CS 478 Computational Geometry

Spring 2023

Implementing Three Voronoi Diagram Computation Algorithms and Comparing Their Performance

Progress 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 visualizing 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:

- Randomized 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 decided to use Python as our programming language, and Pygame as our API to draw both the UI and visualization of the Voronoi Diagram. We will use Git to share our coding environment.

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.

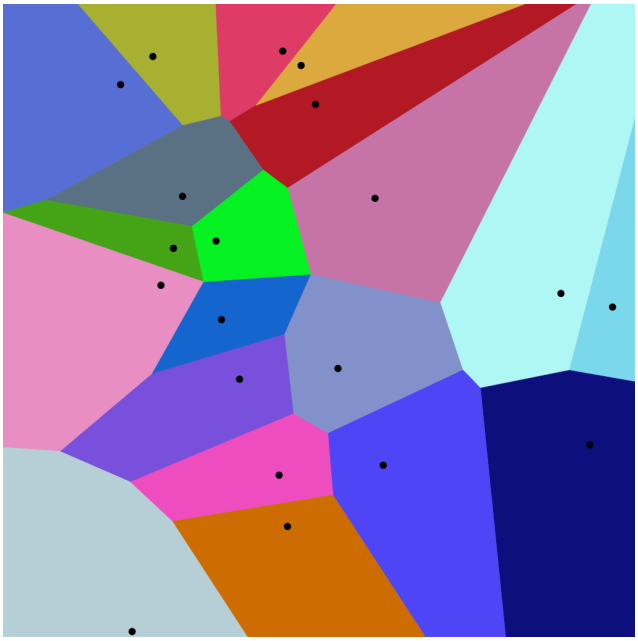

Figure 1: 20 points and their Voronoi cells [1]

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 planar straight line graph **G** = (**V**, **E**) that represents the Delaunay triangulation

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

1. For each face **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. Insert a vertex **u**=(**e.v1**+**e.v2**) / 2 to **G'**
     - 1.3.2. Insert an edge **e'= (u**, **C)** to **G'.E**.
   - 1.4. Find the neighboring triangles of **T**, **N(T)**.
   - 1.5. For each neighboring 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**
     - 1.5.2. Remove the edges (**C, u**) and (**u**, **C_n**), and remove the vertex **u**; if such vertex **u** exists
2. For each edge **e'** in **G'**:
   - 2.1. If **e'.v1** is a circumcenter and **e'.v2** is not or vice versa:
     - 2.1.1. Extend **e'** into infinity from the vertex that is not a circumcenter

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 neighboring triangles of a triangle is already available in constant time if a doubly connected edge list (DCEL) data structure is used [7].

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:

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

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

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

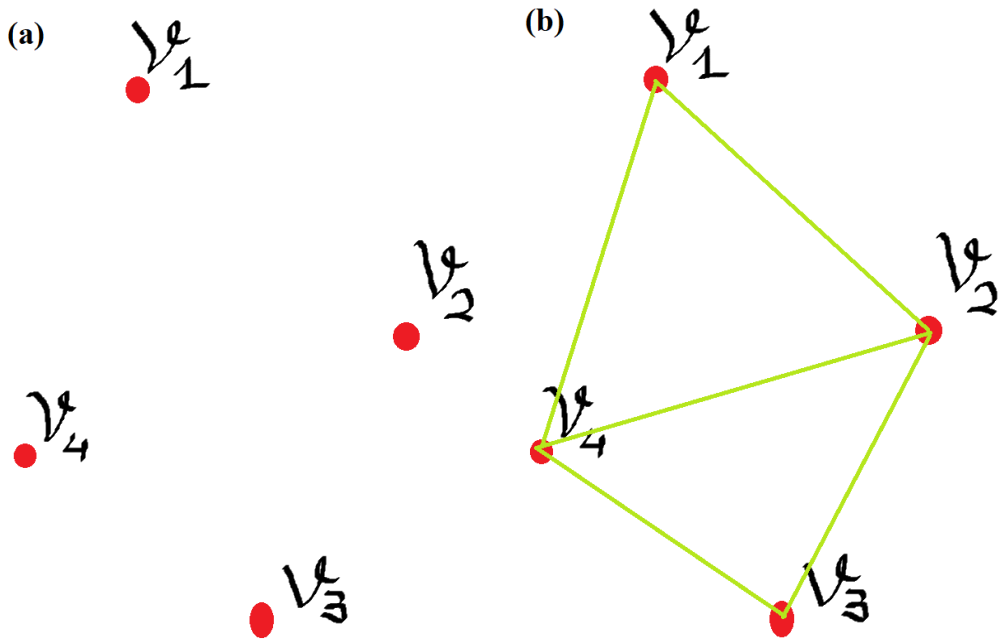

Figure 2: (a) Starting set for delaunay triangulation, and (b) its corresponding delaunay triangulation

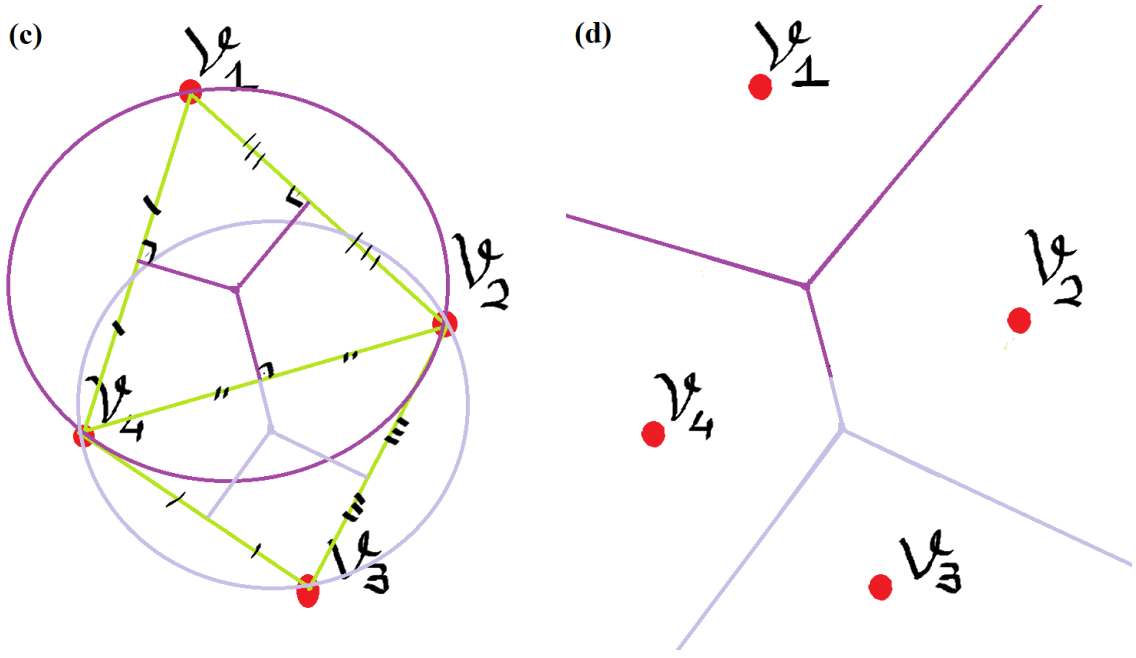

Figure 3: As discussed above, we can draw each excircle of a triangle, draw bisectors from each center to the corresponding edges, and connect each neighboring triangle's circumcenter, as done in (c). After that, we can remove all triangles and circles, and extend the bisectors to get the Voronoi diagram,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

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= (V, E)**

1. Let **P'** be a random permutation of P, **P'** ∈ S(**P**).
2. Initialize an empty triangulation **T = (V, E)**
3. Let T_s be the super triangle that contains all points in **P'**
4. Add T_s into **T**
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**

6. For each triangle T_i in **T**:

6.1. If there is a vertex **u** $\in T_i$ s.t. **u** $\in T_s$

6.1.1. Remove T_i from **T**

7. Return **T**

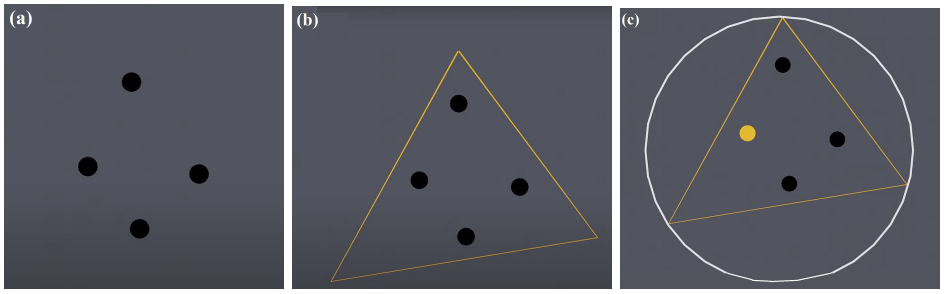

Figure 4: Initialization of the Bowyer-Watson algorithm with a super triangle (b) for the points in (a). The super triangle's circumcircle contains the next random paint, the super triangle will be marked as a bad triangle (c).

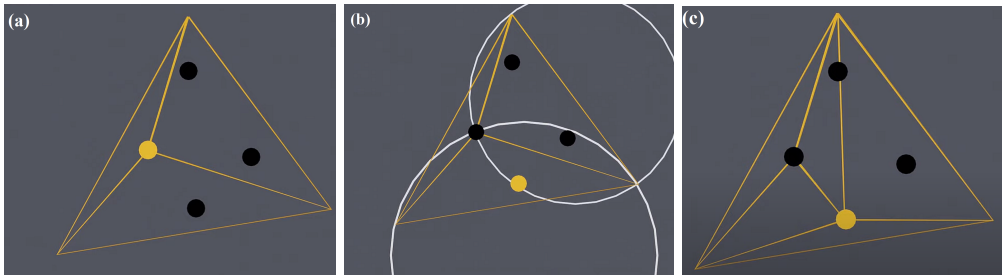

Figure 5: The convex hull of the super triangle is itself, therefore new edges will be drawn from the vertices of that convex hull towards the random point. Then, all the bad triangles are removed from the triangulation, therefore the circumcircle of the super triangle will not be checked again (a).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

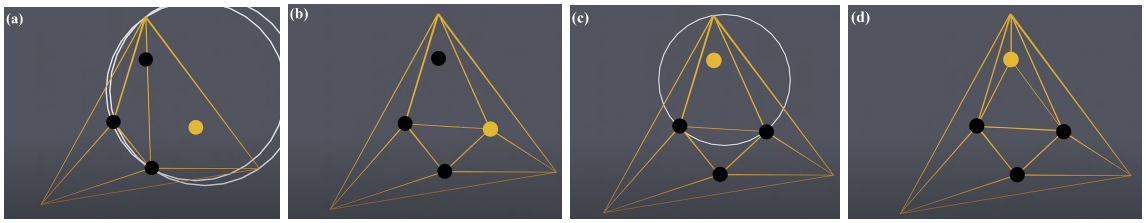

Figure 6: The iteration continues for the 3rd (a, b) and the last random point (c, d).

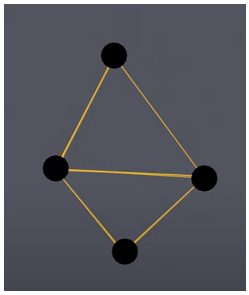

Figure 7: The final result of the triangulation, after removing all triangles whose edges are shared by the super triangle

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 [8, 9]:

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. While **Q** is not empty**:**
    3.1. Let **q** be the next event in **Q**
    3.2. Dequeue 1 item from Q
    3.3. Cache the y coordinate of **q** as LastY
    3.4. If **q** was visited
        3.4.1. Mark **q** as not visited
        3.4.2. Continue
    3.5. If **q** is a place event:
        3.5.1. Create a new parabola from **q.p** and add it to **G.E**
    3.6. Otherwise:
        3.6.1. Remove the parabola **q.arch** from **G.E**
    3.7. Mark **q** as visited
    3.8. Finish the root edge
    3.9. For each edge **e** in **E**
        3.9.1. If **e** has a neighboring edge:
            3.9.1.1. Delete it from **E**
4. Return **G**

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. The new events are created inside the routines 3.5.1 and 3.6.1. Those routines are constant time except for the priority queue operations [8, 9].

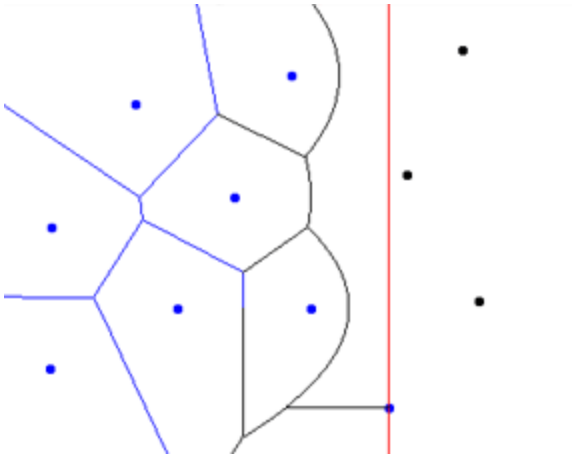

Figure 4: Visualization of an iteration of Fortune's algorithm [8, 9].

3.3. The Flipping Algorithm

The Flipping algorithm [4] is another algorithm that generates the Delaunay Triangulation of a set of points. The algorithm needs to start with an arbitrary triangulation, therefore, to use this algorithm, a preprocessing needs to be done to generate an arbitrary triangulation from a set of points. For this, we decided to use the simplified version of Lexicographic Triangulation [4], whose steps are explained below.

Steps for generating Lexicographic Triangulation:

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

Output: A planar straight line graph **G=(V, E)** representing the triangulation

1. Sort **P** with respect to the x-axis in ascending order.
2. Create an empty planar straight line graph **G=(V, E)=({}, {})**.
3. Insert the first point in P, p_1 into **G**, and remove p_1 from **P**.
4. For each point p_i in **P:**
   - 4.1. Insert p_i into **G.V**
   - 4.2. For each vertex **v** that is not p_i in **G.V**:
     - 4.2.1. Let **b** = 1
     - 4.2.2. For each edge **e** in **G.E**:
       - 4.2.2.1. If the line segment **L**, from **v** and p_i does not intersect **e:**

4.2.2.1.1. Set **b** = 0

4.2.3. If **b** is 1:

4.2.3.1. Insert the edge (**v**, p_i) into **G.E**

5. Return **G**

The worst-case time complexity of the lexicographic triangulation algorithm is O(N^3). The algorithm above can be implemented by finding the convex hull of vertices of **G** and p_i combined (conv(V(**G**)∪ p_i)), and then adding edges of this convex hull, and the interior edges from p_i to the vertices of the previous convex hull if this vertex does not intersect any other edge instead, as discussed in [4]. However, this version would still iterate all the edges in **G** for all vertices of the previous convex hull in each iteration, therefore would still have the time complexity O(N^3). Therefore, for simplicity, the algorithm above is used instead since finding the convex hull of a set of points is another difficult problem like Delaunay Triangulation.

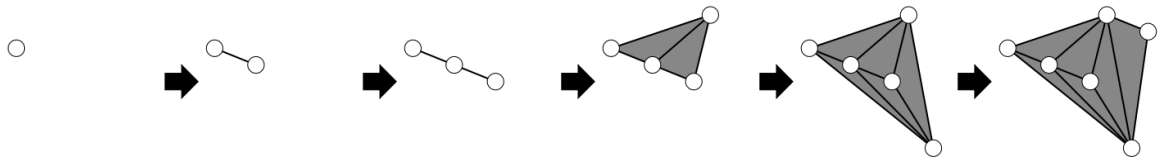

Figure 5: Incremental construction of a lexicographic triangulation from a set of points [4]

After finding an arbitrary triangulation **G**, the flipping algorithm converts this triangulation into the Delaunay triangulation of **P**, using the following steps [4]:

Input: A set of points **P** = {$p_1, p_2,..., p_n$}, and a planar straight line graph **G**=(**V**, **E**) representing the lexicographic triangulation of **P**

Output: A planar straight line graph **G'** representing the Delaunay triangulation of **P**

1. Create a copy of **G** as **G' = (V, E)**.
2. While there exists an edge **e** in **G'.E** that is not locally Delaunay:
   2.1. Let **e**=(**a**, **b**) be the endpoints of e, and **e'**=(**c**, **d**) be the other diagonal of the quad **Q**=(**a**, **b**, **c**, **d**) formed by faces of **e, e.RightF** and **e.LeftF**,
   2.2. If **Q** is strictly convex

2.2.1. Remove edge **e** from **G'**

2.2.2. Add edge **e'** into **G'**

3. Return **G'**.

The worst-case time complexity of the algorithm above is O(N^2), as the algorithm might have to check the whole graph to find an illegal edge in each iteration.

It should be noted that checking if an edge **e** is locally Delaunay or not is a constant time operation since there are always three edges and vertices in a triangulated graph. The operation can be computed like the following:

1. Let the faces of **e** be **F1** and **F2**.
2. Compute the circumcenter of **F1** and **F2** as **C1** and **C2**, where **r1** and **r2** are the radius of the circumcircles of **F1** and **F2**
3. For each vertex **v** in **F1**:
   3.1. If the distance between **v** and **C2** is smaller than or equal to **r2**
      3.1.1. Return **False**
4. For each vertex **u** in **F2**:
   4.1. If the distance between **u** and **C1** is smaller than or equal to **r1**
      4.1.1. Return **False**
5. Return **True**

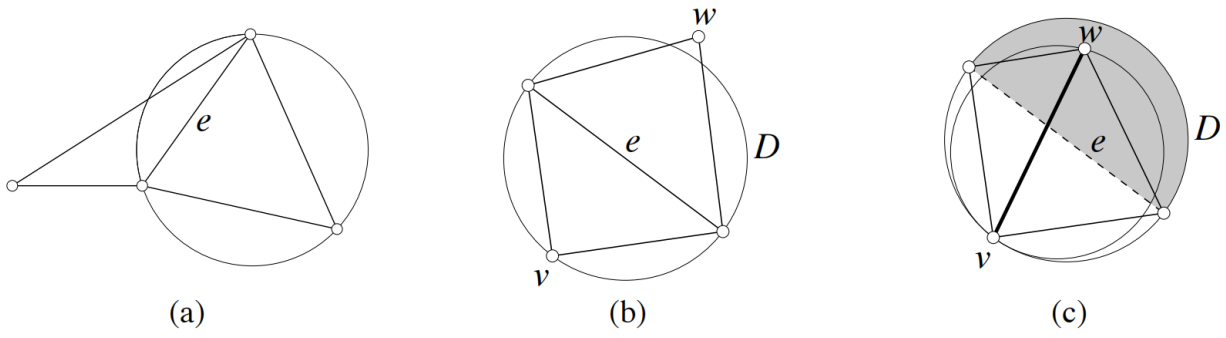

Figure 6: Legality checks for three different edges in (a), (b), and (c). (a) is not convex, therefore it is skipped. (b) is convex and edge e is locally Delaunay, therefore no flipping was made. However, in (c), the triangle below e has a circumcircle which also includes w, therefore e must be flipped into e'=(w, v) [4].

In total, the Flipping algorithm will have O(N^3) worst-case time complexity for each test case, since preprocessing will be made for each test case.

4. Future Work

So far, we have completed the literature survey of the algorithms to be used, and given certain pseudocodes for each algorithm and their preprocessing steps, if any. In the next month, we will implement these algorithms in Python and test their performances using the Delaunay triangulation datasets on the University of California website [10].

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] 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]

[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] 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].

[8] 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].

[9] phlummox-mirrors. “Ivan Kutskir's Voronoi Diagram implementation in C++”, GitHub, 2020. [Online]. Available: https://github.com/phlummox-mirrors/ivank-voronoi . [Accessed Mar. 24, 2023].

[10] J. Shewchuk. “Delaunay Triangulation Project”, University of California at Berkeley, 2019. [Online]. Available: https://people.eecs.berkeley.edu/~jrs/274/proj.html. [Accessed Mar. 24, 2023].