Voronoi To The World

Michael Betancourt

November 2025

Table of contents

Voronoi diagrams arise naturally in various spatial analyses. More importantly for this author, they also tend to be aesthetically attractive, making them useful for crafting compelling illustrations.

Efficiently constructing a Voronoi diagram, however, is not particularly straightforward. In this note I work through the geometric structure of Voronoi diagrams and an algorithm that leverages that structure to construct Voronoi diagrams with the maximum possible performance.

1 Tessellating A Plane

A **Voronoi diagram**, also known as a **Dirichlet tessellation**, is an organization of a plane into subsets defined by the nearest proximity to a given collection of points.

From what I can tell the standard reference for Voronoi diagrams, and other topics in computational geometry, is Berg et al. (2008). This text is often referred to as "the 4Ms", which I

can only infer refers to the fact that three out of the four authors share the first name "Mark" with the forth, non-Mark author being an honorary "M". I humbly suggest that "Mark, Marky, Mark, and the Funky Bunch" is more appropriate, but to each their own.

To formally define a Voronoi diagram consider a two-dimensional Euclidean plane $\mathbb{R}^2$ and a collection of **sites**

$$(s_1 = (x_1, y_1), \ldots, s_N = (x_N, y_N)) \subset \mathbb{R}^2.$$

Give a particular site s_n we can construct a subset of points that are closer to that site than any other site,

$$\mathsf{c}_n = \{p \in \mathbb{R}^2 \mid d(p, s_n) < d(p, s_{n'}) \text{ forall } n' \neq n\},$$

where $d(p, p')$ is a distance function. These subsets are also known as **cells**. The Voronoi diagram of $(s_1, \ldots, S_N)$ is the corresponding collection of these proximity-based subsets (Figure 1).

(a) (b)

Figure 1: (a) A Voronoi diagram defined by a collection of sites $(s_1, \ldots, s_N)$, here shown in red, is a collection of subsets, each containing points that are closer to one site than the others. (b) The boundary between these subsets consists of points that are equally distant to two or more sites at the same time.

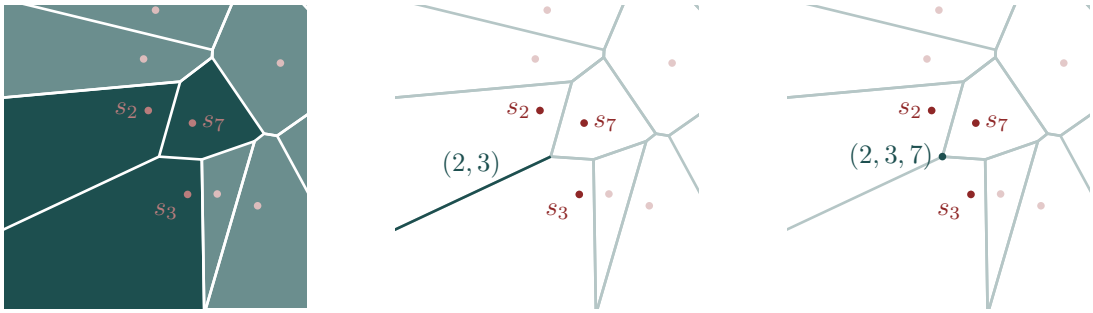
The standard Voronoi diagram considers the Euclidean distance function,

$$d(p, p') = \sqrt{(x - x')^2 + (y - y')^2}.$$

In theory, however, this construction can be generalized to other distance functions. At the same time Voronoi diagrams can be defined on higher-dimensional real spaces and even more general non-Euclidean ometric spaces.

A Voronoi diagram *almost* forms partition of a Euclidean plane. The snag is that points that are equally distant from two or more sites don't fall into any of the cells, but rather form a singular boundary between the cells. The combination of the Voronoi cells and the boundary between them, however, does form a proper partition.

Because each cell contains one, and only one, site, we can always label the cells by the associated site (Figure 2a). Similarly we can label the linear segment that forms the boundary between two neighboring cells by the sites associated with those two cells, and the points separating the corners of three or more neighboring cells by the sites associated with those cells (Figure 2b, Figure 2b).



(a) (b) (c)

Figure 2: (a) Each of the cells in a Voronoi diagram can be labeled with the unique site that it contains. (b, c) Similarly the boundaries between neighboring cells can be labeled with the sites contained by those cells. For the purposes of labeling the the order of the sites is arbitrary; here $(3, 2)$ is just as good as $(2, 3)$ while $(2, 7, 3)$, $(3, 7, 2)$, $(3, 2, 7)$, $(7, 3, 2)$, and $(7, 2, 3)$ are just as good as $(2, 3, 7)$.

2 The Geometry of Voronoi Diagrams

The boundary between the cells of a Voronoi diagram is useful in its own right. In particular it defines an undirected graph, with linear edges separating neighboring pairs of cells and punctual vertices separating neighboring triplets of cells.

In practice it is often easier to construct this **Voronoi graph** and then derive the Voronoi cells from it rather than trying to construct the Voronoi cells directly. The vertices and edges of the Voronoi graph exhibit rich geometric structure that provides the foundation for powerful construction algorithms.

2.1 Voronoi Vertices

The vertices of the Voronoi graph are defined by points that are the same distance from three neighboring Voronoi cells (Figure 3). Consequently we can uniquely label each vertex with a triplet the sites contained by those cells,

$$(s_{n_1}, s_{n_2}, s_{n_3}).$$

Because not every triplet of Voronoi cells are neighbors, however, not every triplet of sites defines a valid vertex. Fortunately Euclidean geometry provides tools for distinguishing the valid vertices.

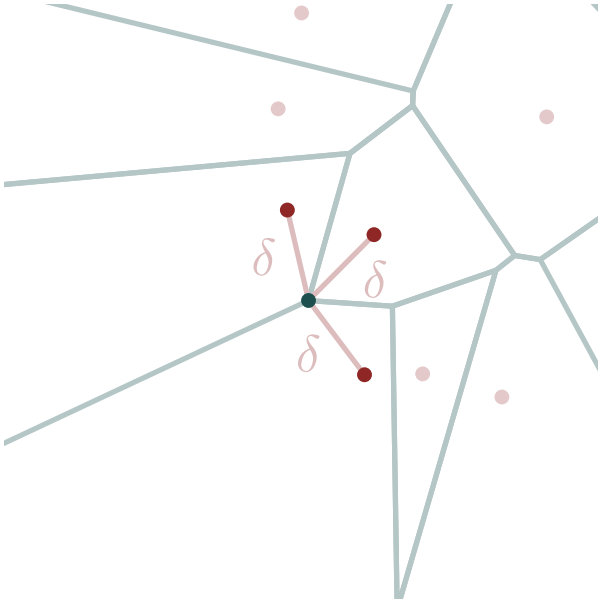


Figure 3: Each vertex in a Voronoi graph is given by a point on the boundary between three neighboring Voronoi cells. This point is by definition equally distant to the sites contained by those cells.

Any triplet of distinct points falls onto a unique **circumcircle** with center

$$x_c = \frac{1}{2}\frac{\left(x_{n_1}^2 + y_{n_1}^2\right)(y_{n_2} - y_{n_3}) + \left(x_{n_2}^2 + y_{n_2}^2\right)(y_{n_3} - y_{n_1}) + \left(x_{n_3}^2 + y_{n_3}^2\right)(y_{n_1} - y_{n_2})}{x_{n_1}(y_{n_2} - y_{n_3}) + x_{n_2}(y_{n_3} - y_{n_1}) + x_{n_3}(y_{n_1} - y_{n_2})}$$

$$y_c = \frac{1}{2}\frac{\left(x_{n_1}^2 + y_{n_1}^2\right)(x_{n_3} - x_{n_2}) + \left(x_{n_2}^2 + y_{n_2}^2\right)(x_{n_1} - x_{n_3}) + \left(x_{n_3}^2 + y_{n_3}^2\right)(x_{n_2} - x_{n_1})}{x_{n_1}(y_{n_2} - y_{n_3}) + x_{n_2}(y_{n_3} - y_{n_1}) + x_{n_3}(y_{n_1} - y_{n_2})}$$

and radius

$$\begin{aligned} r &= \sqrt{(x_c - x_{n_1})^2 + (y_c - y_{n_1})^2} \\ &= \sqrt{(x_c - x_{n_2})^2 + (y_c - y_{n_2})^2} \\ &= \sqrt{(x_c - x_{n_3})^2 + (y_c - y_{n_3})^2}. \end{aligned}$$

Consequently the center of the circumcircle defined by three sites is equidistant to those sites, and hence defines a potential vertex in the Voronoi graph.

The three sites defining a circumcircle are the three *closest* sites to this potential vertex if and only if no other sites are inside of the circumcircle. In other words Voronoi vertices are defined by *empty* circumcircles.

(a) (b)

Figure 4: Not every triplet of sites defines a vertex in a Voronoi graph. (a) Vertices are defined by triplets of sites that define empty circumcircles. (b) Triplets of sites whose circumcircles contain other sites do not define proper vertices.

2.2 Voronoi Edges

The edges in a Voronoi graph are comprised of points equidistant to two neighboring cells (Figure 5), allowing us to uniquely label each Voronoi edge with the pair of sites contained by those cells. Once, again, however, we have to take care because arbitrary pairs of sites do not always define valid edges.

Before worrying out isolating proper edges let's first work out the geometry of potential edges. Any pair of sites defines a line that connects them and a mid point on that line that is equally distant to the sites. The **perpendicular bisector** of two sites is the line perpendicular to this connecting line that intersects with the midpoint (Figure 6a). Every point along the perpendicular bisector between two sites is equally distant from those two sites and, consequently, could contribute to a Voronoi edge (Figure 6b).

In order to determine which segment, if any, of a perpendicular bisector forms a Voronoi edge we come back to circles. Any point along a perpendicular bisector is the center of a unique circle that includes the two defining sites. If no *other* sites are on or inside of this circle then those two sites are the nearest neighbors to each other and that point is, by definition, part of a Voronoi edge.

As we move along the perpendicular bisector these circles will eventually intersect with another site. When they do the point along the bisector will be equally distant to three sites – the two initial sites and this new site – but the interior of the circle will remain empty. In other words the Voronoi edge terminates as soon as we hit a Voronoi vertex.

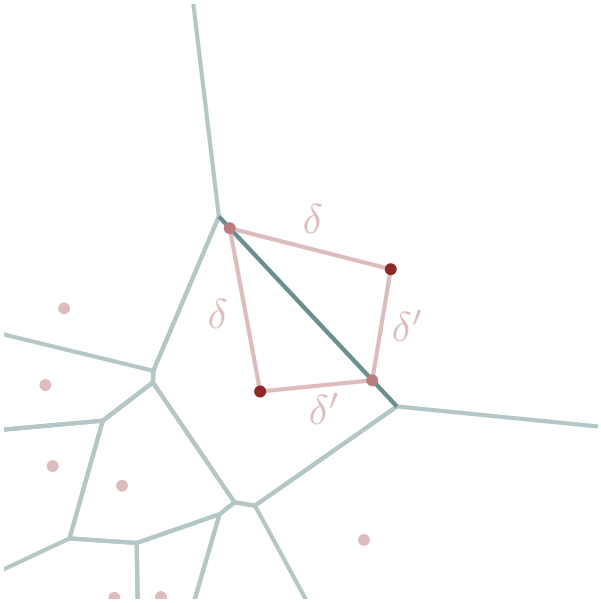
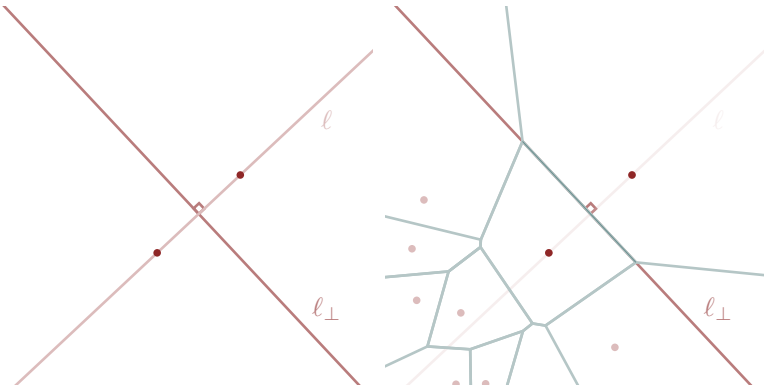
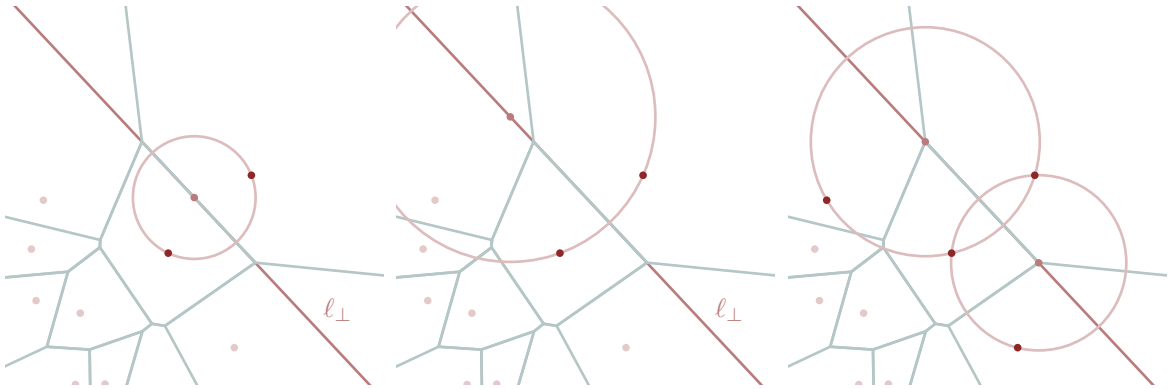


Figure 5: Each edge in a Voronoi graph is defined by linear boundary between two neighboring Voronoi cells. All of the points on a Voronoi edge are equally distant to the two sites contained by those cells.



(a) (b)

Figure 6: (a) The perpendicular bisector between two points is the line $\ell_\perp$ that is perpendicular to line connecting those points, ℓ, and intersects the midpoint between them. (b) Every Voronoi edge falls along a perpendicular bisector between two sites, but generally is only a segment of that line. Moreover not every perpendicular bisector contains a Voronoi edge.



(a) (b) (c)

Figure 7: Circles centered at points along a perpendicular bisector and intersecting with a pair of sites are useful for identifying contributions to Voronoi edges. (a) If there are no other sites on or inside of the circle then the center is part of a Voronoi edge. (b) If there is one or more sites inside of the circle then the center does not contribute to a Voronoi graph at all. (c) If there is another site on the circle then the center defines a Voronoi vertex. Each Voronoi edge terminates when these circles first meet a third site.

2.3 Voronoi Faces

The edges of a Voronoi graph trace around each site, defining faceted **faces** in the graph that correspond to the Voronoi cells (Figure 8). When working over the entire, unbounded, Euclidean plane not all of these faces will be closed. While some of the sites will be entirely enclosed by a path of edges, some will be bounded on only one side by a path of edges. I will refer to these as closed faces and open faces, respectively.

(a) (b)

Figure 8: The edges of a Voronoi graph trace out faces which identify the Voronoi cells. (a) I will refer to faces that are completely enclosed by edges as closed faces and (b) faces that are only partially enclosed as open faces.

3 Fortune's Algorithm

There are here are multiple ways to construct a Voronoi graph, but most of them suffer from wasteful computation. For large enough numbers of sites these approaches quickly become too expensive to be practical. Fortunately there is one elegant approach that, while not at all initially obvious, is able to construct a Voronoi graph as efficiently as possible.

3.1 Inefficient Approaches

Before diving into an efficient solution let's consider some of the more straightforward but ultimately inefficient approaches.

One way to identify all of the vertices in a Voronoi graph is to loop over each triplet of sites and then construct the corresponding circumcircles. For each of these circumcircles we then loop over the remaining sites to check for inclusion, with the centers of the empty circumcircles defining vertices.

We could then construct candidate Voronoi edges by looping over each pair of sites and checking to see if they are associated with any vertices. For each valid pair we could scan across the corresponding perpendicular bisector, construct circles, and check if any other sites appear on or within them.

Because there are

$$\binom{N}{3} = \frac{N\,(N-1)\,(N-2)}{6}$$

triplets of sites and

$$\binom{N}{2} = \frac{N\,(N-1)}{2}$$

pairs of sites to consider, however, the number of operations we need to implement this brute-force approach scales *cubically* with the number of sites. Adding twice as many sites requires six times more operations, and hence six times the cost.

That said, much of this cost is wasted on the redundant queries of each site. We should be able to construct Voronoi graphs more efficiently if we can query each site fewer times. One more efficient approach is to loop over each site, construct the $(N-1)$ half-planes bounded by the perpendicular bisector between the active site and each other site, and then construct the corresponding Voronoi face from the intersection of those half-planes. This cost of this approach scales only quadratically.

We can, however, do even better. By systematically scanning across a plane **Fortune's algorithm** Berg et al. (2008) is able construct a Voronoi graph with a number of operations proportional to $N \log N$. This performance turns out to be theoretically optimal. The main downside of Fortune's algorithm is that is requires some care to understand and then implement in a way that achieves that optimal scaling.

3.2 The Sweep Line

Fortune's algorithm processes the sites not by working through a list but rather by geometrically scanning across the ambient plane with the use of a **sweep line**. Sites on one side of the sweep line are **active** and can contribute to the construction of the Voronoi graph while sites on the other side are **inactive** and cannot yet contribute. As the sweep line progresses more sites are processed and more of the Voronoi graph is built up.

In theory a sweep line can progress in any direction, but most references consider vertical lines that scan horizontally from left to right, horizontal lines that scan vertically from top to bottom, or horizontal lines that scan vertically from bottom to top. These axis-aligned orientations of the sweep line make geometric calculations substantially more straightforward.

Here I will consider a vertical sweep line that scans horizontally from left to right and denote the horizontal position of the sweep line at any time by S.

3.3 The Beach Line

A sweep line partitions the ambient plane into two subsets, one containing all of the active sites and one containing all of the inactive sites. This allows us to focus the construction of the Voronoi graph on just one subset of the entire plane and ignore the other. That said, not all of the Voronoi graph to the left of the sweep line is completely determined by the active sites; some features of the Voronoi diagram can be influenced by inactive sites just beyond the sweep line.

Consider, for example, a single active site

$$s_n = (x_n, y_n)$$

and the sweep line position

$$S > x_n.$$

If a point (x, y) to the left of the sweep line is closer to s_n than the sweep line,

$$(x - x_n)^2 - (y - y_n)^2 < (S - x)^2,$$

then we it will be closer to s_n than any other site that might be hiding beyond the sweep line. On the other hand any point to the left of the sweep line that is closer to the sweep line than it is to s_n,

$$(x - x_n)^2 - (y - y_n)^2 > (S - x)^2,$$

could be be closer to an inactive site on the other side of the sweep line that we have not yet taken into account.

Consequently when determining site proximity we cannot analyze the entire region to the left of the sweep line. Instead we can analyze only the subset of points that are closer to s_n than

they are to the sweep line. The boundary of this analyzable subset is defined by the points equidistant to s_n and the sweep line, which traces out the rotated parabola (Figure 9)

$$x = f(y) = \frac{1}{2}\left(S + x_n - \frac{(y - y_n)^2}{S - x_n}\right).$$

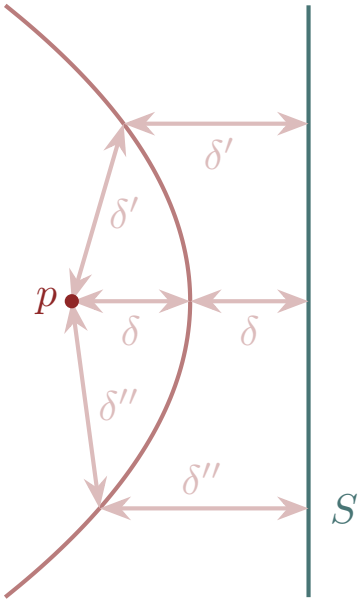


Figure 9: The subset of points that are equally distant from a focus, s, and a vertical line S, form a rotated parabola. In Fortune's algorithm the points to the left of this parabola are safe to analyze for their contribution to the Voronoi graph.

When there are multiple active sites then the region that we can safely analyze is comprised of the points that are closer to *any* of the active sites than they are to the sweep line. The boundary of this region is defined by the segmented envelope of the parabolas between each active site and the sweep line (Figure 10). Given its resemblance to the shape made by waves washing up on a beach (Figure 11) this boundary is known as the **beach line**.

Each parabolic segment, or **arc**, in a beach line is associated with a single site. The parabola between any given site and the sweep line, however, can contribute to the beach line multiple times. Consequently we cannot use the originating sites to *uniquely* label the arcs.

Neighboring arcs on the beach line are generated from neighboring sites. The discontinuities along the beach line that separate each pair of neighboring arcs are known as **breakpoints** (Figure 12). Each breakpoint is uniquely labeled by an ordered pair of the sites associated with those arcs. Note that ordering is important here: (s_{n_1}, s_{n_2}) and (s_{n_2}, s_{n_1}) correspond to two *different* breakpoints.

Given a sweep line position S we can readily calculate the intersection of each pair of neighboring parabolas, and hence the position of the corresponding breakpoint. The vertical position y

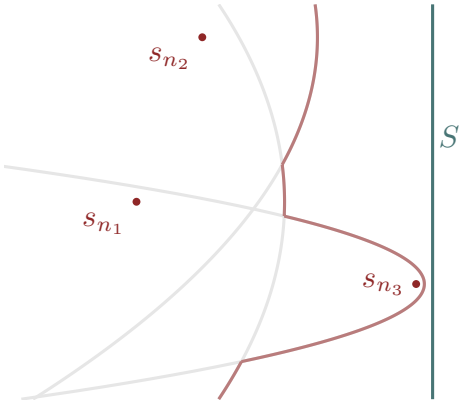


Figure 10: The beach line is the boundary separating points that are closer to any active site, here s_{n_1}, s_{n_2}, and s_{n_3}, and points that are closer to the the sweep line, S. This boundary is comprised of a sequence of intersecting parabolic arcs. Note that the parabola between s_{n_1} and S contributes multiple arcs to the beach line here.

Figure 11: The parabolic arcs that make up a beach line are similar to the shape made by waves washing up onto a beach, hence the name. Photo :copyright: Michael Betancourt.

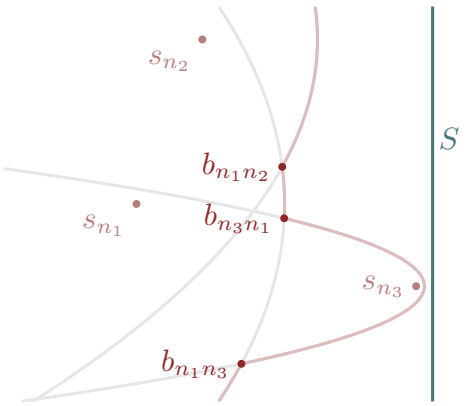


Figure 12: The breakpoints in a beach line separate the component arcs. Each breakpoint is located at the intersection between the parabolas generated by two neighboring sites, equally distant from the sweep line and those two sites while also being further from any other sites. Each ordered pair of neighboring sites defines a unique breakpoint. Here, for example, $b_{n_1 n_2}$ and $b_{n_2 n_1}$ are distinct breakpoints.

where the two parabolas generated by s_{n_1} and s_{n_2} intersect is implicitly given by the quadratic equation

$$\begin{aligned} 0 = &\left(\underbrace{x_{n_2} - x_{n_1}}_{a} \right) y^2 \\ &- 2 \left(\underbrace{y_{n_2} \left(S - x_{n_1}\right) - y_{n_1} \left(S - x_{n_2}\right)}_{b} \right) y \\ &+ \left(\underbrace{y_{n_2}^2 \left(S - x_{n_1}\right) - y_{n_1}^2 \left(S - x_{n_2}\right) - \left(S - x_{n_1}\right) \left(S - x_{n_2}\right) \left(x_{n_2} - x_{n_1}\right)}_{c} \right), \end{aligned}$$

which admits two solutions,

$$y_{\pm} = b \pm \frac{1}{2\,a} \sqrt{b^2 - a\,c}.$$

To determine which of these solutions is associated with the breakpoint $b_{n_1 n_2}$, and hence which is associated with $b_{n_2 n_1}$, we'll need to consider the relative position of the sites.

If s_{n_1} is ahead of s_{n_2},

$$x_{n_1} > x_{n_2}$$

then the breakpoint $b_{n_1 n_2}$ will be *above* the breakpoint $b_{n_2 n_1}$ (Figure 13a). In this case

$$a = x_{n_2} - x_{n_1}$$

is negative so the higher solution is given by

$$y_{n_1 n_2} = y_- = b - \frac{1}{2\,a}\sqrt{b^2 - a\,c}.$$

On the other hand if s_{n_1} is behind s_{n_2},

$$x_{n_1} < x_{n_2}$$

then the breakpoint $b_{n_1 n_2}$ will be *below* the breakpoint $b_{n_2 n_1}$ (Figure 13b). Because the leading coefficient is now positive,

$$a = x_{n_2} - x_{n_1},$$

the appropriate solution is *also* given by

$$y_{n_1 n_2} = y_- = b - \frac{1}{2\,a}\sqrt{b^2 - a\,c}.$$

In other words the the sign of a conveniently adjusts the geometry correctly so that we can *always* take

$$y_{n_1 n_2} = b - \frac{1}{2\,a}\sqrt{b^2 - a\,c}$$

and

$$y_{n_2 n_1} = b + \frac{1}{2\,a}\sqrt{b^2 - a\,c}.$$

Figure 13: The parabolas defined by two points s_{n_1} and s_{n_2} intersect at two different points, one of which gives the breakpoint $b_{n_1 n_2}$ and the other giving $b_{n_2 n_1}$. (a) If $x_{n_1} > x_{n_2}$ then $b_{n_1 n_2}$ is given by the higher of the two intersections, but (b) if $x_{n_1} < x_{n_2}$ then $b_{n_1 n_2}$ is given by the lower of the two intersections.

When it comes to constructing Voronoi graphs the breakpoints are the most important part of the beach line. Because they always falls onto the parabolas generated by two neighboring

sites, each breakpoint is equally distant from those two sites and the sweep line. At the same time each breakpoint is further from all other active sites as well as any inactive sights hiding beyond the sweep line.

All of this is to say that, by construction, every breakpoint is the center of a circle containing the two associated sites but includes no other sites. In other words, each breakpoint *always falls on a Voronoi edge no matter the position of the sweep line.* Specifically the breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_1}$ both fall on the Voronoi edge between s_{n_1} and s_{n_2}.

3.4 Evolving The Beach Line

As we scan the sweep line across a Euclidean plane the distances between the active sites and the sweep line will change. This, in turn, changes the shape of the corresponding parabolas and, ultimately, the shape of the beach line. As the beach line evolves the breakpoints trace out the Voronoi edges, incrementally generating the entire Voronoi graph!

Every now and then the number of arcs comprising the beach line, known as the **topology** of the beach line, also changes. When the sweep line passes over a new site, for example, a new arc is added to the beach line. Similarly arcs can be removed from the beach line when they are squeezed to a point by their neighboring arcs.

Conveniently we don't need to *continuously* scan the sweep line in order to construct a Voronoi diagram. The structure of a Voronoi diagram is completely determined by the behavior of the beach line at the finite number of sweep line positions where the topology changes. If we can efficiently determine these topology-changing **events** then we can efficiently build up a Voronoi diagram in a finite number of operations.

3.4.1 Events Where The Beach Line Expands

Whenever the sweep line crosses a new site, that site becomes active and contributes a new parabolic arc to the beach line. Because we know the position of all of the sites we know at exactly which positions these **site events** will occur.

If the sweep line is positioned on a new site then the parabola of points equidistant from that site and the sweep line is singular, consisting of just the new site itself. This forms a point discontinuity with the rest of the beach line (Figure 14a). Although not *technically* correct, these singular arcs are often visualized with a horizontal line extending from the new site to the beach line (Figure 14b).

As the sweep line progresses past the new active site, the singular parabola widens into a more well-behaved parabola (Figure 14c). This not only restores the continuity of the beach line but also splits an existing parabolic arc into two new arcs. The introduction of a new active site always splits an existing arc into a triplet of arcs, the outer two of which are generated by the same site.

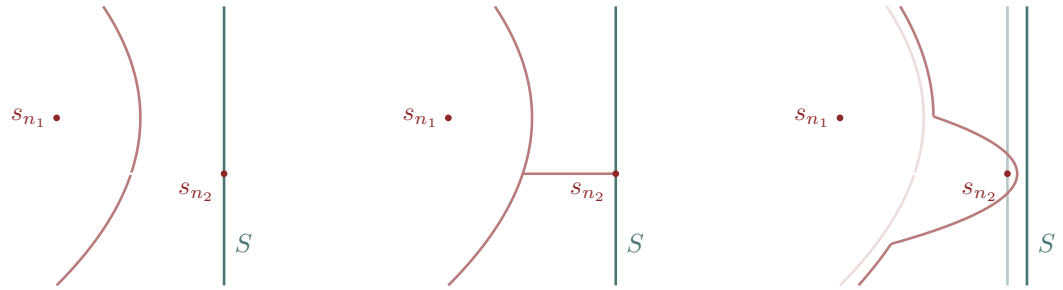


Figure 14: At a site event the sweep line hits an inactive site which then becomes active. (a) Initially the new active site introduces a new singular arc which splits the beach line. (b) This singular arc is often visualized with a continuous horizontal line. (c) As the sweep line proceeds past the site event, this singular arc widens into a well-behaved parabola, separating one of the beach line arcs into two arcs.

For example let's say that the sweep line passes the new site s_{n_2}. If the height of this site y_{n_2} intersects with an arc generated by the site s_{n_1} then the site event will split that arc into a triplet of arcs, the outer two of which are generated by s_{n_1} and the center of which is generated by s_{n_2}. These three arcs are then separated by two new breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_1}$.

Both of these new breakpoints fall on a Voronoi edge in between the sites s_{n_1} and s_{n_2}. As the sweep line progresses the middle arc widens and the breakpoints separate, tracing out more and more of that edge (Figure 15).

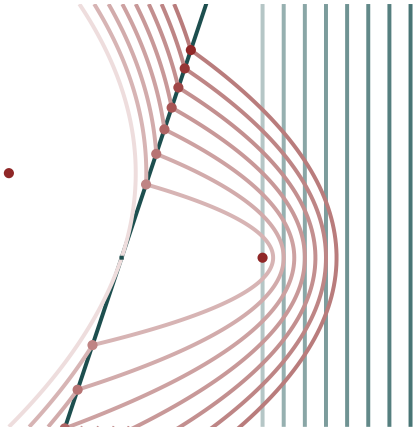


Figure 15: As the sweep line moves past a site event, the two new breakpoints separate and scan across the Voronoi edge between the two neighboring sites.

3.4.2 Events Where The Beach Line Contracts

The topology of the beach line can also change when one of the parabolic arcs collapses in between its two neighboring arcs (Figure 16). In this case the collapsed arc is removed from the beach line and its two neighboring arcs become new neighbors.

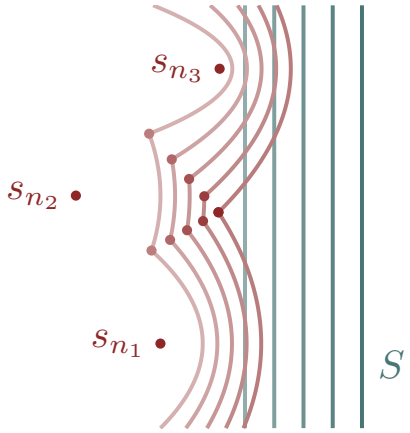


Figure 16: As the sweep line progresses, arcs on the beach line can collapse in between their two neighboring arcs. When this happens the neighboring breakpoints converge to a point. This convergence point is equally distant from the three active sites but further from all other sites, defining a vertex in the Voronoi graph.

Consider three neighboring arcs generated from the active sites s_{n_2}, s_{n_2}, and s_{n_3}. If any two of these sites are the same then the central arc will not collapse as the sweep line progresses so we can safely assume that these sites are distinct.

When the central arc collapses its two neighboring breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_3}$ will merge into a new breakpoint $b_{n_1 n_3}$. At that moment

$$b_{n_1 n_2} = b_{n_2 n_3} = b_{n_1 n_3}$$

are equally distant from s_{n_2}, s_{n_2}, and s_{n_3}. Because this intersection is on the beach line it also cannot be closer to any inactive sites hiding beyond the sweep line. In other words a collapsing arc on the beach line always defines a Voronoi vertex!

This duality between arc collapses and Voronoi vertices is useful not only for building up the Voronoi graph but also for predicting when arcs will collapse. By now it should be no surprise that this prediction will involve circles.

For any triplet of neighboring arcs that correspond to the distinct sites s_{n_1}, s_{n_2}, and s_{n_3} we can always construct a unique circumcircle with center (x_c, y_c) and radius r. By construction the center (x_c, y_c) is equally distant to all three sites, and hence identifies exactly where the central arc will collapse to a point.

The center is also equally distant to vertical lines at $x = x_c - r$ and $x = x_c + r$, which define the two possible sweep line configurations where the central arc will collapse. When $S = x_c - r$, however, the sweep line will not yet have reached any of the three sites and they will not yet have contributed any arcs to the beach line. Consequently a central arc generated by s_{n_2} can collapse only when $S = x_c + r$ (Figure 17).

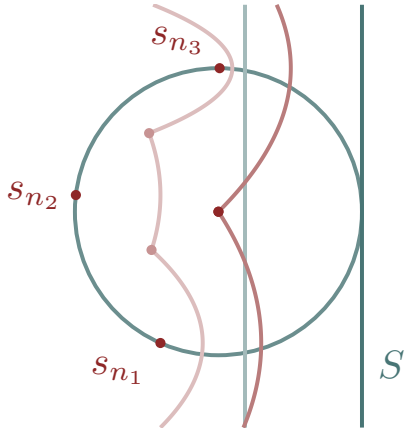


Figure 17: If the central arc in a triplet of neighboring arcs collapses then it will do so when the sweep line is at the far edge of the circumcircle spanned by those three sites. This geometric extrapolation allows us to schedule future vertex events.

That said, this collapse can contribute to the Voronoi graph only if it occurs to the right of the beach line, where the structure of the Voronoi graph has not yet been established. If x_c falls to the left of the current beach line then advancing the sweep line forward will cause the central arc to not collapse but rather expand, with the breakpoints *diverging* away from each other.

One particularly straightforward way to determine if a central arc will collapse as the sweep line progresses forward is to directly check if $y_{n_1 n_2} = y_{n_2 n_3}$ at $S = x_c + r$. While this works fine in theory, it can be awkward to implement in practice as checking the equality of two floating point numbers is always tricky.

We can also use the relative positions of s_{n_1}, s_{n_2}, and s_{n_3} to determine whether some arcs will converge or diverge. For example if site s_{n_2} is *behind* both s_{n_1} and s_{n_3},

$$x_{n_2} < x_{n_1}$$
$$x_{n_2} < x_{n_3},$$

then the Voronoi edges, and the breakpoints that flow along them, will converge (Figure 18a). On the other hand if s_{n_2} is *ahead* of the other two sites,

$$x_{n_2} > x_{n_1}$$
$$x_{n_2} > x_{n_3},$$

then the Voronoi edges will diverge, pulling the breakpoints away from each other (Figure 18b).

The mixed cases where

$$x_{n_1} < x_{n_2} < x_{n_3}$$

or

$$x_{n_1} > x_{n_2} > x_{n_3}$$

are indeterminate (Figure 18c, Figure 18d). Depending upon the vertical positioning of the sites the central arc can collapse or expand as the sweep line progresses. The behavior in these cases can probably be worked out with some clever geometry, but I have yet not been able to figure out that geometry myself. Consequently I will use the more direct approach mentioned above.

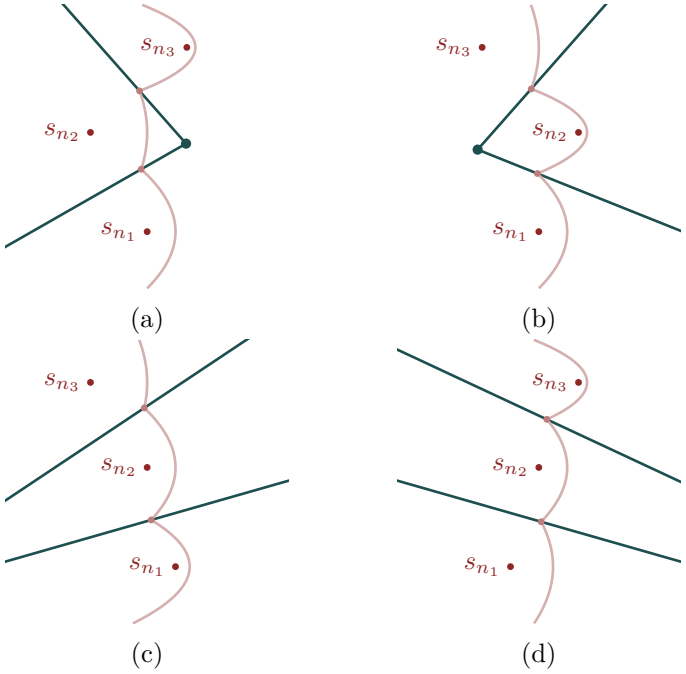


Figure 18: The central arc of any triplet of neighboring beach line arcs, here generated by the sites s_{n_1}, s_{n_2}, and s_{n_3}, will not always collapse as the sweep line progresses forward. (a) If s_{n_2} is behind the other two sites then the arc it generates will eventually collapse, but (b) if s_{n_2} is ahead of the other two sites then the arc it generates will only ever expand. (c, d) When s_{n_2} is in between the two other sites we need additional information to determine if the corresponding arc will collapse.

If a central arc will eventually collapse as the sweep line progresses then that collapse defines **vertex event**, also known as a **circle event**, at $S = x_c + r$. At this sweep line position the breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_3}$ will intersect at (x_c, y_c), defining a new vertex that we can add

to the Voronoi graph (Figure 19). This vertex then spawns a new Voronoi edge along which the new merged breakpoint $b_{n_1 n_3}$ will flow.

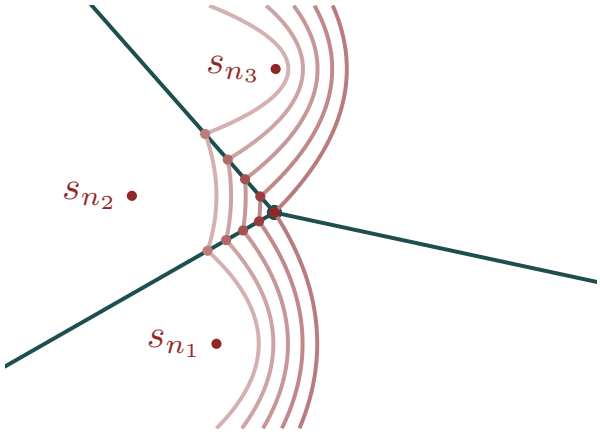


Figure 19: When a central arc collapses, two Voronoi edges terminate at a Voronoi vertex and a new Voronoi edge between the sites that generate the remaining arcs is created.

3.5 The Algorithm

Now that we know how and when the topology of a beach line can change we can organize those changes into an algorithm that builds up a Voronoi graph dynamically.

Fortune's algorithm starts with an **event queue** consisting of events defined by sweep line positions where an arc is added to or removed from the beach line. The algorithm then proceeds iteratively, processing each event in the queue one at a time, expanding the structure of the Voronoi graph, and adding new events as necessary. Initially the event queue is filled with only site events, one for each site, but vertex events are also added as the site events are processed.

In this section we'll review the basic steps required to process both site and vertex events, as well as clean up after the event queue has been fully processed. Section 4 will focus on the implementation of these steps, which becomes a good bit more complicated if we want to ensure efficient computation.

3.5.1 Site Events

Processing a site event requires adding a new arc to the beach line. To do this we have to scan across the current beach line to find the existing arc that meets the height of the new event. We then split that arc in two, inserting a new arc generated by the new site into the beach line along with two new, neighboring breakpoints.

Inserting a new arc into the beach line can also modify the remaining event queue.

For one the new arc will disrupt any triplets of neighboring arcs that contain the split arc. If any of those disrupted arc triplets correspond to future vertex events then they have to be removed from the event queue.

At the same time inserting a new arc also creates two triplets of neighboring arcs, each of which could potentially spawn a vertex event. If the central arc of any of these triplets eventually collapses then we need to add a new vertex event to the event queue where the central arc will be removed.

To make this all a bit more concrete consider an initial beach line comprised of three successive arcs, α_1,

$$(\alpha_1, \alpha_2, \alpha_4, \alpha_2, \alpha_3).$$

When the sweep line then passes a new site it introduces a new arc α_4. Let's say that this arc splitsα_2, resulting in the new beach line

$$(\alpha_1, \alpha_2, \alpha_4, \alpha_2, \alpha_3).$$

This site event breaks the initial triple $(\alpha_1, \alpha_2, \alpha_3)$ and invalidates any associated vertex events. It also, however, introduces two new triples, $(\alpha_1, \alpha_2, \alpha_4)$ and $(\alpha_4, \alpha_2, \alpha_3)$, which have to be considered for new vertex events.

3.5.2 Vertex Events

Our first task when we encounter a vertex event in the event queue is to expand the current Voronoi graph. This starts by adding a new Voronoi vertex, connecting it to two of the previously existing Voronoi edges, and then creating a new Voronoi edge. Exactly how we accomplish this expansions depends on how the graph itself is being implemented.

Once the Voronoi graph has been updated we then remove the central arc from the beach line and update the breakpoints appropriately.

Lastly we have to clean up the event queue. The removal of the central arc will invalidate any vertex event in the event queue where the central arc was previously neighbors with the removed arc. On the other hand the removal also creates two new triplets of neighboring arcs along the beach line, each of which could spawn a new vertex event. If any of these triplets feature a collapsing central arc then we have to add a corresponding vertex event to the event queue.

Again to make this a bit more concrete let's consider an initial beach line comprised of the five successive arcs,

$$(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5).$$

At a vertex event the central arc α_3 collapses, resulting in the new beach line

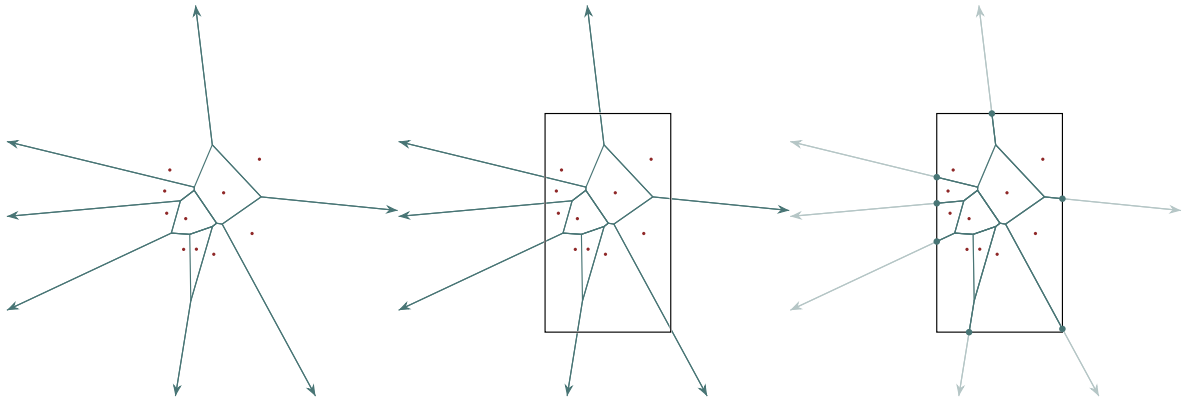
$$(\alpha_1, \alpha_2, \alpha_4, \alpha_5).$$

.

This vertex event breaks two initial triples, $(\alpha_1, \alpha_2, \alpha_3)$ and $(\alpha_3, \alpha_4, \alpha_5)$, invalidating any associated vertex events still in the event queue. At the same time the deletion of α_3 introduces two new triples, $(\alpha_1, \alpha_2, \alpha_4)$ and $(\alpha_2, \alpha_4, \alpha_5)$, each of which could define new vertex events.

3.5.3 Clean Up

As we process events the queue will eventually empty. When there are no more events to process the algorithm will have found all of the Voronoi vertices and all of the Voronoi edges. That said some of those edges will be dangling, anchored to a vertex on only one side because they stretch out to infinity in the other direction (Figure 20a).



(a) (b) (c)

Figure 20: (a) After running Fortune's algorithm some of the edges in the Voronoi graph will be connected to vertices on only one side. The other side extends out towards infinity. (b) When visualizing a Voronoi graph in practice we can consider only a finite range of values. The best we can do is visualize the graph within a finite bounding box that contains all of the Voronoi vertices. (c) For visualization purposes we can anchor any dangling edges to points where they meet the bounding box.

We can handle these dangling edges in a few different ways. For example from a graph theory perspective we could anchor any dangling edges to a single "vertex at infinity" that quantifies the common unboundedness of those edges. This approach, however, does not yield particularly compelling visualizations.

For visualization purposes it is more useful to create a rectangular **bounding box** that contains all of the Voronoi vertices (Figure 20b), and then anchor the dangling edges to points

along this rectangle (Figure 20c). Specifically the new anchor for a dangling edge will be the point where the perpendicular bisector between the two associated sites intersects with the bounding box.

4 Implementing Fortune's Algorithm

Once we've built up enough familiarity with planar geometry, the motivation for Fortune's algorithm, particularly the utility of a beach line and its evolution, becomes less obscure. That said, there is still work to do to ensure that we can implement the algorithm in a way that achieves its maximum possible performance.

For example when processing a site event we need to be able to efficiently search through the arcs on the current beach line. If we naively scan through the *entire* beach line every time we process a new site event then the the number of operations needed to implement Fortune's algorithm with scale with not $N \log N$ but rather N^2. To achieve that $N \log N$ scaling we need to implement the beach line in a way that allows for efficient searching.

Similarly we need to keep track of the next event while we add and delete events from the event queue. We could accomplish this by sorting the entire event queue after every update, but this is wasteful and also spoils the optimal $N \log N$ scaling.

Fortunately (pun very much intended) if we leverage some standard data structures from computer science then we can avoid these potential slowdowns and implement Fortune's algorithm as efficiently as possible.

4.1 Implementing Voronoi Graphs

In theory graphs can be implemented with a just list of points for each vertex and list of pairs of points for each edge. While these lists allow us to draw a graph easily enough, they don't provide enough information to perform basic graph operations, such as tracing along paths of edges to derive faces.

The **doubly-connected edge list** (Berg et al. 2008) is a data structure that efficiently captures not only the components of a graph but also the relationships between those components. These relationships in turn make it straightforward to efficiently implement graph algorithms.

A doubly-connected edge list is itself built up from lists of three primitive data structures: half-edges, vertices, and faces.

4.1.1 Half-Edges

Doubly-connected edge lists don't actually represent undirected edges. Instead the data structure splits each undirected edge into two directed edges known as **half-edges** (Figure 21).

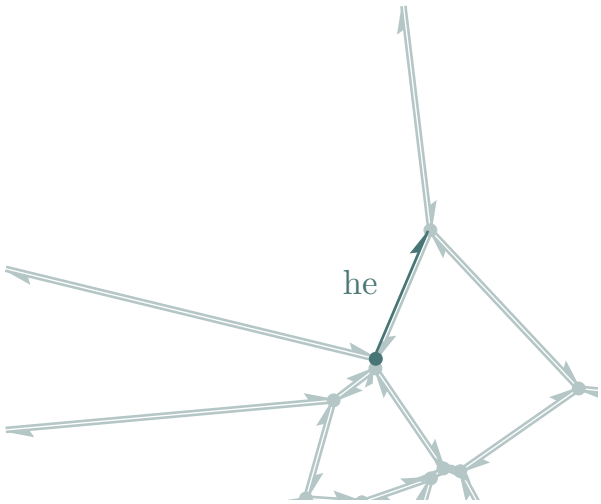


Figure 21: Doubly-connected edge lists represent undirected edges with a pair of directed half-edges. Each half-edge includes references to other structures in the doubly-connected edge lists, including the vertex from which the half-edge originates and neighboring half-edges.

To situate them within the full graph, each half-edge is endowed with a collection of references to other objects in the doubly-connected edge list. For example each half-edge includes a reference to the vertex from which the half-edge originates. Similarly each half-edge includes references to three other half-edges – the *previous* half-edge that points into it, the *next* half-edge that it points into, and the *twin* half-edge that completes it to form a full undirected edge (Figure 22). Finally each half-edge includes a reference to the face located to its left.

These references make a variety of graph operations straightforward to implement. For instance the originating vertex and twin half-edge references provide enough information to draw a half-edge as a directed line segment, starting at the position of the originating vertex and then ending at the position of the twin's originating vertex.

Similarly we can use the previous and next references to trace through paths in the graph. These paths can then be used to quantify topological information about the graph, such as the configuration of the faces.

Voronoi graphs generally feature unbounded edges, with only one side terminating at a vertex and the other shooting off towards infinity. In this case some of the half-edges will feature empty next half-edge references while their twins will feature empty originating vertex and previous half-edge references.

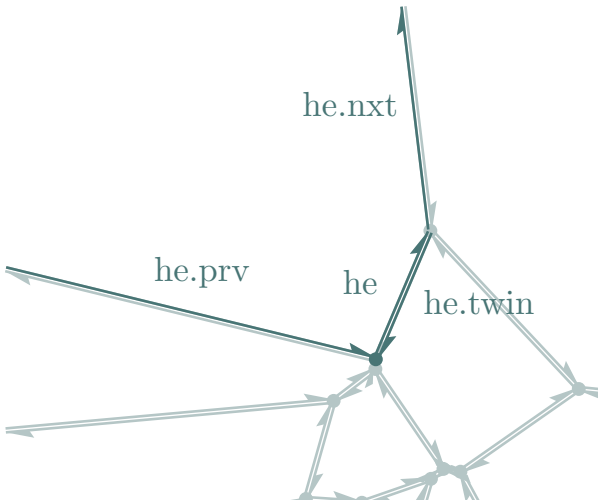


Figure 22: Each half-edge in a doubly-connected edge lists includes reference three other half-edges. The previous half-edge, here labeled "he.prv", identifies the half-edge that points into it while the next half-edge, here labeled "he.nxt", identifies the half-edge that it points into. Finally the twin half-edge, here labeled "he.twin", identifies the half-edge that completes it to form a full undirected edge.

4.1.2 Vertices

Each vertex in a doubly-connected edge list is specified with its spatial position and a reference to any one half-edge that originates from it (Figure 23). The relationship of a vertex to the rest of graph can then be completely reconstructed from this information.

For example all of the half-edges that originate at a given vertex can always be accessed by repeatedly following the twin and next references from the one included half-edge (Figure 24). Equivalently we can iteratively follow the previous and then twin references. In the same way we can also access the half-edges that *terminate* at a given vertex.

Once we can access all of the neighboring half-edges we can then access additional information about the local structure of the graph. Neighboring vertices, for instance, are given by the originating vertex references of the twin of any half-edge that originates at the initial vertex.

4.1.3 Faces

Finally each face in the graph is specified by any one half-edge along its boundary (Figure 25a). This provides enough information to trace out the entire face boundary by following the next or previous references from that one included half-edge (Figure 25b, Figure 25c).

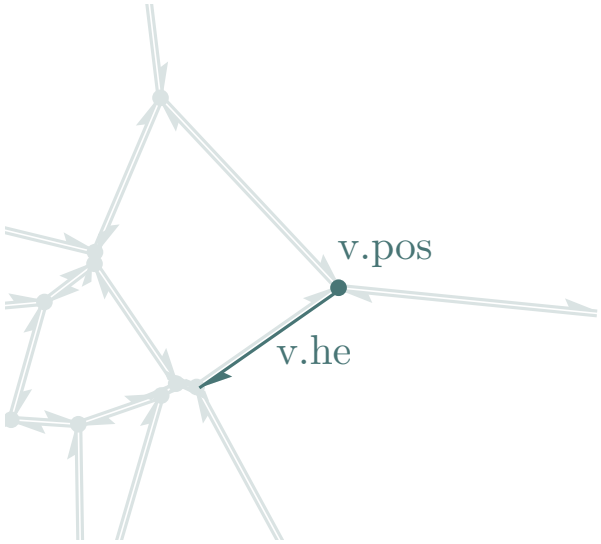


Figure 23: Each vertex in a doubly-connected edge list includes its spatial position, here denoted "v.pos" and a reference to any one half-edge that originates from it, here denoted "v.he".

When working with more complex graphs, such as graphs containing disconnected faces or faces that are completely surrounded by other faces, we need to include additional half-edge references, such as one along the exterior boundary and one along each internal boundary. Because Voronoi graphs do not exhibit these topological features, however, we won't need to worry about this additional information here.

4.1.4 Satellite Data

Depending upon the application the half-edge, vertex, and face data structures can also include additional information in the form of **satellite data**. For example when using a doubly-connected edge list to represent a Voronoi graph it will be helpful to include information about the associated sites, three for each vertex object and one for each face object. Note that we don't need to store the two sites associated with an edge; this information can be recovered from the sites of the left faces of the corresponding half-edges.

4.1.5 Bounding Box and Pseudo-Vertices

The visualization of Voronoi graphs is easier if we add a few more objects to the doubly-connected edge list data structure. In particular, a natural addition is the configuration of a bounding box as well as **pseudo-vertices** along that bounding box that can be used to terminate any dangling half-edges. This ensures, for example, that every half-edge includes

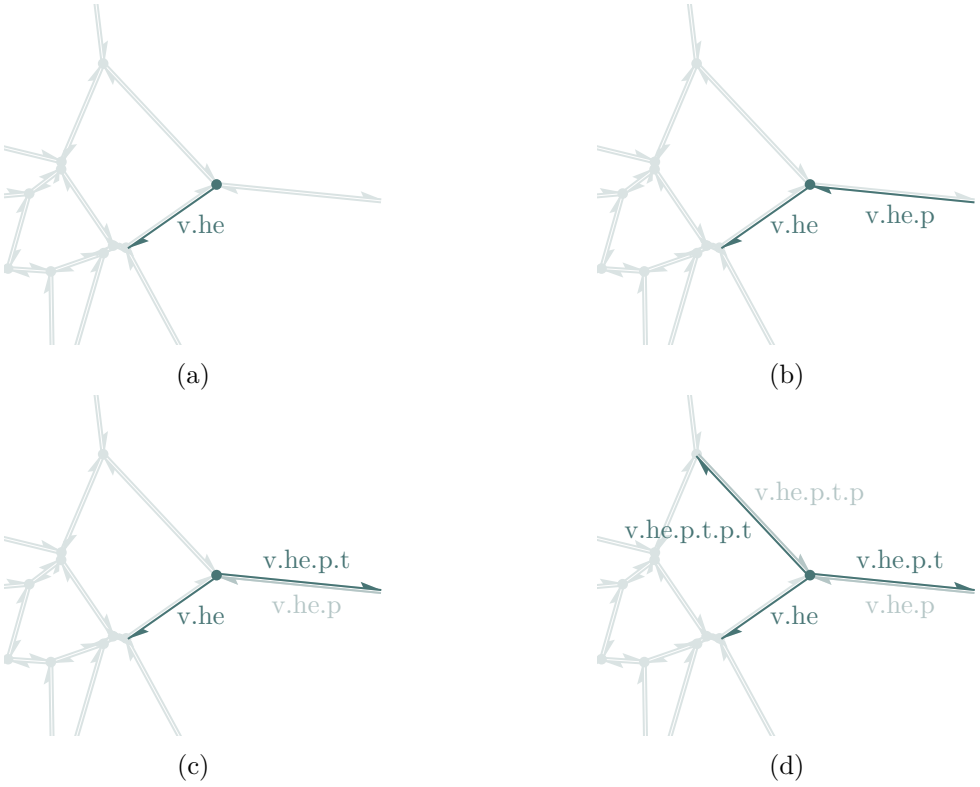
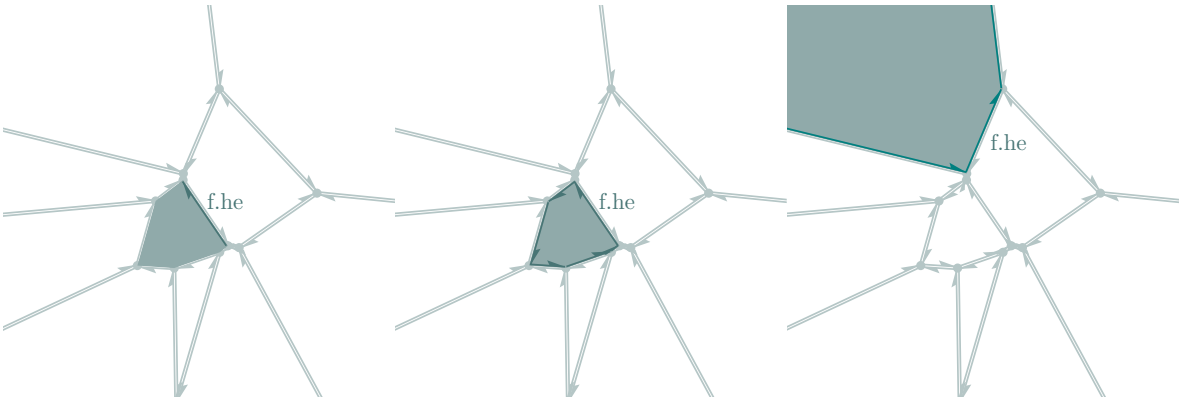


Figure 24: (a) The one half-edge referenced in a vertex object is enough to reconstruct the entire structure of the graph around that vertex. For example if we follow the reference to (b) the previous half-edge, "v.he.p", (c) and then its twin, "v.he.p.t", we recover another half-edge that originates from that vertex. (d) Repeating this operation recovers all of the half-edges that originate from that vertex.



(a) (b) (c)

Figure 25: (a) Each face in a doubly-connected edge list includes a reference to any one half-edge along its boundary. By following the next and previous references from that one half-edge we can always trace out the entire boundary of the face. (b) If the face is closed then we can follow *either* the next or previous references until we return to the initial half-edge. (c) If the face is open then we have to follow *both* the next and previous half-edges until they we encounter an empty reference.

a non-empty originating vertex references which in turns makes visualization functions more uniform.

4.2 Implementing Beach Lines

A beach line consists of parabolic arcs and the breakpoints that separate them. Critically these objects exhibit a consistent *ordering*.

As we work through Fortune's algorithm we will need to be able to efficiently search through these ordered objects. Moreover, we need to be able to add and remove arcs and breakpoints without disrupting the ordering. Conveniently all of these operations can be implemented with a slightly modified **binary search tree** (Cormen et al. 2022) that lacks an explicit key.

Many references, in particular Berg et al. (2008), recommend a binary search tree with two types of nodes: one to represent arcs and one to represent the breakpoints. Unfortunately this representation introduces some awkward implementation problems that are poorly documented. Your humble author has not yet been able to work these issues out.

In this note I will adopt a different representation inspired by, although slightly different from, the approach taken in pvigier's blog. Here beach lines will be implemented with a binary search tree where every node represents an arc and its lower breakpoint paired together (Figure 26).

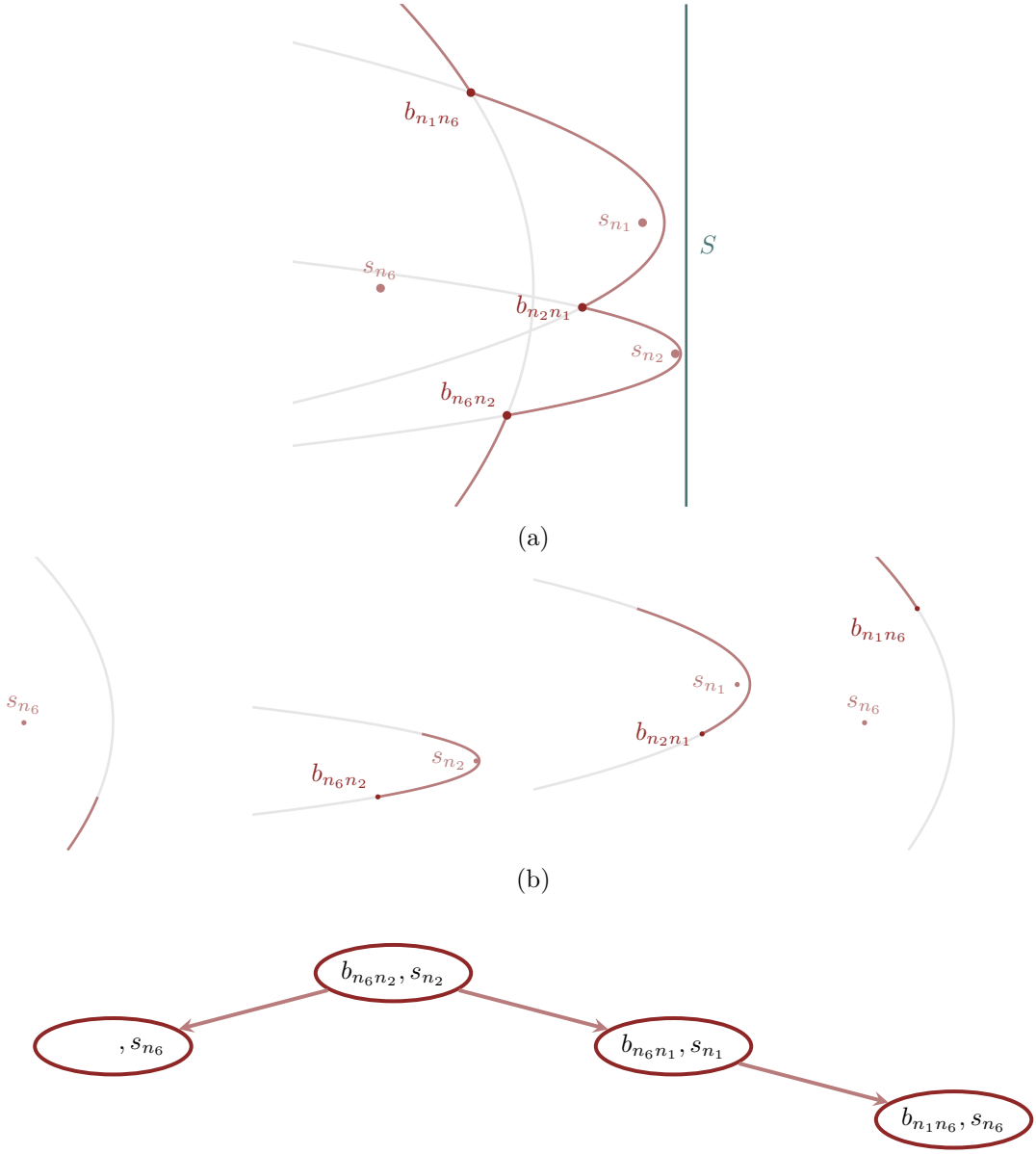


Figure 26: (a) Every beach line (b) can be decomposed into a collection of ordered arc and lower breakpoint pairs. (c) These ordered arc/breakpoint pairs can then implemented with a key-less binary search tree data structure.

Beyond a reference to their parent and child nodes in the tree, each nodes contains satellite data that allows us to reconstruct the local structure of the beach line for any sweep line configuration, and hence implement the sequential updates in Fortune's algorithm.

4.2.1 Node Data

Each node in a beach line tree includes a reference to the site that generates the corresponding arc. Because the parabola generated by a single site can contribute to the beach line multiple times, multiple nodes can share the same site reference.

Once we know the generating site we can dynamically compute the shape of the arc for any position of the sweep line. If we also know the position of the neighboring breakpoints we can then plot the full extent of the arc in the beach line.

To that end the site information is complemented with a reference to the Voronoi half-edge that originates from the lower breakpoint. From that reference we can derive the sites associated with the Voronoi edge, in particular from the left face of the half-edge and the left face of its twin. In turn those sites allow us to compute the intersection of the corresponding parabolas, and hence the position of the lower breakpoint.

Because the first arc in a beach line is always unbounded below, it does not feature a lower breakpoint. Consequently the smallest node in any beach line tree will always feature an empty half-edge reference. Every other node, however, will feature a non-trivial reference.

As the sweep line progresses the half-edges associated with each breakpoint will not yet have an originating Voronoi vertex. For visualization purposes we can always display these half-edges as if they originated from the current position of the corresponding breakpoint (Figure 27).

Finally, to avoid repeated searches through the event queue each node also includes a reference to any vertex events that are centered on the corresponding arc. This allows us to, for example, quickly find events that need to be removed from the event queue when an arc is removed from the beach line.

4.2.2 Tree Operations

The key difference (absolutely an intentional pun) between these beach line trees and generic binary search trees is that the latter don't feature a traditional key. Ordering of the nodes in a beach line tree is determined not by static key values but rather by the relative vertical position of the arcs and breakpoints along the corresponding beach line. As the sweep line progresses the absolute positions change but the relative positions, and hence the structure of the binary tree, remains invariant.

Beach line trees share most of the same operations as standard binary search trees, such as predecessor, successor, and deletion operations. Without static keys, however, beach line

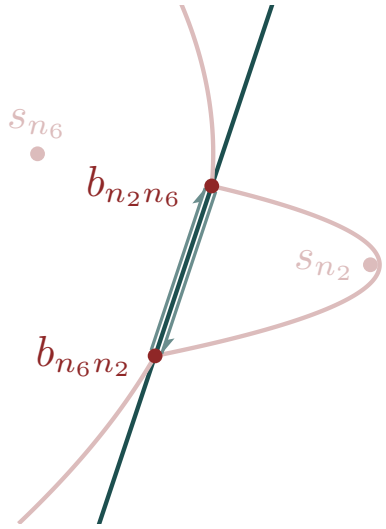


Figure 27: Initially the half-edges stored in each node are not anchored to any Voronoi vertices. When visualizing the progression of Fortune's algorithm we can instead anchor each half-edge to the current position of the corresponding breakpoint. This ensures that the half-edges always span at least a segment of full Voronoi edge, shown here in dark teal.

trees do not feature traditional key search or key insertion operations. In order to implement Fortune's algorithm we need to introduce slight different versions of these operations.

For example Fortune's algorithm requires an **arc search** where we find the arc in a beach line that intersects with a particular vertical position y when the sweep line is at position S. To find the corresponding arc node in a beach line tree we can modify the standard key search operation, comparing y to dynamically-computed breakpoint heights at each step instead of keys (Figure 28).

Once we find an appropriate node we then need to be able to insert nodes immediately before or after it, expanding the beach line with a new arc and breakpoint. Fortunately these **insert-before** and **insert-after** operations are straightforward to implement (Figure 29).

For example to insert a new node after an existing node we check to see if it has a right child. If it does not then we can insert the new node as the existing node's right child. If there is a right child then we find the successor to the existing node, which will always have no left child otherwise it would not be a proper successor. Then we can insert the new node in that empty left child position.

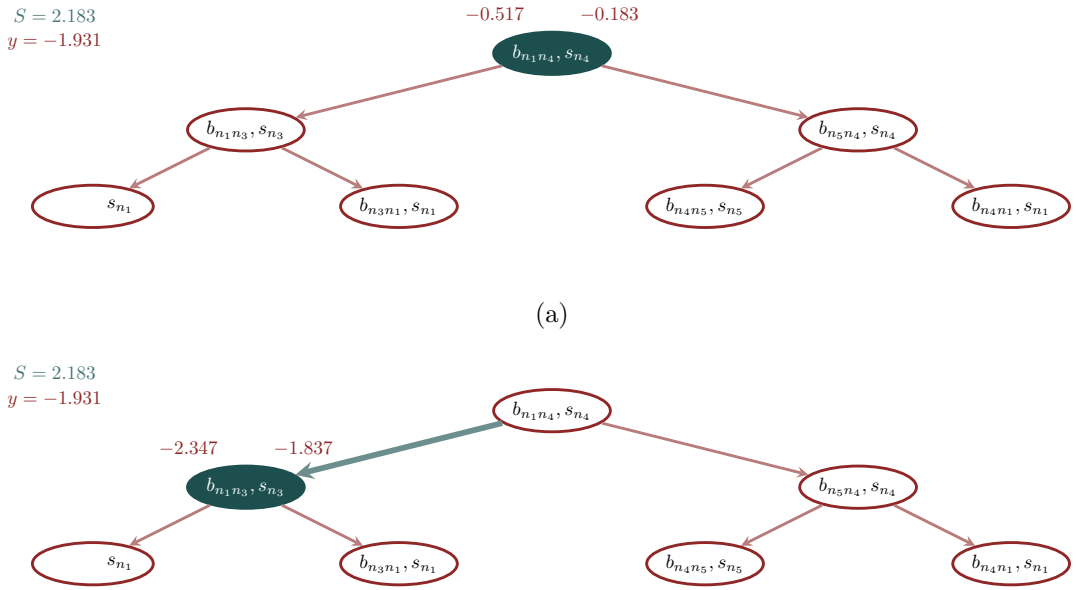
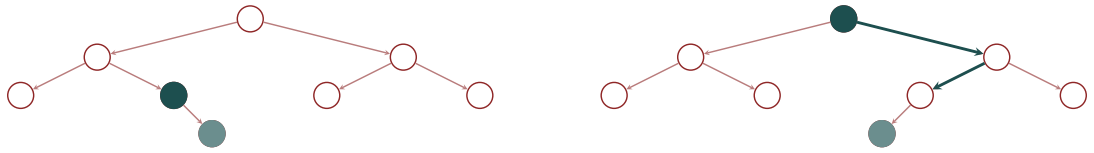


Figure 28: Unlike the nodes in standard binary search trees, the nodes in beach line trees are not equipped with static keys. Given a sweep line position, however, we can search for positional intersections. (a) To find the arc that intersects $y = -1.931$ we start at the root node. We first compute the height of the lower breakpoint for the current sweep line position, here -0.517, and then move to the left child because the breakpoint height is above y. If y were above the lower breakpoint height then we would compute the current height of the upper breakpoint and move to the right child if it is below y. Otherwise we would terminate the search. (b, c) We then repeat this process, progressing down the tree until the search terminates. Here we terminate after one iteration.



(a) (b)

Figure 29: Without static keys beach line trees do not support key insertion operations. They do, however, allow for insertion of a new node, here shown in light teal, before or after a given node, here shown in dark teal. (a) For example the insert-after operation begins by checking if the right child of the given node is empty and, if it is, then inserts the new node as a new right child. (b) If the right child is occupied then the insert-after operation moves to the successor node. Because the left child of the successor will always be empty if the initial right child is not, the operation can safely insert the new node there. The insert-below operation is symmetric, only moving backwards instead of forwards.

4.2.3 Balancing

The number of operations needed to implement all of these beach line tree operations scales with the height of a particular tree. Now the height of most trees scales logarithmically with the number of nodes, with in the context of Fortune's algorithm also scales logarithmically with the number of sites. Consequently for most trees these operations require only a logarithmic number of operations, and a *typical* evaluation of Fortune's algorithm will achieve an overall $N \log N$ cost scaling.

Unfortunately the height of *every* possible binary tree is not always logarithmic in the number of nodes. In the worst case the height of a tree can be linear in the number of nodes, pushing up the overall cost scaling of Fortune's algorithm to quadratic.

In order to realize the optimal performance of Fortune's algorithm we need to be able to avoid excessively tall, or **unbalanced** trees. Conveniently this can be accomplished with the use of **self-balancing binary trees**, in particular **red-black trees** (Cormen et al. 2022). These data structures modify the standard insertion and deletion operations to ensure that nodes are well-distributed across the entire tree.

With the use of red-black trees and balanced insertion and deletion operations we can ensure $N \log N$ cost scaling for any evaluation of Fortune's algorithm.

4.3 The Event Queue

The event queue in Fortune's algorithm can be efficiently implemented with a **priority queue** data structure (Cormen et al. 2022). Priority queues are equipped with an operation for

"popping" the next element off of the queue as well as operations for inserting and deleting elements while maintaining the overall order of the elements.

To implement Fortune's algorithm we'll need to consider two types of events. Vertex events contain the position of the sweep line where an arc will collapse and a reference to the node in the beach line tree that represents that arc. Site events, on the other hand, contain the position where sweep line will encounter a new site as well as a reference to that site.

4.4 Event Processing

Fortune's algorithm proceeds by popping the next event off of the event queue, using that event to advanced the sweep line, and then updating the Voronoi graph as necessary. With the operations of doubly-connected edge lists and binary search trees, processing each event is relatively straightforward.

4.4.1 Processing Site Events

Processing a site event begins by using the arc search operation of the beach line tree to find the arc that intersects with the height of the new site by searching through the beach line tree.

Because splitting this arc will invalidate any existing vertex event centered on it, we have to remove any corresponding vertex event from the event queue. This is straightforward given the vertex event reference held in each node.

As the sweep line passes the new site, the intersecting arc will split into three arcs, with two new breakpoints between them (Figure 30a, Figure 30b). This expansion is implemented by creating two new nodes, one associated with the new site and one with the site generating the existing arc, and then inserting them into the beach line tree with two balanced insert-after operations (Figure 30c).

The half-edges of these two new nodes are always twinned to each other (Figure 31). Together the half-edges define a new Voronoi edge associated with the old and new sites.

Lastly incorporating the new site creates up to two new triplets of neighboring arcs that might generate new vertex events. Fortunately the nodes in the beach line tree representing these new triplets can be readily accessed with the predecessor and successor beach line operations.

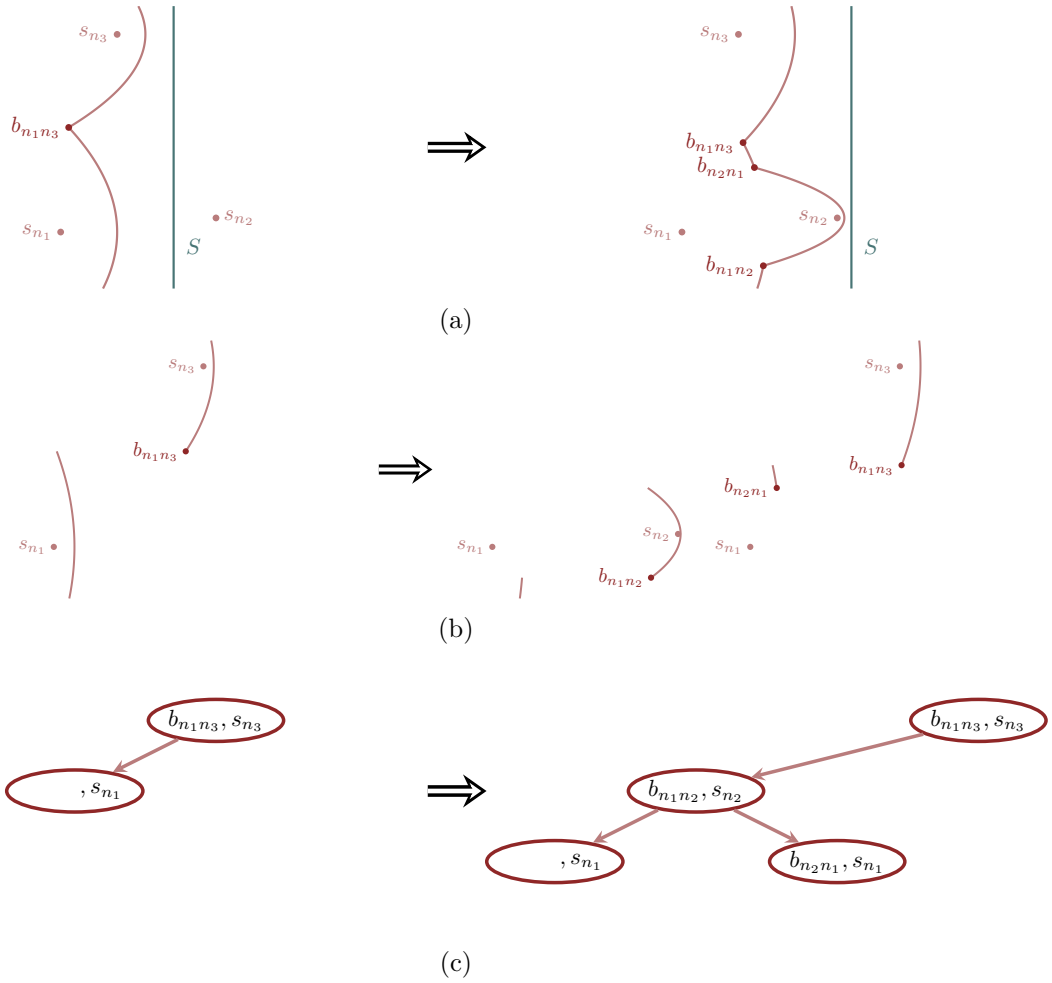


Figure 30: (a) A site event splits one of the arcs in the initial beach line into three arcs, with two new breakpoints. (b) This corresponds to two new arc-breakpoint pairs being introduced. (c) The addition of the two new arc-breakpoint pairs is readily implemented by twice applying the insert-after operation to the initial beach line tree.

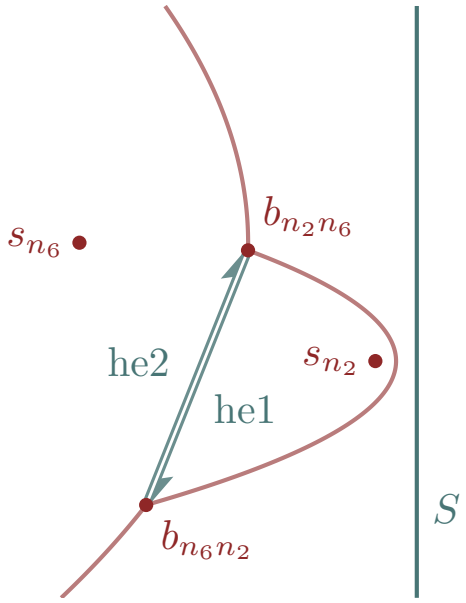


Figure 31: The introduction of a new site introduces a new Voronoi edge which is implemented with two new half-edges in the Voronoi graph that are twinned with each other. A reference to each of these half-edges is stored in the the two new nodes added to the beach line tree. For example here a reference to the half-edge `he1` is stored in the node $(b_{n_6n_2}, s_{n_2})$ while a reference to `he2` is stored in the node $(b_{n_2n_6}, s_{n_6})$.

4.4.2 Processing Vertex Events

The first step when processing a new vertex event is to remove any other vertex events that will become invalid after we remove the collapsing arc. This can be done efficiently by using the predecessor and successor operations of the beach line tree to find the neighboring arcs, and then removing any vertex events referenced in those arcs.

Next we manage the expansion of the Voronoi graph. We begin by adding a new vertex to the doubly-connected edge list at the point where the collapsing arc becomes singular and the neighboring breakpoints intersect.

Then we have to deal with the Voronoi edges. We create a new half-edge originating at the new vertex and its twin that, for the moment, is left unanchored (Figure 32). Lastly we have to carefully connect the previous and next references between these two new half-edges, the two half-edges associated with the neighboring breakpoints, and their twins (Figure 33).

At this point we're ready to remove the collapsing arc and replace its two neighboring breakpoints with a new merged breakpoint (Figure 34a, Figure 34b). This can be done by updating the half-edge reference of the successor node and then applying a balanced deletion operation (Figure 34c).

Lastly there are two new triplets of neighboring arcs that we have to check for possible new vertex events. Once again we can efficiently access these arcs by repeatedly applying the predecessor and successor beach line tree operations.

4.5 Clean Up

Once the event queue has been exhausted we are left with a doubly-connected edge list that contains all of the information about the Voronoi graph, and hence the Voronoi diagram.

At this point the beach line tree will *not* be empty. In fact the remaining breakpoint nodes reference all of the half-edges with unbounded origins. If we want to visualize the Voronoi graph neatly then we want to anchor these dangling edges by creating a bounding box and then adding a pseudo-vector to the doubly-connected edge list for the intersection of each dangling half-edge with that boundary.

5 Demonstration

To put all of this discussion into practice let's work through a full implementation of Fortune's algorithm in `R`. Although doubly-connected edge lists, binary search trees, and event queues can all be implemented with functional programming I strongly prefer a more object-oriented approach. To achieve object-oriented programming in `R` I will use `R6` classes (Chang 2025), at the cost of some clumsy interactions with base `R` graphics. At least using `R6` classes gets us

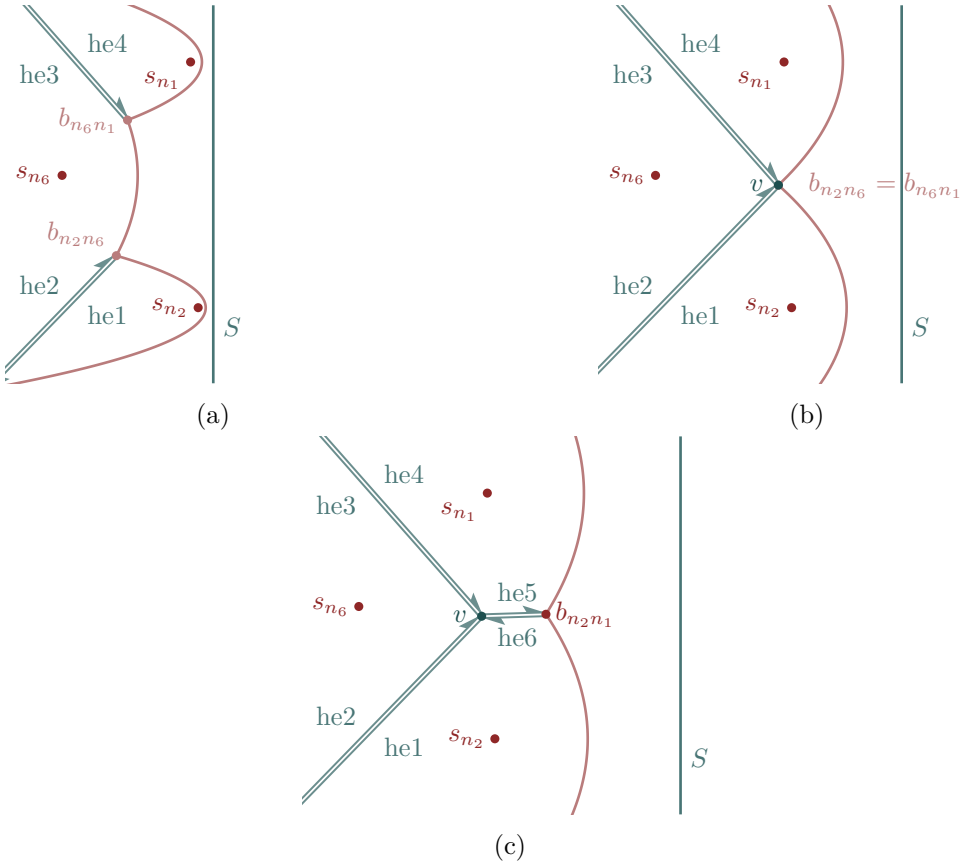


Figure 32: Vertex events require substantial updates to the half-edges of the Voronoi graph. (a) The breakpoints that merge in a vertex event contain references to two half-edges, here `he1` and `he3`. (b) When processing a vertex event we delete these references (c) Then we create two new half-edges, one of which is referenced by the new vertex, here `he5`, and one of which is referenced by the new breakpoint, here `he6`.

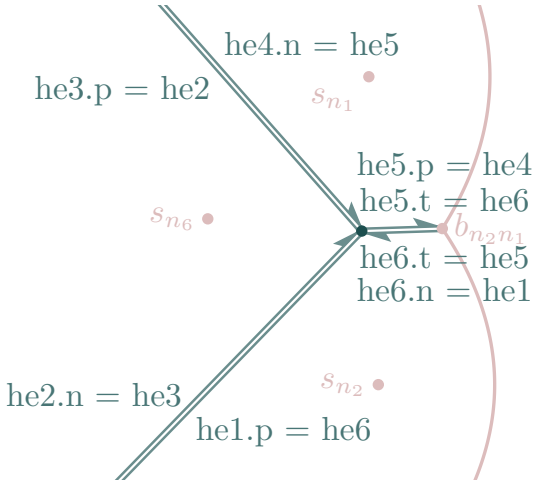


Figure 33: The trickiest part of processing a vertex event is making sure that the four existing half-edges and the two new half-edges all properly reference each other. These references ultimately define the structure of the Voronoi graph.

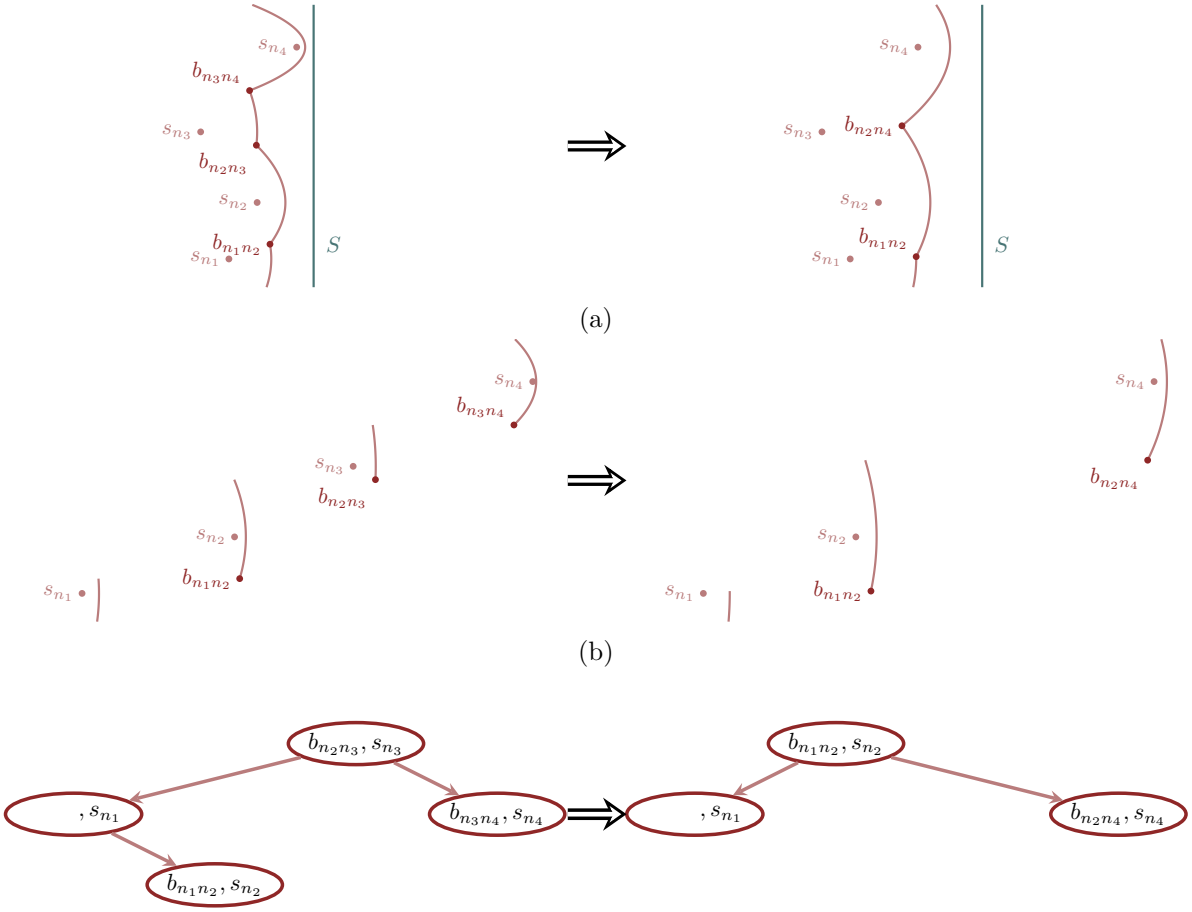
pass-by-reference semantics for class instances instead of R's usual call-by-need/lazy evaluation semantics!

```
library(R6)
```

```
Warning: package 'R6' was built under R version 4.3.3
```

Speaking of graphics, let's tweak the local graphics configuration.

```
par(family="serif", las=1, bty="l",
    cex.axis=1, cex.lab=1, cex.main=1,
    xaxs="i", yaxs="i", mar = c(5, 5, 3, 1))

c_light <- c("#DCBCBC")
c_light_highlight <- c("#C79999")
c_mid <- c("#B97C7C")
c_mid_highlight <- c("#A25050")
c_dark <- c("#8F2727")
c_dark_highlight <- c("#7C0000")

c_light_teal <- c("#6B8E8E")
c_mid_teal <- c("#487575")
c_dark_teal <- c("#1D4F4F")
```



(a)

(b)

(c)

Figure 34: (a) A vertex event removes one of the arcs in the initial beach line, replacing the two neighboring breakpoints with a new merged breakpoint. (b) This corresponds to removing one of the initial arc-breakpoint pairs and then updating the breakpoint information in another arc-breakpoint pair. (c) The removal of the deprecated arc-breakpoint pair is implemented by applying the deletion operation to the initial beach line tree.

5.1 Beach Line

To define a beach line we'll first need nodes.

This class implements arc-breakpoint nodes with satellite data for the associated site, half-edge, and vertex event. Note that the site variables are integer indices which will be used to access site information from global arrays.

The variable `red` denotes the coloring of the node which is used for balancing binary trees.

```
node <- R6Class("node",
  public = list(
    site = NULL,
    parent = NULL,
    left = NULL,
    right = NULL,
    red = FALSE,
    id = NULL,
    he = NULL,  # Half-edge originating at (predecessor$site, site)
    scheduled_vertex = NULL,
    initialize = function(beachline) {
      if (is.integer(beachline)) {
        self$id <- beachline
      } else {
        self$parent <- beachline$nil
        self$left <- beachline$nil
        self$right <- beachline$nil

        beachline$last_node_id <- beachline$last_node_id + 1
        self$id <- beachline$last_node_id
      }
    }
  )
)
```

The `id` variable allows us to determine if any two `node` instances refer to the same object. In many programming languages this is done by directly comparing memory addresses, but because `R6` classes are built up from `R` environments this is not possible here.

```
`==.node` = function(x, y) { x$id == y$id }
`!=.node` = function(x, y) { x$id != y$id }
```

A beach line tree itself is defined by its root node. Here we'll be using a self-balancing red-black binary tree which uses the coloring of the nodes to limit the height of any given tree. Red-black binary trees requires a dedicated null, or nil, node.

```
beachline <- R6Class("beachline",
  public = list(
   nil = NULL,
   root = NULL,
   last_node_id = NULL,
   initialize = function() {
     self$nil <- node$new(as.integer(0))
     self$root <- self$nil
     self$last_node_id <- as.integer(1)
   }
  ))
```

The `beachline` class also gives us a place to keep track of the node indices.

5.1.1 Black-Red Binary Tree Operations

The `insert_fixup` function balances a binary tree after we add the node `z`.

```
beachline$set("private", "left_rotate",
function(x) {
  if (x$right == self$nil)
    return()

  y <- x$right
  x$right <- y$left

  if (y$left != self$nil)
    y$left$parent <- x

  y$parent <- x$parent

  if (x$parent == self$nil) {
    self$root <- y
  } else {
    if (x$parent$left == x)
      x$parent$left <- y
    else
```

```
      x$parent$right <- y
  }

  y$left <- x
  x$parent <- y
})
```

```
beachline$set("private", "right_rotate",
function(x) {
  if (x$left == self$nil)
    return()

  y <- x$left
  x$left <- y$right

  if (y$right != self$nil)
    y$right$parent <- x

  y$parent <- x$parent

  if (x$parent == self$nil) {
    self$root <- y
  } else {
    if (x$parent$right == x)
      x$parent$right <- y
    else
      x$parent$left <- y
  }

  y$right <- x
  x$parent <- y
})
```

```
beachline$set("private", "insert_fixup",
function(z) {
  while (z$parent$red) {
    if (z$parent == z$parent$parent$left) {
      y <- z$parent$parent$right
      if (y$red) {
        z$parent$red <- FALSE
        y$red <- FALSE
        z$parent$parent$red <- TRUE
```

```
        z <- z$parent$parent
      } else {
        if (z == z$parent$right) {
          z <- z$parent
          private$left_rotate(z)
        }
        z$parent$red <- FALSE
        z$parent$parent$red <- TRUE
        private$right_rotate(z$parent$parent)
      }
    } else {
      y <- z$parent$parent$left

      if (y$red) {
        z$parent$red <- FALSE
        y$red <- FALSE
        z$parent$parent$red <- TRUE
        z <- z$parent$parent
      } else {
        if (z == z$parent$left) {
          z <- z$parent
          private$right_rotate(z)
        }
        z$parent$red <- FALSE
        z$parent$parent$red <- TRUE
        private$left_rotate(z$parent$parent)
      }
    }
  }
  self$root$red <- FALSE
})
```

The `insert_fixup` function then allows us to implement balanced `insert-before` and `insert-after` operations.

```
beachline$set("public", "insert_before",
function(x, y) {
  if (x$left == self$nil) {
    x$left <- y
    y$parent <- x
  } else {
    z <- self$predecessor(x)
```

```
    z$right <- y
    y$parent <- z
  }

  y$left <- self$nil
  y$right <- self$nil
  y$red <- TRUE

  private$insert_fixup(y)
})
```

```
beachline$set("public", "insert_after",
function(x, y) {
  if (x$right == self$nil) {
    x$right <- y
    y$parent <- x
  } else {
    z <- self$successor(x)
    z$left <- y
    y$parent <- z
  }

  y$left <- self$nil
  y$right <- self$nil
  y$red <- TRUE

  private$insert_fixup(y)
})
```

Deleting a node from a red-black tree takes some care. To properly remove a node we'll need to be able to calculate the boundaries of sub-trees.

```
beachline$set("public", "min",
function(x) {
  if (x == self$nil)
    return(self$nil)
  while (x$left != self$nil)
    x <- x$left
  x
})
```

```
beachline$set("public", "max",
function(x) {
  if (x == self$nil)
    return(self$nil)
  while (x$right != self$nil)
    x <- x$right
  x
})
```

Then we'll need to carefully balance the tree.

```
beachline$set("private", "transplant",
function(u, v) {
  if (u$parent == self$nil) {
    self$root <- v
  } else {
    if (u == u$parent$left) {
      u$parent$left <- v
    } else {
      u$parent$right <- v
    }
  }
  v$parent <- u$parent
})
```

```
beachline$set("private", "delete_fixup",
function(x) {
  while (x != self$root && !x$red) {
    if (x == x$parent$left) {
      w <- x$parent$right

      if (w$red) {
        w$red <- FALSE
        x$parent$red <- TRUE
        private$left_rotate(x$parent)
        w <- x$parent$right
      }

      if (!w$left$red && !w$right$red) {
        w$red <- TRUE
        x <- x$parent
      } else {
```

```
      if (!w$right$red) {
        w$left$red <- FALSE
        w$red <- TRUE
        private$right_rotate(w)
        w <- x$parent$right
      }

      w$red <- x$parent$red
      x$parent$red <- FALSE
      w$right$red <- FALSE
      private$left_rotate(x$parent)
      x <- self$root
    }
  } else {
    w <- x$parent$left

    if (w$red) {
      w$red <- FALSE
      x$parent$red <- TRUE
      private$right_rotate(x$parent)
      w <- x$parent$left
    }

    if (!w$right$red && !w$left$red) {
      w$red <- TRUE
      x <- x$parent
    } else {
      if (!w$left$red) {
        w$right$red <- FALSE
        w$red <- TRUE
        private$left_rotate(w)
        w <- x$parent$left
      }

      w$red <- x$parent$red
      x$parent$red <- FALSE
      w$left$red <- FALSE
      private$right_rotate(x$parent)
      x <- self$root
    }
  }
}
```

```
  x$red <- FALSE
})
```

```
beachline$set("public", "delete",
function(z) {
  y <- z
  y_old_red <- y$red

  if (z$left == self$nil) {
    x <- z$right
    private$transplant(z, z$right)
  } else if (z$right == self$nil) {
    x <- z$left
    private$transplant(z, z$left)
  } else {
    y <- self$min(z$right)
    y_old_red <- y$red
    x <- y$right

    if (y != z$right) {
      private$transplant(y, y$right)
      y$right <- z$right
      y$right$parent <- y
    } else {
      x$parent <- y
    }

    private$transplant(z, y)
    y$left <- z$left
    y$left$parent <- y
    y$red <- z$red
  }

  if (!y_old_red)
    private$delete_fixup(x)
})
```

Neighboring nodes can be accessed with `precursor` and `successor` operations.

```
beachline$set("public", "predecessor",
function(x) {
  if (x$left != self$nil) {
```

```
    return(self$max(x$left))
  } else {
    y <- x$parent
    while (   y != self$nil
              && x == y$left) {
      x <- y
      y <- y$parent
    }
    return(y)
  }
})
```

```
beachline$set("public", "successor",
function(x) {
  if (x$right != self$nil) {
    return(self$min(x$right))
  } else {
    y <- x$parent
    while (   y != self$nil
           && x == y$right) {
      x <- y
      y <- y $parent
    }
    return(y)
  }
})
```

5.1.2 Visualization Functions

Visualizing the nodes in a binary search tree is particularly elegant with recursive functions. That said any function that interacts with base `R` graphics objects has to be external to the `R6` class.

```
beachline$set("public", "height",
function(node=NULL) {
  if (is.null(node))
    node <- self$root

  if (node == self$nil)
    return(0)
```

```
  left_height <- self$height(node$left)
  right_height <- self$height(node$right)

  1 + max(left_height, right_height)
})
```

Just to minimize the text, we'll display only the site that generates the arc associated with each node and not any information about the lower breakpoint.

```
plot_node <- function(nil, node,
                      x, y,
                      delta_x, delta_y,
                      text_cex) {

  if (node$left != nil) {
    lines(c(x, x - delta_x), c(y, y - delta_y),
          col=c_mid_teal, lwd=2)
    plot_node(nil, node$left,
              x - delta_x, y - delta_y,
              delta_x / 2, delta_y, text_cex)
  }

  if (node$right != nil) {
    lines(c(x, x + delta_x), c(y, y - delta_y),
          col=c_mid_teal, lwd=2)
    plot_node(nil, node$right,
              x + delta_x, y - delta_y,
              delta_x / 2, delta_y, text_cex)
  }

  if (node$red)
    points(x, y, pch=16, col=c_dark, cex=3)
  else
    points(x, y, pch=16, col="black", cex=3)

  text(x, y, col="white", cex=text_cex,
       labels=node$site)
}
```

```
plot_tree <- function(tree, delta=0.5, text_cex=0.75) {
  H <- tree$height()
```

```
  par(mar=c(0, 0, 0, 0))

  plot(NULL,
       xlab='', ylab='', xaxt="n", yaxt="n", frame.plot=F,
       xlim=c(-delta * (2**(H - 1) + 1), delta * (2**(H - 1) + 1)),
       ylim=c(-(H - 0.5) * delta, 0.5 * delta))

  plot_node(tree$nil, tree$root, 0, 0,
            delta * 2**(H - 2), delta, text_cex)
}
```

5.1.3 Fortune's Algorithm Geometry Calculations

As we've seen, implementing Fortune's algorithm requires a lot of geometric calculations.

For example we'll need to be able to compute the vertical position of the breakpoint separating two neighboring arcs on a beach line. Site positions are accessed by indexing the global arrays `xs` and `ys`.

```
beachline$set("public", "pair_intersection",
function(n1, n2, S) {
  if (xs[n1] == S)
    return(ys[n1])
  if (xs[n2] == S)
    return(ys[n2])

  a <- xs[n2] - xs[n1]
  b <- ys[n2] * (S - xs[n1]) - ys[n1] * (S - xs[n2])
  c <- (   ys[n2]**2 * (S - xs[n1])
         - ys[n1]**2 * (S - xs[n2])
         - (S - xs[n1]) * (S - xs[n2]) * (xs[n2] - xs[n1]) )

  e <- b / a
  d <- sqrt( e**2 - c / a )
  e - sign(a) * d
})
```

The horizontal position of a breakpoint can be recovered from the parabola defining each arc. Here `(fx, fy)` defines the focal point of the parabola and `S` the vertical directrix.

```
beachline$set("public", "parabola_xs",
function(parabola_ys, fx, fy, S) {
  0.5 * ( (fx + S) - (parabola_ys - fy)**2 / (S - fx) )
})
```

We can't forget the circumcircles.

```
beachline$set("public", "circumcircle",
function(n1, n2, n3) {
  D <- 2 * (     xs[n1] * (ys[n2] - ys[n3])
              + xs[n2] * (ys[n3] - ys[n1])
              + xs[n3] * (ys[n1] - ys[n2]) )
  xc <- (    (xs[n1]**2 + ys[n1]**2) * (ys[n2] - ys[n3])
           + (xs[n2]**2 + ys[n2]**2) * (ys[n3] - ys[n1])
           + (xs[n3]**2 + ys[n3]**2) * (ys[n1] - ys[n2]) ) / D

  yc <- (    (xs[n1]**2 + ys[n1]**2) * (xs[n3] - xs[n2])
           + (xs[n2]**2 + ys[n2]**2) * (xs[n1] - xs[n3])
           + (xs[n3]**2 + ys[n3]**2) * (xs[n2] - xs[n1]) ) / D

  dx <- xc - xs[n1]
  dy <- yc - ys[n1]

  if (dx > dy) {
    r <- abs(dx) * sqrt(1 + (dy / dx)**2)
  } else {
    r <- abs(dy) * sqrt(1 + (dx / dy)**2)
  }

  c(xc, yc, r)
})
```

We can use these geometric functions to, for example, find the node representing the arc that intersects with a given vertical position.

```
beachline$set("private", "arc_search",
function(z, y, S) {
  while (1) {
    z_left <- self$predecessor(z)
    if (z_left != self$nil) {
      by <- self$pair_intersection(z_left$site, z$site, S)
```

```
      if (y <= by) {
        z <- z$left
        next
      }
    }

    z_right <- self$successor(z)
    if (z_right != self$nil) {
      by <- self$pair_intersection(z$site, z_right$site, S)
      if (y > by) {
        z <- z$right
        next
      }
    }

    break
  }

  return(z)
})
```

5.1.4 Fortune's Algorithm Event Functions

With all of these auxiliary functions we are finally ready to implement event processing functions.

One operating that appears over and over again when processing events is checking triplets of arcs to see if they define a valid vertex event and, if so, adding that event to the event queue `queue`.

As we discussed in Section 3.4.2 there is undoubtably a more elegant way to verify that `alpha` will collapse as the sweep line progresses than comparing `upper_int` to `lower_int`, but it will do for now.

```
beachline$set("private", "schedule_new_vertex_event",
function(queue, alpha_l, alpha, alpha_r, S) {
  # Skip vertex if any of the neighboring arcs are nil
  if (alpha_l == self$nil || alpha_r == self$nil) {
    return()
  }

  # Skip vertex if any of the sites are the same
```

```
  if (alpha_l$site == alpha_r$site) {
    return()
  }

  # Construct circumcircle
  int <- self$circumcircle(alpha_l$site,
                           alpha$site,
                           alpha_r$site)
  S_new <- int[1] + int[3]


  lower_int <- self$pair_intersection(alpha_l$site,
                                      alpha$site, S_new)
  upper_int <- self$pair_intersection(alpha$site,
                                      alpha_r$site, S_new)

  if (   S_new > S
      && (abs(upper_int - lower_int) < (1e6 * .Machine$double.eps)) ) {
    vertex_event <- event$new(NULL, S_new, NULL, alpha)
    alpha$scheduled_vertex <- vertex_event
    queue$insert_event(vertex_event)
  }
})
```

Now we can process site events.

```
beachline$set("private", "process_site_event",
function(queue, graph, new_event) {
  new_site <- new_event$site

  # Initialize beachline if empty
  if (self$root == self$nil) {
    self$root <- node$new(self)
    self$root$site <- new_site
    return()
  }

  # Find beachline arc that intersects
  # vertical position of new site
  alpha <- private$arc_search(self$root,
                             ys[new_site],
                             new_event$x)
```

```
  # Delete any scheduled vertex
  # events at intersecting arc
  if (!is.null(alpha$scheduled_vertex)) {
    queue$delete_event(alpha$scheduled_vertex)
    alpha$scheduled_vertex <- NULL
  }

  # Split intersecting arc
  alpha_r <- node$new(self)
  alpha_r$site <- new_site

  self$insert_after(alpha, alpha_r)

  alpha_rr <- node$new(self)
  alpha_rr$site <- alpha$site

  self$insert_after(alpha_r, alpha_rr)

  # Instantiate and connect new half-edges
  new_he <- graph$new_half_edge(left_face=alpha$site)     # (alpha, alpha_r)
  new_he$twin <- graph$new_half_edge(left_face=new_site,  # (alpha_r, alpha_rr)
                                     twin=new_he)

  alpha_r$he <- new_he
  alpha_rr$he <- new_he$twin

  # Attempt to schedule vertex events
  # for the two new arc triplets
  alpha_l <- self$predecessor(alpha)
  private$schedule_new_vertex_event(queue,
                                    alpha_l, alpha, alpha_r,
                                    new_event$x)

  alpha_rrr <- self$successor(alpha_rr)
  private$schedule_new_vertex_event(queue,
                                    alpha_r, alpha_rr, alpha_rrr,
                                    new_event$x)
})
```

Vertex event processing follows.

```
beachline$set("private", "process_vertex_event",
function(queue, graph, new_event) {
  alpha <- new_event$arc

  alpha_r <- self$successor(alpha)
  alpha_rr <- self$successor(alpha_r)

  alpha_l <- self$predecessor(alpha)
  alpha_ll <- self$predecessor(alpha_l)

  # Clean up deprecated vertex events in the queue
  if (!is.null(alpha_r$scheduled_vertex)) {
    queue$delete_event(alpha_r$scheduled_vertex)
    alpha_r$scheduled_vertex <- NULL
  }

  if (!is.null(alpha_l$scheduled_vertex)) {
    queue$delete_event(alpha_l$scheduled_vertex)
    alpha_l$scheduled_vertex <- NULL
  }

  # Create vertex
  int <- self$circumcircle(alpha_l$site, alpha$site, alpha_r$site)
  v <- vertex$new(int[1:2], alpha_l$site, alpha$site, alpha_r$site)
  graph$add_vertex(v)

  # Connect vertex
  alpha_r$he$origin <- v
  alpha$he$origin <- v

  new_he <- graph$new_half_edge(origin=v,
                                left_face=alpha_r$site)
  new_he$twin <- graph$new_half_edge(left_face=alpha_l$site,
                                     twin=new_he)

  v$he <- new_he

  # Connect half-edges
  alpha$he$twin$nxt <- alpha_r$he
  alpha_r$he$prv <- alpha$he$twin

  alpha_r$he$twin$nxt <- new_he
```

```
  new_he$prv <- alpha_r$he$twin

  new_he$twin$nxt <- alpha$he
  alpha$he$prv <- new_he$twin

  # Update active half-edges
  alpha_r$he <- new_he$twin

  # Delete deprecated arcs
  self$delete(alpha)

  # Check for new vertex events
  private$schedule_new_vertex_event(queue,
                                    alpha_ll, alpha_l, alpha_r,
                                    new_event$x)

  private$schedule_new_vertex_event(queue,
                                    alpha_l, alpha_r, alpha_rr,
                                    new_event$x)
})
```

Lastly we'll wrap both of these event processing functions together.

```
beachline$set("public", "process_next_event",
function(queue, graph) {
  next_event <- queue$pop_next_element()

  if (!is.null(next_event$site)) {
    private$process_site_event(queue,
                               graph,
                               next_event)
  } else {
    private$process_vertex_event(queue,
                                 graph,
                                 next_event)
  }
})
```

5.1.5 Fortune's Algorithm Visualization Functions

To monitor the progress of Fortune's algorithm we'll define some functions that allow us to plot the shape of the beach line for any sweep line position, as well as the behavior of any

preliminary Voronoi edges.

```
plot_arcs <- function(bl, S, x=NULL, y_min, y_max) {
  if (is.null(x))
    x <- bl$root

  if (x == bl$nil)
    return(y_max)

  # Recurse right
  y_max <- plot_arcs(bl, S, x$right, y_min, y_max)

  # Plot arc
  if (is.null(x$he)) {
    # Plot parabola
    if (y_min + 1e-3 < y_max) {
      parabola_ys <- seq(y_min, y_max, 1e-3)
      lines(bl$parabola_xs(parabola_ys, xs[x$site], ys[x$site], S),
            parabola_ys, lty=2, lwd=2, col=c_mid)
    }
  } else {
    # Compute left breakpoint
    by <- bl$pair_intersection(x$he$left_face, x$site, S)
    bx <- bl$parabola_xs(by, xs[x$site], ys[x$site], S)

    # Plot parabola
    if (by + 1e-3 < y_max) {
      parabola_ys <- seq(by, y_max, 1e-3)
      lines(bl$parabola_xs(parabola_ys, xs[x$site], ys[x$site], S),
            parabola_ys, lty=2, lwd=2, col=c_mid)
    }

    # Plot left breakpoint
    points(bx, by, col=c_light, pch=16)
    text(bx + 0.1, by,
        labels=paste0(x$he$left_face, ",", x$site))

    # Save breakpoint position for plotting half-edges
    x$he$pseudo_origin <- c(bx, by)

    # Update upper bound
    y_max <- by
  }
```

```
  # Recurse left
  y_max <- plot_arcs(bl, S, x$left, y_min, y_max)

  return(y_max)
}
```

```
plot_beachline <- function(bl, S,
                           xlim=c(-4, 4),
                           ylim=c(-4, 4)) {

  plot(xs, ys, col=c_dark, pch=16,
       xlim=xlim, ylim=ylim,
       main="")

  # Plot sweep line
  abline(v=S, col=c_mid_teal, lty=2)

  # Plot site parabolas given the sweep line
  for (i in which(xs < S)) {
    text(xs[i] + 0.1, ys[i], i)
    parabola_ys <- seq(ylim[1], ylim[2], 0.01)
    lines(bl$parabola_xs(parabola_ys, xs[i], ys[i], S),
          parabola_ys, lwd=2, col="#DDDDDD")
  }

  # Plot beach line
  if (bl$root != bl$nil)
    r <- plot_arcs(bl, S, y_min=ylim[1], y_max=ylim[2])
}
```

```
plot_beachline_half_edges <- function(bl, S, x=NULL,
                                      lwd=2, col=c_mid_teal,
                                      delta=0.08, eps=0.04,
                                      theta=60, gamma=0.0075) {
  if (is.null(x))
    x <- bl$root

  if (x != bl$nil) {
    # Recurse right
    plot_beachline_half_edges(bl, S, x$right,
                              lwd, col, delta,
```

```
                              eps, theta, gamma)

    # Plot arc
    if (!is.null(x$he)) {
      if (is.null(x$he$origin)) p1 <- x$he$pseudo_origin
      else                      p1 <- x$he$origin$pos

      if (is.null(x$he$twin$origin)) p2 <- x$he$twin$pseudo_origin
      else                           p2 <- x$he$twin$origin$pos

      plot_half_edge(p1, p2, lwd, col, delta, eps, theta, gamma)
    }

    # Recurse left
    plot_beachline_half_edges(bl, S, x$left,
                              lwd, col, delta,
                              eps, theta, gamma)
  }
}
```

5.2 Event Queue

We'll overload the `event` class to handle both site and vertex events at the same time. The variable `site` is used for only site events while the variable `arc` is used for only vertex events.

```
event <- R6Class("event",
  public=list(
  x=NULL,
  site=NULL,
  arc=NULL,
  queue_idx=NULL,
  initialize = function(i, x, site, arc) {
    self$queue_idx <- i
    self$x <- x
    self$site <- site
    self$arc <- arc
  }))
```

The event queue itself is a pretty straightforward implementation of a priority queue data structure. When creating a new event queue instance we pass in a vector of indices for the available sites which are then used to create the site events that initially populate the queue.

```
event_queue <- R6Class("event_queue",
  public=list(
   idxs=NULL,
   events=NULL,
   length=0,
   initialize = function(sites, xs) {
     self$length <- length(sites)
     self$idxs <- 1:self$length
     for (i in seq_along(sites)) {
       new_event <- event$new(i, xs[i], i, NULL)
       self$events <- append(self$events,
                             new_event)
     }
     for (i in (self$length %/% 2):1) {
       private$min_heapify(i)
     }
   },
   is_empty = function() {
     self$length == 0
   },
   is_not_empty = function() {
     self$length > 0
   }))
```

```
event_queue$set("private", "min_heapify",
function(i) {
  l <- 2 * i
  r <- l + 1

  if (   l <= self$length
      &&   self$events[[self$idxs[l]]]$x
         < self$events[[self$idxs[i]]]$x) {
    smallest <- l
  } else {
    smallest <- i
  }

  if (   r <= self$length
         &&   self$events[[self$idxs[r]]]$x
         < self$events[[self$idxs[smallest]]]$x) {
    smallest <- r
  }
```

```
  if (smallest != i) {
    n <- self$events[[self$idxs[i]]]$queue_idx
    self$events[[self$idxs[i]]]$queue_idx <-
      self$events[[self$idxs[smallest]]]$queue_idx
    self$events[[self$idxs[smallest]]]$queue_idx <- n

    n <- self$idxs[i]
    self$idxs[i] <- self$idxs[smallest]
    self$idxs[smallest] <- n

    private$min_heapify(smallest)
  }
})
```

```
event_queue$set("public", "insert_event",
function(new_event) {
  self$length <- self$length + 1
  self$idxs <- c(self$idxs, length(self$events) + 1)

  new_event$queue_idx <- self$length
  self$events <- append(self$events, new_event)

  i <- self$length
  while (   i > 1
            &&   self$events[[self$idxs[i %/% 2]]]$x
            > self$events[[self$idxs[i]]]$x) {
    n <- self$events[[self$idxs[i]]]$queue_idx
    self$events[[self$idxs[i]]]$queue_idx <-
      self$events[[self$idxs[i %/% 2]]]$queue_idx
    self$events[[self$idxs[i %/% 2]]]$queue_idx <- n

    n <- self$idxs[i]
    self$idxs[i] <- self$idxs[i %/% 2]
    self$idxs[i %/% 2] <- n

    i <- i %/% 2
  }
})
```

```
event_queue$set("public", "insert_site_event",
function(site, xs) {
```

```
  self$insert_event(event$new(NULL, xs[site], site, NULL))
})
```

```
event_queue$set("public", "insert_vertex_event",
function(S, arc) {
  self$insert_event(event$new(NULL, S, NULL, arc))
})
```

```
event_queue$set("public", "pop_next_element",
function() {
  if (self$length == 0) {
    print("The queue is empty!")
    return(NULL)
  }

  next_element <- self$events[[self$idxs[1]]]

  self$idxs[1] <- self$idxs[self$length]
  self$events[[self$idxs[1]]]$queue_idx <- 1

  self$idxs <- self$idxs[-self$length]
  self$length <- self$length - 1

  private$min_heapify(1)

  next_element
})
```

```
event_queue$set("public", "delete_event",
function(event) {
  event$x <- self$events[[self$idxs[1]]]$x - 1

  i <- event$queue_idx
  while (   i > 1
         &&   self$events[[self$idxs[i %/% 2]]]$x
            > self$events[[self$idxs[i]]]$x) {
    n <- self$events[[self$idxs[i]]]$queue_idx
    self$events[[self$idxs[i]]]$queue_idx <-
      self$events[[self$idxs[i %/% 2]]]$queue_idx
    self$events[[self$idxs[i %/% 2]]]$queue_idx <- n

    n <- self$idxs[i]
```

```
    self$idxs[i] <- self$idxs[i %/% 2]
    self$idxs[i %/% 2] <- n

    i <- i %/% 2
  }
  e <- self$pop_next_element()
})
```

```
event_queue$set("public", "check_min_heapify",
function() {
  if (self$length > 0) {
    for (i in 1:self$length)  {
      if (i %/% 2 == 0) next
      print(   self$events[[self$idxs[i %/% 2]]]$x
            <= self$events[[self$idxs[i]]]$x)
    }
  }
})
```

```
event_queue$set("public", "next_event",
function() {
  if (self$length > 0)
    self$events[[self$idxs[1]]]
})
```

5.3 Doubly-Connected Edge List

In order to represent a Voronoi graph we'll need classes for half_edges, vertices, and faces and then the doubly-connected edge list that encapsulates them.

Like the beach line nodes the half-edges uses a global indexing to allow for equality comparisons. The `pseudo_origin` variable is used for plotting dangling half-edges that have not yet been anchored to vertices.

```
half_edge <- R6Class("half_edge",
  public=list(
   pseudo_origin=NULL,
   origin=NULL,
   left_face=NULL,
   twin=NULL,
   prv=NULL,
```

```
    nxt=NULL,
    visited=FALSE,
    id=NULL,
    initialize = function(graph, o, lf, t, p, n) {
      self$origin <- o
      self$left_face <- lf
      self$twin   <- t
      self$prv    <- p
      self$nxt    <- n

      graph$last_half_edge_id <- graph$last_half_edge_id + 1
      self$id <- graph$last_half_edge_id
    }))
```

```
`==.half_edge` = function(x, y) { x$id == y$id }
`!=.half_edge` = function(x, y) { x$id != y$id }
```

```
vertex <- R6Class("vertex",
  public=list(
    pos=NULL,
    n1=NULL,
    n2=NULL,
    n3=NULL,
    he=NULL,
    initialize = function(p, n1, n2, n3) {
      self$pos <- p
      self$n1 <- n1
      self$n2 <- n2
      self$n3 <- n3
    }))
```

```
face <- R6Class("face",
  public=list(
    site=NULL,
    he=NULL,
    initialize = function(site, he) {
      self$site <- site
      self$he <- he
    }))
```

```
dcel <- R6Class("dcel",
  public=list(
    vertices=NULL,
    pseudo_vertices=NULL,
    faces=NULL,
    xlim=NULL,
    ylim=NULL,
    last_half_edge_id = NULL,
    initialize = function() {
      self$vertices <- list()
      self$pseudo_vertices <- list()
      self$faces <- list()
      self$last_half_edge_id <- as.integer(1)
    },
    add_vertex = function(v) {
      self$vertices <-
        append(self$vertices, v)
    },
    N_vertices = function() {
      length(self$vertices)
    },
    add_pseudo_vertex = function(pv) {
      self$pseudo_vertices <-
        append(self$pseudo_vertices, pv)
    },
    N_pseudo_vertices = function() {
      length(self$pseudo_vertices)
    },
    add_face = function(f) {
      self$faces <-
        append(self$faces, f)
    },
    N_faces = function() {
      length(self$faces)
    }))
```

```
dcel$set("public", "new_half_edge",
function(origin=NULL, left_face=NULL,
         twin=NULL, prv=NULL, nxt=NULL) {
  half_edge$new(self, origin, left_face, twin, prv, nxt)
})
```

There are all kinds of operations that we can implement from the structure encoded in a doubly-connected edge list. For example we can automatically compute a bounding box that contains all of the vertices. The parameter `q` specifies how much multiplicative margin we add around the smallest bounding box.

```
dcel$set("public", "compute_bounding_box",
function(q=0.05) {
  # Set bounding box edges to
  # extreme vertex positions
  self$xlim <- c(self$vertices[[1]]$pos[1],
                 self$vertices[[1]]$pos[1])
  self$ylim <- c(self$vertices[[1]]$pos[2],
                 self$vertices[[1]]$pos[2])

  for (v in self$vertices[-1]) {
    if (v$pos[1] > self$xlim[2]) {
      self$xlim[2] <- v$pos[1]
    } else if (v$pos[1] < self$xlim[1]) {
      self$xlim[1] <- v$pos[1]
    }

    if (v$pos[2] > self$ylim[2]) {
      self$ylim[2] <- v$pos[2]
    } else if (v$pos[2] < self$ylim[1]) {
      self$ylim[1] <- v$pos[2]
    }
  }

  # Add proportional margins to bounding box
  dx <- self$xlim[2] - self$xlim[1]
  self$xlim[1] <- self$xlim[1] - q * dx
  self$xlim[2] <- self$xlim[2] + q * dx

  dy <- self$ylim[2] - self$ylim[1]
  self$ylim[1] <- self$ylim[1] - q * dy
  self$ylim[2] <- self$ylim[2] + q * dy
})
```

Given a bounding box we can then introduce pseudo-vertices to anchor any dangling half-edges.

```
dcel$set("public", "anchor_dangling_edges",
function(z, xs, ys) {
  # Dangling edge
  he <- z$he

  if (!is.null(he)) {
    nu <- he$twin$origin$pos
    zeta <- -nu

    i_left <- he$left_face
    i_right <- he$twin$left_face
    dangling_he <- he

    # Anchored edges
    he <- he$nxt
    zeta <- zeta + he$twin$origin$pos

    he <- he$twin$nxt
    zeta <- zeta + he$twin$origin$pos

    # Compute bisecting vector in the opposite
    # direction of the anchored edges
    delta <- c(ys[i_right] - ys[i_left],
               xs[i_left]  - xs[i_right])

    # Compute closest bounding box intersection
    bh <- ifelse(delta[1] > 0, self$xlim[2], self$xlim[1])
    bv <- ifelse(delta[2] > 0, self$ylim[2], self$ylim[1])

    pv_x <- bh
    pv_y <- nu[2] + delta[2] * (pv_x - nu[1]) / delta[1]
    if (pv_y < self$ylim[1] || pv_y > self$ylim[2]) {
      pv_y <- bv
      pv_x <- nu[1] + delta[1] * (pv_y - nu[2]) / delta[2]
    }

    # Anchor dangling edge to a new pseudo-vertex
    # positioned at the bounding box intersection
    pv <- vertex$new(c(pv_x, pv_y), NULL, NULL, NULL)
    dangling_he$origin <- pv
    pv$he <- dangling_he
    self$add_pseudo_vertex(pv)
```

```
    # Recurse
    self$anchor_dangling_edges(z$left, xs, ys)
    self$anchor_dangling_edges(z$right, xs, ys)
  }
})
```

More relevant to the construction of a Voronoi diagrams, we also can follow the half-edges to automatically identify all of the faces in the graph, and hence cells in the diagram.

```
dcel$set("private", "reset_visitation",
function() {
  for (v in self$vertices) {
    he <- v$he
    he$visited <- FALSE
    he$twin$visited <- FALSE
    he <- he$prv
    he$visited <- FALSE
    he$twin$visited <- FALSE
    he <- he$twin$prv
    he$visited <- FALSE
    he$twin$visited <- FALSE
  }
})
```

```
dcel$set("public", "find_faces",
function(he=NULL) {

  if (is.null(he)) {
    private$reset_visitation()
    he <- self$vertices[[1]]$he
  }

  init_he <- he
  open_face <- FALSE
  closed_face <- FALSE

  while (!he$visited) {
    he$visited <- TRUE
    he <- he$nxt
    if (is.null(he)) {
      open_face <- TRUE
```

```
      break
    }
    if (he == init_he) {
      closed_face <- TRUE
      break
    }
  }

  he <- init_he
  if (open_face) {
    # Reverse to beginning of open boundary
    while(!is.null(he$prv)) {
      he <- he$prv
    }

    self$add_face(face$new(he$left_face, he))

    # Shoot along open boundary
    while(!is.null(he$nxt)) {
      if(!he$twin$visited) self$find_faces(he$twin)
      he <- he$nxt
    }
  }
  if (closed_face) {
    self$add_face(face$new(he$left_face, he))

    # Shoot along closed boundary
    while(1) {
      if(!he$twin$visited) self$find_faces(he$twin)
      he <- he$nxt
      if (he == init_he) break
    }
  }
})
```

Why spend all of this time creating graphs if we can't visualize them with aesthetic flair?

```
plot_half_edge <- function(pi, pf, lwd, col=c_mid_teal,
                           delta, eps, theta, gamma) {
  r <- pf - pi
  v <- r / sqrt(sum(r**2))
```

```
  r_perp <- c(r[2], -r[1])
  w <- r_perp / sqrt(sum(r_perp**2))

  # Perpendicular offset
  gammax <- -gamma * w[1]
  gammay <- -gamma * w[2]

  # Arrow body
  lines(c(pi[1], pf[1]) + gammax,
        c(pi[2], pf[2]) + gammay,
        lwd=lwd, col=col)

  # Arrow head
  phi_rad <- (90 - theta) * (3.14159265 / 180)
  p3 <- pf - (delta + eps * tan(phi_rad)) * v - eps * w
  p4 <- pf - delta * v

  polygon(c(pf[1], p3[1], p4[1], pf[1]) + gammax,
          c(pf[2], p3[2], p4[2], pf[2]) + gammay,
          col=col, border=NA)
}
```

```
plot_vertex_edges <- function(v, lwd=2, ucol=c_mid_teal,
                              delta=0.1, eps=0.05,
                              theta=60, gamma=0.025) {
  points(v$pos[1], v$pos[2],
         pch=16, col=c_light_teal)

  he <- v$he

  while(1) {
    if (is.null(he$origin)) p1 <- he$pseudo_origin
    else                    p1 <- he$origin$pos

    if (is.null(he$twin$origin)) p2 <- he$twin$pseudo_origin
    else                         p2 <- he$twin$origin$pos

    plot_half_edge(p1, p2,
                   lwd, ifelse(he$visited, c_mid, ucol),
                   delta, eps, theta, gamma)

    he <- he$prv$twin
```

```
    if (is.null(he) || he == v$he) break
  }
}
```

```
plot_face_boundary <- function(f, lwd=2, col=c_mid_teal,
                               delta=0.1, eps=0.05,
                               theta=60, gamma=0.025) {
  he <- f$he

  while (1) {
    points(he$origin$pos[1], he$origin$pos[2],
           pch=16, col=col)

    plot_half_edge(he$origin$pos, he$twin$origin$pos,
                   lwd, col,
                   delta, eps, theta, gamma)

    he <- he$nxt
    if (is.null(he) || he == f$he) break
  }
}
```

```
plot_face <- function(f, col=c_mid_teal) {
  # Trace boundary
  he <- f$he

  perimeter_xs <- c()
  perimeter_ys <- c()

  while (1) {
    perimeter_xs <- c(perimeter_xs, he$origin$pos[1])
    perimeter_ys <- c(perimeter_ys, he$origin$pos[2])

    if (is.null(he$nxt)) {
      perimeter_xs <- c(perimeter_xs, he$twin$origin$pos[1])
      perimeter_ys <- c(perimeter_ys, he$twin$origin$pos[2])
      break
    }

    he <- he$nxt

    if (he == f$he) {
```

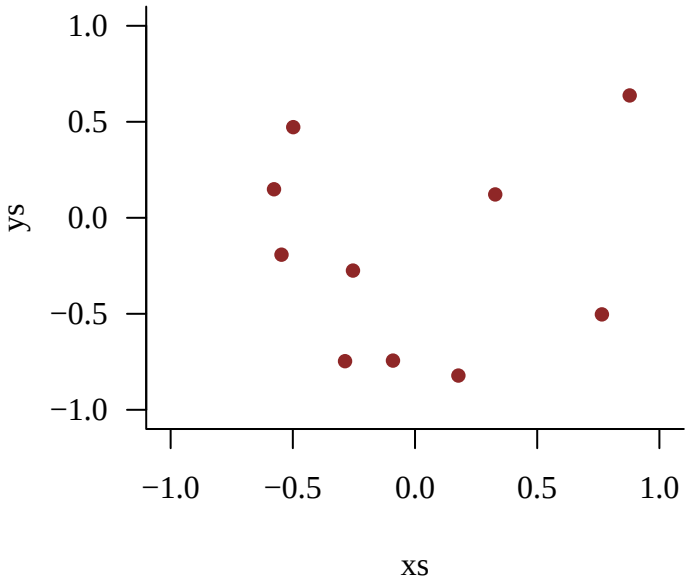
```
      # Close the perimeter
      perimeter_xs <- c(perimeter_xs, he$origin$pos[1])
      perimeter_ys <- c(perimeter_ys, he$origin$pos[2])
      break
    }
  }

  polygon(perimeter_xs, perimeter_ys, col=col, border=NULL)
}
```

5.4 Running Fortune's Algorithm

Alright, let's cut the pedagogical tension and finally build a Voronoi diagram.

We start by defining a collection of sites.

```
N <- 10

set.seed(54482248)
xs <- runif(N, -1, 1)
ys <- runif(N, -1, 1)


par(mfrow=c(1, 1), mar=c(5, 5, 4, 1))
plot(xs, ys, col=c_dark, pch=16,
     xlim=c(-1.1, 1.1), ylim=c(-1.1, 1.1))
```



Next we initialize the event queue and beach line tree.

```
queue <- event_queue$new(1:N, xs)

bl <- beachline$new()
voronoi <- dcel$new()

cat('Initial queue:\n')
```

```
Initial queue:
```

```
for (i in queue$idxs) {
  arc <- queue$events[[i]]$arc
  if (is.null(queue$events[[i]]$arc)) {
    cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x), ':',
              'Site Event at', queue$events[[i]]$site), '\n')
  } else {
    cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x), ':',
              'Vertex Event at', queue$events[[i]]$arc$site), '\n')
  }
}
```

```
  S = -0.577 : Site Event at 6
  S = -0.546 : Site Event at 2
  S = -0.498 : Site Event at 1
```

```
  S = 0.177 : Site Event at 9
  S = -0.090 : Site Event at 10
  S = -0.286 : Site Event at 3
  S = -0.254 : Site Event at 7
  S = 0.328 : Site Event at 8
  S = 0.765 : Site Event at 4
  S = 0.878 : Site Event at 5
```

We can completely exhaust the event queue by running.

```
while (queue$is_not_empty()) {
  bl$process_next_event(queue, voronoi)
}
```

Here, however, we'll proceed a little bit more deliberately. This function processes the next event with extreme verbosity, communicating the type of the event, visualizing both the current beach line tree and the current beach line, and then displaying the status of the event queue.

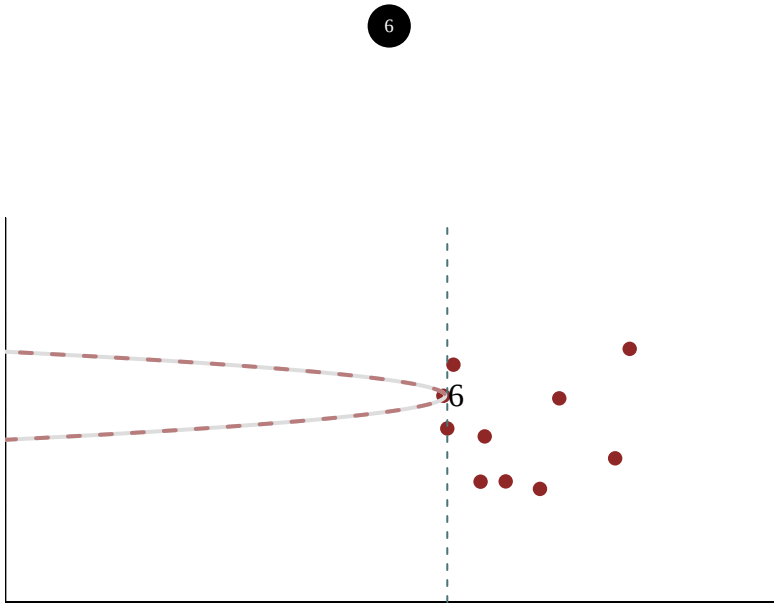
```
process_next_event <- function() {
  if (is.null(queue$next_event()$arc)) {
    cat(paste0('Processing Site Event (',
               queue$next_event()$site, ')\n'))
  } else {
    cat(paste('Processing Vertex Event (',
              queue$next_event()$arc$site, ')\n'))
  }

  bl$process_next_event(queue, voronoi)

  par(mfrow=c(2, 1))
  plot_tree(bl, text_cex=0.6)
  if (is.null(queue$next_event()$x)) {
    plot_beachline(bl, 2,
                   c(-4, 2), c(-2, 2))
    plot_beachline_half_edges(bl, S, col=c_dark_teal,
                              delta=0.08, eps=0.04, gamma=0.02)
    for (v in voronoi$vertices)
      plot_vertex_edges(v, ucol=c_light_teal,
                        delta=0.08, eps=0.04, gamma=0.02)
  } else {
    plot_beachline(bl, queue$next_event()$x,
                   c(-4, 2), c(-2, 2))
```

```
    plot_beachline_half_edges(bl, S, col=c_dark_teal,
                              delta=0.08, eps=0.04, gamma=0.02)
    for (v in voronoi$vertices)
      plot_vertex_edges(v, ucol=c_light_teal,
                        delta=0.08, eps=0.04, gamma=0.02)
  }

  cat('\n')
  cat('Remaining queue:\n')
  for (i in queue$idxs) {
    arc <- queue$events[[i]]$arc
    if (is.null(queue$events[[i]]$arc)) {
      cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x),
                ':',
                'Site Event at', queue$events[[i]]$site),
                '\n')
    } else {
      cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x),
                ':',
                'Vertex Event at', queue$events[[i]]$arc$site),
                '\n')
    }
  }
}
```

Initially the beach line is empty. The first site event populates the beach line with a single parabola.

```
process_next_event()
```

```
Processing Site Event (6)
```

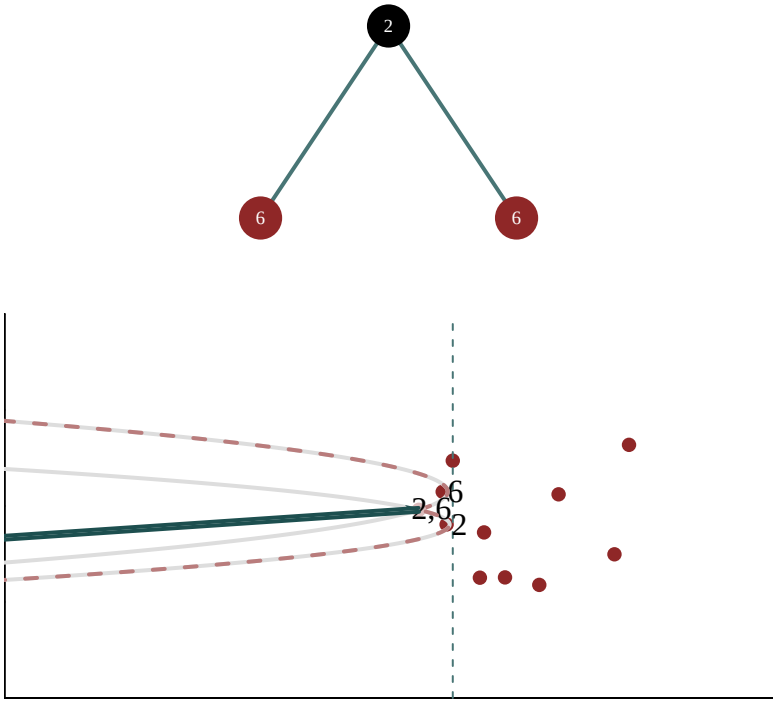



```
Remaining queue:
  S = -0.546 : Site Event at 2
  S = -0.090 : Site Event at 10
  S = -0.498 : Site Event at 1
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = -0.286 : Site Event at 3
  S = -0.254 : Site Event at 7
  S = 0.328 : Site Event at 8
  S = 0.765 : Site Event at 4
```

The next site event splits this monolithic parabola up into three arcs and introduces the first Voronoi half-edges. Note that the lower arc is too small to be seen in this plot.

```
process_next_event()
```

```
Processing Site Event (2)
```

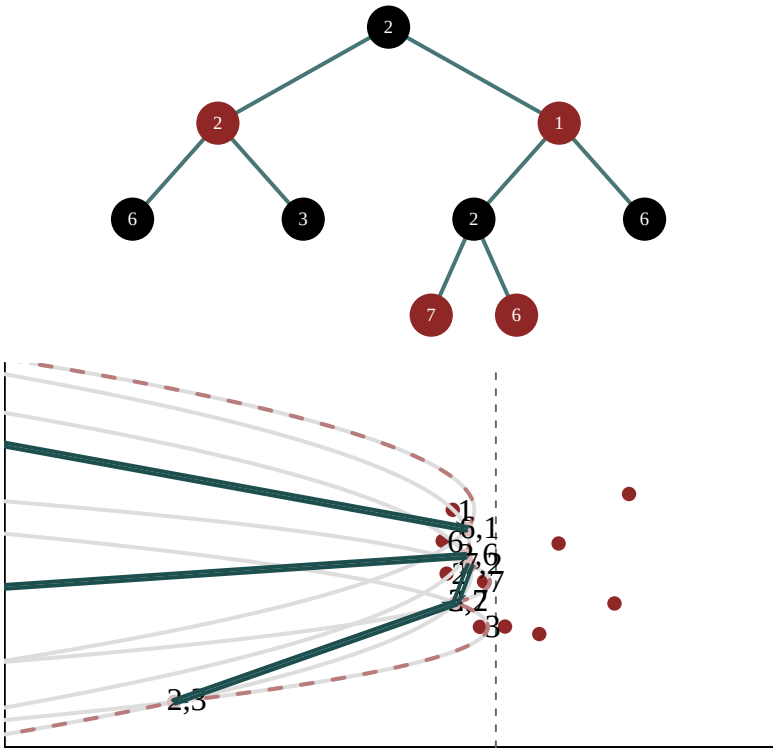



```
Remaining queue:
  S = -0.498 : Site Event at 1
  S = -0.090 : Site Event at 10
  S = -0.286 : Site Event at 3
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = 0.765 : Site Event at 4
  S = -0.254 : Site Event at 7
  S = 0.328 : Site Event at 8
```

At this point we'll jump through the next few events until we're about to encounter our first vertex event.

```
for (i in 1:3) {
  bl$process_next_event(queue, voronoi)
}

par(mfrow=c(2, 1))

plot_tree(bl, text_cex=0.6)
```

```
plot_beachline(bl, queue$next_event()$x,
               c(-4, 2), c(-2, 2))
plot_beachline_half_edges(bl, S, col=c_dark_teal,
                          delta=0.08, eps=0.04, gamma=0.02)
```




```
for (v in voronoi$vertices)
  plot_vertex_edges(v, ucol=c_light_teal,
                    delta=0.08, eps=0.04, gamma=0.02)

cat('')
cat('Remaining queue:')
```

```
Remaining queue:
```

```
for (i in queue$idxs) {
  arc <- queue$events[[i]]$arc
  if (is.null(queue$events[[i]]$arc)) {
    cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x),
              ':',
```

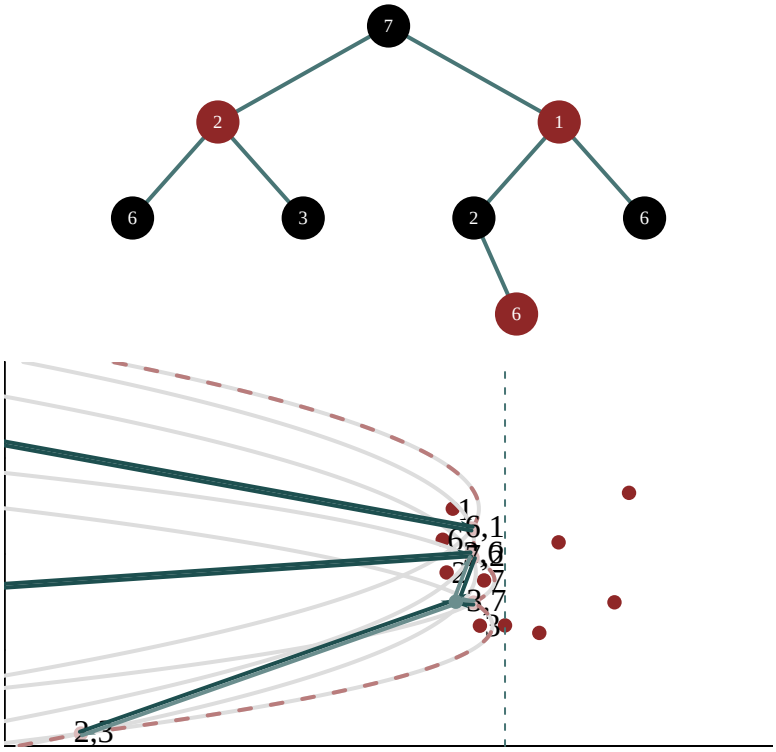
```
              'Site Event at', queue$events[[i]]$site),
              '\n')
  } else {
    cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x),
              ':',
              'Vertex Event at', queue$events[[i]]$arc$site),
              '\n')
  }
}
```

```
  S = -0.162 : Vertex Event at 2
  S = -0.049 : Vertex Event at 2
  S = -0.090 : Site Event at 10
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = 0.765 : Site Event at 4
  S = 0.328 : Site Event at 8
  S = 1.487 : Vertex Event at 6
```

In the first vertex event we start paring the beach line down. Here the arc node generated by site 2 that sits between the breakpoints b_{32} and b_{27} is removed while our first Voronoi vertex is created.

```
process_next_event()
```

```
Processing Vertex Event ( 2 )
```



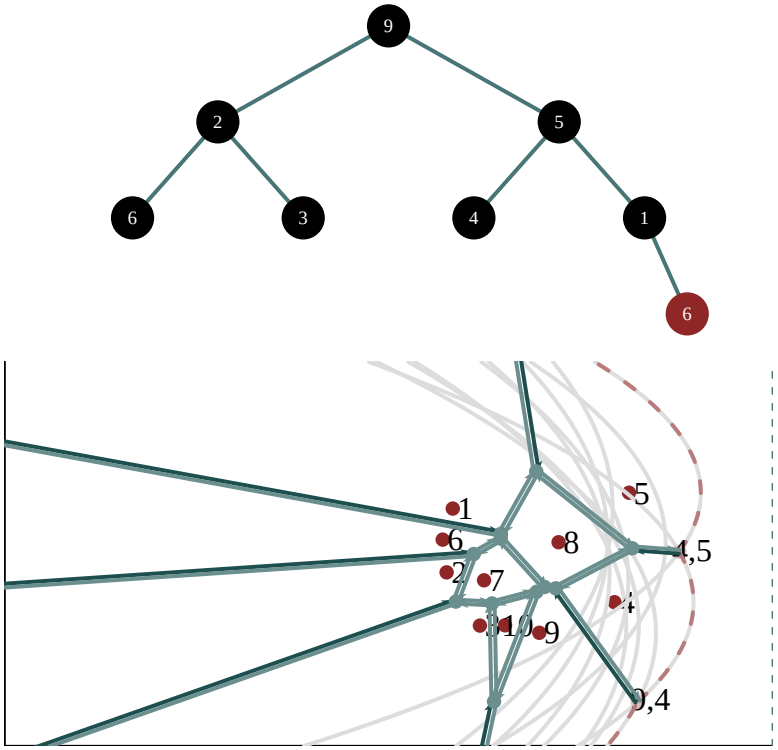

```
Remaining queue:
  S = -0.090 : Site Event at 10
  S = -0.049 : Vertex Event at 2
  S = 0.328 : Site Event at 8
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = 0.765 : Site Event at 4
  S = 1.487 : Vertex Event at 6
```

To not draw this process out *too* long let's run Fortune's algorithm to completion.

```
while (queue$is_not_empty()) {
  bl$process_next_event(queue, voronoi)
}
```

The remaining breakpoint nodes each contain a dangling Voronoi half-edge.

```
par(mfrow=c(2, 1))
plot_tree(bl, text_cex=0.6)

plot_beachline(bl, 2,
               c(-4, 2), c(-2, 2))
plot_beachline_half_edges(bl, S, col=c_dark_teal,
                          delta=0.08, eps=0.04, gamma=0.02)
for (v in voronoi$vertices)
  plot_vertex_edges(v, ucol=c_light_teal,
                    delta=0.08, eps=0.04, gamma=0.02)
```



To visualize the entire Voronoi graph within the confines of a finite plot we'll need a bounding box.

We can always specify a bounding box of our own. This can be useful, for example, if we want particular large or symmetric bounding boxes for visualization purposes.

```
voronoi$xlim <- c(-3, 3)
voronoi$ylim <- c(-3, 3)
```

In most cases, however, it's most convenient to dynamically adapt a bounding box to the Voronoi vertices.

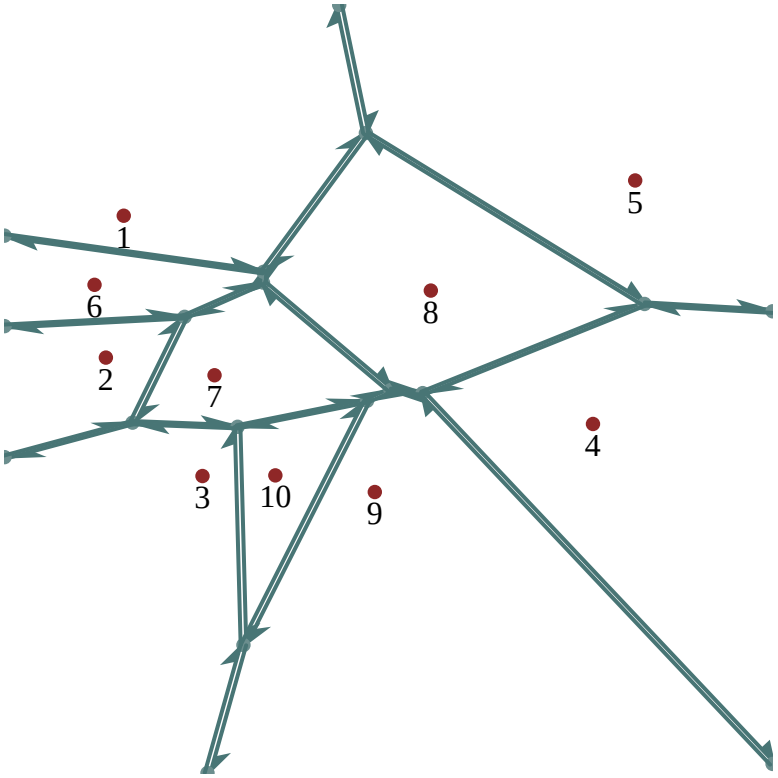
```
voronoi$compute_bounding_box(0.25)
```

Given a bounding box geometry we can anchor dangling half-edges to pseudo-vertices on the bounding box.

```
voronoi$anchor_dangling_edges(bl$root, xs, ys)
```

Perhaps the simplest way to plot the Voronoi graph is to plot each Voronoi vertex along with the half-edges originating from and terminating at those vertices.

```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))
plot(xs, ys, axes=FALSE, ann=FALSE, col=c_dark, pch=16,
     xlim=voronoi$xlim, ylim=voronoi$ylim,
     main="")

for (n in 1:N)
  text(xs[n], ys[n] - 0.1, n)

for (v in voronoi$vertices)
  plot_vertex_edges(v, delta=0.08, eps=0.04, gamma=0.0075)

for (pv in voronoi$pseudo_vertices)
  plot_vertex_edges(pv, delta=0.08, eps=0.04, gamma=0.0075)
```

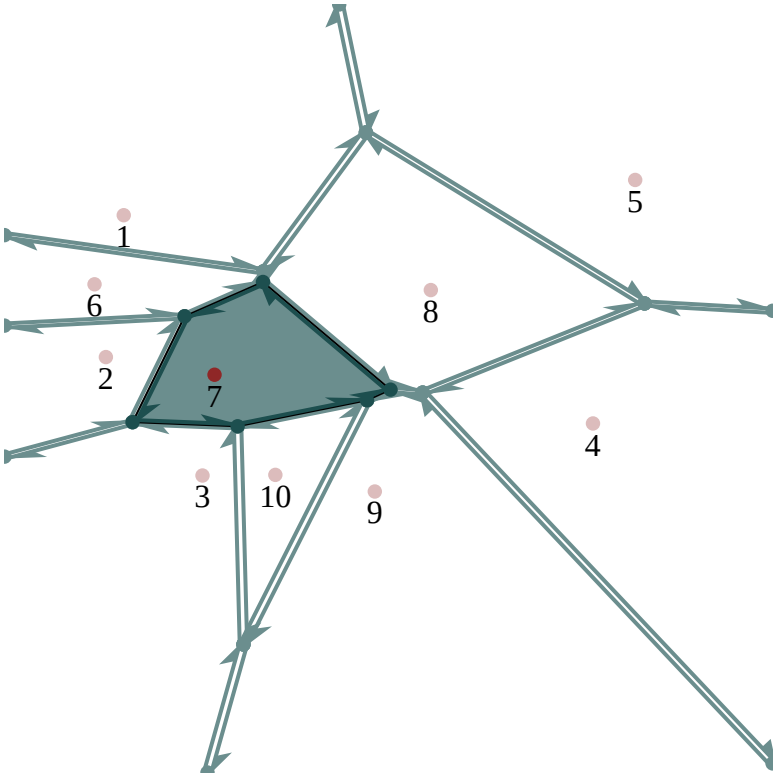


The real content of the Voronoi diagram, however, is in the faces.

```
voronoi$find_faces()
```

```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))
plot(xs, ys, axes=FALSE, ann=FALSE, col=c_light, pch=16,
     xlim=voronoi$xlim, ylim=voronoi$ylim,
     main="")

for (n in 1:N)
  text(xs[n], ys[n] - 0.1, n)

for (f in voronoi$faces) {
  plot_face_boundary(f, col=c_light_teal,
                     delta=0.08, eps=0.04, gamma=0.01)
}

f <- voronoi$faces[[1]]
plot_face(f, col=c_light_teal)
plot_face_boundary(f, col=c_dark_teal,
```

```
                    delta=0.08, eps=0.04, gamma=0.01)
points(xs[f$site], ys[f$site], col=c_dark, pch=16)
text(xs[f$site], ys[f$site] - 0.1, f$site)
```



The beautiful thing about Fortune's algorithm is that scaling up to more sights is not particularly expensive.

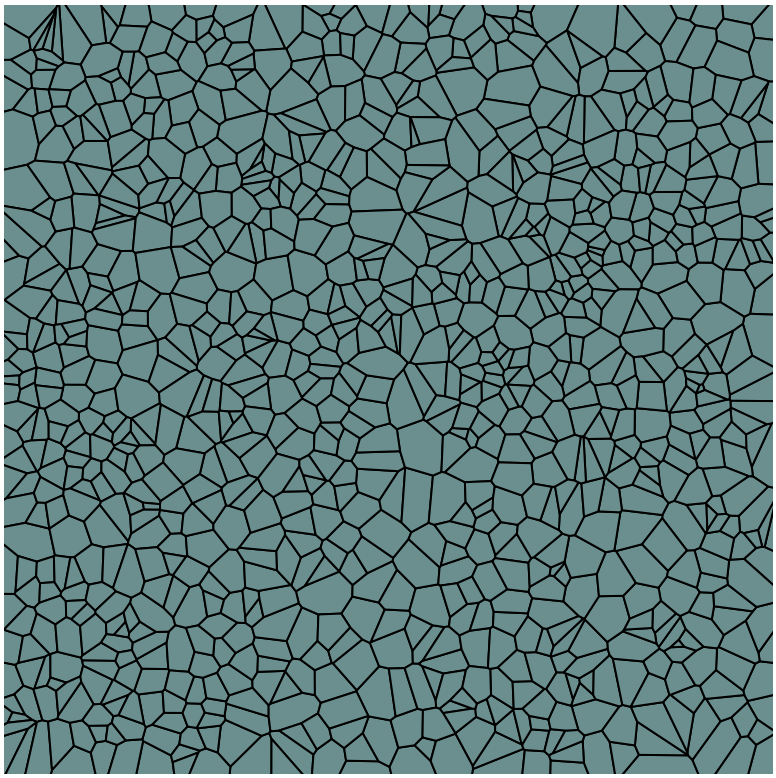
```
N <- 1000

xs <- runif(N, -1, 1)
ys <- runif(N, -1, 1)


queue <- event_queue$new(1:N, xs)

bl <- beachline$new()
voronoi <- dcel$new()


while (queue$is_not_empty()) {
```

```
  bl$process_next_event(queue, voronoi)
}
```

```
voronoi$compute_bounding_box(0.25)

voronoi$anchor_dangling_edges(bl$root, xs, ys)
```

```
voronoi$find_faces()
```

```
par(mfrow=c(1, 1), mar=c(0, 0, 0, 0))

plot(xs, ys, col=c_light, pch=16,
     xlim=c(-1, 1), ylim=c(-1, 1),
     main="")

for (f in voronoi$faces) {
  plot_face(f, col=c_light_teal)
}
```



6 Conclusion

Voronoi diagrams are straightforward to define but subtle to construct in practice, especially if we want to maximize computational efficiency. Fortune's algorithm systematizes the construction of Voronoi diagrams, but understanding how the algorithm actually works requires immersing ourselves in planar geometry. Moreover, effectively implementing Fortune's algorithm in practice requires familiarity with a variety of nontrivial data structures.

From a more optimistic perspective, Fortune's algorithm provides a wealth of learning opportunities…

Acknowledgements

A very special thanks to everyone supporting me on Patreon: Adam Fleischhacker, Alejandro Morales, Alessandro Varacca, Alex D, Alexander Noll, Amit, Andrea Serafino, Andrew Mascioli, Andrew Rouillard, Ara Winter, Ari Holtzman, Austin Rochford, Aviv Keshet, Avraham Adler, Ben Matthews, Ben Swallow, Benoit Essiambre, boot, Brendan Galdo, Bryan Chang, Brynjolfur Gauti Jónsson, Cameron Smith, Canaan Breiss, Cat Shark, Cathy Oliveri, Charles Naylor, Chase Dwelle, Chris Jones, Christina Van Heer, Christopher Mehrvarzi, Colin Carroll, Colin McAuliffe, Damien Mannion, dan mackinlay, Dan W Joyce, Dan Waxman, Dan Weitzenfeld, Daniel Edward Marthaler, Danny Van Nest, David Burdelski, Doug Rivers, Dr. Jobo, Dr. Omri Har Shemesh, Dylan Maher, Dylan Spielman, Ed Cashin, Edgar Merkle, Edoardo Marcora, Eric LaMotte, Erik Banek, Eugene O'Friel, Felipe González, Fergus Chadwick, Finn Lindgren, Francesco Corona, Geoff Rollins, Granville Matheson, Gregor Gorjanc, Guilherme Marthe, Håkan Johansson, Hamed Bastan-Hagh, haubur, Hector Munoz, Henri Wallen, hs, Hugo Botha, Ian Costley, idontgetoutmuch, Ignacio Vera, Ilaria Prosdocimi, Isaac Vock, Isidor Belic, jacob pine, Jair Andrade, James C, James Hodgson, James Wade, Janek Berger, Jarrett Byrnes, Jason Martin, Jason Pekos, Jason Wong, jd, Jed , Jeff Burnett, Jeff Dotson, Jeff Helzner, Jeffrey Erlich, Jerry Lin , Jessica Graves, Joe Sloan, John Flournoy, Jonathan H. Morgan, Josh Knecht, JU, Julian Lee, June, Justin Bois, Kádár András, Karim Naguib, Karim Osman, Konstantin Shakhbazov, Kristian Gårdhus Wichmann, Lars Barquist, lizzie , LOU ODETTE, Luís F, Mads Christian Hansen, Marek Kwiatkowski, Mariana Carmona, Mark Donoghoe, Markus P., Márton Vaitkus, Matěj, Matthew, Matthew Kay, Matthieu LEROY, Mattia Arsendi, Maurits van der Meer, Max, Michael Colaresi, Michael DeWitt, Michael Dillon, Michael Lerner, Mick Cooney, MisterMentat , N Sanders, N.S. , Name, Nathaniel Burbank, Nicholas Cowie, Nick S, Nikita Karetnikov, Octavio Medina, Ole Rogeberg, Oliver Crook, Olivier Ma, Patrick Kelley, Patrick Boehnke, Pau Pereira Batlle, Peter Johnson, Pieter van den Berg , ptr, quasar, Ramiro Barrantes Reynolds, Raúl Peralta Lozada, Ravin Kumar, Rex, Riccardo Fusaroli, Richard Nerland, Robert Frost, Robert Goldman, Robert kohn, Robin Taylor, Ryan Gan, Ryan Grossman, S Hong, Sean Wilson, Sergiy Protsiv, Seth Axen, shira, Simon Duane, Simon Lilburn, Simon Steiger, Simone, Spencer Boucher, sssz, Stefan Lorenz, Stephen

Lienhard, Steve Harris, Steven Forrest, Stew Watts, Stone Chen, Susan Holmes, Svilup, Tate Tunstall, Tatsuo Okubo, Teresa Ortiz, Theodore Dasher, Thomas Siegert, Thomas Vladeck, Tobychev , Tomas Capretto, Tony Wuersch, Virgile Andreani, Virginia Fisher, Vitalie Spinu, Vladimir Markov, VO2 Maximus Decius, Will Farr, Will Lowe, Will Wen, woejozney, yolhaj , yureq , Zach A, and Zhengchen Cai.

References

Berg, Mark de, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry*. Third. Springer-Verlag, Berlin.

Chang, Winston. 2025. *R6: Encapsulated Classes with Reference Semantics*.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. Fourth. Cambridge, MA: MIT Press.

Fortune, Steven. 1987. "A Sweepline Algorithm for Voronoĭ Diagrams." *Algorithmica* 2 (2): 153–74.

License

A repository containing all of the files used to generate this chapter is available on GitHub.

Original Computing Environment

```
sessionInfo()
```

```
R version 4.3.2 (2023-10-31)
Platform: x86_64-apple-darwin20 (64-bit)
Running under: macOS 15.6.1

Matrix products: default
```

```
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-x86_64/Resources/lib/libRlapack.dylib;

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/New_York
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] R6_2.6.1

loaded via a namespace (and not attached):
 [1] compiler_4.3.2  fastmap_1.1.1   cli_3.6.2       tools_4.3.2
 [5] htmltools_0.5.7 yaml_2.3.8      rmarkdown_2.25  knitr_1.45
 [9] jsonlite_1.8.8  xfun_0.41       digest_0.6.33   rlang_1.1.2
[13] evaluate_0.23
```