Final Report for Implementing Three Voronoi Diagram Computation Algorithms and Comparing Their Performance

CS478

Students

Burak Öztürk 21901841

Alp Tuğrul Ağçalı 21801799

1. Introduction

In this project, as a group, we implemented three algorithms used to create the Voronoi diagram. The names of these algorithms were Randomized Incremental Algorithm, Fortune's Algorithm, The Flipping Algorithm. In addition to these, we have also designed a user interface where these algorithms are tested, and the results and steps can be seen.

1.1 Problem Definition

A Voronoi diagram is defined by three sets: a set of sites (points) S, a set of edges E and a set of vertices V.

Set S is the input parameter of the function, and the goal is to divide the plane into regions with edges such that every region only contains one site s[1] and every arbitrary point p in that region is closer to s than to any other site[2]. And the vertices are the points that 3 or more edges intersect like v in Figure 1.

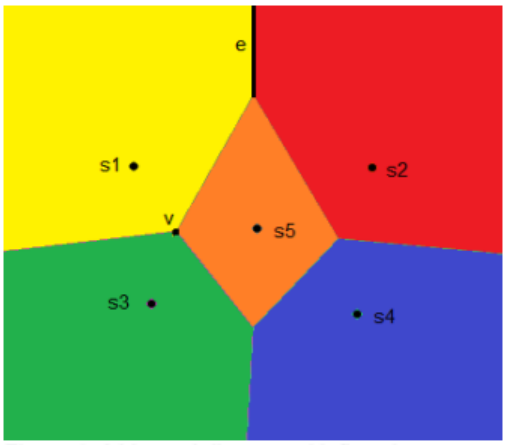


Figure 1: A Voronoi diagram with five sites.

This trait of Voronoi diagrams implicitly means an edge e is a line segment that two regions r1 and r2 share is the set of points that are the same distance from sites s1 and s2 (from r1 and r2 respectively).

Similarly, vertices are the points that n ≥ 3 regions Rn = {r1, r2 , …, rn}share is the set of points that are the same distance from sites Sn = {s1 , s2 , …, sn} (from s1 and s2 respectively) but in this case the set only contains one point so these are vertices.[3]

1.2 Approaches

We studied three approaches that can be used to generate a Voronoi diagram from a set of sites. In this part, those approaches are discussed.

1.2.1 Randomized Incremental Algorithm

An incremental algorithm is an iterative algorithm that inserts a new element at each step while maintaining some properties over the elements[4]. This algorithm starts with a supertriangle that

encapsulates all the points in the set. Then it adds all the points one by one to the supertriangle and legalizes edges to keep the Delaunay property for all formed triangles.

Updating the triangulation can include adding and deleting vertices, adding, resizing, or deleting edges. As an example, suppose we want to construct a triangulation T' by adding a point p to T, a Delaunay triangulation of k points. If the point p we added falls within one or more of the circumcircles to which the vertices of T are the center, the vertices that are centers to these circles cannot be part of the new diagram. This is because, by definition, there are no sites within the Delaunay corner circles.

As a result, the diagram is updated each time a new point is added. And in this way, a Delaunay triangulation is created by adding all the points one by one. After acquiring the final Delaunay triangulation, it is converted to its dual Voronoi diagram.

1.2.2 Fortune's Algorithm

Fortune's Algorithm consists of sweep and beach lines, both of which move in the plane. Sweep line is a straight line perpendicular to the x-axis. It is assumed to go from left to right. As the sweep line moves to the right, the sites to the left of the line will be included in the Voronoi diagram, while those on the right will not be taken into account. Beach line, on the other hand, is not a straight line. It is a line to the left of the sweep line and consists of parabola segments. Beach Line parabolas represent points midway between the Sweep Line and the point, for each point to the left of the Sweep Line. As the Sweep Line progresses, the intersections of these parabolas define the edges of the Voronoi diagram.

1.2.3 The Flipping Algorithm

The last algorithm we will consider takes its name from the method it uses to achieve the Delaunay triangulation. It flips diagonals of quadrilaterals to achieve minimum inner angles. This is the key for the Voronoi-Delaunay duality. The duality states that any Voronoi diagram can be converted to its Delaunay triangulation counterpart by only a trivial operation. 8 Thus, the main idea is using the sites of the diagram as the input point set of the Delaunay triangulation and then flipping edges of the triangles to form the Voronoi diagram.

2. Implementation Details

In the Implementation details section, we will explain the implementation details of two algorithms that have been completed and an unfinished algorithm.

2.1 Randomized Incremental Algorithm

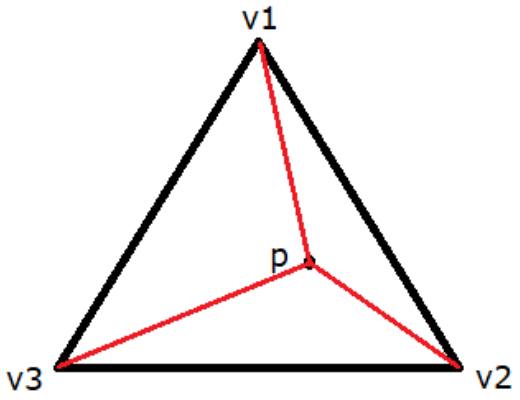
2.1.1 Summarized Algorithm

1. Find a supertriangle that encapsulates all the input points.
2. Add all the points to the triangulation in a random order and add new edges to keep triangulation.
3. After each addition, check new edges with recursion to ensure Delaunay property.
4. Discard the supertriangle with its connected edges to obtain Delaunay triangulation.
5. Connect the circumcenters of all the triangles to obtain the edges of the Voronoi diagram.
6. Add outgoing rays that are perpendicular to lone edges of the corner triangles.
7. Resulting graph is the Voronoi diagram.

2.1.2 Algorithm

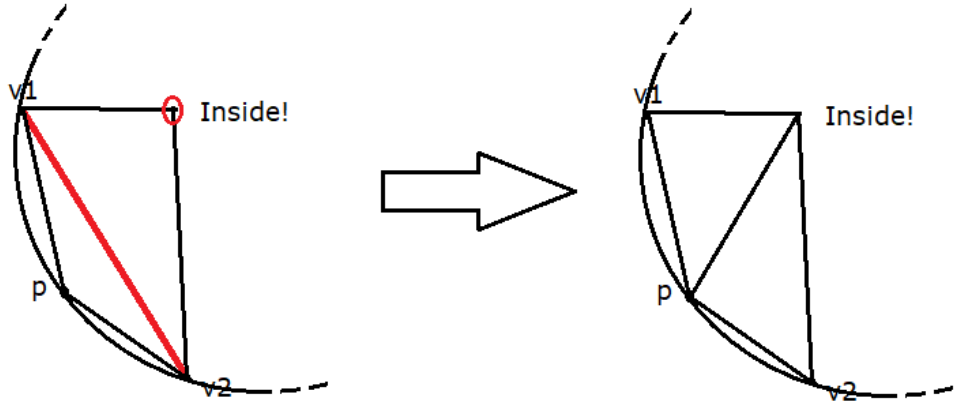
The algorithm starts by three vertices of a triangle that encapsulates all the input points. It defines the triangle using minimum and maximum of x, y values of the input point set.

Then the points are added one by one to the triangulation that started as the supertriangle. To add a point to the triangulation, the triangle that contains the point is found by testing all the triangles. After the triangle is found, it is divided into three new triangles by three edges from the point to the original triangle's vertices:

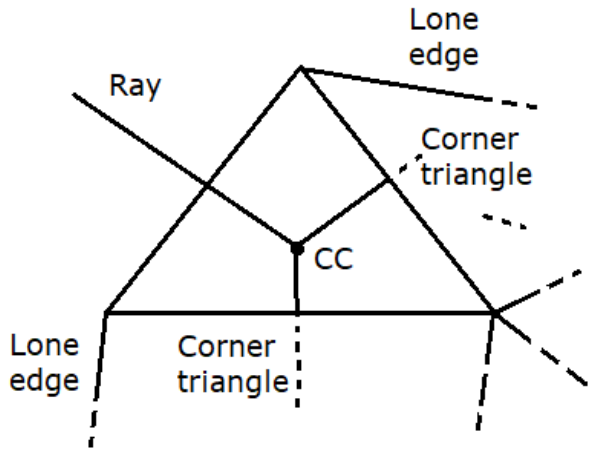


With the new point and edges, the graph still keeps the triangulation property but it may not necessarily keep the Delaunay property. Remember that there are three new triangles formed in the example above; v1-p-v2, v2-p-v3, v3-p-v1. All three needs to be checked and some edges may be flipped if necessary. This is called “legalizing”.

Let’s look to v1-p-v2 triangle from above. To check the Delaunay property, the circumcircle is formed and checked if the third vertex v3 of the triangle that shares the v1-v2 edge with the input triangle is inside the circumcircle. If it is, the edge v1-v2 is flipped to be p-v3. Then the edges v1-v3 and v2-v3 are legalized recursionally.



After all the points gone through this process, the final triangulation is the Delaunay triangulation of the point set. Then, all the triangles’ circumcenters are found and connected with their neighbor triangles’ circumcircles. One or two sides of some of the triangles do not have any neighbors. Those triangles are called corner triangles and their neighborless sides are called lone edges. For each lone edge of each corner triangle, a ray is added that starts from the circumcircle of the triangle and is perpendicular to the respective lone edge of the triangle.



Finally, the graph formed by the edges between circumcircles and rays is the Voronoi diagram.

2.1.3 Pseudocode

The single operation functions such as midpoint(edge) are skipped.

```
function legalize_edge(point p, triangle ptri, triangulation tris):
    common1, common2 = two vertices of ptri that are not p
    for tri in tris:
        if tri is the neighbor of ptri:
            last_p = the third vertex of tri that is not shared with ptri
            if last_p is in the circumcircle of ptri:
                flip common1-common2 to be p-last_p
                legalize_edge(p, new_triangle_1, tris)
                legalize_edge(p, new_triangle_2, tris)

function add_point(point p, triangulation tris):
    // A triangle is defined by three vertices v1, v2 and v3
    cont_tris = all the triangles in tris that contain p

    if cont_tris is not empty:
        if cont_tris is containing one triangle:
            tris.remove(cont_tris)
            tris.add([(v1, v2, p), (v2, v3, p), (v3, v1, p)])

            for each new triangle new_tri:
                legalize_edge(p, new_tri, tris)
        else:
```

```
                    // This means p is on an existing edge.
                    // This situation is not implemented.
function delaunay (point_set P, tkinter variables):
    v1, v2, v3 = supertriangle(P)
    triangulation = [(v1, v2, v3)]

    for point p in P:
        canvas.clear()

        add_point(p, triangulation)
        result = [tri for tri in triangulation if tri does not share any sides with supertriangle]
        for triangle tri in result:
            canvas.draw(tri)

        wait(delay_amount)
    return result // Delaunay triangulation

function voronoi(triangulation tris, tkinter variables):
    triangle_circumcenters = [circumcenter(tri) for tri in tris]
    cells = empty list
    lines = empty list

    for tri in tris:
        neighbors = [other_tri for other_tri in tris if tri != other_tri
                        and tri shares an edge with other_tri]
        cells.add({center: triangle_circumcenters[tri], neighbors: neighbors})

    for cell in cells:
        for other_cell in cell[neighbors]:
            new_line = (cell[center], other_cell[center])
            lines.add(new_line)
            canvas.draw(new_line)
            canvas.wait(delay_amount)

        if cell[neighbor] is containing less than 3 triangles:
            // This means tri is a corner triangle.
            // This is not implemented completely and currently commented out.
            find all the sides of the tri that does not have neighbors
            draw rays from tri's cc and are perpendicular to the found sides
    return lines // Voronoi diagram
```

2.1.4 Analysis

Theoretically, this algorithm can have a worst time-complexity of O(nlogn) but since our implementation is using simple lists of shapes (triangles, lines), it performs closer to $O(n^2)$ complexity. This is because the jobs like finding neighboring triangles etc. requires traversing all the inputs resulting in an extra O(n) multiplier. These can be optimized with a more interconnected data structure such as DCEL.

2.2 Fortune's Algorithm

In this section we will talk about the implementation Details of Fortune's Algorithm

2.2.1 Summarized Algorithm:

1. Create an empty BST and an empty priority queue for Beach and Sweep lines.
2. Insert sites into the priority queue.
3. while the priority queue is not empty:
   a. Pop the next event.
   b. If the event is a Site Event:
      i. Insert the point into the BST.
      ii. Check circle events created by the new site and add them to the priority queue.
   c. If the event is a Circle Event:
      i. Remove the arc associated with the circle event from BST.
      ii. Add the Voronoi vertex.
      iii. Update the edges.
      iv. Check circle events created by the removal of the arc.
4. Process the remaining infinite edges.

2.2.2 Algorithm

Firstly, the data types available in the theoretical version of the algorithm are a priority queue to contain Site Events and Circle Events located in the Sweep Line, and a Binary Search Tree to contain arcs located in the Beach Line. However, in our implementation a priority queue is used

for Events whereas a linked list is used for arcs. The effects of this on the performance of the algorithm will be examined in the analysis section of this algorithm.

The first thing to do in our Implementation, as in the theoretical version, is to put the given points in the priority queue. Then an event is removed from the queue, and it is decided whether it is a Site Event or a Circle Event. Two different ways are followed according to the type of this event.

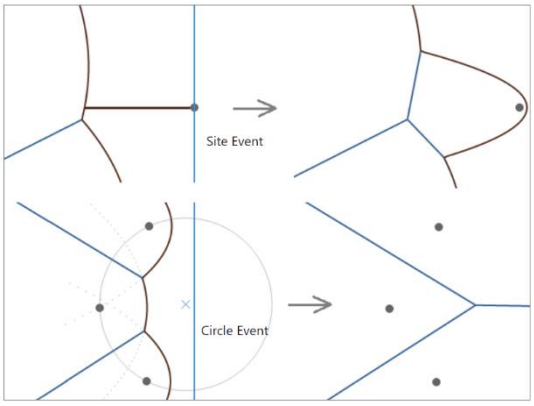


Figure 1 http://www.eecs.tufts.edu/~vporok01/c163/

- **If Event Is a Site Event**

For an event to be a site event means that it is one of the points given at the beginning. If the event is a site event, first of all, the Linked List representing the Beach Line is checked. If no arc is added, an arc is added to represent this site.

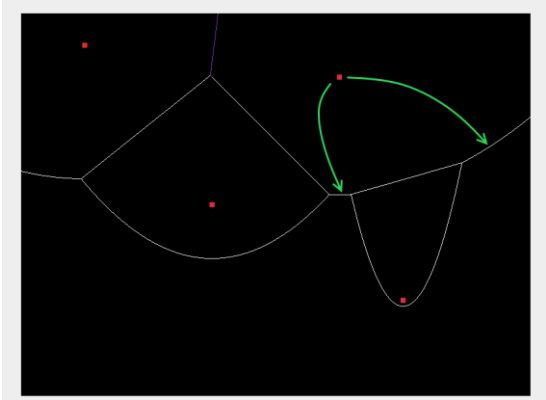


Figure 2https://jacquesheunis.com/post/fortunes-algorithm/

If there is another or more arcs in the Linked List, first it is checked whether any of these arcs intersect with the arc that the newly added point will create. If any of these arcs intersect, it is checked whether the arc after the immediately intersecting arc intersects. If it intersects at the next arc, the new arc is placed between two arcs that intersect with the new arc. If the next arc does not intersect, the first intersecting arc is bisected and the newly added arc lies in the middle of them. After the new arc is formed, the root of two (above edge and below edge of the new site) Voronoi Edges is founded at the intersection of the first intersecting arc with the new arc. (The starting points are added.) Then, newly formed arc, the arc below and the arc above are tested for if there is a Circle event waiting for them. This calculated circle event is the center of the circle formed by the sites forming the arcs on both sides of the site forming an arc.

If there are no intersecting arcs in the linked list, the new arc is added to the list. A Voronoi edge origin is added, whose X coordinate is equal to the x coordinate of the left line of the bounding

box and whose Y coordinate is the middle of the Y coordinates of the arc and the new arc below it in the list.

- If Event Is a Circle Event

As mentioned before, the Circle event is the center of the Circumcircle formed by three sites. If there is no other site within this Circumcircle, it also means that this point becomes a Voronoi vertex. So, first, if this circle event is a valid circle event, the point of the event is added as Voronoi Vertex. Then the endpoints of the edges (previously having a starting point) above and below the arc associated with this event are found and new edges are drawn. (it can be just one). And finally the associated spring is removed from the linked list.

Finally, new circle events are searched for arcs above and below the removed arc.

2.2.3 Pseudocode

Algorithm Voronoi:

```
function voronoi():
    insert points to site event and expand bounding box
    while events are not empty:
        get event
    if event is a Point:
        process point event
    else:
        process circle event
    finalize edges and draw output

function process_point_event(site_event):
    if arc is empty:
        create a new arc
    else:
        iterate through arcs:
            if intersection exists between current arc and site_event:
                modify arcs and segments accordingly
                check for new circle events
                return

        if no intersections found:
            append site_event to arc and insert new segment
```

```
function process_circle_event(circle_event):
    if circle event is valid:
        start a new Voronoi edge
        remove associated arc and update Voronoi edges
        check circle events on either side

function check_circle_event(parabola_arc):
    invalidate existing event at arc
    if arc has both previous and next arcs:
        calculate center of the circle passing through three arc points
        if circle center is right of the sweep line:
            create new circle event

function calculate_intersection(point1, point2, directrix):
    calculate intersection of parabolas defined by points and directrix
    return intersection point

function check_parabola_intersection(new_point, parabola_arc):
    if intersection exists between new_point and parabola_arc:
        calculate and return intersection point
    else:
        return None
```

2.2.4 Analysis

Theoretically, Fortune's Algorithm Voronoi can create a diagram using O(N) space and O(NlogN) time. In our implementation, the time seems to be O(NlogN) and space is O(N) again just like the original algorithm.

The Flipping Algorithm

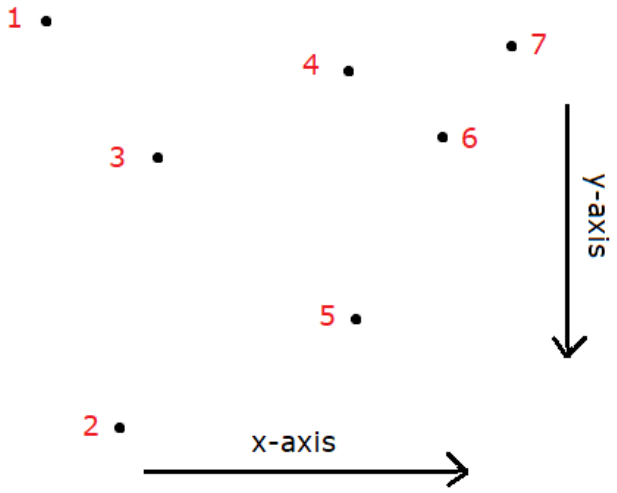
2.2.5 Summarized Algorithm

1. Sort the input points lexicographically. (By x-values and if equal, by y-values)
2. Connect the first three to form the first triangle of the triangulation T.
3. For each point p, add every edge e that starts at p and ends at a point of T if e does not cross any edge from T.
4. Result after adding every point is the lexicographic triangulation.

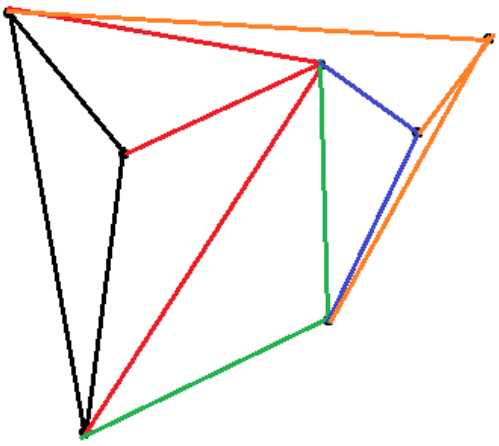
5. For every edge e of the lexicographic triangulation, if e is not locally Delaunay, push e to a stack S.
6. While S is not empty, pop an edge e from S. If e is not locally Delaunay, replace e by its respective flipped edge that is formed by the third vertices of the two incident triangles. Push other four edges of the four triangles to the S.
7. When S is empty, resulting triangulation is a Delaunay triangulation.
8. Connect the circumcenters of all the triangles to obtain the edges of the Voronoi diagram.
9. Add outgoing rays that are perpendicular to lone edges of the corner triangles.
10. Resulting graph is the Voronoi diagram.

2.2.6 Algorithm

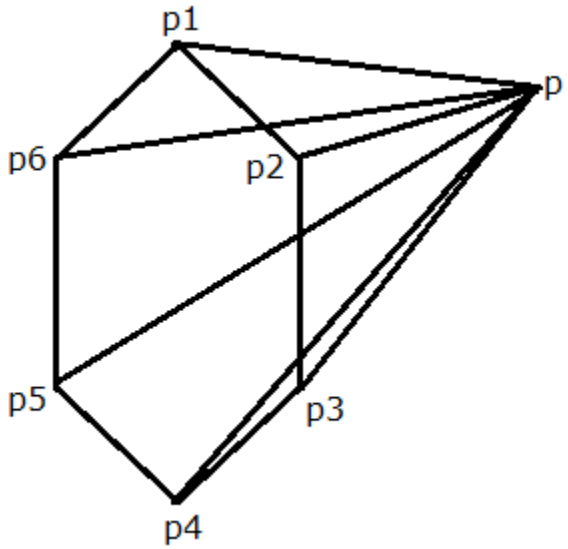
Flipping algorithm starts by sorting the input points lexicographically as described above.



The first three points becomes the first triangle of the triangulation. After the first three triangles, every point is added to the triangulation, forming new triangles.



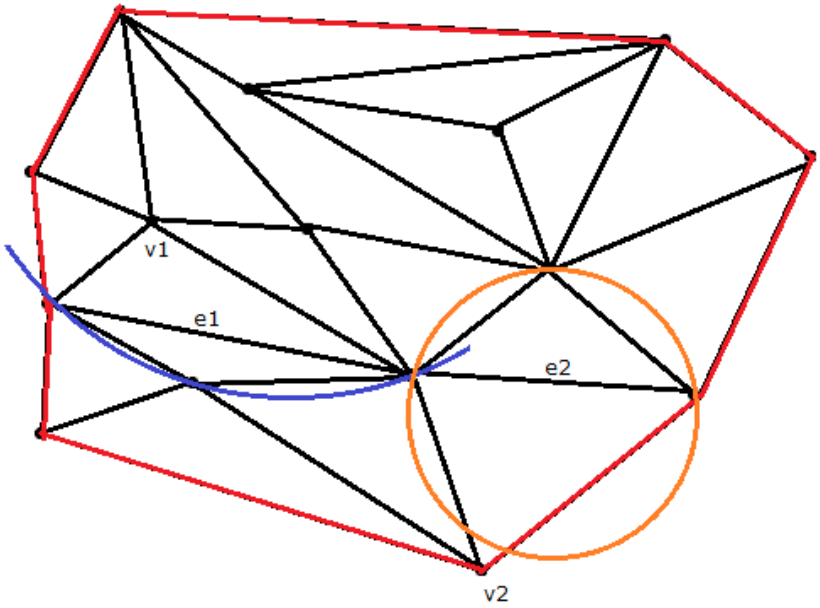
In the example above; red, green, blue and orange edges are added in that order and the black is the first triangle. At every step, convex hull of the current triangulation is calculated and used to determine new edges of the triangulation.



p1-p-p2: clockwise
p2-p-p3: clockwise
p3-p-p4: c. clockwise
p4-p-p5: c. clockwise
p5-p-p6: c. clockwise
p6-p-p1: c. clockwise

As seen, if the hexagon in the example is assumed to be the convex hull of the current state of the triangulation, edges formed with p and the points in clockwise turns (p1, p2, p3) are legal. Thus, two new triangles (p1-p-p2, p2-p-p3) are added to the triangulation for the point p. After adding all the points to the triangulation, the result is the basis for our implementation of the flipping algorithm.

For the next part, local Delauney property needs to be defined. For an edge of the triangulation to be locally Delaunay, it has to fulfill one of two conditions. It can be part of the convex hull of the triangulation or third vertex of each triangle incident to the edge is not inside the circumcircle of the other triangle.



In this example; all red edges are locally Delaunay, e1 is not locally Delaunay since v1 is in the blue circle and e2 is locally Delaunay since v2 is not in orange circle (and opposite vertex is not in the respecting circle though it is not shown).

Flipping part of the algorithm starts by an empty stack. All edges are traversed and non-locally Delaunay edges are pushed to the stack. Then while stack in not empty, an edge is popped from the stack. If the edge is not locally Delaunay and it is flippable (if the respective quadrilateral is convex), it is flipped and then four sides of the quadrilateral are pushed to the stack. After the stack is empty, resulting triangulation is globally Delaunay.

After obtaining the Delaunay triangulation, converting it to the Voronoi diagram is same as the Incremental Algorithm.

2.2.7 Pseudocode

```
function convex_hull(point_set P):
    sortedP = P sorted by x and y
    first = sortedP.pop(0)
    hull = [first]

    sortedP = sortedP sorted by joint turn direction, -y and x
    for p in sortedP:
            hull.add(p)
```

```
            while hull has more than two elements and (hull[-3], hull[-2], hull[-1]) is c. clockwise:
                hull.pop(-2)
    return hull

function poly_point_intersect(polygon poly, point p):
    valid_vertices = empty set
    n = number of vertices in poly

    for i = 1 to n:
        if (poly[i], point, poly[(i+1) % n]) is clockwise:
            valid_vertices.add(i)
            valid_vertices.add((i+1) mod n)

    edges = [(point, poly[v]) for v in valid_vertices]

function def_triangulation(point_set P):
    sortedP = P sorted by x and y

    edges = [(P[1], P[2]), (P[2], P[3]), (P[1], P[3])]
    tris = [(P[1], P[2], P[3])]

    for i, p in sortedP except the first three points: // i is the index and starts with 4
        ch = convex_hull(first i points sortedP)

        new_edges = poly_point_intersect(ch, p)
        edges += new_edges

        for every consequtive pair of edges e1, e2 in new_edges:
            tris.add((e1, p, e2))

    return edges, tris
```

The functions after this point are problematic and buggy or are not fully implemented.

```
function locally_delaunay (edge e):
    v1 = e[1]
    v2 = e[2]

    find tri1 and tri2 that share e
```

```
    v3 = [v for v in tri1 if v != e1 and v != e2]
    if v3 is not empty:
            v3 = v3[1]
            v4 = [v for v in tri2 if v != e1 and v != e2]
            if v4 is not empty:
                    v4 = v4[1]

    if point_in_circle(v3, circumcircle(v1, v2, v4)) or point_in_circle(v4, circumcircle(v1, v2,
v3)):
            return false
    return true

function flip (edge e):
    v1 = e[1]
    v2 = e[2]

    find tri1 and tri2 that share e
    v3 = [v for v in tri1 if v != e1 and v != e2]
    if v3 is not empty:
            v3 = v3[1]
            v4 = [v for v in tri2 if v != e1 and v != e2]
            if v4 is not empty:
                    v4 = v4[1]

    edges.remove(edge)

    tris.remove(tri1)
    tris.remove(tri2)

    tris += [(v1, v3, v4), (v3, v4, v2)]

    new_edge = (v3, v4)
    edges.add(new_edge)

    return new_edge

function flip_edges (edges, tris):
    stack = empty stack

    for edge in edges:
```

```
            if not locally_delaunay(edge):
                stack.push(edge)

    while stack is not empty:
            next_edge = stack.pop()

            if next_edge in edges:
                new_edge = flip(next_edge)

                if not locally_delaunay(new_edge)
                    stack.append(new_edge)
```

Resulting edges list is the Delaunay triangulation and converting to the Voronoi diagram is the same as the Incremental Algorithm as said before.

2.2.8 Analysis

Since the code is not working, it cannot be tested practically. Considering it's theoretical worst time-complexity is $O(n^2)$ and we are using lists of triangles and edges as data structures, our implementation's time-complexity would be asymptotically larger, probably at $O(n^3)$. It has the same inefficiencies as the Incremental Algorithm implementation.

3. Results Of A project

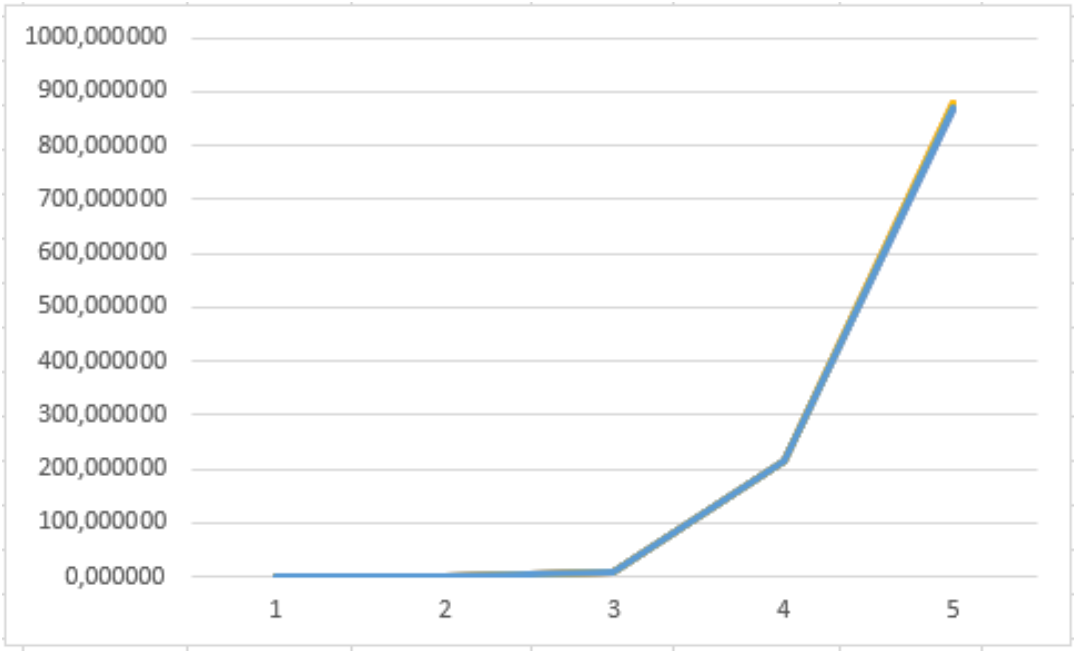
In this section, we will share the results of the time that the algorithms can achieve. We will do the tests by measuring the times of 100, 500, 1000, 5000,10000 points. Note that our computers were busy with many programs such as word, excel, IDEs etc. at the performance tests.

3.1 Randomized Incremental Algorithm

The results we got at the end of the two tests we made in this algorithm are as follows. The results are in seconds.

Test	Increase Multiplier	Average	Increase Multiplier2	1st Test	2nd Test	3rd Test	4th Test	5th Test
100	1	0,071929	1	0,068364	0,070128	0,069069	0,074440	0,077646
500	5	2,005885	27,88685962	2,037960	2,011545	1,955151	2,016450	2,008320
1000	2	8,219128	4,097506762	8,005747	8,250260	8,305079	8,191455	8,343098
5000	5	213,797859	26,01223242	215,230592	213,736928	213,727476	213,046349	213,247953
10000	2	873,535911	4,085802884	867,729055	878,130396	871,570660	879,781720	870,467724

Multipliers from the experiments supports our $O(n^2)$ time-complexity claim as seen.

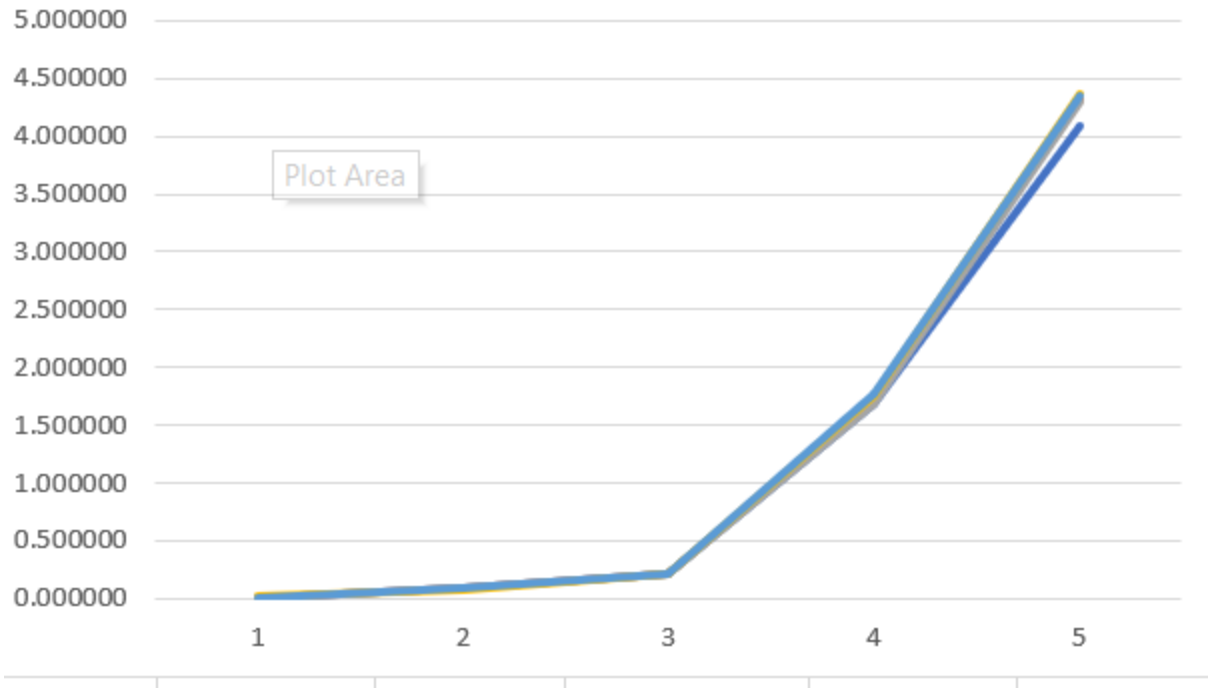


All five test are present in this graph but they are on top of one another.

3.2 Fortune's Algorithm

The results we got at the end of the two tests we made in this algorithm are as follows. The results are in milliseconds.

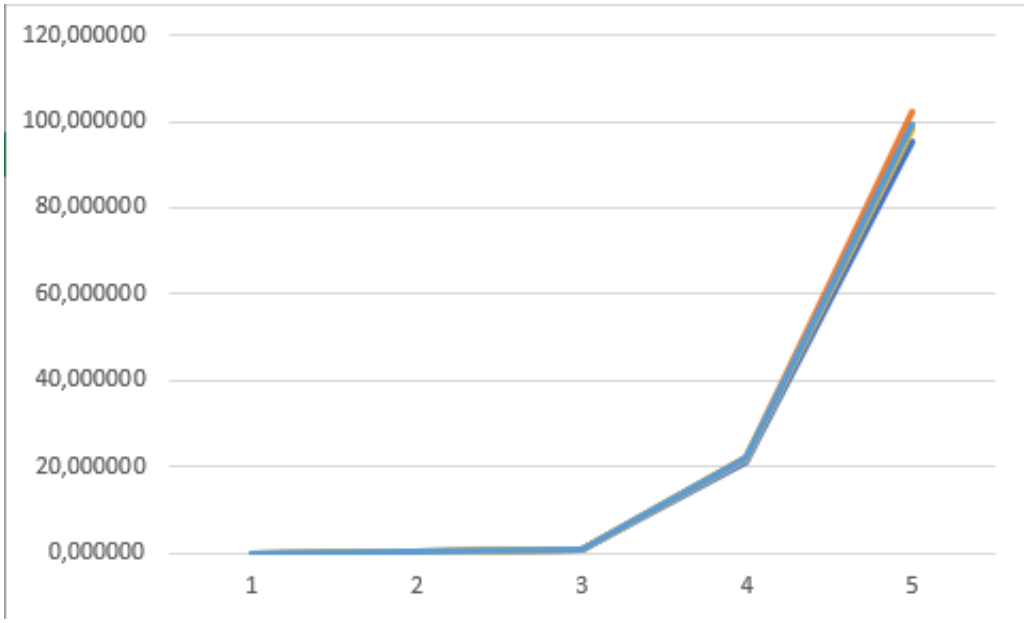
Test	Increase Multiplier	Average	Increase Multiplier2	1st Test	2nd Test	3rd Test	4th Test	5th Test
100	1	0.012858	1	0.010897	0.012983	0.013277	0.015737	0.011397
500	5	0.085591	6.656543974	0.085338	0.089719	0.086140	0.081627	0.085133
1000	2	0.210097	2.454648268	0.205720	0.207401	0.207183	0.218428	0.211753
5000	5	1.723742	8.20451108	1.681488	1.722963	1.684916	1.757998	1.771346
10000	2	4.284489	2.48557395	4.091304	4.316465	4.302746	4.370570	4.341358



3.3 The Flipping Algorithm

Only Lexicographic triangulation is working for The Flipping Algorithm thus we only tested that part.

Test	Increase Multiplier	Average	Increase Multiplier2	1st Test	2nd Test	3rd Test	4th Test	5th Test
100	1	0,007289	1	0,008217	0,007152	0,006994	0,007234	0,006846
500	5	0,189906	26,05526728	0,199537	0,189228	0,190253	0,185072	0,185440
1000	2	0,754723	3,974193535	0,755689	0,775956	0,738861	0,752312	0,750797
5000	5	21,685285	28,73278152	20,931617	21,980373	21,664345	21,841470	22,008622
10000	2	98,870270	4,559325296	95,538187	102,504401	98,412021	98,526559	99,370182



This part is $O(n^2)$ as well since for every point, convex hull of the triangulation is calculated.