Efficient and Rapid Grid Voronoi Diagram Computation

Mingeon Jeong, Minseop Lee, Dongwon Lee, Chaewoon Lee

November 29, 2023

Abstract

The aim of this study is to discretely construct Voronoi diagrams by determining the closest seed to grid points, departing from the traditional method that constructs Voronoi diagrams using line segments and intersections. To achieve this, six distinct algorithms were devised leveraging the grid's properties. Specifically, approaches involved adapting Dijkstra's algorithm to visualize the grid as a graph, recursive techniques based on divide and conquer, optimizing by limiting seed regions using binary search and considering only enclosed seeds, and proposing a method for constructing grid Voronoi diagrams by extending conventional Voronoi diagram construction. Additionally, empirical time measurements were taken using randomly generated test data for each method. Linear regression was applied to determine the constants in the theoretical time complexity equations concerning the number of seeds and grid size. Results indicate that our proposed six algorithms exhibit better performance compared to the naive algorithm, as evident in execution times, theoretical time complexities, and predicted time calculation formulas. Particularly, the time complexity was reduced from the traditional $O(NM^2)$ to as low as $O(NlogN + NM + M^2)$ and $O(M^2log^2M)$.

Keywords: Grid Voronoi diagram, algorithm optimization, Dijkstra's algorithm, divide and conquer, quad tree, sweeping.

Contents

1 Introduction

The Voronoi diagram is a fundamental geometric structure that has found diverse applications across fields ranging from computational geometry and computer graphics to geographic information systems and optimization [1]. It serves as a powerful tool for spatial analysis, offering insights into proximity relationships, resource allocation, and nearest neighbor queries. As the scale and complexity of datasets continue to grow, the demand for efficient algorithms to compute Voronoi diagrams becomes increasingly crucial [2].

This study addresses the challenge of computing Voronoi diagrams in the context of grid-based datasets. Grid Voronoi diagrams are particularly relevant in scenarios where spatial data is organized into a regular grid, such as image processing, terrain modeling, and geographic analysis [3]. However, as grid resolutions increase and the number of grid points escalates, traditional methods for Voronoi diagram computation exhibit limitations in terms of runtime efficiency and memory consumption.

The primary objective of this study is to propose an innovative approach for the efficient and rapid computation of grid Voronoi diagrams. By leveraging recent advancements in algorithm design and parallel processing techniques, we aim to develop a solution that not only scales gracefully with the size of the grid but also maintains a high level of accuracy and robustness. This research builds upon prior work in the field of computational geometry and contributes to the ongoing pursuit of practical solutions for real-world applications.

In this paper, we present the methodology, implementation details, and experimental evaluation of our proposed algorithm. We compare its performance against existing methods using a random test datasets, showcasing its advantages in terms of computation time, memory utilization, and scalability. The results obtained highlight the significant contributions of our approach in addressing the challenges posed by the computation of grid Voronoi diagrams.

The remainder of the paper is structured as follows: Section 2 provides an overview of related work in Voronoi diagram computation and existing techniques for grid-based datasets. Section 3 introduces the problem formulation and outlines the key components of our proposed algorithm. Section 4 elaborates on the technical implementation details, highlighting the novel strategies employed to

achieve computational efficiency. Section 5 presents the experimental evaluation, offering quantitative analysis of our algorithm's performance. Finally, Sections 6–8 discuss the implications of our findings, outlines future research directions, and concludes the study. References and Appendices can be found in the last section of this paper.

2 Related Work

2.1 Fortune's Algorithm

The Fortune algorithm is an algorithm that leverages the properties of parabolas. A parabola is a collection of points where the distances from the focus and the directrix are equal. Therefore, the intersections of two parabolas with different foci (P and Q) and the same directrix lie on the perpendicular bisector of the line segment joining P and Q. This property is used to find the Voronoi Diagram in $O(NlogN)$ time.

- *sweep line*: A straight line denoted as $x = x_0$ used to perform the sweeping. Throughout the following steps, when the sweep line is at $x = x_0$, only consider points with x-coordinates less than or equal to x_0.
- *beach line*: It represents the rightmost boundary of a set of parabolas, which are part of multiple parabolas. Since the directrix is fixed at $x = x_0$, you can determine the beach line by storing the foci as a list, which is referred to as *Focus*.
- *site event*: An event where a point P_i is added. When this event is executed, point P_i is added to the *Focus*. This event is triggered when a point with $x = x_0$ exists.
- *circle event*: An event where one parabola in the *beach line* is blocked by another parabola and disappears. When this event is executed, the point representing the obscured parabola is removed from the Focus. This event is included in the *Focus*, and it occurs when the $x = x_0$ is to the right of the rightmost point of the circumcircle of the three adjacent points with their y-coordinates. The center of this circumcircle is referred to as the *circle point*.

Specific algorithms are as follows:

1. Sort the points in order that x coordinate are ascending. After sorting, let the ith point be denoted as P_i.
2. Create a (`Doubly Linked List`) `Focus` connected on both sides. This linked list contains the points in increasing order of their y-coordinates.
3. For the *sweep line* at $x = x_0$, increase x_0:
   - When a *site event* occurs: Insert point P_i into `Focus`. In this process, a binary search is used based on the y-coordinate to find the appropriate position. Additionally, during this process, the intersection between the newly formed parabola and the existing *beach line* is connected to form a line segment. And this line segment is added to the segments forming the Voronoi Diagram.
   - When a *circle event* occurs: If a *circle event* happens at points P_{i-1}, P_i, P_{i+1}, the point P_i is removed from the `Focus`. Moreover, thereafter, as two parabolas intersect, the intersection points and the *circle event* are connected to add two line segments (one of which will overlap with the existing segment).
4. The line segments obtained by repeating the above process until $x_0 = \infty$ form the Voronoi Diagram.

2.2 Bowyer–Watson Algorithm

The Bowyer–Watson algorithm is a method that solves the Delaunay triangulation, which is the dual relationship of the Voronoi Diagram, in a time complexity of $O(N^2)$. However, if a Quad Tree is used, it can be implemented with a time complexity of $O(N \log N)$. The implementation of this algorithm follows these steps:

1. Define the largest equilateral triangle *Supra-Triangle* that encompasses all the points.
2. Take the P_i and iterate through the points in a predetermined order.
   - For the first point: Connect it with the aforementioned largest equilateral triangle *Supra-Triangle*.
   - For points other than the first one: Remove triangles T_i made by the circumcircles of already drawn triangles that include P_i. Form new triangles by connecting the vertices of those triangles with P_i.
3. Repeat the above process for N points, and upon removal of the largest equilateral triangle *Supra-Triangle*, the Delaunay triangulation is created.

2.3 Jump Flooding Algorithm

Jump Flooding Algorithm is an algorithm used to obtain a Voronoi diagram or perform Distance transforms using a grid of M by M [4]. The time complexity is $O(M^2 \log M)$ and the implementation of this algorithm is as follows:

1. Marking cells that include sites. We call these points to *seed*. Flooding is begin from these *seed*.
2. Start with $M/2$ and reduce the step size k by half, and mark the color according to the below steps.

   for each marked cell $P = (x, y)$, we consider ALL $Q = (x + i, y + j)$ where $i, j \in \{-k, 0, k\}$
   - if Q is not marked, mark with the site of P
   - if Q is marked, mark with more closer site seed between *seed* of Q and *seed* of P
3. Repeating the above task $O(\log M)$ times results in a Voronoi diagram.

An important aspect of this algorithm is that its time complexity is not directly tied to N. In essence, when N significantly surpasses M, this algorithm proves to be quite advantageous. Nevertheless, if a single cell contains two or more sites, this algorithm becomes unsuitable for deriving the Voronoi diagram. In such cases, the approach requires an increase in M to accommodate these complexities.

2.4 Using divide and conquer

Using the divide and conquer algorithm, it is possible to construct the Voronoi Diagram in $O(n \log n)$.

1. Divide: Divide the points such that each side has half of the points based on their x-coordinates.
2. Base Case: When there are three or fewer points, compute the Voronoi Diagram directly.
3. Merge: Connect the boundary points into a *monotone chain* C. Let's designate the points belonging to that chain in order as C_1, C_2, C_3, $\cdots$, C_k. Than the perpendicular bisectors between C_i and C_{i+1} constitute the Voronoi Diagram of the boundary.

The following describes the process of creating *monotone chain*:

1. Select the point with the highest y-coordinate in each divided diagram. Designate these points as P_1 and P_2.
2. Draw a line perpendicular to the line passing through P_1 and P_2, finding the intersection with the boundaries of the cells containing P_1 and P_2.
3. Choose the intersection with the highest y-coordinate among those found in the previous step and connect a segment up to that point.
4. Determine the two points creating the perpendicular bisector, which is made by the highest y-coordinate intersection. As one of these two points will be P_1 and P_2, select the remaining two points (excluding the overlapping point) and restart the process from step 2.

When creating the *monotone chain* each vertex is checked at most once, and each edge is checked at most twice, resulting in a time complexity of $O(N)$. Therefore, the overall time complexity of the divide and conquer process becomes $O(N \log N)$.

2.5 Incremental Algorithm

Using the Incremental Algorithm, it is possible to construct the Voronoi Diagram in $O(N^2)$ time. The mechanism of this algorithm is somewhat similar to obtaining the *monotone chain* in the Voronoi Diagram algorithm using divide and conquer. The implementation of the algorithm follows these steps:

1. For each point P_i, repeat that following:
   - For $i = 1$: Consider the cell of that point as the entire area.
   - For $i >= 2$: Let's assume that the cell containing P_i is the cell of P_j. Then, the perpendicular bisector of the line segment P_iP_j forms a line in the Voronoi Diagram. Now, find the intersection of this line with the boundary of the cell containing P_j. Iterate by finding the cell again from these intersections and drawing perpendicular bisectors.
2. When the above process concludes for all N points, the remaining line segments form the Voronoi Diagram.

Due to checking up to N points in each iteration in the above process, the time complexity is $O(N^2)$.

3 Problem Definition and Methodology

3.1 Problem Definition

The well-known $O(N \log N)$ Voronoi diagram algorithm calculates the vertices and the connections of the lines that constitute the diagram. This research deals with finding discrete solutions for Voronoi diagrams using a grid. In general, we aim to capture the smallest rectangle containing the given points and divide it into shapes that are as close to a square as possible. Therefore, we will only consider cases where the entire area is square. To be more precise, it can be described as follows.

- Problem Statement: When given N *seed*s on an M by M grid, write a program to find and output the point on each grid that is closest to one of the N *seed*s. The coordinates of thet M^2 *seed*s are denoted as $(i, j)(1 \leq i \leq M, 1 \leq j \leq M, i$ and j are integers). If there are two or more equally close *seed*s, output the one with the smaller number.

- Input Format: The first line contains N and M. Starting from the second line, the coordinates (x, y) of N *seed*s are given. The x and y coordinates of each point are between 0 and M, and no two *seed*s have the same x or y coordinates. No three *seed*s are collinear, and the coordinates of the *seed*s can be any real number.
- Output Format: Output the numbers of the M^2 points representing the completed Voronoi diagram. The first input *seed* is numbered as 0, and the last input *seed* is numbered as $N-1$.

3.2 Naive Approach (*Naive*)

Naive approach of grid Voronoi Diagram is compute distance between grid points and *seed*s. The time complexity is $O(NM^2)$. Figure 1 illustrates the basic operation process of the *naive* algorithm.

Algorithm 1 Naive Voronoi Diagram Algorithm

```
for x = 1 to M do
    for y = 1 to M do                          ▷ For all grid point; O(M^2)
        d ← ∞                                  ▷ initialization
        ans[x][y] ← 0
        for i = 1 to N do                      ▷ Compute distance between all seed
            if d > dis((x_i, y_i), (x, y)) then
                d ← dis((x_i, y_i), (x, y))
                ans[x][y] ← i
            end if
        end for
    end for
end for
```

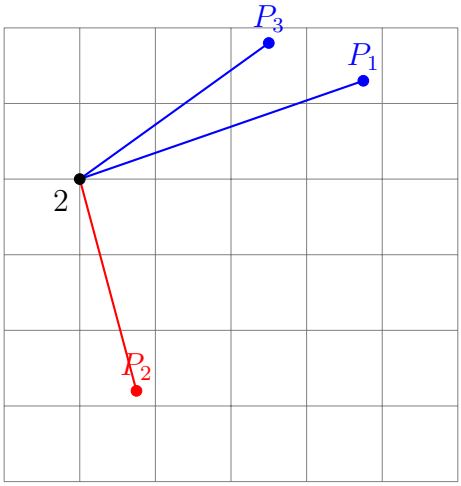


Figure 1: calculate the distance to all *seed*s. and write down the number of the closest *seed*. Since the black dot is closest to the 2nd *seed*, write down 2 next to the black dot.

4 Implementation Details

4.1 Approximate Solution Using Dijkstra's Algorithm (*Dijkstra*)

Represent the given grid as a graph where each grid point is a node, and we connect them with edges in four or eight directions to adjacent grid points. In this graph, we will use the cells containing the *seed* as starting points to apply the idea of Dijkstra's algorithm.

First, we need the following assumption: If we mark the areas that are closest to the same *seed*, they will always be adjacent grid points. Now, the currently closest point found is placed in a heap, and we extract the one with the smallest distance to the nearest *seed*. If the point is k-th closest to the *seed*, we check if it's closer to the k-th *seed* for the adjacent four or eight grid points. If it's closer, we update the closest *seed* and add it back to the heap. If not, we leave that vertex unchanged and move on. Figure 2 illustrates the initial step of this algorithm.

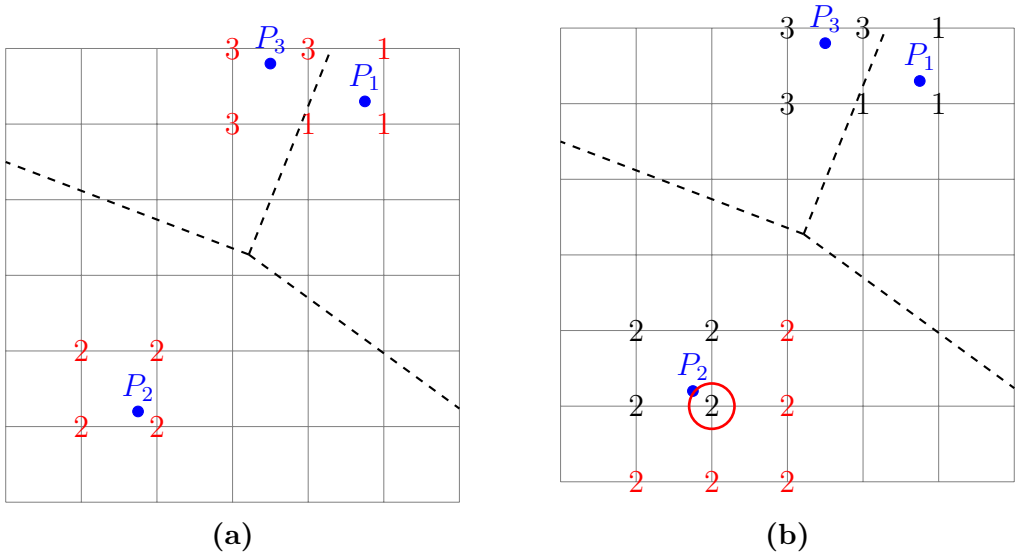


Figure 2: (a) Update the surroundings of the initial P_i first (highlighted in red), while the rest are initialized to -1. (b) From the state in (a), update the surrounding 8 grid points closest to the *seed* among the grid points in the priority queue and insert the updated grid points back into the priority queue.

By repeating this process, it is possible to solve the problem in $O(M^2 \log M)$ under the given assumption. The reason is that the maximum number of times a grid point is added to the priority queue is $O(M^2)$. Additionally, the number of updates also influences the time complexity. When N is smaller than M^2, typically, the entire grid is divided into approximately $\sqrt{N}$ squares. As a result, the total length of the boundaries of the Voronoi diagram averages at $O(N\sqrt{N})$. Hence, with update occurrences amounting to $O(N\sqrt{N})$, an additional $O(N\sqrt{N})$ term is added to the time complexity. Conversely, when N surpasses M^2, on average, there exists one grid per grid cell. Consequently, updates occur $O(N)$ times, thereby introducing an $O(N)$ term to the time complexity.

However, in reality, areas close to the same *seed* are not always adjacent grid points, so it is possible to increase the accuracy by saving information about the 2nd or 3rd closest *seed*s and comparing the adjacent grid points with respect to those *seed*s. Alternatively, you can increase the number of adjacent grid points to be compared. However, as the accuracy increases, the constant factors also increase, leading to a potential doubling of the time required. The following is a pseudocode for implementation. In the time measurements, we employed code that considers the balance between time and accuracy by including consideration for the second-closest *seed*.

Algorithm 2 Grid Voronoi Diagram Using Dijkstra's Algorithm

```
constant T that the number of maximum values to be stored
the array ans that stores T closest seeds
Initialize ans array with −1 values
Create a priority queue pq
function GET_DIS(x, y, i)
    return square distance between (x, y) and i-th seed
```

```
end function
function UPDATE(x, y, i)
    if ans[x][y][0] equal to i then
        return −1                                   ▷ Return -1 if the i-th seed is already closest
    end if
    for t ← 0 to T − 1 do
        if i is closer than ans[x][y][t] from (x, y) then
            for s ← T − 1 down to t + 1 do
                ans[x][y][s] ← ans[x][y][s − 1]     ▷ Shift elements in the array
            end for
            ans[x][y][t] ← i                        ▷ Update the closest seed
            return t
        end if
    end for
    return −1                                       ▷ Return -1 if the point is not closer than existing points
end function
for i ← 1 to n do
    for four vertices (x, y) of the grid that include the i-th seeds do
        if x or y is out of bounds then
            continue
        end if
        r ← update(x, y, i)
        if r ≥ 0 then
            Push x, y, and updated distance into pq
        end if
    end for
end for
while pq is not empty do
    Extract i and j from top of pq
    Extract d from top of pq
    Initialize t that t-th closest distance is equal to d
    if t = T then
        continue
    end if
    for 8 adjust points do
        Calculate coordinate of adjust points (x, y)
        if x or y is out of bounds then
            continue
        end if
        r = update(x, y, ans[i][j][t])
        if r ≥ 0 then
            Push x, y, and updated distance into pq
        end if
    end for
end while
```

4.2 Fast Algorithm in Sparse Grid (*Sparse*)

In this algorithm, the Voronoi Diagram on a grid signifies spatial proximity by associating each cell with the index of the nearest *seed* from a provided list of coordinates. To ensure efficient assignments

for close-by cells, we incorporated multi-resolution strategies.

Our algorithm operates under the premise that the grid area proximate to any given *seed* from the coordinate list typically exhibits a contiguous pattern. To efficiently populate a contiguous region, we devised a strategy where, if the index of the *seed* nearest to the vertices of a defined rectangle is consistent, the entire rectangular region is filled. Drawing from this understanding, after initializing the grid, distances to each point are computed in sequence to populate the grid. Such filling is feasible due to the intrinsic convex polygonal attributes of the Voronoi Diagram.

Central to our algorithm is the 'fill_grid' function. For any specified grid section, the function ascertains the *seed* closest to every vertex of that section. If a single *seed* is closest to all vertices, that grid segment is filled with the index corresponding to that *seed*. Otherwise, the function recursively subdivides the region and reapplies the process. Figure 3 illustrates this process.

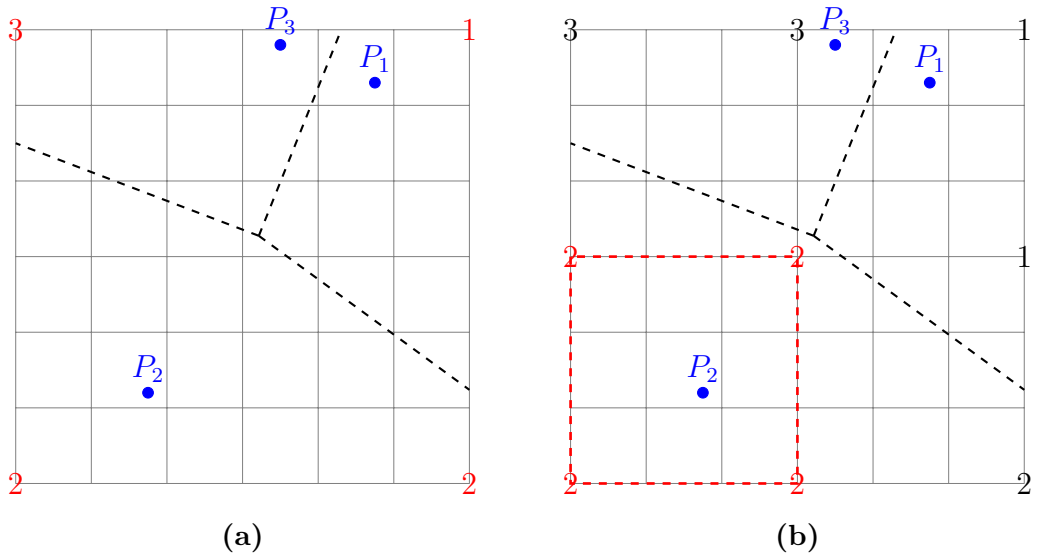


Figure 3: (a) Initially, it checks if the four vertices representing the entire area are the same as the closest *seed*. As they are not all the same, the area is divided into four and the function is recursively called. (b) Among the divisions in (a), for the rectangle in the bottom left, the four vertices are all closest to *seed* 2, so the interior is filled entirely with 2. This square no longer needs further division.

The *Sparse* Algorithm becomes more effective when N is smaller, as it increases the instances where the four corners of the rectangle align with the closest seed. Particularly, when N surpasses M^2, it continuously subdivides until it reaches a grid of size 1, resulting in a function call count of $O(M^2)$. When N is less than M^2, the discussion about time complexity becomes less straightforward due to variations in the Voronoi diagram's shape influencing the function's call count.

Algorithm 3 Sparse Algorithm

```
function SOLVE(points, n, m)
    grid ← INITIALIZE_GRID((0, 0), (m, m))
    grid_info ← {(1, 1), (m, m)}
    FILL_GRID(grid, grid_info, points)
    return grid
end function
function INITIALIZE_GRID(top_left, bottom_right)
    Define grid based on dimensions of top_left and bottom_right
    return grid
end function
function FILL_GRID(grid, grid_info, points)
```

Find coordinate of each corner using $grid\_info$
Find closest points for each corner
if grid section is same as 1x1 or smaller than 2x2 **then**
    Assign all point in grid section with each closest point
else if all corner points have same closest point **then**
    Fill entire grid section with closest point
else
    Divide grid into 4 sections
    Recursively call FILL_GRID
end if
return $grid$
end function
SOLVE($points, n, m$)

4.3 Fast Algorithm in Dense Grid (*Dense*)

For each grid point (i, j), we create two arrays: `arr1`, which stores the cumulative count of *seed* points within a rectangle defined by $(0, 0)$ and (i, j) as its opposite corners, and `arr2`, which records the locations of *seed* points in the grid space.

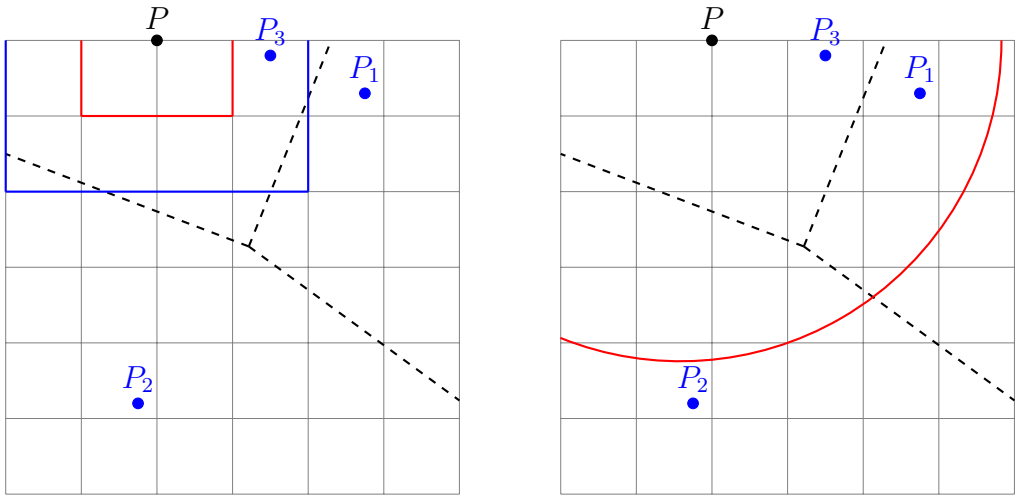
Utilizing the cumulative sum array `arr1`, we employ binary search to determine the minimum value of k for each grid point (i, j) where there exist *seed* points within a rectangular region with both x and y coordinates differing by at most k. `arr1[i][j]` stores the number of *seed* contained in the square between $(i-1, j-1)$ and (i, j), therefore, the farthest possible point is at $(i-k-1, j-k-1)$, and the maximum difference in distance is $d = \sqrt{2}(k+1)$.

Now, since there exist points within a rectangular region with both x and y coordinates differing by at most k, it is sufficient to examine only those *seed* points whose x and y coordinates differ by at most $d = \sqrt{2}(k+1)$. In this case, by utilizing an array `arr2` that stores the positions of grid points where each *seed* point resides, we can create a list of points where both the x and y coordinate differences are at most d. Applying a naive algorithm to this list of points allows us to determine the nearest *seed* point. This process enables the construction of a Voronoi diagram. Figure 4 illustrates this process.

In this approach, the problem can be solved with a time complexity of $O(M^4/N + M^2 \log M)$. The process of calculating this time complexity unfolds as follows. First, the process of determining k involves binary search, which has a time complexity of $O(\log M)$. Subsequently, when finding a list of *seed*s within a distance d from the grid points, and if we denote the number of *seed*s in this list as p the time complexity of determining the nearest seed point is $O(d^2 + p)$. In this context, we can prove $\bar{d}^2 = O(M^2/N)$ and $\bar{p} = O(1)$, and the time complexity for M^2 grid points is $O(M^2) \times O(\log M + M^2/N + 1)$, resulting in $O(M^4/N + M^2 \log M)$. Consequently, when disregarding constants, this approach exhibits a speed that is roughly comparable to the naive algorithm at the scale of $N = M$, and it offers faster performance in scenarios where the *seed* point density is high for $N > M$.

Algorithm 4 algorithm for dense grid

array ans and initialize to 0.
Sort the points array in ascending order based on the x-coordinate.
array cnt that stores the number of points belonging to the grid with (i, j) being the bottom right corner.
array vec that stores the indexes of points belonging to the grid with (i, j) being the bottom right corner.
array sum that stores the sum of the array cnt from $(0, 0)$ to (i, j).
function BINARY_SEARCH($array, start, end, x, y$)



(a) **(b)**

Figure 4: (a) Demonstrates the process of finding $k = 2$ using binary search. When the maximum difference is 2, the point is included, but it's not included when the maximum difference is 1, thus $k = 2$. (b) The range of possible solutions based on the found k, with a radius of $3\sqrt{2}$. The 2nd *seed* cannot be the closest *seed*, so it doesn't need to be considered.

```
    while start < end do
        mid ← ⌊(start + end)/2⌋
        if no dots in the area [x − mid, x + mid] × [y − mid, y + mid] then
            start ← mid + 1
        else
            end ← mid
        end if
    end while
    return start
end function
for x = 1 to M do
    for y = 1 to M do
        d ← ⌈√2 · binary_search(sum, 1, m − 1, x, y) + √2⌉
        array point_list that stores the indexes of points that should be checked
        for z = x − d to x + d do
            for w = y − d to y + d do
                if cnt[z][w] > 0 then
                    for all i in vec[z][w] do
                        add i to point_list
                    end for
                end if
            end for
        end for
        assign the point to ans using naive approach
    end for
end for
```

4.4 Add Optimization to Dense Algorithm (*Quad*)

In the *Dense* algorithm, the part that contributes significantly to the time complexity involves checking all the *seed*s in a possible d^2 regions. If the number of *seed*s is small, resulting in low density, not all d^2 regions may encompass many *seed*s. However, because it's necessary to verify the actual existence of *seed*s within these regions, it involves d^2 operations.

Utilizing the concept of a quad tree can expedite this time complexity to $O(\log^2 M)$. By storing the numbers of *seed*s encompassed in the entire region at the root of the tree and recursively diving into four quadrants to implement a Quad tree, storing the numbers of *seed*s contained in each quadrant, it's possible to partition an arbitrary region of square $d \times d$ into $O(\log^2 M)$ regions. Therefore, the problem can be solved for each grid point in $O(\log^2 M + p)$ time. Consequently, the overall time complexity becomes $O(M^2 \log^2 M)$. Figure 5 illustrates this idea.

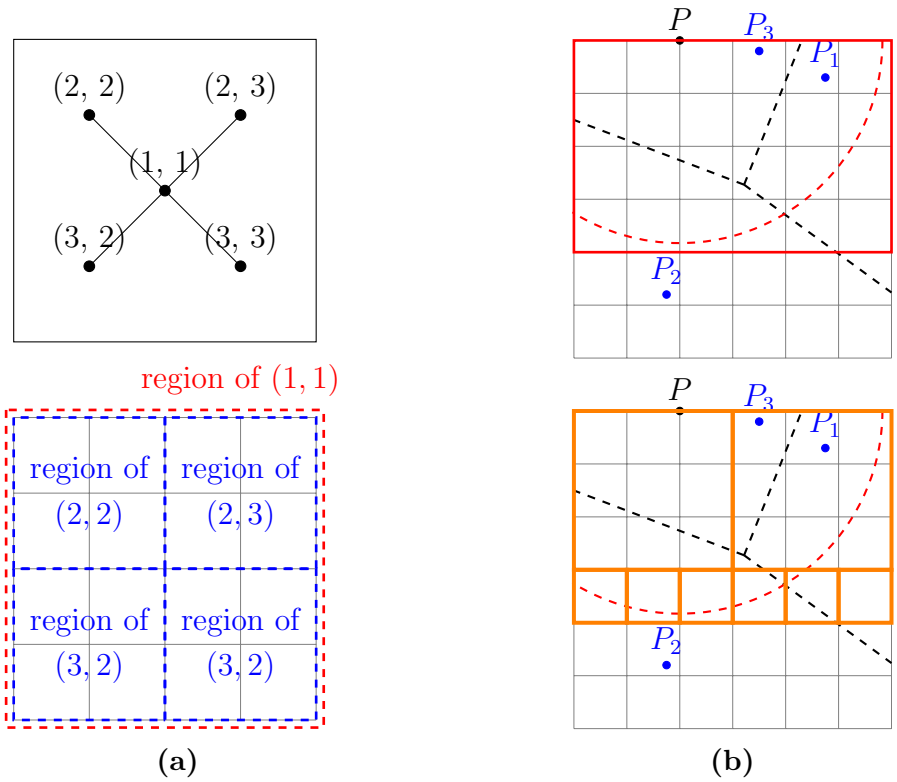


Figure 5: (a) Illustrates the process of creating the quad tree when $M = 4$. The node corresponding to $(1, 1)$ represents the entire area, while $(2, 2), (2, 3), (3, 2), (3, 3)$ represent each quadrant. Recursively, $(4, 4)$ would correspond to a single cell at the top-left. (b) Deriving the possible area of a solution from Figure 4(b) using quad tree. When converting the circular area into a rectangle, it forms a red rectangle. Dividing this rectangle according to the regions of the quad tree results in subdivisions depicted by orange rectangles.

Algorithm 5 quad tree

```
1: array tree is quad tree that store indexes of seed
2: procedure INIT(i, j, s, e, l, r)
3:     if s = e and l = r then
4:         make leaf node
5:         tree[i][j] ← vec[i][j]
6:         return
7:     end if
8:     Divide grid into 4 sections
```

```
    Recursively call INIT
    add all elements of each quadrants to tree[i][j]
end procedure

procedure GET(i, j, s, e, l, r, st, ed, lt, rt)
    [s, e] × [l, r] is a region of tree[i][j]
    [st, ed] × [lt, rt] is a region of query
    if region of tree[i][j] and region of query is not overlap then
        return
    end if
    if tree[i][j] is empty then
        return
    end if
    if region of tree[i][j] is included in region of query then
        add all elements of tree[i][j] to point_list
        return
    end if
    other cases, recurisvely call GET
end procedure
```

4.5 Use Delaunay Triangulation and BFS (*BFS*)

To explain the BFS algorithm, the sweeping algorithm which will explain at 4.6, we first need to generate Delaunay triangles. First, we utilize Delaunay triangulation to obtain the lines representing the Voronoi division for the given *seed* points. Consider a set of points forming these triangles. For each triangle, find its circumcenter. Connecting these circumcenters, we can construct a Voronoi diagram, which enables us to locate the edges within the diagram. These edges are defined by equations, which can be stored.

By equations of edges, we assign closest seed for each adjacent to the Voronoi lines. Subsequently, we assign a value of -1 to the grid points adjacent to the Voronoi lines. Then, initiate a breadth-first search (BFS) starting from the grid point adjacent to the line, adding adjacent points in the up, down, left, and right directions to the queue. The process stops upon all grid point get a value. During this process, grid points in the queue are assigned the number of the closest *seed* point, thereby completing the Voronoi diagram in the grid. Figure 6 illustrate this process.

This method allows us to solve the problem with a time complexity of $O(NlogN + NM + M^2)$. The breakdown of the time complexity is as follows: Delaunay triangulation incurs a time complexity of $NlogN$. The subsequent process of assigning -1 involves the number of lines $O(N)$ and the number of grid points adjacent to each line $O(M)$, resulting in a time complexity of $O(NM)$. Lastly, in the BFS process, since each grid point is examined at most once, a time complexity of $O(M^2)$ occurs.

Algorithm 6 BFS Algorithm

```
Obtaining Voronoi Lines Using Delaunay Triangulation
array mat that stores -1 when each point is an adjacent point of Voronoi lines.
function BFS(mat)
    In the graph, nodes correspond to grid points.
    visited = {}
    queue = list()
    The start_point corresponds to the grid point closest to each of the n seed points.
    add start_point to queue
    visited[start_point] = True
```

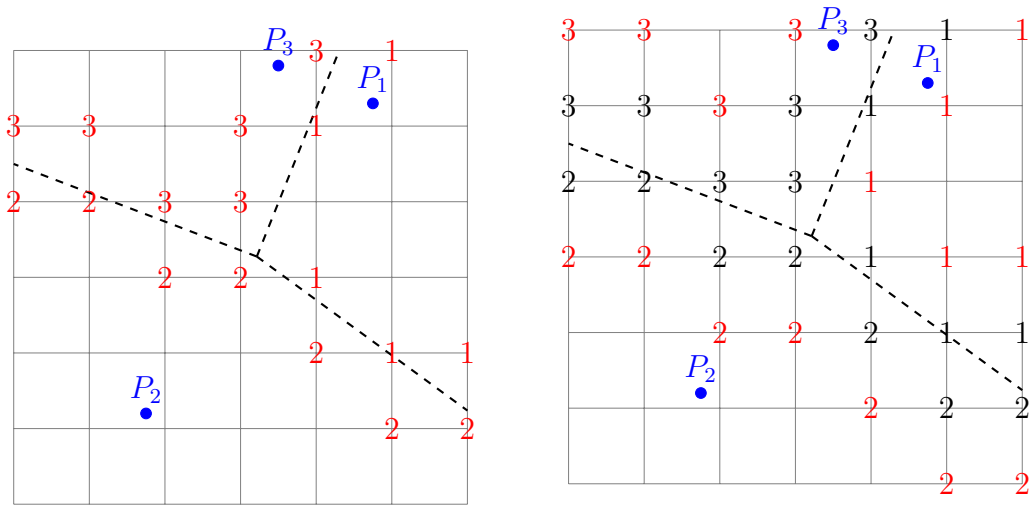


Figure 6: (a) Find the closest *seed* for grid points adjacent to the segments forming the Voronoi diagram and insert them. (b) The grid state after the completion of updates by all initially entered elements in (a).

```
    while not is_empty(queue) do
        current_point = queue.pop(0)
        if mat[current_point.x][current_point.y] ≠ -1 then
            add adjacent nodes to queue
        end if
        ans[node.x][node.y] ← the adjacent seed points of current_point
    end while
end function
```

4.6 Use Delaunay Triangulation and Sweeping (*Sweeping*)

To explain the sweeping algorithm, we introduce intersections by equations of edges of Voronoi Diagram. When searching for intersections in these equations, the method involves finding intersections in grid along line parallel to the x-axis. By locating these intersections, we can traverse along the x-axis, aiming to identify areas enclosed by line segments. This traversal leads us to discover the segments where intersections occur. We can then create a segment array, which stores the situations from an intersection to the end of a traversal. This involves saving the intersections and their corresponding area indices within the array. Accessing the segment array with these indices provides the intersections, which when filled along the x-axis, effectively color the area. Figure 7 illustrate above process.

The time complexity of the sweeping algorithm is determined by two processes: generating equations and coloring areas based on these equations. The complexity of generating equations is $O(NM)$, where each line segment is stored in the segment with its intersection, which is of the order of $O(M)$. The process of coloring areas involves traversing all line segments parallel to the x-axis, thus encompassing all points once, leading to a complexity of $O(M^2)$. Additionally, the initial segment creation using Delaunay triangulation adds the time complexity of Delaunay's method, which is $O(NlogN)$. Therefore, the overall time complexity of the sweeping algorithm is $O(NlogN + NM + M^2)$.

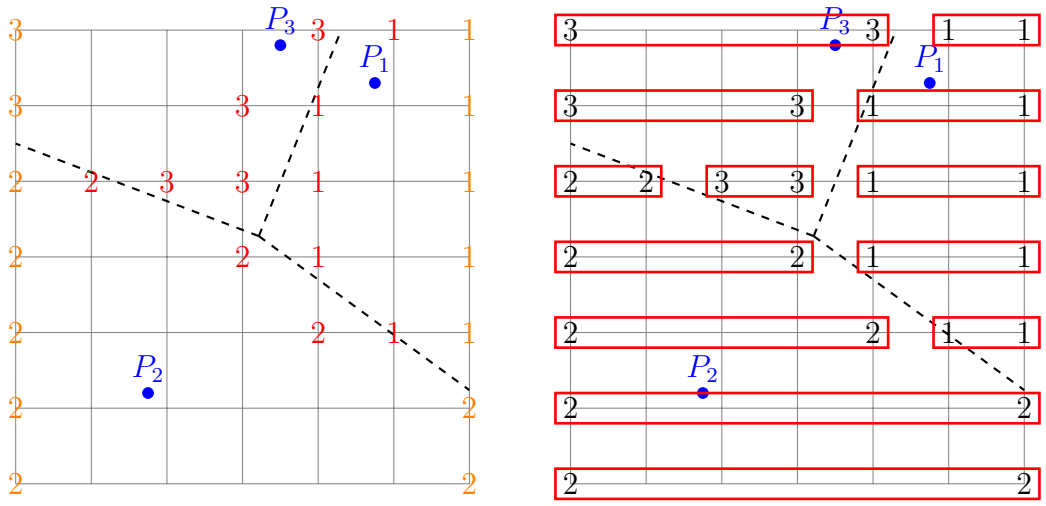


Figure 7: (a). For each horizontal line, identify the endpoints, the closest seed, and the closest seed around the boundary point. The former is represented in orange, while the latter is indicated in red. (b). Prior to encountering lines that constitute the Voronoi diagram for each horizontal line as shown in (a), areas enclosed by red rectangles, where the closest seed remains unchanged, are filled with the same value.

5 Experimental Evaluation

The time measurements were conducted using Polygon (`https://polygon.codeforces.com/`), enabling random data generation and the measurement of execution time and memory usage for each source code. The server specifications include an Intel(R) Core(TM) i3-8100 CPU @ 3.60GHz. Moreover, in the forthcoming sections, the seven types of algorithms will be briefly referred to as *Naive, Dijkstra, Sparse, Dense, Quad, BFS, Sweeping*. These names are also indicated in the titles of each subsection within the 'Implementation Details'.

5.1 Overview

The following table presents the execution times of seven algorithm types for several (n, m) pairs. If the execution time exceeded 15 seconds, it's denoted as TLE (Time Limit Exceeded).

(n, m)	$(100, 1000)$	$(1000, 1000)$	$(10000, 1000)$	$(100000, 1000)$
Naive	279	1,707	TLE	TLE
Dijkstra	584	798	1,119	2,136
Sparse	144	1,481	TLE	TLE
Dence	TLE	3,001	1,157	1,728
Quad	593	732	960	1,782
BFS	161	202	349	1,731
Sweeping	149	176	330	1,783

Algorithm 7 Sweeping Algorithm

```
m, n ← dimensions of the grid
Initialize 3D array segment[m + 1][n] with values [−2, −2]
procedure ProcessLineSegments(list_line)
    for each line segment in list_line do
        x_range ← RangeManipulation(linesegment)
        for each x in x_range do
            Compute y-coordinates where the line segment intersects
            Store these intersections in segment array
        end for
    end for
end procedure
procedure CreateBoard
    Initialize 2D array board[m + 1][n] with values −2
    for each x coordinate do
        for each segment in segment[x] do
            Identify range of y-coordinates (low and high)
            Update board accordingly
        end for
    end for
end procedure
```

(n, m)	$(3, 4000)$	$(500, 4000)$	$(1500, 4000)$	$(10000, 4000)$	$(100000, 4000)$	$(300000, 4000)$
Naive	1,468	14,864	TLE	TLE	TLE	TLE
Dijkstra	7,175	14,827	TLE	TLE	TLE	TLE
Sparse	1,333	3,904	12,972	TLE	TLE	TLE
Dence	TLE	TLE	TLE	TLE	TLE	TLE
Quad	5,389	8,777	9,908	10,870	13,263	TLE
BFS	2,592	3,366	3,466	3,751	5,173	8,397
Sweeping	2,246	2,482	2,576	2,865	4,310	7,551

5.2 Computation Time Comparison

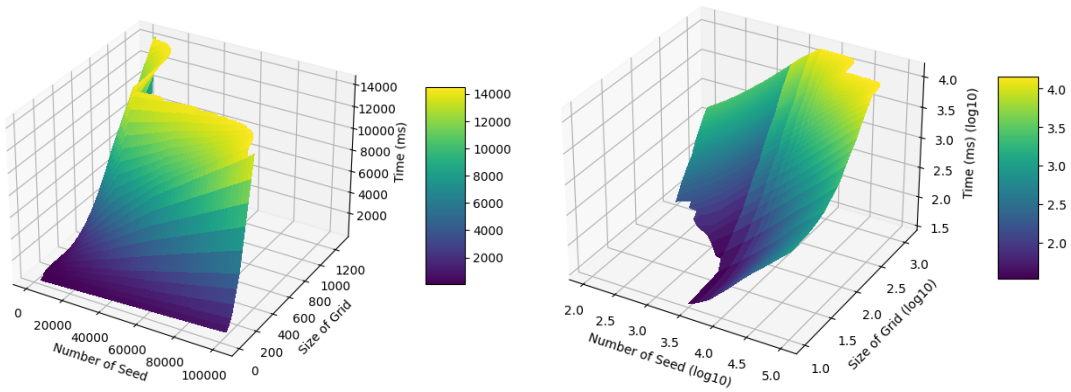
We measured the time for a total of 88 (N, M) pairs, ranging from 100 to 102,400 for N (increasing in powers of two) and from 10 to 1,280 for the Size of Grid M (also increasing in powers of two), repeated 10 times each. Figure 8 illustrates the time variation concerning (N, M) for the seven algorithm types in a 3D graph. Each graph depicts linear scale on the left and logarithmic scale on the right.

5.3 Computation Time by Size of Grid

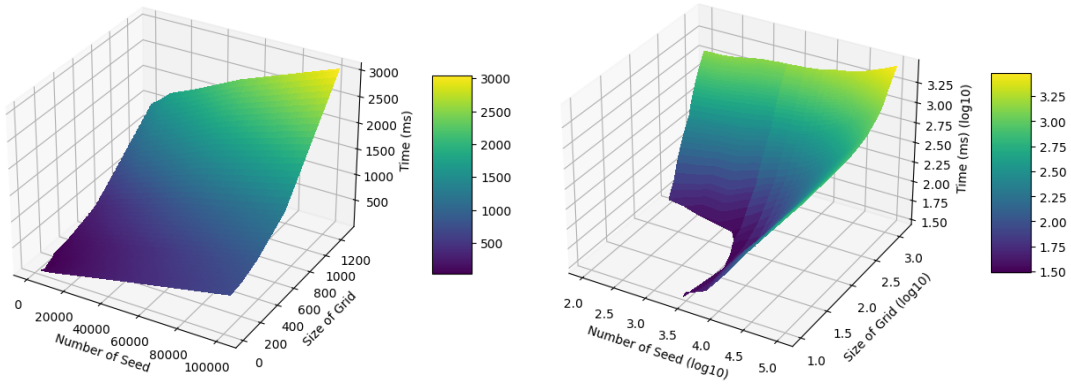
We measured the time for a total of 27 (N, M) pairs, maintaining the Number of seed N at 10,000 while varying the Size of Grid M from 10 to $1.26 \approx 10^{1/3}$ times increments, reaching up to 4,070, resulting in 27 different values. Each (N, M) pair was repeated 10 times for measurement. Figure 9 illustrates the time variation concerning M for the seven algorithm types in 2D graphs. Each graph depicts linear scale on the left and logarithmic scale on the right.

5.4 Computation Time by Number of Seed

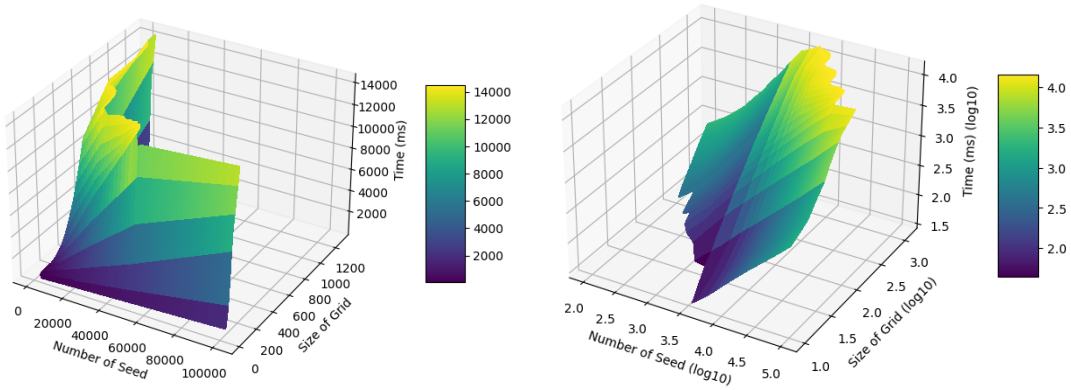
The Size of Grid M was held constant at 1000, while the Number of seed N ranged from 100 to $1.26(H10^{1/3})$ times its previous value, reaching a total of 31 different values up to 102592. Measure-



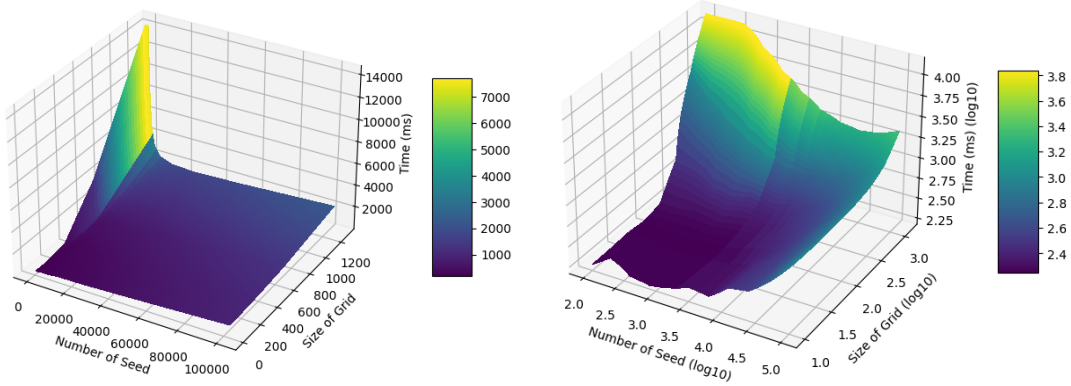
(a) *naive*



(b) *dijkstra*

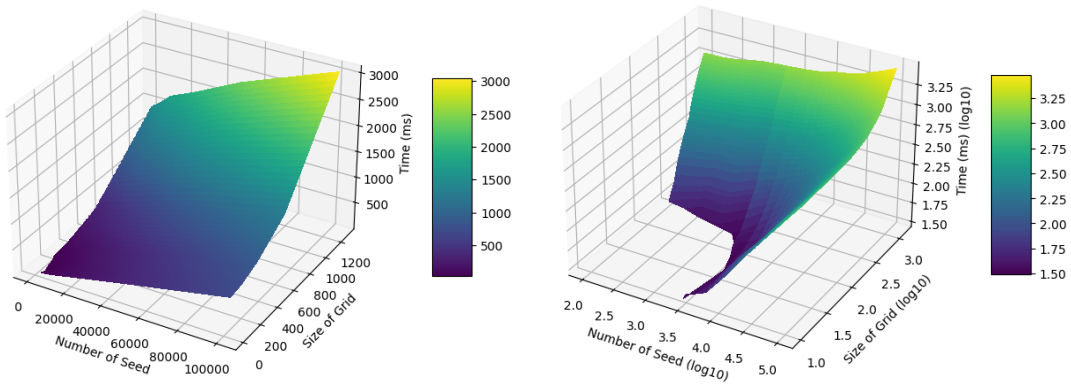


(c) *sparse*

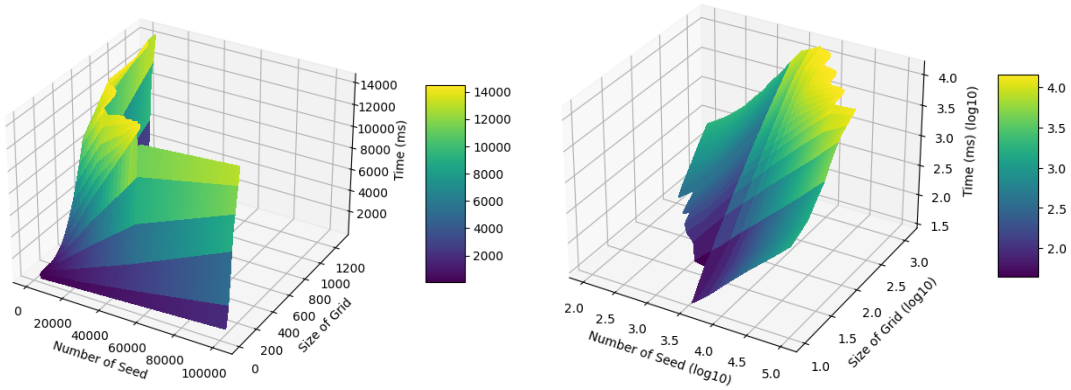


(d) *dense*

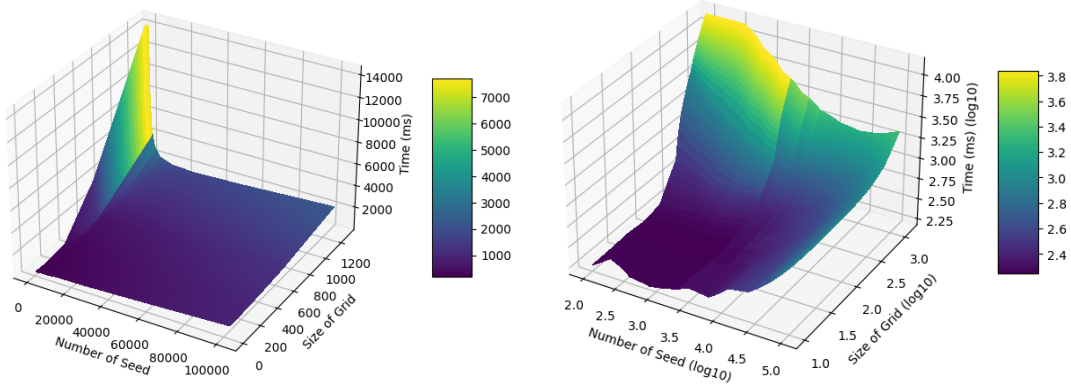
The following page will continue...



(e) *dijkstra*

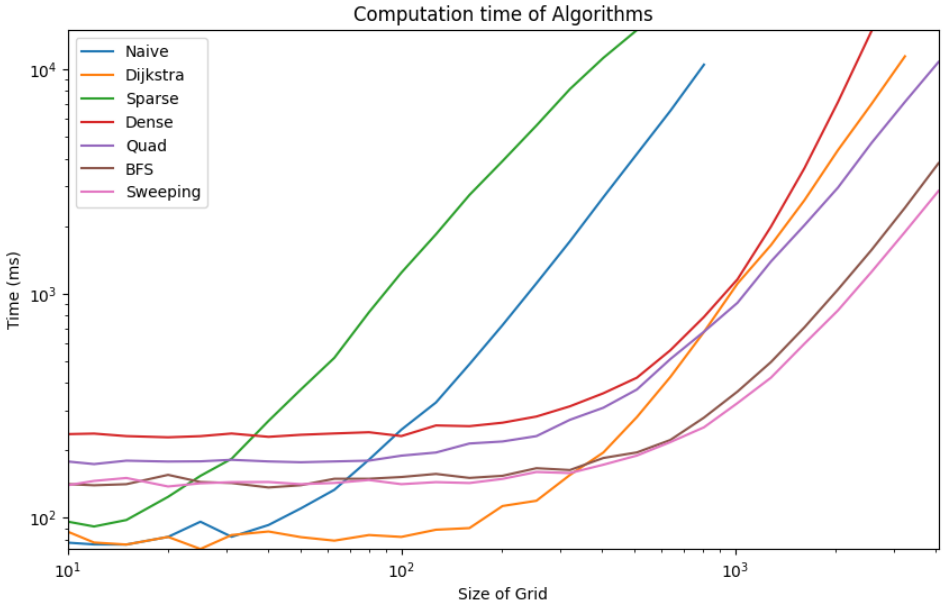


(f) *sparse*

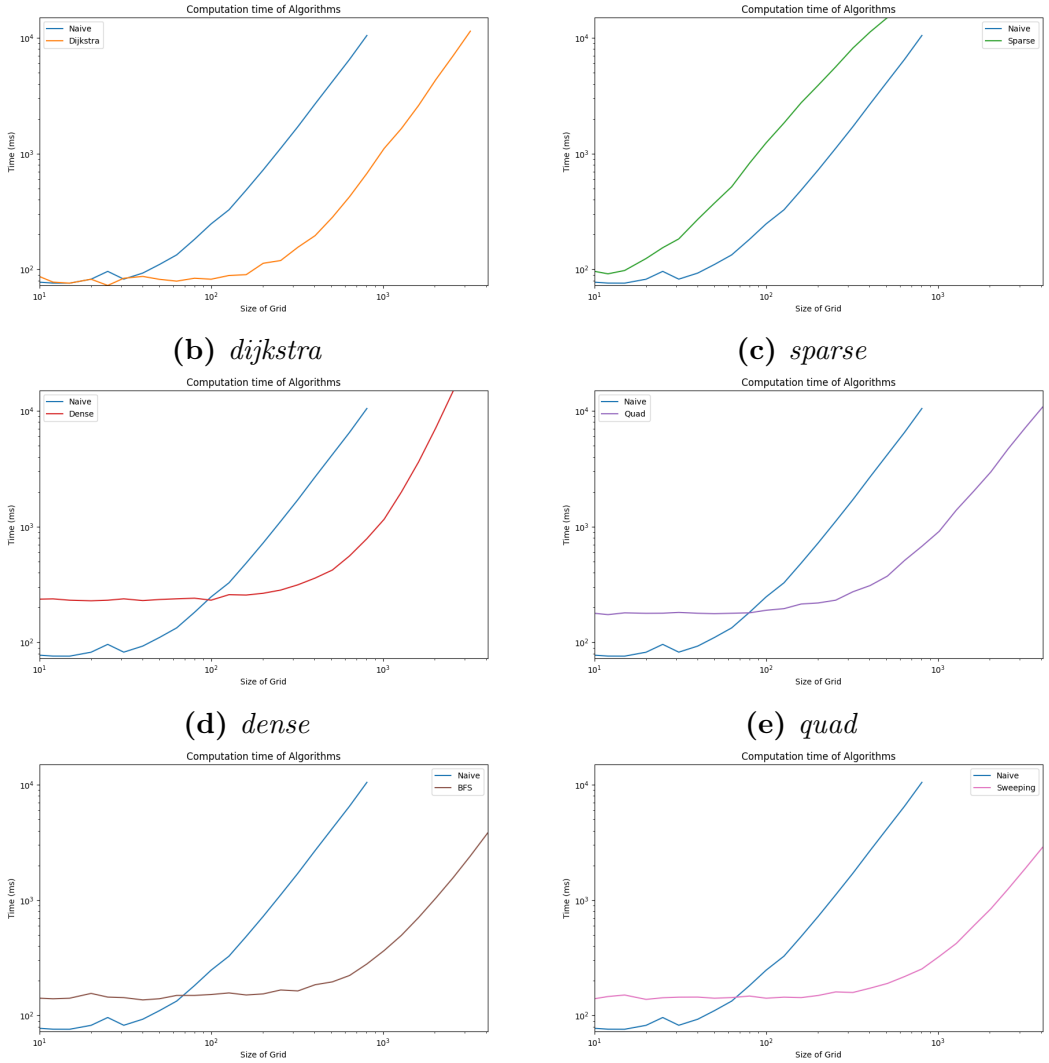


(g) *dense*

Figure 8: Execution time of 7 algorithms concerning Number of seed N and Size of grid M. The left side displays a linear scale graph, while the right side presents a logarithmic scale graph. (a) through (g) represent the execution times for each of the *naive*, *dijkstra*, *sparse*, *dense*, *quad*, *BFS*, and *sweeping* algorithms.



(a) all



(b) *dijkstra* **(c)** *sparse*

(d) *dense* **(e)** *quad*

(f) *BFS* **(g)** *sweeping*

Figure 9: Execution time for 7 algorithms with $N = 10000$ across Size of Grid M. (a) represents the overall graph, while (b) to (g) display the execution times of 6 algorithms, namely *naive*, *dijkstra*, *sparse*, *dense*, *quad*, *BFS*, and *sweeping*, individually, in relation to the Size of Grid M.

ments were taken by repeating the process 10 times for each of the 31 (N, M) combinations. Figure 10 represents the execution time concerning N for each of the 7 algorithms, displayed as a logarithmic scale 2D graph.

5.5 Constant Estimation

Based on the execution times and the theoretically predicted time complexity equations for different (N, M) pairs, I derived the constants in the time complexity equations. I used the 'sklearn' module in Python to perform linear regression during this process. The time is measured in milliseconds (ms). "Not Found" indicates that the theoretical time complexity equation couldn't be identified.

- *Dijkstra*

$$T = 79.69 + 1.975 \times 10^{-5} \cdot N\sqrt{N} + 5.369 \times 10^{-4} \cdot M^2 \log M \quad R^2 = 0.9795 \quad (N < M^2)$$

$$T = 6.25 + 6.932 \times 10^{-3} \cdot N + 3.483 \times 10^{-4} \cdot M^2 \log M \quad R^2 = 0.9994 \quad (N > M^2)$$

- *Sparse*

$$\text{Not Found} \quad (N < M^2)$$

$$T = 183.74 + 1.332 \times 10^{-5} \cdot NM^2 \quad R^2 = 0.9885 \quad (N > M^2)$$

- *Dense*

$$T = 318.73 + 1.873 \times 10^{-6} \cdot \frac{M^4}{N} + 1.213 \times 10^{-4} \cdot M^2 \log M \quad R^2 = 0.9848$$

- *Quad*

$$T = 108.49 + 8.343 \times 10^{-3} \cdot N + 1.331 \times 10^{-5} \cdot M^2 \log^2 M \quad R^2 = 0.9703$$

- *BFS*

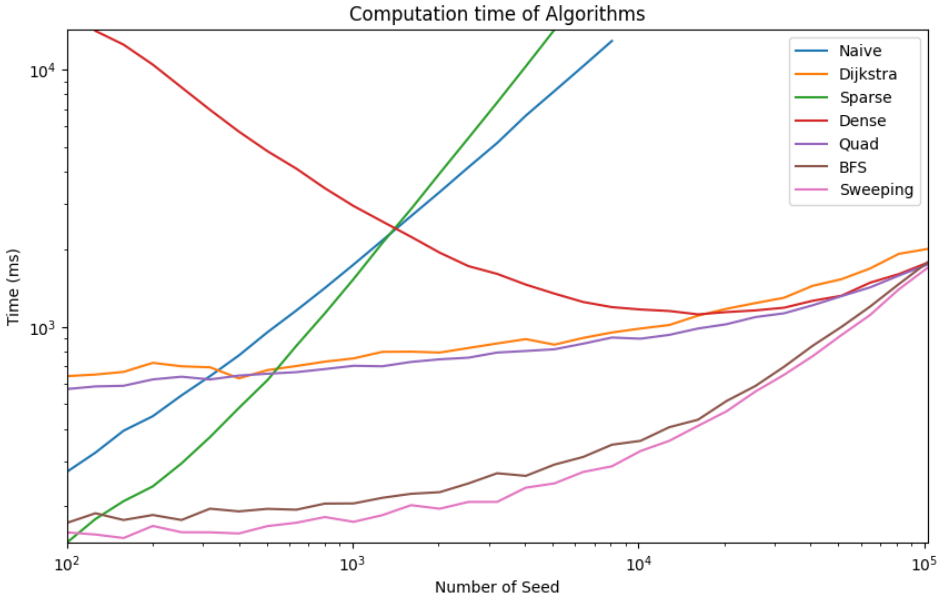
$$T = 20.82 + 1.311 \times 10^{-3} \cdot N \log N + 3.907 \times 10^{-7} \cdot NM + 1.961 \times 10^{-4} \cdot M^2 \quad R^2 = 0.9985$$

- *Sweeping*

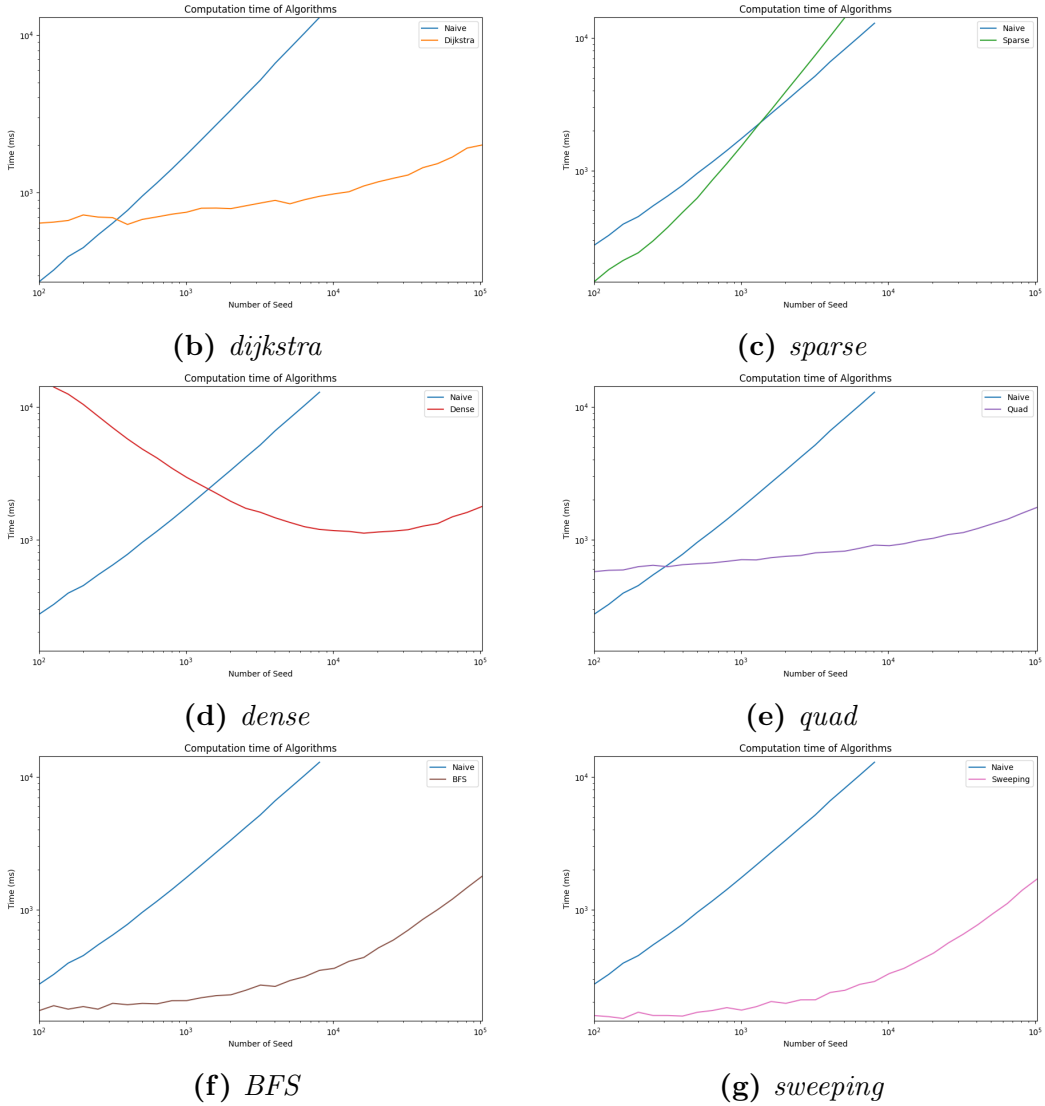
$$T = 26.63 + 1.294 \times 10^{-3} \cdot N \log N + 5.912 \times 10^{-7} \cdot NM + 1.578 \times 10^{-4} \cdot M^2 \quad R^2 = 0.9982$$

6 Discussion

This section discusses the theoretical aspects of each algorithm and analyzes the characteristics of each based on Figure 9, Figure 10, and their theoretical time complexities. Prior to the discussion, the consistent values shown in Figure 9 for small M can be attributed to time dependent on N, given the logarithmic scale of the graph, resulting in a slope related to the degree of M. Similarly, in Figure 10, the consistent values for small N relate to time dependent on M, with the slope associated with the degree of N. Now, let's delve into the theoretical discussion of each algorithm. Due to the similarities between BFS and sweeping, they will be discussed together, while the others will be addressed separately.



(a) all



(b) *dijkstra* **(c)** *sparse*

(d) *dense* **(e)** *quad*

(f) *BFS* **(g)** *sweeping*

Figure 10: Execution Time of 7 Algorithms concerning Number of seed N with $M = 1000$. (a) represents the overall graph, while (b) to (g) display the execution times of 6 algorithms, namely *naive*, *dijkstra*, *sparse*, *dense*, *quad*, *BFS*, and *sweeping*, individually, in relation to the Number of seed N.

6.1 *Dijkstra*

The basic idea of this algorithm starts from a *seed* and expands one cell at a time to compute the Voronoi diagram. While BFS could be used to implement this, it lacks accuracy because the Voronoi diagram may not have regions adjacent to a single *seed*. To address this, an extension of BFS using Dijkstra's algorithm was employed.

Dijkstra's algorithm determines the shortest path from a starting point to each point in a graph with weighted edges based on a recurrence relation. To ensure the validity of this recurrence relation, it expands from the closest point outward, utilizing a priority queue to swiftly find the nearest point.

While BFS is fixed, Dijkstra's algorithm can be applied to various similar forms of recurrence relations, allowing it to compute modified shortest paths like the second shortest path. In this paper, the *dijkstra* algorithm adopts a method to approximate as closely as possible when regions aren't directly attached to a single *seed*. By storing the second closest *seed* and including it in the updating process, when considering only the nearest *seed*, among 70 randomly chosen data points at $m = 1000$, only 48 provided accurate results. However, considering the second closest *seed* resulted in 69 accurate answers out of 40 randomly chosen data points at $m = 1000$, and considering the third closest *seed* resulted in accurate answers for all data points. Moreover, it showed a tendency to produce more errors as N increased.

However, considering more *seed* points increases processing time. If considering T *seed*s, the updates in *dijkstra* algorithm are T times more, and since T times the duration is needed for each update, it becomes T^2 times overall. Considering this, $T = 2$ seems the most appropriate constant concerning both time and accuracy.

When $T = 2$, in Figure 9, the preprocessing time complexity in terms of N appears significantly smaller, almost at the level of the naive algorithm. However, due to the large constant attached to M, the speed diminishes compared to other algorithms as M increases. Also, in Figure 10, it shows a significant impact from the constant term, leading to a significant increase in time concerning M. Moreover, as the slope concerning N is minimal, it is largely unaffected by N, especially useful when M is small and N is large.

6.2 *Sparse*

When calculating the Voronoi diagram, a time-consuming aspect is having to compute the distance from each grid point to all *seed*s, even when there's a high probability that adjacent grid points belong to the same region due to imprecise boundary knowledge. Utilizing the convex polygonal property of the Voronoi diagram can resolve this. If all four vertices of a rectangle belong to the same region, then all points within the rectangle belong to that region. By dividing large rectangles into smaller ones when regions are extensive, we can reduce wasted time.

However, this method is effective only when N is greater than M^2. In cases where, on average, each grid contains more than one *seed*, this approach doesn't save time as the regions become too narrow. Additionally, although the time complexity is similar to the naive algorithm, due to more function calls and having to find the nearest *seed* more times, the constant is larger.

For these reasons, this algorithm performs better when *seed*s are sparsely distributed, leading to the nomenclature *sparse* for this algorithm. In Figure 9, it shows a graph similar to the naive algorithm, but the constant being larger causes the graph of the naive algorithm to shift to the left. Moreover, in Figure 10, the constant term is closest to 0, indicating that the number of function calls significantly impacts N when N is small. That is, although the exact time complexity isn't determined, it's expected that when N is larger than M^2, it will show a higher degree in N and a lower degree in M.

6.3 *Dense*

Contrary to the *sparse* algorithm, in cases with high *seed* densities, significant time can be saved by not considering distant *seed*s. The approach involves initially identifying a range where the nearest point might exist and examining points only within that range.

To implement this approach, the number and indexes of *seed*s included in the grid representing the area between points $i-1, j-1$ and i, j are stored. Then, binary search is utilized to find k for each grid point, representing a square with a side length of $2k$ centered at the grid point that contains a *seed*. This results in an area within a radius of $\sqrt{2}(k+1)$, allowing verification of points using preprocessed data for each grid.

Typically, a few points are expected to be within an area roughly $\sqrt{2}k$ in size. However, if k is large, even without points in each grid, looping through all grids becomes necessary, resulting in a time proportional to k^2. Hence, this method is efficient only when *seed* density is low, whereas for higher densities, this approach isn't effective due to the narrowness of each region. Therefore, this algorithm is named *dense*.

In Figure 9, a graph explosively increases concerning M, indicating that the influence of M starts becoming significant compared to other algorithms when $N = 10^4$. Also, in Figure 10, for N below 10^4, as N decreases, the time increases peculiarly, which aligns with the theoretical properties and results. The observed pattern of passing through a minimum and then increasing again in the graph occurs because p starts to influence N significantly when the density is sufficiently high. However, it should be perceived as an expected outcome, similar to how *dijkstra* or *dijkstra* are affected.

6.4 *Quad*

The most time-consuming part of the *dense* algorithm involves finding all points within the considered range after obtaining the value through binary search. However, this can be resolved by applying the idea of a merge sort tree to a quad tree, storing each quadrant in a quad tree and combining it with the concept of a segment tree representing the sum of areas each node represents in the quad tree and storing arrays in each node instead of a number.

However, applying the merge sort tree concept faced an issue with memory usage. Although each *seed* could be included in a maximum of $O(\log^2 M)$ quad tree nodes, needing to process $O(M^2)$ arrays implies that for memory usage below 1GB, M needs to be less than $2^{12} = 4096$.

By utilizing a quad tree, the k^2 factor in the *dense* algorithm can be effectively reduced to $\log^2 M$. Although it has longer preprocessing times, it effectively computes the Voronoi diagram even in scenarios with low *seed* density, without significant time differences.

Considering Figure 9, it's evident that both the preprocessing time concerning N and the increase in time concerning M are considerably smaller in the *dense* algorithm compared to *quad* algorithm. This is because in *quad* algorithm, when the length of the array stored in a quad tree node is 0, further division doesn't occur. In contrast, the *dense* algorithm necessitates examining all scenarios. Additionally, observing Figure 10, it's notable that initially, when N is less than 10^4, *quad* algorithm's time is significantly faster than the *dense* algorithm, indicating resolution of issues in the *dense* algorithm. As N increases, the difference diminishes, eventually resulting in similar computation times. Also, *quad* algorithm, similar to *dijkstra* algorithm, exhibits minimal time escalation due to an increase in N, making it beneficial when M is small and N is large.

6.5 *BFS* and *Sweeping*

BFS and *sweeping* both utilize results obtained from computing the Voronoi diagram. These results include the boundaries dividing each region, identifying which two *seed*s' perpendicular bisectors they are, and the points corresponding to intersections of these boundaries. As these results can be swiftly computed in $O(N \log N)$, leveraging the Voronoi diagram's outcomes can expedite grid filling.

In the case of *BFS*, initiating BFS from the grid's boundaries ensures that even if regions aren't directly adjacent to a single *seed*, the starting point includes regions near the boundary, thereby avoiding exceptions. Thus, addressing the issues encountered in *BFS* starting from *seed*s leads to a straightforward completion of the grid's Voronoi diagram. The time complexity of this algorithm reaches $O(N \log N + NM + M^2)$, considering the maximum $O(NM)$ for determining boundary start points.

sweeping capitalizes on the notion that the region a point belongs to changes only when a line intersects the Voronoi diagram. Therefore, it computes intersection points for each horizontal line with the lines comprising the diagram and fills the grid's Voronoi diagram by moving in the direction of increasing x-coordinates. Similar to the process in *BFS*, saving points near the boundaries suffices, resulting in identical preprocessing for both methods. Hence, both methods exhibit the same time complexity.

In Figure 9, due to the computation time for the Voronoi diagram, it's evident that the time complexity concerning N surpasses that of *dijkstra* algorithm. However, it's observable that with increasing M, completion time improves rapidly. Additionally, observing Figure 10, the time complexity concerning M is smaller, yet the impact of N outweighs that of *dijkstra* or *quad* algorithms. Thus, it's effective in scenarios with small N and large M. Lastly, *BFS* has a larger constant than *sweeping*. Utilizing *BFS* often involves checking each point more than once, making it slower compared to the precision of a single check in *sweeping*.

7 Conclusion

In this study, we focused on developing an algorithm to quickly compute a lattice-based Voronoi diagram, as opposed to the traditional Voronoi diagram. The aim of the lattice Voronoi diagram is to divide the plane into a grid and find the nearest *seed* point for each grid point. This can be considered a different problem from the traditional Voronoi diagram, which involves finding the Voronoi lines that partition the plane. Previous studies on traditional Voronoi diagrams primarily focused on identifying the lines forming the diagram and their intersections. However, in the case of lattice Voronoi diagrams, the focus of our research was on reducing the number of distance measurements between each grid point and the *seed* points, as it is necessary to assign the closest *seed* point to every grid point. Including the simplest algorithm, the *Naive* algorithm, we designed a total of seven algorithms and compared them by graphing their execution times based on the number of grid points and seed points. Additionally, except for the *Sparse* algorithm, we derived the theoretical time complexity for the algorithms and then performed linear regression with the actual execution time data to determine the precise coefficients for the time complexity equations. The results showed that *BFS* and *Sweeping* generally offered fast performance, but in cases with fewer grid sizes and *seed* points, *Sparse* was faster, and in cases with smaller grid sizes and a larger number of *seed* points, *Dijkstra* and *Quad* were more efficient. Such findings can accelerate problem-solving in spatial analysis, image processing, and terrain modeling. Furthermore, by presenting seven distinct algorithmic approaches, this study introduces ideas that can be utilized as perspectives in solving various mathematical problems in the future.

8 Future Work

8.1 Problem of Linear Regression

In this study, the coefficients of the theoretical time complexity were derived using linear regression based on the execution time according to the number of grids and seed points. For linear regression, the Mean Squared Error (MSE) was used as the Loss Function. This approach minimizes the sum

of the squares of the differences between the actual and predicted values. However, this method has an inherent limitation: while it predicts well for larger actual values, it tends to be less accurate for relatively smaller actual values. Even if a high correlation is observed in the derived time complexity formula, it can be challenging to deem it highly reliable due to this issue. To address this, potential solutions include applying a logarithmic transformation to the execution time before regression to reduce scale or creating and applying a new Loss Function that assigns weights according to the magnitude of the actual values.

8.2 Using Multi-Processing

Currently, all of the algorithms we have developed are algorithms that run on a single processor and have been tested on servers that do not support hyperthreading. However, in recent years, multi-processing technology is very important and widely used in the computing field. Multi-processing refers to processing multiple tasks simultaneously using multiple CPU cores. The problem of this study can be solved more efficiently by using the multi-processing. Recently, the confluence of the conventional streams of development in computer system design has been removed these limiting factors, making possible a new class of multiprocessor systems [5]. There are tasks that can be processed in parallel in all seven algorithms developed, and computing this task with a multi-processor will show better computing speed than performing complex calculations with a single processor. Therefore, as a follow-up study, it will be possible to improve algorithms suitable for multiprocessing and develop new algorithms using the characteristics of multiprocessing.

9 References

[1] https://dl.acm.org/doi/abs/10.1145/116873.116880
[2] Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys (CSUR), 23(3), 345-405.
[3] Lau, B., Sprunk, C., & Burgard, W. (2010, October). Improved updating of Euclidean distance maps and Voronoi diagrams. In 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (pp. 281-286). IEEE.
[4] Proceedings of the 2006 symposium on Interactive 3D graphics and games - SI3D '06. I3D '06. Redwood City, California: Association for Computing Machinery. pp. 109–116.
[5] https://dl.acm.org/doi/pdf/10.1145/327070.327215