Voronoi To The World

Michael Betancourt

November 2025

Table of contents

Voronoi diagrams arise naturally in many 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 we'll work through the geometric structure of Voronoi diagrams and an algorithm that leverages that structure to implement 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 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 sometimes referred to as "the 4Ms" which I can only infer refers to three out of the four authors having 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.

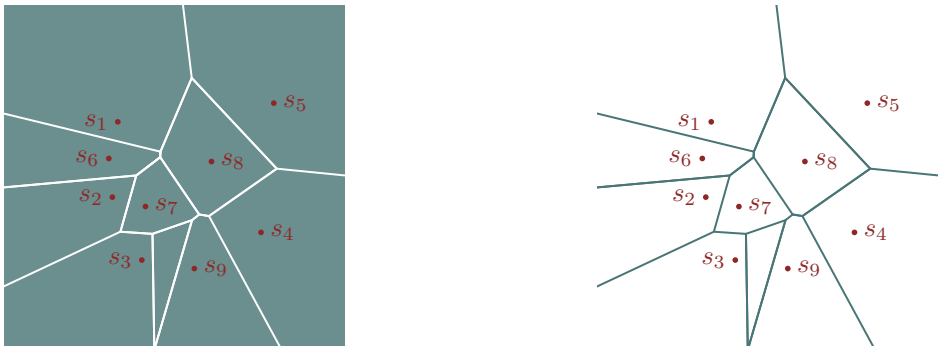
To define a Voronoi diagram formally consider a two-dimensional Euclidean plane $\mathbb{R}^2$ and a collection of **sites**

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

Give a particular site s_n we can construct a subset of points, also known as a **cell**, 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. The Voronoi diagram of a collections of sites is the corresponding collection of nearest-proximity 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.

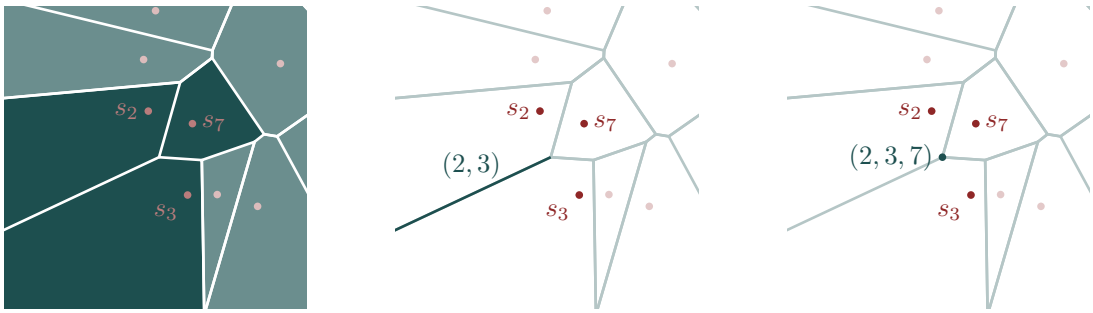
The standard Voronoi diagram considers the Euclidean distance function,

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

This construction, however, 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 metric spaces.

A Voronoi diagram *almost* forms partition of a Euclidean plane. The obstruction 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 punctual boundaries separating the corners of three neighboring cells by the sites associated with those cells (Figure 2b, Figure 2b).



(a) (b) (c)

Figure 2: (a) Each cell comprising 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, so $(3, 2)$ 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.

Often it is easier in practice 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 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,

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

That said, not every triplet of Voronoi cells are neighbors and not every triplet of cells defines a valid vertex. Fortunately Euclidean geometry provides tools for identifying all of the valid vertices.

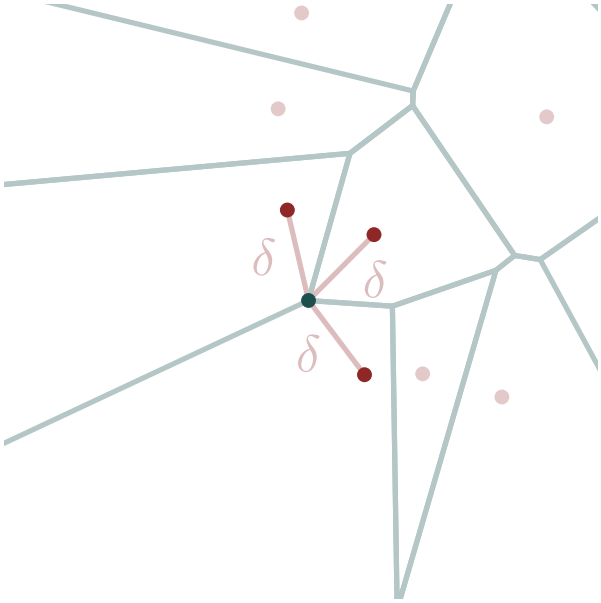


Figure 3: Each vertex in a Voronoi graph is given by a point on the boundary between three neighboring Voronoi cells. This point is by definition equally distant to the three sites contained by those cells.

Each triplet of sites 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

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

By construction the center of the circumcircle is equidistant from all three sites, and hence defines a potential vertex in the Voronoi graph.

The three sites defining the 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 the Voronoi graph. (a) Vertices are defined by triplets of sites that define empty circumcircless. (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 how to identify the points that define 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

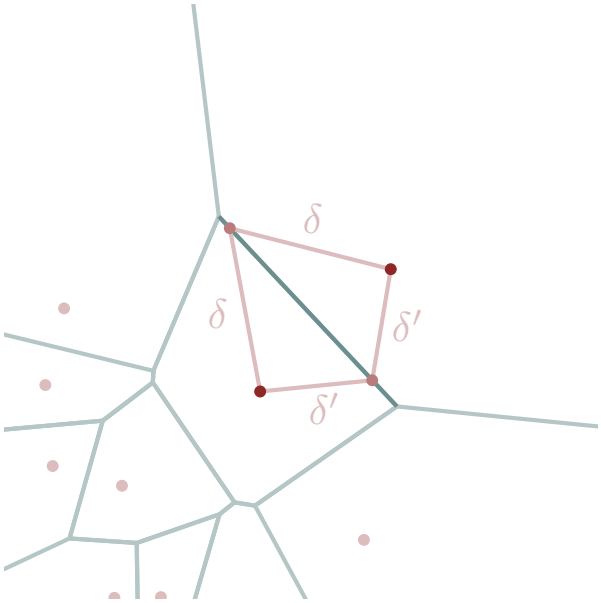
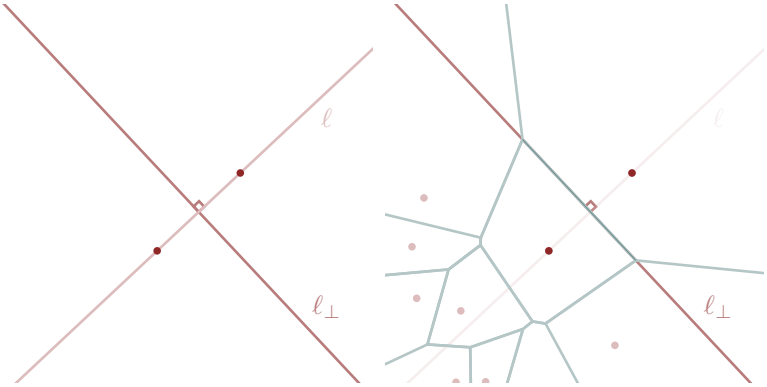


Figure 5: Each edge in a Voronoi graph is defined by linear boundary between two neighboring Voronoi cells. All of the points on a Voronoi edge is equally distant to the two sites contained by those cells.

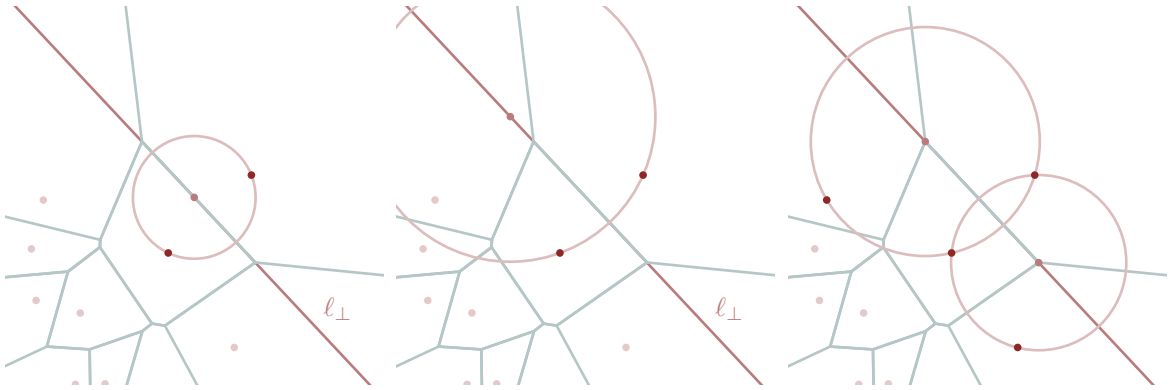


(a) (b)

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 not every perpendicular bisector contains a Voronoi edge and not every point of a perpendicular bisector that does contain an edge is part of that edge.

those two sites are the nearest neighbors to each other and that point is 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 is 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.



(a) (b) (c)

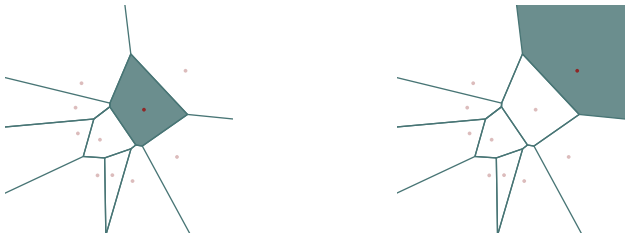
Figure 7: Circles centered at points along a perpendicular bisector and intersecting with a pair of sites are useful for identifying contributions to Voronoi edges. (a) If there are no other sites on or inside of the circle then the center contributes to a Voronoi edge. (b) If there is one or more sites 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 fully 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.

3 Fortune's Algorithm

There are here are multiple ways to construct a Voronoi graph, most of them suffer from wasteful computation. For large enough numbers of sites these approaches quickly become



(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.

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 efficient solutions 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 scanning across the points between the two Voronoi vertices associated with those cites. Next we could take any point along each candidate edge, construct a circle centered on that point and intersecting with the two associated sites, and check if any other sites are inside. The candidates with empty circles define Voronoi edges.

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.

That said, much of this cost is wasted on the repeated querying 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 half-planes defined 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 requires only quadratic number of operations.

We can, however, do even better. By systematically scanning across a plane **Fortune's algorithm** Berg et al. (2008) is able construct 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 systematically 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. I will denote the horizontal position of the scan at any time by S. I will also refer to any site to the left of the sweep line, $x_n < S$ as active.

3.3 The Beach Line

A sweep line partitions the ambient plane into two pieces, allowing the construction of the Voronoi graph to focus on just a subset of the entire plane. 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 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) on 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 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.

When determining geometric features such as the nearest sites 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 a rotated parabola (Figure 9)

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

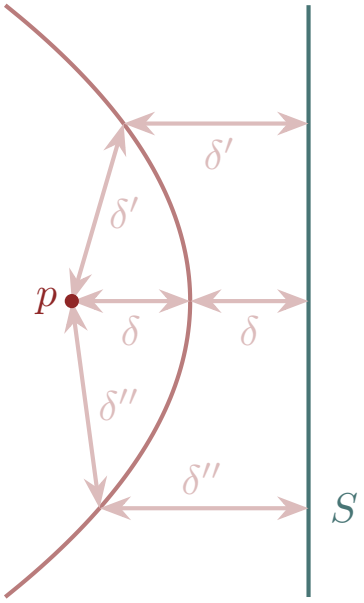


Figure 9: The subset of points that are equally distant from a focus, s, and a vertical line S, form a rotated parabola. In Fortune's algorithm the points interior to this parabola are safe to analyze for their contribution to the Voronoi graph.

When there are multiple active sites then the region that we can safely analyze the points that are closer to *any* of the active sites than 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 each site and the sweep line, however, can contribute to the beach line multiple times.

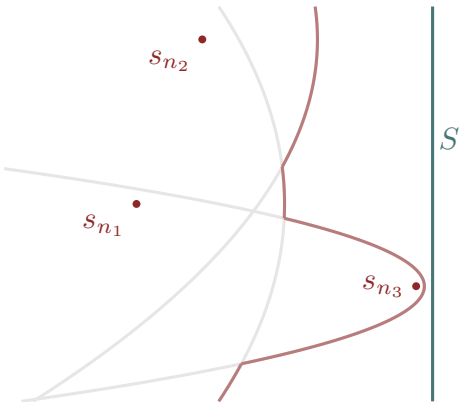


Figure 10: The beach line is the boundary separating points that are closer to any active site, here s_{n_1}, s_{n_2}, and s_{n_3}, and points that are closer to the the sweep line, S, is comprised of multiple parabolic arcs. Note that the parabola between s_{n_1} and S contributes multiple arcs to the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

Consequently we cannot use the originating sites to *uniquely* label the arcs.

By construction neighboring arcs on the beach line are derived 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. 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 configuration we can readily calculate the intersection of each pair of neighboring parabolas and hence the position of any given breakpoint. Formally if the lower arc is a segment of the parabola generated by s_{n_1} and the higher arc is a segment of the parabola generated by s_{n_2} then the vertical position of the breakpoint between those arcs is given by one of the two solutions to the quadratic equation

$$\begin{aligned} 0 = & \left(x_{n_2} - x_{n_1} \right) y_{n_1 n_2}^2 \\ & + \left(y_{n_2} \left(S - x_{n_1} \right) - y_{n_1} \left(S - x_{n_2} \right) \right) y_{n_1 n_2} \\ & + \left(y_{n_2}^2 \left(S - x_{n_1} \right) - y_{n_1}^2 \left(S - x_{n_2} \right) - \left(S - x_{n_1} \right) \left(S - x_{n_2} \right) \left(x_{n_2} - x_{n_1} \right). \right) \end{aligned}$$

The correct solution is the one that falls in between the height of the two neighboring sites,

$$y_{n_1} < y_{n_1 n_2} < y_{n_2}.$$

When it comes to constructing Voronoi graphs the breakpoints are the most important part of the beach line. Because each breakpoint falls onto the parabolas of two neighboring sites,

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.

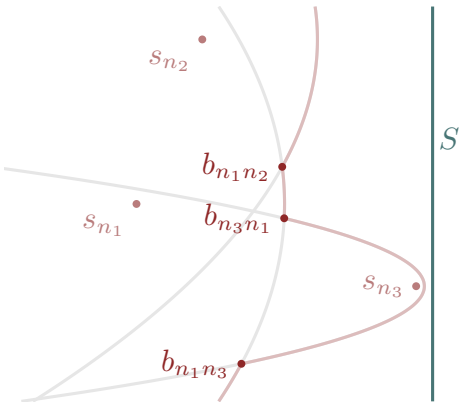


Figure 12: The breakpoints in a beach line separate the component arcs. Each breakpoint is located at the intersection between the parabolas defined by two neighboring sites, equally distant from those two sites and further from any other sites. We can use an ordered pair of these sites to uniquely label each breakpoint.

it is equally distant from those two sites and the sweep line. At the same time no other active site is closer than those two sites, otherwise the parabolas generated by them would not define neighboring arcs on the beach line. In addition every inactive site is also further because they're all beyond the sweep line.

All of this is to say that, by construction, each breakpoint is the center of a circle that intersects with the two associated sites but includes no other sites. Consequently each breakpoint *is always a part of a Voronoi edge*.

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 change. This in turn changes the shape of the corresponding parabolas, and hence the shape of the beach line. As the beach line evolves in this way the breakpoints trace out the edges of the Voronoi diagram.

Every now and then the number of arcs comprising the beach line, referred to 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. All of the information we need 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 in advance we at exactly which positions these **site events** will occur.

If the sweep line falls exactly on a new site then the parabola of points equidistant from the 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 13a). Although not *technically* correct, these singular arcs are often visualized with a horizontal line extending from the new site to the beach line (Figure 13b).

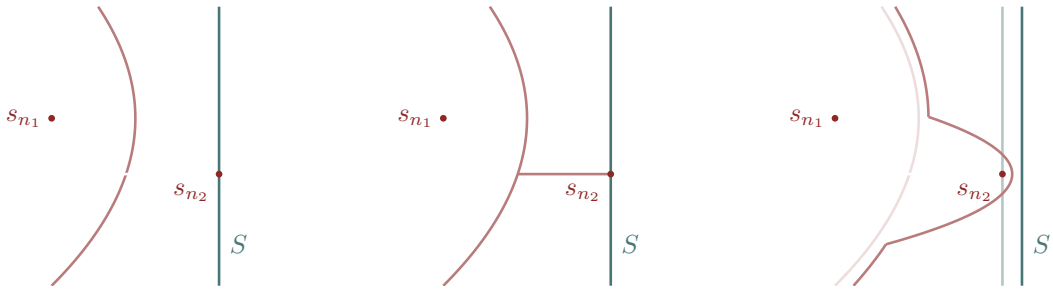


Figure 13: At a site event the sweep line passes an inactive site which then becomes active. (a) Initially the new active site introduces a new singular arc which forces a discontinuity in the beach line. (b) This singular arc is often visualized with a straight line. (c) As the sweep line proceeds past the site event the singular arc widens into a well-behaved parabola, splitting one of the initial beach line arcs into a triplet of arcs.

As the sweep line progresses past the new active site the singular parabola widens into a more well-behaved parabola (Figure 13c). 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.

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 separated by two new breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_1}$.

Both of the breakpoints fall onto 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 $b_{n_1 n_2}$ and $b_{n_2 n_1}$ separate, tracing out more and more of the edge (Figure 14)

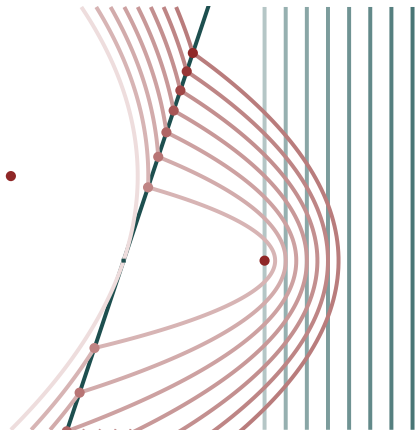


Figure 14: As the sweep line moves past a site event the two new breakpoints separate and scan across the Voronoi edge between the two neighboring sites.

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 15). In this case the collapsed arc is removed from the beach line and its two neighbors become neighbors themselves.

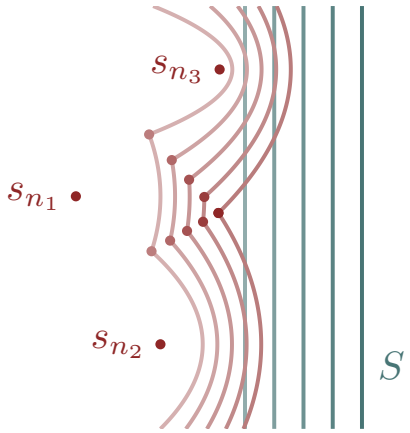


Figure 15: As the sweep line progresses arcs on the beach line can collapse in between their two neighboring arcs, with the neighboring breakpoints converging to a point. This point is equally distant from the three active sites and further from all other sites, defining a vertex in the Voronoi graph.

Now an arc in between two arcs that are generated from the same active site, such as the central arc introduced when the sweep line crosses a site event, always widens faster than its neighboring arcs as the sweep line progresses. Consequently a triplet of arcs generated by

only two sites, $(s_{n_1}, s_{n_2}, s_{n_1})$, will never collapse. Rather collapsing arcs are *always* defined by triplets of arcs generated by three distinct sites, $(s_{n_1}, s_{n_2}, s_{n_3})$.

When the central arc collapses its two neighboring breakpoints $b_{n_1 n_2}$ and $b_{n_2 n_3}$ will intersect. At that moment the breakpoints are equally distant from the sweep line and all tree of the associated sites, s_{n_1}, s_{n_2}, and s_{n_3} . Moreover, because this intersection is on the beach line it cannot be closer to any in active 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. At this point it should not be a surprise that this prediction will involve circles.

For any triplet of neighboring arcs on a beach line that correspond to the *distinct* sites $(s_{n_1}, s_{n_2}, s_{n_3})$ 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. The center is also equally distant to vertical lines at $x = x_c - r$ and $x = x_c + r$. Consequently if the central arc collapses then it will do so when the sweep line is at either $S = x_c - r$ or $S = x_c + r$ (Figure 16).

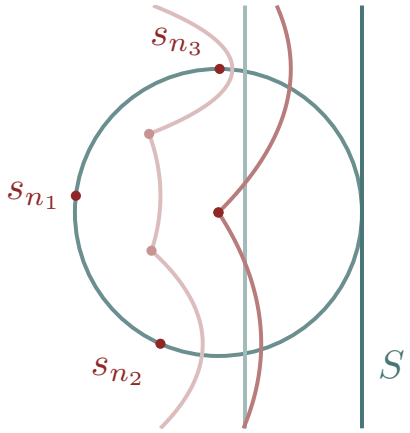


Figure 16: If the central arc in a triplet of neighboring arcs generated by distinct sites will collapse then it will do so when the sweep line is at the far edge of the circumcircle spanned by those tree sites. This geometric extrapolation allows us to schedule future vertex events.

A collapse at $S = x_c - r$ can't contribute information about the Voronoi graph because it is ahead of the current beach line where the structure of the Voronoi graph has already been established. If the central arc collapses at this point then it will *expand* as the sweep line progresses further.

Instead we can focus on a new **vertex event**, also known as a **circle event**, at $S = x_c + r$. If the the central arc generated by s_{n_2} collapses when the sweep line reaches this position then

$b_{n_1 n_2}$ and $b_{n_2 n_3}$ will intersect at (x_c, y_c). In turn this intersection defines a new vertex that we can add to the Voronoi graph (Figure 17).

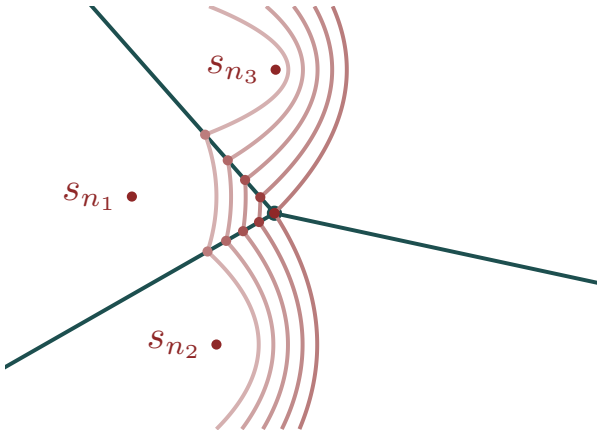


Figure 17: When a central arc collapses two Voronoi edges terminate at a Voronoi edge and a new Voronoi edge between the sites that generate the remaining arcs is created.

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 a dynamic 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 the next event in the queue to expand the structure of the Voronoi graph and add new events as necessary. Initially the event queue is filled with only site events, one for each site, and vertex events are 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. The details depend on how we implement Fortune's algorithm, which will be the focus of the next section.

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 at the same height as 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 breakpoints.

Inserting a new arc into the beach line can also modify the remaining event queue. For example inserting a new arc will disrupt some of the triplets of neighboring arcs generated by distinct

sites in the previous beach line. If any of those arc triplets correspond to a future vertex events in the event queue then they have have to be removed.

At the same time inserting a new arc also creates new triplets of neighboring arcs, each of which could potentially spawn a vertex event. After checking that the individual arcs are generated by distinct sites we have to check whether or not the central arc actually collapses beyond the current sweep line. If all of these conditions are met then we add a new vertex event to the event queue where the central arc will be removed.

3.5.2 Vertex Events

When we encounter a vertex event we first have to remove the central arc from the beach line, along with one of its neighboring breakpoints.

At this point we expand the current Voronoi graph. This starts by adding a new Voronoi vertex, connecting it to two of the previously dandling Voronoi edges, and then creating a new Voronoi edge. Exactly how we accomplish this depends on how the graph is implemented.

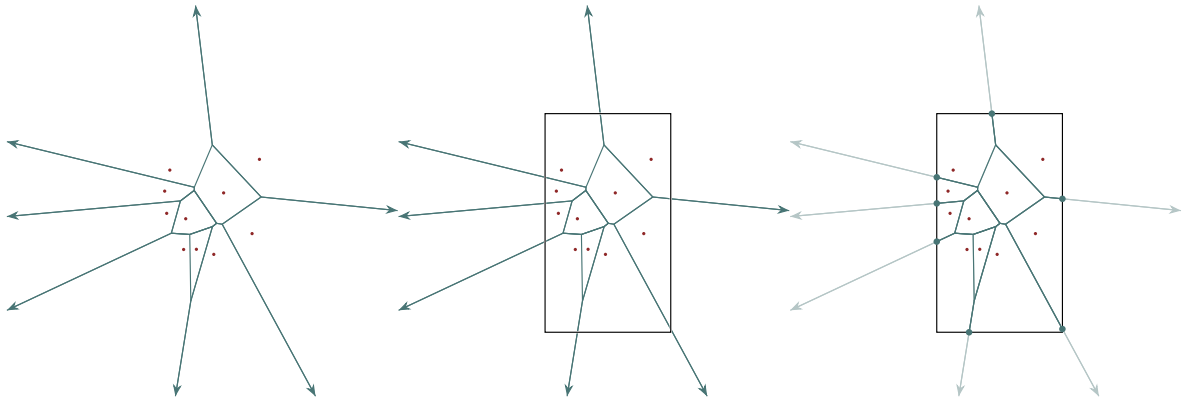
Lastly we have to clean up the event queue. The removal of the central arc, for example, will invalidate any vertex event in the remaining event queue where the central arc was previously neighbors with the removed arc. On the other hand the removal also creates two new triplets of neighboring arcs along the beach line, each of which could spawn a new vertex event. If any of these triplets consist of arcs generated by distinct sites with the central arc collapses beyond the current sweep line then we add a corresponding vertex event to the event queue.

3.5.3 Clean Up

As we process the events the event 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 (Figure 18a).

We can handle these dangling edges in a few different ways. For example from a graph theory perspective we can 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 **bounding box** that contains all of the Voronoi vertices, and ideally all of the sites, (Figure 18b) and then anchor the dandling edges to points along this rectangle (Figure 18c). Specifically the anchor for a dangling edge will be at the point where the perpendicular bisector between the two sites associated with the edge intersect with the bounding box.



(a) (b) (c)

Figure 18: (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 dangles out towards infinity. (b) When visualizing a Voronoi graph we can consider only a finite range of values. This requires limiting the visualization to a bounding box that contains all of the Voronoi vertices. (c) For visualization purposes we can anchor the dangling edges to points where they meet 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 the 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 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 are straightforward to specify 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 basic 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 represent undirected edges directly. Instead the data structure splits undirected edges into two directed edges known as **half-edges** (Figure 19).

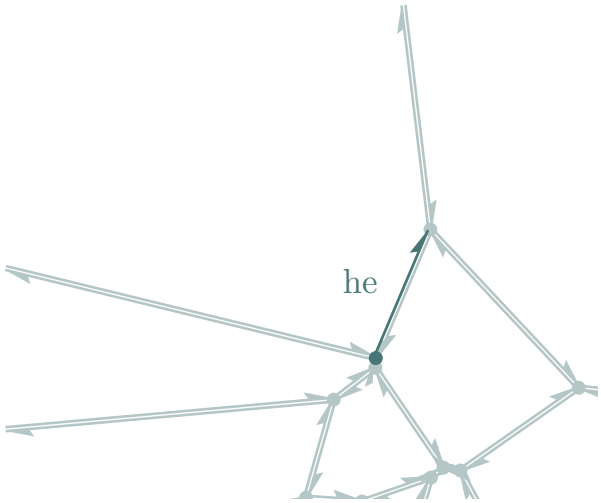


Figure 19: Doubly-connected edge lists represent undirected edges with a pair of directed half-edges. Each half-edge includes references to other structures in the doubly-connected edge lists, including the vertex from which the half-edge originates and neighboring half-edges.

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 20). Finally each half-edge includes a reference to the face located to the left of the half-edge.

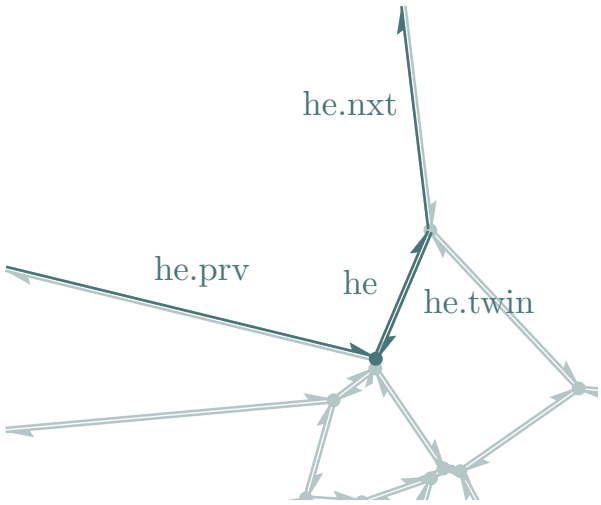


Figure 20: Each half-edge in a doubly-connected edge lists includes reference three other half-edges. The previous half-edge, here labeled "he.prv", identifies the half-edge that points into it while the next half-edge, here labeled "he.nxt", identifies the half-edge that it points into. Finally the twin half-edge, here labeled "he.twin", identifies the half-edge that completes it to form a full undirected edge.

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 and their twins will features empty originating vertices and previous half-edge references.

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 21). The relationship of a vertex to the

rest of graph can be completely reconstructed from this information alone.

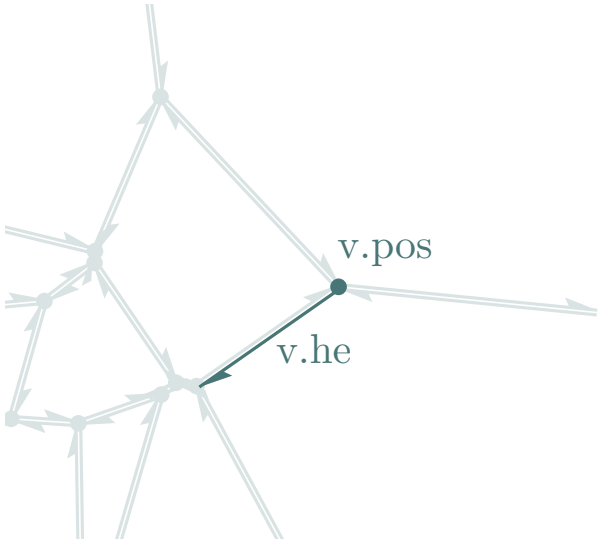


Figure 21: Each vertx in a doubly-connected edge list includes its spatial position, here denoted "v.pos" and a reference to any one half-edge that originates from it, here denoted "v.he".

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 included half-edge (Figure 22). Equivalently we can iteratively follow the previous and then twin references. In the same way we can 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 immediately given by the originating vertex references of the twin of any half-edge that originates at a given vertex.

4.1.3 Faces

Finally each face in the graph is specified by any one half-edge along its boundary (Figure 23a). The provides enough information to trace out the entire boundary by following the next or previous references from that one included half-edge (Figure 23b, Figure 23c).

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

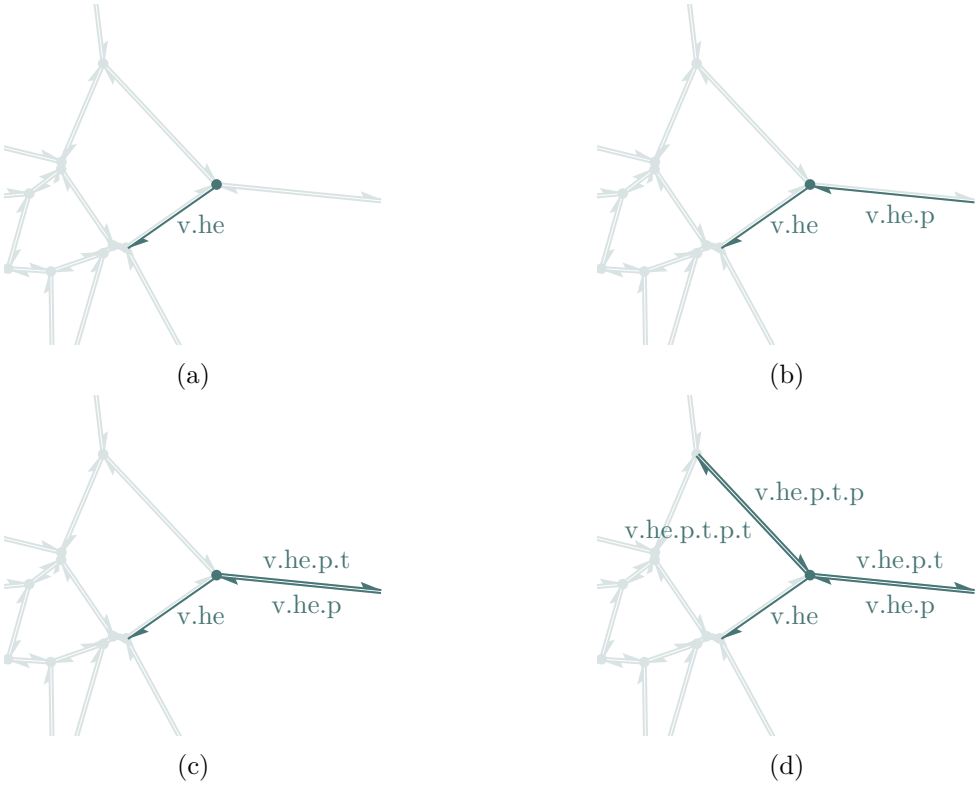
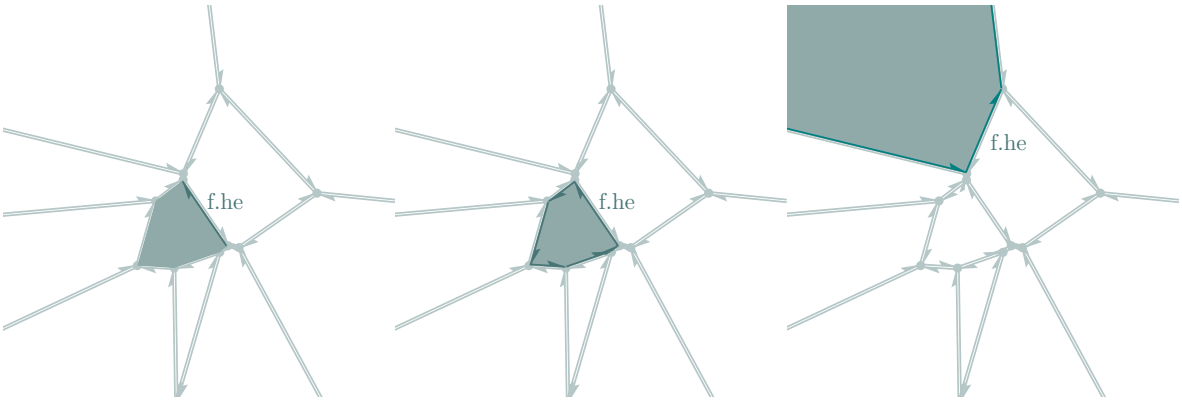


Figure 22: (a) The one half-edge referenced in a vertex object is enough to reconstruct the entire structure of the graph around that vertex. For example if we follow the reference to (b) the previous half-edge, "v.he.p", (c) and then its twin, "v.he.p.t", we recover another half-edge that originates from that vertex. (d) Repeating this operation recovers all of the half-edges that originate from that vertex.



(a) (b) (c)

Figure 23: (a) Each face in a doubly-connected edge list includes a references 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, regardless of whether the face is (b) closed or (c) open.

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 objects and one for each face object. We don't need to store the two sites associated with an edge, as these 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 we will add 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 that every half-edge includes a non-empty originated vertex references which in turns makes visualization functions more uniform.

4.2 Implementing Beach Lines

A beach line consists of parabolic arcs and separated breakpoints, all of which *ordered*. As we work through Fortune's algorithm we'll need to be able to not only add and remove arcs and breakpoints but also efficiently search through these ordered objects. All of these operations

can be implemented with a slightly modified **binary search tree** (Cormen et al. 2022) where each node represents either an arc or a breakpoint (Figure 24).

4.2.1 Arc Nodes

When using a binary search tree to represent a beach line each arc is represented by a non-empty leaf node. I will refer to these as **arc nodes**.

Beyond a reference to their parent nodes, each of these arc nodes also contain a reference to the site that generates the arc. This allows us to, for example, dynamically compute the shape of the arc for any position of the sweep line. Because the parabola generated by a single site can contribute to the beach line multiple times, multiple arc nodes can share the same site reference.

To avoid having to search through the event queue we also include in each arc node a reference to any vertex events that centered on that arc. This allows us to, for example, quickly find events that need to be removed from the event queue.

4.2.2 Breakpoint Nodes

The internal nodes of a binary search tree represent the breakpoints between the arcs in a beach line. I will unsurprisingly refer to these as **breakpoint nodes**.

As with any binary search tree, each internal node contains references to theirs parents and children in the tree. When implementing Fortune's algorithm we will also need to augment the internal nodes with additional satellite data.

Firstly we'll need to include a reference to the two sites that generate the neighboring arcs on each side of the breakpoint. This information allows us to dynamically calculate the position of each breakpoint for any position of the sweep line. Unlike the site references in the arc nodes, these ordered pairs of sites uniquely label each breakpoint node.

We will also use the breakpoint nodes to store incomplete Voronoi edges. Because there are always two breakpoints that fall on the same Voronoi edge, however, we can't store a unique edge in each breakpoint node. What we can do is split each edge into half-edges and then store a unique half-edge in each breakpoint node. This makes the implementation of Fortune's algorithm more symmetric.

As the sweep line progresses these incomplete half-edges will not yet have an originating 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).

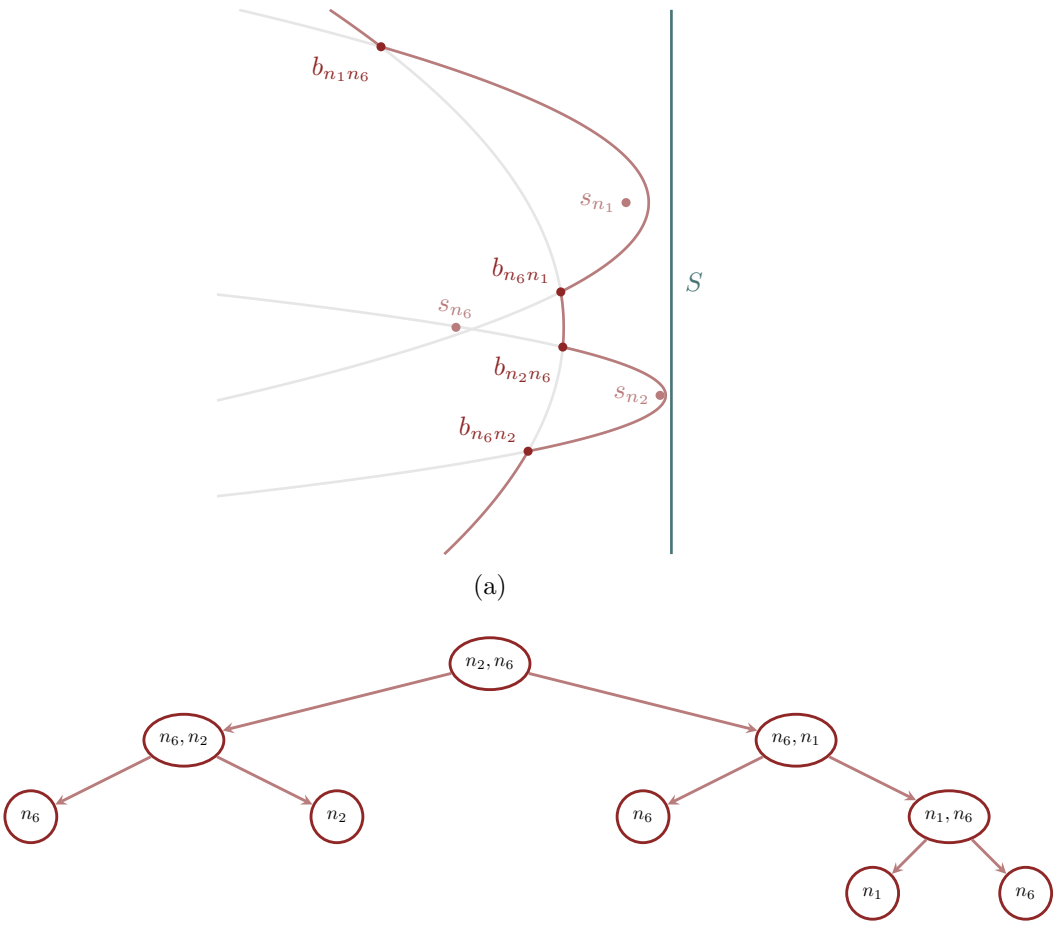
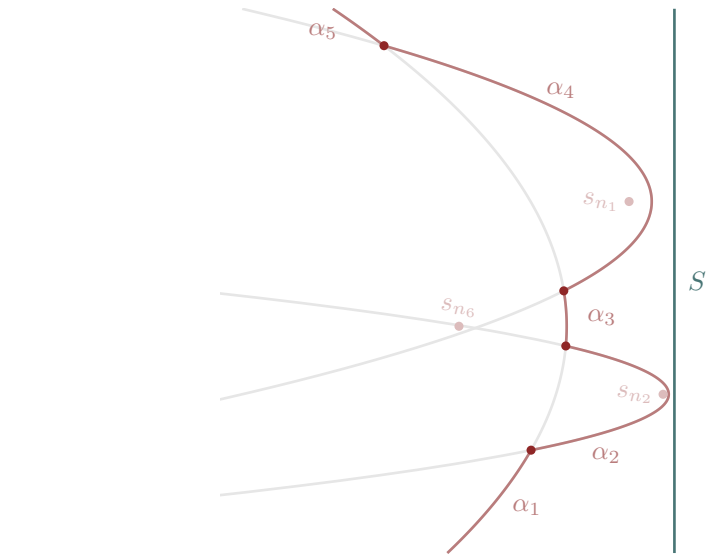
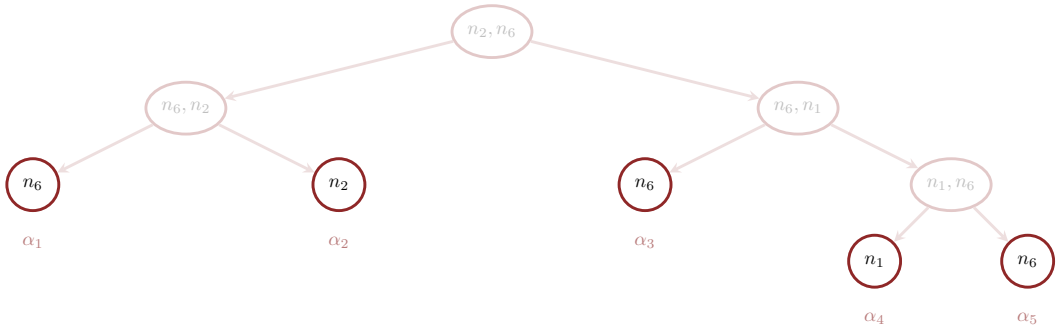


Figure 24: Every (a) beach line can be implemented with a (b) slightly modified binary search tree where internal nodes represent the breakpoints in the beach line and leaf nodes represent the arcs. Critically the ordering of nodes in the binary tree preserves the spatial relationships of the objects in the beach line. This main difference between this binary search tree and a standard binary search is a lack of static keys within each node.



(a)



(b)

Figure 25: (b) Each leaf node in the binary search tree corresponds to (a) a parabolic arc in the beach line. Because of the shape of these arcs changes with the position of the sweep line these arc nodes do not include any geometric information. Instead they include a reference to the site that generates the arc which then allows us to calculate the current arc shape whenever necessary.

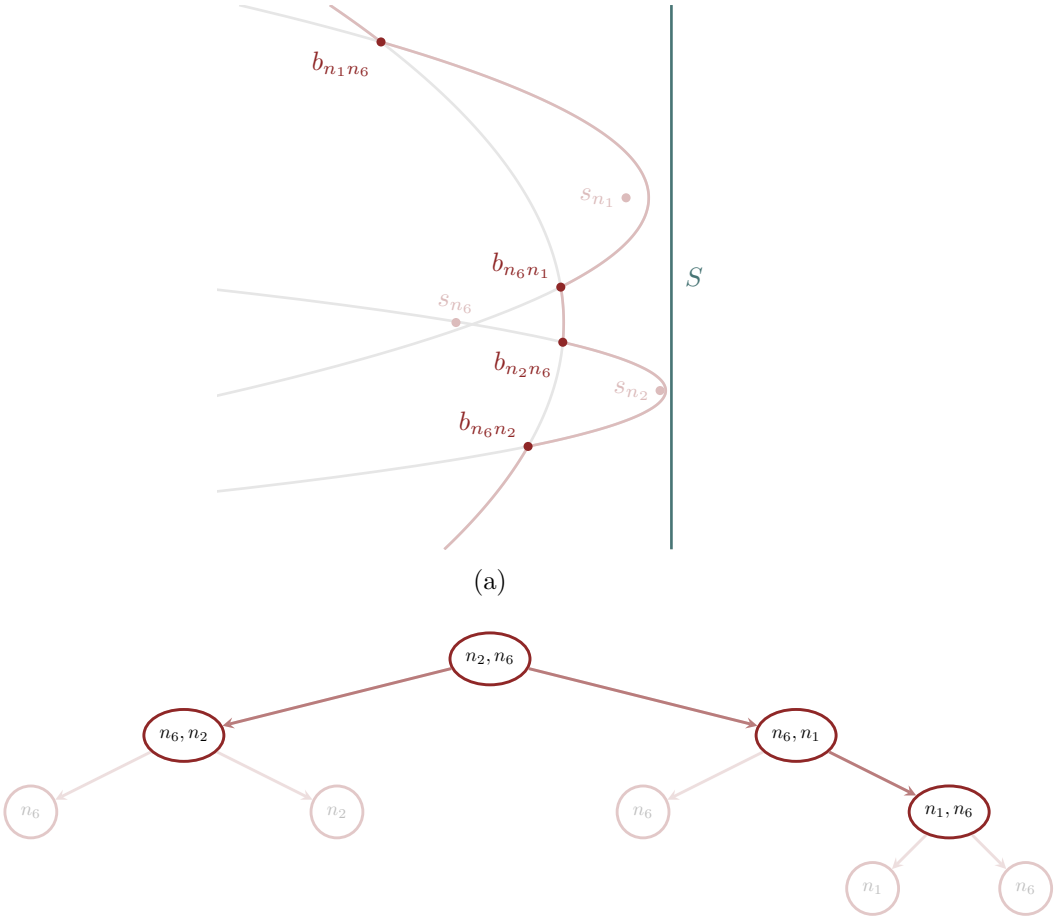


Figure 26: (b) Each internal node in the binary search tree corresponds to (a) a breakpoint on the beach line. These nodes are equipped with references to the two neighboring sites as well as a reference to one Voronoi half-edge.

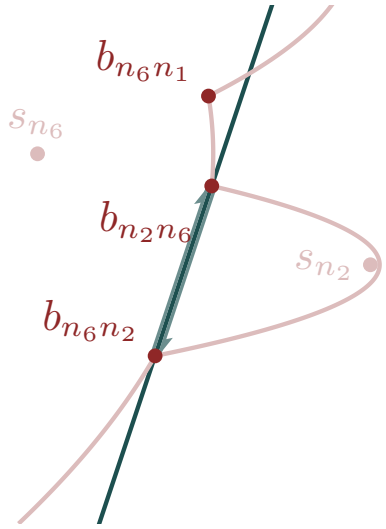


Figure 27: Initially the half-edges stored in each breakpoint node are not anchored to any Voronoi vertices. When visualizing the progression of Fortune's algorithm we can always anchor each half-edge to the current position of the corresponding breakpoint. This ensures that the half-edges always span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

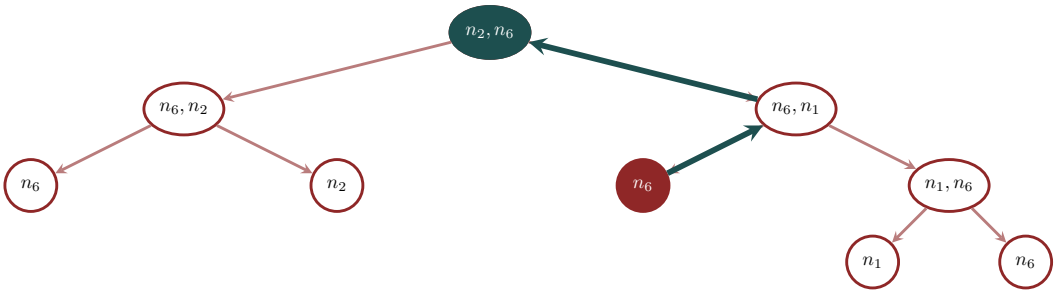
4.2.3 Nearest-Neighbor Search

Binary search trees feature *predecessor* and *successor* operations that take any node and efficiently find the next smallest and next largest node, respectively (Figure 28). The predecessor and successor of an arc node are always the two neighboring breakpoint nodes, while the predecessor and success of a breakpoint node are always the two neighboring arc nodes.

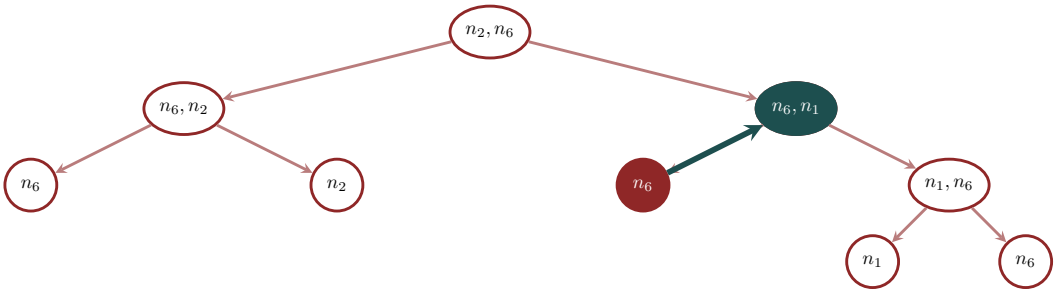
To implement Fortune's algorithm, however, we will need to slightly extend these nearest neighbor search operations. For example when considering triplets of neighboring arcs for potential vertex events we need to be able to find not just the next smallest and next largest nodes but rather the next smallest and next largest *arc* nodes. That said, operations require just calling the standard predecessor and successor operations twice (Figure 29).

4.2.4 Beach Line Search

The key difference (absolutely an intentional pun) between specialized beach line trees and generic binary search trees is that, while both are ordered, beach line trees 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.

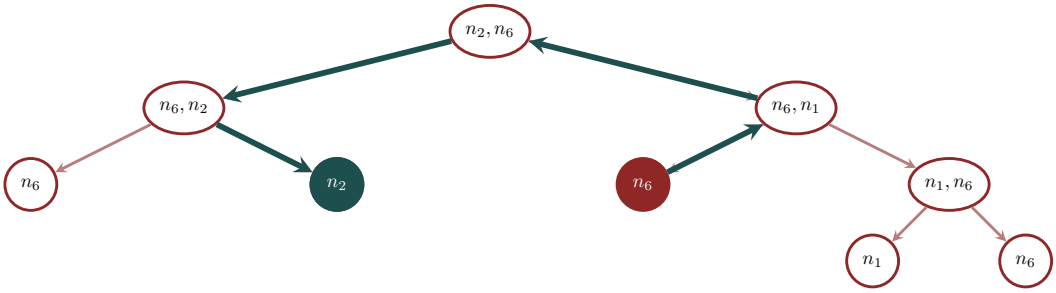


(a)

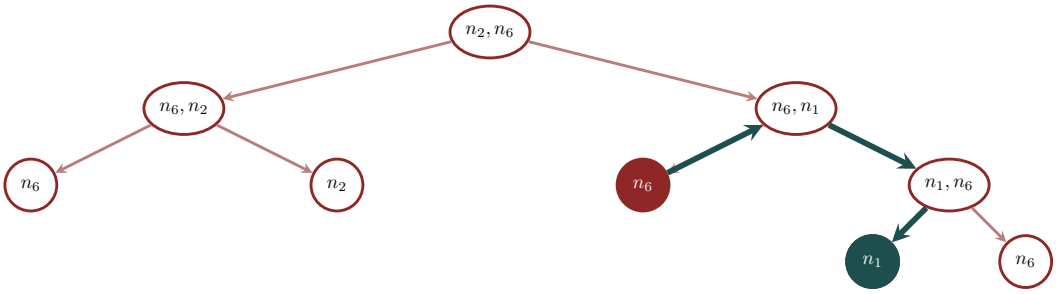


(b)

Figure 28: Binary search trees are equipped with (a) predecessor and (b) successor search operations which allow us to navigate across neighboring objects on the beach line.



(a)



(b)

Figure 29: Implementing Fortune's algorithm is a bit easier if we introduce (a) predecessor arc search and (b) successor arc search operations which allow us to quickly construct triplets of neighboring arcs that might spawn new Voronoi vertices.

Without explicit keys beach line trees do not feature traditional key search operations. They do, however, admit conditional positional searches.

In particular Fortune's algorithm requires being able to find the arc in a beach line that intersects with a particular vertical position y. To find the corresponding arc node in a beach line tree we can modify the standard key search operation, comparing current vertical positions at each step instead of keys (Figure 30).

This operation begins by setting the current position of the sweep line, S, and then accessing the root node of the binary tree. If the root node is an arc node then the beach line consists of a single parabola and we're already done. Otherwise the root node is a breakpoint node and we can compute the current vertical position of the breakpoint given S.

If the breakpoint height is larger than y then we know that the intersecting arc is below the breakpoint in the beach line, and to the left of the current node in the binary tree. In that case we move to the left child of the current node. When the breakpoint height is smaller than y then we have to move up the beach line, which is accomplished by moving to the right child of the current node.

Regardless of which direction we take, we then just repeat these operations until we reach a childless arc node. Because arc nodes correspond to parabolic curves there is no single height that we can compare to y at this point. Fortunately we no longer have to make comparisons as the first arc node we find will always represent the arc that intersects with y.

4.2.5 Insertion and Deletion

In a binary search tree the node deletion operation depends on only the relative ordering of the nodes, and not on any explicit key values. Consequently removing nodes is the *mostly* the same operation for beach line trees and standard binary search trees. The only subtlety is that after removing a node we will have to update the site references in any neighboring breakpoints.

The lack of explicit keys in a beach line tree, however, makes the standard key insertion operation invalid. That said, Fortune's algorithm doesn't actually require a generic key insertion. Objects are added to the beach line in one and only one way: splitting an existing arc into three arcs and two breakpoints. This can be compartmentalized into a *subtree insertion*.

4.2.6 Balancing

Number of operations needed for all of the binary tree operations we have discussed here scales with the height of a given tree. Now the height of most trees is logarithmic in the number of nodes, which in the context of Fortune's algorithm is also logarithmic in the number of sites. Consequently the tree operations usually require only a logarithmic number of operations, and a *typical* evaluation of Fortune's algorithm will achieve an overall $N \log N$ cost scaling.

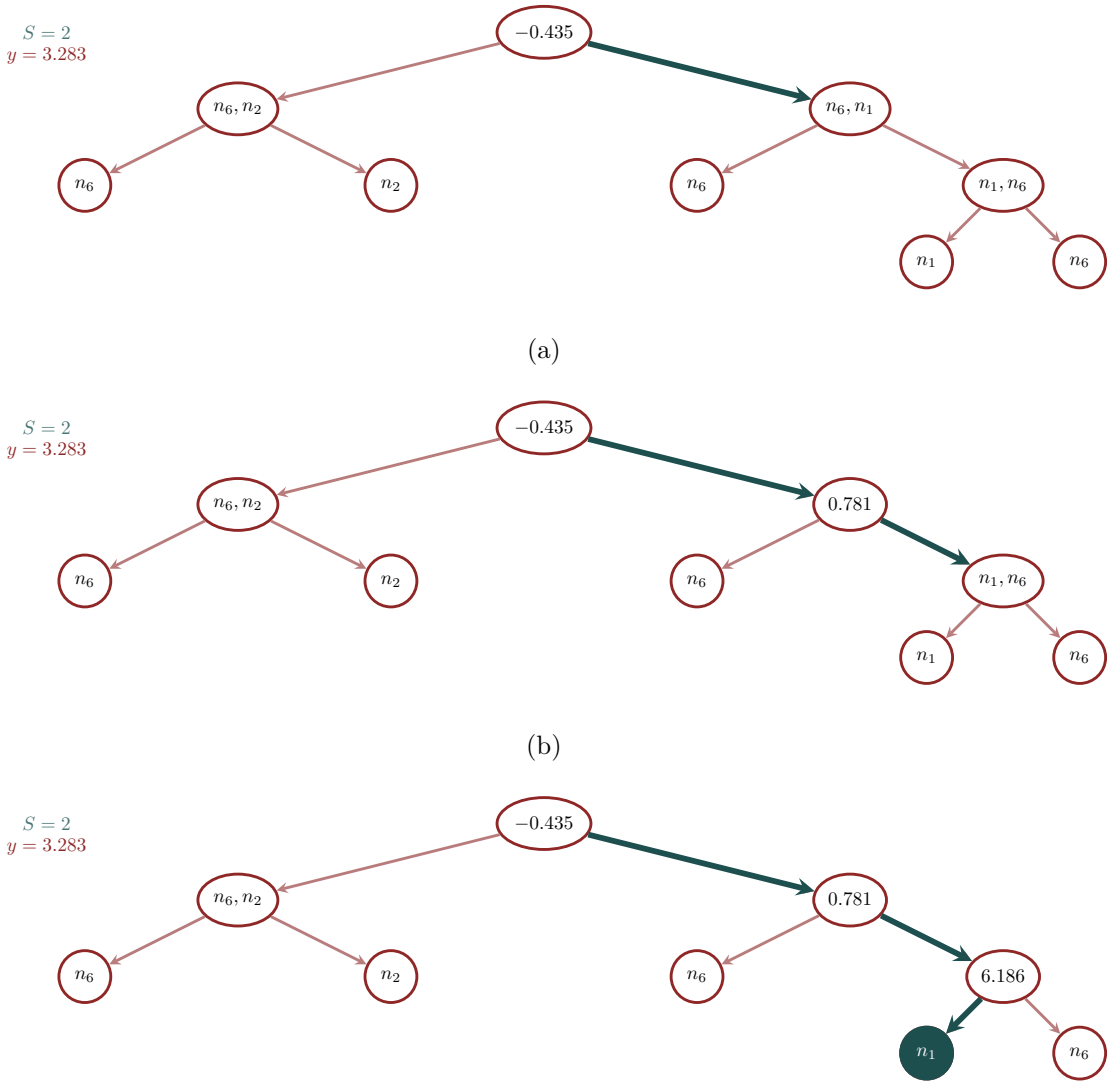
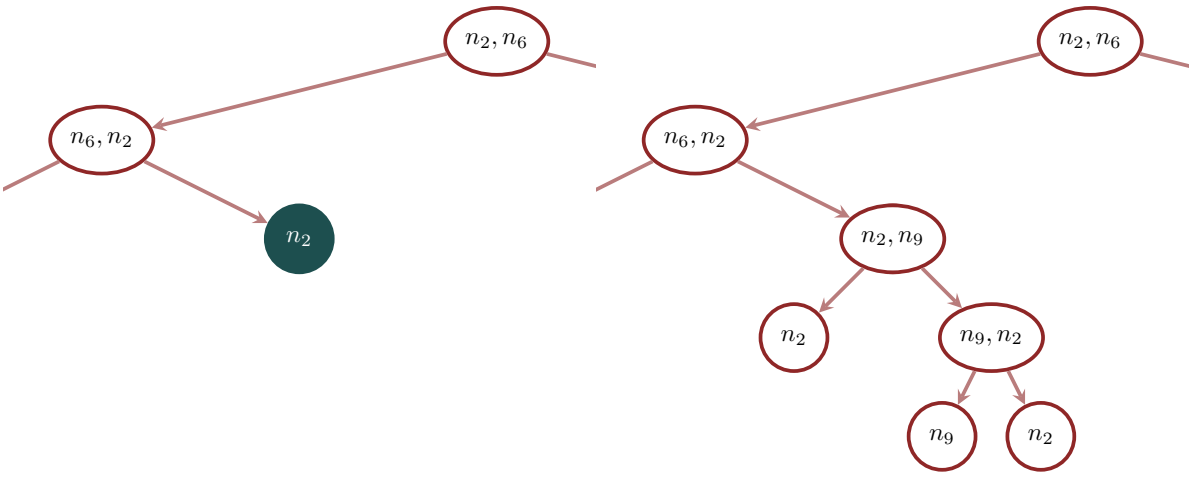


Figure 30: Unlike the nodes in standard binary search trees, the nodes in beach line binary trees are not equipped with static keys. That said for the internal breakpoint nodes we can dynamically compute vertical positions given the current configuration of the beach line, which allows us to search the beach line for positional intersections. To find the arc that intersects $y = 3.832$ we (a) start at the root node and then (b, c) progress down the binary tree, evaluating current breakpoint positions as needed until we reach an arc node.



(a) (b)

Figure 31: In Fortune's algorithm beach line trees are expanded through a subtree insertion operation where (a) an arc node is replaced by a (b) subtree representing a triplet of arc nodes and the two breakpoints between them. Note that the site references in the initial neighboring breakpoint nodes are still correct, and hence don't have to be updated, after the subtree insertion.

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 a little bit of effort we can adapt these balanced operations to beach line trees, which then allows us to achieve optimal performance when implementing 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 the queue as well as operations for inserting and deleting elements while maintaining the overall order of the event queue.

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 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 beach line, and then updating the Voronoi graph as necessary. With doubly-connected event queues, binary search trees, and their operations processing each event is relatively straightforward.

4.5 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 arc predecessor and arc successor operations of the beach line tree to find the neighboring arcs, and then remove 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 arc collapses 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 link up the previous and next references between these two new half-edges, the two half-edges in the neighboring breakpoints, and their twins (Figure 33).

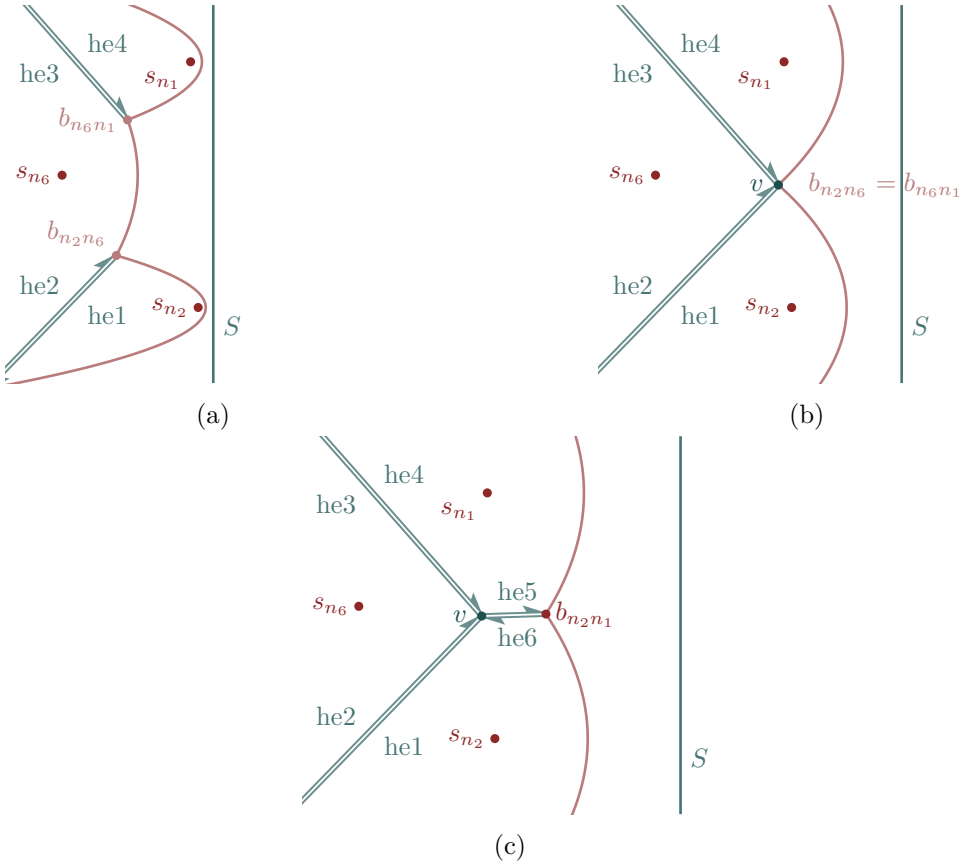


Figure 32: Vertex events require substantial updates to the half-edges of the Voronoi graph. (a) The breakpoints that merge in a vertex event contain references to two half-edges, here `he1` and `he3`. (b) When processing a vertex event we delete these references. (c) Then we create two new half-edges,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

Now we're ready to remove the collapsing arc. This requires removing the corresponding arc node as well as one of its neighboring breakpoint nodes, rebalancing the binary search tree after each removal (Figure 34). Which breakpoint node we remove is a matter of convention, but to avoid any ambiguity I will maintain a convention where we always remove the smaller breakpoint.

At this point the remaining breakpoint node, the larger breakpoint in my chosen convention, needs to be updated. Firstly its left site needs to be updated to the site that generates its new

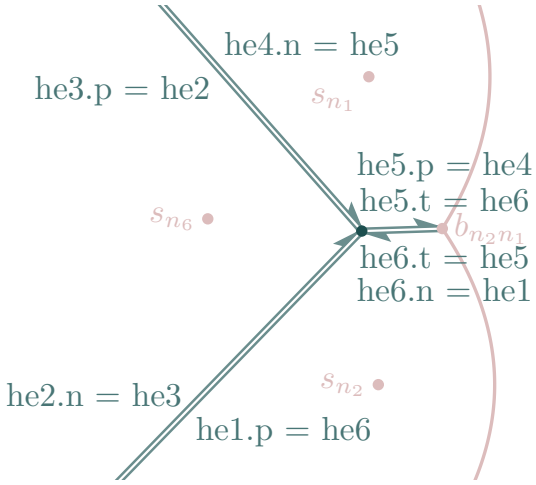


Figure 33: The trickiest part of processing a vertex event is making sure that the four existing half-edges and the two new half-edges all properly reference each other. These references ultimately form the skeleton of the Voronoi graph.

left neighboring arc; this new neighbor can be accessed by calling the predecessor operation on the updated tree. Secondly we need to set its half-edge references to the new twin half-edge that we just created.

Lastly there are two new triplets of neighboring arcs that we have to check for possible new vertex events. We can efficiently access these arcs by repeatedly using the arc predecessor and arc successor operations

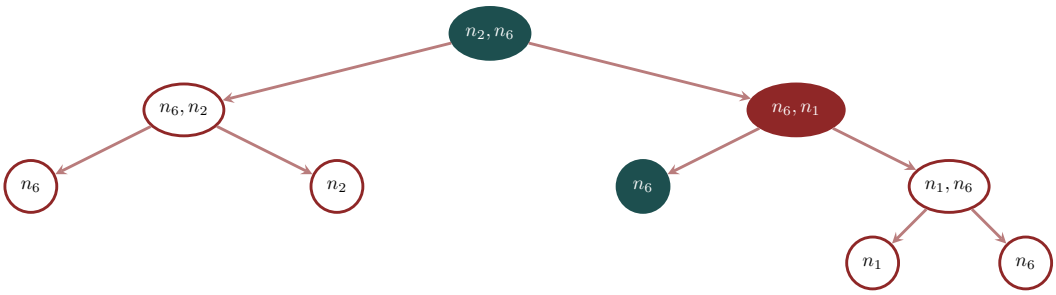
4.5.1 Processing Site Events

Because site events do not update the Voronoi diagram they are a little bit easier to process.

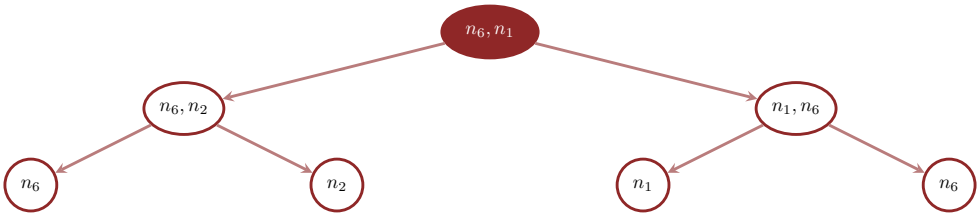
First things first we have to find the arc that intersects with the height of the new site by searching through the beach line tree. Because splitting this arc will invalidate any existing vertex event centered on it, we have to remove any corresponding vertex event from the event queue.

In order to incorporate the new site into the beach line we have to apply the subtree insertion operation on the intersecting node, replace it with a subtree that includes two arc nodes associated with the original site, one arc node associated with the new site, and two interior breakpoint nodes separating them. When working with a balanced binary tree we also have to make sure that this insertion operation properly rebalances the updated tree.

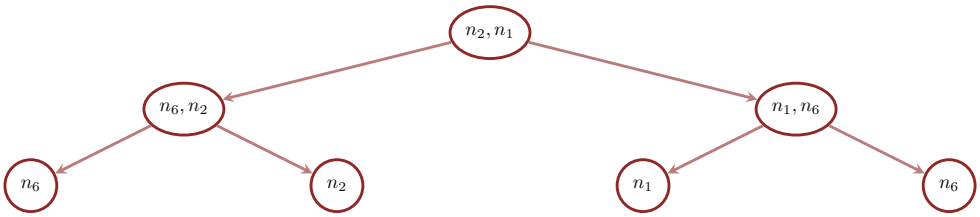
The two new breakpoint nodes are then filled with two new half-edges that are twinned to each other (Figure 35). One half-edge faces the old site and the other faces the face the new site.



(a)



(b)



(c)

Figure 34: (a, b) Removing an arc from a beach line requires deleting two nodes from the corresponding binary tree, a leaf node for the arc itself and an internal node for one of the neighboring breakpoints. Which breakpoint we remove is a matter of convention. (c) After removing the two nodes from the binary tree we need to update the site information in the remaining breakpoint node. Here the left site has to be updated from n_6 to n_2.

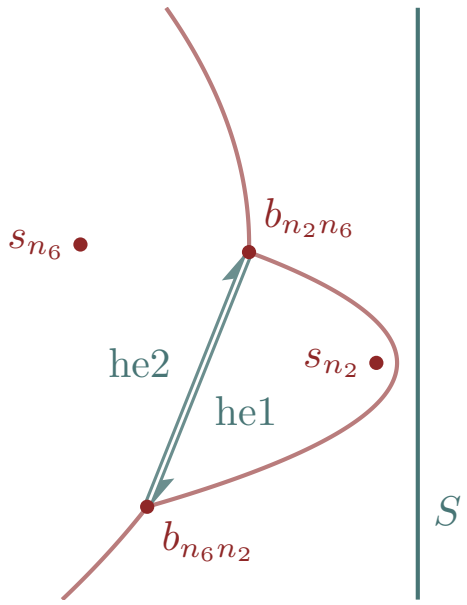


Figure 35: The introduction of a new site splits an existing arc on the beach line into three arcs separated by two new breakpoints. This also creates two new half-edges in the Voronoi graph that are twinned with each other. A reference to each of these half-edges is stored in one of the two new breakpoint nodes that are inserted into the beach line tree. Here a reference to the half-edge `he1` is stored in the breakpoint node `(n1, n2)` while a reference to `he2` is stored in the breakpoint node `(n2, n1)`.

Finally we have to consider potential new vertex events that. Incorporating the new site creates up to two new triplets of neighboring arcs that might generate new vertex events. Fortunately the arc nodes in these new triplets are once again readily accessed with the arc predecessor and arc successor beach line operations.

5 Clean Up

Once the event queue is 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 binary search tree implementing the beach line will *not* be empty. The remaining breakpoint nodes all contain half-edges with unbounded origins. If we want to visualize the Voronoi graph neatly then we'll need 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 the bounding box.

6 Demonstration

To put all of this discussion into practice let's work through a full implementation of Fortune's algorithm in `R`. To be honest `R` is at best an awkward language for this application, which is much more natural in languages that are more friendly to object-oriented programming.

Here I get by with `R6` classes (Chang 2025), but we'll have to suffer with some of its clumsy interactions with base `R` graphics. At least using `R6` classes gets us 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, we'll start with some graphics settings.

```
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")
```

6.1 Beach Line

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

This class implements both breakpoint and arc nodes by including the satellite data for each and using only one type of data at a time. In particular the variables `site` and `scheduled_vertex` are used for only arc nodes while `site_left`, `site_right`, and `he` are used for only breakpoint nodes. 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,
                  scheduled_vertex = NULL,
                  site_left = NULL,
                  site_right = NULL,
                  he = NULL,
                  parent = NULL,
                  left = NULL,
                  right = NULL,
                  red = FALSE,
                  id = 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 two `node` instances refer to the same object. In many programming languages this is done by directly comparing memory addresses, but `R6` classes don't seem to play will with memory.

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

An actual beach line tree is defined by a 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,
                       last_half_edge_id = NULL,
                       initialize = function() {
                         self$nil <- node$new(as.integer(0))
                         self$root <- self$nil
                         self$last_node_id <- as.integer(1)
                         self$last_half_edge_id <- as.integer(1)
                       }
                     ))
```

The `beachline` class also gives us a place to keep track of the node indices, as well as half-edge indices that we'll use later.

6.1.1 Black-Red Binary Tree Operations

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

```
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
})
```

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 subtrees.

```
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 rebalance 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$red <- FALSE
        w$right$red <- FALSE
        private$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)
})
```

Fortune's algorithm requires accessing lots of neighboring nodes.

```
beachline$set("public", "arc_successor",
function(x) {
  if (x$right != self$nil) {
    # Stay still
    y <- x
  } else {
    # Ascend left
    y <- x$parent
    while (   y != self$nil
           && x == y$right) {
      x <- y
      y <- y$parent
    }
    # Ascend right
    while (   y != self$nil
           && y$right == self$nil) {
      y <- y$parent
    }
  }
  if (y != self$nil)
    y <- self$min(y$right)
  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)
  }
})
```

```
beachline$set("public", "arc_predecessor",
function(x) {
  if (x$left != self$nil) {
    # Stay still
    y <- x
  } else {
    # Ascend right
    y <- x$parent
    while (   y != self$nil
           && x == y$left) {
      x <- y
      y <- y$parent
    }
    # Ascend left
    while (   y != self$nil
           && y$left == self$nil) {
      y <- y$parent
    }
  }
  if (y != self$nil)
    y <- self$max(y$left)
  return(y)
})
```

```
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)
  }
})
```

Tree walks are useful for displaying the current ordering of the nodes.

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

  if (x != self$nil) {
    self$inorder_arc_walk(x$left)

    if (!is.null(x$site))
      cat(paste0(x$site, ', '))

    self$inorder_arc_walk(x$right)
  }
})
```

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

  if (x != self$nil) {
    self$inorder_walk(x$left)

    if (is.null(x$site)) {
      cat(paste0('(', x$site_left, ',',
                 x$site_right, '), '))
    } else {
      cat(paste0('(', x$site, '), '))
    }

    self$inorder_walk(x$right)
```

```
  }
})
```

6.1.2 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)
})
```

```
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)
}
```

```
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)

  if (!is.null(node$site)) {
    text(x, y, col="white", cex=text_cex,
         labels=node$site)
  } else {
    text(x, y, col="white", cex=text_cex,
         labels=paste0('(', node$site_left, ",",
                       node$site_right, ')'))
  }
}
```

6.1.3 Fortune's Algorithm Geometry Calculations

As we've seen, implementing Fortune's algorithm requires a lot of geometry 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]
  beta <- 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]) )

  d <- sqrt(beta**2 - a * c)
  y_int <- (beta + d) / a
  if (y_int < ys[n1] | y_int > ys[n2])
    y_int <- (beta - d) / a

  return(y_int)
})
```

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
```

```
  r <- sqrt( (xc - xs[n1])**2 + (yc - ys[n1])**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$set("private", "arc_search",
function(z, y, S) {
  if (is.null(z$site)) {
    by <- self$pair_intersection(z$site_left,
                                 z$site_right, S)
    if (y <= by)
      z <- private$arc_search(z$left, y, S)
    else
      z <- private$arc_search(z$right, y, S)
  }
  return(z)
})
```

6.1.4 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 arc nodes to see if they define a valid vertex event and, if so, adding that event to the event queue `queue`. There is probably a more elegant way to verify that `alpha_mid` will collapse than comparing `upper_int` to `lower_int`.

```
beachline$set("private", "schedule_new_vertex_event",
function(queue, alpha_left, alpha_mid, alpha_right, S) {
  # Skip vertex if any of the sites are nil
  if (alpha_left == self$nil || alpha_right == self$nil)
    return()

  # Skip vertex if any of the sites are the same
  if (alpha_left$site == alpha_right$site) {
    return()
  }
```

```
  # Construct circumcircle
  int <- self$circumcircle(alpha_left$site,
                           alpha_mid$site,
                           alpha_right$site)
  S_new <- int[1] + int[3]


  lower_int <- self$pair_intersection(alpha_left$site,
                                      alpha_mid$site, S_new)
  upper_int <- self$pair_intersection(alpha_mid$site,
                                      alpha_right$site, S_new)

  if (abs(upper_int - lower_int) < 1e-12) {
    vertex_event <- event$new(NULL, S_new, NULL, alpha_mid)
    alpha_mid$scheduled_vertex <- vertex_event
    queue$insert_event(vertex_event)
  }
})
```

Now we can process site events.

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

  if (self$root == self$nil) {
    # Initialize beach line if empty
    self$root <- node$new(self)
    self$root$site <- new_site
  } else {
    # Find beach line arc that intersects
    # vertical position of new site
    gamma <- private$arc_search(self$root,
                               ys[new_site],
                               new_event$x)
    old_site <- gamma$site

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

```
    }

    # Expand intersecting arc into new subtree
    gamma$site_left <- old_site
    gamma$site_right <- new_site
    gamma$scheduled_vertex <- NULL
    gamma$site <- NULL

    gamma$left <- node$new(self)
    gamma$left$parent <- gamma
    gamma$left$site <- old_site
    gamma$left$red <- TRUE
    private$insert_fixup(gamma$left)

    gamma$right <- node$new(self)
    gamma$right$parent <- gamma
    gamma$right$site_left <- new_site
    gamma$right$site_right <- old_site
    gamma$right$red <- TRUE

    zeta <- gamma$right
    private$insert_fixup(zeta)

    zeta$left <- node$new(self)
    zeta$left$parent <- zeta
    zeta$left$site <- new_site
    zeta$left$red <- TRUE
    private$insert_fixup(zeta$left)

    zeta$right <- node$new(self)
    zeta$right$parent <- zeta
    zeta$right$site <- old_site
    zeta$right$red <- TRUE
    private$insert_fixup(zeta$right)

    # Instantiate and connect new half-edges
    gamma$he <- half_edge$new(self, NULL, old_site,
                              NULL, NULL, NULL)
    gamma$he$twin <- half_edge$new(self, NULL, new_site,
                                   gamma$he, NULL, NULL)
    zeta$he <- gamma$he$twin
```

```
    # Attempt to schedule vertex events
    # for the two new arc triplets
    alpha1 <- self$arc_predecessor(zeta$left)
    alpha2 <- self$arc_predecessor(alpha1)
    private$schedule_new_vertex_event(queue,
                                      alpha2, alpha1, zeta$left,
                                      new_event$x)

    beta1 <- self$arc_successor(zeta$left)
    beta2 <- self$arc_successor(beta1)
    private$schedule_new_vertex_event(queue,
                                      zeta$left, beta1, beta2,
                                      new_event$x)
  }
})
```

Vertex event processing follows.

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

  beta1 <- self$arc_successor(gamma)
  beta2 <- self$arc_successor(beta1)

  alpha1 <- self$arc_predecessor(gamma)
  alpha2 <- self$arc_predecessor(alpha1)

  # Clean up deprecated vertex events in the queue
  if (!is.null(beta1$scheduled_vertex)) {
    queue$delete_event(beta1$scheduled_vertex)
    beta1$scheduled_vertex <- NULL
  }

  if (!is.null(alpha1$scheduled_vertex)) {
    queue$delete_event(alpha1$scheduled_vertex)
    alpha1$scheduled_vertex <- NULL
  }

  # Create vertex and update edges
  int <- self$circumcircle(alpha1$site, gamma$site, beta1$site)
  v <- vertex$new(int[1:2], alpha1$site, gamma$site, beta1$site)
```

```
  graph$add_vertex(v)

  # Connect half-edges
  bp1 <- self$predecessor(gamma)
  bp2 <- self$successor(gamma)

  new_he <- half_edge$new(self, v, beta1$site,
                          NULL, NULL, NULL)
  new_he$twin <- half_edge$new(self, NULL, alpha1$site,
                               new_he, NULL, NULL)

  v$he <- new_he

  bp1$he$origin <- v
  bp2$he$origin <- v

  bp2$he$prv <- bp1$he$twin
  bp1$he$twin$nxt <- bp2$he

  bp2$he$twin$nxt <- new_he
  new_he$prv <- bp2$he$twin

  new_he$twin$nxt <- bp1$he
  bp1$he$prv <- new_he$twin

  # Delete deprecated nodes
  self$delete(gamma)
  self$delete(bp1)

  # Update remaining breakpoint
  bp2$site_left <- alpha1$site
  bp2$he <- new_he$twin

  # Check for new vertex events
  private$schedule_new_vertex_event(queue,
                                    alpha2, alpha1, beta1,
                                    new_event$x)

  private$schedule_new_vertex_event(queue,
                                    alpha1, beta1, beta2,
                                    new_event$x)
})
```

```
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,
                               next_event)
  } else {
    private$process_vertex_event(queue,
                                 graph,
                                 next_event)
  }
})
```

6.1.5 Fortune's Algorithm Visualization Functions

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

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

  if (is.null(z$site)) {
    by <- self$pair_intersection(z$site_left,
                                 z$site_right, S)
    bx <- self$parabola_xs(by, xs[z$site_left],
                           ys[z$site_left], S)
    z$he$pseudo_origin <- c(bx, by)
    self$fill_breakpoints(S, z$left )
    self$fill_breakpoints(S, z$right)
  }
})
```

```
plot_beachline <- function(bl, S,
                           xlim=c(-4, -2),
                           ylim=c(-2, 2)) {
  bl$fill_breakpoints(S)

  plot(xs, ys,
```

```
       axes=FALSE, ann=FALSE, frame.plot=F,
       col=c_dark, pch=16,
       xlim=xlim, ylim=ylim,
       main="")

  # Plot sweep line
  abline(v=S, col=c_mid_teal, lty=2, lwd=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)
    plot_arc(bl, bl$root, ylim[1], ylim[2], S)
}
```

```
plot_arc <- function(bl, z, y_min, y_max, S) {
  if (!is.null(z$site)) {
    # Leaf node
    if (y_max > y_min + 1e-3) {
      parabola_ys <- seq(y_min, y_max, 1e-3)
      lines(bl$parabola_xs(parabola_ys, xs[z$site],
                           ys[z$site], S),
            parabola_ys,
            lty=2, lwd=2, col=c_mid)
    }
  } else {
    # Internal node
    by <- bl$pair_intersection(z$site_left,
                               z$site_right, S)
    bx <- bl$parabola_xs(by, xs[z$site_left],
                         ys[z$site_left], S)
    points(bx, by, col=c_light, pch=16)
    text(bx + 0.1, by,
         labels=paste0(z$site_left, ",", z$site_right))

    if (is.null(z$he$origin)) {
```

```
      p1 <- z$he$pseudo_origin
    } else {
      p1 <- z$he$origin$pos
    }
    if (is.null(z$he$twin$origin)) {
      p2 <- z$he$twin$pseudo_origin
    } else {
      p2 <- z$he$twin$origin$pos
    }

    lines(c(p1[1], p2[1]), c(p1[2], p2[2]),
          lwd=2, col=c_mid_teal)

    plot_arc(bl, z$left,  y_min, by, S)
    plot_arc(bl, z$right, by, y_max, S)
  }
}
```

6.2 Event Queue

Similarly to how we handled the `node` class, 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]]]
})
```

6.3 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.

```
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(tree, o, lf, t, p, n) {
                          self$origin <- o
                          self$left_face <- lf
                          self$twin   <- t
                          self$prv    <- p
                          self$nxt    <- n

                          tree$last_half_edge_id <- tree$last_half_edge_id + 1
                          self$id <- tree$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,
                  initialize = function() {
                    self$vertices <- list()
                    self$pseudo_vertices <- list()
                    self$faces <- list()
                  },
                  add_vertex = function(v) {
                    self$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)
                  }))
```

We can do so much with 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$set("public", "anchor_dangling_edges",
function(z, xs, ys) {
  if (is.null(z$site)) {
    # Dangling edge
    he <- z$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
    self$add_pseudo_vertex(pv)

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

Most relevant to the construction of a Voronoi diagrams we 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
    }
  }
})
```

```
dcel$set("public", "trace_face",
function(f) {
  print(paste('Face', f$site))
  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
    }
  }

  cat('  ', sprintf('%.3f/%.3f,', perimeter_xs, perimeter_ys), '\n')
})
```

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, 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$he
  if (   !is.null(he$origin)
         && !is.null(he$twin$origin)) {
    plot_half_edge(he$origin$pos, he$twin$origin$pos,
                   lwd, ifelse(he$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
    plot_half_edge(he$twin$origin$pos, he$origin$pos,
                   lwd, ifelse(he$twin$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
  }

  he <- he$prv
  if (   !is.null(he$origin)
         && !is.null(he$twin$origin)) {
    plot_half_edge(he$origin$pos, he$twin$origin$pos,
                   lwd, ifelse(he$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
    plot_half_edge(he$twin$origin$pos, he$origin$pos,
                   lwd, ifelse(he$twin$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
  }

  he <- he$twin$prv
  if (   !is.null(he$origin)
         && !is.null(he$twin$origin)) {
    plot_half_edge(he$origin$pos, he$twin$origin$pos,
                   lwd, ifelse(he$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
    plot_half_edge(he$twin$origin$pos, he$origin$pos,
                   lwd, ifelse(he$twin$visited, c_mid, c_mid_teal),
                   delta, eps, theta, gamma)
  }
}
```

```
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
  }
}
```

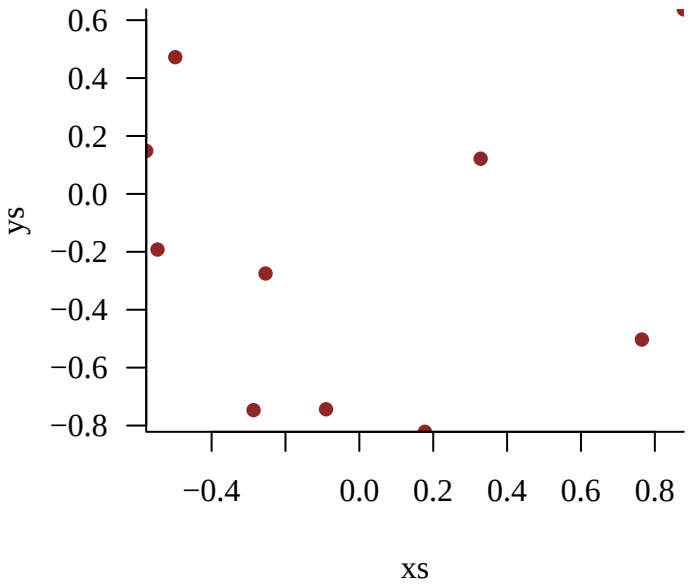
```
plot_face <- function(f, col=c_mid_teal) {
  # Trace boundary
  he <- f$he

  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)
}
```

6.4 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)
```



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, plotting both the current beach line tree and the current beach line, and then displaying the status of the event queue.

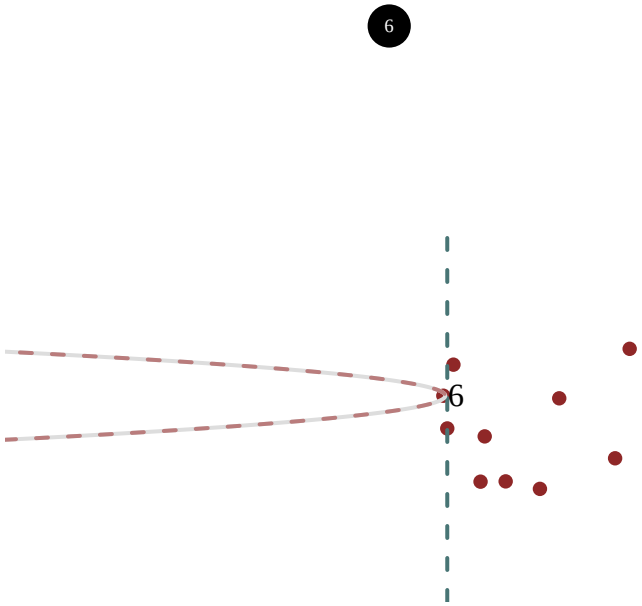
```
process_next_event <- function() {
  if (is.null(queue$next_event()$arc)) {
    cat(paste0('Processing Site Event (',
               queue$next_event()$site, ')\n'))
  } else {
```

```
    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))
  } else {
    plot_beachline(bl, queue$next_event()$x,
                   c(-4, 2), c(-2, 2))
  }

  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)
```

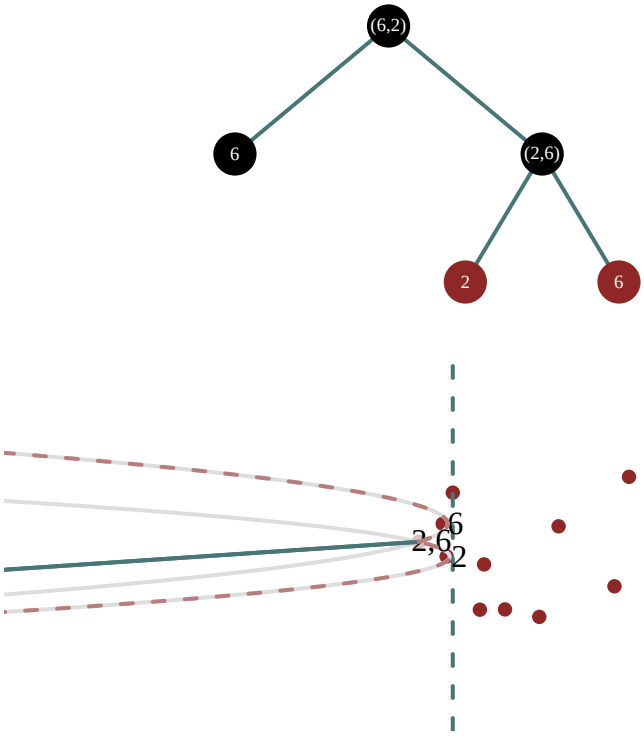



```
Remaining queue:
  S = -0.546 : Site Event at 2
  S = -0.090 : Site Event at 10
  S = -0.498 : Site Event at 1
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = -0.286 : Site Event at 3
  S = -0.254 : Site Event at 7
  S = 0.328 : Site Event at 8
  S = 0.765 : Site Event at 4
```

The next site event splits this monolithic parabola up into three arcs. Note that the breakpoint b_{26} is too far left to be seen in this plot.

```
process_next_event()
```

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

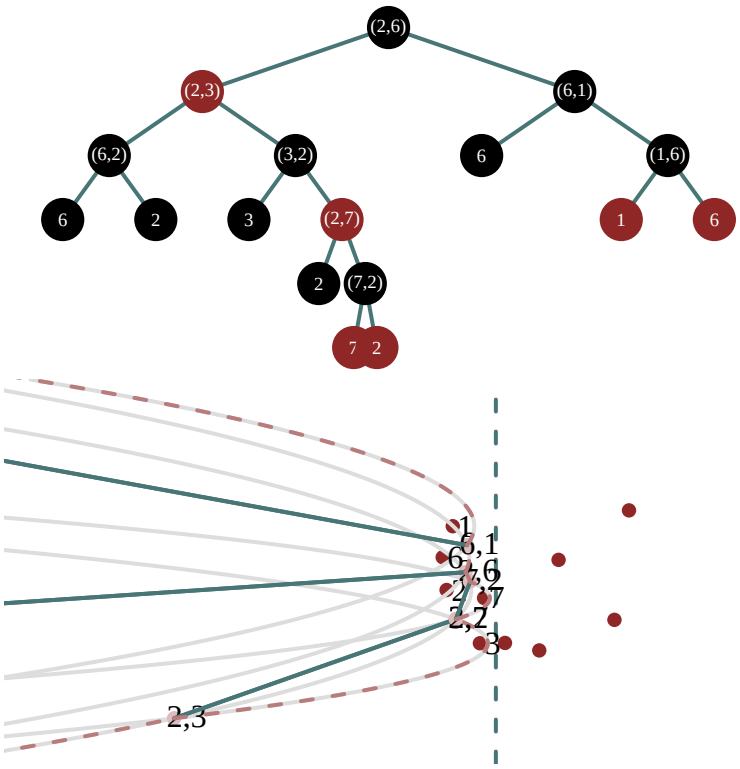



```
Remaining queue:
  S = -0.498 : Site Event at 1
  S = -0.090 : Site Event at 10
  S = -0.286 : Site Event at 3
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = 0.765 : Site Event at 4
  S = -0.254 : Site Event at 7
  S = 0.328 : Site Event at 8
```

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)
if (is.null(queue$next_event()$x)) {
```

```
  plot_beachline(bl, 2,
                 c(-4, 2), c(-2, 2))
} else {
  plot_beachline(bl, queue$next_event()$x,
                 c(-4, 2), c(-2, 2))
}
```




```
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')
```

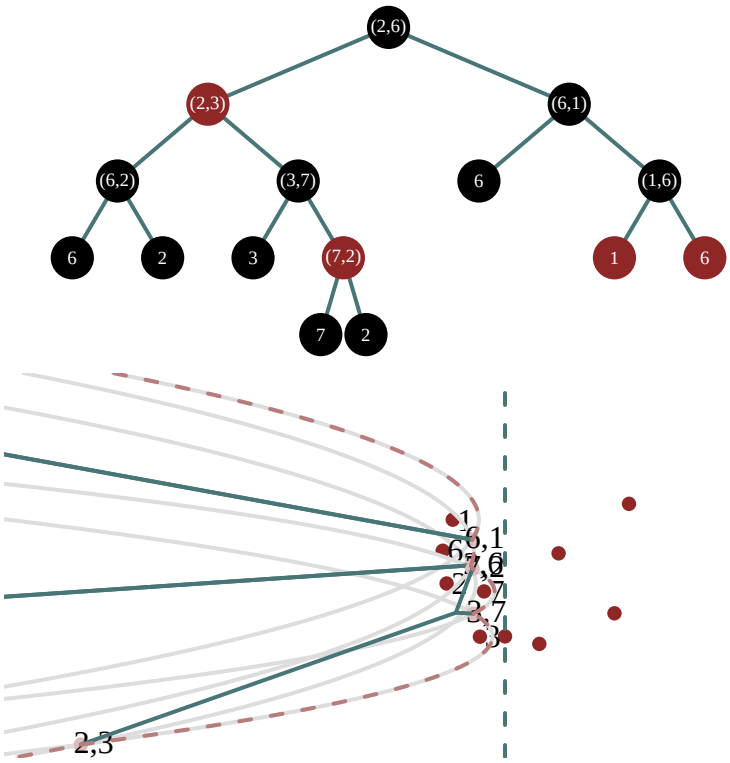
```
  } else {
    cat(paste('  S =', sprintf('%.3f', queue$events[[i]]$x),
              ':',
              'Vertex Event at', queue$events[[i]]$arc$site),
              '\n')
  }
}
```

```
  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 )
```



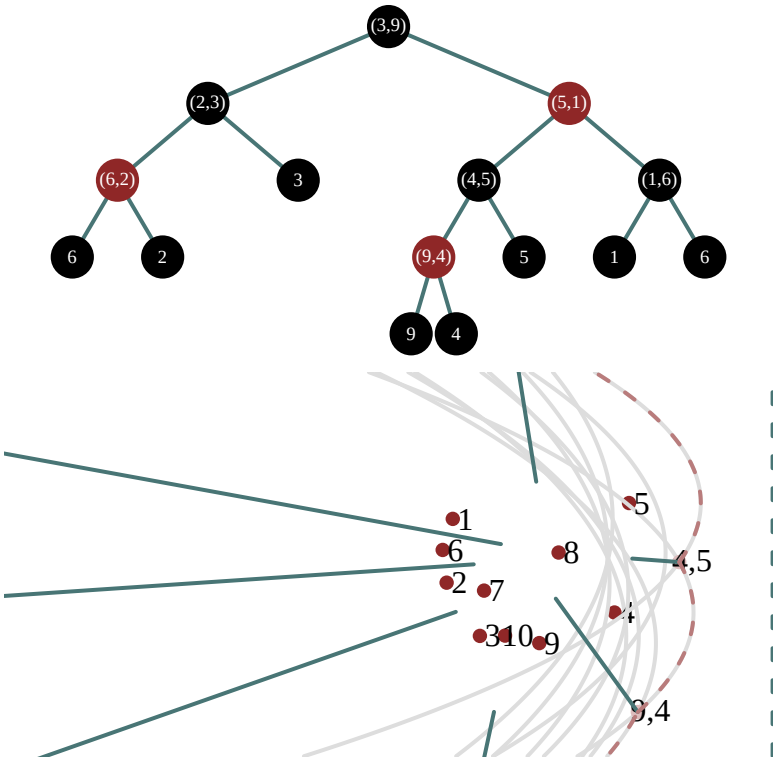

```
Remaining queue:
  S = -0.090 : Site Event at 10
  S = -0.049 : Vertex Event at 2
  S = 0.328 : Site Event at 8
  S = 0.177 : Site Event at 9
  S = 0.878 : Site Event at 5
  S = 0.765 : Site Event at 4
  S = 1.487 : Vertex Event at 6
```

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)
if (is.null(queue$next_event()$x)) {
  plot_beachline(bl, 2,
                 c(-4, 2), c(-2, 2))
} else {
  plot_beachline(bl, queue$next_event()$x,
                 c(-4, 2), c(-2, 2))
}
```



To visualize the entire Voronoi graph within the confines of a finite plot we first compute a bounding box.

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

Note that we can always specify a bounding box of our own. This can be useful, for example, if we want larger or more symmetric bounding boxes for visualization purposes.

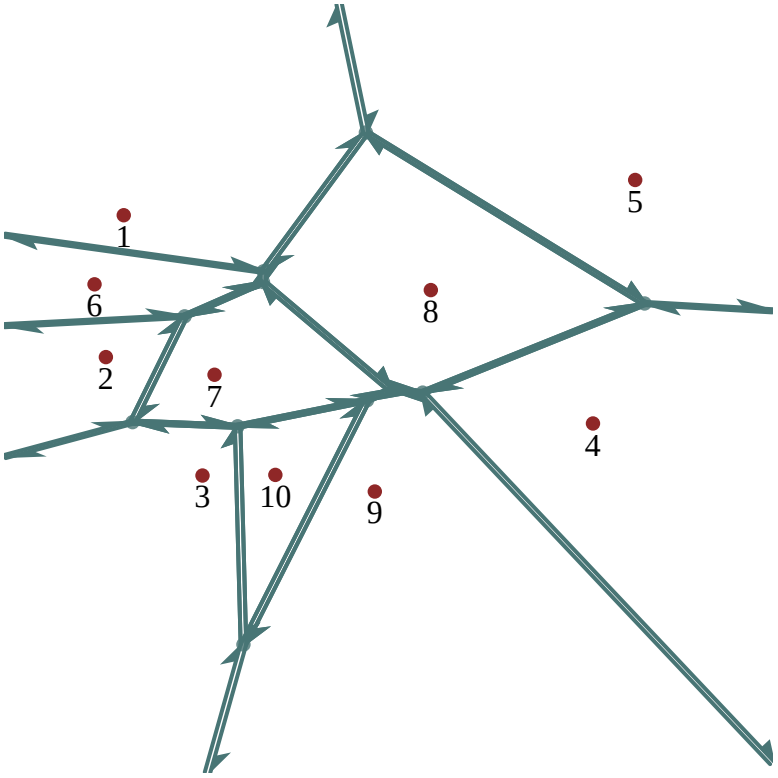
```
voronoi$xlim <- c(-3, 3)
voronoi$ylim <- c(-3, 3)
```

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)
}
```



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

```
voronoi$find_faces()
```

We can always examine the face boundaries directly, which is useful for exporting the Voronoi geometry to other environments.

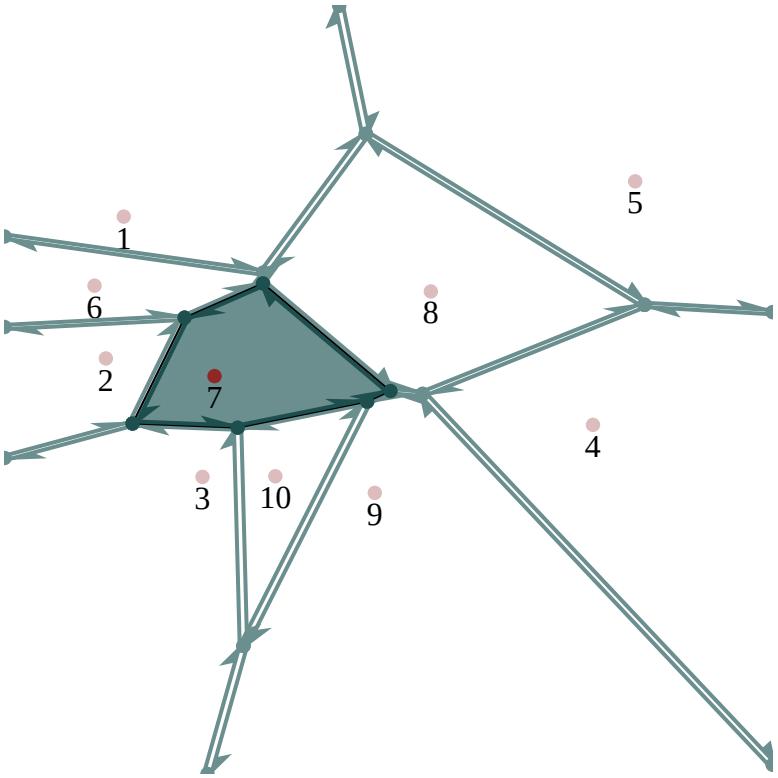
```
for (f in voronoi$faces) {
  voronoi$trace_face(f)
}
```

```
[1] "Face 7"
   -0.475/-0.497, -0.192/-0.516, 0.157/-0.395, 0.220/-0.345, -0.124/0.160, -0.335/-0.001, -0
[1] "Face 3"
   -0.273/-2.137, -0.176/-1.538, -0.192/-0.516, -0.475/-0.497, -0.819/-0.658,
[1] "Face 9"
   1.248/-2.095, 0.306/-0.359, 0.220/-0.345, 0.157/-0.395, -0.176/-1.538, -0.273/-2.137,
[1] "Face 4"
   1.248/0.025, 0.904/0.059, 0.306/-0.359, 1.248/-2.095,
[1] "Face 5"
   0.082/1.459, 0.154/0.859, 0.904/0.059, 1.248/0.025,
[1] "Face 1"
   -0.819/0.379, -0.122/0.209, 0.154/0.859, 0.082/1.459,
[1] "Face 6"
   -0.819/-0.045, -0.335/-0.001, -0.124/0.160, -0.122/0.209, -0.819/0.379,
[1] "Face 2"
   -0.819/-0.658, -0.475/-0.497, -0.335/-0.001, -0.819/-0.045,
[1] "Face 8"
   -0.122/0.209, -0.124/0.160, 0.220/-0.345, 0.306/-0.359, 0.904/0.059, 0.154/0.859, -0.122/0
[1] "Face 10"
   -0.176/-1.538, 0.157/-0.395, -0.192/-0.516, -0.176/-1.538,
```

Or we can just plot the faces directly.

```
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)
```



7 Conclusion

Voronoi diagrams are straightforward to define but subtle to actually derive in practice. Fortune's algorithm is able to systematize the construction of Voronoi diagrams. Understanding how the algorithm actually achieves this task, however, requires some careful 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:

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
```