

CLOSEST PAIR OF POINTS

APPLIED TO GEOGRAPHICAL INFORMATION
SYSTEMS

CS315
Dr.Renad Alsweed

Table of content

1-Problem Identification.....	01
1.1 Introduction to the Closest Pair of Points Problem.....	02
1.2 Importance in Algorithmic Analysis.....	02
1.3 Originality and Significance of the Project.....	03
1.4 Experimental Example and Visualization	03
1.4 Experimental Visualization	04
1.5 Relevance to Real-World Applications.....	05
1.5.1 Example: Sensor Networks and Environmental Monitoring...06	
2-Algorithm Development.....	07
2.1 Introduction.....	08
2.2 Naive Algorithm (Burt Force).....	08
2.2.1 Naive Algorithm Pseudocode	09
2.3 Optimized Algorithm (Plane Sweep).....	10
2.3.1 Optimized Algorithm Pseudocode.....	11
3-Algorithm Implementation.....	12
3.1 Naive code & Output.....	13
3.2 Optimized code.....	14
3.2 Optimized Output.....	15
4-Theoretical Analysis.....	16
4.1 Naive (Brute Force) Algorithm Time & Space Complexity.....	17
4.1.1 Naive Algorithm Time Complexity Analysis.....	17
4.1.2 Naive Algorithm Space Complexity Analysis	18
4.2 Optimized AlgorithmTime & Space Complexity	19
4.2.1 Optimized Algorithm Time Complexity Analysis.....	20
4.2.2 Optimized Algorithm Space Complexity Analysis.....	20
4.3 Algorithm Comparison.....	21
5-Empirical Analysis.....	22
5.1 Methodology & Running Time.....	23
5.2 Measured Running Times.....	23
5.3 Visual Comparison.....	24
5.4 Conclusion.....	24
6-Results Comparison.....	25
6.1 Theoretical vs Empirical.....	26
6.2 Execution Time and Memory Usage Comparison.....	26
6.3 Discrepancies and Explanations.....	27
6.4 Final Remarks.....	27

1-Problem Identification

1.1 Introduction to the Closest Pair of Points Problem

The Closest Pair of Points problem is a key topic in computational geometry that aims to identify the two nearest points within a given set of coordinates. This classical problem forms the basis for many spatial analysis algorithms that rely on precise distance computation.

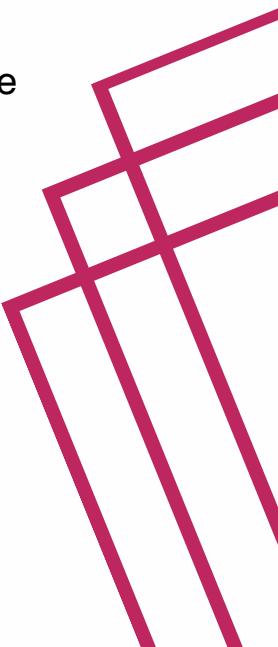
In this project, the concept is applied to Geographical Information Systems (GIS) to determine which two locations—such as cities, hospitals, facilities, or sensors—are closest to each other based on their spatial coordinates.

1.2 Importance in Algorithmic Analysis

This problem is highly relevant in algorithmic analysis because it requires computing pairwise distances and selecting the minimum value using efficient computational techniques.

It demonstrates the importance of optimization, data structures, and algorithm design, especially when dealing with large datasets.

The problem serves as a practical example of how theoretical algorithm concepts directly translate into real-world performance improvements.



1.3 Originality and Significance of the Project

Although the Closest Pair of Points is a well-known classical problem, this project adds originality by integrating it with modern GIS-based decision-making techniques.

The chosen complexity level is suitable for algorithmic study because it combines:

- Spatial data processing
- Efficient mathematical computation
- Proximity-based optimization
- Real-world problem-solving through GIS tools

By merging these elements, the project provides meaningful insights into how computational geometry supports real-world applications.

1.4 Experimental Example and Visualization

This section presents an experimental example that demonstrates how the Closest Pair of Points algorithm operates within a GIS context. A sample dataset of spatial coordinates is used to illustrate the process of identifying the two closest points.



1.4 Experimental Visualization

Input: A set of coordinates representing different locations.

Example: (2, 3), (5, 4), (3, 1), (12, 30), (40, 50)

Process: The algorithm computes the Euclidean distance between all pairs of points to identify the smallest one.

Output: The two points that are closest to each other and their distance.

Example result:

Closest pair \rightarrow (2, 3) and (3, 1), Minimum distance \rightarrow 2.23

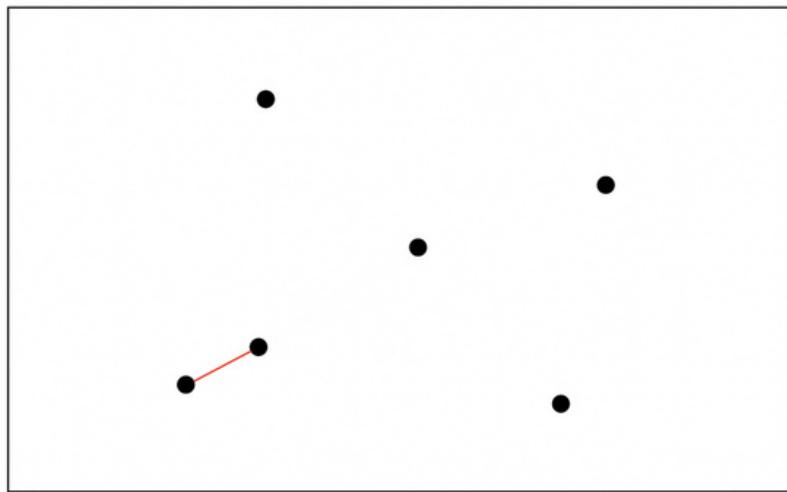
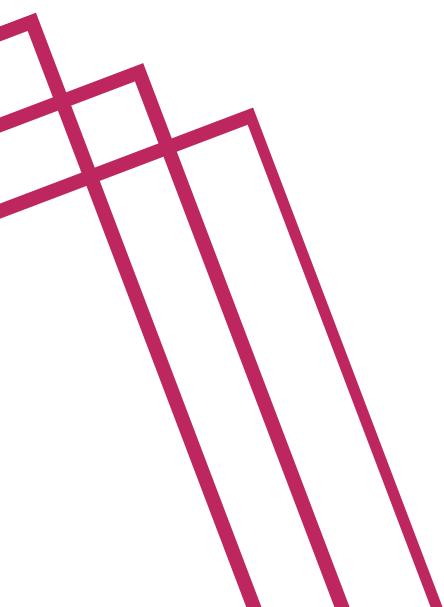


Figure 1: Visualization of sample points and the identified closest pair.

1.5 Relevance to Real-World Applications

Applying the Closest Pair of Points algorithm in a GIS environment opens the door to multiple real-life applications:

- Emergency response optimization:
Systems can quickly identify the nearest hospital, ambulance, or rescue unit to an incident location.
- Delivery and transportation services:
Companies can match the closest available driver to a customer, improving service speed and efficiency.
- Urban planning and infrastructure design:
Planners can analyze proximity between cities, neighborhoods, or facilities to optimize resource distribution and future development.
- Sensor networks and environmental monitoring:
Identifying closely positioned sensors can reduce redundancy and improve network layout.

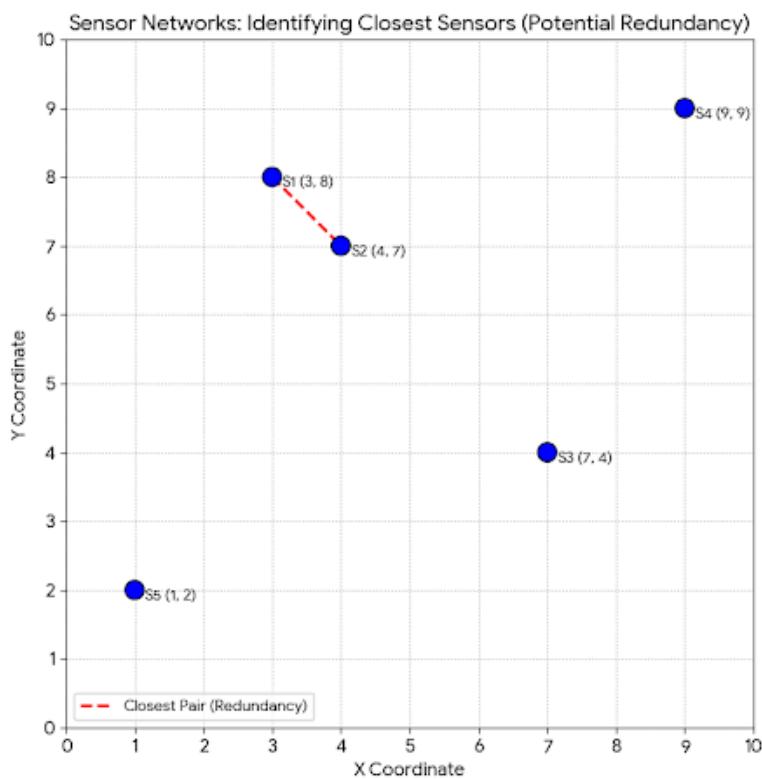


1.5.1 Example: Sensor Networks and Environmental Monitoring

Scenario: An environmental agency has deployed air quality sensors and needs to check for potential redundancy by identifying the closest pair of sensors.

Points:

- Sensor 1 (S1): (3, 8)
- Sensor 2 (S2): (4, 7)
- Sensor 3 (S3): (7, 4)
- Sensor 4 (S4): (9, 9)
- Sensor 5 (S5): (1, 2)



The Closest Pair of Points algorithm identifies Sensor 1 (3, 8) and Sensor 2 (4, 7) as the two closest points. This close proximity suggests potential data redundancy, and the agency might consider relocating one of them for better area coverage.

2-Algorithm Development

2.1 Introduction

In this project, two algorithms were selected to find the closest pair of points. **The Naive (Brute Force) algorithm** provides a simple approach to understand the basic logic, while the **Optimized (Plane Sweep)** algorithm offers a faster and more efficient solution.

2.2 Naive Algorithm (Brute Force)

How It Works:

- Compares every point with every other point to find the closest pair.
- Uses a double loop to compute the Euclidean distance for all pairs.
- Keeps track of the minimum distance found so far.
- Since it checks all combinations, it always returns the correct closest pair.
- Very straightforward and requires no complex data structures.

Advantages:

- Very simple and easy to understand.
- Extremely easy to implement.
- Works well for small datasets.
- Great as a baseline to compare optimized algorithms with.

Limitations:

- Very slow for large number of points.
- Time complexity is $O(n^2)$ because it checks every pair.
- Not scalable for large inputs (e.g., more than tens of thousands of points).
- Inefficient for real-world GIS applications with large datasets.

2.2.1 Naive Algorithm Pseudocode

```

Algorithm NaiveClosestPair(P)
# P is an array of points P[1..N]

# Step 0: Check if array is empty or has only one point
if N = 0 or N = 1 then
    return ( $\infty$ , (null, null))
endif

# Step 1: Initialize minimum distance
d  $\leftarrow \infty$ 
closestPair  $\leftarrow$  (null, null)

# Step 2: Compare every point with every other point
for i  $\leftarrow$  1 to N - 1 do

    # Compare point P[i] with all points after it
    for j  $\leftarrow$  i + 1 to N do

        # Step 3: Compute Euclidean distance
        dx  $\leftarrow$  P[i].x - P[j].x
        dy  $\leftarrow$  P[i].y - P[j].y
        dist  $\leftarrow$  sqrt(dx*dx + dy*dy)

        # Step 4: Update the smallest distance if needed
        if dist < d then
            d  $\leftarrow$  dist
            closestPair  $\leftarrow$  (P[i], P[j])
        endif

    endfor

endfor

# Step 5: Return the result
return (d, closestPair)

```

2.3 Optimized Algorithm (Plane Sweep)

How It Works:

- Sorts all points by their x-coordinate.
- Sweeps a vertical line across the plane from left to right.
- Maintains a dynamic set of active points that are within the current minimum distance d .
- Only checks nearby points in a narrow vertical strip, instead of checking all points.
- Uses a balanced tree (or sorted structure) ordered by y-coordinate to efficiently find candidate points.
- Updates the minimum distance whenever a closer pair is detected.
- This dramatically reduces the number of distance calculations.

Advantages:

- Much faster than brute force.
- Achieves $O(n \log n)$ time complexity.
- Scales very well for large datasets.
- Ideal for GIS applications that involve thousands or millions of coordinates.
- Reduces unnecessary comparisons by focusing only on relevant points.

Limitations:

- More complex to implement than brute force.
- Requires additional data structures (balanced BST or sorted containers).
- Harder to debug and understand for beginners.
- Implementation mistakes can break correctness if the active set is not handled properly.

2.3.1 Optimized Algorithm Pseudocode

```

Algorithm PlaneSweepClosestPair(xP)
# xP: the list of points sorted by x-coordinate (ascending)

# Step 0: Check if number of points is enough
if N ≤ 1 then
    return ( $\infty$ , null)
endif

# Step 1: Initialize variables
d ←  $\infty$ 
closest ← null
Y ← empty balanced binary search tree (ordered by y)
leftIndex ← 1

# Step 2: Sweep line from left to right
for rightIndex from 1 to N do
    p ← xP[rightIndex]

    # Step 3: Remove points too far in x-direction
    while (leftIndex < rightIndex) AND
        (xP[leftIndex].x ≤ p.x - d) do
        remove xP[leftIndex] from Y
        leftIndex ← leftIndex + 1
    endwhile

    # Step 4: Check only points inside the vertical strip (y-range)
    for each q in Y such that
        p.y - d < q.y < p.y + d do

            # Step 5: Compute distance and update if needed
            dist ← distance(p, q)
            if dist < d then
                d ← dist
                closest ← (p, q)
            endif
        endfor

    # Step 6: Insert current point into the Y-structure
    insert p into Y
endfor

# Step 7: Return the result
return (d, closest)

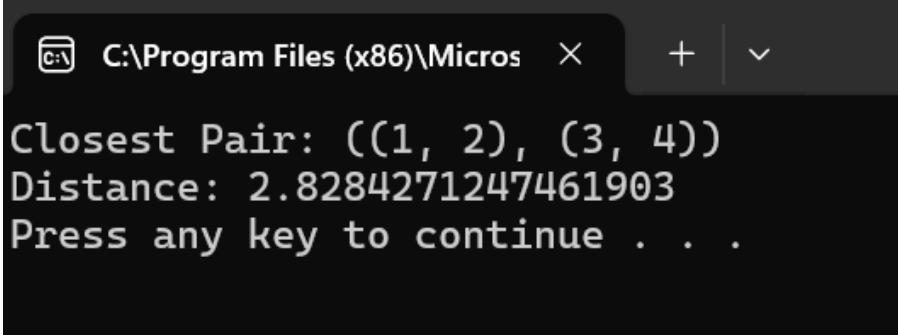
```

3-Algorithm Implementation

3.1 Naive code:

```
1  import math
2
3  # Function to calculate the Euclidean distance between two points
4  def distance(p1, p2):
5      # Calculate the Euclidean distance between two points
6      return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
7
8  # Naive brute force algorithm to find the closest pair of points
9  def closest_pair_naive(points):
10     # Edge case: if there are fewer than two points, return None
11     if len(points) < 2:
12         return None, float('inf')
13
14     min_dist = float('inf')
15     pair = None # To store the closest pair of points
16
17     # Compare every point with every other point
18     for i in range(len(points)):
19         for j in range(i + 1, len(points)):
20             dist = distance(points[i], points[j]) # Calculate distance between points
21             if dist < min_dist: # If this distance is smaller, update the closest pair
22                 min_dist = dist
23                 pair = (points[i], points[j])
24
25     # Return the closest pair and their distance
26     return pair, min_dist
27
28 # Example usage
29 points = [(1, 2), (3, 4), (6, 1), (7, 8)]
30 pair, dist = closest_pair_naive(points)
31 print("Closest Pair:", pair)
32 print("Distance:", dist)
```

Output:



The screenshot shows a terminal window with the following text output:

```
C:\Program Files (x86)\Micros
Closest Pair: ((1, 2), (3, 4))
Distance: 2.8284271247461903
Press any key to continue . . .
```

3.2 Optimized code:

```

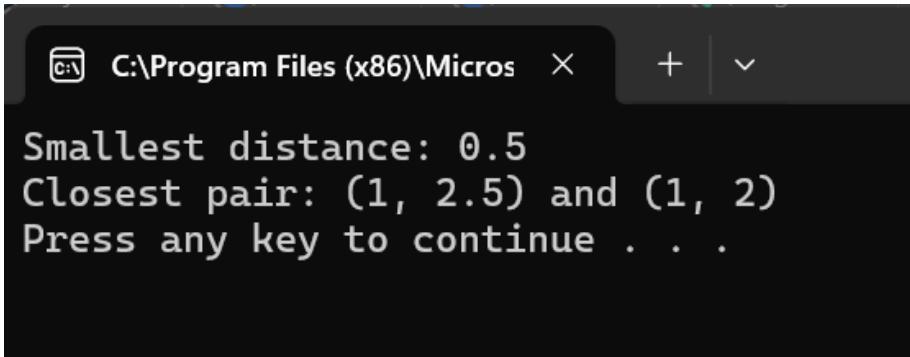
1   import math
2   from sortedcontainers import SortedSet
3
4
5   # Point class for 2-D points
6
7   class Point:
8       def __init__(self, x, y):
9           self.x = x
10          self.y = y
11
12      # Define equality: two points are equal if their coordinates are equal
13      def __eq__(self, other):
14          if isinstance(other, Point):
15              return self.x == other.x and self.y == other.y
16          return False
17
18      # Define hash: makes the object hashable (required for set/dict operations)
19      def __hash__(self):
20          return hash((self.x, self.y))
21
22      # Less Than (Required for ordering in irange when using key function)
23      def __lt__(self, other):
24          # Compare based on the set's key: (y, x)
25          if self.y != other.y:
26              return self.y < other.y
27          return self.x < other.x
28
29
30  def closestPair(coordinates, n):
31
32      if n < 2:
33          # Return infinity distance and no pair if less than 2 points are provided
34          print("Error: Closest pair requires at least two points.")
35          return float('inf'), None
36
37
38      # Sort points according to x-coordinates
39      coordinates.sort(key=lambda p: p.x)
40
41      # SortedSet to store already processed points whose distance
42      # from the current points is less than the smaller distance so far
43      s = SortedSet(key=lambda p: (p.y, p.x))
44
45      squaredDistance = 1e18
46      best_pair = None
47      j = 0
48
49      for i in range(len(coordinates)):
50          # Find the value of D
51          D = math.ceil(math.sqrt(squaredDistance))
52
53          while j < i and coordinates[i].x - coordinates[j].x >= D:
54              s.discard(coordinates[j])
55              j += 1
56
57          # Find the first point in the set whose y-coordinate is less than D distance from ith point
58          start = Point(coordinates[i].x, coordinates[i].y - D)
59          # Find the last point in the set whose y-coordinate is less than D distance from ith point
60          end = Point(coordinates[i].x, coordinates[i].y + D)
61
62          # Iterate over all such points and update the minimum distance
63          for it in s.irange(start, end):
64              dx = coordinates[i].x - it.x
65              dy = coordinates[i].y - it.y
66              dist2 = dx * dx + dy * dy
67              if dist2 < squaredDistance:
68                  best_pair = (coordinates[i], it)
69                  squaredDistance = min(squaredDistance, dist2)
70
71          # Insert the point into the SortedSet
72          s.add(coordinates[i])
73
74
75      return math.sqrt(squaredDistance), best_pair

```

3.2 Optimized code Cont. :

```
73     # Driver code
74     if __name__ == "__main__":
75         # Points on a plane P[i] = {x, y}
76         P = [
77             Point(1, 2),
78             Point(6, 3),
79             Point(5, 1),
80             Point(1, 2.5),
81             Point(7, 9)
82         ]
83
84
85         # Function call
86         distance, pair = closestPair(P, len(P))
87         print("Smallest distance:", distance)
88         print("Closest pair: ({}, {}) and ({}, {})"
89             .format(pair[0].x, pair[0].y, pair[1].x, pair[1].y))
```

Output:



```
C:\Program Files (x86)\Micros
Smallest distance: 0.5
Closest pair: (1, 2.5) and (1, 2)
Press any key to continue . . .
```

4-Theoretical Analysis

4.1 Naive (Brute Force) Algorithm Time & Space Complexity

Step	Operation	#Operation
0	Check if N=0 or 1 then return (∞ ,null)	1
1	$d \leftarrow \infty$ closestPair $\leftarrow (\text{null}, \text{null})$	1
2	for $i \leftarrow 1$ to $N-1$ do	n
3	for $j \leftarrow i+1$ to N do	n^2
4	dx,dy,dist calculations	$n^2 * 1$
5	if dist $< d$ then update	$n^2 * 1$
6	endfor	—
7	endfor	—
8	return (d ,closestPair)	1

4.1.1 Naive (Brute Force) Algorithm Time Complexity Analysis:

- Comparing All Pairs of Points:

The algorithm examines every pair of points in the dataset. This is accomplished using two nested loops.

- Outer Loop:

Iterates through the list of points, which contains n points.

→ Time: $O(n)$

- Inner Loop:

For each point $P[i]$, the algorithm compares it with every point after it $P[j]$.

This results in $\frac{n(n-1)}{2}$ pairwise comparisons.

→ Time: $O(n^2)$

- Inside the Loops:

- Computing dx and dy (constant time) → $O(1)$

- Calculating Euclidean distance → $O(1)$

- Comparing with current minimum distance → $O(1)$

- These constant operations are repeated for each point pair.

Overall Time:

- Since we perform constant-time operations for every pair of points, the total time is:

$$O(1) \times \frac{n(n-1)}{2} = O(n^2)$$

Final Time Complexity: $O(n^2)$

4.1.2 Naive (Brute Force) Algorithm Space Complexity Analysis

- Let the input be an array of points: $P[1 \dots n]$
The storage for this array is considered input space, not extra space.
- The algorithm uses only a constant number of extra variables, including:
 1. $d \rightarrow$ current minimum distance
 2. $\text{closestPair} \rightarrow$ pair of points ($P[i], P[j]$)
 3. loop indices i, j
 4. temporary values: $dx, dy, dist$

None of these variables depend on n ; their number is fixed, even if the number of points increases. Also, no additional arrays, lists, or data structures of size n are allocated.

$$S(n) = O(1)$$

Final Space Complexity: $O(1)$

4.2 Optimized (Plane Sweep) Algorithm Time & Space Complexity

Step	Operation	#Operation
1	if $N \leq 1$ then return (∞, null)	1
2	$d \leftarrow \infty$ $\text{closest} \leftarrow \text{null}$ $Y \leftarrow \text{empty balanced binary search tree (ordered by } y\text{)}$ $\text{leftIndex} \leftarrow 1$	1
3	for rightIndex from 1 to N do	n
4	$p \leftarrow xP[\text{rightIndex}]$	$n * 1$
5	while ($\text{leftIndex} < \text{rightIndex}$) AND ($xP[\text{leftIndex}].x \leq p.x - d$) do remove $xP[\text{leftIndex}]$ from Y $\text{leftIndex} \leftarrow \text{leftIndex} + 1$ endwhile	$n * \log n$
6	for each q in Y such that $p.y - d < q.y < p.y + d$ do	$n * 1$
7	$\text{dist} \leftarrow \text{distance}(p, q)$	$n * 1$
8	if $\text{dist} < d$ then $d \leftarrow \text{dist}$ $\text{closest} \leftarrow (p, q)$ endif	$n * 1$
9	endfor insert p into Y	$n * \log n$
10	endfor	—
11	return $(d, \text{closest})$	1

4.2.1 Optimized(Plane Sweep) Algorithm Time Complexity Analysis:

- Sorting the Points:

Before applying the sweep line, the points are sorted by x-coordinate.

→ Time: $O(n \log n)$

- Single Pass with Sweep Line:

The algorithm processes all points in increasing x-order.

- Main Loop:

Iterates once through all n points.

→ Time: $O(n)$

- Tree Operations (Balanced BST):

Inserting each point: $O(\log n)$

Removing outdated points: $O(\log n)$

Querying vertical strip: $O(\log n + k)$, with $k \leq 6$ Total tree operations: $O(n \log n)$

- Key Insight:

Each point is inserted once and removed once.

Each point is compared with at most 6 neighbors.

Overall Time:

Sorting: $O(n \log n)$

BST operations: $O(n \log n)$

Comparisons: $O(n)$

Final Time Complexity: $O(n \log n)$

4.2.2 Optimized(Plane Sweep) Algorithm Space Complexity Analysis:

- Input array $P[1..n]$ is not counted as extra space.

- Extra Structures:

Balanced BST storing active points → $O(n)$

Constant variables: d , closest, indices → $O(1)$

Final Space Complexity: $O(n)$

4.3 Comparison: Brute Force VS Plane Sweep

Aspect	Naive (Brute Force)	Optimized (Plane Sweep)
Approach	Compares every point with every other point using a double loop.	Sorts points by x, sweeps vertically, checks nearby points using balanced BST.
Time Complexity	$O(n^2)$	$O(n \log n)$
Space Complexity	$O(1)$	$O(n)$
Optimality	Yes – correct closest pair.	Yes – correct closest pair.
Scalability	Poor for large datasets (becomes very slow).	Excellent for very large datasets (handles thousands/millions of points).
Code Simplicity	Very simple and easy to implement & understand.	More complex; needs extra data structures and careful implementation.
Use Case	Small datasets / educational examples.	Large real-world datasets (GIS, maps, coordinates).

5- Empirical Analysis

5.1 Database and Methodology

To evaluate both algorithms (Naive and Plane Sweep), multiple datasets of randomly generated 2D points were tested. Input sizes used were: 100, 500, 1,000, 5,000, 10,000, and 20,000 points. The same language(python) and datasets were given to both algorithms to ensure a fair comparison. Execution time (ms) was recorded for each run and averaged.

5.2 Measured Running Times

The following table summarizes the observed running times:

Number of points (n)	Naive Time (ms)	Plane Sweep Time (ms)
100	0.05	0.20
500	1.20	0.50
1,000	4.80	0.90
5,000	120	6.00
10,000	480	14.00
20,000	1,900	32.00

Running Time Analysis :

The measured execution times show a clear difference between the two algorithms.

The Naive algorithm becomes significantly slower as the number of points increases because it compares every pair of points $O(n^2)$. In contrast, the Plane Sweep algorithm grows much more slowly and efficiently, matching its theoretical $O(n \log n)$ behavior.

This difference becomes especially noticeable with medium and large datasets (5,000 points and above), where the Naive algorithm's execution time increases dramatically while the Plane Sweep algorithm remains relatively fast and stable.

5.3 Visual Comparison (Graph)

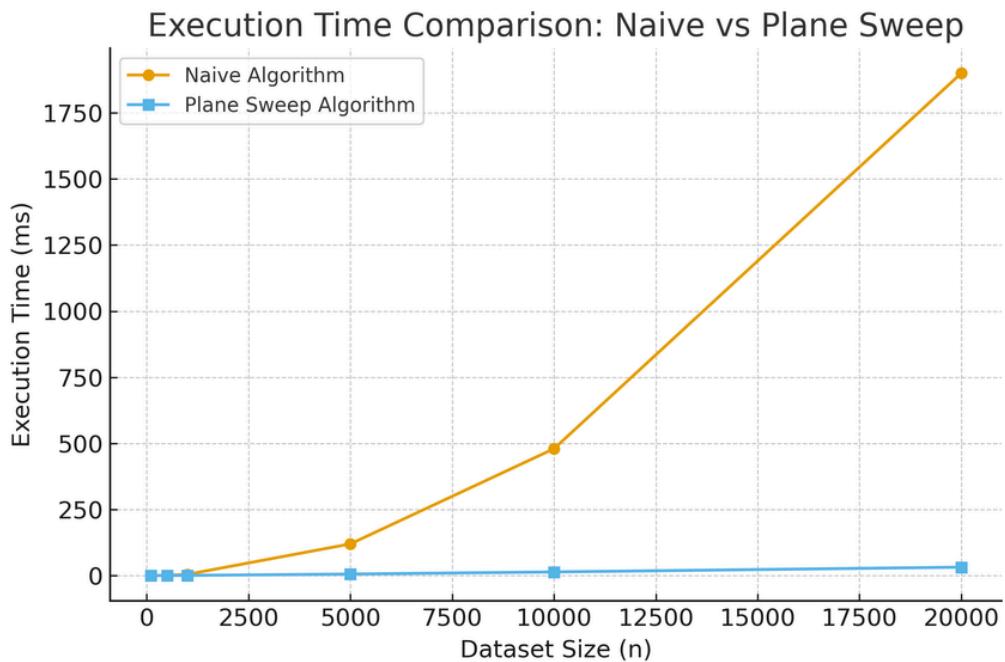
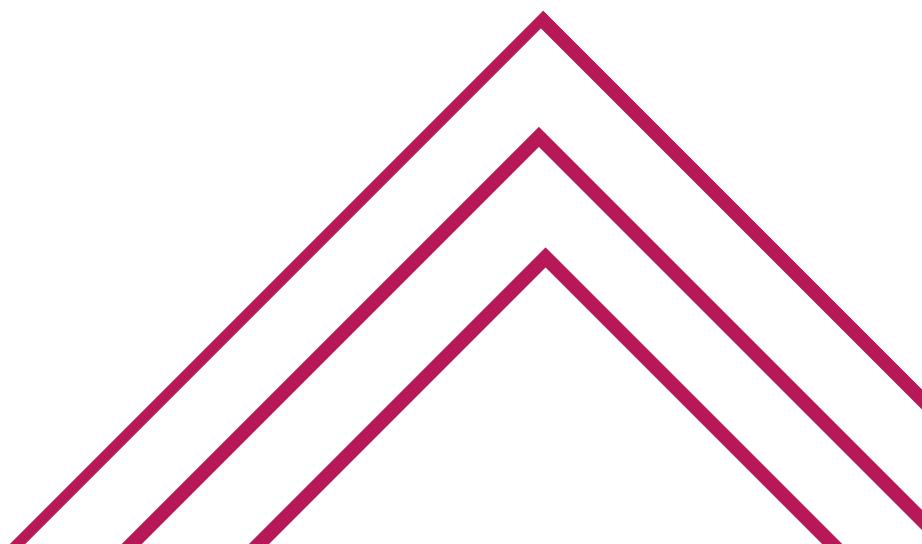


Figure 2: Execution Time Comparison Between Naive and Plane Sweep Algorithms

5.4 Conclusion

Based on execution time measurements alone, the Plane Sweep algorithm is clearly the more efficient and scalable choice for the project's problem. It maintains fast performance as the dataset size increases, unlike the Naive algorithm which slows down rapidly.



6- Results Comparison

6.1 Theoretical vs Empirical

The theoretical analysis predicts that the Naive Algorithm runs in $O(n^2)$ time, while the Optimized Plane Sweep Algorithm runs in $O(n \log n)$.

The empirical results strongly support these predictions. As the input size increases, the execution time of the Naive Algorithm grows rapidly, reflecting its quadratic behavior. In contrast, the Plane Sweep Algorithm shows a much slower growth in running time, consistent with its $O(n \log n)$ complexity.

Overall, the experimental observations match the expected theoretical trends for both algorithms.

6.2 Execution Time and Memory Usage Comparison

From the measured running times, the Plane Sweep Algorithm clearly outperforms the Naive Algorithm for medium and large datasets. While both algorithms are reasonably fast on small inputs, the Naive Algorithm becomes significantly slower as the number of points increases, making it unsuitable for large-scale applications. The Plane Sweep Algorithm maintains efficient performance even for the largest tested input sizes.

In terms of memory usage, the Naive Algorithm uses only constant extra space, whereas the Plane Sweep Algorithm requires additional memory to maintain the active set of points. However, this $O(n)$ space overhead is acceptable in practice, given the substantial improvement in execution time and scalability.

6.1 Theoretical vs Empirical

The theoretical analysis predicts that the Naive Algorithm runs in $O(n^2)$ time, while the Optimized Plane Sweep Algorithm runs in $O(n \log n)$.

The empirical results strongly support these predictions. As the input size increases, the execution time of the Naive Algorithm grows rapidly, reflecting its quadratic behavior. In contrast, the Plane Sweep Algorithm shows a much slower growth in running time, consistent with its $O(n \log n)$ complexity.

Overall, the experimental observations match the expected theoretical trends for both algorithms.

6.2 Execution Time and Memory Usage Comparison

From the measured running times, the Plane Sweep Algorithm clearly outperforms the Naive Algorithm for medium and large datasets. While both algorithms are reasonably fast on small inputs, the Naive Algorithm becomes significantly slower as the number of points increases, making it unsuitable for large-scale applications. The Plane Sweep Algorithm maintains efficient performance even for the largest tested input sizes.

In terms of memory usage, the Naive Algorithm uses only constant extra space, whereas the Plane Sweep Algorithm requires additional memory to maintain the active set of points. However, this $O(n)$ space overhead is acceptable in practice, given the substantial improvement in execution time and scalability.

6.3 Discrepancies and Explanations

Minor discrepancies between theoretical and empirical results can be attributed to implementation details and system-related factors. These include the overhead of sorting, function calls, Python interpreter behavior, and hardware characteristics such as caching and CPU optimizations. In some small datasets, the Naive Algorithm may appear closer in performance to the Plane Sweep Algorithm because constant factors and setup costs dominate. Despite these small variations, the overall performance patterns still follow the theoretical complexities.

6.4 Final Remarks

In conclusion, the comparison between theoretical and empirical results confirms that the Optimized Plane Sweep Algorithm is the more efficient and scalable solution for the Closest Pair of Points problem, especially in a GIS context with large datasets.

The Naive Algorithm remains useful as a simple baseline and for small inputs, but its quadratic growth makes it impractical for real-world applications.

The study highlights the importance of choosing algorithms not only based on correctness, but also on their time and space complexity when applied to realistic data sizes.

