Divide and Conquer Algorithm Implementation in Robot Collision Avoidance

Algorithm Implementation and Evaluation

Albert Ghazaly – 13522150
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: albertghzly@gmail.com

***Abstract*— This paper introduces a new approach to robot collision avoidance by using the divide and conquer algorithm to find the closest pairs of objects. Ensuring robots navigate safely without collisions is essential, especially as they operate in increasingly complex environments. Many existing methods are often slow or struggle to handle a large number of obstacles effectively. Our method breaks down the problem into smaller parts, solves each part separately, and then combines the results. This makes the process faster and more efficient. We tested our approach in different simulated settings and found that it improves both speed and accuracy. This paper explains how the algorithm works, how we implemented it, and how well it performed in our tests, showing its promise for real-world robotic applications.**

Keywords*— Robot collision avoidance, divide and conquer algorithm, closest pair of points problem, time complexity

I. Introduction

Robotic technology has advanced significantly, leading to a new era where robots must navigate complex environments safely and efficiently. A critical component of this navigation is collision avoidance. Collision avoidance is defined, according to various sources, as the ability to prevent collisions with obstacles while moving through an environment. The primary focus of this paper is to enhance collision avoidance by applying the divide and conquer algorithm to identify the closest pairs of objects. Identifying these pairs is crucial because it allows the robot to predict potential collision points and adjust its path accordingly.

Traditional methods for collision avoidance, such as potential field approaches and reactive techniques, often encounter limitations in terms of computational efficiency and scalability. These methods can struggle to process large numbers of obstacles quickly enough to ensure real-time responsiveness, potentially leading to suboptimal navigation paths or collisions. Our approach addresses these challenges by breaking down the collision avoidance problem into smaller, more manageable parts, solving each part individually, and then combining the results. This strategy improves both the speed and efficiency of the process.

In this paper, we will not only explain the divide and conquer algorithm but also demonstrate its implementation and results through source code examples and experimental data. Our goal is to show how this algorithm can be effectively used in real-world robotic applications, enhancing both safety and efficiency. We will discuss the time complexity of the algorithm, which is crucial for understanding and analyzing its performance.

The role of the divide and conquer algorithm in our approach is to systematically decompose the problem of collision avoidance. By focusing on the closest pair of objects, the algorithm allows the robot to navigate through environments more effectively. The divide and conquer strategy involves splitting the environment into smaller sections, processing each section to find close object pairs, and then merging the results to form a comprehensive solution. This method ensures that the robot can handle complex environments with numerous obstacles efficiently.

In the following sections, we will describe the divide and conquer algorithm in detail (Section II), Propose algorithm for the program containing each step of implementation (Section III), Implements the algorithm to a real program using python (Section IV), and discuss the algorithm performance compared to another algorithm (Section V), and analyses the cause of the algorithm performance comparison (Section VI). Our research shows that this approach not only improves computational efficiency but also enhances the accuracy of collision avoidance in dynamic environments.

II. Theoretical Basis

A. Divide and Conquer

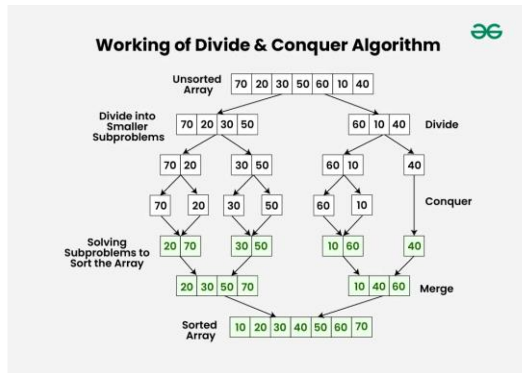

Fig 2.1 Divide and Conquer Algorithm
Source: https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm-data-structure-and-algorithm-tutorials/

The divide and conquer algorithm is a fundamental paradigm in computer science, renowned for its efficiency in solving complex problems by breaking them down into simpler subproblems. The divide and conquer approach involves three main steps:

1) Divide

The first step in the divide and conquer paradigm is to divide the original problem into smaller, independent subproblems. This division is usually performed in a manner that the subproblems are of approximately equal size, ensuring a balanced workload. In the context of the closest pair problem, the set of points (or objects) is typically divided based on a median line.

2) Conquer.

Once the problem is divided into smaller subproblems, each subproblem is solved recursively. The recursion continues until the subproblems are small enough to be solved directly. In many cases, this direct solution is straightforward and computationally inexpensive

3) Combine

The final step involves combining the solutions of the subproblems to form the solution to the original problem. This is often the most intricate part of the divide and conquer strategy, as it requires careful consideration to ensure that the combined solution is correct and efficient.

B. Brute Force

Brute force is a straightforward method for solving problems by exhaustively searching through all possible solutions and selecting the best one. This approach involves systematically checking every possible solution and evaluating its quality against a predefined criterion. While conceptually simple, brute force algorithms can be inefficient, especially for large problem instances, as they require examining every possible solution.

Despite its simplicity, brute force can be useful for solving small-scale problems or serving as a benchmark for evaluating the performance of more sophisticated algorithms. In some cases, optimizations can be applied to brute force algorithms to improve their efficiency, such as pruning branches of the search tree or using heuristic techniques to guide the search.

C. Euclidean Distance

Euclidean distance is a measure of the straight-line distance between two points in a Euclidean space. It is calculated using the Pythagorean theorem, which states that the square of the hypotenuse (the side opposite the right angle) of a right triangle is equal to the sum of the squares of the other two sides. In two-dimensional space, the Euclidean distance between points (x_1, y_1) and (x_2, y_2) is given by the formula:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Euclidean distance is commonly used in various fields, including mathematics, physics, and computer science, as a measure of similarity or dissimilarity between points or objects in space. It serves as a fundamental tool in tasks such as clustering, classification, and optimization.

D. Robot Collision Avoidance (RCA)

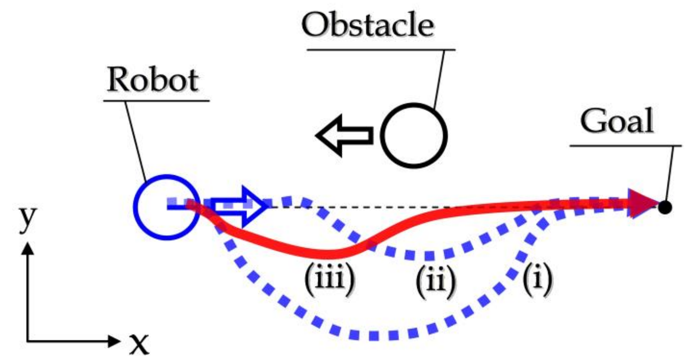

Fig 2.2 Robot Collision Avoidance
Source: https://www.intechopen.com/chapters/19455

Robot collision avoidance is a critical aspect of robotic navigation, particularly in environments where robots operate alongside humans or other objects. The primary goal of collision avoidance is to prevent robots from colliding with obstacles in their environment, thereby ensuring the safety of both the robot and its surroundings.

In dynamic environments, such as warehouses, factories, or public spaces, robots must navigate autonomously while avoiding obstacles that may appear unexpectedly. Collision avoidance algorithms continuously monitor the robot's surroundings using sensors, cameras, or other perception systems. These algorithms analyse the sensor data to detect potential obstacles and determine the robot's path to avoid collisions.

Several approaches can be used for robot collision avoidance, including reactive methods, potential field methods, and model-based planning. Reactive methods rely on simple rules or behaviours to respond to immediate obstacles in the robot's path. Potential field methods model the robot's environment as a field of attractive and repulsive forces, guiding the robot away from obstacles. Model-based planning algorithms use

predictive models of the robot's environment to plan collision-free paths.
Effective collision avoidance algorithms balance safety, efficiency, and responsiveness. They must be capable of adapting to changing environments and unpredictable obstacles while ensuring that the robot can navigate efficiently towards its goal.

E. Closest Pair of Points Problem

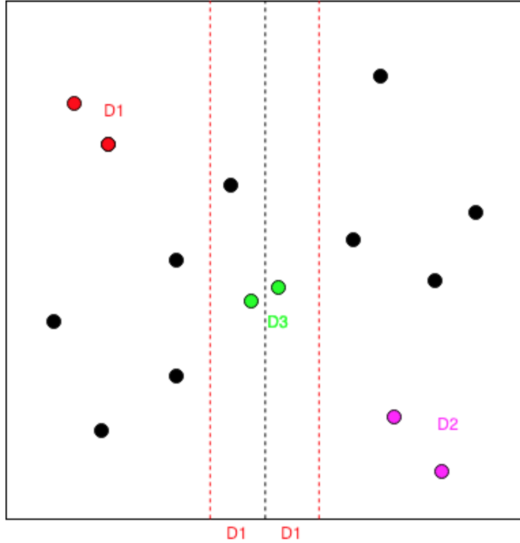

Fig 2.3 Closes Pair of Points Problem
Source: https://medium.com/@shenhuang_21425/improved-closest-pair-of-points-37e79ade474d

The closest pair of points problem is a fundamental computational problem with applications in various fields, including computational geometry, pattern recognition, and robotics. Given a set of n points in a two-dimensional space, the goal is to identify the pair of points with the smallest distance between them.
Efficiently solving the closest pair problem is essential for optimizing robotic navigation systems and ensuring safe operation in dynamic environments. In the context of robot collision avoidance, identifying the closest pairs of obstacles allows the robot to predict potential collision points and adjust its path accordingly.
Several algorithms exist for solving the closest pair problem, with the divide and conquer approach being one of the most efficient. The divide and conquer algorithm divides the problem into smaller subproblems, recursively solves each subproblem, and combines the solutions to find the overall closest pair of points.
Other algorithms for solving the closest pair problem include the brute force method, which involves comparing the distance between every pair of points and selecting the pair with the smallest distance. While brute force is simple, it is less efficient than divide and conquer for large datasets.
Efficiently solving the closest pair problem is crucial for real-time robotic applications, where quick decision-making is essential for avoiding collisions and navigating complex environments. By leveraging algorithms such as divide and conquer, robots can effectively identify and avoid potential collision points, ensuring safe and efficient navigation.

F. Time Complexity

In theoretical computer science, the time complexity is the computational complexity that describes the amount of computer time it takes to run an algorithm. Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, supposing that each elementary operation takes a fixed amount of time to perform. The asymptotic notation, can be used to measure time complexity of algorithm which are Big-O notation, Omega notation, Theta notation.

1) Big-O notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm. If $f(n)$ describes the running time of an algorithm, $f(n)$ is $O(g(n))$ if there exist a positive constant C and n_0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

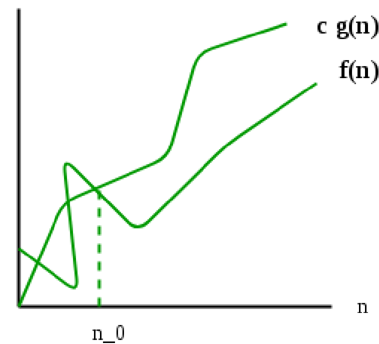

Fig 2.4 Big-O notation visualization
Source: https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/

2) Omega notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm. Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Omega(g)$, if there is a constant $c > 0$ and a natural number n0 such that $c * g(n) \leq f(n)$,
for all $n \geq n0$

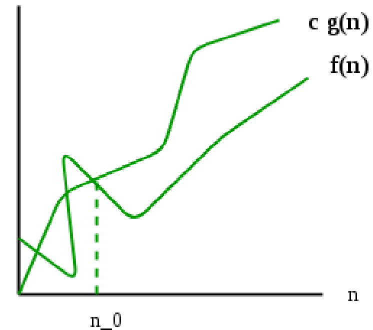

Fig 2.5 Omega notation visualization
Source: https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/

3) Theta notation (Θ-notation)

Theta notation represents the upper and the lower bound of the running time of an algorithm, it is used for analysing the average-case complexity of an algorithm. Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Theta(g)$, if there are constants $c1, c2 > 0$ and a natural number n_0 such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$, for all $n \geq n0$

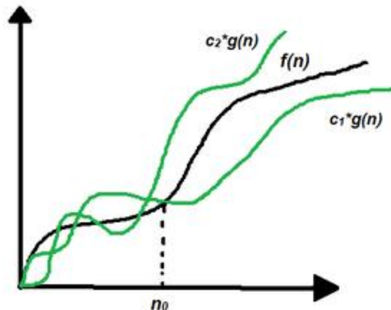

Fig 2.6 Theta notation visualization
Source: https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/

G. Computational Complexity Theory

There are considered three Types of Complexity:

1) P (Polynomial Time)

The class P consists of all decision problems that can be solved by a deterministic Turing machine in polynomial time. This means there exists an algorithm that can solve any instance of the problem with a time complexity expressed as a polynomial function of the input size, denoted as $O(n^k)$, where n is the size of the input and k is a constant. Problems in P are considered efficiently solvable because the time required to solve these problems grows at a manageable rate as the input size increases. Examples of problems in P include sorting algorithms (e.g., quicksort, mergesort), graph algorithms (e.g., shortest path), and basic arithmetic operations. The significance of class P lies in its practical applicability; algorithms that run in polynomial time are generally feasible to execute even for reasonably large inputs, making P a critical class for real-world problem-solving.

2) NP (Nondeterministic Polynomial Time)

The class NP includes all decision problems for which a given solution can be verified in polynomial time by a deterministic Turing machine. Unlike class P, where the solution itself can be found in polynomial time, NP problems may not have known efficient solving algorithms, but if a solution is provided, its correctness can be checked quickly. This implies that for an NP problem, while finding a solution might be computationally intensive, confirming that a solution is correct is relatively easy. Classic examples of NP problems include the traveling salesman problem, where the task is to determine the shortest possible route that visits a set of cities and returns to the origin, and the Boolean satisfiability problem (SAT), which involves determining if there is an assignment of truth values to variables that makes a Boolean expression true. The relationship between P and NP is central to computational complexity theory, with the unresolved question of whether $P = NP$ being one of the most profound open problems in computer science.

3) NP-Hard

NP-hard problems are those that are at least as difficult as the hardest problems in NP. Formally, a problem L is $NP-hard$ if every problem in NP can be reduced to L in polynomial time. This means that an efficient solution to an NP-hard problem would imply efficient solutions to all NP problems, making *NP-hard* problems highly significant in complexity theory. However, *NP-hard* problems do not have to be in *NP*; they may not even be decision problems. They often include optimization problems, such as the traveling salesman problem when formulated to find the shortest possible route (as opposed to merely deciding if a route shorter than a given length exists). Solving an *NP-hard* problem efficiently would have wide-ranging implications, but currently, no polynomial-time algorithms are known for *NP-hard* problems, and they are generally considered intractable for large inputs. The concept of *NP-hardness* helps in understanding the relative difficulty of computational problems and guides researchers in focusing on approximation algorithms or heuristic methods for practical solutions.

III. Proposed Algorithm

A. Distance Calculation

For calculating the distance of point one (x_1, y_1) to point two (x_2, y_2), this project will use Euclidean Distance for some purposes. One of the purposes is to gain the detailed and exact value of calculated distance. That is important because to get the closest pair of points, we need to have the precise value of distance to minimize duplication points within the result. In other words, using Euclidean Distance method is one of the optimization that is used for this algorithm.
To calculate the distance using Euclidean Distance method, given two points $p_1\,(x_1, y_1)$ and $p_2\,(x_2, y_2)$, the value of Euclidian Distance is

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Therefore, in the program we need math library to calculate the Euclidean Distance

B. Divide and Conquer (main purpose)

Divide and Conquer is the algorithm that is created for optimizing the process of searching the closest pairs of points in robot collision avoidance. This algorithm was chosen because it fits the usability of the conditions by utilizing the recursive technique for better performance. The algorithm consists of three main steps:

1) Divide:

- The first step is to divide the set of points along a vertical line that bisects the plane into left and right subsets.
- This dividing line is chosen such that it evenly splits the points, ensuring roughly equal numbers of points on each side.
- The points are sorted based on their x-coordinates, and the median point is selected as the dividing point.This division reduces the problem into two smaller subproblems, each containing a subset of the original points.

2) *Conquer:*

- Recursively, the algorithm finds the closest pair of points in each of the left and right subsets.
- If the number of points in either subset is small enough (e.g., two or three points), a brute force approach is used to find the closest pair directly.
- For larger subsets, the divide and conquer algorithm is applied recursively to each subset to find their closest pairs.

3) *Combine:*

- Once the closest pairs of points in the left and right subsets are found, determine the minimum distance δ between any pair of points that spans the dividing line.
- Check if there exist closer pairs of points that cross the dividing line. To do this, consider only those points that lie within a distance δ of the dividing line on either side. Sort these points by their y-coordinates and iterate through them to find any pairs with distances smaller than δ.
- Update δ if a closer pair is found.
- Return the pair of points with the smallest distance found among the left, right, and across-divide pairs.

These steps systematically ensures that the divide and conquer algorithm efficiently identifies the closest pair of points in the set. The combination of recursion and careful consideration of points near the dividing line allows for an optimal solution to be found.

C. *Brute Force (comparison purpose)*

The Brute Force implementation on closest pairs of points search is used to compare the Divide and Conquer algorithm implementation to the same exact problem performance. The Brute Force algorithm also provides the sample of the time execution comparison that helps the analysis by providing proper evidence. The Brute Force algorithm in this case is the exact same implementation of Brute Force algorithm on Sorting. This is because the Brute Force algorithm implementation in this problem requires the Brute Force Exhaustive behavior as the Brute Force sort does. Given the set of points $p\ (x_p, y_p)$, the Brute Force method steps:

1) *Generate All Pair Combinations:*

The brute force method involves generating all possible pair combinations of points from the given set. For n points, this results in $\frac{n(n-1)}{2}$ pairs.

2) *Calculate Distances:*

For each pair of points, calculate the Euclidean distance between them using the distance formula:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

3) *Find Minimum Distance:*

After computing distances for all pairs, identify the pair with the smallest distance. This involves iterating through all computed distances and keeping track of the minimum distance found so far. The pair of points corresponding to the minimum distance is the closest pair.

4) *Return Result:*

Once the closest pair is identified, return the pair of points along with their distance.

IV. PROGRAM IMPLEMENTATION

The program implementation consists of three main sub-program which are The Euclidean Distance implementation, The Divide and Conquer implementation. And The Brute Force implementation. Those algorithm codes can be visualized as the cut of program such as:

A. *Euclidean Distance Implementation*

```
def distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

Fig 4.1 The Euclidean Distance Code
Source: Author Creation

B. *Brute Force Implementation*

```
def brute_force_closest_pair(points):
    n = len(points)
    min_distance = float('inf')
    closest_pair = None

    for i in range(n):
        for j in range(i+1, n):
            dist = distance(points[i], points[j])
            if dist < min_distance:
                min_distance = dist
                closest_pair = (points[i], points[j])

    return closest_pair
```

Fig 4.2 The Brute Force Code
Source: Author Creation

C. *Divide and Conquer Implementation*

```python
def closest_pair_in_strip(strip, d):
    min_distance = d
    closest_pair = None

    strip.sort(key=lambda x: x[1])

    for i in range(len(strip)):
        for j in range(i+1, len(strip)):
            if strip[j][1] - strip[i][1] < min_distance:
                dist = distance(strip[i], strip[j])
                if dist < min_distance:
                    min_distance = dist
                    closest_pair = (strip[i], strip[j])
            else:
                break

    return closest_pair

def closest_pair_recursive(points):
    n = len(points)

    if n <= 3:
        return brute_force_closest_pair(points)

    mid = n // 2
    mid_point = points[mid]

    left_closest = closest_pair_recursive(points[:mid])
    right_closest = closest_pair_recursive(points[mid:])

    d = min(distance(left_closest[0], left_closest[1]), distance(right_closest[0], right_closest[1]))

    strip = [point for point in points if abs(point[0] - mid_point[0]) < d]

    strip_closest = closest_pair_in_strip(strip, d)

    if strip_closest:
        return min(left_closest, right_closest, strip_closest, key=lambda x: distance(x[0], x[1]))
    else:
        return min(left_closest, right_closest, key=lambda x: distance(x[0], x[1]))
```

Fig 4.3 The Divide and Conquer Code
Source: Author Creation

V. Result and Analysis

A. Result

The result of the program when executed is measured three times and each attempt differs to its number of points, starting from 10 and incrementing by multiplying by 10. The result of three attempts such as

1) Attempt 1 (n=10)

```
Number of Points:  10
Closest pair (dnc): ((78.9466346195119, 56.06694157287345), (82.77088466649604, 56.97220165878086))
Duration: 0.22912025451660156 ms
Closest pair (bf): ((78.9466346195119, 56.06694157287345), (82.77088466649604, 56.97220165878086))
Duration: 0.07605552673339844 ms
```

Fig 5.1 Result 1 (n=10)
Source: Athor Creation

2) Attempt 2 (n=100)

```
→ dnc /bin/python3 /home/albert/Documents/kuliah/stima/main.py
Number of Points:  100
Closest pair (dnc): ((38.03267803595883, 48.12896306835638), (37.57782259205983, 48.917517578004244))
Duration: 1.3310909271240234 ms
Closest pair (bf): ((37.57782259205983, 48.917517578004244), (38.03267803595883, 48.12896306835638))
Duration: 4.418134689331055 ms
```

Fig 5.2 Result 2 (n=100)
Source: Athor Creation

3) Attempt 3 (n=1000)

```
→ dnc /bin/python3 /home/albert/Documents/kuliah/stima/main.py
Number of Points:  1000
Closest pair (dnc): ((84.73115311388817, 75.11647943893797), (84.8449339737941, 75.11984872694808))
Duration: 27.578353881835938 ms
Closest pair (bf): ((84.73115311388817, 75.11647943893797), (84.8449339737941, 75.11984872694808))
Duration: 584.5470428466797 ms
```

Fig 5.3 Result 3 (n=1000)
Source: Athor Creation

Using the result data, we can map the result to be a table as below

Number of Points (n)	Divide and Conquer Time Execution (ms)	Brute Force Time Execution (ms)
10	0.229	0.076
100	1.331	4.418
1000	27.578	584.587

Table 5.1 Test Execution Result Table

B. Analysis

In robot collision avoidance, efficiently identifying the closest pair of points among detected obstacles is critical for safe and effective navigation. Two primary algorithms for solving the closest pair of points problem are the brute force method and the divide and conquer method. This analysis provides a detailed comparison of these two algorithms, focusing on their implementation, time complexity, and computational complexity, with specific attention to their application in robot collision avoidance.

1) Brute Force Algorithm

The brute force algorithm for finding the closest pair of points involves generating all possible pairs of points from the given set, calculating the Euclidean distance for each pair, and then identifying the pair with the smallest distance. This method involves iterating through all possible pairs, which results in $\frac{n(n-1)}{n}$ comparisons for n points. The brute force algorithm has a time complexity of $O\ (n^2)$ due to this exhaustive search through all pairs. This quadratic growth makes the brute force method inefficient for large datasets, although it is straightforward and easy to implement. Despite its simplicity, the computational expense becomes significant as the number of points increases, making it less practical for real-world applications involving large numbers of points. It is important to note that the closest pair of points problem falls into the class P, as it can be solved in polynomial time by the brute force metho

2) Divide and Conquer Algorithm

The divide and conquer algorithm for finding the closest pair of points involves dividing the set of points into two subsets along a median vertical line, recursively finding the closest pair of points in each subset, and then combining the results to identify the overall closest pair. The combination step involves calculating the minimum distance from the left and right subsets, identifying points within this minimum distance near the dividing line, and checking these points for closer pairs. The divide and conquer algorithm has a time complexity of $O(\ nlog(n)\ )$,which is more efficient than the brute force method for large datasets. This method's space complexity is $O(n)$ due to the space needed for recursive stack calls and temporary arrays used during merging. While more complex to implement, the divide and conquer algorithm's efficiency for large datasets makes it highly suitable for applications requiring quick and accurate computations, such as robot collision avoidance. Similar to the brute force method, the closest pair of points problem remains in class P when using the divide and conquer approach.

3) *Comparative Analysis Using Empirical Data*

The empirical data provided illustrates the performance of both algorithms across different numbers of points. For $n = 10$, the brute force method has an execution time of $0.076\ ms$, whereas the divide and conquer method takes $0.229\ ms$. At this small scale, the brute force method performs slightly better due to its lower overhead.

As the number of points increases to $n = 100$, the brute force method's execution time increases to $4.418\ ms$, while the divide and conquer method's execution time is significantly lower at $1.331\ ms$. This demonstrates the quadratic growth of the brute force method, making the divide and conquer method more efficient for larger datasets.

For $n = 1000$, the brute force method's execution time escalates to $584.587\ ms$, while the divide and conquer method's execution time remains relatively low at $27.578\ ms$. This vast difference highlights the superior scalability of the divide and conquer method, as it handles large datasets much more efficiently.

VI. Conclusion

The brute force method is simple to implement and works well for small datasets, but its $O(n^2)$ time complexity makes it impractical for large-scale problems due to the exponential growth in computational time. In contrast, the divide and conquer algorithm, with its $O(\ n\log(n)\ )$ time complexity, provides a much more efficient solution for large datasets, making it highly suitable for real-time applications like robot collision avoidance where quick and efficient computations are critical.

VII. Review

Overall, the paper provides a thorough and well-structured analysis of the brute force and divide and conquer algorithms for finding the closest pair of points in the context of robot collision avoidance. The theoretical explanations are detailed and clear, and the empirical analysis robustly supports the claims about each algorithm's performance. The paper effectively bridges the gap between theoretical computational geometry and practical robotic applications, making a strong case for the use of the divide and conquer algorithm in real-world scenarios. The review highlights the paper's clarity, thoroughness, and practical relevance, making it a valuable contribution to the field of robotic collision avoidance.

In addition, it also must be noted that this paper only covers the divide and conquer implementation on robot collision avoidance by implementing it in the closest pair of points of each object search. The rest doesn't get covered enough in this paper. For example, avoidance program, which consists of how robot reacts to avoid an obstacle, and the movements of avoidance in simulation are not included in this paper. This is the picture of how robot avoids in simulation using Gazebo Simulator:

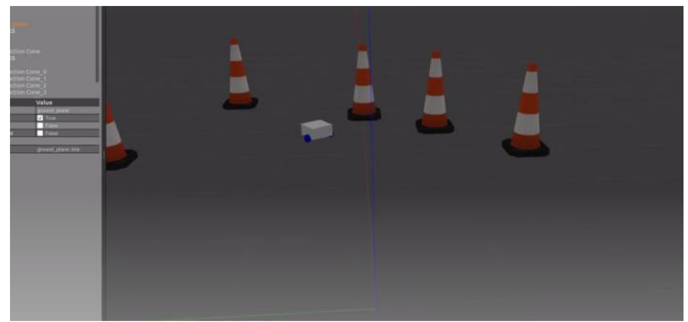

Fig 7.1 Picture of Robot Avoiding Obstacles in Gazebo Simulator
Source: Author Creation

That is the picture of the simulation, the robot successfully avoid the obstacles which are cones in this case. The simulation is also recorded and saved as a video in link below. The video also covers the explanation of the algorithms and the demonstration of the algorithm using robot simulator.

Video Link at YouTube

Here is the link that consists of the concise explanation of algorithms and the demonstration using Gazebo simulator:
https://www.youtube.com/watch?v=-9z4Kl2WKGw

Acknowledgment

. First of all, the Author would like to thank our lecturers in Class 3, Mr. Rila Mandala and Mr. Monterico Adrian for giving us numerous lesson. Not to forget, the Author would also like to all the teachers in ITB, including Mr. Rinaldi Munir, for striving to share knowledge to us while knowing it is not easy to handle many students at the same time. Lastly, the Author would like to tell that The lesson that you taught was the reason the Author can accomplish this paper

References

[1] https://www.geeksforgeeks.org/introduction-to-divide-and-conquer-algorithm-data-structure-and-algorithm-tutorials/, accessed 10 June 2024, 09.25 P.M.

[2] https://medium.com/@shenhuang_21425/improved-closest-pair-of-points-37e79ade474d, accessed 10 June 2024, 10.00 P.M.

[3] https://botpenguin.com/glossary/computational-complexity-theory , accessed 10 June 2024, 10.20 P.M.

[4] https://www.geeksforgeeks.org/types-of-asymptotic-notations-in-complexity-analysis-of-algorithms/ . accessed 10 June 2024 10.30 P.M.

[5] https://www.intechopen.com/chapters/19455, accessed 11 June 2024 09.00 A.M.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

Albert Ghazaly (13522150)