HOMEWORK REPORT

ME5413 AUTONOMOUS MOBILE ROBOTICS

GROUP 28

Anse Min (A0285307B)
Huang Yiming(A0285028B)
Niu Dixiao(A0284913X)
(Master of Science (Robotics), NUS)
(Master of Science (Mechanical Engineering), NUS)

A REPORT SUBMITTED FOR THE ASSIGNMENT OF ME5413
DEPARTMENT OF MECHANICAL ENGINEERING
NATIONAL UNIVERSITY OF SINGAPORE

Supervisor:

Prof. Marcelo H Ang Jr

3/4/2024

1 Task 1: Global Planning

The A* (A-Star) algorithm is a direct search method that efficiently solves the shortest path in a static road network. It is also effective for solving many search problems. Essentially, A* is an optimization of the breadth-first search (BFS) algorithm. Beginning from the starting point, the algorithm first explores the neighbouring points around it, and then moves on to the points neighbouring the already explored points. This process continues until it reaches the end point.

1.1 Implementation of A-Star Algorithm

The A* algorithm can be roughly divided into 5 steps: initialisation, neighbourhood point exploration, costing and updating, checking obstacles and boundaries, and path backtracking.

At the beginning of the algorithm, the cost (gscore) of the starting point is set to 0, and the Euclidean distance from the starting point to the end point is calculated as the fscore (heuristic score) of the starting point. The starting point is then added to the priority queue (oheap). The algorithm then explores all possible neighbour points for the current point. The algorithm considers neighbouring points that are directly adjacent (up, down, left, right) and diagonally adjacent, with corresponding move costs of 0.2 and 0.282, respectively. It calculates and updates the cost for each neighbouring point, as well as the cost of moving from the starting point through the current point to that neighbouring point (tentative_g_score). If the cost is lower than the previously recorded cost or if the neighbour point has not been explored yet (not in the priority queue), update the path cost and total score (gscore and fscore) for this neighbour point and add it to the priority queue. During the exploration process, the algorithm checks if these points are out of bounds or obstacles (represented by 255 in the grid). If they are, these points are ignored and not added to the priority queue. The algorithm repeats the above process, selecting the point with the lowest fscore from the priority queue as the current point to be explored. This continues until the priority queue is empty or a target point is found. When the goal point is found, the algorithm constructs a path from the start point to the end point by backtracking through the code_from dictionary. The pseudo-code for the algorithm is shown in the appendix (Algorithm 1). The planned path from Start to Food was shown in Fig.1

1.2 Try Different Heuristic Functions

The A* algorithm employs a heuristic function, denoted as $h(x)$, to estimate the minimum cost or distance from any node n to the goal node. The function prioritises the exploration of paths that appear to be closest to the goal, thereby improving the search efficiency. In this paper, we use the Euclidean distance as the heuristic function(Eq. 1), which, compared to the common Manhattan distance(Eq. 3, allows the target to move in any direction without being limited to just moving from one grid to a neighbouring grid. We also use Chebyshev Distance as comparsion. The Chebyshev distance heuristic is used in a grid where movement can be horizontal, vertical, and diagonal. It calculates the maximum of the absolute differences of their Cartesian coordinates, effectively measuring the minimum number of steps needed considering diagonal movement as well.

$$d_{\text{euclidean}}(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{1}$$

$$d_{\text{manhattan}}(p, q) = |p_x - q_x| + |p_y - q_y| \tag{2}$$

$$d_{\text{chebyshev}}(p, q) = \max(|p_x - q_x|, |p_y - q_y|) \tag{3}$$

We analysed the influence of various heuristic functions on different tasks by changing them and comparing the distances from the starting point to the snack store, movie theatre, and food points. The result was shown in Table.

From the results, it can be seen that the results of the three heuristic functions are similar at most moments, and the errors are basically within 1%. However, in the path planning of start-movie, the error of Manhattan distance and other heuristic functions is more than 10%. It can be assumed that the three heuristics are similar in most of the moments, but the Manhattan distance heuristic may be less effective in some cases.

Table 1: ADE and FDE for Different Model

Heuristic Function	**start-snacks**	**start-store**	**start-movie**	**start-food**
Euclidean	144.18	164.04	182.61	235.37
Manhattan	143.95	163.52	210.04	232.89
Chebyshev	144.01	168.41	180.25	233.03

1.3 Switch Start and Goal Positions

The A* algorithm's calculated distance between points is significantly influenced by adjusting the start and end locations, leading to observable variations in the results. This variability is primarily due to the heuristic nature of the algorithm, which uses a predefined method (such as Euclidean, Manhattan or Chebyshev distance) to estimate the cost from each node to the destination. Altering the start and end points changes the heuristic landscape, which affects cost estimations and, consequently, the algorithm's chosen path.

For instance, if the heuristic employs Euclidean distances, it reduces straight-line distances by prioritising paths that seem to lead straight to the goal. If the start and end locations are altered, the 'directness' of the paths may also change, resulting in a distinct set of optimal paths. We get the results in Table 2 by swapping start and end.

Table 2: Distances when Changing Start and End Location

From/To	**start**	**snacks**	**store**	**movie**	**food**
start	0.0	154.9	167.8	182.2	237.5
snacks	144.2	0.0	116.7	106.2	139.8
store	164.0	122.8	0.0	238.1	123.5
movie	182.6	125.4	255.5	0.0	117.4
food	235.7	139.6	115.5	202.1	0.0

2 Task 2: Travelling Shopper Problem

2.1 Brute Force Algorithms

The first method implemented is the brute force algorithm. Brute force algorithms solve problems by methodically exploring all possible solutions to identify the one that meets the problem's criteria. This approach, characterized by its simplicity and the absence of shortcuts, involves checking every potential candidate to ascertain whether it constitutes the solution to the problem. This is mathematically represented by:

$$D_r = \sum_{i=1}^{N-1} d(p_i, p_{i+1}) + d(p_N, s) \tag{4}$$

where, for each route r, p_i and p_{i+1} are consecutive locations in the route, $d(x, y)$ denotes the distance between locations x and y, N is the number of locations, and s is the start location. The shortest route is the one that minimizes D_r.

Algorithm 2 shows the implementation of this equation. The application of a brute force algorithm involved several steps, beginning with the generation of all permutations of the locations to be visited. This process, facilitated by Python's **'itertools.permutations'** function, created every possible sequence in which the locations could be visited, each sequence representing a distinct route. The algorithm then proceeded to calculate the total distance for each route by summing up the distances between successive locations, as specified in a predefined dictionary of distances. This exhaustive evaluation allowed for the identification of the shortest possible route among all the permutations.By maintaining a record of the shortest distance encountered and its associated route, the algorithm continuously updated these values whenever a shorter route was discovered. This iterative process ultimately yielded the most efficient path for visiting all specified locations and returning to the starting point, thereby minimizing

the overall distance traveled. The brute force approach's main advantage lies in its guarantee to find the optimal solution by thoroughly examining all possible routes.

The image (Figure 3) depicts a floor plan with a highlighted route in red showcasing the result of the brute force algorithm applied. The algorithm has evidently computed an optimal path that visits all specified locations—labeled 'start', 'store', 'food','movie', and 'start' respectively, ensuring that the total distance traveled is minimized. The route is delineated by a red line that seamlessly connects these points, avoiding unnecessary detours and backtracking, indicative of the exhaustive nature of the brute force approach which has examined all possible permutations to find the shortest possible loop connecting all the points of interest.

2.2 Nearest Neighbor Algorithm

Second method implemented is the nearest neighbors algorithm. The nearest neighbor algorithm is a straightforward, heuristic method that can effectively tackle the Traveling Salesman Problem (TSP) by iteratively visiting the closest unvisited location until all destinations have been reached. It is particularly well-suited for this problem, where the goal is to start from the VivoCity level 2 escalator and visit four distinct locations—'snacks', 'store', 'movie', and 'food'—before returning to the starting point. Given the modest number of locations, the nearest neighbor algorithm is advantageous due to its simplicity and low computational overhead. This approach leverages the distance table generated from Task 1 to efficiently determine an optimal route that minimizes travel time, making it an excellent fit for this scenario where quick, real-world decision-making is paramount. The mathematical formulation for selecting the next location n using the nearest neighbor heuristic from the current location c is given by:

$$n = \arg\min_{l \in U} d(c, l) \tag{5}$$

where U is the set of unvisited locations and $d(c, l)$ is the distance between the current location c and a location l in U.

The algorithm was implemented to systematically identify the shortest onward path at each step (Algorithm 3). Starting from the Level 2 escalator, the algorithm iteratively selected the closest unvisited location from the remaining destinations—'snacks', 'store', 'movie', and 'food'. By calculating the distance to each unvisited location using a predefined distance table, the algorithm updated the current position to the nearest location and recorded this journey in a route list, while also tallying the total distance traveled. This process continued until all locations were visited, at which point the algorithm concluded the tour by returning to the starting point, thereby forming a complete and efficient loop. Subsequently, the AggregatePaths function was applied to this route, utilizing an A* search algorithm with a Euclidean distance heuristic to plot an actual navigable path on the floor plan, accounting for spatial constraints and optimizing the walking path between the shops. This approach provided a quick and practical solution for the shopper to minimize their travel distance while visiting all the specified points of interest.

The Figure 2 presents a clear visualization of the path determined by the nearest neighbor algorithm applied to the Traveling Shopper Problem at VivoCity. Starting from the designated 'start' location, the path, marked in red, shows the algorithm's journey as it selects the nearest next location from the current one, successively visiting 'snacks', 'store', 'movie', and 'food', in the order that minimizes the distance traveled between each step. The algorithm appears to take a logical and efficient route that avoids backtracking and unnecessary detours, reflecting the nearest neighbor heuristic's characteristic of local optimization. The route eventually returns to the starting point, completing the circuit that the shopper would traverse to visit all four locations in a manner that likely reflects the shortest immediate distances, though not necessarily the shortest overall path when considering the global optimum.

2.3 Comparison of the Algorithms

Table 3 and Figure 4 illustrate a nuanced evaluation of the brute force and nearest neighbor algorithms when solving the Traveling Salesman Problem. Despite the brute force approach guaranteeing an optimal solution, as evidenced by the slightly shorter total distance of 660 units compared to the nearest neighbor's 685.1 units, this comes at a cost of increased computational resources. Notably, the brute force

Table 3: Performance Comparison of Algorithms

Algorithm	**Total Distance (units)**	**Execution Time (seconds)**	**Memory Usage (MB)**
Brute Force	660.0	0.00016	0.011
Nearest Neighbor	685.1	0.00011	0.012

method exhibited a longer execution time at 0.00016 seconds, suggesting a more intensive computational process, likely due to evaluating all possible permutations of the route. However, it used slightly less memory, as indicated by the memory usage metric of 0.011 MB versus the nearest neighbor's 0.012 MB. This disparity in memory usage, albeit minor, could be attributed to the brute force method's storage of a large number of permutations for comparison. In contrast, the nearest neighbor algorithm, while slightly less efficient in terms of the total distance due to its heuristic nature, performed faster in execution time, possibly because it makes a series of local decisions without backtracking, thereby reducing the overall computational burden. These findings highlight the trade-off between the computational efficiency and solution optimality when choosing between these two algorithms for practical applications.

2.4 Limitation of the Algorithms

The limitations of the brute force algorithm are primarily tied to its scalability and computational cost. As the number of locations increases, the number of permutations that need to be evaluated grows factorially, which can quickly become infeasible even with a relatively small number of points. This approach is computationally expensive because it requires calculating and comparing the total distance of each possible route to find the optimal path. In effect, the algorithm's execution time skyrockets as more locations are added to the problem, making it impractical for larger datasets where swift computation is essential.The brute force method, while guaranteed to find the most efficient route, does so at the expense of time and processing power, making it impractical for larger datasets.

On the other hand, the nearest neighbor algorithm is much more computationally efficient and scales better to larger problems. However, this efficiency comes at the cost of potentially missing the most optimal route. The algorithm works by selecting the closest next location at each step, which can lead to suboptimal paths due to a local optimization strategy that doesn't consider the global context of the route.

3 Task 3 (Bonus):

3.1 PID Controller

PID (Proportional-Integral-Derivative) controllers are widely adopted for their simplicity and effectiveness in regulating various processes. A PID controller calculates the error between a desired setpoint and the current process variable, then applies a correction based on proportional, integral, and derivative terms. These terms enable the controller to adjust the control input not only based on the current error but also considering the error's history and rate of change. This results in a versatile control strategy that can be tailored to achieve a balance between response speed, stability, and steady-state accuracy across a wide range of applications.

3.2 Stanley Controller

The Stanley controller is a path tracking method distinguished by simplicity and robust performance, particularly in autonomous vehicle navigation.it combines the cross-track error and the vehicle's heading error relative to the path to compute steering commands. This approach allows for effective correction of both the vehicle's position and orientation which ensuring converges swiftly and smoothly onto the desired path. The non-linear nature and straightforward implementation make it well-suited for navigating complex trajectories and handling the dynamic environments encountered in robotics and autonomous vehicle systems.

3.3 Adjusting Parameters

The code has provided only a dumb PID controller and a weird Stanley controller for the steering which the running result is really bad shown in Fig.5. To achieve a better result, parameters of the PID controller and Stanley controller need to be modify where how the parameters affect the system are:

1. Proportional (P): Produces an output that is proportional to the current error. Increasing the P value will generally increase the system's responsiveness, but too high a value can cause overshooting and oscillations around the target path.

2. Integral (I): Integrating previous error over time. Increasing the I value helps eliminate the steady-state error, but too much can lead to overshooting and longer settling times.

3. Derivative (D): Predicts the future trend of the error based on current rate of change. Increasing the D value can help dampen the system, reducing overshooting and improving stability while too high can make the system less responsive and more susceptible to noise.

4. Stanley Gain (K): Increasing K makes the controller more aggressive in correcting for cross-track error, potentially improving tracking performance in tight curves or complex paths. However, setting K too high can result in overly aggressive steering actions, leading to instability.

The way to find a set of proper parameters is to testing the performance using different value. There's no a standard of that and we have to modify them according to the performance and it's a tedious work. After repeating the modification, we found a relatively good result by setting $P = 0.3$, $I = 0.005$, $D = 0.015$ and $K = 1.0$.

3.4 Evaluation

Fig.7 shows the comparison of the tracking path of both sets. As can be seen that original path doesn't follow the '8' size well and it even turned around when the heading angle suddenly changed which the change seems inevitable but to adding derivative gain to avoid it to turn. The improved result shows the path can follow '8' size but due to the sensing error that it doesn't appear to be perfect while inside rviz the tracking follow really well (Fig.6).

The accuracy of adjusted parameters is shown in Fig.8. That the vehicle follows almost perfectly in the straight lane and the error appeared to increase when turning and reached the peak at the curved center for both position and speed, which can be seen the errors reduce apparently compared with the default one. As for the heading, it remained small degrees most of the time, as for the impulse, it's due to the angle calculation principle which can be eliminated, but through setting proper derivative gain to avoid turnaround.

Additionally, the '8' size is changed to test the controller's performance that A axis and B axis were set to be 9m and 4m respectively. That when the tracking object is changing, the previous parameters could not work as well as what it did in the last occasion, which means the parameters still have to adjust to adapt to the new object. By making some small modifications of the parameters did we get a relatively good result which is shown in Fig.9. The result is similar to the above occasion that the error largely increase during the curve. However, the errors were all in an acceptable range that the vehicle can track the path well.

3.5 Additional Announcement

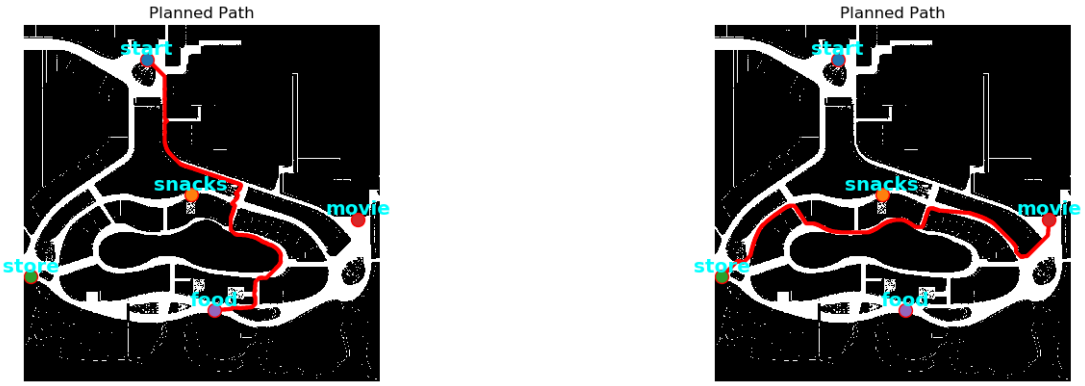
I try to implement MPC to replace the PID, however, I couldn't make the algorithm in c++ running successfully till the due date. It's quit a pity that it still not works and then we come back to use the results from the previous experiments by using the given PID controller.

Thus, there won't be a GitHub repository for the bonus task and the results are then shown in this section of the report.

APPENDICES

Algorithm 1 A* Search Algorithm using Euclidean Distance Heuristic

```
function CalculatePathLength(path)
    return sum of distances between consecutive points in path
end function
function EuclideanHeuristic(point1, point2)
    return Euclidean distance between point1 and point2
end function
function AStarSearch(start, goal, grid)
    Initialize openSet with start
    while openSet is not empty do
        current ← node in openSet with lowest fScore
        if current == goal then
            return ReconstructPath(cameFrom, current)
        end if
        Remove current from openSet
        for each neighbor of current do
            if tentative_gScore < gScore[neighbor] then
                Update cameFrom and gScore for neighbor
                if neighbor not in openSet then
                    Add neighbor to openSet
                end if
            end if
        end for
    end while
end function
```



(a) Planned Path from Start to Food

(b) Planned Path from Store to Movie

Figure 1: Planned Path

Algorithm 2 Brute Force Algorithm for Travelling Salesman Problem

```
function CalculateRouteDistance(route)
    return sum of distances between consecutive locations in route using the distances dictionary
end function
function BruteForceTSP(start, locations)
    shortestDistance ← ∞
    optimalRoute ← None
    for each permutation perm of locations do
        route ← append start to the beginning and end of perm
        currentDistance ← CalculateRouteDistance(route)
        if currentDistance < shortestDistance then
            shortestDistance ← currentDistance
            optimalRoute ← route
        end if
    end for
    return optimalRoute, shortestDistance
end function
function AggregatePaths(optimalRoute)
    allPaths ← []
    for i from 0 to length of optimalRoute - 2 do
        startLocation ← optimalRoute[i]
        nextLocation ← optimalRoute[i + 1]
        startCoords ← coordinates of startLocation
        goalCoords ← coordinates of nextLocation
        path ← AStarSearchEuclidean(startCoords, goalCoords, gridMapImg)
        append path to allPaths
    end for
    return allPaths
end function
```

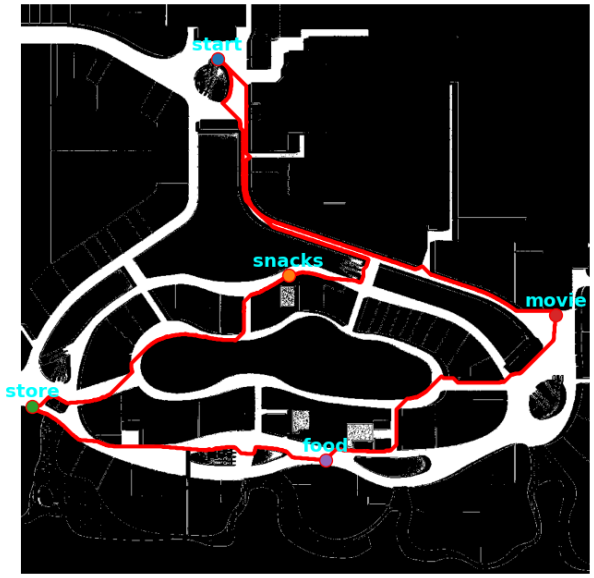


Figure 2: Optimal path calculated by the nearest neighbor algorithm.

Algorithm 3 Nearest Neighbor with Path Aggregation

```
function NearestNeighbor(start, locations, distances)
    unvisited ← locations
    currentLocation ← start
    route ← list containing start
    totalDistance ← 0
    while unvisited is not empty do
        nearest ← null
        nearestDistance ← ∞
        for all loc in unvisited do
            if distances[(currentLocation, loc)] < nearestDistance then
                nearest ← loc
                nearestDistance ← distances[(currentLocation, loc)]
            end if
        end for
        if nearest is null then
            break
        end if
        remove nearest from unvisited
        append nearest to route
        totalDistance ← totalDistance + nearestDistance
        currentLocation ← nearest
    end while
    totalDistance ← totalDistance + distances[(currentLocation, start)]
    append start to route
    return route, totalDistance
end function
function AggregatePaths(route, locations, astar_search_euclidean, grid_map_img)
    allPaths ← empty list
    for i ← 0 to length of route − 2 do
        startLocation ← route[i]
        nextLocation ← route[i + 1]
        startCoords ← convert locations[startLocation] to tuple
        goalCoords ← convert locations[nextLocation] to tuple
        path ← ASTAR_SEARCH_EUCLIDEAN(startCoords, goalCoords, grid_map_img)
        append path to allPaths
    end for
    return allPaths
end function
```

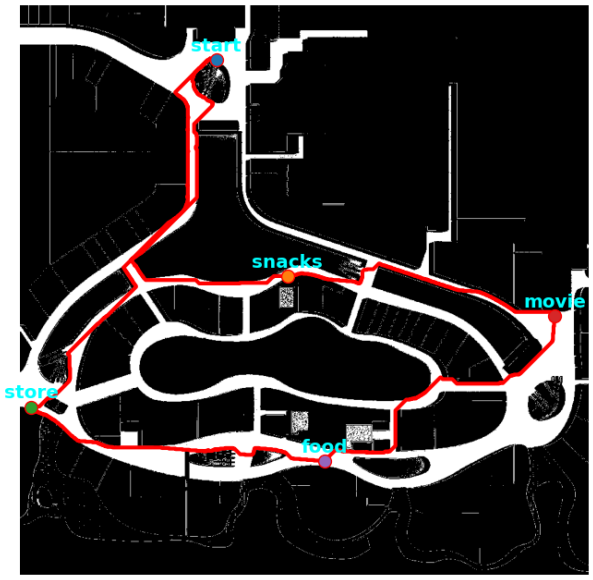


Figure 3: Optimal path calculated by the brute force algorithm.

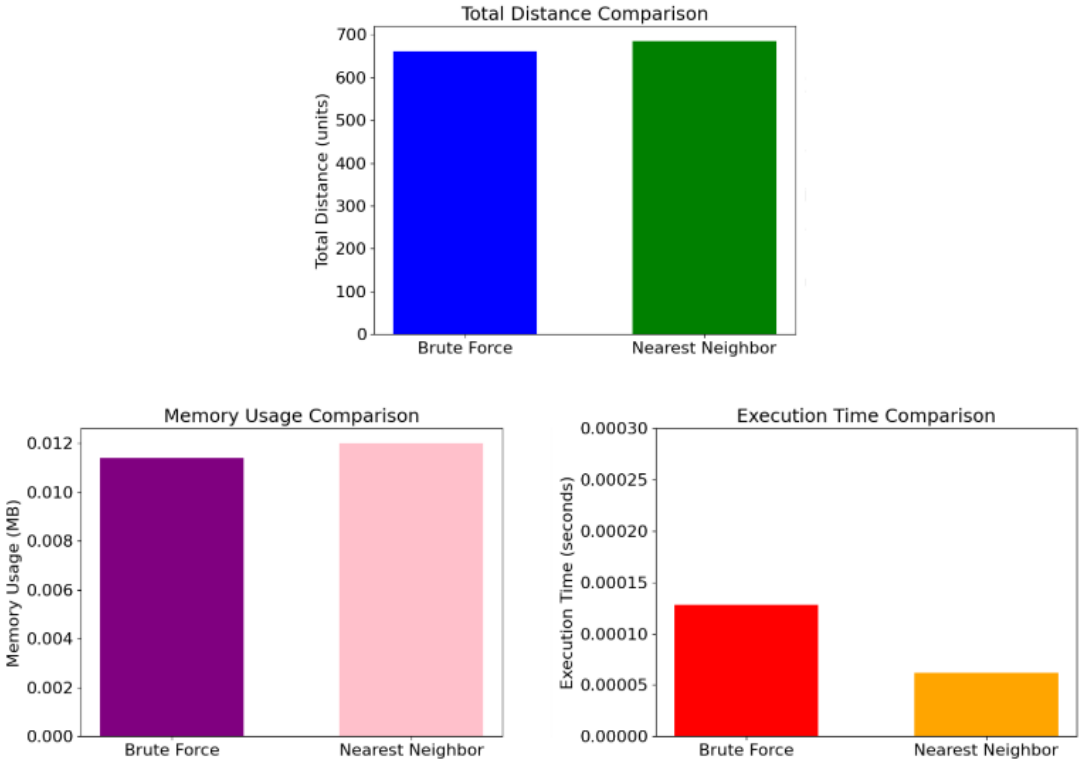
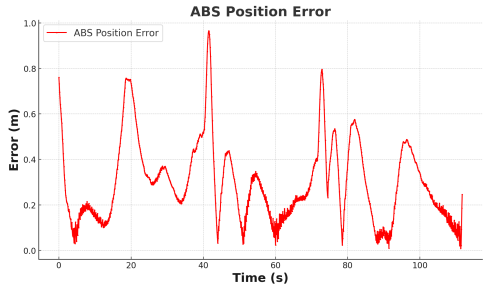
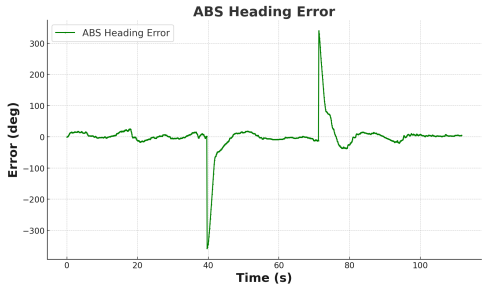


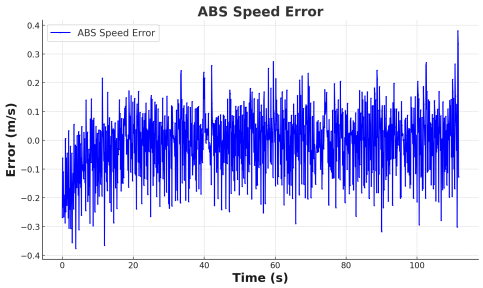
Figure 4: Comparing the two algorithms with various metrics



(a) Position error



(b) Heading error



(c) Speed error

Figure 5: Accuracy using default parameters

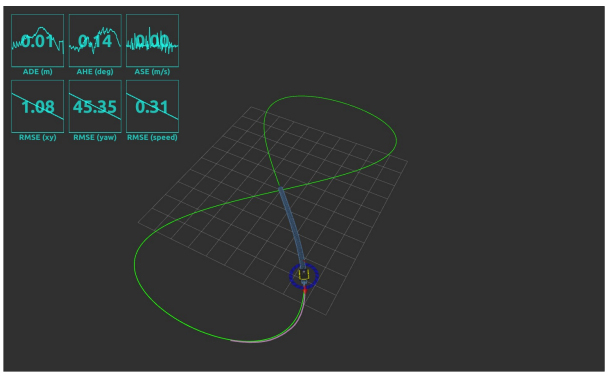
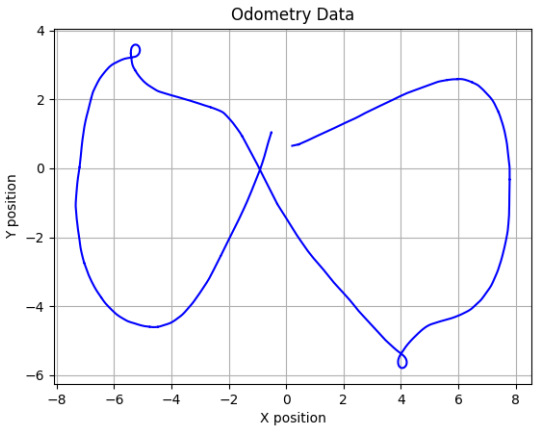
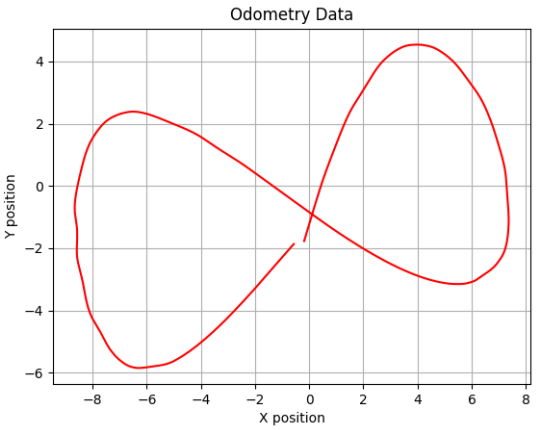


Figure 6: Screen shot of algorithm running

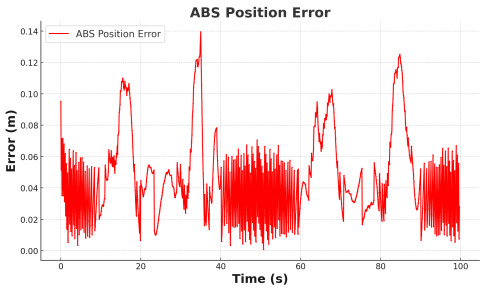


(a) Odom of default parameters

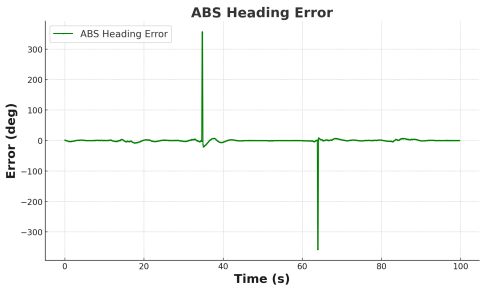


(b) Odom of adjusted parameters

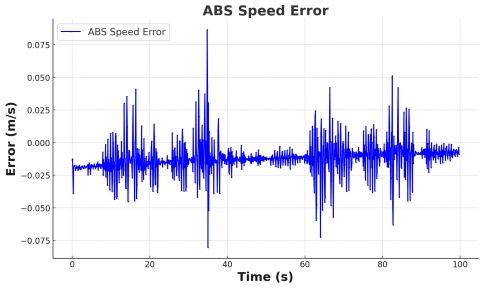
Figure 7: Comparison of odom



(a) Position error

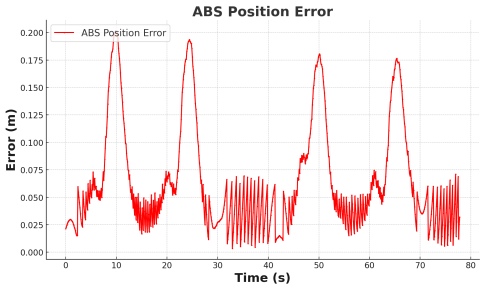


(b) Heading error

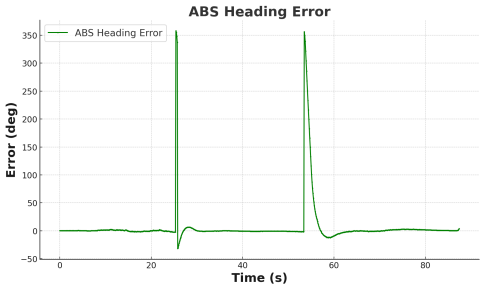


(c) Speed error

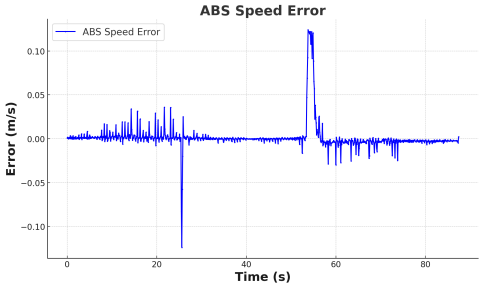
Figure 8: Accuracy using default parameters



(a) Position error



(b) Heading error



(c) Speed error

Figure 9: Accuracy using default parameters