

COLLEGE OF ENGINEERING TRIVANDRUM



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CST 306
ALGORITHM ANALYSIS AND DESIGN
Closest Pair Problem using Divide and Conquer
Algorithm Design Strategy
Assignment - 4

Don Joshi N
Roll no 23
TVE20CS044

Closest Pair Problem using Divide and Conquer Algorithm Design Strategy

1 Task

Closest Pair Problem using Divide and Conquer Algorithm Design Strategy

The Closest Pair Problem(CSP) is an important problem in computational geometry that involves finding the two points in a given set that are closest to each other. Design and implement an algorithm using divide and conquer to solve the CSP with the following requirements:

(a) Problem Definition:

i. Clearly define the Closest Pair Problem, where the task is to find the two closest points in a set of points.

ii. Specify the input format, such as a set of 2D points represented by their coordinates.

(b) Divide and Conquer Approach:

i. Describe the divide and conquer algorithm design strategy and its application in solving the Closest Pair Problem.

ii. Explain how the set of points can be divided and combined to find the closest pair efficiently.

(c) Algorithm Design:

i. Design a divide and conquer algorithm to solve the Closest Pair Problem.

ii. Outline the steps involved, including the divide step, conquer step, and combine step.

(d) Time and Space Complexity Analysis:

i. Analyze the time complexity of the algorithm in the worst-case scenario.

ii. Discuss the space complexity and any additional space requirements of the algorithm.

(e) Implementation:

i. Implement the divide and conquer algorithm in a programming language of your choice.

ii. Test the algorithm on different-sized input sets of points and evaluate its correctness.

(f) Performance Evaluation:

i. Analyze the running time of the algorithm for various input sizes and compare it to the brute-force approach (comparing all pairs).

ii. Present the results using appropriate graphs or tables and discuss the observations and conclusions.

(g) Real-World Application:

i. Discuss a real-world scenario where finding the closest pair of points is important (e.g., robotics, geographic information systems, computer graphics).

ii. Explain how the Closest Pair algorithm can be applied to solve a problem in that scenario.

Write a detailed algorithm description that outlines the steps involved in solving the Closest Pair Problem using divide and conquer. Provide clear pseudocode or code snippets illustrating the key components of the algorithm. Explain any design decisions made during the implementation process.

In addition to the algorithm description, provide a working implementation of the algorithm in a programming language of your choice. Test the algorithm with different input scenarios and evaluate its performance.

Submit your algorithm description, implementation code, and a report discussing the efficiency, performance, and any challenges encountered during the implementation process.

2 Closest Pair Problem

The Closest Pair Problem is a classic computational problem that aims to find the two closest points in a given set of points in a two- or three-dimensional space. The objective is to determine the pair of points with the minimum distance between them. This problem has important applications in various fields, including computational geometry, computer graphics, and geographic information systems. Solving the Closest Pair Problem efficiently requires employing advanced algorithms, such as divide and conquer or geometric hashing. These algorithms typically involve recursive partitioning of the point set and employing pruning techniques to reduce the search space. By finding the closest pair, this problem allows for optimizing various processes, such as determining nearest neighbors, data clustering, and collision detection. The Closest Pair Problem serves as a fundamental problem in computational mathematics, facilitating the development of efficient algorithms and fostering advancements in numerous domains.

2.1 Using divide and conquer algorithm design strategy

The Closest Pair Problem can be effectively solved using the Divide and Conquer algorithm design strategy.

- **Divide:** The first step is to divide the set of points into two equal-sized subsets based on their x-coordinate. This division creates a vertical line that splits the points into a left subset and a right subset. This step is done recursively until a base case is reached.
- **Conquer:** In this step, the algorithm recursively solves the Closest Pair Problem for each subset independently. This is done by applying the Divide and Conquer strategy to both the left and right subsets, resulting in two sets of closest pairs for each subset.
- **Combine:** After obtaining the closest pairs for the left and right subsets, the algorithm needs to consider pairs that cross the dividing line. To find these potential pairs, it identifies a strip of points within a certain distance from the dividing line. By sorting the strip based on their y-coordinate, the algorithm can efficiently examine only the nearby points.
- **Merge:** Finally, the algorithm merges the results obtained from the left subset, right subset, and the strip of points near the dividing line. It selects the pair of points with the minimum distance among all the pairs considered so far.

By employing the Divide and Conquer strategy, the algorithm reduces the problem's complexity by dividing it into smaller subproblems. The recursive approach allows the algorithm to efficiently find the closest pair in each subset and then merge the results to obtain the overall closest pair. This algorithmic design strategy yields an efficient solution for the Closest Pair Problem and has a time complexity of $O(n \log n)$, where n is the number of points.

Overall, the Divide and Conquer algorithm design strategy provides an effective and efficient solution for solving the Closest Pair Problem, making it a widely used approach in computational geometry and related fields.

2.2 Data structures

To solve the Closest Pair Problem using the Divide and Conquer algorithm design strategy, you would typically use the following data structures:

- **Point Array:** An array or list to store the set of points in the input. Each point would typically be represented as a data structure containing the x and y coordinates.

- **Sorted Array:** Sorted arrays can be used to sort the points based on their x or y coordinates. Sorting enables efficient searching and comparison operations.
- **Minimum Distance Pair:** A data structure to store the currently known minimum distance pair of points. This can be updated and compared as the algorithm progresses.
- **Variable :** To keep track of the minimum distance , a variable can be used.

2.3 Real world applications

The Closest Pair Problem has several real-world applications across various fields. Here are some examples:

- **Geographic Information Systems (GIS):** GIS applications often involve finding the closest pair of locations or determining proximity between points of interest. For instance, it can be used to find the nearest gas station, restaurant, or hospital from a given location.
- **Navigation and Routing:** Closest Pair algorithms are employed in navigation systems to identify the nearest points of interest, calculate optimal routes, or determine proximity between vehicles for collision avoidance systems.
- **Computer Vision and Image Processing:** The Closest Pair Problem is relevant in computer vision tasks such as object detection, image recognition, and feature matching. It helps in identifying similar or matching features between images.
- **Computational Biology:** In bioinformatics, the Closest Pair Problem is employed in tasks like DNA sequence comparison, protein structure matching, and genomic analysis to identify similar or closely related sequences or structures.
- **Robotics and Automation:** Closest Pair algorithms play a role in robotic path planning, robotic swarm coordination, and object detection for robot manipulation tasks.

3 Algorithm

- Define the function `findClosestPair(X)` that takes an input array `X` representing a set of points.
- Determine the number of points `n` in the array `X`.
- Check the base cases:
 - If `n` is equal to 2, calculate the distance between the two points `X[0]` and `X[1]`. Return the distance and the two points as the closest pair.
 - If `n` is equal to 3, calculate the distances between all three pairs of points (`X[0]` and `X[1]`, `X[0]` and `X[2]`, `X[1]` and `X[2]`). Return the closest pair among these three pairs.
- If the base cases do not apply, divide the input array `X` into two halves. Find the midpoint `mid` as `n` divided by 2.
- Recursively call `findClosestPair()` on the left half of `X` from index 0 to `mid-1`. Store the resulting closest pair in `dl`, `Al`, and `Bl`.

- Recursively call `findClosestPair()` on the right half of `X` from index `mid` to `n-1`. Store the resulting closest pair in `dr`, `Ar`, and `Br`.
- Determine the closest pair among the left and right halves by calling the `minDist()` function with `dl`, `Al`, `Bl`, `dr`, `Ar`, and `Br`. Store the resulting closest pair in `d`, `A`, and `B`.
- Initialize an empty list `S`.
- Set `dmin` to `d`, which represents the minimum distance found so far.
- Iterate over the points in the left half of `X` from index `0` to `mid-1`.
 - Check if the x-coordinate of the current point (`X[i]`) is within `d` units to the left of the midpoint (`X[mid]`).
 - If it is, iterate over the points in the right half of `X` from index `mid` to `n-1`.
 - * Check if the x-coordinate of the current point (`X[j]`) is within `d` units to the right of the midpoint (`X[mid]`).
 - * If it is, call the `minDist()` function with `dmin`, `A`, `B`, the distance between `X[i]` and `X[j]`, `X[i]`, and `X[j]`. Update `dmin`, `A`, and `B` with the closest pair found.
- Update `d` with the minimum value between `d` and `dmin`.
- Return `d`, `A`, and `B` as the closest pair of points in the input array `X`.

4 Analysis

The time complexity of the `findClosestPair()` function depends on the number of points in the input array `X`. Let `n` represent the number of points. In the base cases, when `n` is 2 or 3, the code performs a constant number of operations, resulting in a time complexity of $O(1)$. In the general case, the function recursively calls itself twice, dividing the input array into two halves. This step has a time complexity of $O(n/2)$ since the array is divided in half at each recursive call. After the recursive calls, the code iterates over the points within a certain range in the left half and the right half of the array. This nested loop has a time complexity of $O(n^2)$ in the worst case, as it considers pairs of points within a given range. Overall, the time complexity of the code can be expressed as a recurrence relation: $T(n) = 2 * T(n/2) + O(n^2)$. By applying the Master Theorem, we can conclude that the time complexity of the code is $O(n^2)$.

The space complexity of the code is determined by the recursive calls and the additional data structures used. The recursive calls consume stack space proportional to the depth of recursion, which is $O(\log n)$ in this case since the array is divided into two halves at each recursive call. The additional data structures used, such as the variables for storing the closest pairs, have a constant space complexity, resulting in $O(1)$. Also, point array takes $O(n)$ and it is same for sorted array. Hence, the overall space complexity of the code is $O(n)$.

. In summary, the time complexity of the code is $O(n^2)$ and the space complexity is $O(n)$.

5 Divide and conquer vs Brute force

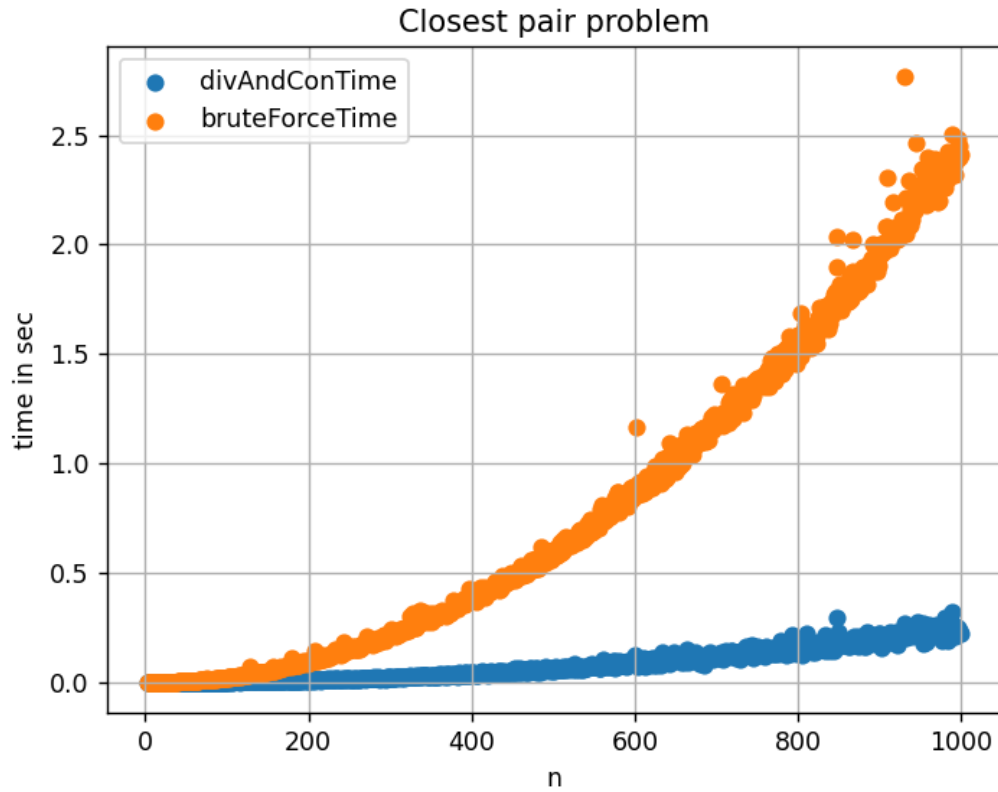


Figure 5.1: Output

The brute force approach directly compares the distance between every pair of points to find the closest pair. It exhaustively checks all possible pairs and selects the one with the minimum distance. While the brute force method guarantees finding the closest pair, it suffers from a time complexity of $O(n^2)$, where n is the number of points. This approach becomes impractical for larger datasets, as the number of pairwise comparisons grows rapidly.

Therefore, the divide and conquer approach provides a more efficient solution for the Closest Pair Problem by dividing the problem into smaller subproblems and utilizing the concept of merging. On the other hand, brute force offers a straightforward and guaranteed method but becomes inefficient for larger input sizes. The choice between the two approaches depends on the specific requirements and constraints of the problem at hand.

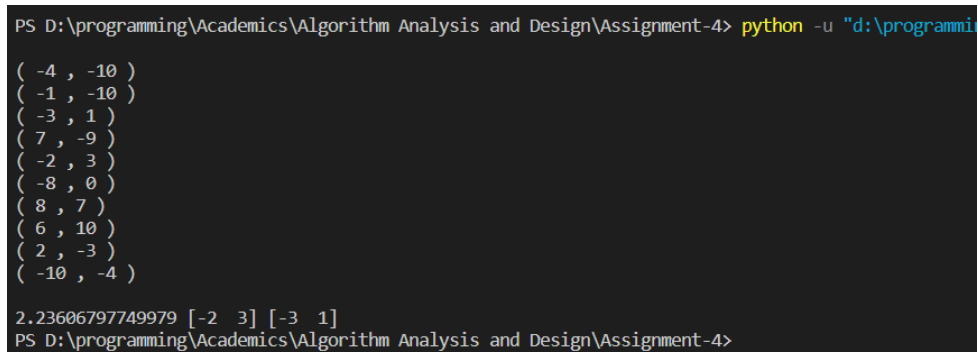
n	divide and conquer Time	Brute force Time
3	3.609999839682132e-05	2.2099997295299545e-05
14	0.000247500000114087	0.00041200000123353675
15	0.00020079999740119092	0.0004721000004792586
16	0.00018679999993764795	0.0005370999970182311
40	0.0006230999970284756	0.005102200000692392
84	0.0027164999992237426	0.015480300000490388
163	0.00561569999990752	0.06048910000026808
198	0.008819599999696948	0.08847080000123242
199	0.009660499999881722	0.0917236999994202
200	0.012905899999168469	0.0981761999973969
201	0.009101099996769335	0.09599659999730648
226	0.014526899998600129	0.12510430000111228
227	0.01333130000057281	0.12869399999908637
228	0.012412700001732446	0.1260495999995328
252	0.015533200003119418	0.15992650000043795
276	0.017631999999139225	0.1948695999994724
321	0.02891720000116038	0.24060480000116513
462	0.05654770000182907	0.4858717000024626
506	0.06028340000193566	0.5864751000008255
507	0.0635906999996223	0.5940865999982634
595	0.09641030000057071	0.8899711000012758
596	0.08769329999995534	0.835220400000253
623	0.07931959999768878	0.9603499999975611
624	0.09040939999977127	0.9904403000000457
625	0.10886419999951613	0.937258699999802
626	0.07793300000048475	0.9058928999984346
648	0.13948470000104862	1.0247735999982979
649	0.1048196999981883	0.9882285000021511
668	0.10503249999965192	1.0973578000011912
669	0.10011119999762741	1.0686258999994607
722	0.11535049999656621	1.2509312000001955
754	0.13053969999964465	1.3780264999986684
769	0.13645739999992657	1.421996000000945
837	0.1657275999968988	1.6162494000018341
838	0.1467943000025116	1.6740303999977186
852	0.17825109999830602	1.7040157000010367
899	0.215915100001439	1.9526168999982474
900	0.19163820000176202	1.9655129999991914
999	0.22659359999670414	2.4114083999957074

6 Result

The implementation of closest pair problem provides an accurate solution. The divide and conquer method gives a pretty quick solution and the space complexity can also be controlled. But more efficiency can be brought in the combine step by selecting only the required points or the points that can really concern the operation. The visual representation also helps to provide a better understanding of the problem and solution can be easily identified and cross checked with

the algorithm generated solution , thus can confirm the accuracy of problem. Also, one problem encountered is that the algorithm only picks one solution from the points. This is can be easily modified and can select all the points with minimum distance if more than one pair with same minimum distance exist.

7 Test



```
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4> python -u "d:\programmi
( -4 , -10 )
( -1 , -10 )
( -3 , 1 )
( 7 , -9 )
( -2 , 3 )
( -8 , 0 )
( 8 , 7 )
( 6 , 10 )
( 2 , -3 )
( -10 , -4 )

2.23606797749979 [-2  3] [-3  1]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4>
```

Figure 7.1: Output

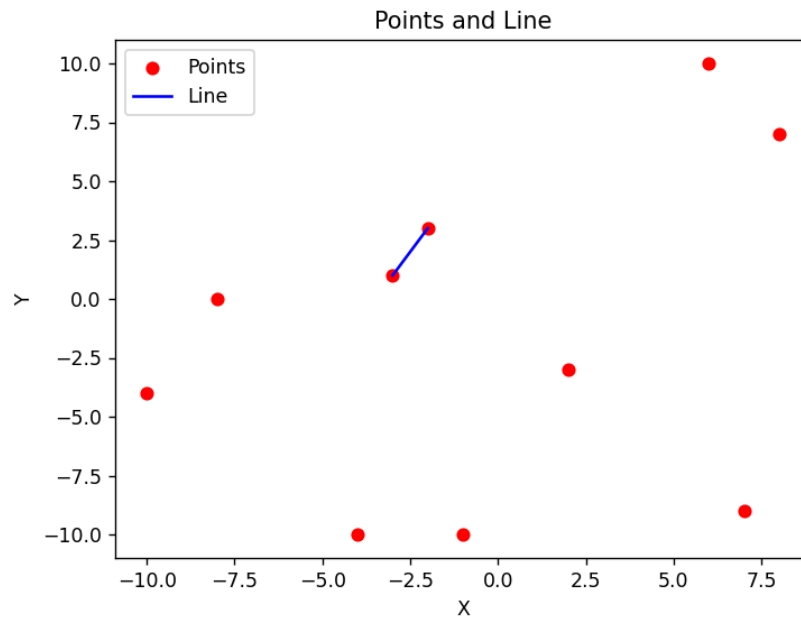


Figure 7.2: Output

```

PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4> python -u "d:\programming\Aca
( -1 , -1 )
( -3 , -5 )
( 9 , -8 )
( 0 , -8 )
( -8 , 0 )
( -1 , 3 )
( 7 , -8 )
( 9 , 2 )
( 8 , 9 )
( -10 , 5 )
( -4 , -9 )
( 3 , -8 )
( 8 , 0 )
( -5 , 5 )
( -2 , -3 )
( -8 , -3 )
( 0 , -2 )
( -2 , 10 )
( -10 , -4 )
( -6 , -1 )

```

```

1.4142135623730951 [-1 -1] [ 0 -2]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4> █

```

Figure 7.3: Output

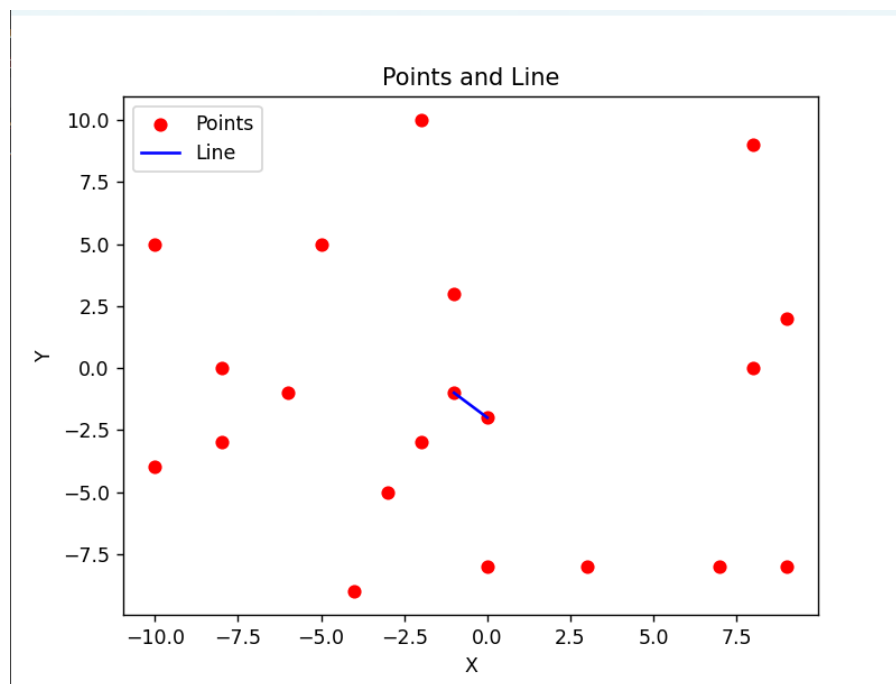


Figure 7.4: Output

```

PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4> python -u "d:\programming\Academics\Algorithm Analy
( 8, -4)
( 2, 4)
(-1, 1)
(-9, -1)
( 4, 0)
(10, 6)
(-8, -2)
(-9, -10)
(-1, -8)
(-6, 6)
( 5, 7)
( 2, 8)
(-2, -5)
(-2, 6)
( 6, 1)
( 7, -6)
(-10, 3)
(-3, -10)
( 2, -5)
(-3, -2)
( 6, -6)
( 5, -9)
( 8, -5)
( 7, -2)
( 4, 5)
(-4, -9)
(-8, 2)
( 5, 9)
(-3, 3)
( 8, 7)

1.0 [ 6 -6] [ 7 -6]
PS D:\programming\Academics\Algorithm Analysis and Design\Assignment-4> █

```

Figure 7.5: Output

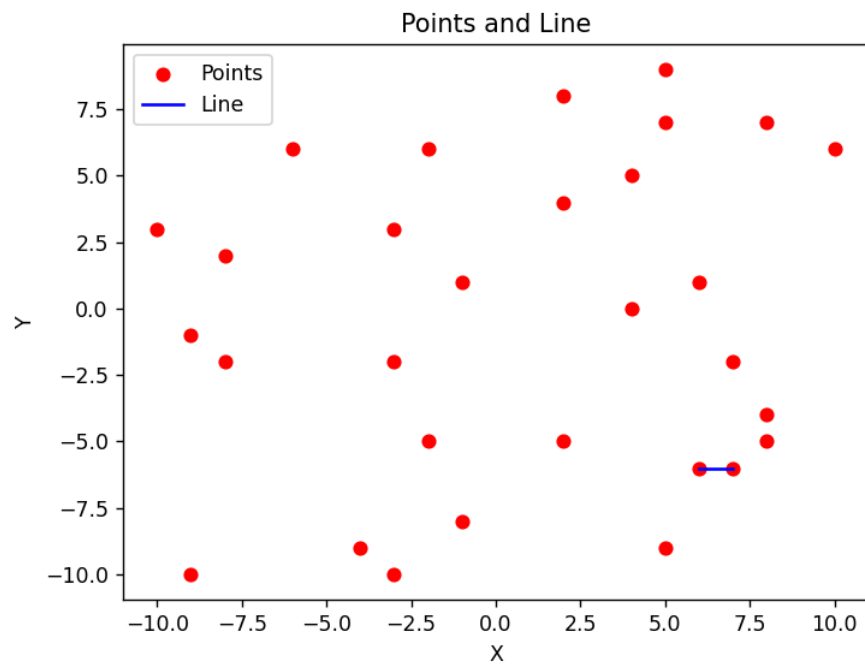


Figure 7.6: Output