Voronoi Diagrams (Fortune's Construction) Executive Summary

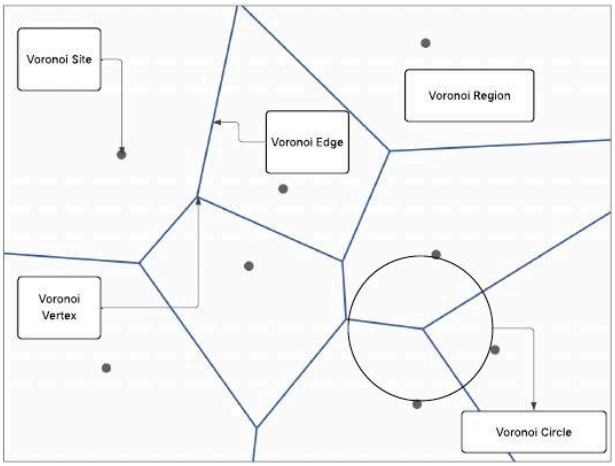
Introduction / Problem Statement

Voronoi diagrams, also known as Dirichlet patterns or tessellations, are diagrams where a Euclidean space is partitioned into regions, known as Voronoi cells or Thiessen polygons, about certain points, sites. They have been used, informally, as early as 1644 by philosopher René Descartes, though they were named after Russian mathematician Georgy Voronoi, who actually defined them and studied the general n-dimensional case in 1908. Voronoi diagrams have various applications in science, mathematics, and art, such as modeling animal territories or crystal lattice growth—famously, physician John Snow used a Voronoi diagram, with the location of water pumps as the points, to discover the sources of infection during the 1854 London cholera epidemic. Despite their uses, however, before Steven Fortune wrote his paper detailing Fortune's Construction in 1987, Voronoi diagrams were generally created using either slow—specifically $O(n^2)$ runtime, where n is the number of points—incremental algorithms that go over all points in the plane and assign them to the nearest site or somewhat complex merge-and-conquer algorithms that break the plane down into more manageable sections but have complicated merges. With his method, however, Fortune was able to provide a means of producing Voronoi diagrams that combined the efficiency of the merge-and-conquer while still retaining the relative simplicity of the incremental to construct Voronoi diagrams with O(n log n) time complexity and O(n) space complexity.

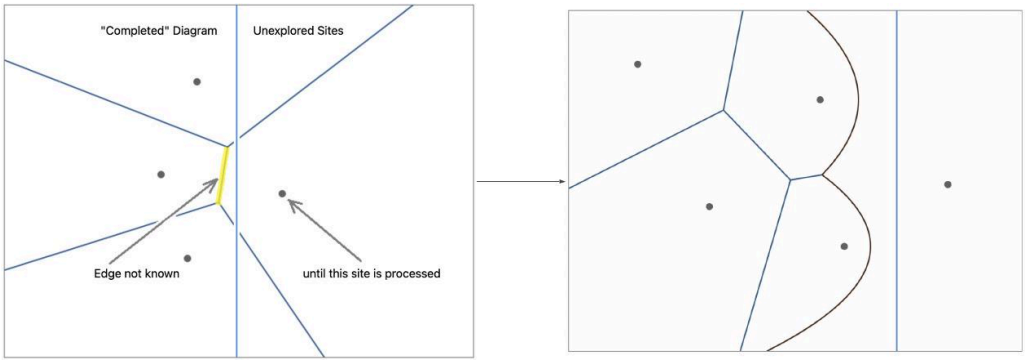
Overall Intuitive Approach / Solution:

To understand how the Fortune's Construction Algorithm works, various aspects of Voronoi diagrams need to be defined first. The following are all definitions of different aspects of a Voronoi diagram. A Voronoi site is the point from which a region is constructed. A Voronoi region is a (possibly) infinite convex polygon where all the points in the region are closer to the region's site compared to any other site in the plane. A Voronoi vertex is where 3 or more Voronoi regions meet, and a Voronoi circle is a circle in the diagram where 3 sites are on the edge and a Voronoi vertex is in the middle. A Voronoi circle does not contain any sites. A

Voronoi edge is a line segment consisting of points that are equidistant to the sites, forming the boundary between two adjacent regions.

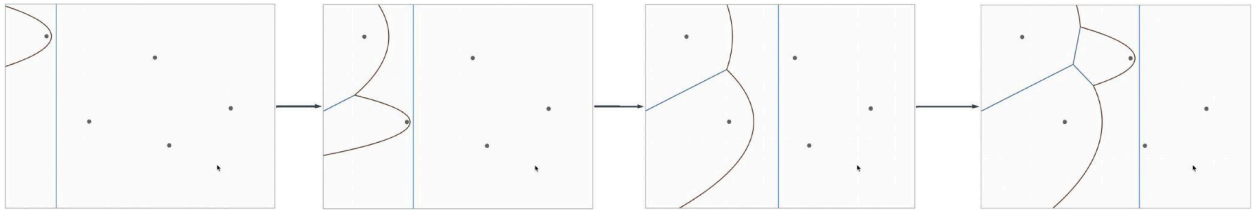


The Fortune's Construction algorithm is a line sweep algorithm that works in O(nlogn) time where a line sweeps from left to right or up to down over the site points and constructs a Voronoi diagram. A traditional sweep line algorithm, like the one shown below constructs edges before they are fully certain which leads to inefficiencies and having to recompute edges of regions after a new site is discovered by the sweep line. Therefore, Steven Fortune came up with a more efficient way to process these sites, using two lines: a sweep line and a beach line. The algorithm also instead of sweeping points it sweeps events, specifically site events and circle events.

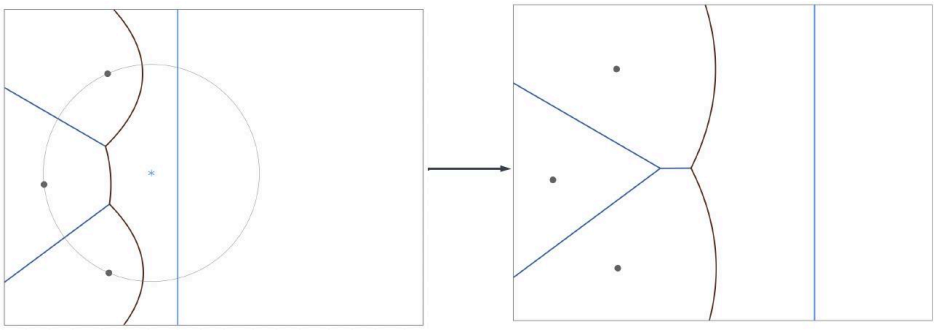


The beachline consists of a series of parabolic arcs such that any point on the beachline is equidistant from its nearest site and the sweep line. As the sweep line passes over each site a parabola with the focus of the site point is added to the beach line. The points where one arc intersects another arc are called breakpoints and as these intersections move with the beach line

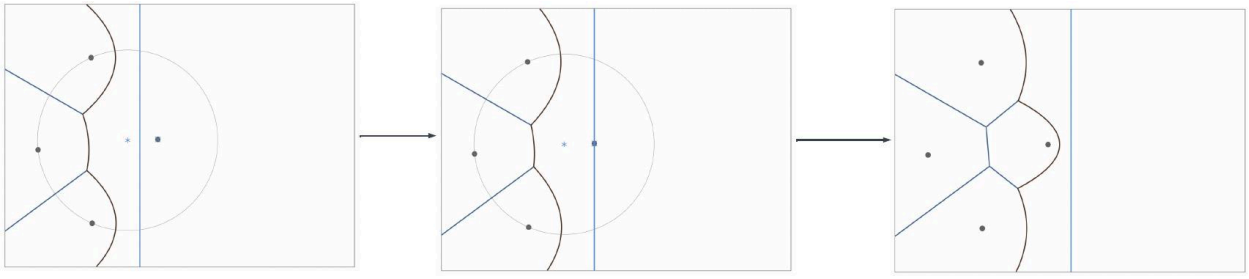
they form the Voronoi edges of the regions. The below picture shows how the sweep line goes through and processes these site events. Site events are stored before runtime in a priority queue since they are location dependent, we know when they are going to happen beforehand (stored by x-coordinate in the example below).



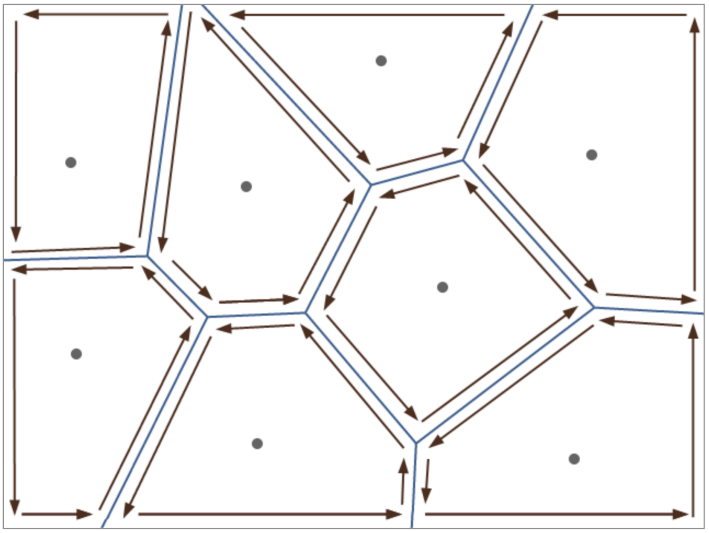
Circle events are detected at runtime. These occur when 3 arcs converge at a point, so basically two breakpoints meet. This convergent point then becomes a Voronoi vertex if it is valid. The below images show the 3 arcs converging, a Voronoi vertex being formed and the middle arc being deleted.



If a site is discovered by the sweep line within the circle then a new arc is added to the beachline and the circle event is cancelled. In the example below, the sweep line encounters a site event before the right edge of the circle so, the circle event is cancelled and a site event is processed instead.



Each edge that is formed from the processing of these site events is stored in a doubly connected edge list where each edge is split into "half" edges where each half edge points in opposite directions. The edges also store their successor and predecessor, which justifies the name doubly connected edge list. Vertices are stored by their coordinate in the plane, with a pointer to a connecting edge.



Implementation:

Below is the complete loop for the Fortune's Construction algorithm. There are several data structures that were implemented to make Fortune's Construction possible and they all serve different purposes within the algorithm.

```
# Initialization
initialize data structures
add site events to Event Heap

# Algorithm
while Event Heap is not empty:
    pop event off Event Heap
    if event is a site event:
        add arc node to Beachline Tree
        add half-edges to Edge Set
        cancel any relevant circle events
        check for circle events with neighboring arcs in tree and add to Event Heap
    else:
        remove arc node from Beachline Tree
        complete half-edges in Edge Set
        check for circle events with neighboring arcs in tree and add to Event Heap

# Postprocessing
for each half-edge in Edge Set, determine if half-edge is complete or needs to be extended
construct polygons out of the half edges
```

To store the site events and circle events, our implementation uses a max Heap. Each element in the heap is an Event object like the one defined below. The custom heap data structure we

implemented also includes a custom priority implementation where higher “y” values are given priority and circle events are given priority over non circle events. In the case of a within events and y value the priority is given to the lower x-value. This comparison implementation is below as well (compare method part of a different Event Heap class).

```
class Event:
    def __init__(self, category, site: Point=None, leaf: Arc_node=None, circle_bottom=None):
        self.category = category
        self.index = None
        if self.category == 'site':
            self.site = site
            self.key = site.y
        elif self.category == 'circle':
            self.leaf = leaf
            self.key = circle_bottom
```

```
def compare(self, a, b):
    a_event, b_event = self.heap[a], self.heap[b]
    if a_event.key > b_event.key:
        return True
    elif a_event.key < b_event.key:
        return False
    elif a_event.category == 'circle' or b_event.category == 'circle':
        return True
    return a_event.site.x < b_event.site.x
```

In addition to the event heap we also have to initialize a data structure for the beach line. The beach line uses a BST that consists of breakpoints as the internal nodes and arcs as the leaf nodes. The Beachline BST also contains a find_arc method that is used in processing site events. This find_arc method is called when a new site event is being processed and the site's parabola needs to be added to the beach line. Since the breakpoints are the internal nodes, it compares the sites x value to the breakpoint’s x value. Based on if it's on the left or right of the breakpoint or not it traverses that path of the tree. When the traversal reaches a leaf node, the arc is split and the new arc is added as well as a breakpoint splitting them. Below the arc node and breakpoint node are defined as well as the find_arc method from the Beachline BST.

```
class Arc_node:
    def __init__(self, site, parent=None, circle_event=None,
edge=None):
        self.parent = parent
        self.site = site
        self.circle_event = circle_event
        self.edge = edge
        self.left, self.right = None, None
```

```
class Breakpoint_node:
    def __init__(self, old_site, new_site, parent=None,
edge=None):
        self.parent = parent
        self.old_site = old_site
        self.new_site = new_site
        self.edge = edge
        self.left, self.right = None, None


     def find_arc(self, site: Point):
         node = self.root
         while type(node) == Breakpoint_node:
             intersection = intersect(node.old_site,
 node.new_site, site.y)
             if site.x < intersection.x:
                 node = node.left
             else:
                 node = node.right
         return node
```

The beachline tree is also instrumental in handling circle events. Below is the pseudocode for handling the circle events and adjusting the tree after a circle event is processed. The process of handling a circle event is such that the arc that collapses is removed along with the breakpoints it formed with the two arcs to its left and right. Then a new breakpoint with the left and right arcs is added to the tree and the left and right arcs become children of this breakpoint.

```
def handle_circle_event(event):
   #L is the event arc that shrinks
   L = event.arc
   A = tree.predecessor_leaf(L)
   B = tree.successor_leaf(L)

   bp1 = L.parent
   bp2 = tree.successor_node(L)

   #remove the old pieces
   tree.remove(bp1)
   tree.remove(bp2)
   tree.remove(L)

   # create & splice in the new breakpoint between A and B
   new_bp = Breakpoint_node(A.site, B.site)
   tree.replace_parent(new_bp)

   #hang the two arcs under it
   tree.insert_left(new_bp,  A)
   tree.insert_right(new_bp, B)
```

From the overview of the algorithm in the previous section, it is understood that the breakpoints form the edges of each Voronoi region and in the actual implementation of the algorithm the concept of an Edge set as well as a half edge are used to efficiently store the edges produced by the sweep. Each half edge acts like a node in a linked list where it points to the next edge and has a previous edge pointing to it. These edges are doubly linked and also have access to information about their twin, which is the other side of the Voronoi edge. Using half edges allows you to loop through the edge set and follow the edges across the diagram, making it easier to construct the polygons. Edges are added to the edge set when site and circle events are processed, using the breakpoints created or retained from each event. After the sweep, the edge set is processed. Edges are clipped to fit inside the max and min axes and then an edge adjacency graph is built, where a map of edges from one vertex to the next is constructed. The polygons are constructed from this adjacency graph using the farthest left turn method and then assigned to each site.

```
class Half_edge:
    def __init__(self):
        self.point = None
        self.origin = None
        self.vector = None
        self.prev, self.next = None, None
        self.twin = None
```

```
class Edge_Set:
    def __init__(self):
        self.edges = set()

    def add_edge(self, point: Point):
        edge = Half_edge()
        edge.twin = Half_edge()
        edge.twin.twin = edge

        edge.point = point
        edge.twin.point = point

        self.edges.add(edge)
        self.edges.add(edge.twin)
        return edge
```

Summary of Programming Challenge

The programming challenge we designed has the student take a set of site points and another set of points of interest and return the x,y coordinates of the site points in order of which has the most points of interest closest to it. While this initially seems simple and solvable via the naive approach of processing each point of interest and assigning them to their nearest site point, with a higher number of sites or points of interest and runtime limitations, the student is forced to find a more efficient method, such as using Fortune's Construction to capture the Voronoi cells of each site point assign the points of interest to the site points from there. One of the learning objectives of the programming challenge is to realize this. Furthermore, when constructing Voronoi diagrams, the output is typically plotted and used visually, but for this programming challenge the additional twist beyond implementing Fortune's Construction lies in the fact that the student must also find the number of points of interest in each Voronoi cell to output the sites in the correct order. There are many ways to do this, and we felt it was a reasonable twist since there are several interesting speedups the student could apply here to make their solution more efficient. It also presents the learning objective of exploring how to actually use the output for Fortune's Construction.

Key Ideas for Solving Programming Challenge:

The main challenge to overcome with the programming challenge is, first and foremost, understanding and implementing Fortune's Construction, since it is a fairly complex and geometrically involved algorithm. The additional twist requires students to check the number of points inside the Voronoi cells. One efficient way to do this would be to implement some of the computational geometry previously discussed in class, though, in order to use Fortune's Construction with those algorithms, students must understand how to get the cells of the Voronoi diagram produced as sets of points making up polygons. This twist is not necessarily complicated, and the model solution was able to easily use lines and triangles to check if the points are within a certain cell, but it does allow students to apply their newfound knowledge of Fortune's Construction with established knowledge of computational geometry.

Conclusion

Overall, Fortune's Construction is a faster, more efficient way of constructing a Voronoi Diagram given a set of sites. Fortune's Construction runs in O(n log n) time compared to n-squared run times for other methods of constructing Voronoi Diagrams. When creating the algorithm, Steven Fortune came up with a unique approach of using a beach line to keep track of visited sites and the portion of the plane they occupy. The algorithm also uses an innovative approach to processing the points in the sweep by considering events for when it meets a site or when certain criteria are met from the beachline. This approach combined with the use of a tree for the beachline and a heap/priority queue for the events allows for this algorithm to be considerably faster than the ones that came before it. We applied this algorithm in our challenge by asking students to return the site point which was nearest to the most points of interest, which is the same as asking which Voronoi cell in a diagram contains the most points of interest and so has students implement Fortune's Construction for themselves as well as performing some computational geometry to find the number of points.

Sources cited

What is a Voronoi diagram? | school of mathematics | university of Bristol. University of Bristol. (n.d.). https://www.bristol.ac.uk/maths/fry-building/public-art-strategy/what-is-a-voronoi-diagram/

Voronoi Diagram construction using Fortune's Algorithm. Tufts University ECE and CS Departments. (n.d.). https://www.eecs.tufts.edu/~vporok01/c163/