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] [this is a math problem, this is a computational geometry problem, this is a spotlight on a common problem, this is related to a lot of other cool problems, this is a problem about space, this is an apology (explanatory), this is a survey of approaches to solving this problem as efficiently and elegantly as possible, this is building off of current implementations to examine the benefits of a) multicore / threaded and / or b) gpu accelerated implementations, this is about re-learning C, this is about wrangling threads and hating CUDA]

Introduction

"... 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 The World

René Descartes spent four years writing *The World*, and abandoned it in 1633 after the Roman Inquisition condemmed Galileo and his heliocentric theory. It remained largely unpublished for thirty-one years after its abandonnment, 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 unpreturbed at the centers of endlessly swirling corpuscules. 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 which do not break apart, but rather, 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 level of agigation associatated 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' cosmology most directly likened to a Voronoi diagram through his description of the heavens as a whole. As 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).

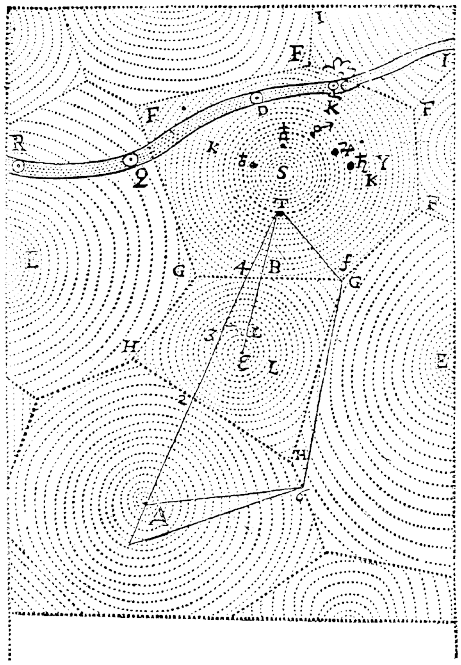


Figure 1.1: Descartes' Heavenly Regions

For those who are already familiar with the Voronoi diagram, it is easy to recognize the visual similarities between Descartes' graphical representation of the heavens and the Voronoi diagram. Perhaps more interesting, however, are the similarities between what he understands the diagram to be and the definition of a Voronoi diagram. As we mentioned earlier, Descartes believes that every star has a heaven associated with it. For example, the matter in the region defined as HGGH is the heaven which rotates

around ϵ. Similarly, the heaven of star S is precisely the matter in the region FGGF. When we define the Voronoi diagram of a set of sites, we will note that every site has a corresponding region. The *firmament*, as Descartes calls it, can be readily adapted to the Voronoi diagram with a shift in jargon: the firmament which separates the heavens from eachother are referred to as *edges* in the Voronoi diagram. Should we have the desire to squint, we might even be able to glimpse an oblique reference to a Voronoi vertex: Descartes writes of the matter which lives at the intersection of three or more circles, at what appear to be "corners" of the firmament, it is likely smaller, and less agitated than the matter surrounding it.

1.2 The Apocryphal Tale of John Snow

If my dear Reader will continue to indulge, fast-forward two hundred years. . . to 1854. It is summer in London, England. Beginning in late August, the Soho district of London experienced a cholera outbreak. The severe dehydration caused by the disease lead to dozens of deaths in a matter of days, and the numbers rose to over five hundred in the first two weeks.

"Dr. Snow's maps illustrate the highest use of cartography: to find out by mapping that which cannot be discovered by other means or, at least, not with as much precision."

The Voronoi Diagram

Suppose you are hoping to send a piece of mail via post office. In this day and age, it's uncommon to send a piece of mail, and you might ask yourself "Where is the nearest post office?"

But why is the post office *there*?

To answer that question, assume the perspective of a city planner whose task is to determine where the next post office should located. The city planner might work under the assumption that people go to the post office closest to where they live. Roughly, the planner determines the areas of service for each post office, which consists of dividing the city into regions whose occupants are closer to the post office in that region than any other office. The area of service, for a particular post office, corresponds directly with the Voronoi cell of that site (the post office). We call this cell, or a region.[1]

2.1 Formalizing the Post Office Problem

One way to understand the Voronoi diagram is via a rigorous exploration of subdivisions, manifolds, and edge algebras.[2] However, for the purposes of this thesis we begin with a more gentle introduction. My goal is to offer a description of the Voronoi diagram that relies on its geometric properties in the Euclidian Plane. The diagram is the subdivision induced under the Voronoi assignment model, which consists in assigning every point in the plane to the site nearest to it. To construct a Voronoi diagram, we need a set of sites, P, and way to measure the distance between two sites, p and q, denoted $\delta(p, q)$. For the purposes of this thesis, we consider the Voronoi diagram in the plane, sites are defined as points, and the distance between any two points is the Euclidian distance.

Let $P := \{p_1, p_2, ..p_n\}$ a set of n distinct points – each one constituting a site – in the plane. The Voronoi diagram of P the subdivision of the plane into n regions where for each region V_i in the set of regions $V := \{V_1, V_2, ..V_n\}$, V_i is the region associated with site p_i. Further, the region V_i is the set of points in the plane such that each point in V_i is closer to p_i than it is to any other site in P. Formally, given a plane S, a region or cell in a Voronoi diagram is the set of points in the plane which

[1] If the aforementioned example doesn't resonate with you, I'd encourage you to peruse appendix A, where I've included a plethora of other examples with the intent of motivating the problem from different perspectives, disciplines, and industries.

[2] If this interests you, skip this thesis and read Guibas and Stolfi's paper.()

satisfy:

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

For our purposes, $S = \Re^2$, the Euclidian plane in two dimensions, and $\delta(p, q)$ is the Euclidian measure of distance in $\Re^2$.

We can also think of the region V_i as the (possibly unbounded) convex polygon which is the result of intersection of $n-1$ half-planes. Let's unpack that a little bit. Given two sites p_i and p_j, imagine the line, l connecting them. [graph] At the midpoint, m, of l, imagine a line b, purpendicular to l and passing through m. Note that b is the perpendicular bisector of l. Then b splits the plane into two half-planes. Define the half-plane containing p_i with $h(p_i, p_j)$; similarly, define the half-plane containing p_j with $h(p_i, p_j)$. To connect this notion back to our first description of a Voronoi region, note that 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. 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 closer to p_i than they are to p_j or p_k. If we performed this intersection over all $n-1$ half-planes, we would have described the Voronoi region of p_i exactly. So we can also describe the region as the intersection of a set of half-planes:

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

Because the half-plane is a convex set[3] (or region), and the intersection of convex regions is also convex, we can say with confidence, nay, with certainty, that the Voronoi region is convex. In section 2.2 we describe an algorithm to construct the Voronoi diagram using an approach which relies heavily on intersection of half-planes and convex regions.

To summarize, 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. 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. The Voronoi diagram of a set of sites P, Vor(P), is the collection of all of these regions. We know that Vor(P) will have n regions; and we can put an upper bound on the number of edges and vertices it has by recognizing that a Voronoi diagram in $\Re^2$ is *nearly* a planar graph. We make the Voronoi diagram of P into a planar graph by introducing a vertex "at infinity". Then, we assign the vertex at infinity to be the source (or destination, when appropriate) to the half infinite edges (that is to say, all edges without two endpoints).

Note that the size of the Voronoi graph is linear in the number of sites. *Euler's Formula* states that any planar graph with v vertices, e edges, and f faces (regions,

[3] A set is convex if the line segment connecting any two points in the set is also in the region.

for our purposes) the following is true: $v - e + f = 2$. Applying this to Vor(P) with an infinite vertex, which has n regions, v vertices, and e edges we have:

$$(v + 1) - e + n = 2.$$

Since every edge has exactly two endpoints and every (and every vertex is of degree at least 3), we can say that $2e \geq 3(v+1)$. Together with Euler's formula, this implies that for a $n \geq 3$, the number of vertices is at most $2n - 5$ and the number of edges is at most $3n - 6$.

2.2 Intersection of Half-Planes: A Naïve Approach

The structure of the planar Voronoi diagram lends itself to relatively simple algorithm for its construction. Using the notions and terminology from above, we can cobble together a naive algorithm to compute the Voronoi diagram of a set of sites. Intuitively, the Voronoi diagram of a set of sites P, Vor(P), can be found by computing each region V_i and defining Vor(P) to be that set of regions.

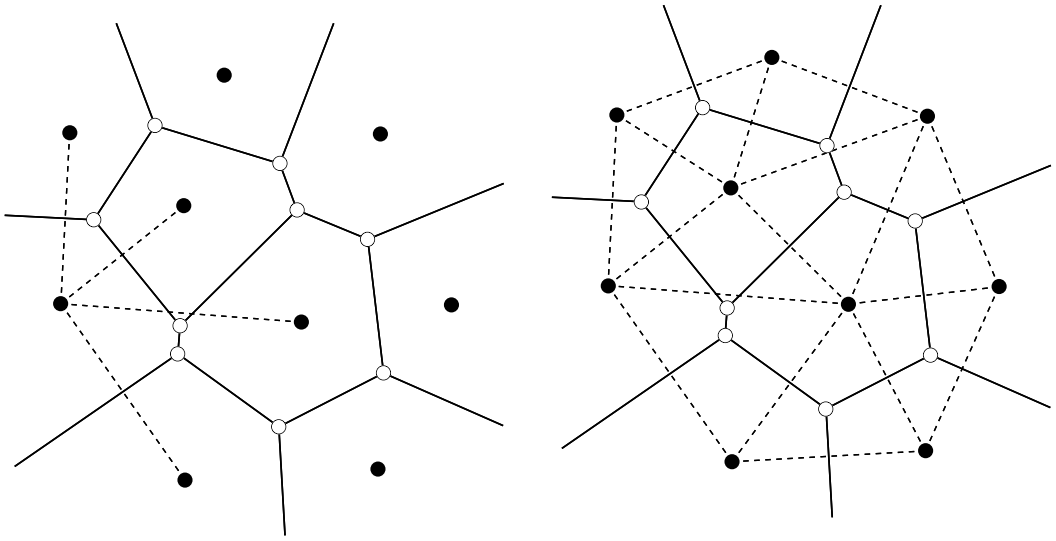
We've seen that the Voronoi region is a convex polygonal region. Futhermore, if $|P| = n$, then no region can have more than $n - 1$ edges. Recall that an equivalent way of describing a Voronoi region is as the intersection of a set of half-planes. We can construct the region for a specific site by taking the intersection of this set of half-planes. Each region, then takes $\mathcal{O}(n \log n)$ time. A region needs to be computed for each of the n sites, so computing the Voronoi diagram with this approach would take $\mathcal{O}(n^2 \log n)$ time.

2.3 The Voronoi Diagram and its Dual

While the Voronoi diagram can be computed directly with relative- though certainly not always computatational- ease, it can also be computed indirectly, that is, through the construction of its dual, the *Delaunay triangulation*. 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. Given a Voronoi diagram on P, 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 2.2a. Do this for all the sites as in figure 2.2b. If we assume that Vor(P) contains no degeneracies, then the resulting diagram is the Delaunay Trangulation, Del(P), as shown in figure 2.2.

Note that we are being a bit cavalier with this description of the Delaunay triangulation. If four or more sites were co-circular, our result would be a Delaunay subdivision, which could be quickly modified (via the addition of edges, if necessary) to be a Delaunay triangulation. Furthermore, the Delaunay triangulation of a Voronoi diagram whose sites are in general position is unique.

Now, 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*. This is the smallest



(a) A site connected to its adjacent sites

(b) All sites connected to their adjacent sites

Figure 2.2: Voronoi Diagram (Solid) and Delaunay Triangulation (Dashed)

convex region containing P, that is roughly, the intersection of all of the convex sets containing P. We will define this set as the smallest convex set generated by taking conbinations of points of P. Precisely, given points q_1,... q_k, a *convex combination* of those points is defined as $\Sigma\alpha_i q_i$ where the α_i's are non-negative and $\Sigma\alpha_i = 1$. So, for our point set P we are interested in the convex hull 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 2.3 illustrates the convex hull of the same point set used in figure 2.2a and **??**. This region of the plane is best understood in terms of a complete description of its boundary.[4] 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.

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 meets 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.

So, given a triangulation, as in figure 2.4, 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 2.4, the triangles which share edge e are Δabd

[4]See figure 2.3

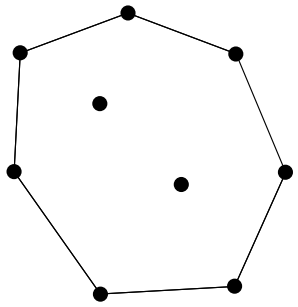


Figure 2.3: Convex hull of a set of points

adn Δbcd. Because a does not lie in the circumcircle of Δbcd, shown as a dashed line in figure 2.4, 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*.

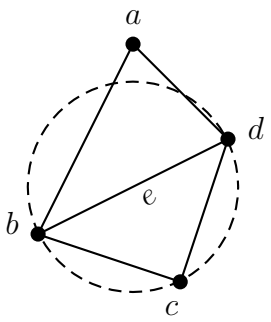


Figure 2.4: Testing the delaunayhood of a small triangulation

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. In section 2.5 we will explore some of the convenient and interesting geometric consequences of a triangulation with the Delaunay property.

2.4 Applications of Voronoi and Delaunay Diagrams

Beyond the intrinsic allure of the Voronoi diagram and its stright 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 preprocess 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 2.3, 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 auxilliary storage.(Preparata & Shamos, 1985; **?**)

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

2.5 Properties of Voronoi and Delaunay Diagrams

This section summarizes the terminology from sections 2.1 to 2.4

2.6 How do we keep track of this?

Before we take the next step of our journey together, dear Reader, a note on storage: there are a couple different ways to store a Voronoi diagram. The authors of algorithms will occasionally suggest data structures which complement the algorithm approach and are space efficient. However, it is often at the discretion of the programmer to choose the structures most appropriate to their application's needs. In sum, these choices can range from devilishly clever [ref Guibas and Stolfi] to painfully ham-fisted. It is important to note that by augmenting the Voronoi diagram with a

[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 $\Re^2$.

point at infinity, we can treat it as a planar graph. Consequently, any data structure fit for use with a planar graph can be used to store and create a Voronoi diagram.

Depending on the approach, we might find it wise to use intermediary data structures, such as priority queues or red-black trees, as we will discover in section 3.1. However, what we wish to focus on in this section are the structures which hold the culmination of all this work: the completed Voronoi (or Delaunay) diagram. And this much is true for any algorithm one might encounter: given a set of n unique sites, $P := \{p_1, p_2, ..p_n\}$, the output will be a handful of Voronoi vertices, V, and a bundle of Voronoi edges, E.

[to do: weave this in here..... use it to motivate these structures.... The mysterious *point at infinity*- discussion of which I have deliberately skirted thus far- now demands explanation?(here? now? really?) We might understand the Delaunay diagram (or, if you're interested in a challenge, the corresponding Voronoi diagram) as bundles of edges, faces, and vertices. As it turns out, the incidence relationships between these precisely define the topological structure of the diagram. We may take for granted that the Voronoi and Delaunay diagrams are topologically equivalent to an undirected graph embedded in the Euclidian plane. When we add a *point at infinity*, we allow ourselves to consider these diagrams as planar graphs embedded on a sphere instead. Though it doesn't immediately seem to simplify things, it allows us to use the infinite elements of the diagrams (edges and faces) the same way we use the finite elements.
]

2.6.1 Directed Edges

[some kind of preamble about how a variety of linked edge structures exist and here is a common one...] The Figure 2.5 provides a visual guide for the terminology we use to traverse planar graphs and to refer to its edges and vertices. Each edge has a notion of the *face* to its left. We might think of a *face* as the portion of the plane that any cycle of directed edges encloses. An edge also has an idea of the *next* edge in the graph and its *twin*. [something about the twin, something about orientation] In figure 2.5, for example, given e, we would access f in terms of e with the expression(?) $e.next.twin.face$.

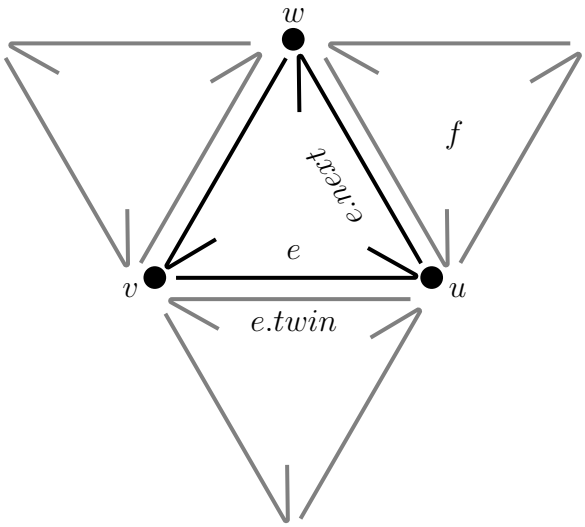


Figure 2.5: Half Edge Terminology

2.6.2 Quad-Edge

Guibas and Stolfi's quad-edge data structure simultaneously represents an embdedding of a planar graph, its dual, and its mirror image. It was, as we will see in section 3.2, the result of an attempt to separate more distinctly the topological properties of the diagram from the geometric properties of the diagram. They note that manipulation of a Delaunay triangulation is simpler than performing the corresponding operations on a Voronoi diagram. Guibas and Stolfi's quad-edge data structure is the motivation for their in-depth discussion of embeddings of subdivisions on orientable manifolds, and though a thorough summary of their discussion is (perhaps) outside the scope of this paper, I will note, briefly, they ways in which their quad-edge data structure leverages their findings. Their quad-edge structure allows one to hop between the primal, dual, and mirror-image diagrams in constant time. It bears a strong resemblance to the winged-edge and half-edge data structures, but goes a step further, and takes advantage of the fact that an edge sits at the boundary of exactly two faces and two vertices. Moving between the primal and the dual, then, is a simple matter of exchanging the conventions around the representation of faces and the vertices. (there must be a better way to put this). This is achieved through their development of an *edge algebra*, which captures the topological properties of a subdivision (and allows them to refine the set of edge functions which can be used to traverse the mesh). [DIAGRAM]

Constructing the Voronoi Diagram

In section 2.1 we saw a naive algorithm for computing the Voronoi diagram of a set of n sites. The time complexity of the half-plane intersection algorithm, as we discovered in section 2.2, is $\mathcal{O}(n^2 \log n)$. Though it provides an intuition of the diagram, 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 sweepline 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 this chapter, we will examine two classic algorithsm to compute the Voronoi diagram: a divide and conquer algorithm, presented by Guibas and Stolfi [ref]; and Steven Fortune's sweepline algorithm.

As it turns out, 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 $\theta(n \log n)$. (Cool fact: Asymptotically, the complexity of the constructing the diagram is no more difficult than the task of finding a region.)

3.1 Fortune's Algorithm: A Sweepline Approach

In the 1980's Steven Fortune proposed a sweepline algorithm with asymptotic running time $\mathcal{O}(n \log n)$ and using $\mathcal{O}(n)$ space. Fortune's algorithm makes use of a sweepline, a horizontal line beginning at the top of the plane and descending to the bottom of the plane. The general strategy of the algorithm is to use a line to scan the plane, updating information about the structure of the diagram whenever the sweepline intersects a point of interest. Some of these points of interest are known in advance, and others arise during the construction of the diagram. With a bit of planning and the clever choice of a quad-edge data structure, the task of constructing the Voronoi diagram with this approach immediately yields its dual, the Delaunay triangulation. [ref guibas]

3.1.1 Conceptual Overview

Fortune's algorithm uses a few key concepts and structures to support the construction of the Voronoi diagram. To gain an understanding of how to algorithm works, it's easiest to apply it to a set of sites. Let $P := \{p_1, p_2, ..p_n\}$ be the a set of n distinct points in $\Re^2$. Then, define the sweepline s to be a horizontal line in $\Re^2$. Note that conceptually, s begins at the "top" of the plane, but in practice, it is sufficient that s be initialized with a y-coordinate greater than the site with the largest y-coordinate.

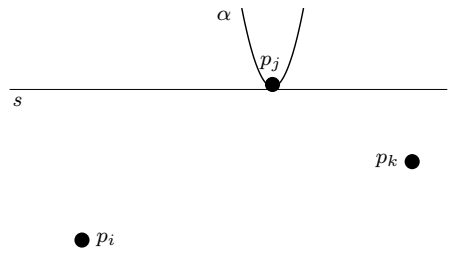
An astute reader might wonder how exactly we construct the Voronoi diagram with nothing but a set of sites and a vague notion of a line that falls down the plane. After all, there's no obvious method by which the intersection of the sweepline with a point of interest might yield a Voronoi vertex or edge.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

Herein lies the elegance of Fortune's algorithm. Consider the portion of the plane that the sweepline has already encountered, this portion constitutes a closed half-plane, s^+. If a site exists in s^+, 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 sweepline s encounters a site p_i we call this a *site event*. We consider the locus of points which are closer to p_i than to s. The set of points equidistant to s and p_i form a parabolic arc. This arc, denoted β_i, is added to a a sequence of parabolic arcs called the *beachfront*. We will refer to the beachfront as $\beta(P)$. The crucial observation is that the points above $\beta(P)$ will be assigned to a site already encountered by s.

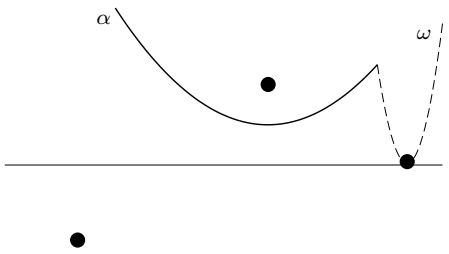
[Make sure there's a clear difference between the notation used for the parabola and the notation used for an arc]. To understand how the beachfront plays a role in constructing the Voronoi diagram, let's consider times where the structure of the beachfront changes. New arcs are added to the beachfront whenever a site event occurs. At the moment where the new arc, β_i, is inserted into the beachfront, it is a simple vertical line (or a parabola of no width). If there is an arc above β_i, β_i will only intersect it at one point since the beachfront is x-monotone. To insert β_i into the beachfront, we split the arc above it into two parabolic arcs and insert β_i in between the split arcs. We record the points where β_i intersects with the arcs adjacent to it, and we refer to these points as *breakpoints*[6] 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 β_i 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 sweepline descended, they would each trace out the Voronoi edges. So we know that arcs get added to the beachfront whenever a site event occurs, in fact, the *only* time an arc gets added to the beachfront is when a site event occurs. [And we can prove that!]

The second way the beachfront's structure changes is when an arc disappears. The

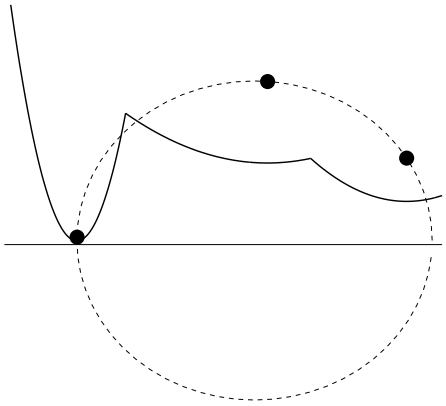
[6]Note that this means that the right breakpoint of β_i is the same as the left breakpoint of the arc which is adjacent on the right. Similarly, the right breakpoint of the arc adjacent to β_i on the left is equal to β_i's left breakpoint.



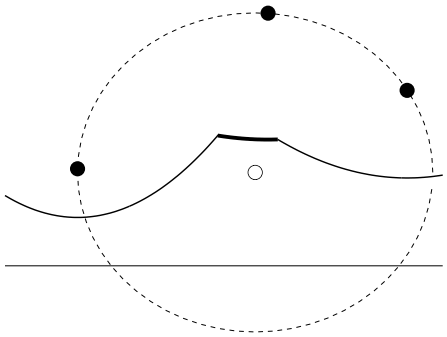
(a) Handling a Site Event



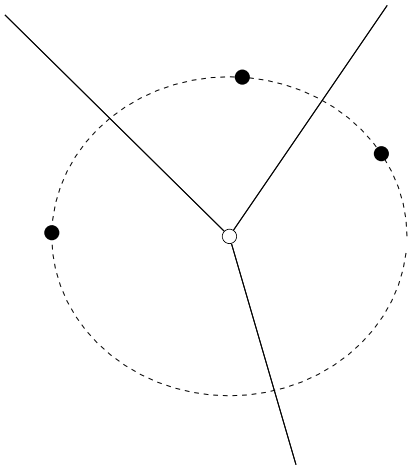
(b) Adding an arc to the beachfront



(c) Add a new Circle Event



(d) Circle's center marks Voronoi vertex (arc to be removed shown in bold)



(e) Handling Circle Event: creating Voronoi edges

Figure 3.6: Constructing the Voronoi Diagram of three sites.

disappearance of an arc coincides with the second type of event, called a *circle event*. We'll define the notion of a circle event by studying a small portion of the beachfront. Consider the arcs α, δ, and γ defined by three sites p_i, p_j, and p_k. Define the circle, C with the three sites p_i, p_j, p_k. [You should probably remind the reader why α and γ are not arcs from the same site, also why the circle contains no sites] Though the reasons aren't immediately obvious, we want to keep track of two points: the center of the circle, q, and the lowest point of the circle, l. If δ 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 sweepline, s, reaches l, the arcs correspoinding to p_i and p_k, α and γ, respectively, intersect at q. [Do I need to explain in more detail why, or can I *just* include a diagram]. At this point, the right breakpoint of α and the left breakpoint of γ become equal to the left and right breakpoints of δ, δ disappears. The edges traced out by the breakpoints of δ meet at q and q becomes a Voronoi vertex.

To summarize, a site event occurs when the sweepline intersects with a site (i.e they both have the same y-coordinate) and a circle event occurs when the sweepline 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. 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.

It's important to note that in this example, we assumed that p_i, p_j, p_k were non-collinear sites, since the circle defined by three collinear points would be a circle of infinite radius (that is to say, a line). Moreover, we glossed over the possibility that C contained any other sites. How would any number of sites on the interior of C affect the circle event itself?

To answer this question, let's first make a brief digression and introduce the second structure that Fortune's algorithm employs: the *event queue*. Armed with an understanding of how site events and circle events effect the structure of the beachfront, we can consider ourselves sufficiently motivated to define a structure that keeps track of these kinds of events. We know that the sweepline 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 sweepline begins, we can simply order the sites by decreasing y-coordinate and initialize the event queue with that infomation. Unlike site events, we don't know exactly when or how many circle events will occur[7], so they complicate the problem insofar as they punctuate the 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 are added to event queue for every triple of consecutive arcs on the beachfront. So when a site event triggers a new arc on the beachfront, we need to check a couple of things, namely, whether or not the new arc a) invalidates one or more circle events already in the event queue; or b) creates one or more circle events. In this manner, handling a site event often has a cascading effect, wherein the event itself doesn't create a circle event, but it might create a situation where one or

[7]Though we can provide an upper bound, see **??**

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. Note that a circle event exists in the event queue if and only if it satisfies a) the property that the circle defined by the sites corresponding to the triple of arcs intersects the sweepline; and b) hasn't already been deleted from the event queue.

[Probably don't need this anymore .... Now that we have an event queue to keep track of the order of circle and site events, we can look back at our initial example. Let p_m be a site located somewhere on the interior of our original circle, C. As s crept downward, the site event corresponding to p_m would have been handled before circle event at l. As a result, an arc β would have been inserted somewhere between α, δ, and γ. More specifically, the arc directly above p_m would have been split into two arcs. The only exception to this would be the edge case where p_m's x-coordinate was exactly the same as either of the breakpoints of δ. In that case, the arc wouldn't have split δ; it would have been inserted directly in between δ and the appropriate adjacent arc. If this were the case, the triple of arcs whose corresponding sites had defined the circle would no longer exist, so the circle event would be invalid. In the former scenario, the new arc would have created at least one new triple of arcs, which would have prompted the creation of more circles, and thus cicle events. It's possible to show [do i have to?] that at least one of these new circle events would have been handled before our original circle event...]

3.1.2 Data Structures and Algorithm Pseudocode

algorithm 1 and its supporting procedures, algorithm 2 and algorithm 3, provide a pseudocode implementation of Fortune's algorithm. The vertex and edge structure for Vor(P) will use a classic doubly connected edge list (DCEL) [do i need to explain this] and we will use a balanced binary search tree (BBST) to maintain the beachfront, β. The event queue, Q, is a plain old priority queue.

Algorithm 1 Sweepline Voronoi Diagram

```
1:  procedure VoronoiSweepline(P)                          ▷ Construct Vor(P)
2:      Q ← P                          ▷ Initialize event queue, Q, with set of sites, P
3:      β ← {}                                  ▷ Initialize beachfront, B, as empty
4:      while Q is not empty do                  ▷ There are more events to handle
5:          e ← Q.pop()                          ▷ Pop the next event off the queue
6:          if e is a site event then
7:              HandleSiteEvent(e)                                  ▷ e is a site p_i
8:          else
9:              HandleCircleEvent(e)          ▷ e is a circle, c_l with low point l
10:         end if
11:     end while
12: end procedure
```

Algorithm 2 Sweepline Voronoi Diagram: Handle Site Event

```
1: procedure HandleSiteEvent(p_i)
       Something compelling
2: end procedure
```

Algorithm 3 Sweepline Voronoi Diagram: Handle Circle Event

```
1: procedure HandleCircleEvent(c_l)
       now blow their minds, this is cool stuff!
2: end procedure
```

3.1.3 Final thoughts on Fortune's Algorithm

Though Fortune's algorithm wasn't the algorithm I spent the most time with on this thesis, it remains my favorite algorithm to witness. Fortune's algorithm revealsmore readily than any other I've encountered- the connection between sorting a set of numbers and computing the Voronoi diagram. [the connection] [perhaps a note on degenerate cases] [a note on space and time complexity]

3.2 Guibas and Stolfi: A Divide and Conquer Approach

Guibas and Stolfi's "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams" 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.[8] As we saw in section 2.6, 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 contstruction and modification of any embedded graph on a two-dimensional manifold.

[as we saw in section 2.6, directed edge data structures, and in particular, guibas and stolfi's quad edge data structure, maintain information about the topological structures of the diagram. we when it comes to the Voronoi and Delaunay diagrams specifically, Guibas and Stolfi introduce two geometric predicates which enforce the "interesting" properties of the Voronoi and Delaunay diagrams.]

[8] 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!

3.2.1 Algorithm Sketch

Like other divide and conquer algorithms, the Guibas and Stolfi's approach to constructing a Delaunay Triangulation relies on repeated partitioning of the problem's data until the partition conforms to a trivial (or base) case scenario. The divide and conquer algorithm is implemented recursively, and we will discuss how the algorithm leverages recursion and defines its base cases soon.

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, I prefer to think of it as three. First, we have the "split," or "divide," stage, in which we split the large problem of computing the Delaunay of P into smaller subproblems. We continue to do this until we are operating on a collection of subproblems of a certain size. At this point, we proceed to the second stage in which we triangulate the trivial subproblems. Finally, we proceed to the "merge" phase, which is the process of strategically combining the results of the subproblems into the final triangulation on n sites.

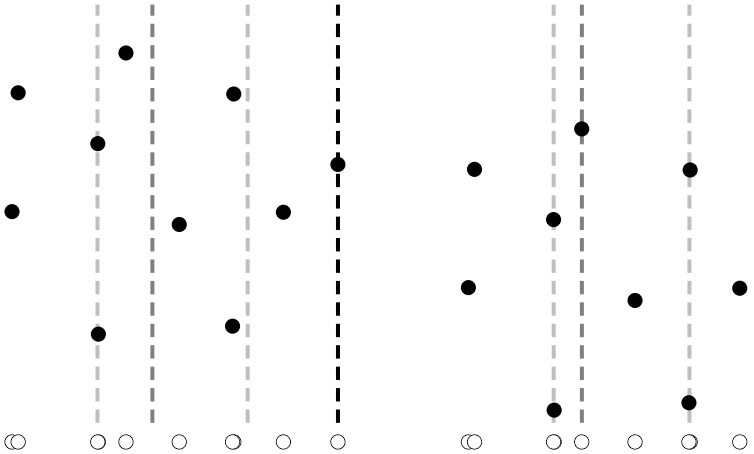
Divide and Triangulate

Given a set of sites P, whose size is n, we wish to discover D_P. To make our operations easier, we assume two things: a) that the x-coordinate of each site in P is unique; and b) that the sites are in *general position*, meaning that there will be no degenerate cases for us to handle.We strategically choose an x-value, which will serve as the vertical line along which we will recursively partition the set of sites. This will yield a left and right side, L and R, respectively.

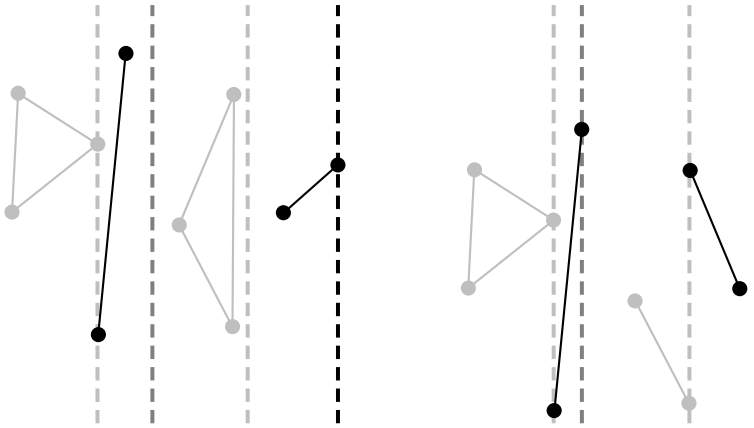
To choose the x-values which will govern our partitions, we sort the sites by x-coordinate and choose the median. Figure 3.7a depicts the vertical lines which partition P into subproblems where the number of sites is either two or three. Note that in this example, some of the sites have very similar 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 3.7b 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 $\mid P \mid$ 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



(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.

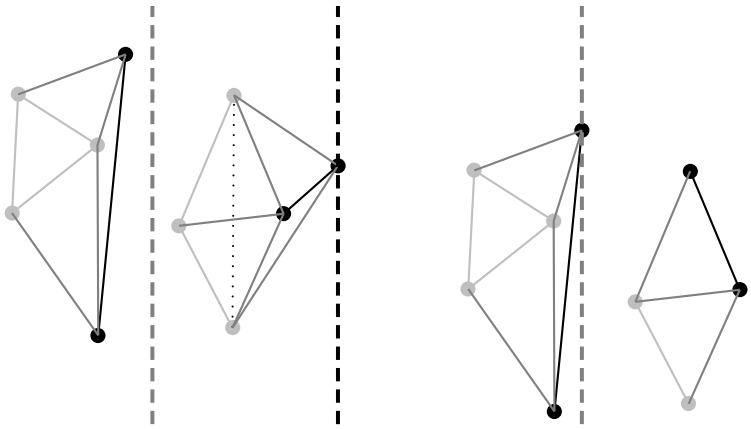
Figure 3.7: Dividing P into subproblems we know how to solve

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.

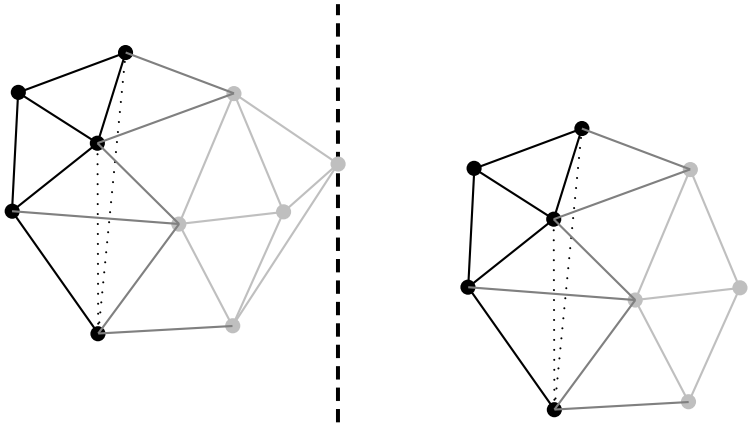
Merge

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 section 3.2.1. To provide you, dear Reader, with a visual intuition of what the merge step entails, I have included figure 3.8. Figure 3.8a illustrates the first iteration of the merge step; it knits together the eight subproblems in figure 3.7b into four subproblems. Figure 3.8b 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 3.7a and 3.7b are edges which had to be removed in order to maintain 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 2.3– to determine which, if any, edges need to be removed during the merge process.



(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 3.8a. Again, the dotted line segments are removed to maintain the Delaunay propertry.

Figure 3.8: Merging subproblems pairwise.

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 3.9. We know this edge to be valid because the lower and upper common tangents of D_L and D_R can easily 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.

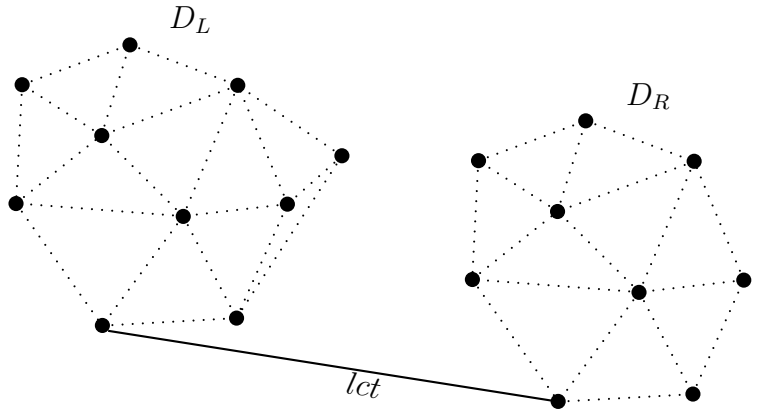


Figure 3.9: Lower Common Tangent, *lct*, of D_L and D_R

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 3.9; 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 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[9] 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*.

[9]As discussed in section 2.6.

We introduce two more edges which we will use throughout this portion of the algorithm. ℓ is the next edge in D_L to be encountered by the rising bubble. Similarly, r is the next edge in D_R to be encountered by the bubble. At the outset of this procedure, we set ℓ 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 ℓ 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:

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

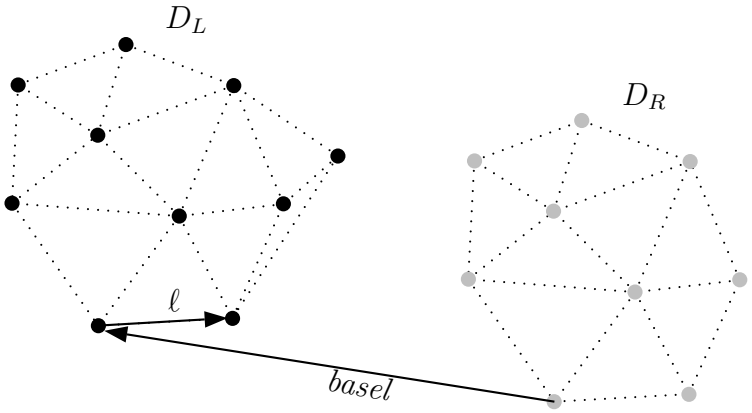
So, in the case of determining whether or not ℓ is valid, we check its validity by ensuring that CCW(ℓ*.twin.source*, *basel.twin.source*, *basel.source*) is greater than 0. If ℓ is valid, then we may continue to the next step, which involves "pruning" D_L in an appropriate way. What we need to do is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

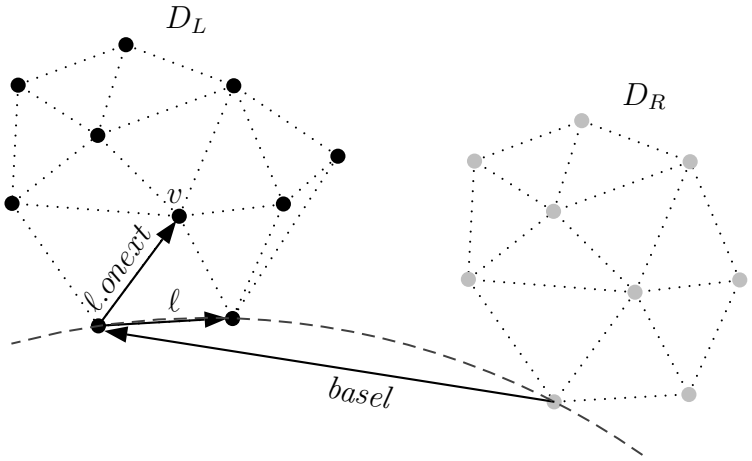
If we wish to think about it in terms of the coordinates of A, B, C, and D, the InCircle predicate is equivalent to:

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

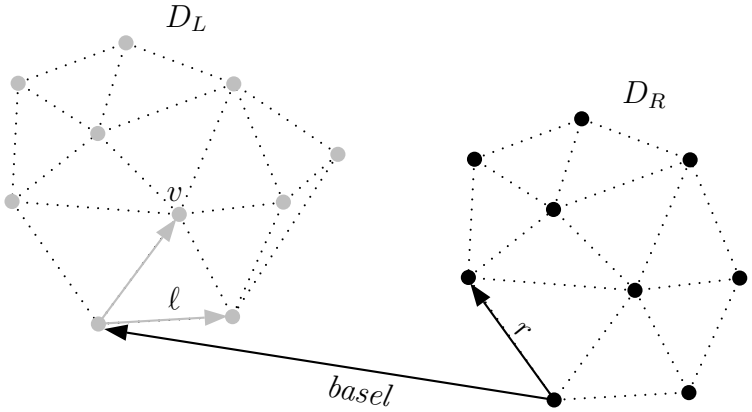
Now the notion of using circles to enforce, or as we might say *witness*, the delaunayhood of a certain edge, is not new. The InCircle predicate is precisely the circle test we performed in section 2.3 and saw in figure 2.4. When we apply this notion of the InCircle to determining whether or not ℓ will remain in the merged diagram D_P, we will do so as follows: if InCircle(*basel.twin.source*, *basel.source*, ℓ*.twin.source*, ℓ*.onext*().*twin.source*) then ℓ*.onext*().*twin.source* is in the circle defined by *basel.twin.source*, *basel.source*, and ℓ*.twin.source*, which means that we must delete the edge ℓ and set the new value of ℓ to be ℓ*.onext*(). We continue to do this, deleting edges as necessary, until the InCircle test fails. Then, we repeat an analagous procedure for r.



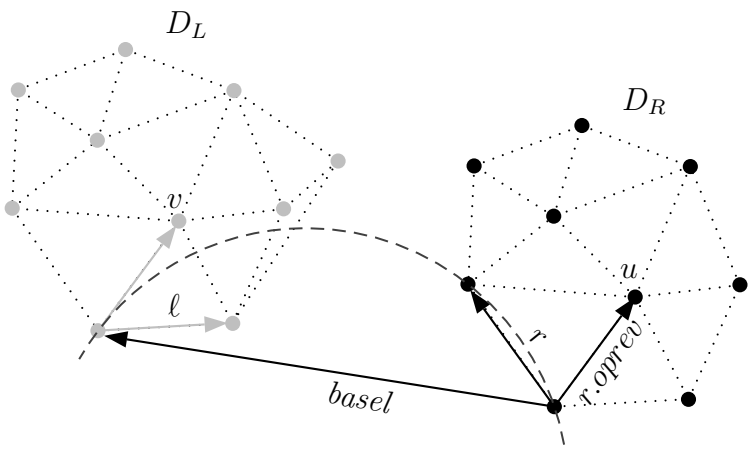
(a) First edge, *basel*, of D_P and ℓ, candidate edge in D_L. Note that ℓ is considered to be valid because CCW(ℓ.*twin.source*, *basel.twin.source*, *basel.source*) is true. That is to say, ℓ.*twin.source*, *basel.twin.source*, *basel.source* is a counterclockwise ordering.



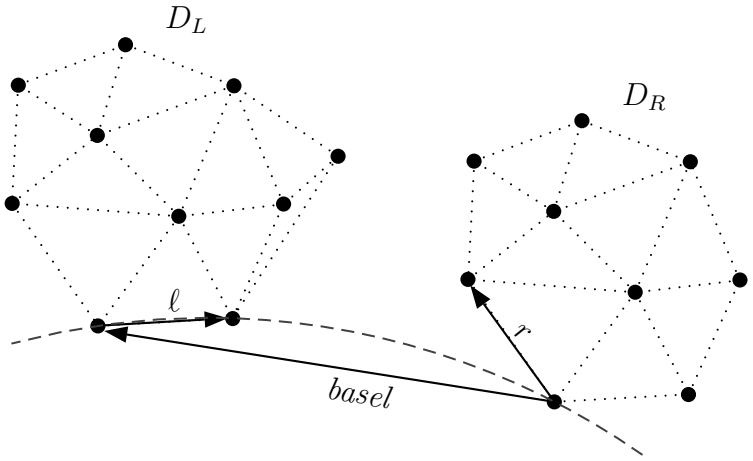
(b) Evaluating whether or not ℓ will be in D_P. The dashed arc is a portion of the "rising bubble" which we say has *witnessed* that v in not in the circle defined by ℓ.*twin.source*, *basel.twin.source*, and *basel.source*.



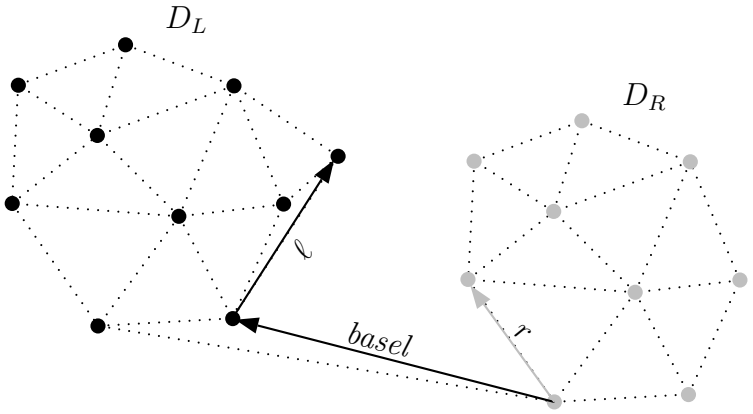
(c) Similarly, now we evaluate whether or not r will be in D_P. Note that it is a valid edge, since r.*twin.source*, *basel.twin.source*, *basel.source* is a counterclockwise ordering.



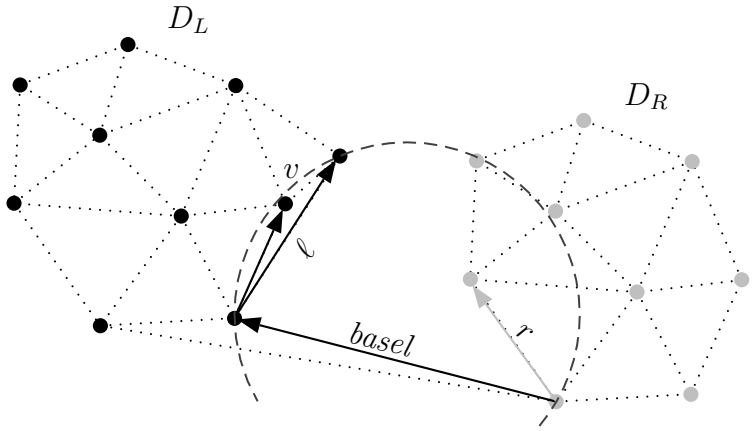
(d) Evaluating whether or not r will be in D_P. The dashed arc is a portion of the "rising bubble" which we say has *witnessed* that u in not in the circle defined by $r.twin.source$, $basel.twin.source$, and $basel.source$.



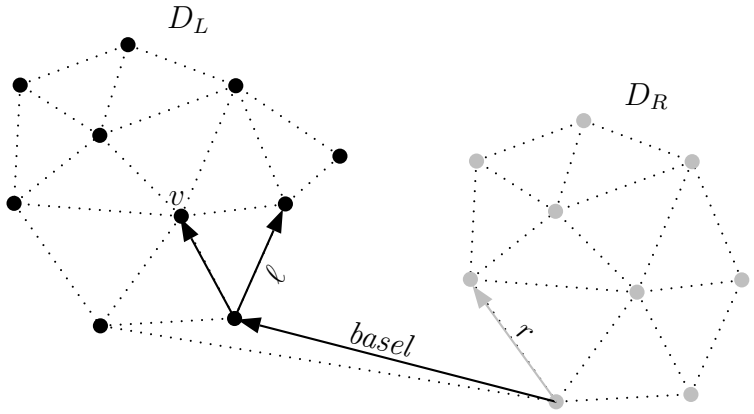
(e) Since ℓ and r are both valid we perform InCircle($\ell.twin.source$, $\ell.source$, $r.source$, $r.twin.source$). Note that $r.twin.source$ is not in the circle formed by $\ell.twin.source$, $\ell.source$, $r.source$, so InCircle returns false.



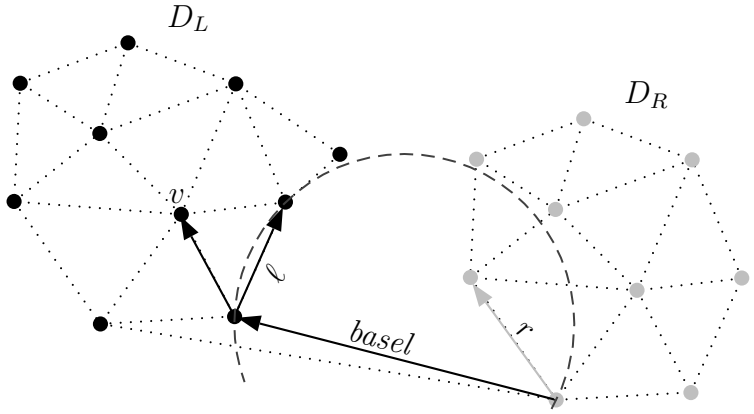
(f) Due to the result of the InCircle test, we update $basel$'s destination to $\ell.twin.source$. We also update ℓ to $basel.twin.onext()$.



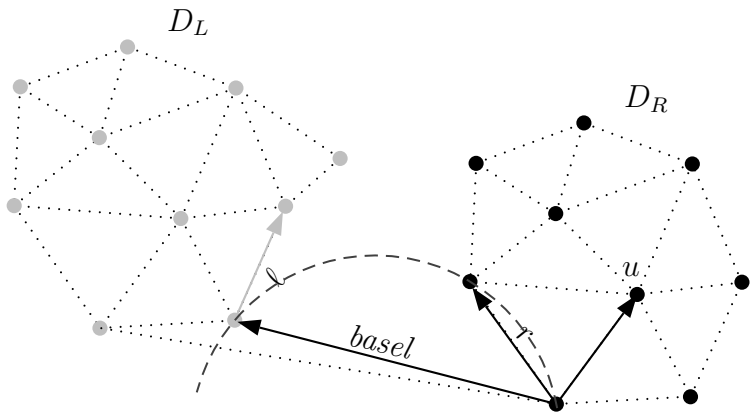
(g) We continue this process noting that ℓ is a valid edge. Because this is the case, we check to see if v, that is, $\ell.onext().twin.source$, is contained in the circle represented by the dashed lines. It is.



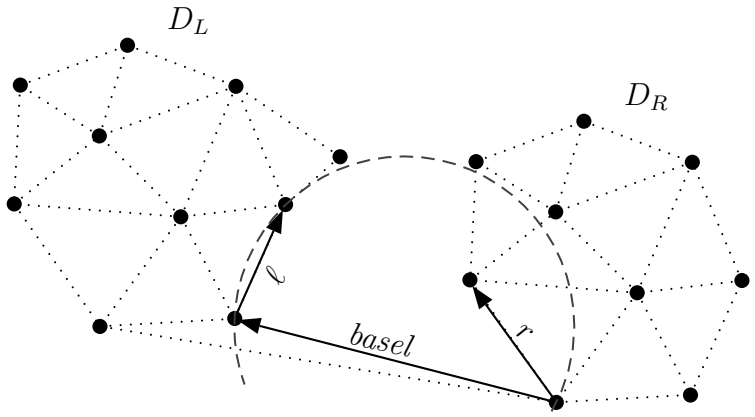
(h) Because v is contained in the circle, we need to remove l and set the new value of l to $l.onext()$, and making the analagous change for v. Note $l.onext()$ is also shown as a directed edge.



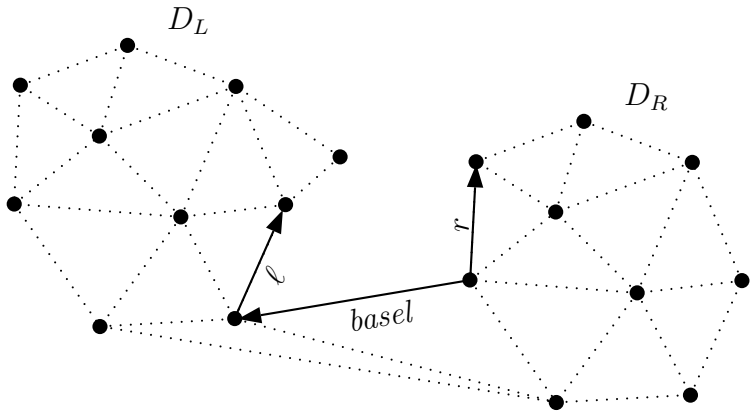
(i) We perform the same test again, checking to see if v is contained in the dashed circle. This time, it is not, so we know that ℓ will be in D_P.



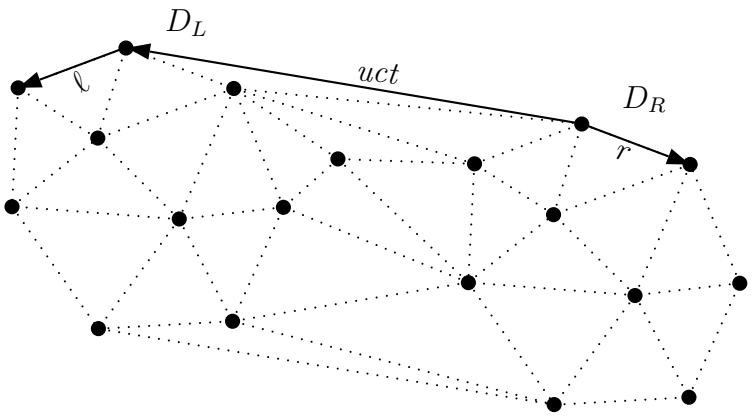
(j) We consider the right side again, noting that r is still valid. We may note that u has not been reassigned. However, since *basel* has changed, we need to check that u is not in the circle formed by *r.twin.source*, *basel.twin.source*, and *basel.source*. Here, it is not.



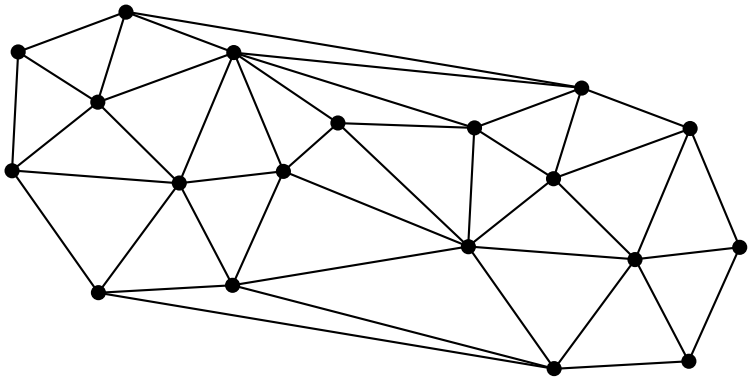
(k) Again, since ℓ and r are both valid we perform InCircle(ℓ*.twin.source*, ℓ*.source*, *r.source*, *r.twin.source*). Note that *r.twin.source* is in the circle formed by ℓ*.twin.source*, ℓ*.source*, *r.source*, so InCircle returns true.



(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 3.10: Knitting together D_L and D_R to form D_P

[SUMMARY of what we just did]

3.2.2 Divide and Conquer Algorithm Pseudocode

algorithm 4 and its supporting procedures [write these], provide a pseudocode implementation of a divide and conquer algorithm, as presented in Guibas and Stolfi's paper. The corresponding procedure for the dual is described in Shamos and Preparata's text.

Algorithm 4 Divide and Conquer Delaunay Diagram

```
1: procedure DelDivConq(P)                          ▷ Construct Del(P)
2:     L, R, ← PartitionSites(P)                    ▷ Partition P by median x-coordinate
3:     Del(L) ←DelDivConq(L)                        ▷ Recursive call
       Del(R) ←DelDivConq(R)
4:     DelMerge(Del(L), Del(R))                     ▷ Merge sub-diagrams
5: end procedure
```

Algorithm 5 Divide and Conquer Delaunay Diagram: Merge

```
1: procedure DelMerge(Del(L), Del(R))
2: end procedure
```

Algorithm 6 Divide and Conquer Delaunay Diagram: Lower Common Tangent

```
1: procedure DelLCT(Del(L), Del(R))
2: end procedure
```

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

Descartes, R. (2004). *Descartes: The World and Other Writings*. Cambridge, United Kingdom: Cambridge University Press.

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