Contents

Chapter 1

Introduction

Let $\|\cdot\| : \mathbb{R}^2 \to \mathbb{R}$ be a norm. Then we define the distance function as

$$\operatorname{dist}(p, q) = \|p - q\| . \tag{1.1}$$

For $1 \leq p < \infty$ we define the L^p norm by

$$\|(x, y)\|_p = \left(|x|^p + |y|^p \right)^{1/p}, \tag{1.2}$$

and we note that $\|\cdot\|_2$ is the well-known Euclidean distance. For $p = 1$, the above reduces to

$$\|(x, y)\|_1 = |x| + |y| . \tag{1.3}$$

Letting $p \to \infty$, we also obtain the norm

$$\|(x, y)\|_\infty = \max\left(|x| , |y| \right). \tag{1.4}$$

Definition 1.1 (Voronoi diagram)**.** Let $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$. The cells corresponding to each point are denoted by

$$\mathcal{V}(p_i) = \{q \in \mathbb{R}^2 \mid \operatorname{dist}(q, p_i) < \operatorname{dist}(q, p_j) \text{ for all } i \neq j\}.$$

The Voronoi diagram of P, denoted $\operatorname{Vor}(P)$, is the subdivision of $\mathbb{R}^2$ consisting of the union of the cells $\mathcal{V}(p_1), \mathcal{V}(p_2), \ldots, \mathcal{V}(p_n)$.

The following figure shows how the Voronoi diagram for 9 random points looks like with regards to some different L^p norms:

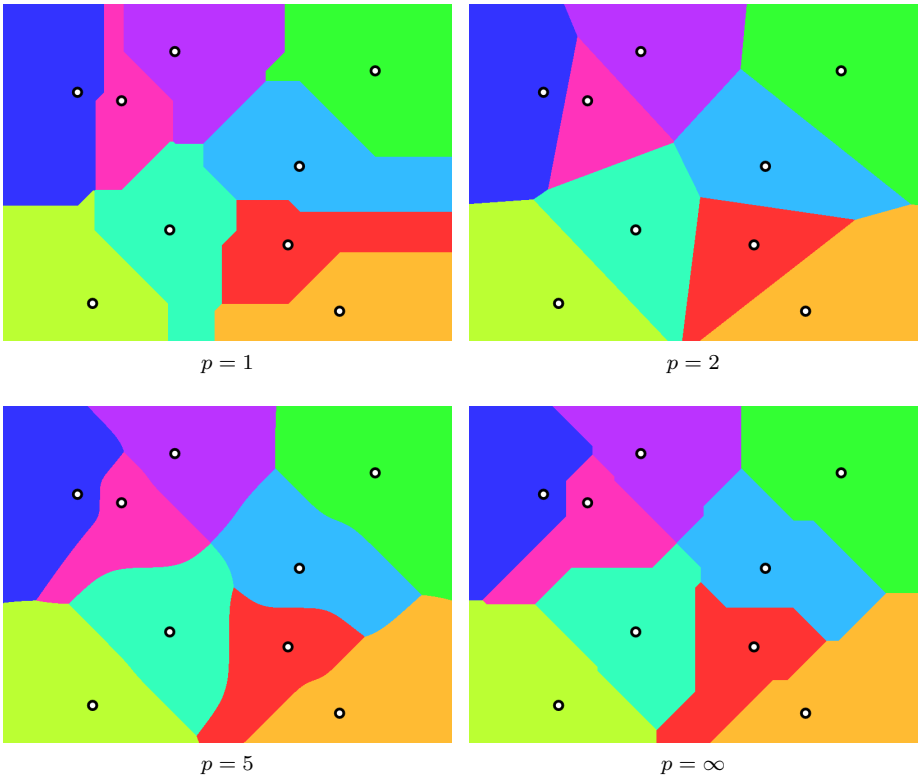


Figure 1.1: $\mathrm{Vor}(P)$ of 9 random points using different $\|\cdot\|_p$

The above figures were generated using a very naive algorithm, which for each each pixel determinates which of the 9 points is the closest with regards to the chosen norm. A demo is available in demos/pixel-voronoi-naive.

Note that some of the cells may be unbounded, for example the bottom left green cell in the above figure. For $p = 1$ and $p = \infty$ the boundaries of the cells $\mathcal{V}(p_i)$ are characterised by lines, rays and segments that can only point in the 8 compass directions. For $p = 2$ the boundaries consist of lines, rays and segments which can point in any direction. Interestingly, for $2 < p < \infty$ it seems that the boundary consists of smooth curves that are not necessarily part of a line.

We now want to look at the graph structure of the Voronoi diagram. For $P = \{p_1, p_2, \ldots, p_n\} \subset \mathbb{R}^2$ the set

$$\mathrm{Vor}_{\mathrm{G}}(P) = \mathbb{R}^2 - \mathrm{Vor}(P) = \{q \in \mathbb{R}^2 \mid \mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j) \text{ for some } i \neq j\}$$

turns out to be an embedding of a graph, where some of the edges are infinite, here's a visualization:

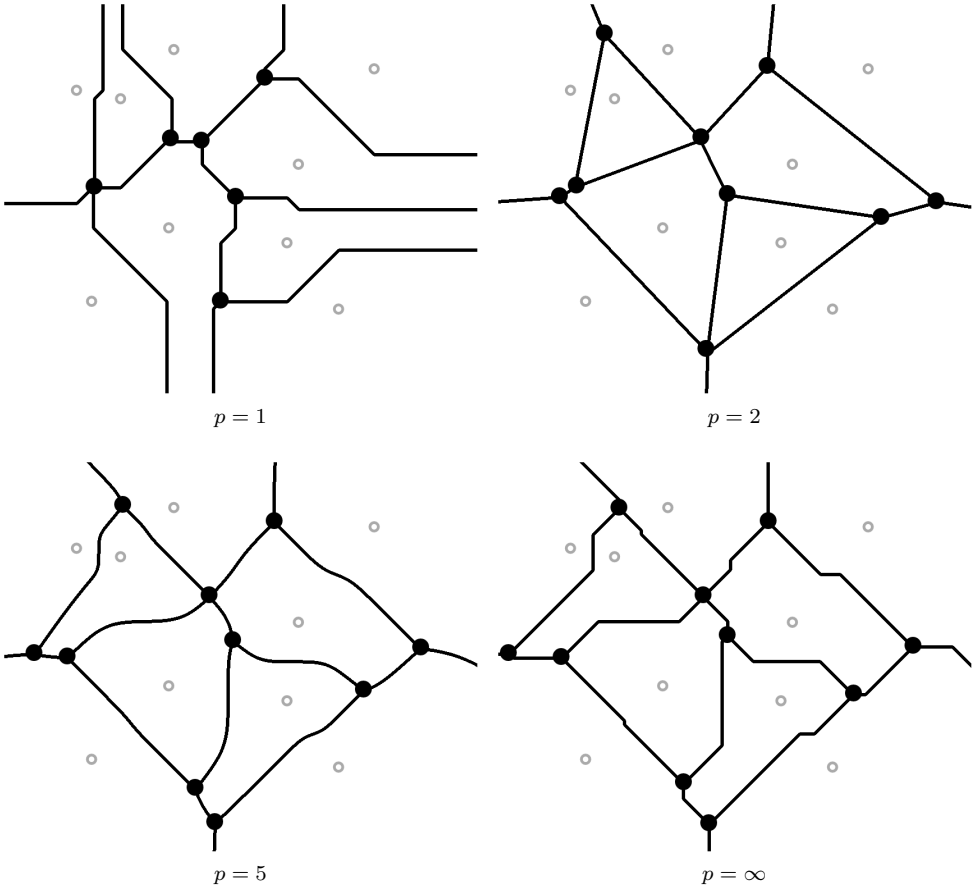


Figure 1.2: $\mathrm{Vor}_{\mathrm{G}}(P)$ of the 9 random points using different $\|\cdot\|_p$.

The above figures were generated by first generating the images from Figure 1.1 and then performing the following algorithm: For each pixel, we look at the surrounding pixels within a small disk about that point, and if it contains exactly 2 different colors, we know that we're looking at an edge, so we color the pixel black, and if we see 3 colors or more, we know that we're at a vertex. If we only see 1 color, then we just color the pixel white.

Note that it's the black vertices and edges which make up the graph, the gray points from P are just there for visualization. Rather than computing $\mathrm{Vor}(P)$, our algorithms will actually compute $\mathrm{Vor}_{\mathrm{G}}(P)$, and from there be able to compute $\mathrm{Vor}(P)$.

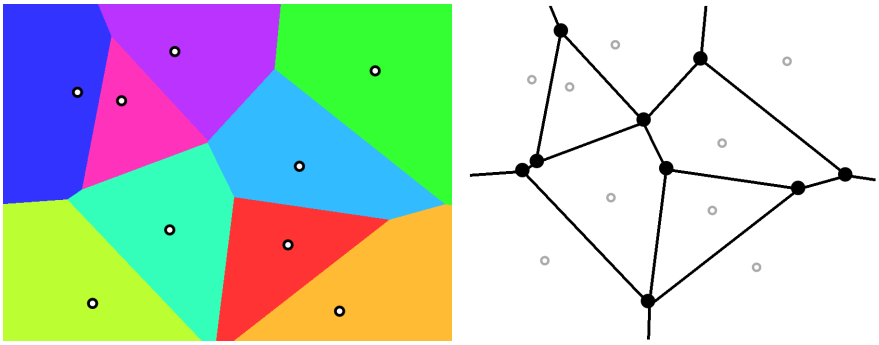
Chapter 2

Properties of Euclidean Voronoi Diagrams

In this chapter we will consider Voronoi diagrams for the L^2 norm, also known as the Euclidean norm. The norm is given by

$$\|(x, y)\|_2 = \sqrt{x^2 + y^2},$$

for all $x, y \in \mathbb{R}$. Here is the example diagram with this norm from earlier:

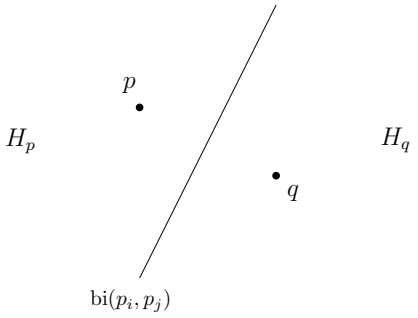


We note that the diagram consists of straight lines, rays and line segments. In the following sections we will describe the shape of the diagram in detail.

2.1 Bisectors, halfplanes and Voronoi cells

From linear algebra we know that $\|v\|_2 = \sqrt{\langle v, v \rangle}$, where $\langle \cdot, \cdot \rangle$ is the usual dot product on $\mathbb{R}^2$. Given two points $p, q \in \mathbb{R}^2$ then the **bisector** of p and q is denoted by $\mathrm{bi}(p, q) \subset \mathbb{R}^2$ and denotes the set of points on a line ℓ which passes

through the midpoint of p and q and is orthogonal (w.r.t. $\langle \cdot, \cdot \rangle$) to the vector $p - q$.

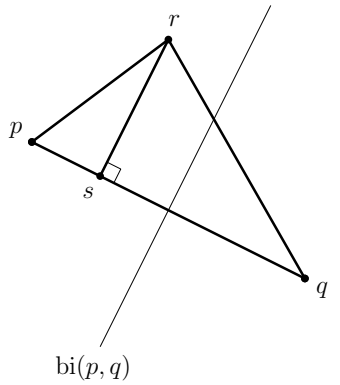


A bisector $\mathrm{bi}(p, q)$ splits the plane into two **half-planes** H_p and H_q such that $p \in H_p$ and $q \in H_q$. We define $h(p, q)$ to be the open half-plane which contains p, that is the interior of H_p. So we have that

$$\mathbb{R}^2 = h(p, q) \cup \mathrm{bi}(p, q) \cup h(q, p).$$

Proposition 2.1. $r \in h(p, q)$ if and only if $\mathrm{dist}(r, p) < \mathrm{dist}(r, q)$.

Proof. Let $r \in h(p, q)$ and let s be the projection of r onto the line segment $\overline{pq}$.



The Pythagorean theorem and the fact that $\overline{ps}$ is shorter than $\overline{sq}$ then gives us

$$\begin{aligned} \|p - r\|^2 &= \|p - s\|^2 + \|s - r\|^2 \\ &< \|q - s\|^2 + \|s - r\|^2 \\ &= \|q - r\|^2 , \end{aligned}$$

which gives us that $\mathrm{dist}(r, p) < \mathrm{dist}(r, q)$. The other direction is symmetrical. □

Corollary 2.2. For every Voronoi cell we have

$$\mathcal{V}(p_i) = \bigcap_{\substack{1 \le j \le n \\ j \neq i}} h(p_i, p_j).$$

Proof. "$\subset$": Let $r \in \mathcal{V}(p_i)$. Then $\operatorname{dist}(r, p_i) < \operatorname{dist}(r, p_j)$ for all $i \neq j$. Prop 2.1 then gives us that this is equivalent to $r \in h(p_i, p_j)$ for all $i \neq j$.

"$\supset$": This argument is symmetrical to the above argument. □

A Voronoi cell is thus the intersection of convex sets and is therefore convex. We conclude that the Voronoi cells are open and convex (possibly unbounded) polygons with at most $n-1$ vertices and $n-1$ edges.

2.2 Shape of the entire diagram

We now look at the shape of the entire Voronoi diagram. From Corollary 2.2 it follows that the edges of $\operatorname{Vor}_{\mathrm{G}}(P)$ are made up of parts of straight lines, namely the bisectors between different points of P. We now classify these based on the structure of the points in P:

Theorem 2.3. If the points in P are collinear then $\operatorname{Vor}_{\mathrm{G}}(P)$ consists of $n-1$ parallel lines. Otherwise, $\operatorname{Vor}_{\mathrm{G}}(P)$ is connected and its edges are either segments or half-lines.

Proof. Assume that the points in P are collinear. By applying an isometry to P, we may assume without loss of generality that the points of P lie on the x-axis:

$$P = \{(x_1, 0), (x_2, 0), \ldots, (x_n, 0)\},$$

where we assume that $x_1 < x_2 < \cdots < x_n$ by rearranging the points if necessary. By definition, we have that $p \in \operatorname{Vor}_{\mathrm{G}}(P)$ if and only if $p \notin \mathcal{V}(x_i, 0)$ for all i. Let $(x, y) \in \mathbb{R}^2$ such that $x_i < x < x_{i+1}$. Then $(x, y) \in \operatorname{Vor}_{\mathrm{G}}(P)$ if

$$\operatorname{dist}((x, y), (x_i, 0)) = \operatorname{dist}((x, y), (x_{i+1}, 0)).$$

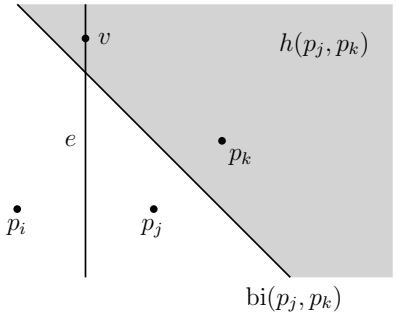
If furthermore $(x, y) \in \operatorname{Vor}_{\mathrm{G}}(P)$ then we get

$$\begin{aligned} & \|(x, y) - (x_i, 0)\| = \|(x, y) - (x_{i+1}, 0)\| \\ \iff & \sqrt{(x - x_i)^2 + y^2} = \sqrt{(x - x_{i+1})^2 + y^2} \\ \iff & |x - x_i| = |x - x_{i+1}|\,. \end{aligned}$$

Thus if $(x, 0) \in \operatorname{Vor}_{\mathrm{G}}(P)$ then $(x, y) \in \operatorname{Vor}_{\mathrm{G}}(P)$ for all $y \in \mathbb{R}$. This shows that $\operatorname{bi}((x_i, 0), (x_{i+1}, 0)) \subset \operatorname{Vor}_{\mathrm{G}}(P)$ for all $i < n$. Every point of $\operatorname{Vor}_{\mathrm{G}}(P)$ is on one of these bisectors, and the bisectors are all parallel, which proves the claim.

Assume that the points in P are not collinear. First, we show that the edges of $\operatorname{Vor}_{\mathrm{G}}(P)$ are either segments or half-lines. Suppose for a contradiction that there is an edge e of $\operatorname{Vor}_{\mathrm{G}}(P)$ that is a full line and assume that $e \subset \partial\mathcal{V}(p_i) \cap \partial\mathcal{V}(p_j)$. Let $p_k \in P$ be a point which is not collinear with p_i and p_j. Then the line $\operatorname{bi}(p_j, p_k)$ is not parallel to the line e, hence they have an

intersection point. Then there exists a point $v \in e \cap {}^\circ h(p_k, p_j)$. The situation is visualized here:



We have that $v \in \partial\mathcal{V}(p_j)$ by definition of e. Now note that

$$\partial\mathcal{V}(p_j) = \partial\left(\bigcap_{a\neq j} h(p_j, p_a)\right) \subset^{1} \bigcup_{a\neq j} \partial h(p_j, p_a) = \bigcup_{a\neq j} \mathrm{bi}(p_j, p_a).$$

As $v \in h(p_k, p_j)$ we have that $\mathrm{dist}(v, p_k) < \mathrm{dist}(v, p_j)$, hence $v \notin \mathrm{bi}(p_j, p_k)$, so $v \notin \partial V(v_j)$ by the above characterization of $\partial\mathcal{V}(p_j)$. This is a contradiction, so e can't be a full line. Now we show that $\mathrm{Vor}_{\mathrm{G}}(P)$ is connected. Assume for the sake of a contradiction that $\mathrm{Vor}_{\mathrm{G}}(P)$ is not connected. Then there exists a $\partial\mathcal{V}(p_i)$ which is not path connected. This can only happen if $\partial\mathcal{V}(p_i)$ consists of two parallel lines. This contradicts the fact that $\mathrm{Vor}_{\mathrm{G}}(P)$ contains no lines. Thus $\mathrm{Vor}_{\mathrm{G}}(P)$ is connected. □

Finally, we show that that the complexity of the vertices and edges is $\mathcal{O}(n)$:

Theorem 2.4. For $n \geq 3$, the number of vertices in $\mathrm{Vor}_{\mathrm{G}}(P)$ is at most $2n-5$ and the number of edges is at most $3n-6$.

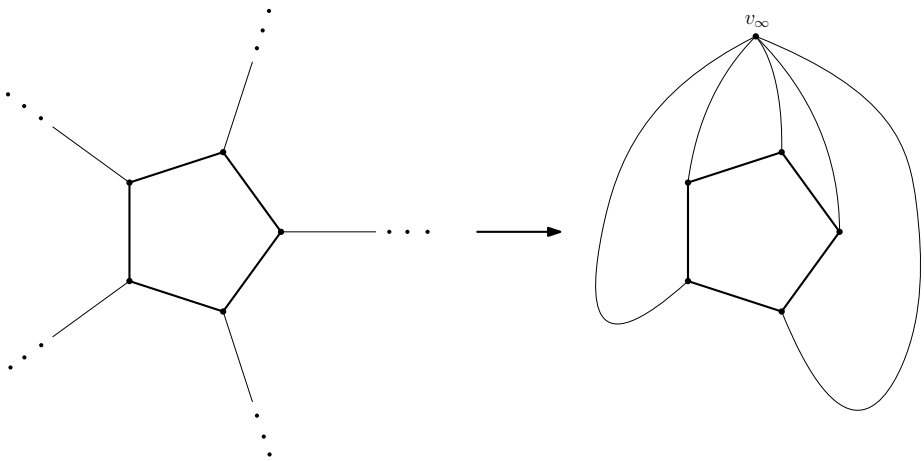
Proof. If the points in P are collinear, then Theorem 2.3 implies the claim. Now assume that the points in P are not collinear. As a first preprocessing step, we start by transforming $\mathrm{Vor}_{\mathrm{G}}(P)$ into an actual plane graph, as some of the edges in $\mathrm{Vor}_{\mathrm{G}}(P)$ may be half-lines. Let $v_1, \ldots, v_k$ denote the vertices of $\mathrm{Vor}_{\mathrm{G}}(P)$. Let $p = \frac{1}{k}(v_1 + v_2 + \cdots + v_k) \in \mathbb{R}^2$ and let

$$r = 1 + \max\{\mathrm{dist}(p, v_1), \mathrm{dist}(p, v_2), \ldots, \mathrm{dist}(p, v_k)\}.$$

Then let $B_r(p) \subset \mathbb{R}^2$ denote the open ball with center p and radius r. We have that $B_r(p)$ contains every vertex v_i and that every half-line edge e of $\mathrm{Vor}_{\mathrm{G}}(P)$ intersects $\partial B_r(p)$ exactly once. Now define $v_\infty \in \mathbb{R}^2$ as any point in $\mathbb{R}^2 - B_r(p)$ and transform every half-line edge e into a path with finite length by connecting the half-lines to the point v_∞. This is possible since $\mathbb{R}^2 - \overline{B_p(r)}$ only contains these half-lines, and every half-line is pointing in a unique direction so we may

[1] Here we used that $\partial(A \cap B) \subset \partial A \cup \partial B$.

then transform the half-lines in order by starting with those which are closest to v_∞. An example of this construction is given here:



In this way we can turn $\mathrm{Vor_G}(P)$ into a planar graph. For a planar graph G, Euler's polyhedra formula from topology states that

$$V - E + F = 2, \tag{2.1}$$

where V is the number of vertices, E is the number of edges and F is the number of faces of G. Let n_v denote the number of vertices of the original $\mathrm{Vor_G}(P)$, and let n_e denote the number of edges. In our modification, we only added a single vertex, so by plugging into (2.1) we obtain the following relationship:

$$(n_v + 1) - n_e + n = 2. \tag{2.2}$$

Note that n is the number of faces, since we have a Voronoi cell for each point in P. Every vertex v in G has $\deg(v) \geq 3$, otherwise there would be a $\mathcal{V}(p_i)$ which is not convex. This means that

$$\sum_{v \in V(G)} \deg(v) \geq 3\,|V(G)| = 3(n_v + 1).$$

Now we want to compute the left side of the above inequality. Given a vertex v we have that $\deg(v)$ counts the number of edges which touch v, and in G every edge touches exactly 2 vertices, which gives us that $\sum_{v \in V(G)} \deg(v) = 2n_e$. Combining these facts, we obtain the inequality:

$$2n_e \geq 3(n_v + 1). \tag{2.3}$$

Multiplying (2.2) by 2, isolating $2n_e$ and then applying (2.3) we get:

$$\begin{aligned} 2(n_v + 1) - 2n_e + 2n = 4 &\iff 2n_e = (2n_v + 1) + 2n - 4 \\ &\implies 3(n_v + 1) \leq 2(n_v + 1) + 2n - 4 \\ &\implies n_v \leq 2n - 5. \end{aligned}$$

Multiplying (2.2) by 3, isolating $3(n_v + 1)$ and then applying (2.3) we get:

$$\begin{aligned} 3(n_v + 1) - 3n_e + 3n = 6 &\iff 3(n_v + 1) = 3n_e - 3n + 6 \\ &\implies 2n_e \geq 3n_e - 3n + 6 \\ &\implies n_e \leq 3n - 6. \end{aligned}$$

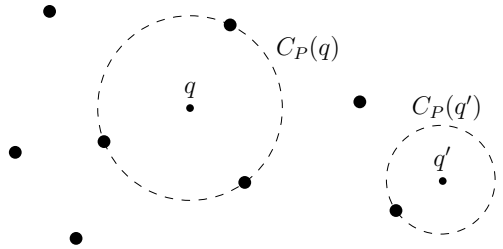
This proves the theorem. □

2.3 Characterizing bisectors in the diagram

We have seen that we have a linear number of vertices and edges $\mathrm{Vor}_{\mathrm{G}}(P)$, but we have a quadratic number of bisectors $\mathrm{bi}(p_i, p_j)$ of which every edge of $\mathrm{Vor}_{\mathrm{G}}(P)$ is a subset of, and every vertex in $\mathrm{Vor}_{\mathrm{G}}(P)$ is an intersection point of two such bisectors. Thus it would be interesting to characterize when a particular bisector is a part of $\mathrm{Vor}_{\mathrm{G}}(P)$. First, we need a definition:

Definition 2.5 (Largest empty circle)**.** For a $q \in \mathbb{R}^2$ we define $C_P(q)$ to be *the largest empty circle of q with respect to P*, which is the largest empty circle with q as its center that does not contain any point of P in its interior. Formally,

$$C_P(q) = B_r(q), \quad \text{where} \quad r = \sup\{\lambda \in \mathbb{R}^+ \mid B_\lambda(q) \cap P = \varnothing\}.$$



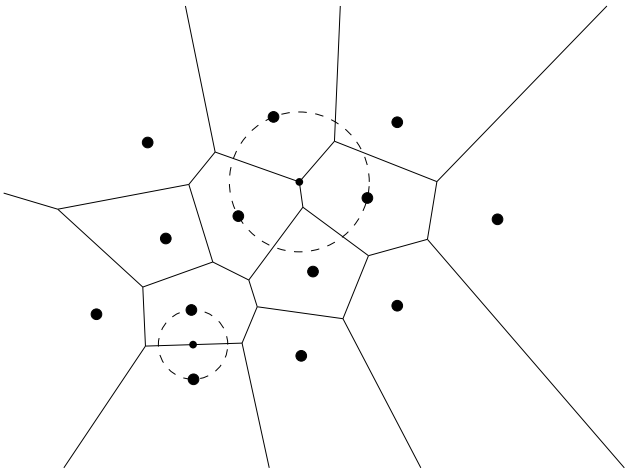
Theorem 2.6. The bisectors and their intersections are characterized by:

(i) $q \in \mathbb{R}^2$ is a vertex of $\mathrm{Vor}_{\mathrm{G}}(P)$ if and only if

$$|\partial C_P(q) \cap P| \geq 3.$$

(ii) $\mathrm{bi}(p_i, p_j)$ defines an edge of $\mathrm{Vor}_{\mathrm{G}}(P)$ if and only if

$$\exists q \in \mathrm{bi}(p_i, p_j) \colon \partial C_P(q) \cap P = \{p_i, p_j\}.$$



Proof. We prove each statement individually:

(i): "$\Leftarrow$": Let $q \in \mathbb{R}^2$ and assume that $|\partial C_P(q) \cap P| \geq 3$. Let p_i, p_j, p_k be three distinct points from $\partial C_P(q) \cap P$. Since $C_P(q) \cap P = \varnothing$ by definition, this means that q is equally close to p_i, p_j, p_k but not closer to any other points in P, so $q \in \partial\mathcal{V}(p_i) \cap \partial\mathcal{V}(p_j) \cap \partial\mathcal{V}(p_k) \subset \mathrm{Vor}_{\mathrm{G}}(P)$, and it is a vertex since it is at an intersection of 3 or more bisectors.

"$\Rightarrow$": Let $q \in \mathbb{R}^2$ be a vertex of $\mathrm{Vor}_{\mathrm{G}}(P)$. A vertex of $\mathrm{Vor}_{\mathrm{G}}(P)$ touches at least 3 different edges, and thus touches at least 3 distinct Voronoi cells $\mathcal{V}(p_i), \mathcal{V}(p_j)$ and $\mathcal{V}(p_k)$. So $q \in \partial\mathcal{V}(p_i) \cap \partial\mathcal{V}(p_j) \cap \partial\mathcal{V}(p_k)$. This gives us that

$$\mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j) = \mathrm{dist}(q, p_k).$$

Denote the above distance by D. Now assume for the sake of a contradiction that there exists $p_\alpha \in P$ such that $\mathrm{dist}(q, p_\alpha) < D$. Then there are parts of the bisectors $\mathrm{bi}(p_\alpha, p_i), \mathrm{bi}(p_\alpha, p_j), \mathrm{bi}(p_\alpha, p_k)$ contained inside $B_D(q)$, which means that $\mathcal{V}(p_i), \mathcal{V}(p_j), \mathcal{V}(p_k)$ do not all meet at q, a contradiction. This means that $C_P(q) \cap P = \varnothing$ and $p_i, p_j, p_k \in \partial C_P(q)$.

(ii): "$\Leftarrow$": Let $q \in \mathrm{bi}(p_i, p_j)$ such that $\partial C_P(q) \cap P = \{p_i, p_j\}$. So $C_P(q) \cap P = \varnothing$, which by definition of $C_P(q)$ means that

$$\mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j) \leq \mathrm{dist}(q, p_k)$$

for all k. So $q \in \mathrm{Vor}_{\mathrm{G}}(P)$ and is either a vertex or an edge. Since $|\partial C_P(q) \cap P| < 3$ part (i) gives us that q is not a vertex, hence it must be an edge, which is a subset of $\mathrm{bi}(p_i, p_j)$.

"$\Rightarrow$": Let $e \subset \mathrm{bi}(p_i, p_j)$ be an edge of $\mathrm{Vor}_{\mathrm{G}}(P)$. For $q \in e$ we have that $\mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j)$, and that q touches $\mathcal{V}(p_i)$ and $\mathcal{V}(p_j)$. By applying the same contradiction proof as in (i) "$\Rightarrow$" we have that there is no point in P which is closer to q than p_i and p_j, thus $\partial C_P(q) \cap P = \{p_i, p_j\}$.

□

2.4 The DCEL data structure

We want to write an algorithm to compute the Voronoi diagram, which leads us to a natural question: how do we store Voronoi diagrams on a computer? We'll need the following geometric data structure:

Definition 2.7 (DCEL)**.** A *double connected edge list* (DCEL) is a data structure which represents a subdivision of $\mathbb{R}^2$. A DCEL consists of a lists of vertices, faces and edges. For every edge we will have two copies of it, with opposite orientations, so we will refer to each copy as a directed edge and call it a half-edge, so we actually store a list of half-edges. These three structures are represented as follows:

Vertex v – represents a vertex of the subdivision. Properties:

- v.position $\in \mathbb{R}^2$: Describes the position of v.
- v.edge is a HalfEdge: Points to a half-edge which has v as its start vertex.

Face f – represents a face of the subdivision. Properties:

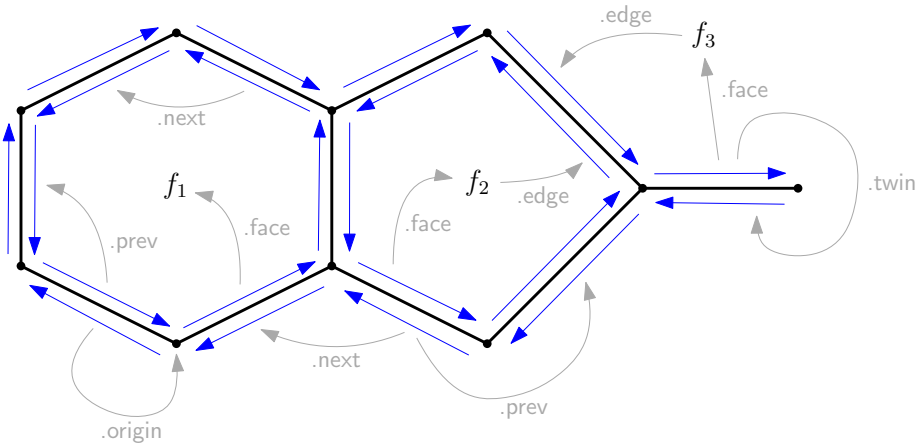
- f.edge is a HalfEdge: Points to a half-edge which lies on ∂f, and which is a part of a cycle of half-edges which goes around f in counterclockwise order.

HalfEdge e – represents a half-edge of the subdivision. Properties:

- e.origin is a Vertex: Since the half-edge is directed, we have a first and a second vertex in relation to the edge's direction, and this points to the first vertex.
- e.twin is a HalfEdge: Points to the half-edge with the same vertices as e, but pointing in the opposite direction.
- e.face is a Face: Points to the face which lies to the left of e.
- e.next is a HalfEdge: Around e.face we have a cycle half-edges which is oriented counterclockwise, and given e in this cycle, e.next gives us the next edge.
- e.prev is a HalfEdge: Around e.face we have a cycle half-edges which is oriented counterclockwise, and given e in this cycle, e.prev gives us the previous edge.

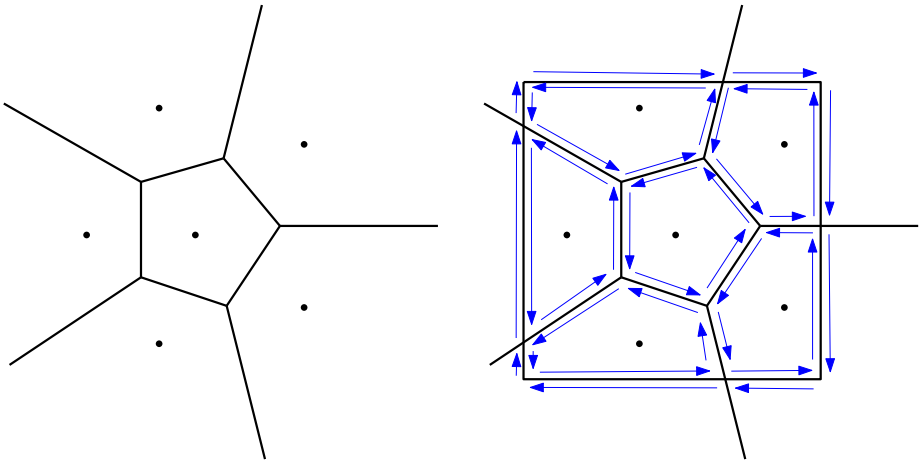
Remark 2.8. In the CompGeo book the DCEL structure allows a face to have holes, but since Voronoi diagrams and Delaunay triangulations don't have holes in their faces, we have chosen to omit this feature.

Example 2.9. Consider a graph G with 9 vertices and 10 edges embedded into $\mathbb{R}^2$, which is given as the black figure in the following:



Then this induces a subdivision of $\mathbb{R}^2$ which we represent as a DCEL. The half-edges are given as the blue arrows, the faces as f_1, f_2, f_3 and the vertices are the vertices of G. Some of the pointers are visible on the figure.

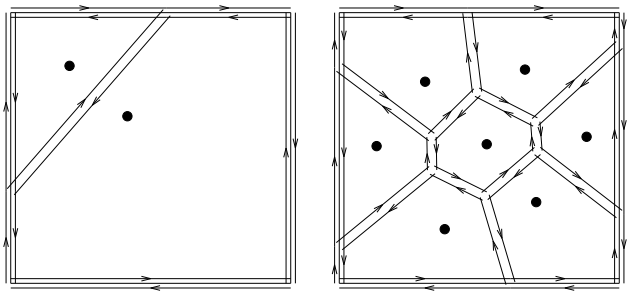
Remark 2.10. Note that the DCEL does not support infinite edges, so what we do is put a bounding box B with some padding around the vertices of $\mathrm{Vor}(P)$, and then intersect the infinite edges and faces with the boundary of B and only keep the part inside the bounding box.



The aim of our algorithms will then be to calculate the DCEL in the right figure.

How does intersecting the edges of $\mathrm{Vor}(P)$ with such a bounding box B affect

the number of edges?



In the worst case, as depicted on the left figure, 4 new edges may be added to a single unbounded face. In the general case however, as depicted on the right figure, we only introduce between 1-3 edges per face. Thus Theorem 2.4 implies that the complexity is still linear.

Chapter 3

Theory for Fortune's algorithm

In this chapter we start our treatment on Fortune's algorithm, and it will be our focus for the next three chapters. It is an algorithm for computing the Euclidean Voronoi diagram which has a running time of $\mathcal{O}(n \log n)$. Before we describe the steps of the algorithm, we have to do some theoretical ground work which will help explain why the algorithm works.

3.1 Voronoi diagrams can be used to sort

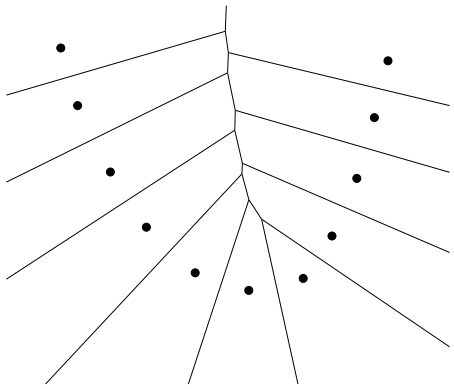
Before we begin, we show that:

Theorem 3.1. The optimal worst-case running time for computing $\mathrm{Vor}(P)$ is $\mathcal{O}(n \log n)$.

Proof. Let $A = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{R}$ and assume that $n \geq 3$. Define $\varphi \colon \mathbb{R} \to \mathbb{R}^2$ given by $\varphi(x) = (x, x^2)$. Now assume we have used an algorithm to compute a Voronoi diagram of the points

$$P = \varphi(A) = \{(a_1, a_1^2), (a_2, a_2^2), \ldots, (a_n, a_n^2)\}.$$

We obtain a diagram which looks similar to this:



We may assume without loss of generality that $a_i \geq 0$ for all i, since we may just add

$$\max\{-a \mid a \in A \cup \{0\} \text{ and } a \leq 0\}$$

to every number in A. Now we claim that

$$0 \leq a < b < c < d < e \implies \begin{cases} \operatorname{dist}(\varphi(c), \varphi(b)) < \operatorname{dist}(\varphi(c), \varphi(a)) \\ \text{and} \\ \operatorname{dist}(\varphi(c), \varphi(d)) < \operatorname{dist}(\varphi(c), \varphi(e)). \end{cases} \tag{3.1}$$

We have

$$\operatorname{dist}(\varphi(x), \varphi(y))^2 = \|\varphi(x) - \varphi(y)\|^2 = (x-y)^2 + (x^2 - y^2)^2$$

so

$$\operatorname{dist}(\varphi(c), \varphi(b)) < \operatorname{dist}(\varphi(c), \varphi(a))$$

if and only if

$$\underbrace{(c-a)^2 - (c-b)^2}_{\lambda} + \underbrace{(c^2-a^2)^2 - (c^2-b^2)^2}_{\mu} > 0.$$

The fact that $x \mapsto x^2$ is strictly increasing on $[0, \infty)$ and $0 \leq a < b < c$ implies that $\lambda > 0$ and $\mu > 0$. Using a similar argument, we obtain that $\operatorname{dist}(\varphi(c), \varphi(d)) < \operatorname{dist}(\varphi(c), \varphi(e))$. Thus (3.1) holds.

Now let $B = (b_1, b_2, \ldots, b_n)$ denote A in sorted order, i.e. $i < j$ implies $b_i < b_j$. We'll now see how we can recover B using $\operatorname{Vor}(P)$. We assume that the algorithm outputs a DCEL Δ of $\operatorname{Vor}(P)$. The property (3.1) implies that $\partial\mathcal{V}(\varphi(b_i))$ and $\partial\mathcal{V}(\varphi(b_j))$ share an edge when $i = j + 1$. This means that given $\mathcal{V}(\varphi(b_i))$ for $i < n$ we may find b_{i+1} by traversing the edges of $\mathcal{V}(\varphi(b_i))$ in Δ until we find the face which belongs to b_{i+1}. We identify this face as the one which minimizes $a_j - b_i > 0$ where $\mathcal{V}(\varphi(a_j))$ is an adjacent face. In linear time we may find ℓ such that $a_\ell < a_i$ for all $i \neq \ell$. Let $b_1 := a_\ell$. Now assume that

$b_i = a_j$ for $i < n$ and some j, and that we have the face $F = \mathcal{V}(\varphi(a_j)) \in \Delta$. We traverse the edges of F until we find the face $F' = \mathcal{V}(\varphi(a_k)) \in \Delta$ which belongs to b_{i+1}, and we let $b_{i+1} := a_k$. In the worst case we iterate through every edge of every face of Δ, but Remark 2.10 gives us that there is $\mathcal{O}(n)$ edges in total, so we find all the b_i in linear time. This means we can use an algorithm which computes $\mathrm{Vor}(P)$ to sort, which proves the claim. □

(TODO: Choose a computation model in order for the above to make sense.)

This means that the promised running time of Fortune's algorithm is optimal.

3.2 Theoretical assumptions

In order to make proving some theoretical properties easier, and to avoid not every enlightening edge cases, we will start out by making some assumptions:

Assumption 3.2. The points in P are in general position, which we define to mean that no two points in P have the same x-coordinate or the same y-coordinate.

Assumption 3.3. The points in P do not all lie on the same line.

Assumption 3.4. No more than 3 points from P lie on the same circle.

Remark 3.5. We may make Assumption 3.2 without loss of generality, because if $\Theta \subset \mathbb{R}$ is the set of all of the angles that $\overline{p_i p_j}$ make with the x-axis for all $p_i \neq p_j$ in P, then Θ is finite and $\mathbb{R} \setminus \Theta$ is infinite, so generating a random number $\theta \in \mathbb{R} \setminus \Theta$ and letting

$$\varphi(x, y) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = (\cos(\theta)x - \sin(\theta)y, \sin(\theta)x + \cos(\theta)y)$$

be the rotation about the origin with the angle θ, then the set

$$\varphi(P) = \{\varphi(p) \mid p \in P\}$$

is in general position with probability 1. After having computed the Voronoi diagram for $\varphi(P)$, we may then rotate the diagram by the angle $-\theta$ to obtain $\mathrm{Vor}(P)$.

Remark 3.6. If P is collinear then every point $p \in P$ lies on a line ℓ. Theorem 2.3 gives us that $\mathrm{Vor}_{\mathrm{G}}(P)$ consists of parallel lines and Theorem 2.6 gives us that these parallel lines are the bisectors of pairs of adjacent points on ℓ. By sorting the points on P along ℓ and then marking all the bisectors between adjacent points we then compute the Voronoi diagram of $\mathrm{Vor}_{\mathrm{G}}(P)$ in $\mathcal{O}(n \log n)$ time. With this out of the way, it is now reasonable to assume Assumption 3.3.

3.3 The beach line

In Computational Geometry a sweep line algorithm is an algorithm which incrementally computes some geometric structure, by continuously sweeping a line from one end of the plane to the other. Fortune's algorithm is such an algorithm, and it works by maintaining a horizontal sweep line $\ell\colon y = \ell_y$, and ℓ sweeps the plane from top to bottom in order to uncover the structure of the Voronoi diagram. In Fortune's algorithm there is also a secondary device, determined from the current position of the sweep line in relation to the points of P. It is called the beach line, and we describe it as follows:

For a point $p = (p_x, p_y) \in \mathbb{R}^2$ and a sweep line $\ell\colon y = \ell_y$ the distance between p and ℓ is

$$\operatorname{dist}(p, \ell) = |p_y - \ell_y| .$$

Define

$$B_i = \{q \in \mathbb{R}^2 \mid \operatorname{dist}(q, p_i) = \operatorname{dist}(q, \ell)\}$$

for all i. If $(p_i)_y > \ell_y$, it turns out we may parametrize B_i by a parabola: Let $p = (p_x, p_y)$ denote p_i and let $q = (x, y) \in B_i$. Since distances are non-negative, instead of looking at the original definition of B_i, it is equivalent to look at satisfying $\operatorname{dist}(q, p)^2 = \operatorname{dist}(q, \ell)^2$. We have:

$$\operatorname{dist}(q, p)^2 = \operatorname{dist}(q, \ell)^2 \iff (p_x - x)^2 + (p_y - y)^2 = (y - \ell_y)^2.$$

This can be transformed into the equation

$$2(p_y - \ell_y)y = x^2 - 2p_x x + p_x^2 + p_y^2 - \ell_y^2. \tag{3.2}$$

Since $p_y \neq \ell_y$ by assumption, we obtain the parabola:

$$y = \frac{1}{2(p_y - \ell_y)}(x^2 - 2p_x x + p_x^2 + p_y^2 - \ell_y^2), \tag{3.3}$$

which parametrizes B_i if $(p_i)_y > \ell_y$. Now we look at the situation where $(p_i)_y = \ell_y$. Then

$$\operatorname{dist}(q, p)^2 = \operatorname{dist}(q, \ell)^2 \iff (p_x - x)^2 + (p_y - y)^2 = (p_y - y)^2.$$

Then it must be the case that $p_x = x$, so B_i is a subset of a vertical line, and is a line segment if there is some B_k above B_i and a half-line which starts at p_i otherwise. Finally, if $(p_i)_y < \ell_y$, we let $B_i = \varnothing$. We now for all i define the maps

$$\beta_i(x) = \begin{cases} \dfrac{x^2 - 2(p_i)_x x + (p_i)_x^2 + (p_i)_y^2 - \ell_y^2}{2((p_i)_y - \ell_y)} & \text{if } (p_i)_y > \ell_y, \\ \infty & \text{otherwise.} \end{cases}$$

Let $\mathsf{LB}(x)$ denote the map which takes the minimum of each β_i, i.e.

$$\mathsf{LB}(x) = \min\{\beta_1(x), \beta_2(x), \ldots, \beta_n(x)\}.$$

Definition 3.7 (Beach line)**.** The *beach line for the points P with regards to the sweep line ℓ* is given by the following subset of $\mathbb{R}^2$:

$$G \cup V,$$

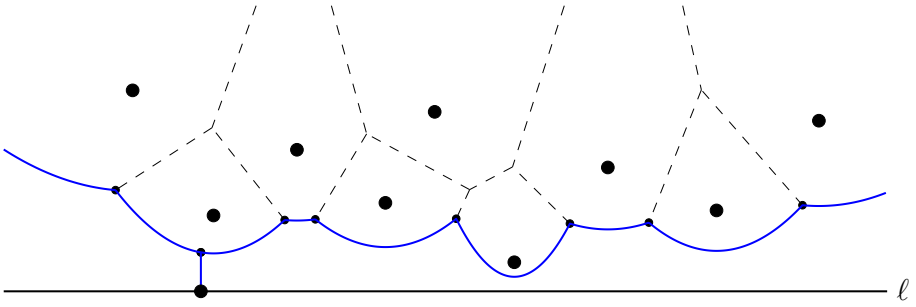
where G is the graph of LB when it is finite

$$G = \{(x, \mathsf{LB}(x)) \in \mathbb{R}^2 \mid \mathsf{LB}(x) < \infty\},$$

and V is all the vertical parts not hidden behind other parabolas

$$V = \{B_i - \{(p_i)_x\} \times (\mathsf{LB}((p_i)_x), \infty) \mid i = 1, \ldots, n \text{ where } (p_i)_y = \ell_y\}.$$

In the figure below the beach line is illustrated by the blue curves:



Remark 3.8. From the definition we see that the beach line consists of parts of parabolas, and vertical line segments or half-lines. For this reason, it is easy to see that the intersection between any vertical line and the beach line has at most one component.

Remark 3.9. For a sweep line ℓ which does not intersect any of the points in P, it follows from the definition of beach line that the map $\mathsf{LB}(x)$ parametrizes the beach line. This was used in the demo to visualize the beach line.

Definition 3.10 (Breakpoint)**.** Every point q on the beach line such that $q \in B_i \cap B_j$ for two different i, j is called a *breakpoint*.

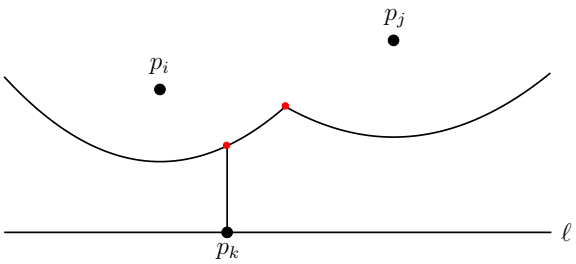


Figure 3.1: The red dots indicate the breakpoints.

3.4 Breakpoints make out the Voronoi diagram

Now we show that the breakpoints exactly trace out $\mathrm{Vor}_{\mathrm{G}}(P)$ as the sweep line ℓ moves from top to bottom.

Proposition 3.11. We have the following:

(i) For every sweep line ℓ: $y = \ell_y$ each breakpoint lies on $\mathrm{Vor}_{\mathrm{G}}(P)$.

(ii) For every point q in $\mathrm{Vor}_{\mathrm{G}}(P)$ there is a position of the sweep line ℓ such that q is a breakpoint.

Proof. We prove each statement individually:

(i): Let ℓ be the sweep line, and assume that it has one or more breakpoints. Let $q \in \mathbb{R}^2$ be such a breakpoint. Then $q \in B_i \cap B_j$ for some $i \neq j$, which means that

$$\mathrm{dist}(q, \ell) = \mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j).$$

The last equality gives us that $q \notin \mathcal{V}(p_k)$ for all k, hence $q \in \mathrm{Vor}_{\mathrm{G}}(P)$.

(ii): Let $q = (q_x, q_y) \in \mathrm{Vor}_{\mathrm{G}}(P)$. Since q is either an edge or a vertex, Theorem 2.6 gives us that $\partial C_P(q) \cap P$ has at least two elements, so let $p_i, p_j \in \partial C_P(q) \cap P$ be two different elements. We have $\mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j)$ by definition of $C_P(q)$, and then we may set

$$\ell_y := q_y - \mathrm{dist}(q, p_i),$$

and obtain

$$\mathrm{dist}(q, \ell) = \mathrm{dist}(q, p_i) = \mathrm{dist}(q, p_j).$$

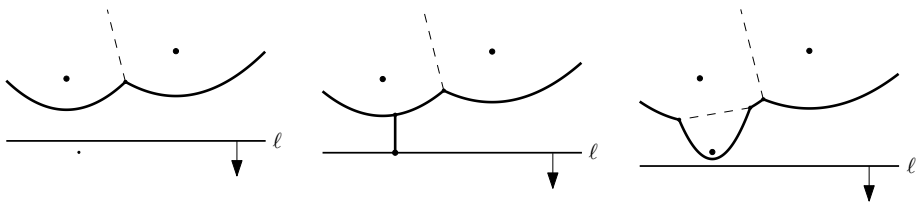
Then B_i and B_j intersect at q, and q is on the beach line since there is no B_k with a point p_k closer to q than p_i and p_j, by definition of $C_P(q)$.

□

3.5 Site and circle events

As the sweep line ℓ sweeps the plane from top to bottom, the combinatorial structure of the beach line changes. We'll categorize these changes into *events*.

First we will consider when new arcs appear on the beach line. As ℓ sweeps down and hits a point, a vertical segment is added to the beach line, and then as ℓ continues to move, the vertical line spreads out into a new parabolic arc, as seen in this figure:



Definition 3.12 (Site event)**.** When ℓ encounters a point $p_i \in P$, that is when $\ell_y = (p_i)_y$, we say that we encounter a *site event*.

Lemma 3.13. The only way in which a new arc can appear on the beach line is through a site event.

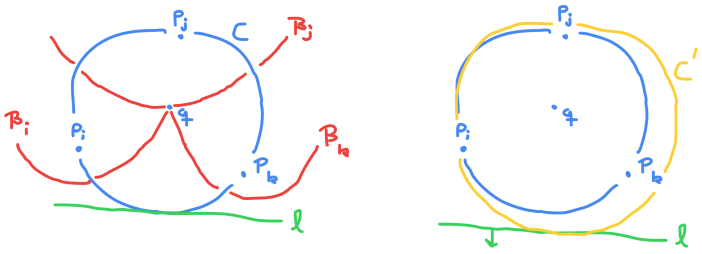
Proof. The only other alternative is for new arcs to arise due to changes in the shape and position of existing parabolas, that is due to some parabola overtaking the beach line and breaking through it. Assume for the sake of a contradiction that a new arc appears on the beach line but $\ell_y \neq p_i$ for all i. Let β_j denote the parabola which contains the new arc, associated to the point $p_j \in P$, which appears on the beach line. We have that β_j is a full parabola since $\ell_y \neq p_j$. Now, we look at the two cases in which β_j can appear as a new arc on the beach line.

The first possibility is that β_j breaks through the middle of an another arc which is a part of the parabola β_i. For this to happen, there is a time at which β_i and β_j either coincide, or they are tangent which means they intersect in exactly one point which is on the beach line. They cannot coincide, since $p_i \neq p_j$, so they must intersect in exactly one point. By Assumption 3.2 we have $(p_i)_y \neq (p_j)_y$ so $\beta_i(x) - \beta_j(x)$ is a second degree polynomial with discriminant

$$D = \frac{(p_x - q_x)^2 + (p_y - q_y)^2}{(p_y - \ell_y)(q_y - \ell_y)}. \tag{3.4}$$

Since $p_y, q_y > \ell_y$ the denominator is strictly positive, and since $p_i \neq p_j$ the numerator is also strictly positive, so $D > 0$. This means that β_i and β_j intersect in two different points, a contradiction.

The second possibility is that β_j appears in between two arcs. Let these arcs be part of parabolas β_i and β_k. Let q be the intersection point between β_i, β_j and β_k, and we assume that the arc on the beach line from β_i is to the left of q, and the arc from β_k is to the right of q, as in this figure:

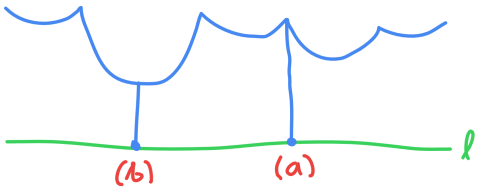


Now let C denote the circle $C_P(q)$ and note that it has p_i, p_j, p_k on its boundary, and it is tangent to ℓ. The cyclic order on C, starting at the point of tangency with ℓ and going clockwise is p_i, p_j, p_k. Now, we imagine an infinitesimal downward motion of ℓ while keeping C tangent to ℓ and p_j, we call the new circle C'. Now either p_i or p_k will be contained in the interior of C', say it's p_k like on the figure. Let c denote the center of C'. Then $\text{dist}(c, p_j)$ is equal to $\text{dist}(c, \ell)$, but

since p_k is contained in the interior of C' then $\text{dist}(c, p_k)$ is strictly smaller than $\text{dist}(c, p_j)$, which means that p_k is closer to ℓ than p_j, which means β_j cannot be on the beach line, a contradiction. □

Corollary 3.14. At any time the beach line consists of at most $2n - 1$ arcs.

Proof. We prove this by induction. The first site event adds a single arc, so for $n = 1$ there is at most $2n - 1 = 1$ arcs on the beach line. Now assume during the execution of the algorithm that we've seen $k < n$ of the n site events, and that the beach line consists of at most $2k - 1$ arcs. When we encounter a new site, we have seen that there are two cases:



In case (a) the have that an arc appears inbetween 2 existing arcs, increasing the total number by one. In case (b) an existing arc is split into two, and a new arc appears in between, which increases the total number by two. This means that after having seen $k + 1$ site events, there can be at most

$$(2k - 1) + 2 = 2(k + 1) - 1$$

parabolic arcs, which proves the claim. □

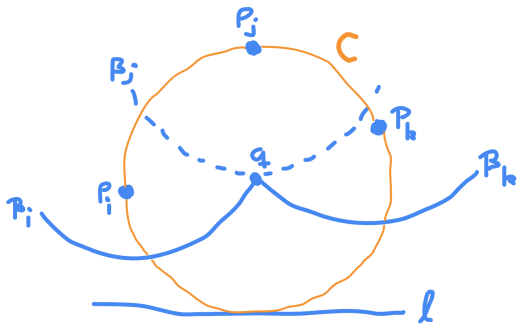
Now we've characterized exactly when new arcs appear on the beach line. We now turn to the question of when arcs disappear from the beach line. Assume we have at least 3 arcs on the beach line, name them $\alpha, \alpha', \alpha''$ and assume that α is adjacent to α', and α' is adjacent to α''. We assume that α' is the arc which is about to disappear. We first note that α and α'' cannot be a part of the same parabola. If this case the case, we'd be in the following situation:

Let β denote the parabola which α and α'' are a part of, and let β' be the parabola which α' is a part of. When α' is about to disappear, then there will be a time at which β and β' are tangent, and then we can reuse the contradiction

argument from the first part of the proof of Lemma 3.13. Thus α, α' and α'' are defined by 3 distinct sites $p_i, p_j, p_k \in P$. At the moment that α' disappears, then the three parabolas $\beta_i \supset \alpha$, $\beta_j \supset \alpha'$ and $\beta_k \supset \alpha''$ intersect in a single point q. We note that

$$\text{dist}(q, \ell) = \text{dist}(q, p_i) = \text{dist}(q, p_j) = \text{dist}(q, p_k).$$

So there is a circle C with center q passing through p_i, p_j, p_k which is tangent to ℓ at its lowest point. The situation is illustrated as follows:



We claim that $C = C_P(q)$. Assume for the sake of a contradiction that there is a site p inside the interior of C. Then

$$\text{dist}(p, q) < \text{dist}(q, \ell). \tag{3.5}$$

Now note the following characterization of being on the beach line: A point r is on the beach line if $\text{dist}(r, \ell) = \text{dist}(r, p_i)$ for all $i \in \mathcal{I}$ and $\text{dist}(r, \ell) < \text{dist}(r, p_j)$ for all $j \in \mathcal{J}$, where $\mathcal{I}$ describes those indices i where $r \in \beta_i$ and $\mathcal{J}$ describes those indices where $r \notin \beta_j$. By assumption q is on the beach line, since it is a point on all of $\alpha, \alpha', \alpha''$ but (3.5) contradicts the characterization we just gave of the beach line. So it must be the case that $C = C_P(q)$. Now note that

$$\{p_i, p_j, p_k\} \subset \partial C_P(q),$$

so Theorem 2.6 (i) gives us that q is a vertex of $\text{Vor}_G(P)$. Compare this to the fact that breakpoints trace out $\text{Vor}_G(P)$ as we proved earlier. This means that when two breakpoints meet and an arc disappears from the beach line, then two edges of $\text{Vor}(P)$ meet at a vertex. We call the event when ℓ reaches the lowest point of a circle through three sites defining consecutive arcs on the beach line a *circle event*. We have thus just proven:

Lemma 3.15. The only way in which an existing arc can disappear from the beach line is through a circle event.

(TODO: Prove Lemma 7.8: Every Voronoi vertex is detected by means of a circle event. (p. 157))

Chapter 4

Data structures for Fortune's algorithm

During the algorithm we will need three data structures:

- A **priority queue** $\mathcal{Q}$ for keeping track of the site and circle events.
- A **doubly-connected edge list (DCEL)** $\mathcal{D}$ for keeping track of the current state of the Voronoi diagram. See Definition 2.7. This will be updated after each site and circle event.
- A self-balancing **binary search tree (BST)** $\mathcal{T}$ for keeping track of the breakpoints and arcs on the beach line.

We explain them in detail in the next sections.

4.1 Priority queue

The priority queue stores the site and circle events, and enables the algorithm to handle them in order. Each element in the priority queue has a priority. For a site event the y-value of the point describes the priority, and for a circle event the priority is given by the y-value of the lowest point of the center of the circle which describes the event. Site events also store a pointer to the site, and circle events also store the center of its definining circle and a pointer to the arc in $\mathcal{T}$ which is disappearing.

For the implementation of the priority queue we will use a binary heap. These are described in CLRS (TODO: Ref), and the implementation has been taken from (TODO: Ref).

4.2 Binary search tree

We will store the current configuration of the beach line in a binary search tree, which has some additional information stored:

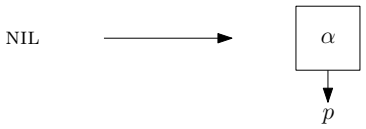
- Leaves correspond to arcs on the beach line. Every leaf has a pointer to a site in P, and it also has a pointer to a potential circle event at which the arc will disappear. If no circle event has been detected yet, the pointer will simply be NIL. Every arc will also store .leftArc and .rightArc pointers to the two arcs that surround it, so that we get a doubly linked list of arcs that are currently on the beach line. These pointers are NIL if the leaf has no neighbour in that particular direction. In our figures the leaves will be depicted by squares.

- Internal nodes correspond to breakpoints on the beach line. Every breakpoint stores an ordered pair (p, q), where p and q are sites in P. The order is important since the intersection of the hyperbolas defined by p and q consists of two points, and the order lets us tell these breakpoints apart. If we consider the beach line as running from the left to the right, then at every breakpoint an arc is leaving, and another is entering it. Thus the tuple (p, q) tells us that we are interested in the breakpoint at which an arc pointing to p leaves, and an arc pointing to q is entering. In our figures the internal nodes will be depicted by circles.

The binary tree will only be updated at site and circle events. First we describe what happens at a site event.

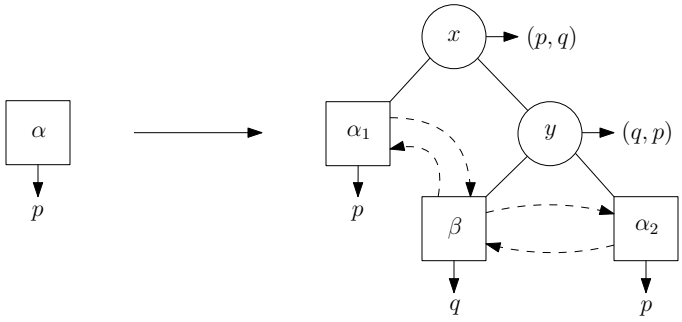
4.2.1 Inserting at site events

During the first site event, the tree will be empty (we say it is NIL), so to add an arc α to it, we simply turn the tree into a leaf, which describes the new arc, and we have it point to the first site p, illustrated as follows:



Now we look at the general case. We assume that α is an arc on the beach line, which points to a point p, and that we at a site event discover a new point q which is located below the arc α (e.g. a vertical line going through q intersects α). At this site event, the arc α will be split into two arcs α_1 to the left, and α_2 to the right, and two breakpoints x and y with $x \leq y$ will be introduced. We

update the tree locally as illustrated in the figure below:

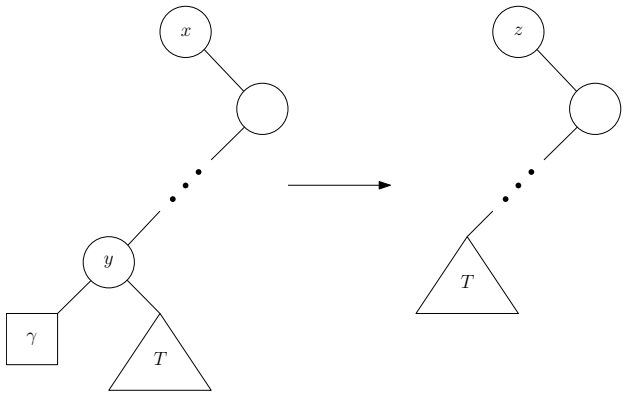


The leaf α gets replaced by the tree on the right. The dashed arrows represent the pointers for our doubly linked lists of arcs, and not depicted but also necessary is that we need to connect α_1 to α.leftArc and connect α_2 to α.rightArc by setting the appropriate pointers.

4.2.2 Deleting at circle events

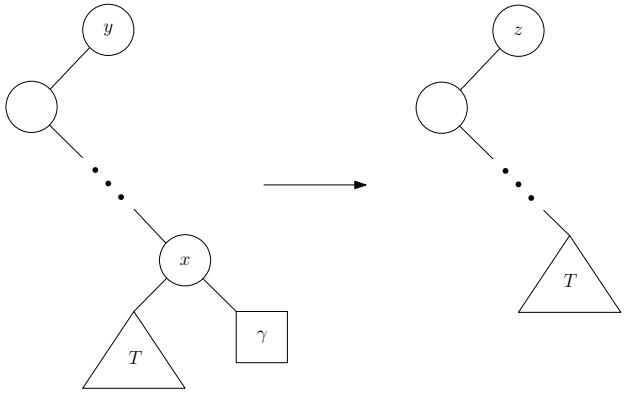
At a circle event two breakpoints x and y, with $x \leq y$, converge into a single breakpoint z and an arc γ disappears. We assume that x points to the tuple (p_i, p_j) and y points to the tuple (p_j, p_k). Then we make z point to the tuple (p_i, p_k).

In terms of the tree we must modify, the fact that the breakpoints x and y are converging means that y is the successor of x. What this means for the structure of the tree depends on whether x is an ancestor of y, or if y is an ancestor of x. First, we assume that x is an ancestor of y. Then y is the lowest internal node on the left spine of x.right, and our modification is as illustrated:



We replace y by $T = y$.right, effectively removing y and γ. Then we replace x by z. This way the tree stays a binary tree, and the new structure of the beach line is now rightfully represented.

On the other hand, if y is an ancestor of x, then x is the lowest internal node on the right spine of $y.\mathsf{left}$. We then make the following modification:



We replace x by $T = x.\mathsf{left}$, effectively removing x and γ. Then we replace y by z. Again, this way the tree stays a binary tree, and the new structure of the beach line is now rightfully represented.

We would like the tree to be balanced in order for search to be fast, so we need to balance the tree after inserting and deleting. We'll look at an approach to do this in the next section.

4.3 Balancing the BST by using a treap

There are multiple viable strategies for balancing a binary search tree. In this section we look at a particular strategy which utilizes randomness. We will introduce the treap data structure, which is a randomized self-balancing binary search tree. The presentation follows the paper (TODO: Ref), but only describes the things that we will need.

Definition 4.1 (Treap)**.** Let T be a tree where each node $x \in T$ has properties

- $x.\mathsf{left}$ is the left subtree of x,
- $x.\mathsf{right}$ is the right subtree of x,
- $x.\mathsf{key} \in \mathbb{R}$,
- $x.\mathsf{priority} \in [0, 1]$.

We say that T is a *treap* if

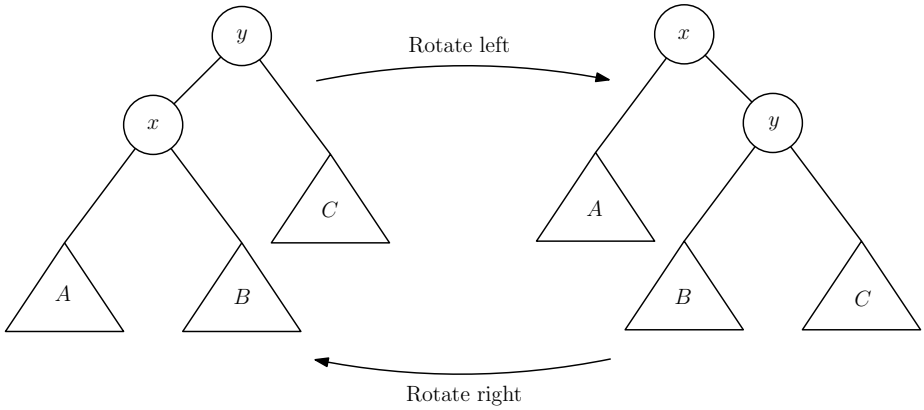
(i) T is a binary tree with respect to .key. That is, for every $x \in T$ we have

$$\forall y \in x.\mathsf{left}\colon y.\mathsf{key} \leq x.\mathsf{key},$$
$$\forall y \in x.\mathsf{right}\colon y.\mathsf{key} \geq x.\mathsf{key}.$$

(ii) T is a max-heap with respect to .priority, that is for each $x, y \in T$:

$$x \text{ is the parent of } y \implies x.\mathsf{priority} \geq y.\mathsf{priority}.$$

Definition 4.2 (Left and right rotations)**.** Given a tree T and two nodes $x, y \in T$ with subtrees A, B, C the operations *rotate left* and *rotate right* are given as follows:



From the diagram it is immediate that rotations preserve the binary tree property, but if T has a .priority property then the order on $x.\mathsf{priority}$ and $y.\mathsf{priority}$ is reversed. Given a binary tree T with priorities we may then make sure it also has the max-heap property by making a finite sequence of left and right rotations, and thus we may turn it into a treap.

The basic operations on a treap are as follows:

- SEARCH(x): This is the same as for a binary tree.
- INSERT(x): At first the insertion is identical to that of a binary tree: first we search for a spot to insert the new element, such that it stays a binary tree after insertion. Once inserted however, it may be the case that the max-heap property is violated. To remedy this, we may rotate x up in the tree until the max-heap property is reestablished, or until we reach the root.
- DELETE(x): The strategy is to rotate x down until it becomes a leaf in a manner which preserves the property that every subtree is a treap, and then we remove the leaf. This is done as follows: when rotating down we have a choice of rotating x with the root y of the left subtree A, or the root z of the right subtree B. We choose to rotate x and y if $y.\mathsf{priority} > z.\mathsf{priority}$, otherwise we rotate x and z, and then it follows by recursion that the treap property eventually is preserved in the entire tree once x is a leaf, and then we clip away x.

Definition 4.3 (Randomized search tree)**.** We define a *randomized search tree* to be a treap T where the priorities are independent, identically distributed continuous random variables.

The main result we will work towards in this section is the following:

Theorem 4.4. A randomized search tree storing n items has the expected performance characteristics listed in the table below:

Performance measure	**Bound on expectation**
Search	$\mathcal{O}(\log n)$
Insertion	$\mathcal{O}(\log n)$
Deletion	$\mathcal{O}(\log n)$
Number of rotations per update	≤ 2

To prove this we will introduce some random variables and then we will work towards proving upper bounds for their expectations. The first random variables we introduce are:

- $D(x)$: the number of nodes on the path from x to the root.
- $SL(x)$ and $SR(x)$: the length of the right spine of the left subtree of x and the length of the right spine of the right subtree of x. By length of the left spine of a tree we mean the number of nodes we pass if we keep following the left pointer from the root, and similarly for the right spine.

Throughout this section we will deal with a treap T with nodes $x_1, x_2, \ldots, x_n$ where node x_i has associated key k_i and priority p_i, and $k_1 < k_2 < \cdots < k_n$. We now introduce the following indicator random variables:

$$A_{i,j} = \begin{cases} 1 & \text{if } x_i \text{ is an ancestor of } x_j \text{ in } T, \\ 0 & \text{otherwise.} \end{cases}$$

$$C_{i;\ell,m} = \begin{cases} 1 & \text{if } x_i \text{ is a common ancestor of } x_\ell \text{ and } x_m \text{ in } T, \\ 0 & \text{otherwise.} \end{cases}$$

Note that we consider each node an ancestor of itself. We then have:

Theorem 4.5. Let $1 \leq \ell \leq n$. Then

(i) $D(x_\ell) = \sum_{i=1}^{n} A_{i,\ell}$.

(ii) $SL(x_\ell) = \sum_{i=1}^{\ell-1} (A_{i,\ell-1} - C_{i;\ell-1,\ell})$.

(iii) $SR(x_\ell) = \sum_{i=\ell+1}^{n} (A_{i,\ell+1} - C_{i;\ell,\ell+1})$.

Proof. (i): Nodes on the path from x_ℓ to the root are exactly the nodes which x_ℓ has as ancestors.

(ii): First we assume that x_ℓ has a left subtree L. This has the nodes x_i with $i < \ell$. The lowest node on the right spine of L is $x_{\ell-1}$. This means that every node on the right spine of L is an ancestor of $x_{\ell-1}$. Nodes in L outside the right spine are not ancestors of $x_{\ell-1}$. Since none of the nodes in L are ancestors of x_ℓ we have that $C_{i;\ell-1,\ell} = 0$ for all $i < \ell$, and hence the formula holds.

Now assume that x_ℓ has no left subtree. If $\ell = 1$ then the sum correctly evaluates to 0. If $\ell > 1$ then it must be the case that $x_{\ell-1}$ is an ancestor of x_ℓ, and then every ancestor of $x_{\ell-1}$ is a common ancestor of $x_{\ell-1}$ and x_ℓ, so the formula again correctly evaluates to 0.

(iii): This argument is symmetrical to the one for (ii). □

If we let $a_{i,j} = \mathbb{E}[A_{i,j}]$ and $c_{i;\ell,m} = \mathbb{E}[C_{i;\ell,m}]$ then by linearity of expectation we get:

Corollary 4.6. Let $1 \leq \ell \leq n$ and let $\ell < m$. Then

(i) $\mathbb{E}[D(x_\ell)] = \sum_{i=1}^{n} a_{i,\ell}$.

(ii) $\mathbb{E}[SL(x_\ell)] = \sum_{i=1}^{\ell-1}(a_{i,\ell-1} - c_{i;\ell-1,\ell})$.

(iii) $\mathbb{E}[SR(x_\ell)] = \sum_{i=\ell+1}^{n}(a_{i,\ell+1} - c_{i;\ell,\ell+1})$.

Our analysis has now been reduced to determining the expectations $a_{i,j}$ and $c_{i;\ell,m}$. Now, if X is an indicator random variable, then

$$\mathbb{E}[X] = \mathsf{Pr}(X = 1),$$

so we get that

$$a_{i,j} = \mathsf{Pr}(A_{i,j} = 1) = \mathsf{Pr}(x_i \text{ is an ancestor of } x_j)$$

and

$$c_{i;\ell,m} = \mathsf{Pr}(C_{i;\ell,m} = 1) = \mathsf{Pr}(x_i \text{ is a common ancestor of } x_\ell \text{ and } x_m).$$

Determining these probabilities is made possible through the ancestor lemma:

Lemma 4.7 (Ancestor lemma)**.** Assuming that all priorities are distinct, then x_i is an ancestor of x_j in T if and only if $p_i \geq p_h$ for all h in between and including i and j.

Proof. Let x_m be the item with the highest priority in T. Let

$$L = \{x_\nu \mid 1 \leq \nu < m\} \quad \text{and} \quad R = \{x_\mu \mid m < \mu \leq n\}.$$

Note that since x_m is the node with the highest priority in T it is actually the root, so we may consider L and R as the left and right subtrees of x_m. Note that L and R are treaps also.

For every $x_\ell \in L$ we have that x_m is an ancestor of x_ℓ and $p_m \geq p_h$ for all $\ell \leq h \leq m$. Thus it follows that any pair with x_m and $x_\ell \in L$ satisfy the ancestor characterization. Similarly every pair with x_m and any node in R satisfies the characterization. Now, note that a pair with $x_i \in L$ and $x_j \in R$ trivially satisfy the characterization, as they are not ancestor related.

We may then by recursion use the same argument on L and R, and this way we end out showing that the characterization is true for all pairs in T. □

Lemma 4.8 (Common ancestor lemma)**.** Let $1 \leq \ell, m, i \leq n$ with $\ell < m$. Assuming that all priorities are distinct, then x_i is a common ancestor of x_ℓ and x_m in T if and only if

$$p_i = \max\{p_\nu \mid \min\{i, \ell, m\} \leq \nu \leq \max\{i, \ell, m\}\}. \tag{4.1}$$

Proof. Equation (4.1) is equivalent to the following cases:

- $p_i = \max\{p_\nu \mid i \leq \nu \leq m\}$ if $1 \leq i \leq \ell$.
- $p_i = \max\{p_\nu \mid \ell \leq \nu \leq m\}$ if $\ell \leq i \leq m$.
- $p_i = \max\{p_\nu \mid \ell \leq \nu \leq i\}$ if $m \leq i \leq n$.

The fact that the lemma is true for each of these cases is a direct consequence of the ancestor lemma. □

Corollary 4.9. $a_{i,j} = \dfrac{1}{|i-j|+1}$.

Proof. By the ancestor lemma x_i is an ancestor of x_j if and only if

$$p_i = \max\{p_h \mid \min\{i, j\} \leq h \leq \max\{i, j\}\}.$$

Since the p_i are independent and identically distributed continuous random variables, this happens with probability

$$\frac{1}{|\{h \in \mathbb{N} \mid \min\{i, j\} \leq h \leq \max\{i, j\}\}|} = \frac{1}{|i-j|+1}.$$

□

Corollary 4.10. $c_{i;\ell,m} = \dfrac{1}{\max\{i, \ell, m\} - \min\{i, \ell, m\} + 1}$.

Proof. By the common ancestor lemma x_i is a common ancestor of x_ℓ and x_m if and only if

$$p_i = \max\{p_\nu \mid \min\{i, \ell, m\} \leq \nu \leq \max\{i, \ell, m\}\}.$$

Like in the proof of Lemma 4.9 we then use that the p_i are i.i.d to conclude that the probability is the reciprocal of the cardinality of the set we are taking the maximum of, and this cardinality is $\max\{i, \ell, m\} - \min\{i, \ell, m\} + 1$. □

Now we are ready to find upper bounds on the expectations of the quantities we are interested in. In order to do this, we will need the harmonic numbers, which are are given by $H_n = \sum_{i=1}^{n} \frac{1}{i}$. Their crucial property is the inequalities

$$\ln n < H_n < 1 + \ln n$$

for all $n > 1$. (TODO: Find proof or reference.)

Theorem 4.11. Let $1 \leq \ell \leq m$. In a randomized search tree with n nodes the following expectations hold:

(i) $\mathbb{E}[D(x_\ell)] = H_\ell + H_{n+1-\ell} - 1 < 1 + 2 \cdot \ln n = \mathcal{O}(\log n)$.

(ii) $\mathbb{E}[SL(x_\ell)] = 1 - \frac{1}{\ell}$.

(iii) $\mathbb{E}[SR(x_\ell)] = 1 - \frac{1}{n+1-\ell}$.

Proof. For (i) we get

$$\begin{aligned}
\mathbb{E}[D(x_\ell)] &= \sum_{i=1}^{n} a_{i,\ell} \\
&= \sum_{i=1}^{n} \frac{1}{|i-\ell|+1} \\
&= \sum_{i=1}^{\ell} \frac{1}{\ell-i+1} + \sum_{i=\ell}^{n} \frac{1}{i-\ell+1} - 1 \\
&= \sum_{i=1}^{\ell} \frac{1}{i} + \sum_{i=1}^{n+1-\ell} \frac{1}{i} - 1 \quad \text{(Reverse left sum and swap index in right)} \\
&= H_\ell + H_{n+1-\ell} - 1 \\
&< (1 + \ln \ell) + (1 + \ln(n+1-\ell)) - 1 \\
&\leq 1 + 2 \cdot \ln n.
\end{aligned}$$

For (ii) we have

$$\begin{aligned}
\mathbb{E}[SL(x_\ell)] &= \sum_{i=1}^{\ell-1} (a_{i,\ell-1} - c_{i;\ell-1,\ell}) \\
&= \sum_{i=1}^{\ell-1} \left(\frac{1}{|i-(\ell-1)|+1} - \frac{1}{\max\{i,\ell-1,\ell\} - \min\{i,\ell-1,\ell\} + 1} \right) \\
&= \sum_{i=1}^{\ell-1} \left(\frac{1}{\ell-i} - \frac{1}{\ell-i+1} \right) \\
&= \frac{1}{\ell-(\ell-1)} - \frac{1}{\ell} \quad \text{(The above is a telescoping sum)} \\
&= 1 - \frac{1}{\ell}.
\end{aligned}$$

For (iii) we note that the proof is basically the same as for (ii) so we omit it. □

By combining Lemma 4.12 and Lemma 4.13 below we obtain a proof of Theorem 4.4, which will complete our analysis of treaps:

Lemma 4.12. The operations SEARCH, INSERT and DELETE take expected $\mathcal{O}(\log n)$ time.

Proof. Searching for the spot for an element x takes expected

$$\mathbb{E}[D(\text{PRED}(x)) + D(\text{SUCC}(x))] = \mathcal{O}(\log n)$$

time, where $\text{PRED}(x)$ finds a predecessor for x, e.g. finding the element in T with the largest key less than or equal to $x.\mathsf{key}$, and $\text{SUCC}(x)$ finds a successor for x, e.g. finding the element in T with the smallest key greater than or equal to $x.\mathsf{key}$.

For insertion we first need to perform a search, and then afterwards the number of times we rotate is at most the length of the path traversed during search. The bound thus follows from the bound for search.

Since a deletion is basically just the reversal of an insertion the bounds follow. □

Lemma 4.13. Let x_ℓ be an element of a treap which is to be deleted, or an element which has just been inserted into a treap. Let $R(x_\ell)$ be the number of rotations which were needed during the update of the treap. Then $\mathbb{E}[R(x_\ell)] \leq 2$.

Proof. First, we note that since in terms of rotations a deletion is an exact reversal of an insertion it suffices to analyze the number of rotations that occur during a deletion. First, we note that if a node x is right-rotated down, then $SL(x)$ decreases by one, and if a node x is left-rotated down, then $SR(x)$ decreases by one. Once x has been rotated down such that it is a leaf y, we have $SL(y) = SR(y) = 0$. It follows by recursion that $R(x) = SL(x) + SR(x)$. Then linearity of expectation and Theorem 4.11 gives us that

$$\mathbb{E}[R(x_\ell)] = \mathbb{E}[SL(x_\ell)] + \mathbb{E}[SR(x_\ell)] = 2 - \left(\frac{1}{\ell} + \frac{1}{n+1-\ell}\right) \leq 2.$$

□

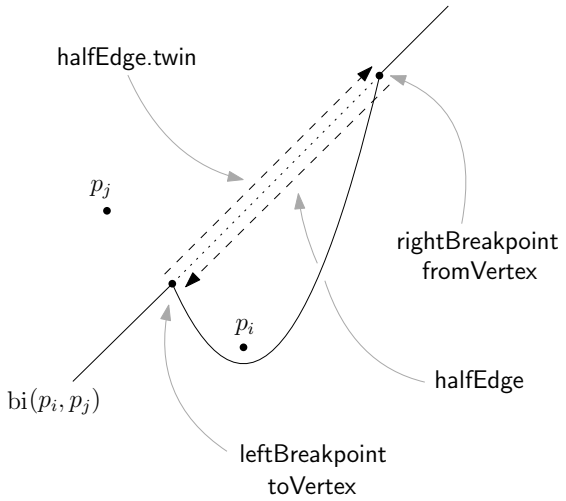
4.4 Doubly-connected edge list

To store the actual Voronoi diagram we use a DCEL, which we defined in Chapter 2. We the details of how we intersect the half-infinite edges with a bounding box are explained in the next chapter, since it relies on how the general algorithm is described.

When initializing $\mathcal{D}$ we create a face for every $p_i \in P$, and this face will describe $\mathcal{V}(p_i)$. We now describe how we modify the DCEL $\mathcal{D}$ at site and circle events.

4.4.1 Updating DCEL at site event

For the first site event where $\mathcal{T}$ is empty we do nothing. Now, consider a site event where a new point p_i is introduced, and assume that a previously discovered point p_j describes the arc which p_i is under. Just above p_i we create two new breakpoints leftBreakpoint and rightBreakpoint which initially are at the same point, but as the sweep line continues its downward motion, a new edge of the Voronoi diagram starts being traced out:

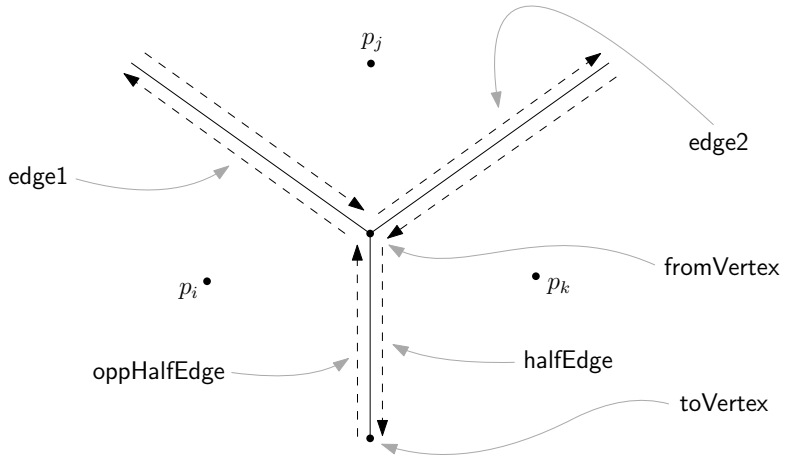


To describe this in $\mathcal{D}$ we create two new vertices fromVertex and toVertex, one for each breakpoint, and for now we have the breakpoints dictate the position of the vertices – that is we attach fromVertex to rightBreakpoint, and we attach toVertex to leftBreakpoint. We then create a new half-edge halfEdge, which has fromVertex as its .origin, and then we create a matching twin for halfEdge which has toVertex as its .origin property. We also connect halfEdge to the face which describing $\mathcal{V}(p_i)$, and halfEdge.twin to the face describing $\mathcal{V}(p_j)$.

The orientations have been chosen such that halfEdge and its future predecessors and successors will form a counterclockwise cycle around the Voronoi cell $\mathcal{V}(p_i)$.

4.4.2 Updating DCEL at circle event

Consider the circle event defined by the sites p_i, p_j, p_k where two breakpoints have just intersected each other, the first breakpoint sliding along $\text{bi}(p_i, p_j)$ and the second sliding along $\text{bi}(p_j, p_k)$. Let vertex1 describe the DCEL vertex which was following the breakpoint along $\text{bi}(p_i, p_j)$ and let vertex2 describe the DCEL vertex which was following the breakpoint along $\text{bi}(p_j, p_k)$. Let edge1 denote vertex1.edge and let edge2 denote vertex2.edge.



Note edge1 and edge2 in the figure. Now, since vertex1 and vertex2 are colliding, they will combine into a new breakpoint, and therefore we must detach them from the breakpoints they were following before. We must also delete one of them from our list of vertices, say vertex2, and then copy vertex1 to be the new .origin of edge2.

Let fromVertex denote vertex1, and create a new DCEL vertex toVertex and a new twin pair of half-edges halfEdge and oppHalfEdge. Attach toVertex to the new breakpoint, and let fromVertex be the origin of halfEdge, and let toVertex be the origin of oppHalfEdge.

Now, since edge1 and edge2 have been connected, we need to set some .next and .prev pointers to make sure they're also connected in the DCEL structure. We will say that we *pair* two DCEL edges (u, v) by setting $u.\text{next} = v$ and $v.\text{prev} = u$. We make the following pairs:

$$(\text{edge1.twin}, \text{edge2}), (\text{oppHalfEdge}, \text{edge1}) \text{ and } (\text{edge2.twin}, \text{halfEdge}).$$

Finally, we connect halfEdge to the face describing $\mathcal{V}(p_k)$ and oppHalfEdge to the face describing $\mathcal{V}(p_i)$.

Again, the orientations have been chosen such that the half-edges will be part of counterclockwise cycles around their associated Voronoi cell.

Chapter 5

Description of Fortune's algorithm

We are now ready to describe Fortune's algorithm. We start with describing an overview of the algorithm, and then in the next section we describe some of the details thoroughly – so anytime the algorithm says "see detail n", then this detail can be found in the next section.

Algorithm 5.1. VoronoiDiagram(P)
Input: A set $P = \{p_1, \ldots, p_n\}$ of point sites in the plane.
Output: The Voronoi diagram Vor(P) given inside a bounding box in a doubly-connected edge list $\mathcal{D}$.

1. Initialize the event queue $\mathcal{Q}$ with a site event for every point in P, initialize the beach line tree $\mathcal{T}$ to be NIL, and let the DCEL $\mathcal{D}$ be empty.
2. Repeat the following until $\mathcal{Q}$ is empty:
   i. Remove the event e with the largest y-coordinate from $\mathcal{Q}$.
   ii. If e is a site event call HandleSiteEvent(e).
   iii. If e is a site event call HandleCircleEvent(e).
3. At this point the internal nodes in $\mathcal{T}$ represent the infinite edges of Vor(P). Compute a bounding box B which contains all points in P, as well as all the vertices of Vor(P), which are contained in $\mathcal{D}$. Intersect the infinite edges in $\mathcal{T}$ with B and let these intersection points be new vertices in $\mathcal{D}$. Add new edges and pointers to make sure we still have a proper DCEL structure. (See detail #)

Procedure 5.2. HandleSiteEvent(e)

1. Let p_i denote the site that e points to.
2. If $\mathcal{T}$ = NIL then let $\mathcal{T}$ store the single arc that is described by p_i and return.

3. Otherwise, $\mathcal{T} \neq$ NIL. Search in $\mathcal{T}$ for the arc α vertically above p_i, that is the arc at which the vertical line through p_i intersects the beach line. (See detail #)

4. If α has a pointer to a circle event e', then remove e' from $\mathcal{Q}$, as this circle event is now a false alarm since α is about to disappear earlier than we initially thought.

5. Create the new arc β defined by p_i and insert it into $\mathcal{T}$ as described in Section 4.2.1. Update $\mathcal{D}$ by creating the new half-edges which will be traced out by the two new breakpoints as described in Section 4.4.1.

6. Check the triple of consecutive arcs where the new arc for p_i is the left arc to see if the breakpoints converge. If so, insert the circle event into $\mathcal{Q}$ and add pointers between the node in $\mathcal{T}$ and the node in $\mathcal{Q}$. Do the same for the triple where the new arc is the right arc. (See detail #)

Procedure 5.3. HANDLECIRCLEEVENT(e)

1. Let α be the arc pointed to by e, which is about to disappear from the beach line.

2. Delete all circle events from $\mathcal{Q}$ which involve α: The one where α is the middle arc has already been deleted, and the other two possible circle events where α is the left and right arc respectively can be found through α's .leftArc and .rightArc pointers. (See detail #)

3. Delete α from $\mathcal{T}$, how this is done is described in Section 4.2.2.

4. Add the center c of the circle describing e as a new vertex of $\mathcal{D}$. Connect the half-edges in $\mathcal{D}$ that converge at e, and create a new half-edge which starts at c and setup the appropriate pointers. The details are given in Section 4.4.2.

5. As α disappears from the beach line, we get new triples of consecutive arcs which might have converging breakpoints that can lead to a circle event. Check these and add circle events if needed. (See detail #)

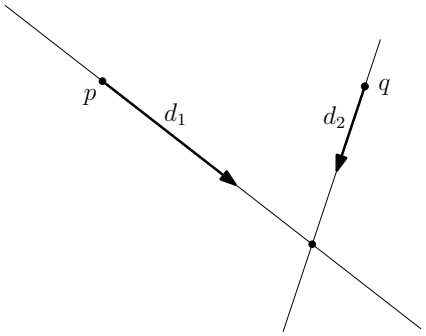
5.1 Details

Detail 1: Intersecting lines, rays and segments

As a subroutine in several steps during the algorithm we will need to intersect line segments or rays with each other. We start by describing a solution to this in general. This detail assumes that the reader is familiar with basic linear algebra. We want to find the intersection between 2 lines. We parametrize the lines as follows:

$$\gamma_1(t) = p + td_1 \quad \text{and} \quad \gamma_2(s) = q + sd_2,$$

where p and q are points on the lines, and d_1 and d_2 are direction vectors which tells us which way the lines point. The situation is illustrated as follows:



To find an intersection point, we must find $s, t \in \mathbb{R}$ such that

$$\gamma_1(t) = \gamma_2(s).$$

That is, we want to solve

$$p + td_1 = q + sd_2.$$

This can be rewritten into the matrix equation

$$A \begin{pmatrix} s \\ -t \end{pmatrix} = q - p,$$

where $A = \begin{pmatrix} | & | \\ d_1 & d_2 \\ | & | \end{pmatrix}$ is the 2×2 matrix which has d_1 and d_2 as left and right columns, respectively. The equation system has a unique solution if d_1 and d_2 are linearly independent, and if they are, the solution is given by

$$\begin{pmatrix} s \\ -t \end{pmatrix} = A^{-1}(q - p).$$

This linear independence property is equivalent to checking that the determinant $\det(A)$ is non-zero. So, in order to check if two lines intersect, we first check if $\det(A) \neq 0$. If not, we say the lines don't intersect. Otherwise, they intersect, and we use the above solution to find the intersection point.

Now in the case of line segments and rays, we also need some constraints on s and t, it is not enough that the lines themselves intersect.

To intersect line segments, where the first line segment is given as the points between p_1 and p_2, and the second line segment is given as the points between q_1 and q_2, then we let $p = p_1$ and $q = q_1$ and then set $d_1 = p_2 - p_1$ and $d_2 = q_2 - q_1$ and then we solve the equation system as above. If there is a solution to the equation system, then we also need to check that $s \in [0, 1]$ and $t \in [0, 1]$ in order for the intersection point to lie on both of the line segments.

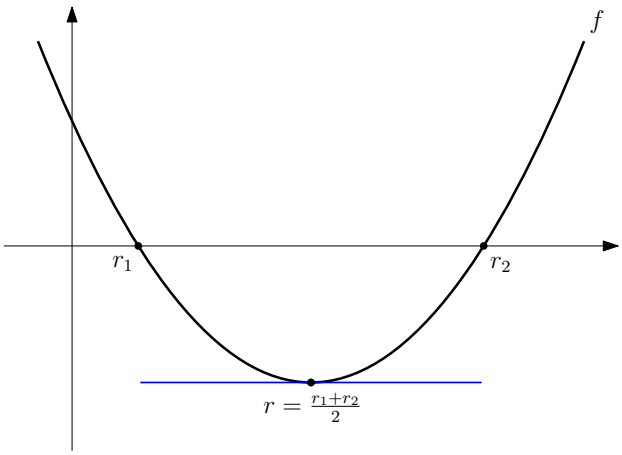
Finally, we look at intersecting two rays. We describe the rays as the lines at the start of this detail, but now we have the requirement that $s, t \geq 0$, which gives us that the rays start at p and q, and then they shoot out in the direction of d_1 and d_2, respectively.

Detail 2: Choose breakpoint based on the ordering of tuple

Let p_i and p_j be two sites and let β_i and β_j be the hyperbolas that they describe. If a breakpoint stores the tuple (p_i, p_j) then we want a way to find the x coordinate of that breakpoint. Since the intersection of two hyperbolas may contain 2 intersection points, we need to pick the correct one. We already described this when discussing internal tree nodes at the start of Section 4.2, but let's recap: The order is important since the intersection of the hyperbolas defined by p_i and p_j consists of two points, and the order lets us tell these breakpoints apart. If we consider the beach line as running from the left to the right, then at every breakpoint an arc is leaving, and another is entering it. Thus the tuple (p_i, p_j) tells us that we are interested in the breakpoint at which an arc pointing to p_i leaves, and an arc pointing to p_j is entering the beach line. We will need the following result:

Proposition 5.4. Let $f(x) = ax^2 + bx + c$ be a polynomial with discriminant $D > 0$ with roots $r_1 < r_2$. Then $r = \frac{1}{2}(r_1 + r_2)$ is the only solution to $\frac{df}{dx}(r) = 0$ and the expressions $\frac{df}{dx}(r_1)$ and $\frac{df}{dx}(r_2)$ are non-zero and have opposite signs.

This fact can be visualized as follows:



Proof of Proposition 5.4. It is well-known that we may factor f as follows:

$$f = a(x - r_1)(x - r_2) = ax^2 - a(r_1 + r_2)x + ar_1r_2.$$

Since two polynomials are equal if and only if their coefficients are equal we get $b = -a(r_1 + r_2)$, which gives us

$$\frac{df}{dx}(r) = 2ar + b = 2a\left(\frac{r_1 + r_2}{2}\right) - a(r_1 + r_2) = 0.$$

This is the only solution since $\frac{df}{dx}$ is a first degree polynomial. Now note that $\frac{d^2f}{dx^2}(x) = 2a \neq 0$ and $r_1 < r < r_2$ which gives us that

$$\operatorname{sgn}\left(\frac{df}{dx}(r_1)\right) = -\operatorname{sgn}\left(\frac{df}{dx}(r_2)\right) \neq 0.$$

□

When intersecting two of the paraboals of the beach line, we will find two intersection points, because of our assumptions. Proposition 5.4 then gives us that at these intersection points r_1, r_2 we have that

$$\begin{cases} \frac{d(\beta_i - \beta_j)}{dx}(r_k) \neq 0 \text{ for } k = 1, 2 \\ \operatorname{sgn}\left(\frac{d(\beta_i - \beta_j)}{dx}(r_1)\right) = -\operatorname{sgn}\left(\frac{d(\beta_i - \beta_j)}{dx}(r_2)\right) \end{cases}$$

We then want to locate a specific breakpoint between two arcs, and the above will help us to do this.

To intersect the two parabolas β_i and β_j we write

$$(\beta_i - \beta_j)(x) = ax^2 + bx + c,$$

where (for $p = p_i$, $q = p_j$, $h_p = p_y - \ell_y$ and $h_q = q_y - \ell_y$)

$$\begin{aligned} a &= \frac{1}{2}\left(\frac{1}{h_p} - \frac{1}{h_q}\right), \\ b &= \frac{q_x}{h_q} - \frac{p_x}{h_p}, \\ c &= \frac{q_y(p_x^2 + p_y^2) - p_y(q_x^2 + q_y^2) + \ell_y(q_x^2 + q_y^2 - p_x^2 - p_y^2) + \ell_y^2(p_y - q_y)}{2h_p h_q}. \end{aligned}$$

The square root of the discriminant is then

$$d = \sqrt{b^2 - 4ac} = \sqrt{\frac{(p_x - q_x)^2 + (p_y - q_y)^2}{h_p h_q}}.$$

The x-values of the intersection points are then given by the well-known formulas

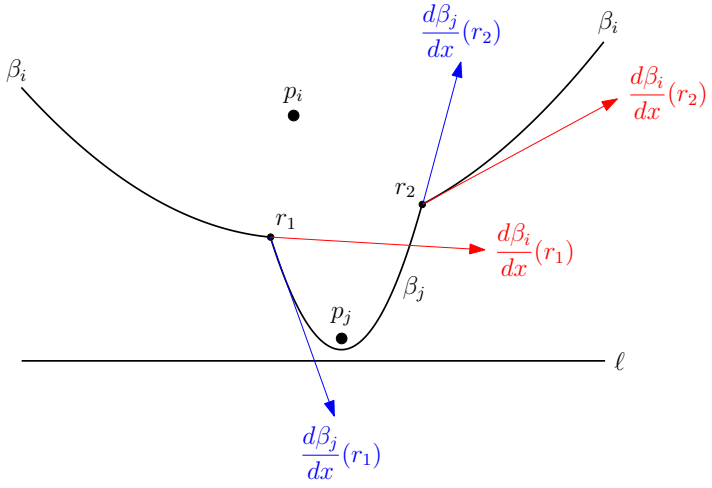
$$r_1 = \frac{-b - d}{2a}, \quad r_2 = \frac{-b + d}{2a},$$

which gives us the intersection points $q_1 = (r_1, \beta_i(r_1))$ and $q_2 = (r_2, \beta_i(r_2))$. Now, we want to find the breakpoint which at which an arc of β_i exits the beach line, and an arc of β_j enters the beach line. Proposition 5.4 gives us a

way of picking which one of q_1 and q_2 is the breakpoint that we need. For β_i to exit and β_j to enter, we need to pick k such that

$$\frac{d\beta_i}{dx}(r_k) > \frac{d\beta_j}{dx}(r_k).$$

This is illustrated in the following figure, with a slight abuse of notation:



Proposition 5.4 guarantees that either

$$\frac{d\beta_i}{dx}(r_1) > \frac{d\beta_j}{dx}(r_1) \text{ and } \frac{d\beta_i}{dx}(r_2) < \frac{d\beta_j}{dx}(r_2)$$

or

$$\frac{d\beta_i}{dx}(r_1) < \frac{d\beta_j}{dx}(r_1) \text{ and } \frac{d\beta_i}{dx}(r_2) > \frac{d\beta_j}{dx}(r_2),$$

so it is possible to make the right choice. Now, note that by some simple algebraic manipulations we have that

$$\frac{d\beta_i}{dx}(r_k) > \frac{d\beta_j}{dx}(r_k)$$

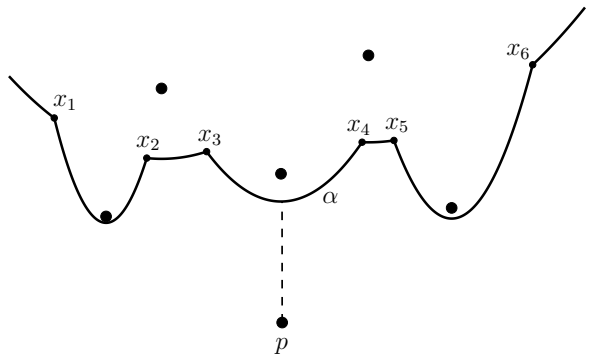
if and only if

$$(r_k - p_x)(q_y - \ell_y) > (r_k - q_x)(p_y - \ell_y).$$

This gives us a criterion for deciding which intersection point describes the breakpoint in question, and this is the criterion used in the implementation.

Detail 3: How to find the arc vertically above a point

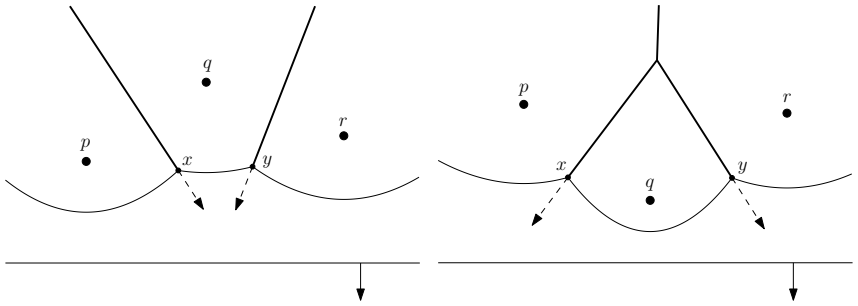
At a site event when we discover a new point p we want to find the arc α vertically above p, as illustrated here:



Let $x_1, x_2, \ldots, x_k$ denote the breakpoints on the beach line. These are stored as internal nodes in our BST $\mathcal{T}$. Since the keys for the internal nodes are the x-values of the breakpoints, we may locate the arc α using binary search in $\mathcal{T}$. Starting at an internal node x in $\mathcal{T}$ we visit its left subtree if $x.\mathsf{key} < p_x$, and we visit its right subtree if $x.\mathsf{key} \geq p_x$. The key property is computed at every check, since it is a function of the current position of the sweep line, see Detail 2 for how the key is computed. Eventually we will reach the leaf which stores the arc α.

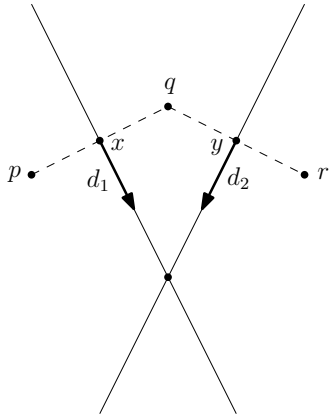
Detail 4: How to check if two breakpoints are converging

Let p, q, r be three sites from P which define 3 consecutive arcs on the beach line. Let x and y be two breakpoints, where x is sliding along $\mathrm{bi}(p, q)$ and y is sliding along $\mathrm{bi}(q, r)$ as we vary ℓ. We want to check whether x and y converge, and if so, what is the location of their intersection, and when during the sweep of ℓ will this occur. The two possible scenarios are illustrated below:



In the divergent case we include the case where the bisectors are collinear.

To check for convergence, we transform the problem into a problem of intersecting two rays. Let x and y denote the current location of the breakpoints, and let x' and y' denote the breakpoints new positions after moving the sweep line some arbitrary amount downwards, and then let $d_1 = x' - x$ and $d_2 = y' - y$. Then $s \mapsto x + sd_1$ and $t \mapsto y + td_2$ parametrize $\mathrm{bi}(p, q)$ and $\mathrm{bi}(q, r)$, respectively. The setup, in the case where the rays do intersect, looks like this:



Now, as we saw in Detail 1, then the two rays converge if $t \geq 0$ and $s \geq 0$.

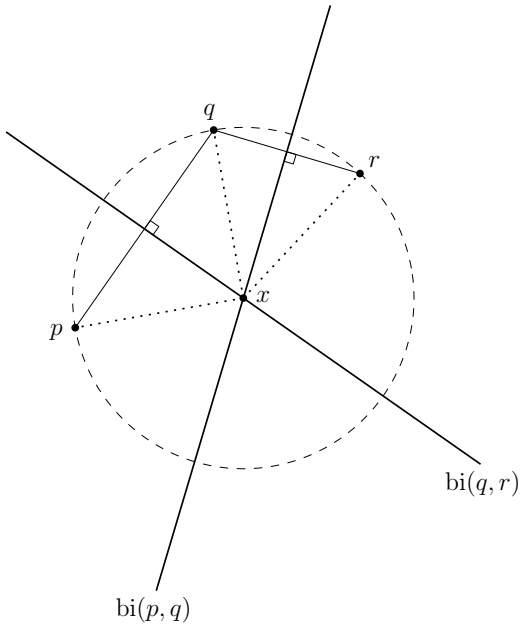
This can be interpreted as follows: If s is positive, then that means that x will hit y in the future, and likewise if t is positive, then y will hit x in the future. This is important as we are treating the events chronologically, and if x and y already intersected in the past (or the lines they describe, rather) then they can define no future circle event.

Detail 5: Finding a circle through 3 points

As a part of the algorithm, we need to find the circle C through 3 points p, q, r. It turns out if we intersect $\mathrm{bi}(p, q)$ and $\mathrm{bi}(q, r)$ we find the center of C, and then to find the radius we just need to find the distance from the center to one of the points. This is because if $x \in \mathrm{bi}(p, q) \cap \mathrm{bi}(q, r)$ then

$$\mathrm{dist}(x, p) = \mathrm{dist}(x, q) = \mathrm{dist}(x, r),$$

so x is exactly the center of a circle through p, q, r. This is illustrated in this figure:



To intersect the bisectors, we form the midpoints

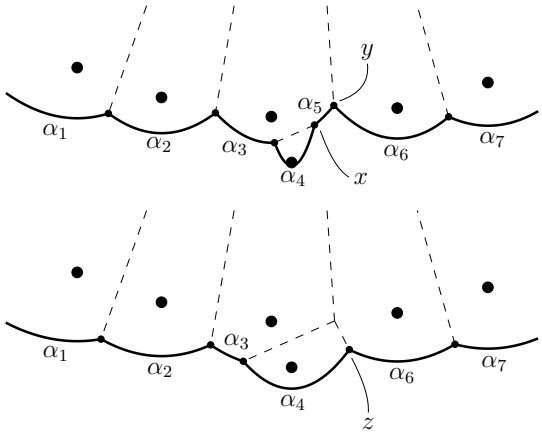
$$m_1 = \frac{1}{2}(p+q) \quad \text{and} \quad m_2 = \frac{1}{2}(q+r)$$

and then we let d_1 and d_2 denote $q-p$ and $r-q$ rotated 90 degrees counterclockwise. Then $s \mapsto m_1 + sd_1$ and $t \mapsto m_2 + td_2$ parametrize $\text{bi}(p,q)$ and $\text{bi}(q,r)$, respectively. Then we solve the linear system as in Detail 1.

Detail 6: Deleting false alarms during a circle event

At a circle event an arc disappears from the beach line, along with two breakpoints. Consider the following example, where at one point in time we have the arcs $\alpha_1, \alpha_2, \ldots, \alpha_7$ on the beach line along with the breakpoints x and y that α_5 lies inbetween, and then after a circle event the arc α_5 disappears after the

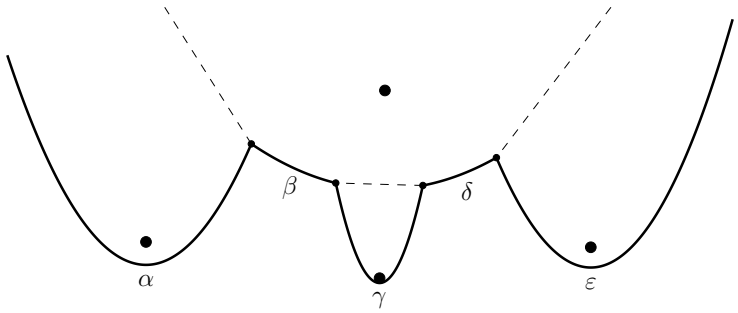
breakpoints x and y intersect and get replaced by a new breakpoint z:



When this happens, we have to remove the circle events that involve the breakpoints x and y merging with any other breakpoints. Since we set up a linked list of arcs, we can find the arcs α_4 and α_6 through the .leftArc and .rightArc pointers that α_5 has, and if these arcs point to a circle event, then we remove those circle events from $\mathcal{Q}$ if they exist. These are of course found before deleting α_5. This covers removing the false alarms, since we have removed circle events from every arc that surrounds x and y. This example is general enough to explain the general case.

Detail 7: Checking consecutive arcs for circle events

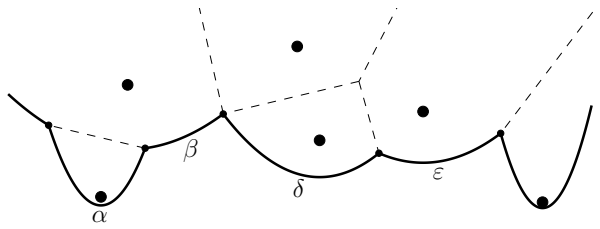
First we consider the case of a site event. Assume the beach line is made out of the arcs $\alpha, \beta, \gamma, \delta$ and ε, and assume that γ is the new arc we just created. Just a moment later the beach line looks something like this:



The arcs we need to check for converging breakpoints are then (α, β, γ) and $(\gamma, \delta, \varepsilon)$. To be more precise, if $p_\alpha, p_\beta, p_\gamma, p_\delta$ and p_ε denote the associated sites, then we need to check the convergence of the breakpoints defined by (p_α, p_β) and (p_β, p_γ), and to check the convergence of the breakpoints defined by (p_γ, p_δ)

and $(p_\delta, p_\varepsilon)$, and in case of convergence we must add the corresponding circle events to $\mathcal{Q}$.

Now we consider the case of a circle event. Assume the beach line just before the circle event contained the consecutive arcs $\alpha, \beta, \gamma, \delta$ and ε. At the circle event the arc γ then disappears, now connecting β and δ, leaving us with the following picture:



We now have triples of consecutive arcs which were not triples before, namely (α, β, δ) and $(\beta, \delta, \varepsilon)$. We must check these, and in case of convergence we must add the corresponding circle events to $\mathcal{Q}$.

Detail 8: Intersecting a bounding box with DCEL

a

5.2 Correctness

Lemma 5.5. Algorithm 5.1 can be implemented such that it runs in $\mathcal{O}(n \log n)$ time and uses $\mathcal{O}(n)$ storage.

Proof. (TODO: .) □

Chapter 6

Application: Computing the Delaunay triangulation

kjdfg

Appendix A

Notation

$X - Y$	Set difference
$\|X\|$	The number of elements in a finite set X.
$\iff$	If and only if
$\implies$	Implication
$\mathbb{R}$	The real numbers.
$\mathbb{R}^n$	The vector space of n-tuples of real numbers.
$\|\cdot\|$	Norm.
$\|\cdot\|_p$	The L^p norm.
$\|x\|$	Absolute value if x is a number.
$\mathrm{dist}(p, q)$	The distance between p and q, given by $\|p - q\|$.
$\langle \cdot, \cdot \rangle$	An inner product.
$\subset$	Subset (not strict, e.g. $A = B \implies A \subset B$).
P	A set of points $\{p_1, p_2, \ldots, p_n\}$ that we want to apply an algorithm to.
p_i	A point in P (see above).
n	If not otherwise specified, n is the number of points in P (see above).
$\mathrm{Vor}(P)$	The Voronoi diagram of P.
$\mathcal{V}(p_i)$	The ith Voronoi cell.
$\mathrm{Vor_G}(P)$	Refers to $\mathbb{R}^2 - \mathrm{Vor}(P)$.
$\mathcal{O}(f(n))$	Big O-notation.
$\mathrm{bi}(p, q)$	Bisector of p and q.
$h(p, q)$	Open half-plane containing p with $\mathrm{bi}(p, q)$ as boundary.
$\overline{X}$	The closure of a set $X \subset \mathbb{R}^n$, given by the union of X with its limit points.
$^\circ X$	The interior of a set $X \subset \mathbb{R}^n$, given by the union of all interior points of X.
∂X	The boundary of a set $X \subset \mathbb{R}^n$, given by $\overline{X} - {}^\circ X$.
$\overline{B_r(p)}$	$= \{x \in \mathbb{R}^n \mid \mathrm{dist}(x, p) \leq r\}$, the closed ball with center p and radius r.
$B_r(p)$	$= \{x \in \mathbb{R}^n \mid \mathrm{dist}(x, p) < r\}$, the open ball with center p and radius r.
$\partial B_r(p)$	$= \{x \in \mathbb{R}^n \mid \mathrm{dist}(x, p) = r\}$, the circle with center p and radius r.
$V(G)$	The set of vertices for the graph G.
$E(G)$	The set of edges for the graph G.
$\deg(v)$	The degree of a vertex v in a graph, e.g. the number of edges that touch v.