Efficient and Rapid Grid Voronoi Diagram Computation

Mingeon Jeong, Minseop Lee, Dongwon Lee, Chaewoon Lee

October 18, 2023

// Abstract //

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. 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 [1].

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 [2]. 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*. **Figure 2.1.1** provides an example of the *beach line*.

- *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*. **Figure 2.1.2** illustrates the circle and *circle point*.

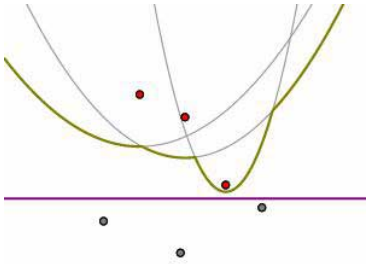


Figure 2.1.1

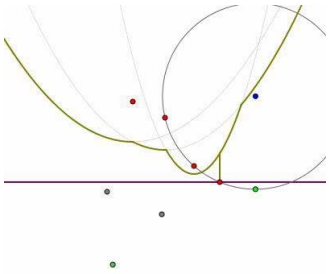


Figure 2.1.2

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:
   - In the case of a *site event*: Insert point P_i into `Focus`. 이 과정에서 알맞은 위치를 찾기 위해 y좌표를 기준으로 이분 탐색(`binary search`)를 사용한다. 또한, 이 과정에서 새롭게 생긴 포물선과 기존의 *beach line*의 교점을 이어 Voronoi Diagram을 이루는 선분에 추가한다.
   - *circle event*가 발생한 경우: 점 P_{i-1}, P_i, P_{i+1}에서 *circle event*가 발생한 경우 `Focus`에서 점 P_i를 제거한다. 또한, 이 이후에는 두 개의 포물선과 교점이 생기므로, 그 교점과 *circle point*를 이어 2개의 선분을 추가한다. (하나는 기존과 겹칠 것이다)
4. $x_0 = \infty$가 될 때까지 위 작업을 반복하여 얻은 선분들은 Voronoi Diagram을 이룬다.

2.2 Bowyer–Watson Algorithm

Bowayer-Watson은 Voronoi Diagram과 쌍대관계인 들로네 삼각분할을 $O(n^2)$의 시간복잡도에 해결하는 방법이다. 만약 `Quad Tree`를 사용할 경우 $O(n \log n)$의 시간복잡도로 구현할 수 있다. 이 알고리즘의 구현 방식은 아래와 같다:

1. 모든 점을 포함하는 가장 큰 정삼각형(*Supra-Triangle*)를 잡는다.
2. i번째 점을 P_i라 두고, 점을 정해진 순서대로 하나씩 돈다.
   - 첫 번째 점의 경우: 위의 가장 큰 정삼각형(*Supra-Triangle*)과 연결한다.

- 첫 번째 점이 아닌 경우: 이미 그린 삼각형들 T_i의 외접원들 중, P_i를 포함하는 외접원을 만드는 삼각형 T_i들을 제거하고, 그 삼각형들의 꼭짓점들과 P_i를 이어 새로운 삼각형을 만든다.

3. N개의 점에 대해 위 작업을 반복하고, 가장 큰 정삼각형(*Supra-Triangle*)을 지우면 돌로네 삼각분할이 만들어진다.

Figure 2.2.1는 위 과정을 그림으로 그린 것이다.

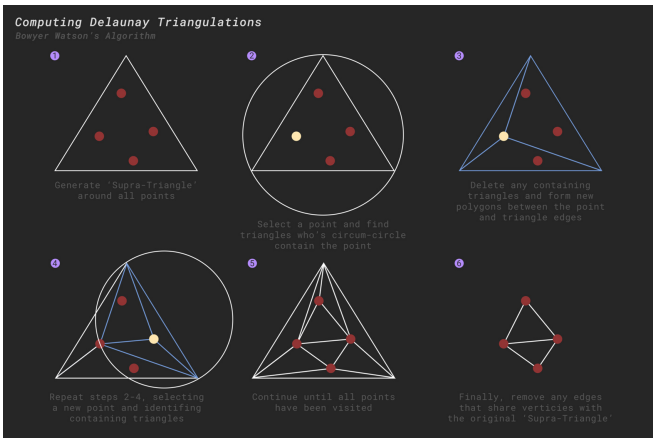


Figure 2.2.1

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 [3]. 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.

Figure 2.3.1 discribe this algorithm.

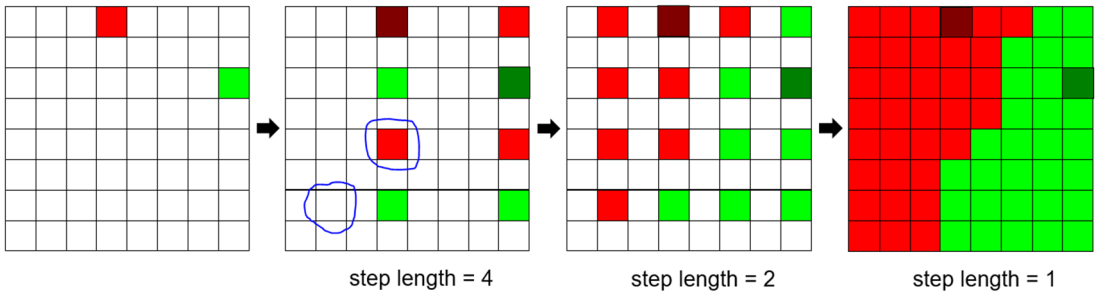


Figure 2.3.1

One of important feature of this algorihtm is that time complexity is not realted to N. In other word, if N is very bigger than M, this algorihtm is useful. However, if two or more cites is included in one cell, this algorihtm can't used to find Voronoi diagram. In that situation, we have to increase M.

2.4 Using divide and conquer

분할 정복 알고리즘을 이용해 Voronoi Diagram을 $O(n \log n)$에 구할 수 있다.

1. 분할: x좌표를 기준으로 양쪽에 각각 절반의 점이 존재하도록 나눈다.
2. 마지막: 점이 3개 이하인 경우 수식적으로 Voronoi Diagram을 구한다.
3. 병합: 경계의 점들을 *monotone chain* C로 잇는다. 그 체인에 속하는 점을 체인 순서대로 C_1, C_2, C_3, $\cdots$, C_k라 두었을 때 C_i와 $C_i + 1$의 수직이등분선이 경계면의 Voronoi Diagram이 된다.

아래는 세 과정을 각각 나타낸 것이다.

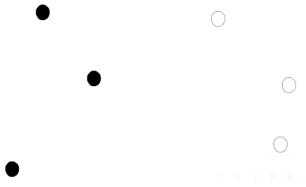


Figure 2.4.1

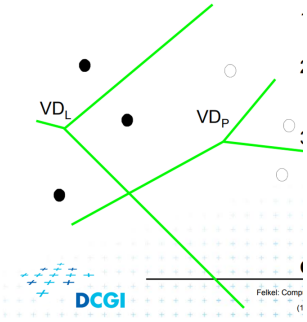


Figure 2.4.2

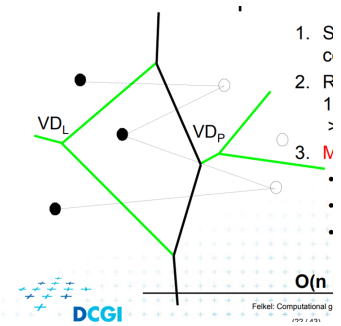


Figure 2.4.3

이제 *monotone chain*을 만드는 과정을 살펴보자.

1. 각각의 분할된 다이어그램에서 가장 y좌표가 큰 한 점을 고른다. 이 점을 P_1, P_2라 두자.
2. P_1과 P_2를 수직이등분선하는 직선을 그은 뒤, P_1, P_2의 셀의 경계와의 교점을 구한다.
3. 위에서 구한 교점들 중 가장 y좌표가 큰 교점을 선택하여, 그 지점까지 선분을 잇는다.
4. 위의 교점을 만든 수직이등분선을 만드는 두 점을 구한다. 즉, 그 두 점의 수직이등분선은 y좌표가 가장 큰 교점을 만든다. 이 두 점 중 하나는 P_1 또는 P_2일 것이므로, 겹치는 점을 제외한 나머지 두 점을 선택한다. 다시 2로 돌아가 과정을 반복한다.

*monotone chain*을 만들 때, 각 정점은 최대 1번 확인하고, 각 간선은 최대 2번 확인하므로 시간복잡도는 $O(n)$이다. 즉, 분할정복 과정의 전체 시간복잡도가 $O(n \log n)$이 된다. **Figure 2.4.4**는 위 과정을 그림으로 그린 것이다.

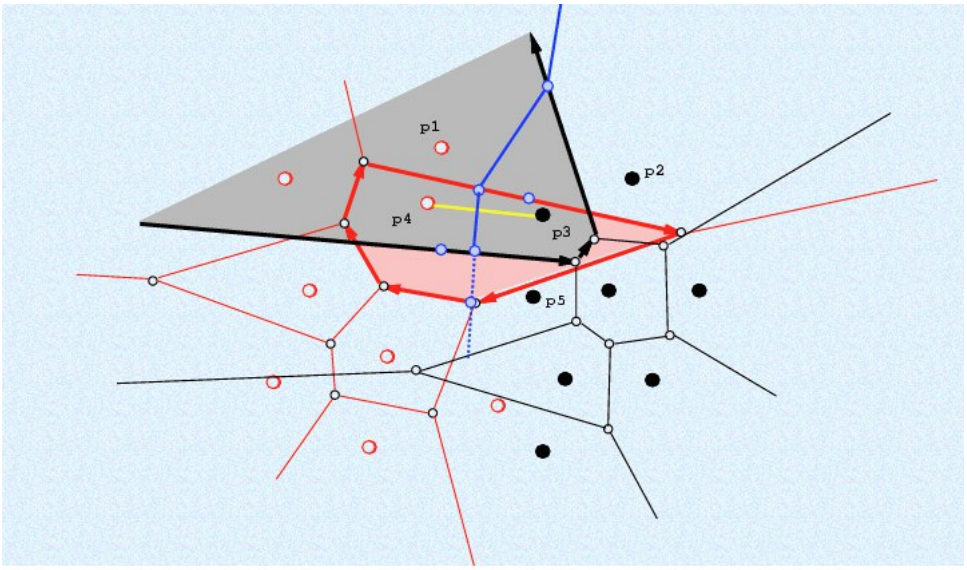


Рис. 4 Аналогично рисунку 3 обновляем один из концов отрезка. Пускаем луч и т.д

Figure 2.4.4

2.5 Incremental Algorithm

Incremental Algorithm을 이용하면 $O(n^2)$의 시간에 Voronoi Diagram을 구할 수 있다. 이 알고리즘의 기작은 분할 정복을 이용한 Voronoi Diagram 알고리즘에서 *monotine chain*을 구하는 것과 비슷하다. 알고리즘의 구현 방식은 아래와 같다:

1. 각 점 P_i에 대하여 다음을 반복한다.
    - $i = 1$: 그 점의 셀을 전체 영역으로 잡는다.
    - $i >= 2$: P_i가 포함된 셀을 P_j의 셀이라고 하자. 그러면 선분 P_iP_j의 수직이등분선은 Voronoi Diagram을 이루는 선이 된다. 이제 이 선과 P_j의 셀의 경계선과의 교점을 구하자. 그 교점에서 다시 셀을 찾고 수직이등분선을 긋는 것을 반복한다.
2. n개의 점에 대해 위 작업이 종료되면 남는 선분들이 Voronoi Diagram이 된다.

위 과정에서 한번에 반복당 최대 n개의 점을 확인하기 때문에 시간복잡도는 $O(n^2)$이다. **Figure 2.5.1**는 위 과정을 그림으로 그린 것이다.

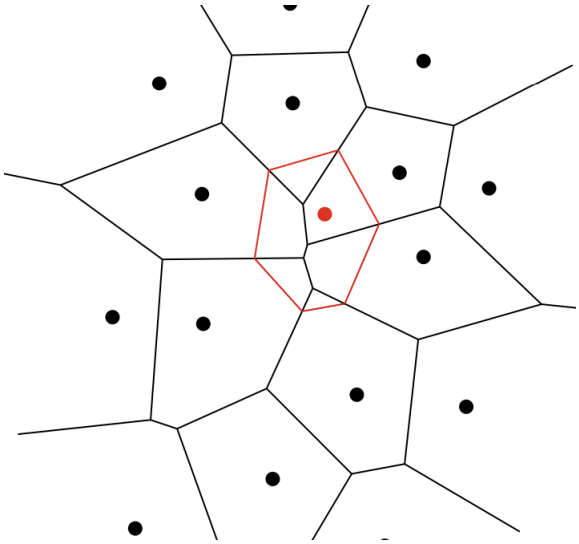


Figure 2.5.1

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 points 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 points. The coordinates of the M^2 points 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 points, 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 points are given. The x and y coordinates of each point are between 0 and M, and no two points have the same x or y coordinates. No three points are collinear, and the coordinates of the points can be any real number.

- Output Format: Output the numbers of the M^2 points representing the completed Voronoi diagram. The first input point is numbered as 0, and the last input point is numbered as $N-1$.

3.2 Naive Approach

Naive approach of grid Voronoi Diagram is compute distance between grid points and cites. The time complexity is $O(NM^2)$.

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 cites
            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
```

4 Implementation Details

4.1 Simple Optimization in Naive Algorithm

In naive code, there is a high probability that the closest seed of (i, j) will be the closest seed of surrounding vertices of (i, j). By taking advantage of this, one can reduce the number of times the minimum distance is updated by first exploring the adjacent four grid points or eight grid points, thus saving time.

4.2 Approximate Solution Using Dijkstra's Algorithm

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.

By repeating this process, it is possible to solve the problem in $O(M^2 \log M)$ under the given assumption. 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 seeds and comparing the adjacent grid points with respect to those seeds. 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.

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 $t \leftarrow 0$ to T **do**
        if i is closer than $ans[x][y][t]$ from (x, y) **then**
            for $s \leftarrow T - 1$ down to $t + 1$ **do**
                $ans[x][y][s] \leftarrow ans[x][y][s - 1]$ ▷ Shift elements in the array
            end for
            $ans[x][y][t] \leftarrow i$ ▷ 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 $i \leftarrow 1$ 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
        Call UPDATE(x, y, i)
        if $r \geq 0$ **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 $r \geq 0$ **then**
            Push $x, y,$ and updated distance into pq
        end if
    end for
end while

4.3 Fast Algorithm in Sparse Grid

In this algorithm, the Voronoi Diagram on a grid signifies spatial proximity by associating each cell with the index of the nearest point 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 point 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 point 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 point closest to every vertex of that section. If a single point is closest to all vertices, that grid segment is filled with the index corresponding to that point. Otherwise, the function recursively subdivides the region and reapplies the process.

Multi-resolution Algorithm shows a time complexity of $\mathrm{O}(N^2\mathrm{M})$, a result derived using the Master Theorem.

Algorithm 3 Multi-resolution Algorithm

function MULTI_RESOLUTION($points, n, m$)
  $grid \leftarrow$ INITIALIZE_GRID($(0,0),(m,m)$)
  $grid\_info \leftarrow \{(1,1),(m,m)\}$
  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 smaller than 2x2 **then**
    Assign each corner with its closest point
  else if all corners point to same closest point **then**
    Fill entire grid section with that point's index
  else
    Divide grid into 4 sections
    Recursively call FILL_GRID
  end if
  return $grid$
end function
MULTI_RESOLUTION($points, n, m$)

4.4 Fast Algorithm in Dense Grid

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.

Now, since there exist points within a distance of d from (i, j), it is sufficient to examine only those seed points whose x and y coordinates differ by at most d. 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.

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 points within a distance d from the grid points, and if we denote the number of points 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($array, start, end, x, y$)
    while $start < end$ **do**
        $mid \leftarrow \lfloor (start + end)/2 \rfloor$
        if no dots in the area $[x - mid, x + mid] \times [y - mid, y + mid]$ **then**
            $start \leftarrow mid + k$
        else
            $end \leftarrow mid$
        end if
    end while
    return $start$
end function
for $x = 1$ to M **do**
    for $y = 1$ to M **do**
        $d \leftarrow \lceil \sqrt{2} \cdot \text{binary\_search}(sum, 1, m - 1, x, y) + 1 \rceil$
        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

4.5 Add Optimization to Dense Algorithm

4.6 Using $O(N \log N)$ Voronoi Diagram Algorithm

5 Experimental Evaluation

5.1 Naive Algorithm

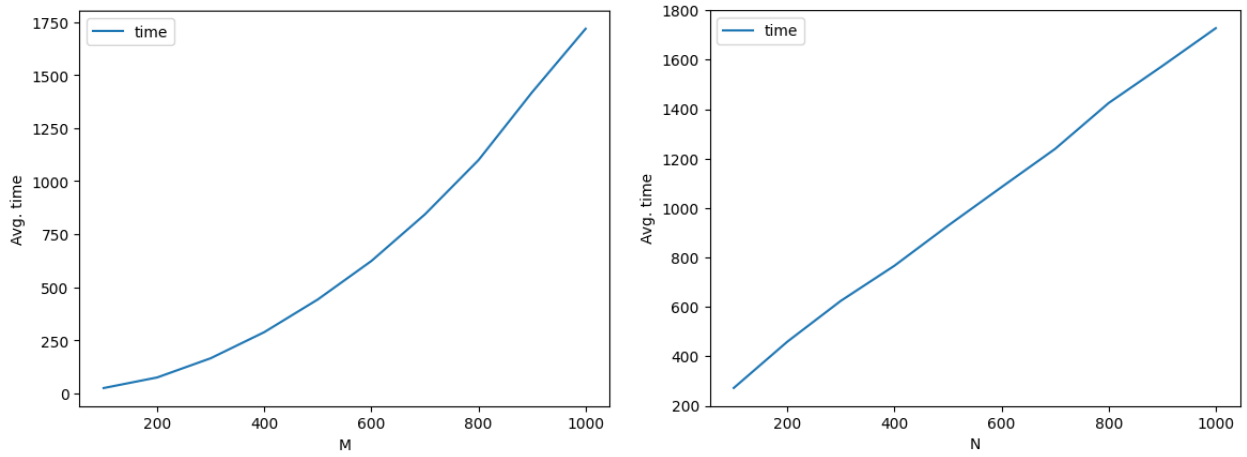


Figure 5.1

As shown in the upper graph, you can confirm that the execution time is proportional to both M^2 and N. Below is the formula estimated using the sklearn module in Python. The time unit is in milliseconds (ms).

$$T = 0.0017171M^2 + 8.5583150 \text{ where } N = 1000$$

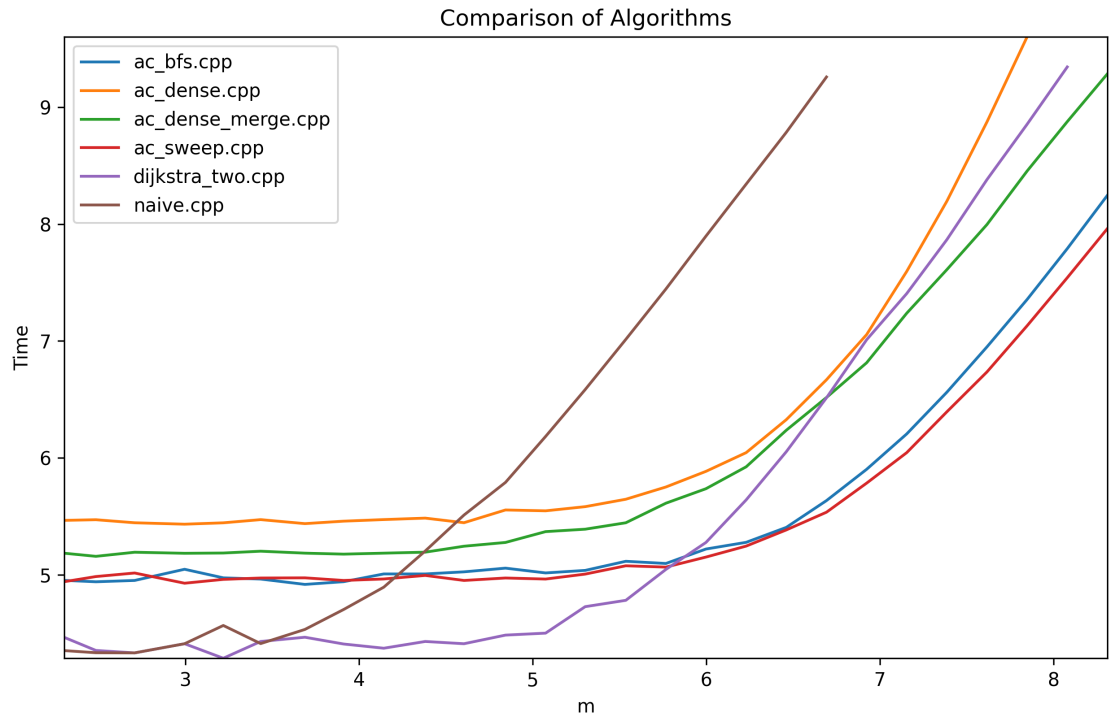
$$T = 1.6047212N + 127.7733333 \text{ where } M = 1000$$

5.2 time comparison

The two tables below present the results of measuring the time in milliseconds (ms) for various values of n with a fixed m. If the running time exceeds 15 seconds, it is indicated as TLE.

(n, m)	(100, 1000)	(1000, 1000)	(10000, 1000)	(100000, 1000)
naive	280	1715	TLE	TLE
dijkstra	280	389	561	1512
sparse	145	1481	TLE	TLE
dence	TLE	2792	1013	1700
Delaunay	155	202	343	1731

(n, m)	(3, 5000)	(500, 5000)	(1500, 5000)	(10000, 5000)	(100000, 5000)	(300000, 5000)
naive	2,199	TLE	TLE	TLE	TLE	TLE
dijkstra	7,175	11,995	12,463	14,897	TLE	TLE
sparse	2,090	5,381	TLE	TLE	TLE	TLE
dence	TLE	TLE	TLE	TLE	TLE	TLE
Delaunay	3,930	5,101	5,225	5,569	6879	10061


Comparison of Algorithms
ac_bfs.cpp
ac_dense.cpp
ac_dense_merge.cpp
ac_sweep.cpp
dijkstra_two.cpp
naive.cpp
Time
m
3
4
5
6
7
8
5
6
7
8
9

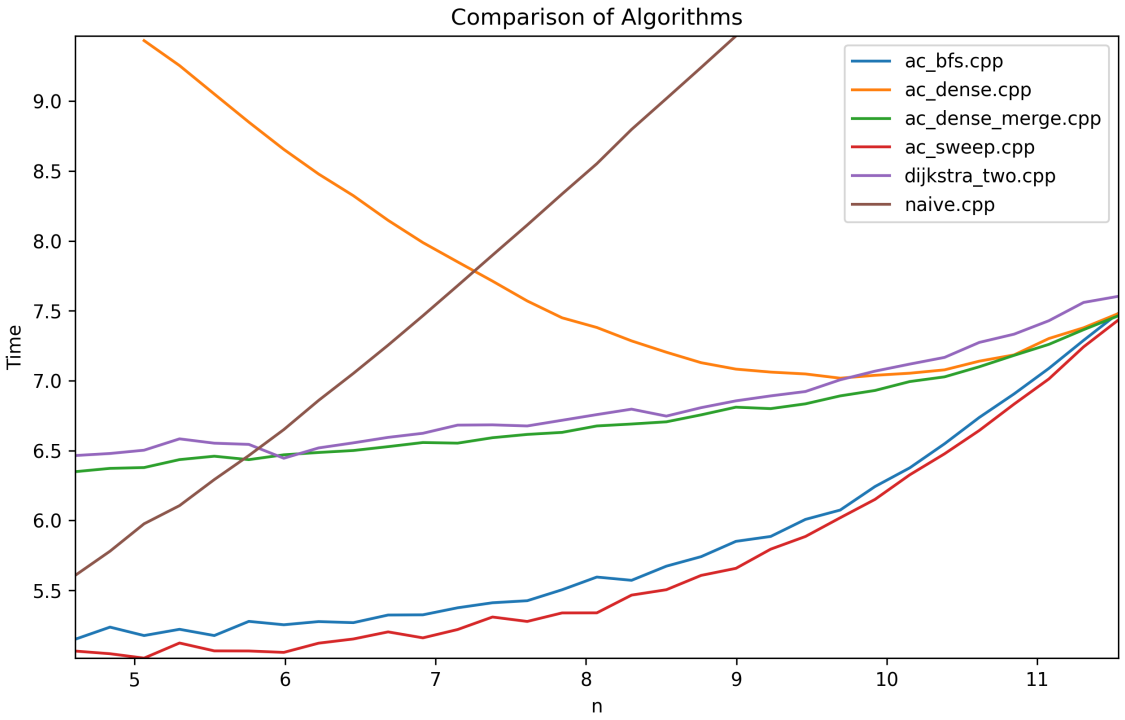

Comparison of Algorithms
ac_bfs.cpp
ac_dense.cpp
ac_dense_merge.cpp
ac_sweep.cpp
dijkstra_two.cpp
naive.cpp
Time
n
5
6
7
8
9
10
11
5.5
6.0
6.5
7.0
7.5
8.0
8.5
9.0

Figure 5.2

6 Discussion

7 Future Work

8 Conclusion

9 Appendices

10 References

[1] Aurenhammer, F. (1991). Voronoi diagrams—a survey of a fundamental geometric data structure. ACM Computing Surveys (CSUR), 23(3), 345-405.
[2] 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.
[3] 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.