CS 478 - Project Progress Report

Implementing Two Voronoi Diagram Computation Algorithms and Comparing Their Performance

Barkın Saday

21902967

Contents

1 Introduction

Voronoi diagrams are fundamental structures in computational geometry with applications in fields such as meteorology, urban planning, computer graphics, and machine learning. Given a set of points (called sites) in a plane, a Voronoi diagram partitions the plane into regions, where each region consists of all points closest to a particular site. The resulting tessellation provides insights into spatial relationships and proximity-based structures.

This project aims to implement and compare three distinct Voronoi diagram computation algorithms: the *Randomized Incremental Algorithm* and *Fortune's Algorithm* (Sweepline). These approaches offers unique advantages in terms of computational efficiency and practical use cases. The primary goal is to analyze their performance in terms of execution time and efficiency across various input sizes and distributions.

To facilitate visualization and analysis, an interactive software application will be developed. This application will provide tools for zooming, translating, and observing the step-by-step execution of Fortune's Algorithm. Additionally, experimental tests will be conducted on different datasets to compare the performance of these algorithms. The final report will summarize the findings, highlighting the advantages and trade-offs of each approach.

This progress report outlines the theoretical background of Voronoi diagrams, details the selected algorithms, discusses data structures and implementation strategies, and presents an initial experimental design for performance evaluation.

2 Background and Literature Review

2.1 Definition of Voronoi Diagrams

A Voronoi diagram is a fundamental geometric structure that partitions a plane based on proximity to a given set of points, known as *sites*. Formally, given a set of sites $S = \{s_1, s_2, ..., s_n\}$ in a two-dimensional plane, the Voronoi region $V(s_i)$ for a site s_i is defined as:

$$V(s_i) = \{p \in \mathbb{R}^2 \mid d(p, s_i) \leq d(p, s_j) \text{ for all } j \neq i\} \tag{1}$$

where $d(p, s)$ denotes the Euclidean distance between point p and site s. The collection of all such regions forms the Voronoi diagram of S.

Voronoi diagrams are widely used in computational geometry and have applications in fields such as nearest neighbor search, spatial analysis, and mesh generation [1].

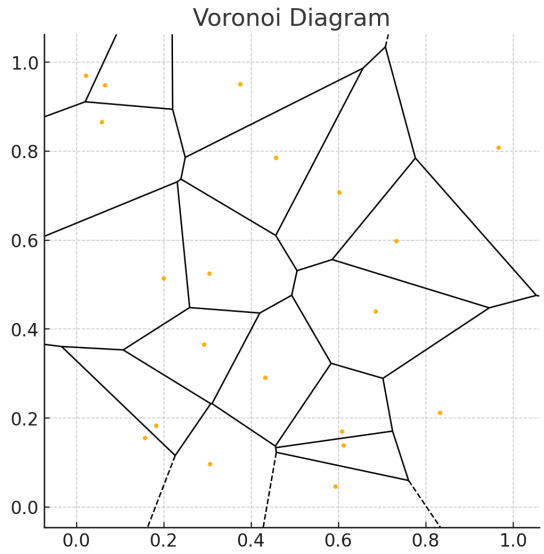

Figure 1: Example of a Voronoi Diagram for a given set of points.

2.2 Delaunay Triangulation and Its Relation to Voronoi Diagrams

The *Delaunay triangulation* is a geometric structure closely related to the Voronoi diagram. Given a set of points, its Delaunay triangulation is a triangulation where no point in the set lies inside the circumcircle of any triangle.

A key property is that the Delaunay triangulation is the *dual graph* of the Voronoi diagram. This relationship is useful in computational geometry, as many Voronoi diagram algorithms leverage Delaunay triangulation for efficient computation.

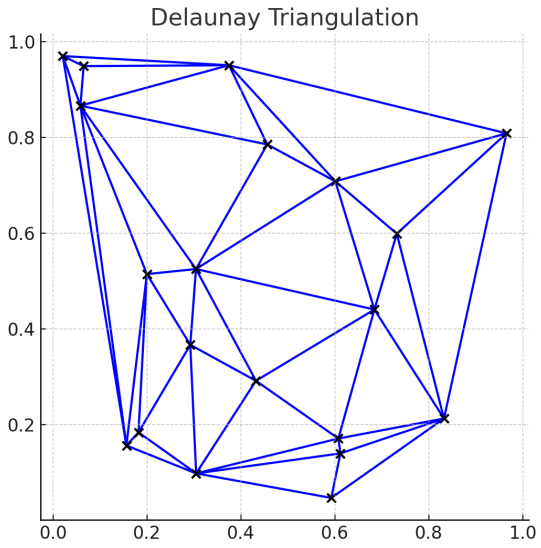

Figure 2: Relationship between Voronoi Diagram (black edges) and Delaunay Triangulation (blue edges).

2.3 Applications of Voronoi Diagrams

Voronoi diagrams are widely used in various fields:

- **Geographical Mapping:** Used for spatial partitioning, such as defining regions of influence for cell towers.
- **Computer Graphics:** Helps in texture synthesis and mesh generation.
- **Path Planning:** Used in robotics and AI for finding optimal paths.
- **Machine Learning:** Used in clustering algorithms like k-means.

2.4 Computational Complexity

Different algorithms exist for computing Voronoi diagrams, each with different computational complexities:

- **Naïve approach:** $O(n^2)$ complexity.
- **Fortune's Algorithm:** Optimal $O(n \log n)$ time complexity.
- **Randomized Incremental Algorithm:** Expected $O(n \log n)$ complexity.
- **Flipping Algorithm:** Based on iterative improvements, not strictly $O(n \log n)$.

A comparison of these algorithms will be the focus of this project.

3 Algorithms and Design Decisions

3.1 Randomized Incremental Algorithm

3.1.1 Overview

The *Randomized Incremental Algorithm* constructs a Voronoi diagram by inserting sites in a random sequence, updating the diagram incrementally with each addition. This method offers expected $O(n \log n)$ time complexity, making it efficient for practical applications [2].

3.1.2 DCEL Data Structure

To manage the planar subdivision of the Voronoi diagram, we employ the **Doubly Connected Edge List (DCEL)**. This structure captures the relationships between vertices, edges, and faces, facilitating efficient traversal and updates [3].

Each edge in the DCEL is represented by two half-edges with the following fields:

- **V1 (Origin):** Starting vertex of the half-edge.
- **V2 (Terminus):** Ending vertex; defines the half-edge's direction.
- **F1 (Left Face):** Face to the left of the half-edge.
- **F2 (Right Face):** Face to the right of the half-edge.
- **P1 (Next Edge at V1):** Next half-edge encountered counterclockwise around V1.
- **P2 (Next Edge at V2):** Next half-edge encountered counterclockwise around V2.

This structure is particularly suitable for Voronoi diagrams due to its ability to efficiently represent planar subdivisions. Also, it is a structure that is discussed in detail in lectures which makes me familiar with it.

3.1.3 Algorithm Steps (On high-level)

1. **Initialization:** Start with an initial Voronoi diagram of three non-collinear points, forming a triangle.
2. **Randomized Insertion:** Insert each subsequent site in a random order to maintain expected $O(n \log n)$ performance, similar to the strategy used in randomized quicksort.
3. **Locate Affected Region:** Identify the Voronoi cell containing the new site. This can be optimized using a conflict graph or point location structures.
4. **Update DCEL:**

- *Split Existing Cell:* Modify the DCEL to incorporate the new site's cell by updating half-edges and vertices.
- *Adjust Neighboring Cells:* Update adjacent cells to reflect the changes, ensuring the diagram's integrity.

5. **Handle Degeneracies:** Implement strategies to manage special cases like collinear points or sites lying on existing edges.

3.1.4 Time Complexity

Even though in worst case the time complexity is $O(n \log n)$. By inserting sites in a random sequence, the algorithm achieves an expected time complexity of $O(n \log n)$. This randomness prevents adversarial inputs from causing worst-case scenarios and ensures the randomness is within the algorithm not in the given input, ensuring robust performance [2].

3.1.5 Implementation Considerations

- **Precision Handling:** Address numerical precision issues to maintain the accuracy of the diagram.
- **Dynamic Updates:** The DCEL's flexibility allows for efficient insertion and deletion of sites, supporting **dynamic** applications.

3.2 Fortune's Algorithm (Sweepline)

3.2.1 Concept and Intuition

Fortune's Algorithm is an efficient method for constructing Voronoi diagrams, operating in optimal $O(n \log n)$ time complexity. It utilizes a sweepline approach, moving a horizontal line from bottom to top across the plane, and dynamically maintains the *beach line*—a complex curve composed of parabolic arcs that represents the boundary between processed and unprocessed regions [4].

Unlike traditional implementations where Fortune's Algorithm is executed in a single batch process, our approach extends it to support **dynamic updates**, allowing sites to be added or moved after the initial Voronoi diagram is computed. This modification ensures real-time updates without recomputing the entire diagram from scratch.

3.2.2 Data Structures

To implement Fortune's Algorithm effectively, the following data structures are employed:

- **Event Queue:** A priority queue that stores events—either *site events* (when the sweepline encounters a new site) or *circle events* (when a disappearing arc creates a Voronoi vertex). Events are processed based on their y-coordinates.

- **Beach Line:** Represented as a balanced binary search tree (BST), such as a Red-Black Tree or AVL Tree, where each node corresponds to an arc of a parabola. This structure allows for efficient insertion, deletion, and traversal operations [4].
- **Doubly Connected Edge List (DCEL):** Used to store the resulting Voronoi diagram by maintaining the relationships between vertices, edges, and faces. This structure facilitates efficient traversal and manipulation of the planar subdivision [3].
- **Dynamic Event Handler:** A structure to efficiently manage the insertion of new points or the relocation of existing ones by identifying affected regions and locally adjusting the beach line instead of recomputing the full diagram.

3.2.3 Algorithm Steps

1. **Initialization:** Insert all site events (each associated with a site point) into the event queue, prioritized by their y-coordinates.
2. **Event Processing:** While the event queue is not empty, extract the event with the highest priority (lowest y-coordinate):
    - **Site Event:** Occurs when the sweepline encounters a new site. A new arc is created on the beach line, and the DCEL is updated to reflect new edges.
    - **Circle Event:** Occurs when an arc on the beach line disappears, indicating the formation of a Voronoi vertex. The algorithm records this vertex and updates the DCEL accordingly.
3. **Updating the Beach Line:** The beach line undergoes dynamic changes as arcs are added or removed. The balanced BST allows for efficient updates and queries, ensuring the algorithm maintains its optimal performance.
4. **Handling Dynamic Site Insertions and Movements:**
    - **Insertion of New Sites:** Instead of reprocessing the entire diagram, locate the affected region, introduce a new site event, and update the beach line locally.
    - **Movement of Existing Sites:** Remove the affected site from the diagram, update neighboring Voronoi edges, and reinsert it at the new location with minimal adjustments.
5. **Finalization:** Once all events are processed, the DCEL contains the complete Voronoi diagram, representing all vertices, edges, and faces.

3.2.4 Time Complexity

Fortune's Algorithm achieves an optimal time complexity of $O(n \log n)$. This efficiency arises from the logarithmic time operations associated with the balanced BST managing the beach line and the priority queue handling events. Each of the n sites generates a site event, and there are at most $2n - 5$ circle events, leading to a total of $O(n)$ events processed [4].

For dynamic updates:

- **Insertion of a new site:** Expected $O(\log n)$ if using localized updates instead of full reprocessing.
- **Movement of an existing site:** Expected $O(\log n)$ if only affected regions are recomputed.

This ensures that the algorithm remains highly efficient even in interactive applications.

3.2.5 Implementation Considerations

- **Precision Handling:** Numerical precision is crucial, especially when calculating intersections of parabolas and handling circle events. Implementations should use robust floating-point arithmetic to mitigate errors [5].
- **Degenerate Cases:** Special cases, such as collinear points or multiple points lying on a common circle, require careful handling to ensure the algorithm's correctness. Implementing symbolic perturbation or simulation of simplicity techniques can address these issues [6].
- **Dynamic Adjustments:** Efficiently managing real-time insertions and modifications requires a well-designed event management system, reducing the number of redundant computations.
- **Visualization:** Step-by-step visualization of the algorithm can enhance understanding. By illustrating the progression of the sweepline, the evolution of the beach line, and the formation of Voronoi edges and vertices, one can gain deeper insights into the algorithm's dynamics [7].

4 Performance Expectations and Experimental Design

4.1 Performance Metrics

To evaluate the efficiency of the implemented Voronoi diagram algorithms, we will measure the following performance metrics:

- **Execution Time:** The total time taken to construct the Voronoi diagram for different input sizes.

- **Efficiency of Dynamic Updates:** Time taken to insert a new site or move an existing site.
- **Memory Usage:** The space required to store the diagram, particularly for large inputs.
- **Effect of Different Point Distributions:** The impact of input distribution (uniform, Gaussian, clustered) on execution time and efficiency.

4.2 Experimental Setup

To ensure a robust evaluation, we will conduct experiments using datasets of varying sizes, ranging from 100 to $1,000,000$ points. We will generate these datasets using the following distributions:

- **Uniform Distribution:** Points are spread evenly across the plane.
- **Gaussian Distribution:** Points are concentrated around a mean value.
- **Clustered Distribution:** Points are grouped in localized regions.

Each experiment will be repeated multiple times, and the results will be averaged to account for variations due to randomness and system conditions.

4.3 Expected Complexity Analysis

Theoretical complexity expectations for both algorithms are as follows:

Algorithm	**Worst-Case Complexity**	**Expected Complexity**	**Supports Dynamic Updates?**
Randomized Incremental	$O(n^2)$	$O(n \log n)$	Yes
Fortune's Algorithm	$O(n \log n)$	$O(n \log n)$	Yes (Modified)

Table 1: Theoretical Complexity of Implemented Algorithms

The **Randomized Incremental Algorithm** is expected to handle incremental site insertions efficiently due to its localized updates. The **Modified Fortune's Algorithm** will use a localized beach line update strategy to support dynamic site movements while maintaining its theoretical efficiency.

4.4 Comparison with External Libraries

To validate our implementations, we will compare their performance against established computational geometry libraries:

- **SciPy's 'scipy.spatial.Voronoi'**: A widely used implementation based on Qhull.

- **Qhull Library**: A robust computational geometry library optimized for convex hulls and Voronoi diagrams [8].

These comparisons will help determine the relative efficiency of our methods in real-world applications.

4.5 Visualization of Results

To effectively present our findings, we will include:

- **Execution Time Graphs:** Performance trends for different dataset sizes.
- **Heatmaps:** Visualizing the effect of input distributions on performance.
- **Step-by-Step Algorithm Visualizations:** Demonstrating how the Voronoi diagram evolves during algorithm execution.

These visualizations will be crucial in understanding the efficiency and behavior of each algorithm under different conditions.

5 Visualization and User Interface Design

5.1 Interface Features

To make the system intuitive and informative, the user interface (UI) is designed to be interactive, modular, and visually clear. Key features include:

- **Point Generation Panel (Initializing):** Allows users to generate input points using various distributions such as Uniform, Gaussian, or Clustered. Users can specify the number of points and distribution parameters, including seed values.
- **Interactive Canvas:** Displays the 2D Voronoi diagram. Users can zoom in/out using the scroll wheel and pan across the canvas via mouse dragging.
- **Algorithm Selector:** A dropdown or set of radio buttons to choose the algorithm for visualization — either the Randomized Incremental Algorithm or Fortune's Algorithm.
- **Dynamic Manipulation Tools:** Users can click to insert new points or drag existing points to new locations (dragging is optional). The Voronoi diagram updates in real time, reflecting changes immediately without full recomputation.
- **Step-by-Step Controls:** Includes play, pause, and step-forward controls to animate the execution of the algorithm. A slider or timeline allows the user to scrub through each step of the algorithm visually.

5.2 Visual Design Mockups

The following mockups illustrate the planned layout and visual elements of the UI. These images serve as placeholders and will be replaced by finalized designs and screenshots from the working implementation.

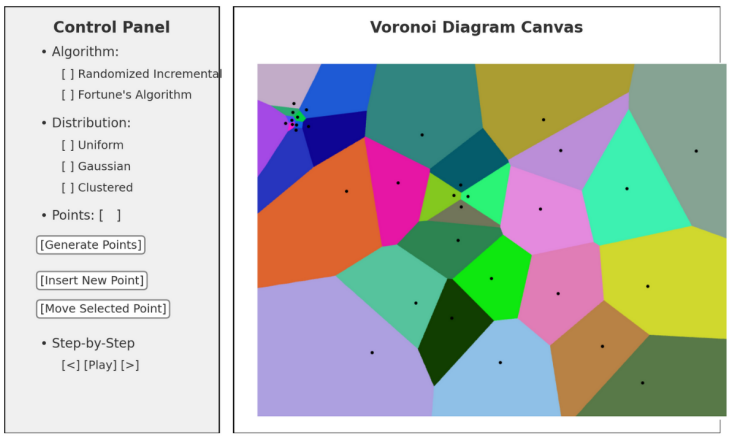

Figure 3: Mockup of UI layout showing control panel and visualization canvas.

5.3 Rendering and Visualization Techniques

To construct the visualizations, the following rendering strategies are adopted:

- **Voronoi Cells:** Each cell is filled with a distinct color using polygon rendering. Shared edges are emphasized with bold lines.
- **Sites and Vertices:** Sites are represented with dots, and Voronoi vertices with small markers.
- **Highlighting and Animation:** During step-by-step visualization, the current active region is highlighted with color or animation. Transition effects visually demonstrate updates to the diagram.

5.4 Responsiveness and Accessibility

The interface is designed to adapt to various screen sizes, with a resizable canvas and scalable UI components. Keyboard shortcuts and tooltips will be provided for quick access and improved usability.

6 Technologies To Be Used

Note: The programming languages and libraries to be used was provided in the **Proposal Report**. However, since the project is still towards the beginning,

implementation-wise, I considered an alternative tool.

6.1 Alternative Tool: Implementing with Unity and C#

Although the current implementation of the project such as the static **Voronoi diagram in figure 1** and **Delaunay Triangulation in figure 2** is done in **Python**. And also the complete project is planned using **Python** and computational geometry libraries such as **SciPy** and **Qhull**, an alternative approach could involve using **Unity with C#**. This would enhance interactivity and provide a more visually engaging experience through **gamification** techniques.

6.2 Benefits of a Unity-Based Approach

Using Unity as the framework for this project would offer several advantages:

- **Improved Interactivity:** Unity provides a robust real-time rendering engine, allowing for smooth animations and dynamic updates when modifying the Voronoi diagram.
- **Gamification Elements:** By adding small interactive features, such as real-time animations for new site insertions and smooth transitions between Voronoi cell updates, the project could be more engaging.
- **Better UI/UX Experience:** Unity's built-in UI tools allow intuitive interaction, including draggable points, zooming, and interactive overlays.
- **Performance Optimization:** Unity's engine is optimized for rendering complex graphical structures, which could provide better performance when handling large numbers of points in the Voronoi diagram.

6.3 Comparison of Technologies to be Used

In either case, the fundamental algorithms and theoretical analysis remain the same. This ensures that the project objectives and findings are unchanged. The high level implementations on the algorithms is not specific to a programming language or a library, thus what is discussed so far still applies.

References

[1] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

[2] L. J. Guibas, D. E. Knuth, and M. Sharir, "Randomized incremental construction of delaunay and voronoi diagrams," *Algorithmica*, vol. 7, no. 1-6, pp. 381–413, 1992.

[3] D. M. Mount, "Lecture 10 - the doubly-connected edge list (dcel)," Lecture Notes, University of Maryland, 2020. [Online]. Available: https://www.cs.umd.edu/class/fall2020/cmsc754/Lects/lect10-dcel.pdf

[4] S. Fortune, "A sweepline algorithm for voronoi diagrams," *Algorithmica*, vol. 2, no. 1, pp. 153–174, 1987.

[5] J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," in *Proceedings of the Twelfth Annual Symposium on Computational Geometry*. ACM, 1996, pp. 141–150.

[6] H. Edelsbrunner and E. P. Mücke, "Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms," *ACM Transactions on Graphics (TOG)*, vol. 9, no. 1, pp. 66–104, 1990.

[7] J. Heunis, "Fortune's algorithm: Implementation details," Blog Post, 2018. [Online]. Available: https://jacquesheunis.com/post/fortunes-algorithm-implementation/

[8] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," ACM Transactions on Mathematical Software, pp. 469–483, 1996. [Online]. Available: http://www.qhull.org