CS 478 Computational Geometry

Spring 2023

Implementing Three Voronoi Diagram Computation Algorithms and Comparing Their Performance

Final Report

Elif Zeynep Soytorun 21901960

Zübeyir Bodur 21702382

Table of Contents

1. Introduction

1.1. Project Description

For the term project, we are implementing and assessing three different algorithms for calculating and visualising a Voronoi Diagram, given a set of points in two-dimensional space.

A Voronoi Diagram is a set of points (called seeds) in a two-dimensional space bounded by regions that depict the areas in that space in which any chosen point is closest to that area's seed [1].

The three algorithms we will use to form the diagram are:

- Randomised Incremental Algorithm (i.e. Bowyer-Watson Algorithm) [2],
- Fortune's Algorithm [3], and
- The Flipping Algorithm [4]

The application will have a simple user interface (UI) so that the user can choose the algorithm to form the diagram and enter the necessary parameters (number of vertices, or a path for a test file) for diagram generation. In addition to that, upon completion of the diagram, the time elapsed will be prompted to the user, so that the performance of the algorithms can be observed.

To test the algorithms, arbitrary point sets in two-dimensional space will be generated and a reasonable number of test cases will be used to compare the algorithms.

1.2. Project Environment

We have used Python as our programming language, and Pygame as our API to draw both the UI and visualisation of the Voronoi Diagram. We used Git to share our coding environment for collaboration.

2. Background

2.1. Voronoi Diagram and Its Relation with the Delaunay Triangulation

The Voronoi Diagram consists of regions that enclose a set of points (seeds) in a two-dimensional space, and these regions show which area of the space is closest to each seed point [1], by using an arbitrary distance metric such as Euclidean, Manhattan distance [5], or other distance measures like skew distance, polygonal convex distance or geodesic distance [6].

Since the Delaunay triangulation is the dual of the Voronoi diagram in Euclidean space, we will compute the Euclidean Voronoi Diagram in this project, and will not discuss other Voronoi Diagrams using other distance metrics.

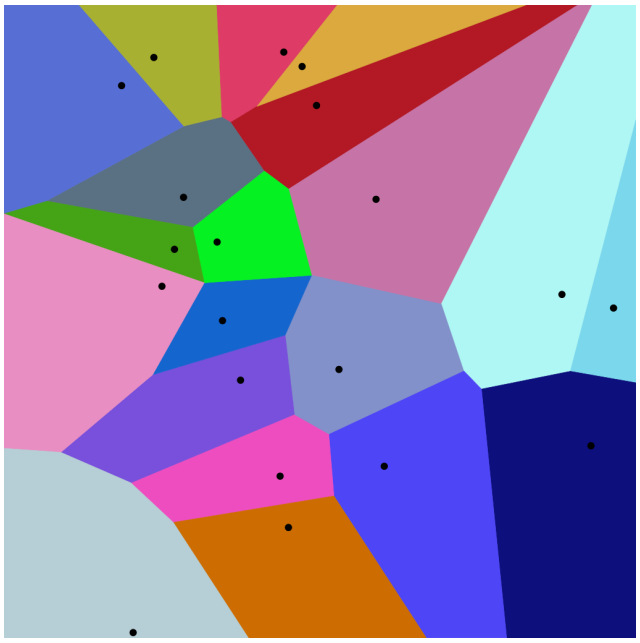

Figure 1: 20 points and their Voronoi cells [1]

However, some algorithms, such as the Bowyer-Wattson (Section 3.1.) algorithm and the Flipping algorithm (Section 3.3) generate the Delaunay triangulation of the given set, rather than directly computing the Voronoi diagram. Since the Delaunay triangulation of a set is the dual of its Voronoi diagram, we can generate the Voronoi diagram given its Delaunay triangulation. We will then use the following approach to convert a Delaunay triangulation into a Voronoi diagram.

Steps for drawing Voronoi Diagram from its Delaunay triangulation:

Input: A map of triangles **G** = (**Triangles**, **Neighbours**) that represents the Delaunay triangulation

Output: A planar straight line graph **G'** = (**V'**, **E'**) that represents the Voronoi Diagram

1. For each triangle **T** in **G**:
    1.1. Compute the circumcenter of **T** as **C**
    1.2. Insert vertex **C** into **G'**
    1.3. For each edge **e** in **T**:
        1.3.1. if **e** is an outer edge of the whole triangulation **G**:
            1.3.1.1. Insert a vertex **u**=(**e.v1**+**e.v2**) / 2 to **G'**
            1.3.1.2. Let **w** be the unit vector from **C** to **u**
            1.3.1.3. Let **bigM** be a large number
            1.3.1.4. If Circumcenter **C** is inside **T**
                1.3.1.4.1. Let **I** = **C** + **bigM*w**
                1.3.1.4.2. Insert the vertex into **G'.V**
                1.3.1.4.3. Insert an edge **e'= (C, I)** to **G'.E**.
            1.3.1.5. else
                1.3.1.5.1. Let the vertex **ov** be the vertex that is pointing towards **e** in **T**
                1.3.1.5.2. If the angle of **ov** is larger than **PI/2:**
                    1.3.1.5.2.1. Let **I** = **C** - **bigM*w**
                    1.3.1.5.2.2. Insert the vertex into **G'.V**
                    1.3.1.5.2.3. Insert an edge **e'= (C, I)** to **G'.E**.
                1.3.1.5.3. else
                    1.3.1.5.3.1. Let **I** = **C** + **bigM*w**
                    1.3.1.5.3.2. Insert the vertex into **G'.V**
                    1.3.1.5.3.3. Insert an edge **e'= (C, I)** to **G'.E**.
    1.4. Find the neighbouring triangles of **T**, **N(T)**.
    1.5. For each neighbouring triangle **T_n in N(T):**
        1.5.1. Insert an edge **e'=(C, C_n)** where **C_n** is the circumcenter of **T_n** to **G'.E**
        1.5.2. Insert vertex **C_n** into **G'.V**
2. **return** G

The complexity of this algorithm is $O(N + 3N + 3N + 3N) = O(N)$, because all operations including the check for extending an edge into infinity can be done in constant time. The latter can be done by caching a circumcenter flag into the vertices of **G'**, indicating that the point is a circumcenter of a triangle in **G**. In addition, finding the faces of **G** can be done in linear time as a preprocessing step. Finding neighbouring triangles of a triangle is already available in constant time if a doubly connected edge list (DCEL) data structure is used [4].

Computing the circumcenter C(c_x, c_y) of a triangle with vertices ABC is also a constant time operation. It can be computed using the following equations:

$$c_x = \frac{1}{D}((A.x^2 + A.y^2)(B.y - C.y) + (B.x^2 + B.y^2)(C.y - A.y) + (C.x^2 + C.y^2)(A.y - B.y))$$

$$c_y = \frac{1}{D}((A.x^2 + A.y^2)(C.x - B.x) + (B.x^2 + B.y^2)(A.x - C.x) + (C.x^2 + C.y^2)(B.x - A.x))$$

$$where\ D = 2(A.x(B.y - C.y) + B.x(C.y - A.y) + C.x(A.y - B.y))$$

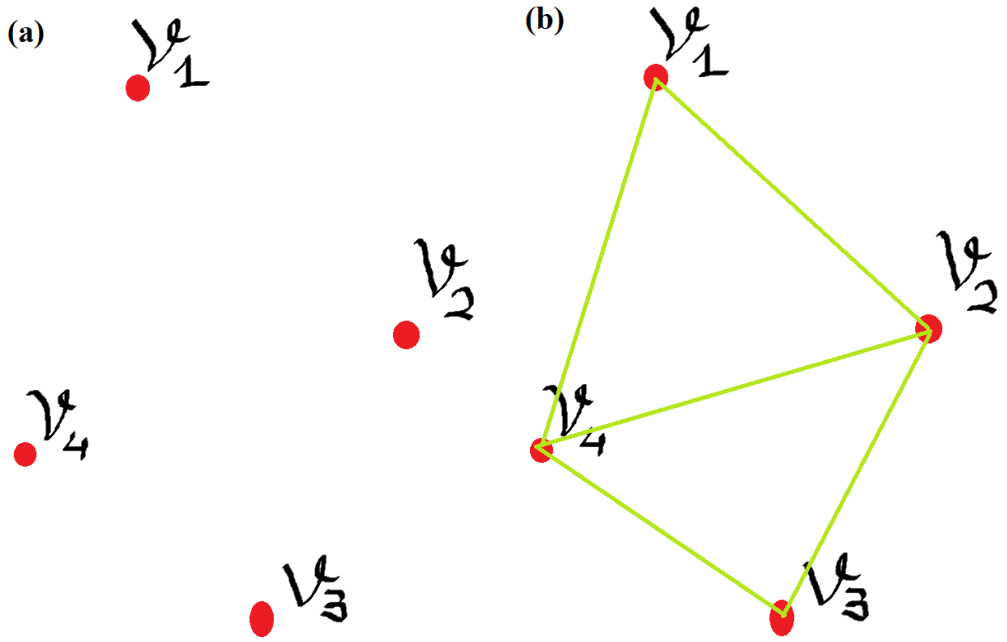

Figure 2: (a) Starting set for delaunay triangulation, and (b) its corresponding delaunay triangulation

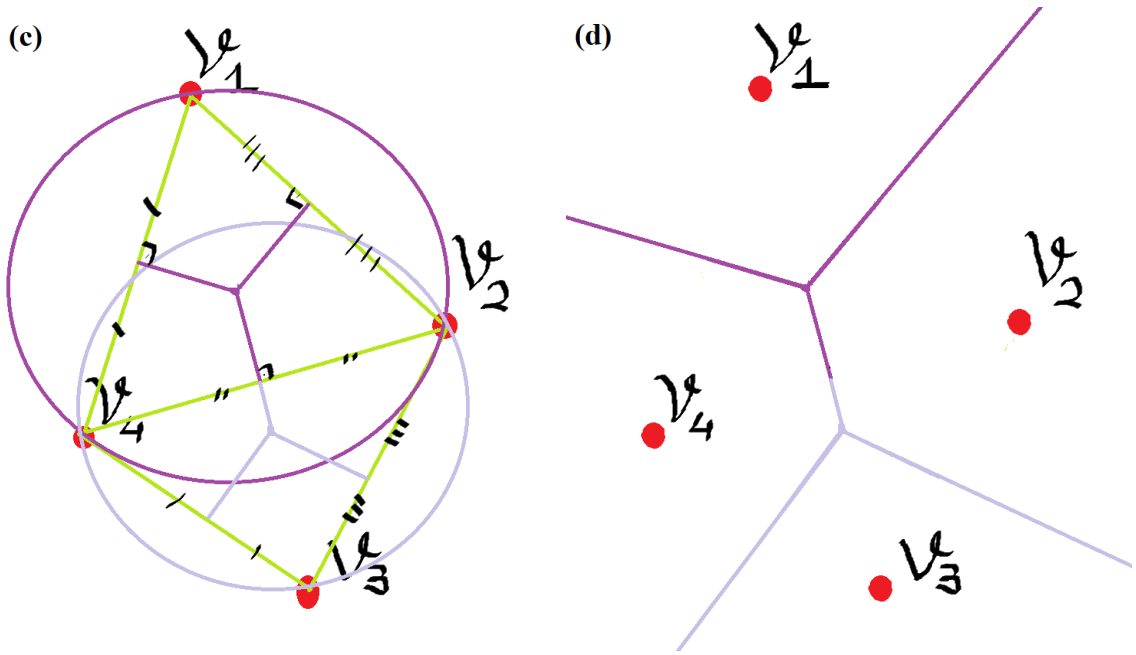

Figure 3: As discussed above, we can draw each excircle of a triangle, draw bisectors from each centre to the corresponding edges, and connect each neighbouring triangle's circumcenter, as done in (c). After that, we can remove all triangles and circles, and extend the bisectors to get the Voronoi diagram, as in (d).

3. Algorithms

This section provides detailed explanations of the algorithms used to compute the Voronoi Diagram of a point set **P**, or its dual, the Delaunay Triangulation.

3.1. Randomised Incremental Algorithm

As discussed above, this algorithm will be used to compute the Delaunay triangulation. Below is the pseudocode that explains the algorithm in detail.4

Steps for Bowyer-Watson algorithm [2]:

Input: A set of n points in the plane **P** = $\{p_1, p_2, \ldots, p_n\}$

Output: The Delaunay triangulation of **P**, **T= (Triangles, Neighbours)**

1. Let **P'** be a random permutation of P, **P'** ∈ S(**P**).
2. Initialise an empty triangulation **T = ({}, {})**
3. Let T_s be the super triangle that contains all points in **P'**
4. Add T_s into **T.** T_s initially has no neighbours.

5. For each point p_i in **P'**:
    5.1. Let **B** = {} be the set of bad triangles
    5.2. For each triangle T_j in **T**:
        5.2.1. Compute **C**, the circumcenter of T_j, and **r,** the radius of the circumcircle of T_j
        5.2.2. If the distance between **C** and p_i is not greater than **r**:
            5.2.2.1. **B** := **B** ∪ T_j
    5.3. Let **A** = {} be the set of edges of the polygon where triangulation will continue
    5.4. For each triangle T_b in **B**:
        5.4.1. For each edge **e** in T_b:
            5.4.1.1. If **e** is not shared by another triangle $T_b' \neq T_b, \exists T_b' \in$ **B:**
                5.4.1.1.1. Add **e** to **A**
    5.5. For each triangle T_b in **B**:
        5.5.1. Remove T_b from **T**:
    5.6. For each edge **e** in **A**:
        5.6.1. Let $T_{new} = (e, (e.V1, p_i), (e.V2, p_i))$ be a new triangle
        5.6.2. Insert T_{new} into **T**
        5.6.3. Update the neighbours of T_{new}
        5.6.4. Set the neighbours of T_{new}'s neighbours into T_{new}
6. For each triangle T_i in **T**:
    6.1. If there is a vertex **u** ∈ T_i s.t. **u** ∈ T_s
        6.1.1. Remove T_i from **T**
7. Return **T**

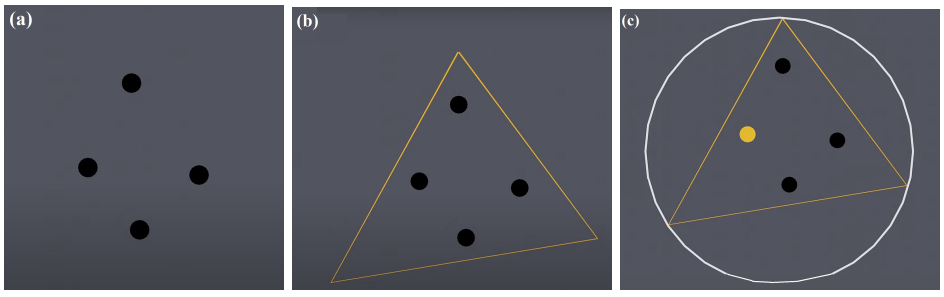

Figure 4: Initialization of the Bowyer-Watson algorithm with a super triangle (b) for the points in (a). The super triangle's circumcircle contains the next random paint, the super triangle will be marked as a bad triangle (c).

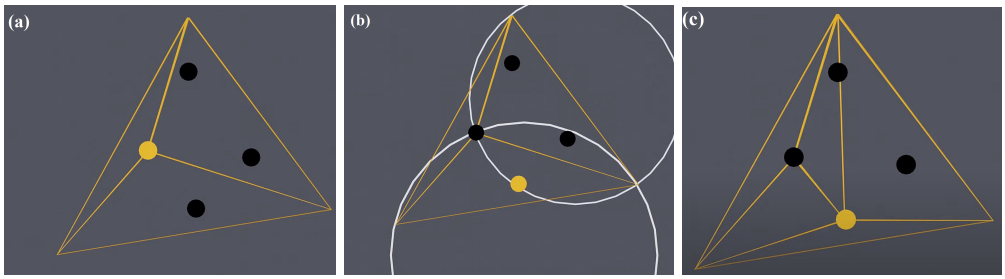

Figure 5: The convex hull of the super triangle is itself, therefore new edges will be drawn from the vertices of that convex hull towards the random point. Then, all the bad triangles are removed from the triangulation, therefore the circumcircle of the super triangle will not be checked again (a). In (b) and (c), the same steps explained will continue for the next random point.

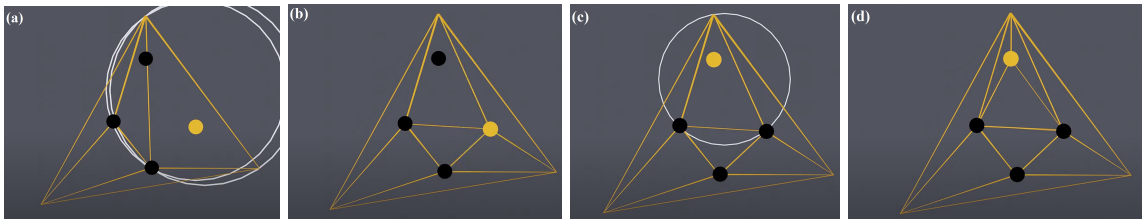

Figure 6: The iteration continues for the 3rd (a, b) and the last random point (c, d).

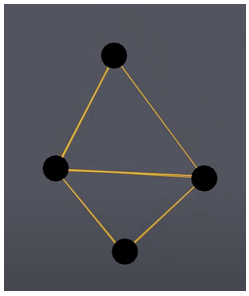

Figure 7: The final result of the triangulation, after removing all triangles whose edges are shared by the super triangle

3.2. Fortune's Algorithm

Fortune's algorithm is a planar sweep line algorithm that updates the Voronoi bisectors and sites by using a horizontal sweep line. In the algorithm, a Voronoi site is discovered when the sweep line passes through a point in the input set, and an intersection is discovered on each iteration of the algorithm [3]. [3] gives an algorithm that computes an incomplete implementation of the Voronoi diagram as a list of bisectors. Below is the full implementation of the algorithm which generates the Voronoi diagram [7, 8]:

Steps for generating the Voronoi Diagram using the planar sweep line algorithm:

Input: Set of points **P** = $\{p_1, p_2, \ldots, p_n\}$

Output: A planar graph **G = (V, E)** representing the diagram

1. Create a vertex set **G.V** = **P** as the places of the Voronoi sites
2. Let Q = $\{p_1, p_2, \ldots, p_n\}$ be an event priority queue where the events with lower y-axis values have higher priority. An event is basically a 3-tuple (**p**, **place_flag**, **arch**) where *p* is the place where the event occurs, *place_flag* is a predicate that tells if this event is a place event or an arch event, and *arch* is the arch above which the event occurs of place_flag is **False**. Unless specified, an event is a place event and can be created from a single point.
3. For each point p in **P**:
    3.1. Insert (p, **False**, NULL) into **Q**
4. Let **ParabolaRoot** be the sweep line status, which is a balanced binary search tree (BBST) of parabolas. The status is equivalent to the beach line that is drawn in the application. Initialise **ParabolaRoot** as an empty BBST.
    4.1. Each node has a parabola vector field, and a left parabola and right parabola field: Node = (vector, leftNode, rightNode)
5. Let **DeletedEvents** be a hash table, which represents the place and parabole events that are completely processed. Initialise it to an empty set.
6. Let **VoronoiEdges** be a list of collected **VoronoiEdge**s so far throughout the algorithm. Initialise as an empty list. **VoronoiEdge** is a 2-tuple (edge, neighbour) which holds its edge location in edge field, and a pointer to its neighbouring **VoronoiEdge**
7. While **Q** is not empty**:**
    7.1. Let **q** be the next event in **Q**
    7.2. Dequeue 1 item from Q
    7.3. Move the sweep line towards **q.p.y**
    7.4. if **q** is in **DeletedEvents**
        7.4.1. Remove **q** from **DeletedEvents**
    7.5. else if **q.place_flag** is **True**:
        7.5.1. InsertParabola(**q.p**)
    7.6. else:
        7.6.1. RemoveParabola(**q**)
8. FinishEdge(**ParabolaRoot**)
9. For each **VoronoiEdge e** in **VoronoiEdges**
    9.1. If **e**.neighbour is not **NULL**:
        9.1.1. Delete **e** from **VoronoiEdges**
10. Retrieve the unique points in **VoronoiEdges** as **VoronoiVertices**
11. Return **G = {VoronoiVertices, VoronoiEdges}**

The algorithm definitions for InsertParabola, RemoveParabola, and FinishEdge are provided as a source instead, as they are too long and complex to be given as a pseudo code:

```
def finish_edge(self, voronoi_parabola):
    """
    """
    if voronoi_parabola.isLeaf:
        return
    mx = -1.0
    if voronoi_parabola.edge.direction[0] > 0.0:
        mx = max(self.width, voronoi_parabola.edge.start[0] + 10)
    else:
        mx = min(0.0, voronoi_parabola.edge.start[0] - 10)

    end_vector = (mx, mx*voronoi_parabola.edge.f + voronoi_parabola.edge.g)
    voronoi_parabola.edge.end = end_vector
    self.points.append(end_vector)
    self.finish_edge(voronoi_parabola.left_child())
    self.finish_edge(voronoi_parabola.right_child())
```

*Figure 8: Source code for FinishEdge function, which recursively scans the **ParabolaRoot** status structure to rebalance the current parabola node, and their left and right child.*

```python
def insert_parabola(self, parabola_vector):
    """
    """
    # Base case
    if self.root_parabola == None:
        self.root_parabola = VoronoiParabola(parabola_vector)
        return

    # Degenerate event - heights are the same
    if self.root_parabola.isLeaf and (self.root_parabola.site[1] - parabola_vector[1]) < 1:
        fp = self.root_parabola.site
        self.root_parabola.isLeaf = False
        self.root_parabola.set_left_child(VoronoiParabola(fp))
        self.root_parabola.set_right_child(VoronoiParabola(parabola_vector))
        s = ((parabola_vector[0] + fp[0])/2.0, self.height) # starting edge
        self.points.append(s)
        if parabola_vector[0] > fp[0]:
            self.root_parabola.edge = VoronoiEdge(s, fp, parabola_vector)
        else:
            self.root_parabola.edge = VoronoiEdge(s, parabola_vector, fp)
        self.edges.append(self.root_parabola.edge)
        return

    # General case
    par = self.get_parabola_by_x(parabola_vector[0])

    if par.circleEvent != None:
        self.deleted_events.add(par.circleEvent)
        par.circleEvent = None

    start = (parabola_vector[0], self.get_y(par.site, parabola_vector[0]))
    self.points.append(start)

    edgeLeft = VoronoiEdge(start, par.site, parabola_vector)
    edgeRight = VoronoiEdge(start, parabola_vector, par.site)

    edgeLeft.neighbour = edgeRight
    self.edges.append(edgeLeft)

    par.edge = edgeRight

    p0 = VoronoiParabola(par.site)
    p1 = VoronoiParabola(parabola_vector)
    p2 = VoronoiParabola(par.site)

    par.isLeaf = False
    par.set_right_child(p2)
    par.set_left_child(VoronoiParabola())
    par.left_child().edge = edgeLeft
    par.left_child().set_left_child(p0)
    par.left_child().set_right_child(p1)

    self.check_circle(p0)
    self.check_circle(p2)
    return
```

*Figure 9: The source code for InsertParabola function, which inserts a new parabola to **ParabolaRoot** status with the given site vector.*

```
def remove_parabola(self, parabola_event):
    """
    """
    p1 = parabola_event.arch
    xl = p1.left_parabola_on_upper_level()
    xr = p1.right_parabola_on_upper_level()
    p0 = xl.left_parabola_on_lower_level()
    p2 = xr.right_parabola_on_lower_level()

    if p0 == p2:
        print("Error: parabola left and right have the same focus")

    if p0.circleEvent ≠ None:
        self.deleted_events.add(p0.circleEvent)
        p0.circleEvent = None
    if p2.circleEvent ≠ None:
        self.deleted_events.add(p2.circleEvent)
        p2.circleEvent = None

    p_vector = (parabola_event.point[0], self.get_y(p1.site, parabola_event.point[0]))
    self.points.append(p_vector)

    xl.edge.end = p_vector
    xr.edge.end = p_vector

    higher = VoronoiParabola()
    par = p1
    while par ≠ self.root_parabola:
        par = par.parent
        if par == xl:
            higher = xl;
        elif par == xr:
            higher = xr
    higher.edge = VoronoiEdge(p_vector, p0.site, p2.site)
    self.edges.append(higher.edge)

    grand_parent = p1.parent.parent
    if grand_parent ≠ None:
        if p1.parent.left_child() == p1:
            if grand_parent.left_child() == p1.parent:
                grand_parent.set_left_child(p1.parent.right_child())
            if grand_parent.right_child() == p1.parent:
                grand_parent.set_right_child(p1.parent.right_child())
        else:
            if grand_parent.left_child() == p1.parent:
                grand_parent.set_left_child(p1.parent.left_child())
            if grand_parent.right_child() == p1.parent:
                grand_parent.set_right_child(p1.parent.left_child())
    self.check_circle(p0)
    self.check_circle(p2)
    return
```

Figure 10: The source code for RemoveParabola function, which removes the parabola associated with the event from ***ParabolaRoot*** *status.*

The rest of the functions such as CheckCircle, GetXOfEdge, GetY, GetEdgeIntersection etc. were skipped for simplicity.

The algorithm runs in O(NlogN) time complexity, as there are O(N) iterations, and in each iteration, insertion and deletion from the priority queue takes O(logN) time. In steps 2, 3, & 7, a heap based priority queue is used, therefore each enqueue and dequeue will take O(logN) time. There are initially N number of events, however, the number of events increase as the sweep line moves by a certain amount along the x-axis in each iteration, and may enqueue new events at each iteration. Therefore, the number of events may increase up to Width * Height * N, depending on how polygons lie on the 2D plane. If width and height is not a function of N, we can conclude that the number of events will be proportional to N, therefore the time complexity of the 7th step will be O(NlogN).

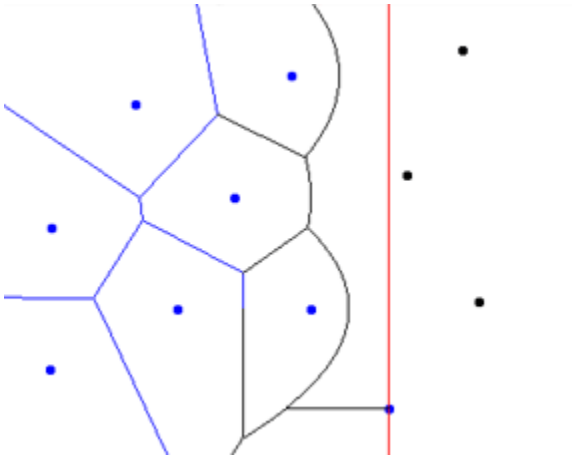

Figure 11: Visualisation of an iteration of Fortune's algorithm [7, 8].

3.3. The Flipping Algorithm

The Flipping algorithm [4] is another algorithm that generates the Delaunay Triangulation of a set of points. The algorithm starts with a super triangle, and then processes each point in the list to create sub triangles by dividing the triangle that locates the point. After triangles are added, if the newly generated triangles' edges are illegal, each edge is flipped recursively [4]. The detailed algorithm is provided below:

Algorithm FlippingAlgorithm(**P**) **begin**:

Input: Set of points **P** = $\{p_1, p_2, \ldots, p_n\}$

Output: A set of triangles which map a set of neighbours, **Triangles = (Triangles, Neighbours)** representing the Delaunay Triangulation

1. Let **Triangles** = {} be a hashmap, which maps a triangle to three triangles which are the current triangle's neighbours
2. Add two triangles T_{s1}, T_{s2} into **Triangles** which are two super triangles that make two big squares combined. Therefore T_{s1} is a neighbour of T_{s2} and vice versa.
3. for each point p in **P**:
   3.1. addPoint(**Triangles**, p)
4. **return Triangles**

procedure addPoint(**Triangles**, p) **begin**

Input: A set of triangles which map a set of neighbours, **Triangles** representing the Delaunay Triangulation, and a point p to be registered into this triangulation.

1. Find the triangle **T** that locates p by iterating over **Triangles**
2. Check if p is on any edge of T, if so:
   2.1. Let e=(p1, p2) be the edge on top of p
   2.2. Let p3 be the third point of the triangle **T**
   2.3. Let $\mathbf{T_{adj}}$ be the adjacent triangle to **T** over edge e
   2.4. Let p4 be the third vertex of $\mathbf{T_{adj}}$
   2.5. Register p and split **T** and $\mathbf{T_{adj}}$ into two over p, let the output triangles be T1, T2, T3 and T4.
   2.6. LegalizeEdge(**Triangles**, (p3, p1), T1)
   2.7. LegalizeEdge(**Triangles**, (p3, p2), T2)
   2.8. LegalizeEdge(**Triangles**, (p4, p1), T4)
   2.9. LegalizeEdge(**Triangles**, (p4, p2), T3)
3. else:
   3.1. Let (a, b, c) be the vertices of **T**
   3.2. Register p and split **T** into three over p, let the output triangles be T1, T2, T3.

3.3. LegalizeEdge(**Triangles**, (a, b), T1)
3.4. LegalizeEdge(**Triangles**, (a, c), T2)
3.5. LegalizeEdge(**Triangles**, (b, c), T3)

4. **return**

procedure LegalizeEdge(**Triangles**, edge, T) **begin**

Input: A set of triangles which map a set of neighbours, **Triangles** representing the Delaunay Triangulation, edge to be legalised and the triangle T that locates the edge.

1. Let $\mathbf{T}_{adj}$ be the adjacent triangle to **T** over edge e
2. Let e=(p1, p2), and p3 be the third vertex of **T**, and p4 be the third vertex of $\mathbf{T}_{adj}$
3. If the circumcircle of **T** contains p4:
   3.1. Flip the edge e with respect to **T** and $\mathbf{T}_{adj}$. Update T and T_{adj} accordingly, and let e'=(p3, p4) be the new edge that splits **T** and $\mathbf{T}_{adj}$.
   3.2. LegalizeEdge((p1, p4), **T**)
   3.3. LegalizeEdge((p4, p2), $\mathbf{T}_{adj}$)
4. **return**

The procedures where the triangles are split into three or four, and an edge is flipped is given below as a source code. Other parts such as circumcircle test, on edge test, point location, retrieving all neighbours from edge, retrieving a neighbour from a triangle etc. were skipped.

```
def flip_edge(self, common_edge, diagonal, t1, t2):
    """
    Just flips an edge. Doesn't check for legality.
    Throws an error if diagonals were not given from t1 to t2
    """
    assert((diagonal[0] in t1) and (diagonal[1] in t2))
    N1, N1_index, N2, N2_index, N3, N3_index, N4, N4_index = self.get_neighbours_of_edge(common_edge, t1, t2)
    self.triangles.pop(t1)
    self.triangles.pop(t2)
    self.circles.pop(t1)
    self.circles.pop(t2)
    t1_new = (common_edge[0], diagonal[0], diagonal[1])
    t2_new = (common_edge[1], diagonal[1], diagonal[0])
    self.triangles[t1_new] = [t2_new, N3, N1]
    self.triangles[t2_new] = [t1_new, N2, N4]
    self.circles[t1_new] = self.circumcenter(t1_new)
    self.circles[t2_new] = self.circumcenter(t2_new)
    if N1 ≠ None and N1_index ≠ -1:
        self.triangles[N1][N1_index] = t1_new
    if N2 ≠ None and N2_index ≠ -1:
        self.triangles[N2][N2_index] = t2_new
    if N3 ≠ None and N3_index ≠ -1:
        self.triangles[N3][N3_index] = t1_new
    if N4 ≠ None and N4_index ≠ -1:
        self.triangles[N4][N4_index] = t2_new
    return t1_new, t2_new
```

Figure 12: Source code for flipEdge subroutine, which flips the drawn edge of a quad by removing the given triangles t1 and t2 and adding new triangles that are split by the other diagonal

```
def register_point_and_split_triangles_into_four_from_edge(self, common_edge, diagonal, t1, t2, point):
    """
    Removes the indicated triangle, registers the
    new triangles and then updates the neighbours.
    Returns the index of the generated triangles as list
    N1 and N2 are neighbours of t1 other than t2
    N3 and N4 are neighbours of t1 other than t1
    Throws an error if the point is not on the common edge
    Throws an error if the common edge is actually not the common edge
    Throws an error if the diagonal vertices are not given in order, from t1 to t2
    t2 can be None, but t1 definetely can't be None, it throws an error otherwise
            diag0
            /|\\
        N1  / | \\  N2
           /T1|T2\\
          /   |   \\----->t1
         /____.____\\
        c0    |     c1
        \\ T3 | T4  /
         \\   |    /---->t2
          \\  |   /
        N3 \\ |  /  N4
            \\| /
             \\/
            diag1
    """
    assert(t1 != None)
    if t2 != None:
        assert(diagonal[0] in t1 and diagonal[1] in t2)
        assert((common_edge[0] in t1) and (common_edge[1] in t1) and(common_edge[0] in t2) and(common_edge[1] in t2))
        assert(self.on_triangle_edge_test(point, t1) == common_edge)
        assert(self.on_triangle_edge_test(point, t2) == common_edge)
        p_index = len(self.coords)
        self.coords.append(point)
        a, b, c = t1
        d, e, f = t2
        T1 = (diagonal[0], common_edge[0], p_index)
        T2 = (common_edge[1], diagonal[0], p_index)
        T3 = (common_edge[0], diagonal[1], p_index)
        T4 = (diagonal[1], common_edge[1], p_index)
        N1, N1_index, N2, N2_index, N3, N3_index, N4, N4_index = self.get_neighbours_of_edge(common_edge, t1, t2)

        self.triangles.pop(t1)
        self.circles.pop(t1)
        self.triangles.pop(t2)
        self.circles.pop(t2)

        self.triangles[T1] = [T3, T2, N1]
        self.circles[T1] = self.circumcenter(T1)
        if N1 != None and N1_index != -1:
            self.triangles[N1][N1_index] = T1

        self.triangles[T2] = [T1, T4, N2]
        if N2 != None and N2_index != -1:
            self.triangles[N2][N2_index] = T2
        self.circles[T2] = self.circumcenter(T2)

        self.triangles[T3] = [T2, T1, N3]
        self.circles[T3] = self.circumcenter(T3)
        if N3 != None and N3_index != -1:
            self.triangles[N3][N3_index] = T3

        self.triangles[T4] = [T2, T1, N3]
        self.circles[T4] = self.circumcenter(T4)
        if N4 != None and N4_index != -1:
            self.triangles[N4][N4_index] = T4
        return T1, T2, T3, T4
    else:
        assert(diagonal[0] in t1)
        assert((common_edge[0] in t1) and (common_edge[1] in t1))
        assert(self.on_triangle_edge_test(point, t1) == common_edge)
        p_index = len(self.coords)
        self.coords.append(point)
        a, b, c = t1
        T1 = (diagonal[0], common_edge[0], p_index)
        T2 = (common_edge[1], diagonal[0], p_index)
        N1, N1_index, N2, N2_index, N3, N3_index, N4, N4_index = self.get_neighbours_of_edge(common_edge, t1, None)
        self.triangles.pop(t1)
        self.circles.pop(t1)

        self.triangles[T1] = [None, T2, N1]
        self.circles[T1] = self.circumcenter(T1)
        if N1 != None and N1_index != -1:
            self.triangles[N1][N1_index] = T1

        self.triangles[T2] = [T1, None, N2]
        if N2 != None and N2_index != -1:
            self.triangles[N2][N2_index] = T2
        self.circles[T2] = self.circumcenter(T2)
        return T1, T2, None, None
```

Figure 13: Source code for splitting two triangles into 4, by retrieving the neighbours around the edge

```
def register_point_and_split_triangle_into_three(self, tri_indices, point):
    """
    Removes the indicated triangle, registers the
    new triangles and then updates the neighbours.
    Returns the index of the generated triangles as list
    Throws an error if the point lies on an edge, or the point is outside
              a
             / \\
        N1  /   \\  N2
           /T1.T2\\
          /       \\
         /____T3___\\
        b     N3     c
    """
    assert(self.on_triangle_edge_test(point, tri_indices) == None)
    assert(self.in_triangle_test(point[0], point[1], tri_indices))
    p_index = len(self.coords)
    self.coords.append(point)
    a, b, c = tri_indices
    T1 = (a, b, p_index)
    T2 = (c, a, p_index)
    T3 = (b, c, p_index)
    N1 = self.triangles[(a, b, c)][2] # adjacent to T1
    N1_index = -1
    N2 = self.triangles[(a, b, c)][1] # adjacent to T2
    N2_index = -1
    N3 = self.triangles[(a, b, c)][0] # adjacent to T3
    N3_index = -1
    if N1 != None:
        #for j, (ja, jb, jc) in enumerate(self.triangles[N1]):
        for j in range(3):
            three_tuple = self.triangles[N1][j]
            if three_tuple != None:
                (ja, jb, jc) = three_tuple
                if (ja, jb, jc) == tri_indices:
                    N1_index = j
    if N2 != None:
        #for j, (ja, jb, jc) in enumerate(self.triangles[N2]):
        for j in range(3):
            three_tuple = self.triangles[N2][j]
            if three_tuple != None:
                (ja, jb, jc) = three_tuple
                if (ja, jb, jc) == tri_indices:
                    N2_index = j
    if N3 != None:
        #for j, (ja, jb, jc) in enumerate(self.triangles[N3]):
        for j in range(3):
            three_tuple = self.triangles[N3][j]
            if three_tuple != None:
                (ja, jb, jc) = three_tuple
                if (ja, jb, jc) == tri_indices:
                    N3_index = j
    self.triangles.pop(tri_indices)
    self.circles.pop(tri_indices)
    self.triangles[T1] = [T3, T2, N1]
    self.circles[T1] = self.circumcenter(T1)
    if N1 != None and N1_index != -1:
        self.triangles[N1][N1_index] = T1
    self.triangles[T2] = [T1, T3, N2]
    self.circles[T2] = self.circumcenter(T2)
    if N2 != None and N2_index != -1:
        self.triangles[N2][N2_index] = T2
    self.triangles[T3] = [T2, T1, N3]
    self.circles[T3] = self.circumcenter(T3)
    if N3 != None and N3_index != -1:
        self.triangles[N3][N3_index] = T3
    return T1, T2, T3
```

Figure 14: Source code for splitting a triangle into three from a point.

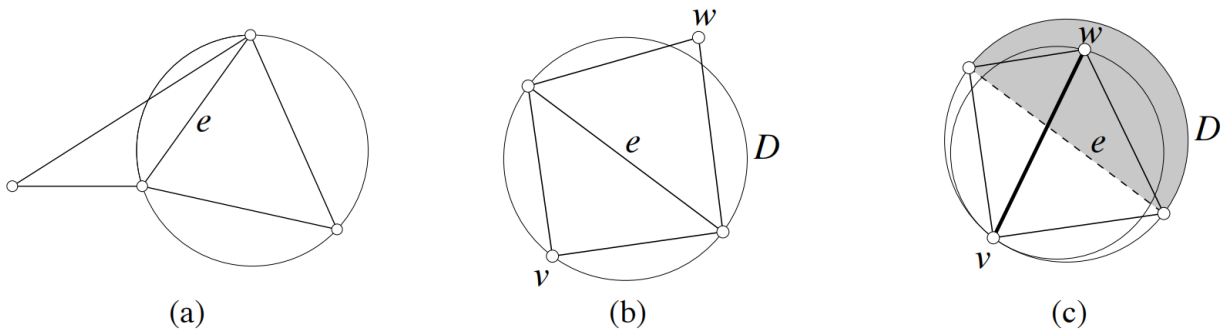

Figure 15: This figure shows how flipEdge function works if an illegal edge is encountered. Legality checks for three different edges in (a), (b), and (c). (a) is not convex, therefore it is skipped. (b) is convex and edge e is locally Delaunay, therefore no flipping was made. However, in (c), the triangle below e has a circumcircle which also includes w, therefore e must be flipped into e'=(w, v) [9].

In total, the Flipping algorithm will have O($NlogN$) worst-case time complexity, since there are N points to be processed and there can be at most O(logN) LegalizeEdge calls per illegal edge.

4. Results

4.1. Randomised Incremental Algorithm

4.1.1. Random Distribution Table

Number of Points (N)	Execution Time (ms)
10	9.494066
20	23.936272
30	38.947105
50	75.798035
100	207.569361
200	732.789993
300	1555.064678
400	2548.89369
500	3425.924778
600	4865.101099
700	6727.952957
800	8420.286417
900	11723.77086
1000	15145.56527
1500	29229.32
2000	51395.3526
2500	78826.85685
3000	117095.7699
4000	201379.4832
5000	301785.0091

Table 1: Randomised Incremental Algorithm Execution Time in Random Point Distribution Depending on Number of Points

4.1.2. Gaussian Distribution Table

Number of Points (N)	Execution Time (ms)
10	9.990454
20	23.93627
30	50.84634
50	81.779
100	212.3027
200	677.1421
300	1665.858
400	2366.626
500	3806.035
600	5643.662
700	7934.047
800	8743.923
900	10906.22
1000	15511.55
1500	28565.66
2000	50652.4
2500	83895.12
3000	116562.6
4000	205645.2
5000	372442.7

Table 2: Randomised Incremental Algorithm Execution Time in Gaussian Point Distribution Depending on Number of Points

4.1.3. Uniform Distribution Table

Number of Points (N)	Execution Time (ms)
10	10.96964
20	22.93897
30	39.90364
50	77.7905
100	214.5047
200	743.0162
300	1338.948
400	2270.815
500	3305.896
600	5030.788
700	6252.051
800	8146.425
900	11096.93
1000	13688.69
1500	30850.82
2000	52443.25
2500	77772.92
3000	118314.8
4000	218453
5000	313556.1

Table 3: Randomised Incremental Algorithm Execution Time in Uniform Point Distribution Depending on Number of Points

4.1.4. Plot

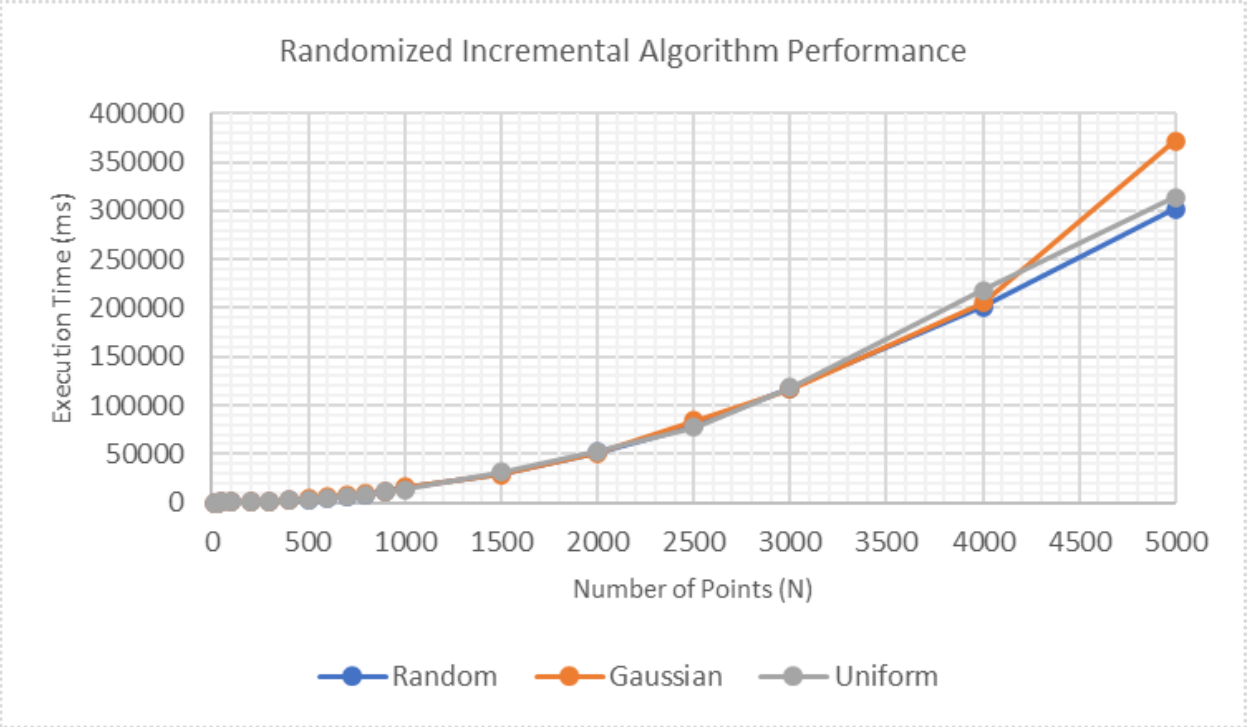

Figure 16: Plot for the performance of the Randomised Incremental Algorithm in 3 different distributions

4.2. Fortune's Algorithm

4.2.1. Random Distribution Table

Number of Points (N)	Execution Time (ms)
10	4.448175
20	23.769617
30	55.008411
50	97.359657
100	272.895098
200	685.216427
300	1152.982473
400	1692.913771
500	2441.297293
600	3007.762432
700	3862.286568
800	5164.619207
900	5389.488220
1000	7131.824017
1500	10943.998337
2000	16849.138737
2500	23064.267874
3000	32028.659344
4000	50157.723665
5000	66932.689428

Table 4: Fortune's Algorithm Execution Time in Random Point Distribution Depending on Number of Points

4.2.2. Gaussian Distribution Table

Number of Points (N)	Execution Time (ms)
10	9.006262
20	30.533075
30	60.588837
50	107.970953
100	281.888723
200	661.725044
300	1198.991299
400	1714.288950
500	2300.536633
600	2788.488626
700	3335.072041
800	4067.402363
900	4989.796877
1000	6481.916428
1500	9500.954151
2000	15304.574251
2500	19785.891771
3000	25710.190296
4000	39921.250343
5000	57427.577257

Table 5: Fortune's Algorithm Execution Time in Gaussian Point Distribution Depending on Number of Points

4.2.3. Uniform Distribution Table

Number of Points (N)	Execution Time (ms)
10	8.506298
20	20.729065
30	48.167706
50	99.585056
100	278.848410
200	657.317400
300	1226.730347
400	1721.216917
500	2341.856003
600	3050.943136
700	3804.420710
800	4433.227062
900	5377.810478
1000	6196.620464
1500	11271.962881
2000	16325.966120
2500	22247.358561
3000	28687.216043
4000	48430.392504
5000	72216.839075

Table 6: Fortune's Algorithm Execution Time in Uniform Point Distribution Depending on Number of Points

4.2.4. Plot

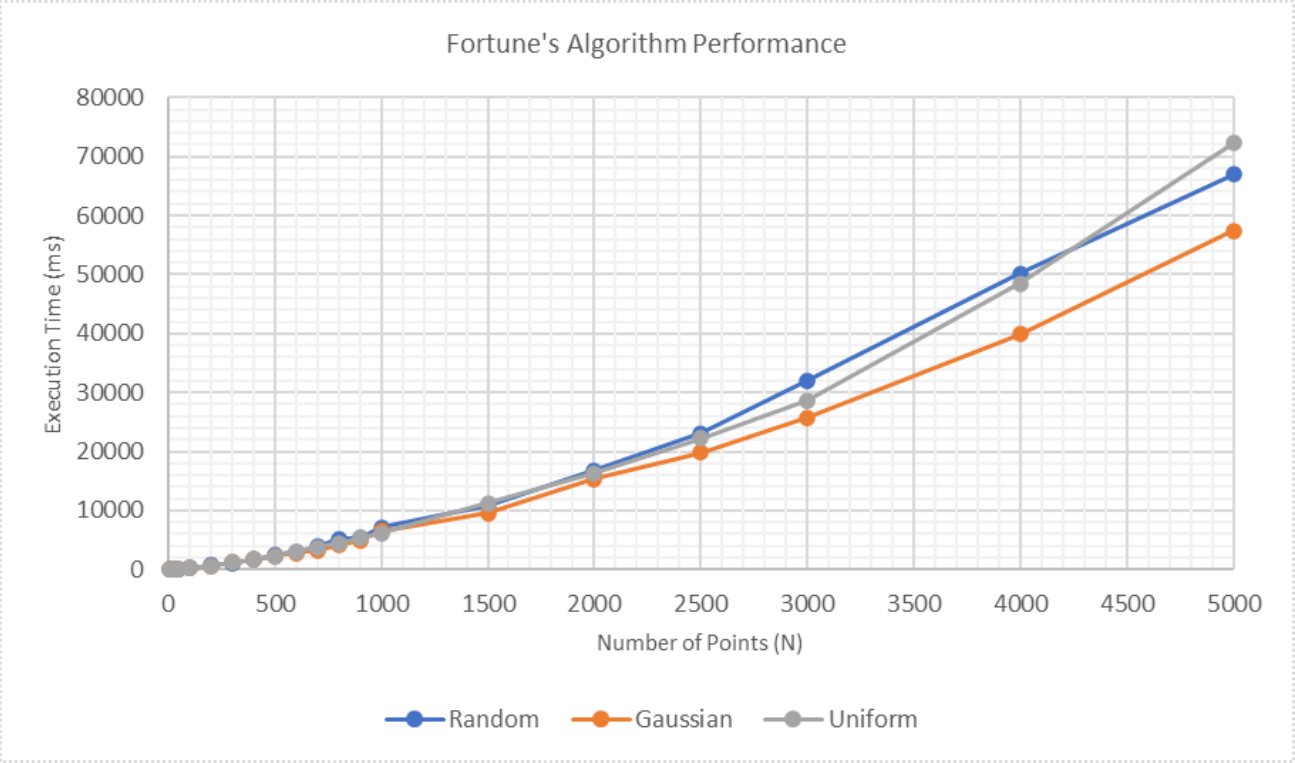

Figure 17: Plot for the performance of the Fortune's Algorithm in 3 different distributions

4.3. The Flipping Algorithm

4.3.1. Random Distribution Table

Number of Points (N)	Execution Time (ms)
10	15.9092
20	34.94072
30	66.34188
50	108.4666
100	233.6867
200	647.3703
300	1148.138
400	1645.171
500	2464.638
600	3414.504
700	4331.15
800	5245.757
900	6674.119
1000	8580.781
1500	16602.85
2000	29095.54
2500	43110.45
3000	59515.87
4000	101386.6
5000	177707.3

Table 7: Flipping Algorithm Execution Time in Random Point Distribution Depending on Number of Points

4.3.2. Gaussian Distribution Table

Number of Points (N)	Execution Time (ms)
10	15.01203
20	33.93126
30	54.90732
50	117.161
100	268.2576
200	579.5009
300	1293.578
400	1783.816
500	2912.387
600	3161.827
700	4646.831
800	5197.187
900	6419.618
1000	7666.325
1500	19090.52
2000	32592.46
2500	50331.56
3000	70471.43
4000	112377.4
5000	172290.6

Table 8: Flipping Algorithm Execution Time in Gaussian Point Distribution Depending on Number of Points

4.3.3. Uniform Distribution Table

Number of Points (N)	Execution Time (ms)
10	15.01632
20	34.90996
30	67.82961
50	90.80124
100	241.112
200	591.4431
300	1008.444
400	1900.84
500	2226.376
600	3103.936
700	3913.758
800	4887.927
900	5923.784
1000	8491.282
1500	16766.16
2000	28186.59
2500	46916.76
3000	66129.82
4000	101991.7
5000	159445.5

Table 9: Flipping Algorithm Execution Time in Uniform Point Distribution Depending on Number of Points

4.3.4. Plot

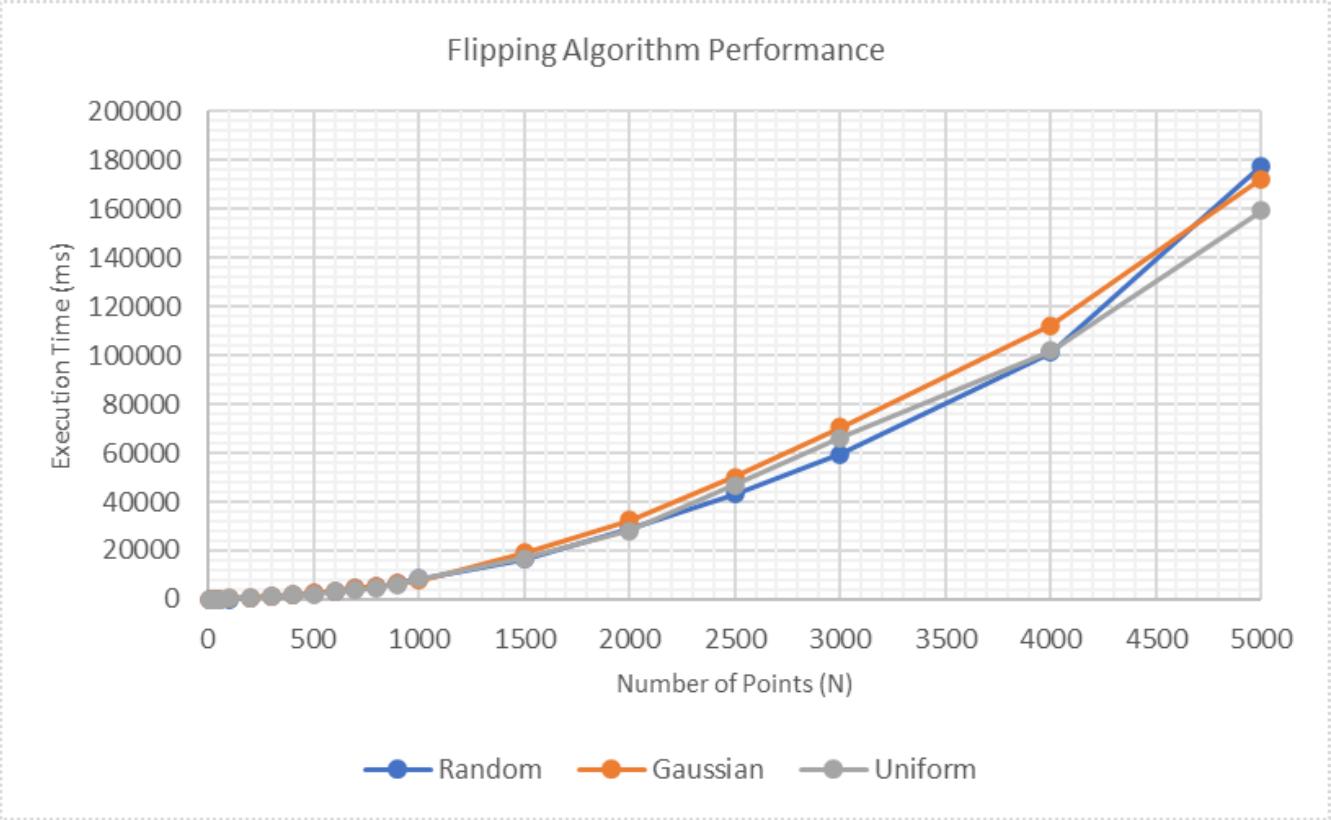

Figure 18: Plot for the performance of the Flipping Algorithm in 3 different distributions

5. Conclusion

From these results, we can make the following observations:

- The Flipping algorithm outperforms the Randomised Incremental algorithm for large N values, and the opposite occurs for small N values.
- Fortune's algorithm clearly outperforms all algorithms for all N values.
- Gaussian distribution is the healthiest distribution for Fortune's algorithm. However, this does not apply for the other algorithms, as it can be seen from 4.1.4 and 4.2.4., Gaussian distribution may not outperform the other distributions. In fact, Gaussian distribution is clearly outperformed by others in the Randomised Incremental algorithm (see 4.1.4).
- Uniform distribution tends to perform similar to Random distribution except in the Flipping algorithm. In the Flipping algorithm tests, we can observe that the tests in the Uniform distribution were fastest.
- We can also observe that, in practice, all algorithms have the asymptotic time complexity O(NlogN), as was explained in the algorithms part of the report.

References

[1] Wikimedia Foundation. “Voronoi diagram”, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Voronoi_diagram. [Accessed Mar. 23, 2023]

[2] Rebay, S. “Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm”. Journal of Computational Physics, May. 1993, vol. 106/1, pp. 127. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021999183710971. [Accessed Mar. 24, 2023]

[3] S. Fortune. “A Sweepline Algorithm for Voronoi Diagrams”, Algorithmica, 1987, vol. 2, pp. 153-174. [Online]. Available: https://link.springer.com/content/pdf/10.1007/BF01840357.pdf [Accessed Mar. 24, 2023.

[4] A. Mukherjee. “COT5520 - Computational Geometry Lecture Slides”, University of Central Florida, 2003. [Online]. Available: http://www.cs.ucf.edu/courses/cot5520/lectures.html. [Accessed Mar. 24, 2023].

[5] K. Gohrani. “Different Types of Distance Metrics used in Machine Learning”, Medium, Nov. 10, 2019. [Online]. Available: https://medium.com/@kunal_gohrani/different-types-of-distance-metrics-used-in-machine-learning-e9928c5e26c7. [Accessed Mar. 24, 2023]

[6] S. Fortune, "Voronoi Diagram and Delaunay Triangulations" in Handbook of Discrete and Computational Geometry, Third Edition, J. E. Goodman, J. O'Rourke & C. D. Tóth, Ed. Boca Raton, FL: CRC Press LLC, 2017, pp. 705-721. [Online]. Available: http://www.csun.edu/~ctoth/Handbook/chap27.pdf. [Accessed Mar. 24, 2023].

[7] K. Wong & H. A. Müller. “An Efficient Implementation of Fortune’s Plane-Sweep Algorithm for Voronoi Diagrams”, Department of Computer Science, University of Victoria, Victoria, BC, V8W 3P6, Canada. [Online]. Available: https://citeseerx.ist.psu.edu/doc/10.1.1.83.5571 . [Accessed Mar. 24, 2023].

[8] njanakiev. “Implementation of Fortune's Algorithm in Processing”, GitHub, 2017. [Online]. Available: https://github.com/njanakiev/fortune-algorithm/tree/master . [Accessed May. 24, 2023].

[9] S. Cheng, T. K. Dey & J. R. Shewchuk.”Delaunay Mesh Generation”, CRC Press, Boca Raton, Florida, Dec. 2012, pp 31-54. [Online]. Available: https://people.eecs.berkeley.edu/~jrs/papers/meshbook/chapter2.pdf. [Accessed Mar. 24, 2023]