronoi diagram. The beachfront 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 2.10, $s^+$ 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:

[7]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.

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 $\alpha$, in figure 2.10, is added to a a sequence of parabolic arcs called the *beachfront*. In figure 2.10, $\alpha$ is the first arc in the beachfront. We will refer to the beachfront as $\beta(P)$.

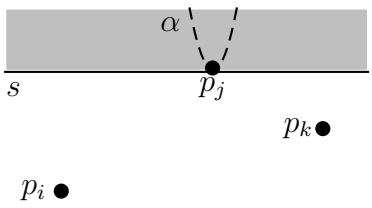



Figure 2.10: Adding the first arc, $\alpha$, to the beachfront

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 2.12b, the inserted arc $\omega$, intersects with the arc directly above it, effectively splitting into two pieces. At the moment where the new arc, $\omega$, is inserted into the beachfront, it is a simple vertical line (or a parabola with no width). To insert $\omega$ into the beachfront, we split the arc above it, $\alpha$ into two arcs and insert $\omega$ in between the split arcs. We record the points where $\omega$ intersects with the arcs adjacent to it, and we refer to these points as *breakpoints*[8] 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 $\omega$ 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 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 $\gamma$, $\alpha$, and $\omega$, which are segments of the parabolas defined by three sites $p_i$, $p_j$, and $p_k$, as in figures 2.12c and 2.12d. 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, $\ell$.

If $\alpha$ 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 $\ell$, the arcs corresponding to $p_i$ and $p_k$, $\gamma$ and $\omega$, respectively, intersect at $q$. At this point, the right breakpoint of $\gamma$ and the left breakpoint of $\omega$ become equal to the left and right breakpoints of $\alpha$, and $\alpha$ disappears. So, we handle the circle event $C$ when the sweep line reaches $\ell$. The effect of this on the beachfront is the removal of arc $\alpha$.

[8]Note that this means that the right breakpoint of $\omega$ 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 $\omega$ on the left is equal to $\omega$'s left breakpoint.

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 $\gamma$, $\alpha$, and $\omega$ 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.[9] So, circle events 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

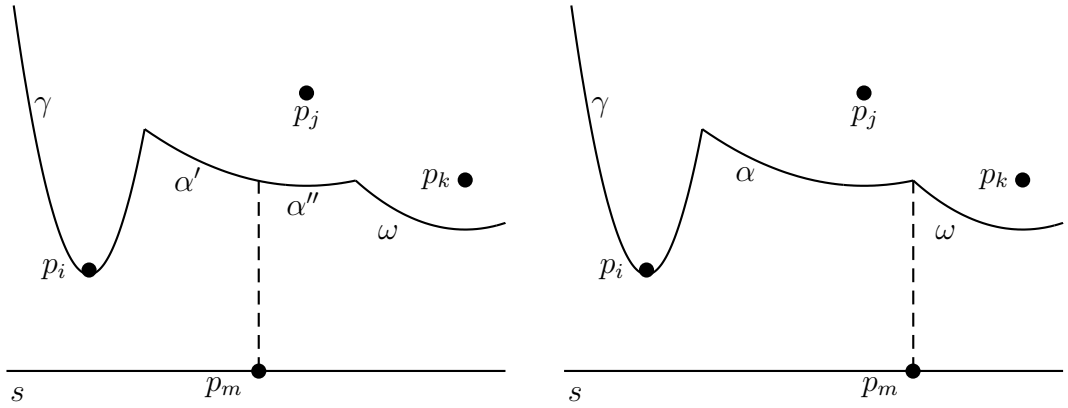
[9]Some circle events will be *false alarms*, a highly technical term we'll be defining soon. Aside from these false alarms, recall that handling a circle event adds a vertex to the finished diagram, and we know from section 1.4 that an upper bound for the number of vertices is $2n-5$ for a Voronoi diagram on $n$ sites.

- The inserted arc breaks up the segment above it, as in figure 2.11a
- The inserted arc intersects the beachfront at the intersection of two arcs, as in figure 2.11b

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 $\gamma$, $\alpha$, $\omega$. Let $\chi$ be the inserted arc corresponding to $p_m$, so arc $\alpha$ is split into $\alpha'$ and $\alpha''$, as in figure 2.11a. The new triples are, from left to right: $\gamma$, $\alpha'$, $\chi$; $\alpha'$, $\chi$, $\alpha''$; and $\chi$, $\alpha''$, $\omega$. Since the second triple has both $\alpha'$ and $\alpha''$ 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: $\gamma$, $\alpha$, $\chi$; $\alpha$, $\chi$, $\omega$; and if there is an arc to the right of $\omega$, call it $\psi$: $\chi$, $\omega$, $\psi$. We add circle events for each of these three triples.

Note that in both of these cases, the arcs associated with $C(p_i, p_j, p_k)$, $\gamma$, $\alpha$, $\omega$, 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.




(a) Site event at $p_m$ splits arc $\alpha$ into $\alpha'$ and $\alpha''$

(b) Site event at $p_m$ is directly below the intersection of $\alpha$ and $\omega$

Figure 2.11: Inserting $\chi$: two ways a site event might affect the beachfront

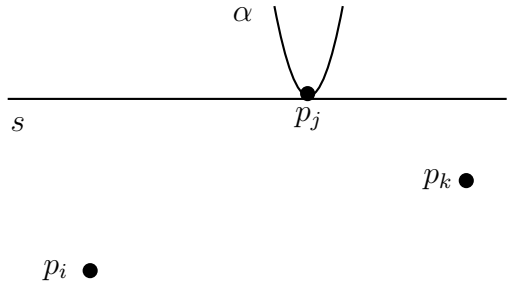
As long as the sites associated with the arcs are not collinear and are distinct, we will be able to describe a circle with a finite radius. Note that a circle event exists in the event queue if and only if it satisfies:

- The property that the circle defined by the sites corresponding to the triple of arcs intersects the sweep line

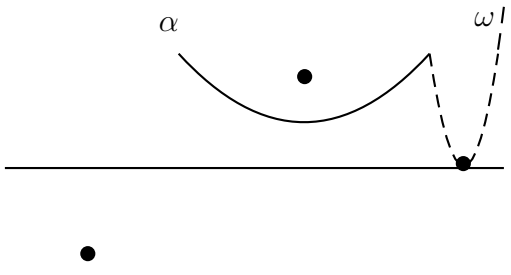
- The circle event hasn't already been deleted from the event queue

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 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.

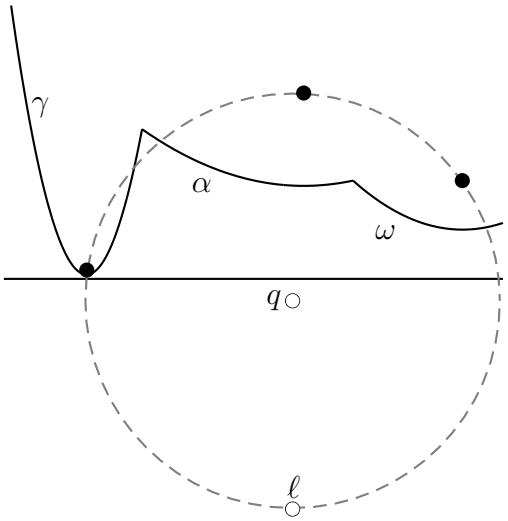
Figure 2.12 illustrates the application of Fortune's algorithm to a small set of three sites: $p_i$, $p_j$, and $p_k$. To understand how to handle a circle event, let's investigate this example. In figures 2.12a to 2.12c we are adding arcs to the beachfront by handling the site events associated with $p_j$, $p_k$, and $p_i$, respectively. In figure 2.12c, 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 2.12d illustrates how the arc $\alpha$ has shrunk as the sweep line has moved downward. The dotted lines reveal how the breakpoints of $\alpha$ have shifted during this time. When the sweep line intersects with $\ell$, these edges will meet at $q$, and we will add a Voronoi vertex at $q$, as shown in figure 2.12e. 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 $\gamma$ and $\omega$ 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.



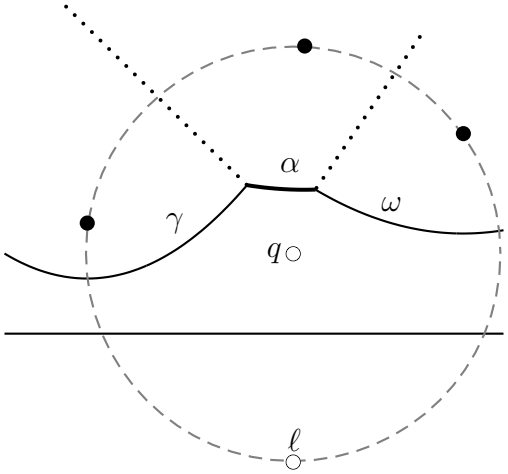

(a) Handling the first site event



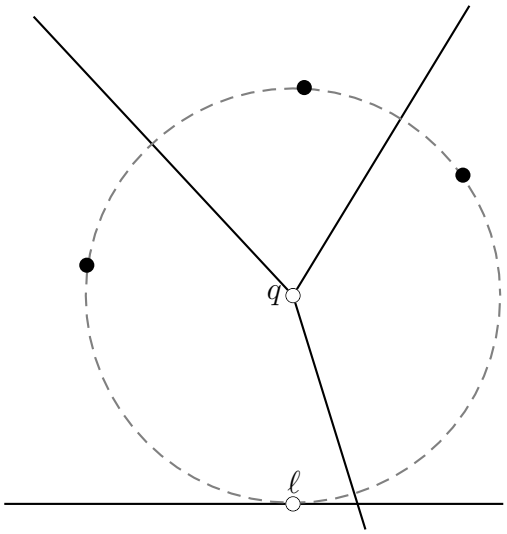

(b) Adding an arc, $\omega$, to the beachfront




(c) Adding a circle event: circle $C(p_i, p_j, p_k)$, with center $q$, and lowest point $\ell$.




(d) Approaching the circle event: Arc to be removed, $\alpha$, shown in bold, edges meeting at $q$ are dotted.




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

Figure 2.12: Constructing the Voronoi Diagram of three sites

### 2.2.3 Algorithm Pseudocode

VoronoiSweepline 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. The vertex and edge structure for Vor($P$) will use a doubly connected edge list (DCEL), as described in **??**. A common structure used to maintain the beachfront, $\beta(P)$, is some flavor of a balanced binary search tree (BBST); de Berg et al., 1997 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** VoronoiSweepline

**Require:** Set of sites, $P$, which are in *general position*

```
1:  procedure VoronoiSweepline(P)                          ▷ Construct Vor(P)
2:      Q ← P                          ▷ Initialize event queue, Q, with set of sites, P
3:      E ← {}                                              ▷ Initialize (empty) DCEL
4:      β ← {}                                          ▷ Initialize (empty) beachfront
5:      while Q is not empty do
6:          e ← 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
```

VoronoiSweepline is the conductor of this symphony. In lines 2, 3, and 4, we initialize our major structures. $\beta$ 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 VoronoiSweepline 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 $Q$.

**Algorithm 3** HANDLESITEEVENT

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

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: $Q$, $\beta$, 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 $\beta$. Otherwise, let $\alpha$ be the arc in the beachfront directly above $p$. If there was a circle event associated with 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 $p$, $\alpha$, into two arcs (as in figure 2.11a where $\alpha$ splits into $\alpha'$ and $\alpha''$). In the RB-Tree, the effect of this is the replacement of a leaf node (representing the arc $\alpha$) with a subtree whose three leaves store $\alpha'$, *new_arc*, and $\alpha''$. 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 $\beta$. Note that these arcs will be $\alpha'$ and $\alpha''$, 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:      α ←GetArcAbove(β, c)
3:      left_arc ←GetLeftArc(β, α)
4:      right_arc ←GetRightArc(β, α)
5:      left_edge ← GetEdge(E, left_arc, α)
6:      right_edge ← GetEdge(E, α, right_arc)
7:      if HasCircleFor(left, α) then
8:          RemoveCircle(Q, left)
9:      end if
10:     if HasCircleFor(right, α) then
11:         RemoveCircle(Q, right)
12:     end if
13:     RemoveArc(β, α)
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, β, left_arc)
20:     DetectCircleEvent(Q, β, 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 $\ell$) and assign it to a variable $\alpha$. Since we are removing $\alpha$ from the beachfront, the circle events on associated with its adjacent arcs (*left_arc* and *right_arc*) are invalid if they involve $\alpha$. 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 $\alpha$ from $\beta$.

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 $\alpha$ from $\beta$ 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*.[10] Finally, we need to see if the removal of $\alpha$ 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 $\alpha$.

Though there are no more events left in $Q$, we might not have removed all of the arcs in $\beta$. The remaining leaves in $\beta$ are the arcs which were not removed by any circle event. The internal nodes of $\beta$ 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 $\mathrm{Vor}(P)$ and find the intersections of the half-infinite edges with the bounding box.

[10] This footnote is for the detail-oriented reader, who might object to my SetEdgeDest function, crying out: "But in **??** you said that we only care about the source of an edge, not the destination! Conveniently, you've left the entire notion of edge twins out of this discussion! What's with that?" And to that, I say: "I'm sorry". I was only trying to make things simpler. You see, we've been dealing with half edges the whole time. When we call AddEdge(*left*,*right*), we create a pair of half edges between those arcs, [CLARIFY] one edge normal to the site associated with *left* and the site associated with *right* and another half edge normal to site associated with *right* and the site associated with *left*. So AddEdge return the half-edge normal to the sites associated with *left* and *right*. When we call SetEdgeDest(*edge*, *dest*), we are actually setting the source of *edge.twin*.

# Constructing the Delaunay Triangulation

## 3.1 A Divide and Conquer Approach

Guibas and Stolfi's "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams", Guibas & Stolfi, 1985, 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– using their quad-edge structure– whose time and space complexity is consistent with the bounds known to be optimal in the worst case.[11] As we saw in **??**, it's also easy for us to witness its elegance, as theirs is a data structure which represents both the Voronoi diagram and its dual simultaneously. Their quad-edge data structure, furthermore, supports the arbitrary construction and modification of any embedded graph on a two-dimensional manifold. 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 predicates ensure that the faces (composed of edges in the graph) maintain the Delaunay property.

### 3.1.1 Algorithm Sketch

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.

So, 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," or "divide," stage, in which we split the large problem of computing the Delaunay of $P$ into smaller subproblems. The second stage involves two recursive steps: we apply Guibas and Stolfi's algorithm to each of the two partitions and thus compute the Delaunay triangulation of each. Finally, we proceed to the "merge" phase, which carefully combines the triangulations of the partitions into a final triangulation on all $n$ sites. The algorithm is implemented recursively, and we will discuss how the algorithm leverages recursion and handles these base cases shortly.

[11] 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!

**Split!**

We assume two things: that the $x$-coordinate of each site in $P$ is unique, and that the sites are in *general position*,[12] meaning that there will be no degenerate cases for us to handle. 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 3.13a 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. Figure 3.13b illustrates how subproblems of size two or three are triangulated.

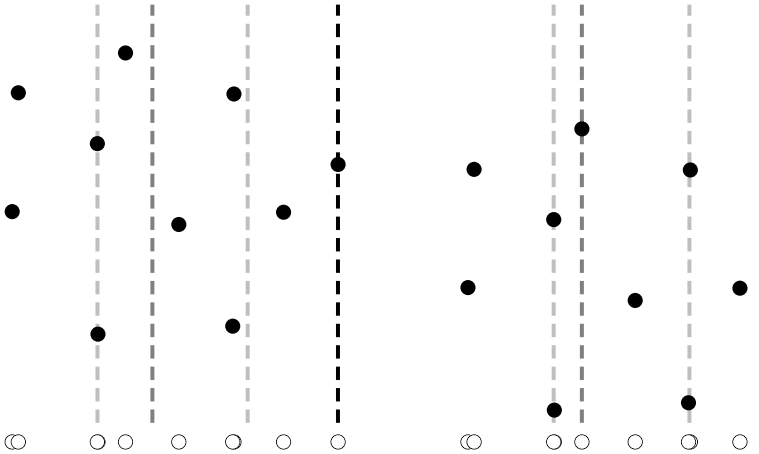
Note that once the size of the set of sites we are operating on reaches two or three, we do not try to partition it further. At this stage, we consider sets of sites with size two or three to be subproblems which conform to one of two base cases. The concept of a base case is central to recursive algorithms, because the base case(s) of a recursive algorithm are what ensure that the algorithm terminates– that no further recursive calls are made. In Guibas and Stolfi's divide and conquer algorithm, we have two base cases: the situations where $|P|$ is either two or three. When either of these is true, we immediately recognize that the Delaunay triangulation of $P$ 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 began with a set of sites, $P$, where $\mid P \mid = n$. We sorted the sites by x-coordinate, and split the set of sites into left and right halves according to the median x-value. We continued to do this until the solution to the problem was a trivial one, at which point we returned the triangulation. At this point, we have a collection of valid Delaunay diagrams on subsets of $P$; our next task will be to understand how they can be combined into the final triangulation $D_P$.

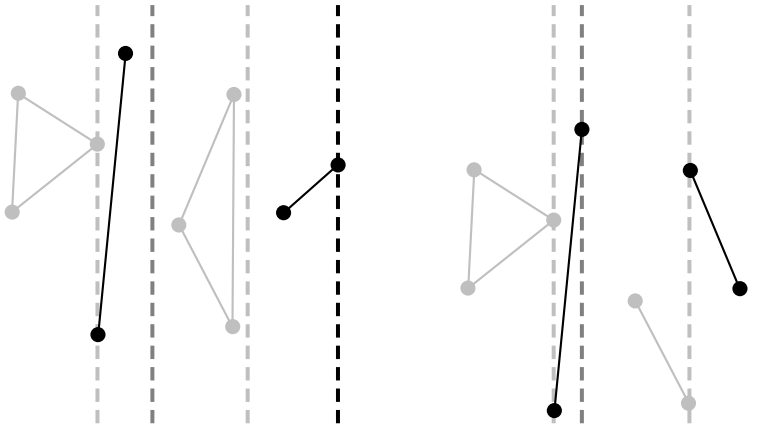
**Merge!**

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 aims to knit the triangulations together pairwise in the order they were partitioned while also maintaining the Delaunay property. So for a given pair of diagrams, say $D_L$ and $D_R$, the task of determining how $D_L$ and $D_R$ are connected falls to the merge step of the algorithm. Any given merge step, then, operates on two adjacent, valid, Delaunay triangulations, which have been separated by a known bisector, which we discussed in **??**. To provide you, dear Reader, with a visual intuition of what the merge step entails, I have included figure 3.14. Figure 3.14a illustrates the first iteration of the merge step; it knits together the eight subproblems in figure 3.13b into four subproblems. Figure 3.14b shows results of the second pass: it combines the diagrams to the left of the dashed vertical line into a single diagram, and, in a similar fashion, combines the pair to the right of the line as well. Note that the dotted lines in figures 3.13a and 3.13b are edges which had to be removed in order to maintain

[12] As discussed in **??**.



(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.

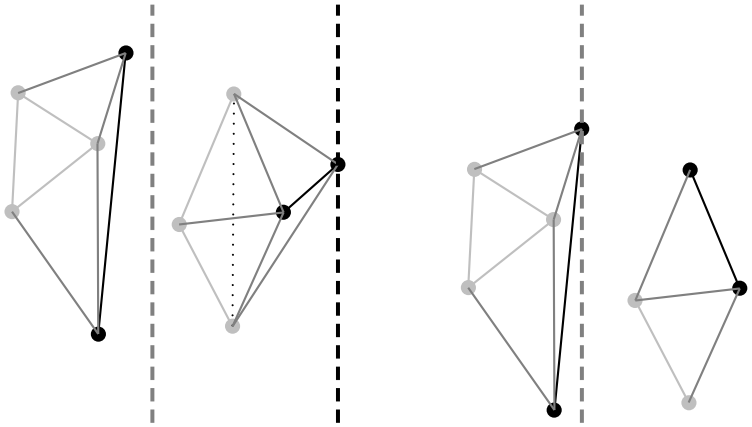


(b) Solving (triangulating) the subproblems.

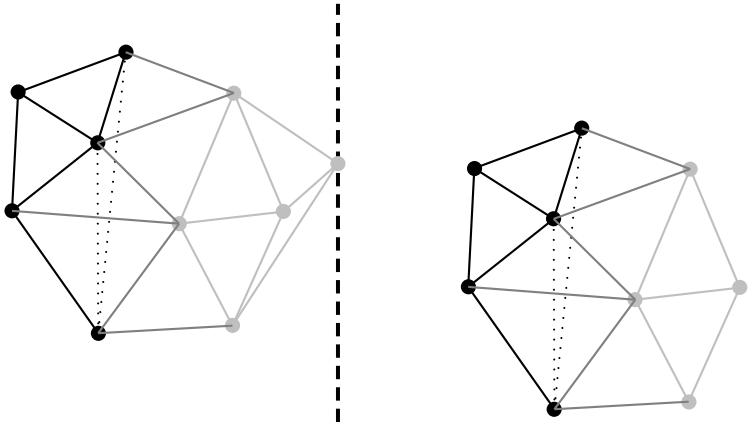
Figure 3.13: Dividing $P$ into subproblems we know how to solve

the delaunayhood of the merged diagram. One of our goals in this section will be understand how Guibas and Stolfi apply the *empty circle property*– introduced in section 1.5– to determine which, if any, edges need to be removed during the merge process.

To understand precisely how the merge step works, we will break it down into two more steps. First, we find the *lower common tangent* of $D_L$ and $D_R$. The lower common tangent is the lowest line which is a) tangent to the polygons formed by the boundary of the convex hull; and b) the line does not intersect either polygon.For our purposes the lower common tangent is also the edge which spans $D_L$ and $D_R$ and is the first valid edge of the merged result, $D_P$, as in figure 3.15. We know this edge to be valid because the lower and upper common tangents of $D_L$ and $D_R$ can easily



(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 3.14a. Again, the dotted line segments are removed to maintain the Delaunay propertry.

Figure 3.14: Merging subproblems pairwise.

be shown to be edges of the convex hull of the sites in $P$, that is, $L \cup R$. The lower common tangent can be found in linear time in the combined sizes of the hulls of $L$ and $R$. Once we have the lower common tangent, we wish to determine what the first *cross-edge* will be. We will think of the cross edges as the edges which connect $D_L$ and $D_R$, and if we choose them carefully, we can maintain the delaunayhood of the diagram.

So, how do we choose these edges? Given *basel*, the directed edge connecting $D_L$ and $D_R$, whose endpoints are those of *lct*, as in figure 3.15; we will perform the second part of the merge step, which utilizes the notion of a *rising bubble* to maintain the Delaunay property. The name reflects the conceptual actions we're about to perform: the rising bubble exists in the algorithm as the series of circle tests performed (roughly) along the vertical line separating $D_L$ and $D_R$. Recall that we can

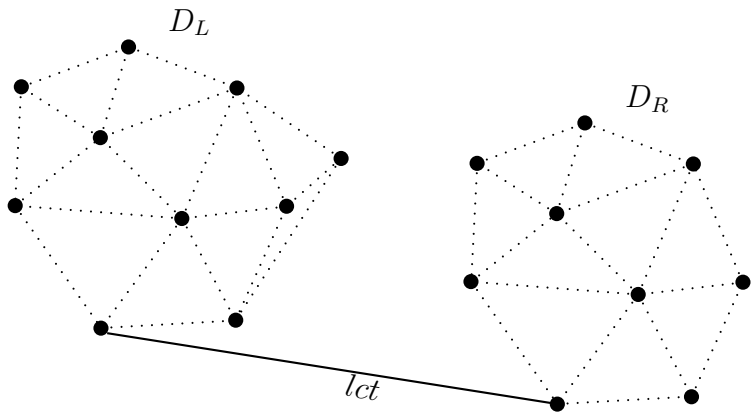



Figure 3.15: Lower Common Tangent, $lct$, of $D_L$ and $D_R$

perform a series of circle tests to enforce the Delaunay condition, which states that a triangulation is Delaunay if and only if the circumcircle of every interior triangle is point-free (that is, contains no other sites).

Before we create the *rising bubble*, recall that *basel* is a directed half-edge[13] going from *rdi.twin.source* to *ldo.source*, as in **??**. The first circle we create uses the left and right endpoints of *basel*. Recall that the vertices of the diagram we are using are located in $D_R$ and $D_L$, respectively. Since *basel* is a directed edge, the vertices which serve as two inputs to our circle test can also be referred to as *basel.source* and *basel.twin.source*.

We introduce two more edges which we will use throughout this portion of the algorithm. $\ell$ 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. At the outset of this procedure, we set $\ell$ to be *basel.twin.onext()* where *onext()* is conveniently defined to be a function which returns the next edge out of the vertex *basel.twin.source*. At this point, we'd like to know whether or not $\ell$ is valid edge.

What does it mean for an edge to be *valid*? A candidate edge is valid if the edge is located "above" *basel*. That is to say, if we took the destination vertex of a given edge $e$ (which we can get easily with *e.twin.source*) and the endpoints of *basel*, the three vertices need to be ordered in a counterclockwise orientation. This predicate, referred to by Guibas and Stolfi as CCW is implemented as:

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

So, in the case of determining whether or not $\ell$ is valid, we check its validity by ensuring that CCW($\ell$.*twin.source*, *basel.twin.source*, *basel.source*) is greater than 0. If $\ell$ is valid, then we may continue to the next step, which involves "pruning" $D_L$ in an appropriate way. What we need to do is consider whether or not any of edges

[13] As discussed in **??**.

from $\ell.source$– including $\ell$– intersect with the cross edge we'd like the insert. To do this we use the InCircle test, the second geometric predicate that Guibas and Stolfi use.

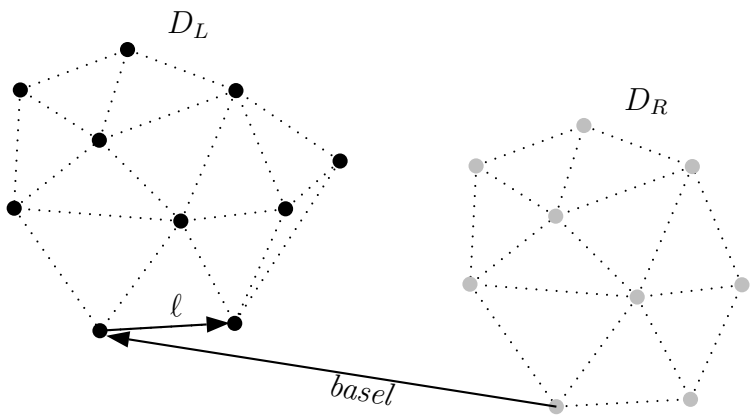
The InCircle test takes as its arguments four sites, $A$, $B$, $C$, and $D$, and envisions a counterclockwise oriented circle $ABC$ and a lonely point $D$. The InCircle is defined by Guibas and Stolfi as follows: "InCircle($A$, $B$, $C$, $D$) is defined to be true if and only if point $D$ is interior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

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

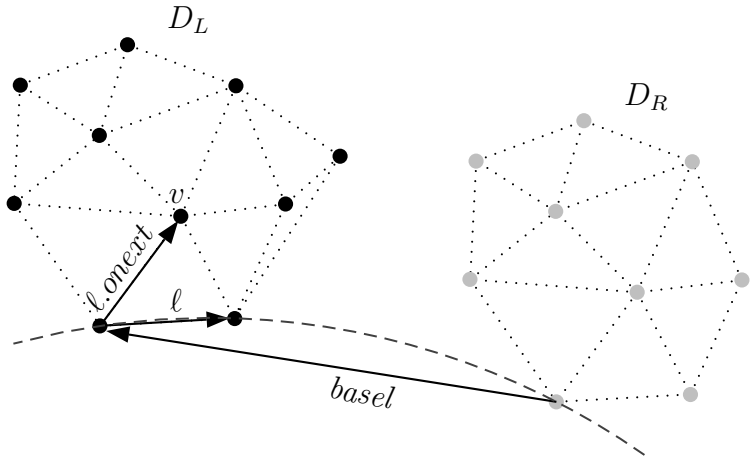
$$InCircle(A, B, C, D) = \begin{vmatrix} x_A & y_A & x_A^2 + y_A^2 & 1 \\ x_B & y_B & x_B^2 + y_B^2 & 1 \\ x_C & y_C & x_C^2 + y_C^2 & 1 \\ x_D & y_D & x_D^2 + y_D^2 & 1 \end{vmatrix} > 0.$$

Now the notion of using circles to enforce, or as we might say *witness*, the delaunayhood of a certain edge, is not new. The InCircle predicate is precisely the circle test we performed in section 1.5 and saw in figure 1.7. When we apply this notion of the InCircle to determining whether or not $\ell$ will remain in the merged diagram $D_P$, we will do so as follows: if InCircle(*basel.twin.source*, *basel.source*, $\ell$.*twin.source*, $\ell$.*onext*().*twin.source*) then $\ell$.*onext*().*twin.source* is in the circle defined by *basel.twin.source*, *basel.source*, and $\ell$.*twin.source*, which means that we must delete the edge $\ell$ and set the new value of $\ell$ to be $\ell$.*onext*(). We continue to do this, deleting edges as necessary, until the InCircle test fails. Then, we repeat an analagous procedure for $r$.

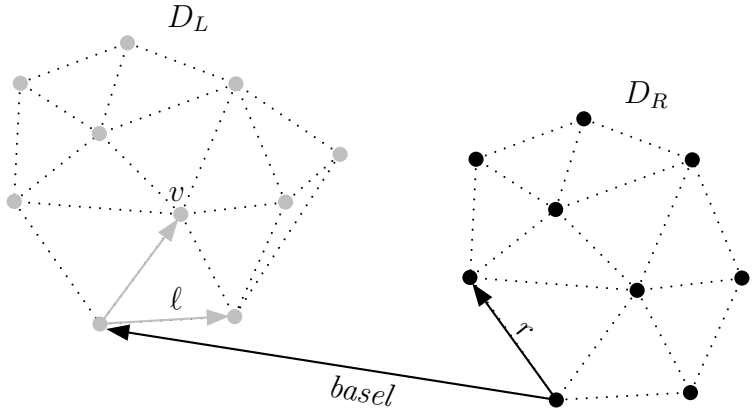
[final summarizing paragraph...]



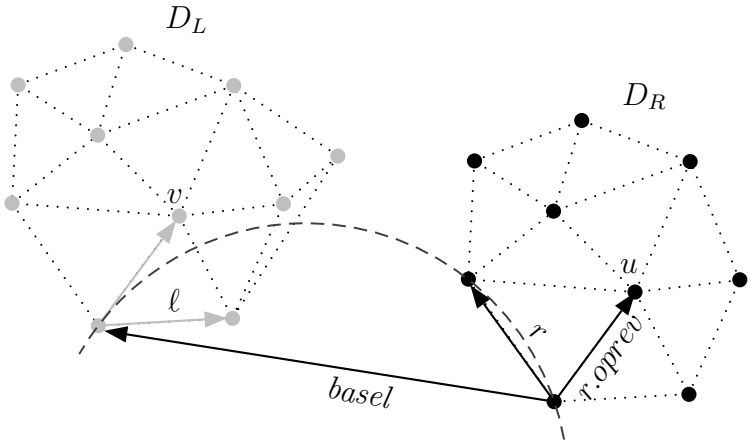

(a) First edge, *basel*, of $D_P$ and $\ell$, candidate edge in $D_L$. Note that $\ell$ is considered to be valid because CCW($\ell.twin.source$, $basel.twin.source$, $basel.source$) is true. That is to say, $\ell.twin.source$, $basel.twin.source$, $basel.source$ is a counterclockwise ordering.



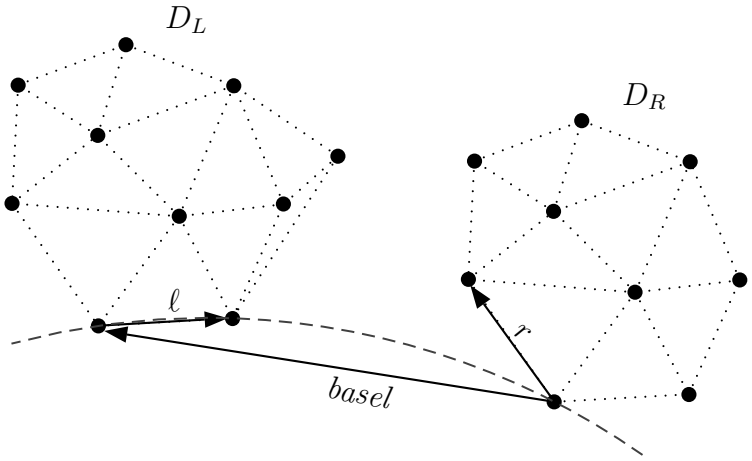

(b) Evaluating whether or not $\ell$ will be in $D_P$. The dashed arc is a portion of the "rising bubble" which we say has *witnessed* that $v$ in not in the circle defined by $\ell.twin.source$, $basel.twin.source$, and $basel.source$.



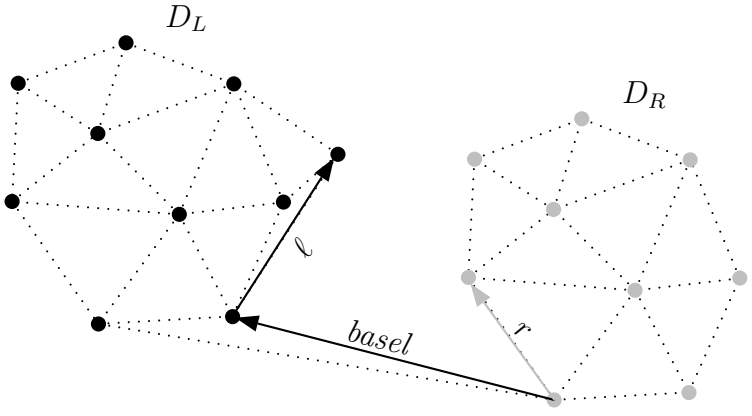

(c) Similarly, now we evaluate whether or not $r$ will be in $D_P$. Note that it is a valid edge, since $r.twin.source$, $basel.twin.source$, $basel.source$ is a counterclockwise ordering.



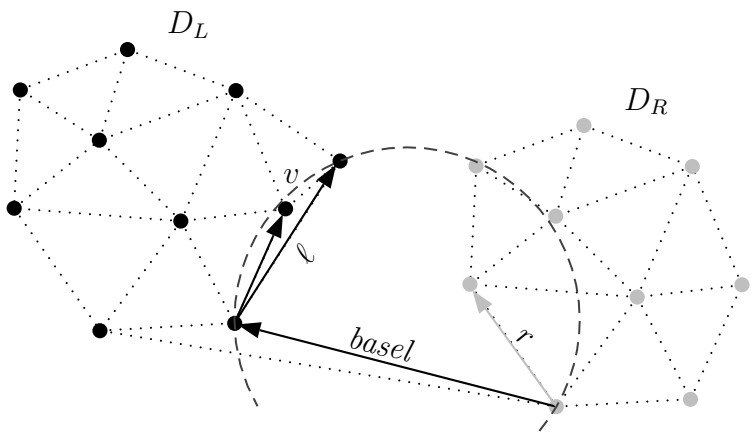

(d) Evaluating whether or not $r$ will be in $D_P$. The dashed arc is a portion of the "rising bubble" which we say has *witnessed* that $u$ in not in the circle defined by $r.twin.source$, $basel.twin.source$, and $basel.source$.



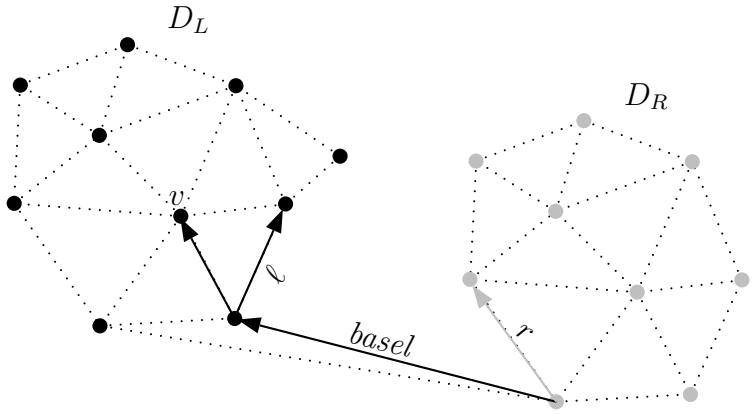

(e) Since $\ell$ and $r$ are both valid we perform InCircle($\ell.twin.source$, $\ell.source$, $r.source$, $r.twin.source$). Note that $r.twin.source$ is not in the circle formed by $\ell.twin.source$, $\ell.source$, $r.source$, so InCircle returns false.



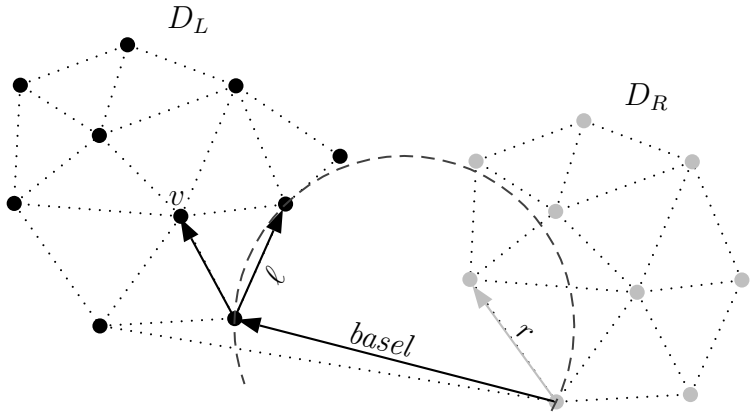

(f) Due to the result of the InCircle test, we update $basel$'s destination to $\ell.twin.source$. We also update $\ell$ to $basel.twin.onext()$.



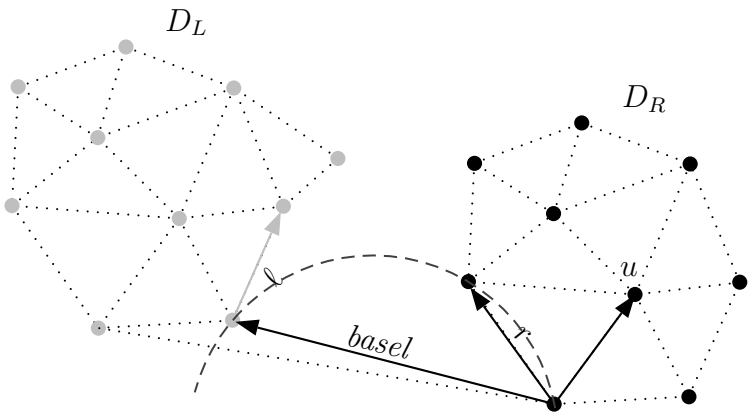

(g) We continue this process noting that $\ell$ is a valid edge. Because this is the case, we check to see if $v$, that is, $\ell.onext().twin.source$, is contained in the circle represented by the dashed lines. It is.



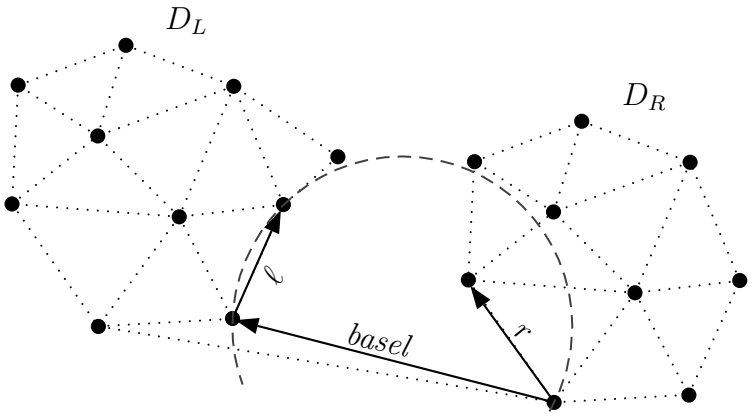

(h) Because $v$ is contained in the circle, we need to remove $l$ and set the new value of $l$ to $l.onext()$, and making the analagous change for $v$. Note $l.onext()$ is also shown as a directed edge.



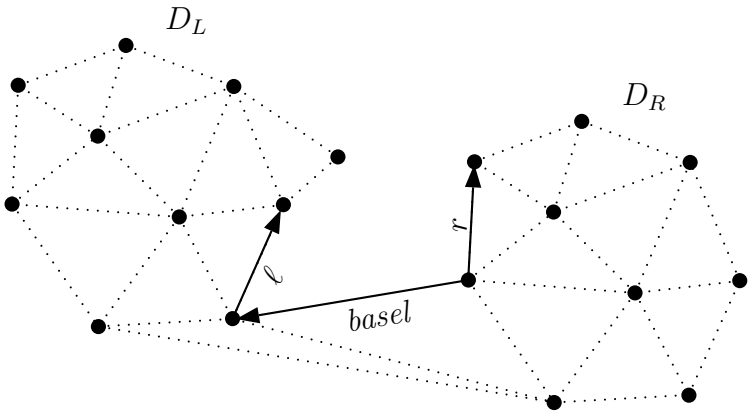

(i) We perform the same test again, checking to see if $v$ is contained in the dashed circle. This time, it is not, so we know that $\ell$ will be in $D_P$.



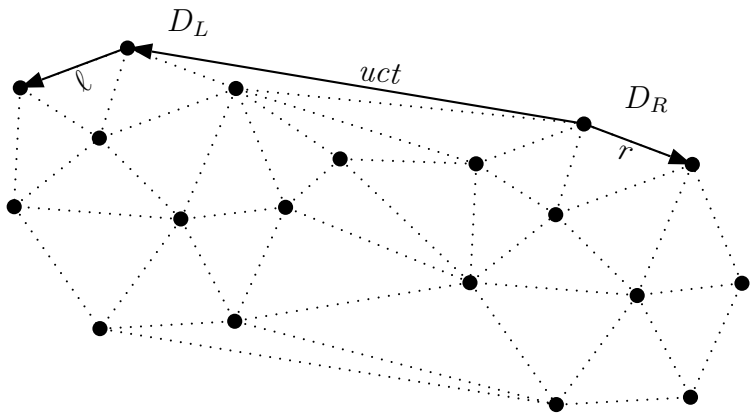

(j) We consider the right side again, noting that $r$ is still valid. We may note that $u$ has not been reassigned. However, since *basel* has changed, we need to check that $u$ is not in the circle formed by *r.twin.source*, *basel.twin.source*, and *basel.source*. Here, it is not.



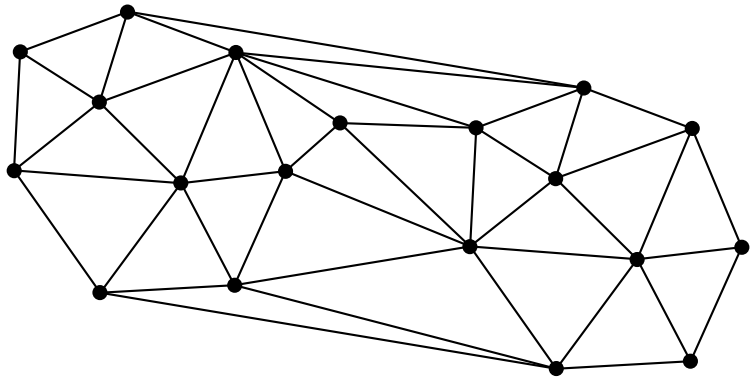

(k) Again, since $\ell$ and $r$ are both valid we perform InCircle(*ℓ.twin.source*, *ℓ.source*, *r.source*, *r.twin.source*). Note that *r.twin.source* is in the circle formed by *ℓ.twin.source*, *ℓ.source*, *r.source*, so InCircle returns true.




(l) Because the InCircle test returned true, we update *basel* so that its source is now *r.twin.source*. We also add the old *basel* edge to $D_P$. Finally, we update $r$.




(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 3.16: Knitting together $D_L$ and $D_R$ to form $D_P$

### 3.1.2 Divide and Conquer Algorithm Pseudocode

DelDivConq and its supporting procedures DelMerge and DelLCT provide a pseudocode implementation of the divide and conquer algorithm presented in Guibas and Stolfi's paper, Guibas & Stolfi, 1985. The corresponding procedure for the dual is described in Shamos and Preparata's text, Preparata & Shamos, 1985.

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. However, using their structure not only gives us constant time access to both the Voronoi and Delaunay diagrams on $P$, it also allows us to dive a little more deeply into the topological underpinnings of subdivisions and manifolds. DelDivConq computes the Delaunay triangulation on a set of sites; however, it doesnt return return the entire edge structure. When we return $D$ at any point in DelDivConq, 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** DelDivConq

**Require:** Set of sites, $P$, which are in *general position*

```
1:  procedure DelDivConq(P)                                   ▷ Construct D_P
2:      if |P| is 2 then                                      ▷ First Base Case
3:          p1, p2 ← Sort(P)
4:          edge ←MakeEdge
5:          edge.org, edge.dest ← p1, p2
6:          return edge, edge.twin
7:      else if |P| is 3 then                                 ▷ Second Base Case
8:          p1, p2, p3 ← Sort(P)
9:          e1 ←MakeEdge
10:         e2 ←MakeEdge()
11:         Splice(e1.twin, e2)
12:         e1.org ← p1
13:         e1.dest ← p2; b1.org ← p2
14:         e2.dest ← p3
15:         e3 ←Connect(e2, e1)
16:         if CCW(p1, p2, p3) then                           ▷ Make a triangle
17:             return e1, e2.twin
18:         else if CCW(p1, p3, p2) then
19:             return e3, e3.twin
20:         else                                              ▷ Case where are sites collinear
21:             return e1, e2.twin
22:         end if
23:     else                                                  ▷ Recursive Case
24:         x ←Median({x | (x, y) ∈ P})
25:         L, R, ← Split(P, x)                               ▷ Partition P according to x
26:         D_L ← DelDivConq(L)
            D_R ← DelDivConq(R)
27:         D ← DelMerge(D_L, D_R)                            ▷ Merge sub-diagrams
28:         return D
29:     end if
30: end procedure
```

As we saw in section 3.1.1, the DelDivConq is a recursive procedure which takes a set of sites $P$ in general position. We will also assume that not all sites are collinear, as special case we covered in section 1.6.2. Given $P$, DelDivConq computes Del($P$) by splitting the problem into subproblems and then merging the results pairwise. Like all recursive procedures which intend to end, DelDivConq has base cases.

Lines 2 to 6 outline the first base case. When the number of sites we are hoping to triangulate is equal to two, then the triangulation is a line segment. To determine the direction and orientation of the directed edge corresponding to this line segment, we first make a call to Sort(P) , which returns the set of sites sorted according to x-coordinate. In line 4, assign *edge* to the result of MakeEdge. MakeEdge

creates a generic, unconnected quad-edge. Recall the various attributes and functions on quad-edges covered in **??**. At this point, *edge* has the following properties: $edge.org \neq edge.dest$, $edge.left = edge.right$, $edge.lnext = edge.rnext = edge.twin$, and $edge.onext = edge.oprev = e$. Now that we have made a single edge, *edge* we return the pair *edge*, *edge.twin*. Since the diagram consists of just this single edge, *edge*, *edge.twin* are the oppositely oriented leftmost and rightmost edges we should return.

A slightly more complex base case, where there are only three sites in $P$, is outlined in lines 7 to 21. Our goal in this case is to try to form a triangle 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 11 is a mysterious one. Splice takes as its arguments two quad edges, $a$ 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 11), $e1$ and $e2$ are totally disconnected from the subdivision, so the effect of Splice is that it sets $e1.twin$ to be in the same ring as $e2$. In this particular case, if we were working with a doubly connected edge list, this could be done 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) in the rest of this DelDivConq.

After updating the source and destination vertices of the edges, we perform the CCW test to determine the orientation of the triangle. This also allows us to return the correct pair of edges. 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.

The final chunk of pseudocode, lines 23 to 28, handle the recursive case. When there are more than three sites in $P$, we need to break the problem down further. We call Median, which takes set of the x-coordinates from the sites in $P$ and returns the median value. In line 25, we pass this value— along with $P$— to another helper function, Split. Split returns two sets of sites, $L$ and $R$ (where $L = p \mid p.x \leq x$). Then, we make the calls to DelDivConq on both $L$ and $R$. Finally, we merge the results with DelMerge and return the result.

**Algorithm 6** DelMerge

```
 1: procedure DelMerge(D_L, D_R)
 2:     ldo, ldi ← D_L
 3:     rdi, rdo ← D_R
 4:     basel ← DelLCT(ldi, rdi)
 5:     if ldi.org is ldo.org then
 6:         ldo ← basel.sym
 7:     end if
 8:     if rdi.org is rdo.org then
 9:         rdo ← basel
10:     end if
11:     loop
12:         ℓ ← basel.sym.onext
13:         if Valid(ℓ) then
14:             while InCircle(basel.dest, basel.org, ℓ.dest, ℓ.onext.dest) do
15:                 next_ℓ ← ℓ.onext
16:                 DeleteEdge(ℓ)
17:                 ℓ ← new_ℓ
18:             end while
19:         end if
20:         r ← basel.oprev
21:         if Valid(ℓ) then
22:             while InCircle(basel.dest, basel.org, r.dest, r.oprev.dest) do
23:                 next_r ← r.oprev
24:                 DeleteEdge(r)
25:                 r ← new_r
26:             end while
27:         end if
28:         if not Valid(ℓ) and not Valid(r) then          ▷ basel is UCT
29:             exit loop
30:         end if
31:         if not Valid(ℓ) or
                (Valid(r) and InCircle(ℓ.dest, ℓ.org, r.org, r.dest)) then
32:             basel ← Connect(r, basel.sym)
33:         else
34:             basel ← Connect(basel.sym, ℓ.sym)
35:         end if
36:     end loop
37:     return ldo, rdo
38: end procedure
```

DelMerge takes two valid Delaunay triangulations and knits them together. Recall that $D_L$ and $D_R$ are each pairs of edges. In particular, keep in mind that *ldi* is the convex hull edge coming out of the rightmost vertex in $D_L$, going in the

counterclockwise direction. *rdi* is the convex hull edge coming out of the leftmost vertex in $D_R$, going in the clockwise direction. We know that all of the vertices in $D_L$ are left of the vertices in $D_R$, so *ldi* is to the left of *rdi*. Furthermore, we know that the convex hulls of $D_L$ and $D_R$ are not overlapping. As we saw in section 3.1.1, the lower common tangent of $D_L$ and $D_R$ is a convex hull edge, which we know must be in the merged result, $D$.

This is as good a place to start as any. We begin by calling DelLCT in line 4, which returns *basel*, the appropriately directed and oriented quad-edge connecting the vertices of the lower common tangent. Conceptually, our goal is to walk the respective hulls of the adjacent diagrams until we find the 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. This is accomplished in line 4 where we update *ldi*'s value to *ldi.lnext*. Recall that *ldi.lnext* points to the next edge which has the same left face as *ldi*. Since the left face of *ldi* is the region exterior to the convex hull, *ldi.lnext* 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 the right triangulation. Similar to our process with *ldi*, we make the corresponding note that *rdi.rnext* is the next edge with the same right face as *rdi*, and is the next hull edge on $D_R$.

We loop in this fashion, advancing *ldi* and *rdi* in the appropriate manner until both CCW tests fail. Note that when both tests fail, the line determined by the endpoints of *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, *basel*, and return it in line 9

Stepping back into DelMerge, we can now begin the journey upward: we're at the "rising bubble" stage, lines 11 through 36. Figure 3.16 on page 35 illustrates this process. The idea behind this loop is to begin with $\ell$ and $r$, which are edges on hulls of their respective triangulations. At the first iteration of the loop, we set $\ell$ to the next outgoing edge whose source is *basel.twin.org*, meaning that $\ell$ 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, $\ell$ and $r$ are always set to the convex hull edge whose respective sources are shared with the "uppermost" known cross-edge that the time, *basel*. 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 13 we check to see whether or not $\ell$ is valid with Valid. Recall that Valid is a helper function that tells us whether or not an edge is above *basel*.

If $\ell$ is a valid edge, we enter a loop in line 14 which we assess whether or not $\ell$ can remain in the merged diagram. This is accomplished with an InCircle test. Recall that *onext* points to the next edge (in counterclockwise direction) whose source is *ldi.org*. If *ldi.onext.dest* is contained in the circle defined by the endpoints of *basel* and the destination of $\ell$, 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 $\ell$ to point to the next edge with the same source.

When we exit this loop, we know that $\ell$ will be in the merged result. We wish to do the same for $r$; this is accomplished in the loop beginning at line 22. The only change is that if we fail the InCircle test, we remove the edge $r$ and update $r$ to $r.oprev$, where *oprev* is the next edge in the clockwise direction with the same source as $r$.

Finally, we have reached line 28, where we can say that we have a grasp on two edges, $\ell$ and $r$, which will be in the merged result. Note that since we might have deleted some edges, $\ell$ 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 *basel.org* or *basel.dest*. We also know that the other endpoint of the new edge will be either *r.dest* or $\ell$*.dest*. There are two possible triangles: *basel.source*, *basel.dest*, $\ell$*.dest*; and *basel.source*, *basel.dest*, *r.dest*. The if statements following line 28 determine and create the appropriate cross edge. Precisely, if $\ell$ and $r$ are not valid, then neither of them are above *basel*, so we must be at the upper common tangent, and thus have no need to create a cross edge. In 31 we check to see if one or both of the following is true: $\ell$ is not valid[14]; or $r$ is valid, and the triangle formed by $\ell$*.dest*, $\ell$*.org* and *r.dest* contains a site (namely, *r.oprev.dest*). Since this triangle would not be locally delaunay, we create the triangle *r.dest*, *basel.dest*, *basel.org* by adding an edge with Connect in line 32. We update *basel* to be this edge, which is the "uppermost" edge spanning $D_L$ and $D_R$. If neither of these cases holds, then we know *basel.source*, *basel.dest*, $\ell$*.dest* to be a triangle which is locally Delaunay, and we return the edge reflecting that in line 34.

**Algorithm 7** DelLCT

```
 1: procedure DelLCT(ldi, rdi)
 2:     loop
 3:         if CCW(rdi.org, ldi.org, ldi.dest) then
 4:             ldi ← ldi.lnext
 5:         else if CCW(ldi.org, rdi.dest, rdi.org) then
 6:             rdi ← rdi.rnext
 7:         else
 8:             basel ←Connect(rdi.sym, ldi)
 9:             return basel
10:         end if
11:     end loop
12: end procedure
```

## 3.2 Some Final Thoughts

[14]Meaning that $\ell$'s destination is an endpoint of *basel*, so we can't create a triangle with *basel.source*, *basel.dest*, $\ell$*.dest*.

# Appendix A

# (The Post Office Problem Re-framed)

Here you are, in a forest; you are studying trees. You are lying in the undergrowth and the shade is covering your face. It smells like pine needles and your fingers are sticky with sap. There is another tree about a dozen feet away. We all know exactly how far away it is, but dozen sounds more prosaic. so it's a dozen feet away, and why is that? [area availiable to a tree]

You're walking the blocks of new york city and the wind is cold. Where is the nearest subway entrance? People are walking past you though, really, you're just shuffling. Where is the nearest subway with an N train? Is it a local or express line? Are you hungry, maybe you're hungry. Or maybe you're the one selling hot dogs. yes. hot dogs in new york city. where should you put your hot dog cart?]

# References

de Berg, M., van Kreveld, M., Overmars, M., & Schwarzkopf, O. (1997). *Computational Geometry: Algorithms and Applications*. Germany: Springer-Verlag.

Descartes, R. (2004). Descartes: The world and other writings. Cambridge, United Kingdom: Cambridge University Press.

Guibas, L., & Stolfi, J. (1985). Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, *4*(2), 74–123.

Preparata, F. P., & Shamos, M. I. (1985). *Computational Geometry: An Introduction*. New York: Springer-Verlag.