DYNAMIC VORONOI MAPPER

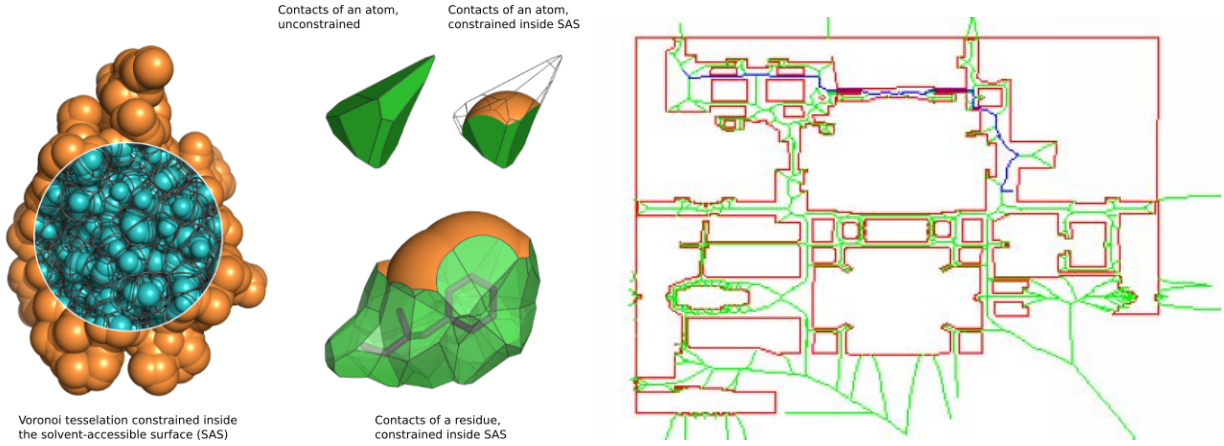
ED5310
ALGORITHMS IN COMPUTATIONAL GEOMETRY

1 Problem Statement

In the realm of computational geometry, Voronoi diagrams stand as foundational structures that partition a given space into distinct regions based on their proximity to a set of defined points. These diagrams find applications across various domains, including computer graphics, geographical information systems, and pattern recognition.

This project aims to delve into the dynamic and evolving nature of Voronoi diagrams, extending the conventional concept to accommodate continuously moving points within a 2D plane. Unlike traditional Voronoi diagrams, where static points define fixed regions, this endeavor seeks to create a dynamic representation that adapts to the continuous movement of points over time.

Understanding Voronoi diagrams in a dynamic context introduces novel perspectives to their practical applications. This research is poised to contribute valuable insights into the adaptability and responsiveness of Voronoi diagrams when subjected to continuously changing spatial configurations. The findings of this study could have implications in fields such as robotics, computational biochemistry, sensor networks, and dynamic spatial analysis, where real-world entities exhibit continuous movement.



(a) Computational Biochemistry (b) Robot Motion Planning

Figure 1: Applications of Voronoi Diagrams

Voronoi diagrams serve as the focal point of the project, echoing the in-depth study of algorithms, including incremental and Fortune's algorithm, witnessed during the course. The project investigates the implications of dynamic point movement on these fundamental geometric structures. While the course extensively covered Fortune's algorithm for Voronoi diagram construction, one of the many approaches to Fortune's algorithm was learned. Instead of employing the conventional approach of cones and intersections, the project embraces the parabolic front algorithm, offering a distinctive perspective on dynamic Voronoi diagrams.

Keywords:*Voronoi Diagrams, Dynamic Spatial Analysis, Continuous Point Movement, Computational Geometry.*

2 Related work

The field of computational geometry has witnessed substantial advancements in the implementation of Voronoi diagrams, both in terms of efficiency and ease of use. Traditionally, C++ has been the base language for implementing intricate geometric algorithms, and Fortune's algorithm utilizing the parabolic front technique is no exception. Researchers have contributed robust and optimized C++ implementations, providing a comprehensive tool set for Voronoi diagram construction.

Concurrently, the Python programming language offers an extensive set of libraries. Notably, the **scipy** library incorporates an in-built function for Voronoi diagram computation. This integration simplifies the implementation process, providing users with a convenient and accessible tool for Voronoi diagram generation.

The dynamic nature of Voronoi diagrams introduces an additional layer of complexity to their computational landscape. While conventional Voronoi diagrams have been extensively studied, dynamic variations have primarily focused on scenarios where a single point's movement triggers changes in individual Voronoi faces. These studies provide valuable insights into the localized effects of point movement, but fall short of capturing the broader dynamics of the entire diagram. This limitation underscores an opportunity for further exploration and development in the field.

3 Address Challenges

In this research undertaking, a fundamental challenge confronted is the creation of a dynamic Voronoi diagram capable of accommodating the continuous movement of multiple points within a 2D plane. Existing studies predominantly focus on the local adaptations of Voronoi faces concerning a single point's movement. This study tries to tackle the broader dynamics of Voronoi regions influenced by the concurrent, continuous movement of multiple points.

Before addressing that, I have also included a dedicated Python implementation of Fortune's algorithm, utilizing the parabolic front methodology. This algorithmic strategy employs parabolas to model the evolving Voronoi diagram. This Python implementation looks into the capabilities of the **scipy** library to implement the Voronoi diagram algorithm from scratch using classes and functions.

4 Methodology

The proposed method aims to extend the traditional Voronoi diagrams into a dynamic realm where multiple points move continuously within a 2D plane. This algorithmic approach combines the elegance of Fortune's algorithm with the adaptability of the parabolic front technique to capture the evolving spatial relationships as points traverse the plane.

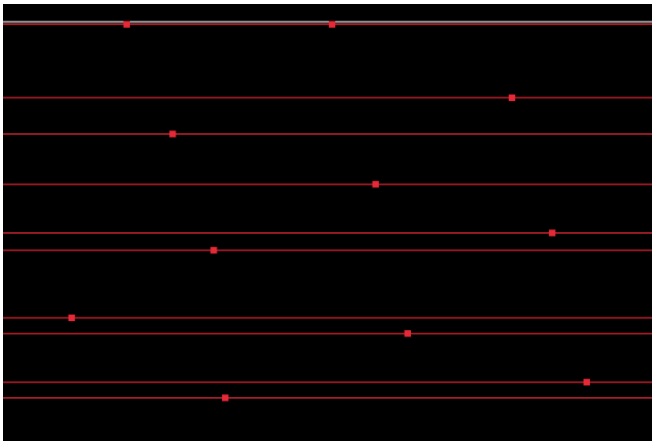
4.1 Fortune's Algorithm

The algorithm commences with the selection of a set of points, each serving as a site in the Voronoi diagram. These initial points act as anchors, fundamentally shaping the evolving structure of the diagram. The algorithm sets up the groundwork by establishing these crucial points, setting the stage for the dynamic adaptation that follows. This algorithm follows a sweeping algorithm.

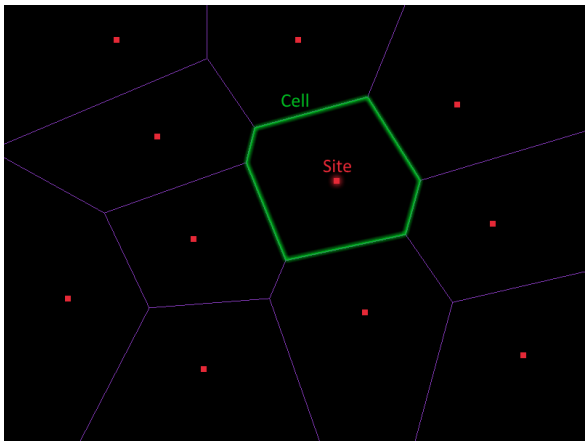
The initial points are first sorted based on their heights from bottom to top in descending order, i.e., the higher y-coordinate points precede the lower y-coordinate points. This is implemented through a priority queue implementation. Given a point and a line, the voronoi diagram for this pair will be a parabola. Here, the site becomes the focus, and the sweep line becomes the directrix. An important point to note is that while plotting from top to bottom, the voronoi diagram above the parabola will not undergo any change. 2

This algorithm makes use of a concept called the **beach line**. Given the "n" number of sites, defining a parabola for each of the sites while considering the sweep line as the directrix gives rise to a sequence of parabolas overlapping laterally. The sequence obtained from the intersection of parabolas constitutes the beach line. The beach line is monotonic along the x dimension. Every vertical line intersects the beach line at exactly one point.Breakpoints on the beach line or intersections between two parabolic arcs indicate their presence on the Voronoi diagram.

The beach line is a data structure that represents the current configuration of parabolic arcs as the sweep line progresses. It is maintained as a priority queue. The beach line helps manage the order and intersection points of the parabolic arcs, ensuring efficient handling of events during the sweep. As the sweep line progresses, it interacts with the beach line, triggering site and circle events. The beach line adjusts to the changing configuration of parabolic arcs. 6

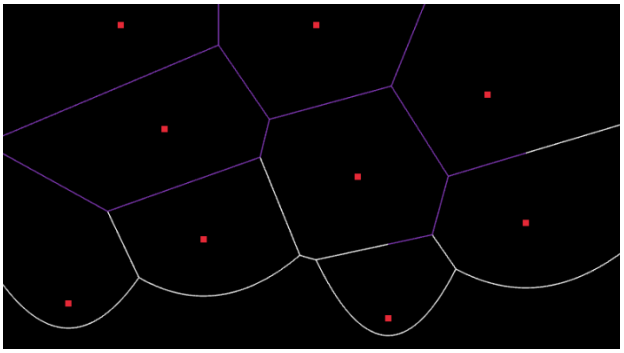


(a) The sorting of initial points in descending y-coordinates

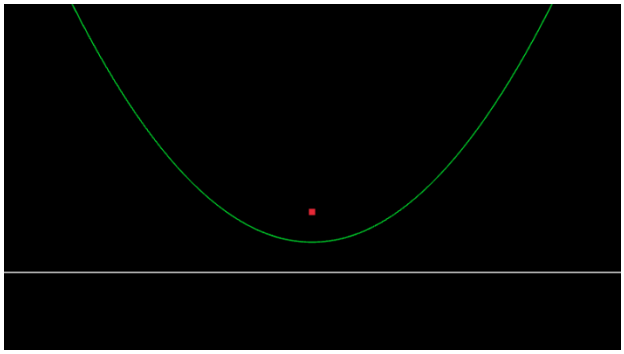


(b) Typical completely generated Voronoi Diagram

Figure 2: Initialization of Voronoi Diagrams



(a) The beach line generated w.r.t. the sweep line at bottom



(b) A voronoi diagram between a parabola and sweep line

Figure 3: Beach Line Representation

The algorithm processes site events through this algorithm. When the vertical sweep line encounters a new site during the execution of Fortune's algorithm, a meticulous process ensues to integrate the new point into the evolving Voronoi diagram. The algorithm determines the existing parabolic arc under which the new site falls. This is achieved through a binary search in the beach line data structure. The identified arc is split into two segments, with the new site positioned between them. The two new edges are introduced as a result of the site event. The identified arc fragment is removed from the beach line, and in its place, the sequence of the new beach line item is inserted. 4

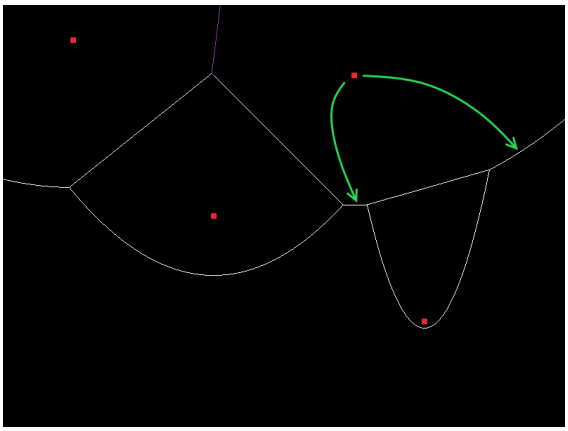
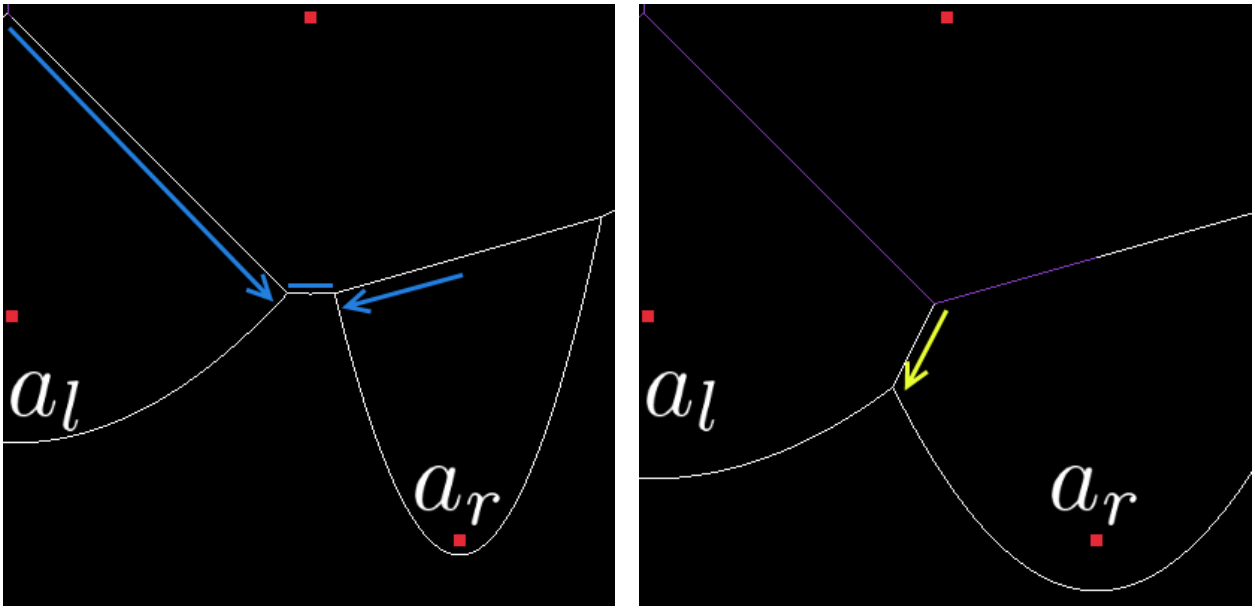


Figure 4: Site event handling

Circle events are another class of events in Fortune's algorithm that are identified when three consecutive sites along the beach line form a circle. These three sites, along with the current position of the sweep line, contribute to the creation of a circle event. Upon detecting a circle event, the algorithm proceeds to remove the associated parabolic arc from the beach line. This removal is a consequence of the circle event indicating the formation of a Voronoi vertex where three parabolic arcs intersect.



(a) Event status just prior to circle event

(b) Beyond the circle event, a voronoi vertex is obtained

Figure 5: Representation of the Circle Event

4.2 Dynamic Voronoi Diagram

The dynamic voronoi diagram is initiated with a set of moving points in a 2D plane. Each point, represented as a dot object, is assigned attributes such as position, color, radius, and initial direction. The animation framework is established using certain in-built libraries, details of which will be covered later. A canvas is created, and the animation proceeds through a specified number of frames. At each frame, the update function is invoked, orchestrating the dynamic movement of dots.

Dots are programmed to move continuously in the plane, with randomized changes in direction to introduce natural-looking dynamics. Boundary conditions are implemented to ensure that dots stay within the defined canvas. The algorithm provides a dynamic and visually appealing tool for exploring spatial partitioning in response to the movement of points in a 2D plane.

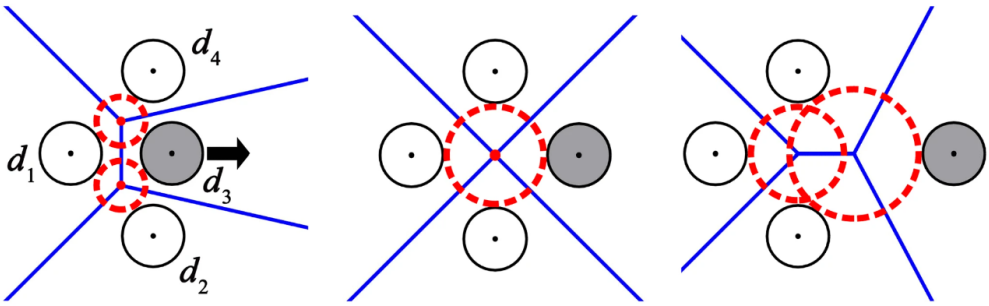


Figure 6: Representation of the Circle Event

5 Code Demonstration

5.1 Brute Force Voronoi Diagram

```
for alpha in range(10):
    def generate_voronoi_diagram(width, height, num_cells):

        image = Image.new("RGB", (width, height), color=(255,255,255))
        putpixel = image.putpixel
        imgx, imgy = image.size
        nx, ny, nr, ng, nb = [], [], [], [], []

        for i in range(num_cells):
            nx.append(random.randrange(imgx))
            ny.append(random.randrange(imgy))
            nr.append(random.randrange(256))
            ng.append(random.randrange(256))
            nb.append(random.randrange(256))

        plt.plot(nx,ny,"ko")
        plt.imshow(image)
        plt.show()

        for y in range(imgy):
            for x in range(imgx):
                dmin = math.hypot(imgx-1, imgy-1)
                j = -1
                for i in range(num_cells):
                    d = math.hypot(nx[i]-x, ny[i]-y)
                    if d < dmin:
                        dmin = d
                        j = i
                putpixel((x, y), (nr[j], ng[j], nb[j]))

        plt.plot(nx,ny,"ko")
        plt.imshow(image)
        plt.title("VORONOI_DIAGRAM")
        plt.savefig("path_to_save_the_figure")
        plt.show()

    generate_voronoi_diagram(500, 500, 25)
```

The presented code aims to generate a Voronoi diagram using a brute-force approach, demonstrating a simple yet effective method for spatial partitioning in a 2D plane. The algorithm initialized a blank RGB image with a specified width and height, where each pixel represents a potential point in the Voronoi diagram. Random cell coordinates and colors are generated to serve as the Voronoi sites. The initial cell points are plotted on the canvas.

The code iterates through each pixel in the image, treating it as a potential point in the Voronoi diagram. For each pixel, it calculates the Euclidean distance to each Voronoi cell and identifies the closest cell. The color of each pixel is set based on the RGB values of the closest cell. This process creates distinct regions in the Voronoi diagram associated with each cell. The final Voronoi diagram is displayed, with Voronoi cell points overlaid for clarity. The code includes functionality to generate multiple examples by varying parameters such as width, height, and number of cells. Each example is saved as an image file with a unique identifier.

The presented code provides a straightforward implementation of Voronoi diagram generation through a brute-force algorithm. While not optimized for large-scale diagrams, it serves as a valuable educational tool and a basis for understanding Voronoi diagram concepts.

5.2 Fortune's algorithm for Voronoi Diagram

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

class Event:
    valid = True
    def __init__(self, x=0.0, p=None, a=None):
        self.x = x
        self.p = p
        self.a = a
        self.valid = True

class Arc:
    pprev, pnext = None, None
    e = None
    s0, s1 = None, None
    def __init__(self, p=None, a=None, b=None):
        self.p = p
        self.pprev = a
        self.pnext = b
        self.e = None
        self.s0 = None
        self.s1 = None

class Segment:
    start, end = None, None
    done = False
    def __init__(self, p):
        self.start = p
        self.end = None
        self.done = False

    def finish(self, p):
        if self.done:
            return
        self.end = p
        self.done = True
```

The **Point** class represents a two-dimensional point in the Cartesian plane. Each point is defined by its x and y coordinates. In the context of Fortune's algorithm, instances of this class serve as fundamental entities for specifying the locations of events, which are essential for constructing the Voronoi diagram. The x and y attributes encapsulate the coordinates of the point, providing a structured representation of geometric positions during the algorithm's execution.

The **Event** class is pivotal in Fortune's algorithm, managing site and circle events. Site events occur when the sweep line encounters a new point, leading to the creation of a new parabolic arc. Circle events mark the formation of a circle between three consecutive sites, resulting in the removal of a parabolic arc. This class plays a crucial role in orchestrating the dynamic evolution of the Voronoi diagram.

The **Arc** class represents parabolic arcs within the beach line in Fortune's algorithm, a key structure in Voronoi diagram construction. It includes attributes for the associated point, the previous and next arcs in the beach line, an associated circle event, and two segments representing Voronoi edges. This class facilitates the dynamic adjustment of the beach line and manages relationships between arcs, circle events, and associated Voronoi segments.

The **Segment** class embodies segments or edges in the evolving Voronoi diagram. Attributes include start and end points and a flag indicating whether the segment is complete. This class is vital for tracking the dynamic edges of the Voronoi diagram throughout the algorithm's execution, contributing to the final representation of Voronoi regions based on the movements of input points.

```
class PriorityQueue:
    def __init__(self):
        self.pq = []
        self.entry_finder = {}
        self.counter = itertools.count()

    def push(self, item):
        if item in self.entry_finder:
            return
        count = next(self.counter)
        entry = [item.x, count, item]
        self.entry_finder[item] = entry
        heapq.heappush(self.pq, entry)

    def remove_entry(self, item):
        entry = self.entry_finder.pop(item)
        entry[-1] = "Removed"

    def pop(self):
        while self.pq:
            priority, count, item = heapq.heappop(self.pq)
            if item != "Removed":
                del self.entry_finder[item]
                return item
        raise KeyError('Empty Priority Queue')

    def top(self):
        while self.pq:
            priority, count, item = heapq.heappop(self.pq)
            if item != "Removed":
                del self.entry_finder[item]
                self.push(item)
                return item
        raise KeyError('Empty Priority Queue')

    def empty(self):
        return not self.pq
```

The **PriorityQueue** class is a crucial component in Fortune's algorithm, implementing a priority queue using a min-heap structure. It efficiently manages site and circle events based on their x-coordinates along the sweep line. The class utilizes a heap, a dictionary *entryfinder*, and a counter to maintain unique priorities for events. The push method inserts a new item, preserving order, while *removeentry* eliminates irrelevant entries. The pop method retrieves the highest-priority event, and top allows inspection without removal. The empty method checks for an empty priority queue, ensuring efficient event handling in Voronoi diagram construction.

The **Voronoi** class encapsulates the intricate workings of Fortune's algorithm, meticulously constructing Voronoi diagrams with dynamic precision. The initialization involves setting up a priority queue for site events and circle events and defining the initial bounding box based on the input points. The algorithm processes these events, creating a parabolic front represented by a linked list of arcs. Each event triggers specific actions; site events introduce new parabolic arcs, while circle events signify the birth of new Voronoi edges.

The *arcinsert* method meticulously integrates a new point into the evolving parabolic front, adjusting the structure and creating new edges. Circle events are detected and processed through the *checkcircleevent* method, ensuring the diagram adapts to changing configurations. The circle and intersect methods intricately compute circle events and intersections, contributing to the continuous evolution of the Voronoi diagram.

As the algorithm progresses, the *finishedges* method ensures the completion of edges, providing a visually coherent representation of the Voronoi diagram. The resulting diagram can be obtained through the *getoutput* method, encapsulating the coordinates of line segments that delineate the Voronoi regions. The *printoutput* method offers a textual representation of these coordinates, allowing for a comprehensive understanding of the constructed Voronoi diagram.

```
class Voronoi:

    def __init__(self, points):
        self.output = []
        self.arc = None
        self.points = PriorityQueue()
        self.event = PriorityQueue()
        self.x0 = -50.0
        self.x1 = -50.0
        self.y0 = 550.0
        self.y1 = 550.0
        for pts in points:
            point = Point(pts[0], pts[1])
            self.points.push(point)
            if point.x < self.x0:  self.x0 = point.x
            if point.y < self.y0:  self.y0 = point.y
            if point.x > self.x1:  self.x1 = point.x
            if point.y > self.y1:  self.y1 = point.y
        dx = (self.x1 - self.x0 + 1) / 5.0
        dy = (self.y1 - self.y0 + 1) / 5.0
        self.x0, self.x1 = self.x0 - dx, self.x1 + dx
        self.y0, self.y1 = self.y0 - dy, self.y1 + dy

    def process(self):
        while not self.points.empty():
            if not self.event.empty() and (self.event.top().x <=
                                           self.points.top().x):
                self.process_event()
            else:
                self.process_point()

        while not self.event.empty():
            self.process_event()

        self.finish_edges()

    def process_point(self):
        p = self.points.pop()
        self.arc_insert(p)

    def process_event(self):
        e = self.event.pop()
        if e.valid:
            s = Segment(e.p)
            self.output.append(s)

            a = e.a
            if a.pprev is not None:
                a.pprev.pnext = a.pnext
                a.pprev.s1 = s
            if a.pnext is not None:
                a.pnext.pprev = a.pprev
                a.pnext.s0 = s

            if a.s0 is not None: a.s0.finish(e.p)
            if a.s1 is not None: a.s1.finish(e.p)

            if a.pprev is not None: self.check_circle_event(a.pprev, e.x)
            if a.pnext is not None: self.check_circle_event(a.pnext, e.x)
```

```
def arc_insert(self, p):
    if self.arc is None:
        self.arc = Arc(p)
    else:
        i = self.arc
        while i is not None:
            flag, z = self.intersect(p, i)
            if flag:
                flag, zz = self.intersect(p, i.pnext)
                if (i.pnext is not None) and (not flag):
                    i.pnext.pprev = Arc(i.p, i, i.pnext)
                    i.pnext = i.pnext.pprev
                else:
                    i.pnext = Arc(i.p, i)
                i.pnext.s1 = i.s1

                i.pnext.pprev = Arc(p, i, i.pnext)
                i.pnext = i.pnext.pprev

                i = i.pnext # now i points to the new arc

                seg = Segment(z)
                self.output.append(seg)
                i.pprev.s1 = i.s0 = seg

                seg = Segment(z)
                self.output.append(seg)
                i.pnext.s0 = i.s1 = seg

                self.check_circle_event(i, p.x)
                self.check_circle_event(i.pprev, p.x)
                self.check_circle_event(i.pnext, p.x)
                return
            i = i.pnext

        i = self.arc
        while i.pnext is not None:
            i = i.pnext
        i.pnext = Arc(p, i)
        x = self.x0
        y = (i.pnext.p.y + i.p.y) / 2.0;
        start = Point(x, y)

        seg = Segment(start)
        i.s1 = i.pnext.s0 = seg
        self.output.append(seg)

def check_circle_event(self, i, x0):
    if (i.e is not None) and (i.e.x  != self.x0):
        i.e.valid = False
    i.e = None

    if (i.pprev is None) or (i.pnext is None): return

    flag, x, o = self.circle(i.pprev.p, i.p, i.pnext.p)
    if flag and (x > self.x0):
        i.e = Event(x, o, i)
        self.event.push(i.e)
```

```
def circle(self, a, b, c):
    if ((b.x - a.x)*(c.y - a.y) - (c.x - a.x)*(b.y - a.y)) > 0:
        return False, None, None
    A = b.x - a.x
    B = b.y - a.y
    C = c.x - a.x
    D = c.y - a.y
    E = A*(a.x + b.x) + B*(a.y + b.y)
    F = C*(a.x + c.x) + D*(a.y + c.y)
    G = 2*(A*(c.y - b.y) - B*(c.x - b.x))

    if (G == 0): return False, None, None

    ox = 1.0 * (D*E - B*F) / G
    oy = 1.0 * (A*F - C*E) / G

    x = ox + math.sqrt((a.x-ox)**2 + (a.y-oy)**2)
    o = Point(ox, oy)
    return True, x, o

def intersect(self, p, i):
    if (i is None): return False, None
    if (i.p.x == p.x): return False, None
    a = 0.0
    b = 0.0
    if i.pprev is not None:
        a = (self.intersection(i.pprev.p, i.p, 1.0*p.x)).y
    if i.pnext is not None:
        b = (self.intersection(i.p, i.pnext.p, 1.0*p.x)).y

    if (((i.pprev is None) or (a <= p.y)) and
        ((i.pnext is None) or (p.y <= b))):
        py = p.y
        px = 1.0 * ((i.p.x)**2 +
             (i.p.y-py)**2 - p.x**2) / (2*i.p.x - 2*p.x)
        res = Point(px, py)
        return True, res
    return False, None

def intersection(self, p0, p1, l):
    p = p0
    if (p0.x == p1.x):  py = (p0.y + p1.y) / 2.0
    elif (p1.x == l):   py = p1.y
    elif (p0.x == l):   py = p0.y   p = p1
    else:
        z0 = 2.0 * (p0.x - l)
        z1 = 2.0 * (p1.x - l)

        a = 1.0/z0 - 1.0/z1;
        b = -2.0 * (p0.y/z0 - p1.y/z1)
        c = 1.0 * (p0.y**2 + p0.x**2 - l**2) / z0 -
            1.0 * (p1.y**2 + p1.x**2 - l**2) / z1

        py = 1.0 * (-b-math.sqrt(b*b - 4*a*c)) / (2*a)

    px = 1.0 * (p.x**2 + (p.y-py)**2 - l**2) / (2*p.x-2*l)
    res = Point(px, py)
    return res
```

```
    def finish_edges(self):
        l = self.x1 + (self.x1 - self.x0) + (self.y1 - self.y0)
        i = self.arc
        while i.pnext is not None:
            if i.s1 is not None:
                p = self.intersection(i.p, i.pnext.p, l*2.0)
                i.s1.finish(p)
            i = i.pnext

    def print_output(self):
        it = 0
        for o in self.output:
            it = it + 1
            p0 = o.start
            p1 = o.end
            print (p0.x, p0.y, p1.x, p1.y)

    def get_output(self):
        res = []
        for o in self.output:
            p0 = o.start
            p1 = o.end
            res.append((p0.x, p0.y, p1.x, p1.y))
        return res
```

5.3 Plotting Voronoi using GUI

The provided code establishes a graphical user interface (GUI) for interactively creating and visualizing Voronoi diagrams. Using the **Tkinter** library, the **MainWindow** class initializes a window with a canvas for drawing points. The *onClickCalculate* method triggers the computation of the Voronoi diagram once the user clicks the "Calculate" button. It collects the drawn points on the canvas, converts them into a format suitable for the Voronoi computation, and then invokes the Voronoi class to generate the diagram. The resulting Voronoi lines are displayed on the canvas, providing a visual representation of the Voronoi regions.

The GUI also incorporates a "Clear" button, handled by the *onClickClear* method, allowing users to reset the canvas and redraw points. The *onDoubleClick* method responds to double-click events on the canvas, enabling users to place points by adding black circles to mark their positions. Overall, this GUI-driven Voronoi diagram plotting tool enhances user interaction by dynamically computing and displaying Voronoi diagrams based on user-input points, offering a user-friendly approach to exploring geometric structures.

```
import tkinter as tk
class MainWindow:
    RADIUS = 3
    LOCK_FLAG = False

    def __init__(self, master):
        self.master = master
        self.master.title("Voronoi")

        self.frmMain = tk.Frame(self.master, relief=tk.RAISED, borderwidth=1)
        self.frmMain.pack(fill=tk.BOTH, expand=1)

        self.w = tk.Canvas(self.frmMain, width=500, height=500)
        self.w.config(background='white')
        self.w.bind('<Double-1>', self.onDoubleClick)
        self.w.pack()

        self.frmButton = tk.Frame(self.master)
```

```
        self.frmButton.pack()

        self.btnCalculate = tk.Button(self.frmButton, text='Calculate',
                                      width=25, command=self.onClickCalculate)
        self.btnCalculate.pack(side=tk.LEFT)

        self.btnClear = tk.Button(self.frmButton, text='Clear',
                                  width=25, command=self.onClickClear)
        self.btnClear.pack(side=tk.LEFT)

    def onClickCalculate(self):
        if not self.LOCK_FLAG:
            self.LOCK_FLAG = True

            pObj = self.w.find_all()
            points = []
            for p in pObj:
                coord = self.w.coords(p)
                points.append((coord[0]+self.RADIUS, coord[1]+self.RADIUS))

            for i in range(len(points)):
                plt.plot(points[i][0], points[i][1], marker="o", markersize=20)
            plt.show()

            vp = Voronoi(points)
            vp.process()
            lines = vp.get_output()
            self.drawLinesOnCanvas(lines)

            for i in range(len(lines)):
                plt.plot([lines[i][0], lines[i][1]], [lines[i][2],
                          lines[i][3]], marker="o", markersize=i+5)
            plt.show()

            print (lines)

    def onClickClear(self):
        self.LOCK_FLAG = False
        self.w.delete(tk.ALL)

    def onDoubleClick(self, event):
        if not self.LOCK_FLAG:
            self.w.create_oval(event.x-self.RADIUS, event.y-self.RADIUS,
                event.x+self.RADIUS, event.y+self.RADIUS, fill="black")

    def drawLinesOnCanvas(self, lines):
        for l in lines:
            self.w.create_line(l[0], l[1], l[2], l[3], fill='blue')
def main():
    root = tk.Tk()
    app = MainWindow(root)
    root.mainloop()

if __name__ == '__main__':
    main()
```

This method for creating a user-friendly Voronoi diagram GUI was inspired by the official Tkinter documentation, providing an intuitive interface for users to interactively draw points and visualize the resulting Voronoi regions.

5.4 In-built function for Voronoi diagram

This Python code implements the Voronoi diagram using the Voronoi class from the **scipy.spatial** module. The program generates a specified number of random points *(ndots)* within a given display area, ensuring a minimum distance *(mindistance)* between them. It then computes the Voronoi diagram based on these points using the Voronoi class and visualizes the result using *voronoiplot2d* from the same module. The background color of the plot is set to white, and the Voronoi edges are drawn in black. Additionally, the random points are plotted on the diagram for reference. This code provides a straightforward way to create and visualize Voronoi diagrams using the built-in functionality of the scipy library, making it accessible for users without the need for custom algorithms.

5.5 CGAL for Python - Voronoi diagram

I also tested out the CGAL implementation for the Voronoi diagram in the Python version. This was mainly to do a qualitative analysis. For this, I imported the **Point2**, **VoronoiDiagram** and **HalfEdge** data structure from the CGAL library. The points (x,y) are then manually defined and converted to the primitive kernel *Point2*. This is the suitable input data structure that is acceptable by the corresponding Voronoi diagram function.

The faces of the Voronoi diagram are stored in a half-edge data structure. The faces are always convex regions of the Voronoi diagram. It is so, because the Voronoi diagram can be represented as an intersection of half planes. The bisector bisects the plane into half planes and the intersection of the convex regions of those half planes is again a convex. The endpoints of the voronoi diagram is validated through a set of few lines of code. If we consider the border sites, one of their endpoints will be infinity, while for the sites completely enclosed, the endpoints will be finite.

5.6 Dynamic Voronoi diagram

```
class Dot:
    def __init__(self, position, color, radius):
        self.position = position
        self.color = color
        self.radius = radius
        self.direction = (np.random.rand(2) - 0.5)
        self.noise = Perlin(6789)

    def update(self, ax):
        self.position += self.direction
        if random.random() > 0.8:
            self.rotate_direction()
        display_width, display_height = ax.get_xlim()[1], ax.get_ylim()[1]
        if self.position[0] < 0:
            self.position[0] = 0
            self.direction[0] *= -1
        if self.position[0] > display_width:
            self.position[0] = display_width
            self.direction[0] *= -1
        if self.position[1] < 0:
            self.position[1] = 0
            self.direction[1] *= -1
        if self.position[1] > display_height:
            self.position[1] = display_height
            self.direction[1] *= -1
        ax.scatter(*self.position, c=[self.color], s=100, edgecolors='black')

    def rotate_direction(self):
        rand_angle = self.noise.one(np.random.randint(100)) / 100
        x = self.direction[0] * math.cos(rand_angle) -
            self.direction[1] * math.sin(rand_angle)
        y = self.direction[0] * math.sin(rand_angle) +
            self.direction[1] * math.cos(rand_angle)
        self.direction = [round(x, 2), round(y, 2)]
```

This Python code defines a **Dot** class for creating dynamic points in a simulated environment for the purpose of visualizing a dynamic Voronoi diagram. Each Dot object is initialized with a position, color, and radius. Additionally, it has a randomly generated initial direction and utilizes **Perlin noise** for added randomness. The update method simulates the movement of the dot based on its direction, with occasional random rotations. The dot's position is constrained within the display area, bouncing off the boundaries when reached. The *rotatedirection* method introduces variability in the dot's movement by applying a random rotation angle using *Perlin noise*. The *ax.scatter* call visualizes the dot on the provided matplotlib axis (ax). Overall, this code is part of a dynamic Voronoi diagram implementation where these dynamic points move within a defined space, creating an evolving Voronoi diagram over time.

```
n_dots = 50
background_color = (1, 1, 1)
dots_color = (0, 0, 0)
min_distance = 100  # Minimum distance between dots

display_width = 960
display_height = 540

def generate_random_dots(num_dots = n_dots, max_attempts=1000):
    dots = []
    for _ in range(num_dots):
        attempts = 0
        while attempts < max_attempts:
            new_dot = Dot((random.uniform(0, display_width),
                      random.uniform(0, display_height)), dots_color, 1)
            if all(np.linalg.norm(np.array(new_dot.position) -
                   np.array(dot.position)) >= min_distance for dot in dots):
                dots.append(new_dot)
                break
            attempts += 1
    return dots

dots = generate_random_dots()
vor = Voronoi([dot.position for dot in dots])

fig, ax = plt.subplots(figsize=(10, 5))
ax.set_xlim(0, display_width)
ax.set_ylim(0, display_height)
ax.set_facecolor(background_color)

def update(frame):
    ax.clear()
    ax.set_xlim(0, display_width)
    ax.set_ylim(0, display_height)
    ax.set_facecolor(background_color)
    for dot in dots:
        dot.update(ax)

    vor = Voronoi([dot.position for dot in dots])
    voronoi_plot_2d(vor, ax=ax, show_vertices=False,
                    line_colors=dots_color, line_width=2)
anim = FuncAnimation(fig, update, frames=range(100), interval=200)
anim.save('voronoi_animation_1.gif', writer='imagemagick', fps=20)
plt.show()
```

This Python code generates a dynamic Voronoi diagram animation using matplotlib. It begins by defining settings such as the number of dots *(ndots)*, background color, dot color, and minimum distance between dots *(mindistance)*. The display dimensions are set to *displaywidth* and *displayheight*. The *generaterandomdots* function creates a specified number of dots with random positions, ensuring a minimum distance between them through a defined number of attempts *(maxattempts)*. The main script initializes a set of random dots and computes the Voronoi diagram for the initial dot positions.

The code then sets up a matplotlib figure and axis, configuring the display settings. The update function is called for each frame of the animation, clearing and updating the axis to simulate dot movement. The Dot class, which represents dynamic points, updates their positions within the display boundaries, with occasional random rotations. The Voronoi diagram is recalculated for the updated dot positions in each frame, creating an evolving animation.

The animation is created using the **FuncAnimation** class from matplotlib, specifying the number of frames and the interval between frames. Finally, the animation is saved as a GIF file named *'voronoianimation2.gif'* using the *ImageMagick* writer and displayed using *plt.show()*. The resulting animation visually demonstrates the dynamic behavior of the Voronoi diagram as the dots move and interact within the defined space.

5.7 Observations

All the observations, which include the plots of voronoi diagram, the animation of voronoi diagram all in the files submitted alongside this.

6 Tools for Implementation

The implementation of the dynamic Voronoi diagram and associated components relies on several key programming tools and libraries, chosen for their effectiveness in handling geometric computations, graphical representation, and animation. The primary language utilized is Python, a versatile and widely-used programming language known for its readability and extensive libraries. The code makes extensive use of the **NumPy** library to efficiently handle numerical operations, particularly for vectorized calculations and array manipulations, crucial for geometric computations involved in Voronoi diagram generation.

Matplotlib, a comprehensive plotting library for Python, serves as the principal tool for graphical visualization. It provides a high-quality rendering of the Voronoi diagram and facilitates the creation of animations to depict the dynamic evolution of the diagram. Specifically, the **FuncAnimation** class from Matplotlib enables the seamless generation of animated visualizations by iteratively updating the displayed frames.

The Voronoi diagram computation is performed using the **SciPy** library, a powerful library for scientific and technical computing. The Voronoi diagram generation benefits from the *Voronoi* and *voronoiplot2d* functions provided by SciPy, streamlining the implementation and ensuring accuracy in the depiction of spatial relationships between the dynamic points.

The **Tkinter** library in Python plays a significant role in creating a graphical user interface (GUI) for the Voronoi diagram generation. Inspired by the **Tkinter** official documentation, the implementation allows users to interactively input points on the canvas, providing a user-friendly experience for custom Voronoi diagram creation.

To introduce randomness and dynamic behavior to the Voronoi diagram, the implementation incorporates the **Perlin** noise generation technique. The Perlin noise, implemented through a custom Perlin class, introduces controlled randomness into the direction and rotation of the dynamic points, resulting in a more visually appealing and engaging animation.

Additionally, the implementation leverages the **ImageMagick** library for saving the dynamic Voronoi diagram animation as a GIF. ImageMagick, a powerful image manipulation tool, provides an effective means of exporting the animated sequence for external use and analysis.

The **Computational Geometry Algorithms Library (CGAL)** serves a pivotal role in the implementation, primarily employed for qualitative validation of the Voronoi diagrams generated. **CGAL**'s reputation for offering efficient and reliable geometric algorithms, along with its robust data structures, makes it an invaluable tool for validating the correctness and precision of the Voronoi diagram computation.

7 Future Work

Future developments in dynamic Voronoi diagrams for robot planning could focus on efficiently handling real-time dynamic environments. This involves adapting the Voronoi representation dynamically as obstacles or other dynamic elements move within the robot's environment.

Dynamic Voronoi methods could be employed for analyzing the binding sites of proteins, particularly in molecular dynamics simulations. This involves dynamically adapting Voronoi representations to identify and analyze changes in binding sites over time, providing insights into protein interactions.

In the context of binding sites in proteins, the dynamic Voronoi methods can be employed in conjunction with Monte Carlo simulations to study protein interactions and binding sites. The combination of dynamic Voronoi diagrams and Monte Carlo simulations offers a powerful approach for exploring the conformational space of proteins, understanding the dynamics of molecular interactions, and predicting binding events.

7.1 Dynamic Binding Site Analysis:

Dynamic Voronoi methods could be employed for analyzing the binding sites of proteins, particularly in molecular dynamics simulations. This involves dynamically adapting Voronoi representations to identify and analyze changes in binding sites over time, providing insights into protein interactions.

7.2 Ligand-Protein Interaction Studies:

Future research could focus on the application of dynamic Voronoi diagrams to study ligand-protein interactions dynamically. This involves capturing the spatial distribution of ligands around proteins and understanding how these interactions evolve over time, contributing to drug discovery and design.

7.3 Protein Flexibility Modeling:

Dynamic Voronoi approaches could be integrated with techniques that model protein flexibility. This involves adapting the Voronoi representation to capture conformational changes in proteins, allowing for a more comprehensive understanding of their structural dynamics.

7.4 Dynamic Structural Alignment:

Dynamic Voronoi-based methods could contribute to dynamic structural alignment of proteins. This involves dynamically adjusting Voronoi regions to align proteins during simulations, aiding in the identification of conserved structural elements and functional regions.

7.5 Integration with Experimental Data:

Dynamic Voronoi methods could be integrated with experimental data, such as cryo-electron microscopy or X-ray crystallography, to refine protein models. This involves dynamically adjusting Voronoi representations to reconcile computational models with experimental observations, improving the accuracy of protein structures.

8 References

- Fortune's Algorithm in C++, Matt Brubeck.
- MCS 481 Computational Geometry online course. This course covers the depth of Fortune's algorithm principles along with the CGAL implementation on the same.
- Jaquesh's github repository, "fortune's algorithm"
- Dynamic Voronoi Diagram for Moving Disks, Chanyoung Song, et al, IEEE Transactions on Visualization and Computer Graphics
- Dynamic Voronoi diagrams in motion planning, H. Bieri, Computational Geometry Methods, Algorithms and Applications
- *https://www.cs.jhu.edu/ misha/Spring20/11a.pdf*, contained a detailed description on the Fortune's algorithm.
- *https://github.com/akihiko47/Voronoi-diagram*, gave me an idea on how to build an animation for the dynamic voronoi diagram.

Anshul Bagaria
BE21B005