ME5413 Autonomous Mobile Robot

Homework 3 Planning

Group 4

Full Name	Student ID
LHW	1
MYC	1
Simon Kenneth	1

Master of Science in Robotics

COLLEGE OF DESIGN AND ENGINEERING

Apr 4th, 2024

1 Task1

In this task, the shortest path between each pair of locations needs to be found. This assignment inplemented various algorithms such as A*, Dijkstra, Greedy Best-First Search, and RRT*. Subsequently, comparison experiments have been conducted on search strategy, heuristic function, heuristic ratio, tie-breaking strategy, etc.

1.1 Methods

1.1.1 Dijkstra Algorithm

Dijkstra's Algorithm is a widely used algorithm for finding the shortest path between nodes in a graph. The algorithm works by repeatedly selecting the node with the smallest distance $g(x)$ from the starting node and updating the distances of its neighboring nodes, until the shortest path to the destination node is found.

Dijkstra's Algorithm is guaranteed to find the shortest path in a weighted graph with non-negative edge weights. It has a time complexity of $O((V + E)logV)$, where V is the number of nodes, and E is the number of edges. The process of the algorithm is illustrated in Alg. 1.

1.1.2 Greedy Best First Search Algorithm

The greedy best-first search algorithm is an informed search algorithm used for the path finding tasks. It works by expanding the node that is closest to the goal node at each step, based on a heuristic function $h(x)$ which estimates the cost or distance from a given node to the goal node. The algorithm maintains an open list of nodes that need to be explored, and at each step, it selects the node from the open list that has the smallest heuristic value.

The greedy best-first search algorithm always chooses the node that appears to be closest to the goal, but it does not consider the actual cost of reaching the node from the start node, which can sometimes lead to suboptimal solutions. Meanwhile, the performance of the greedy best-first search algorithm depends heavily on the quality of the heuristic function. A good heuristic function should be admissible (never overestimate the actual cost to the goal) and should provide tight estimates of the cost to the goal for efficient search.The process of the algorithm is illustrated in Alg. 3.

1.1.3 A* Algorithm

A* algorithm is a combination of both Dijkstra and Greedy BFS algorithm. It employs a "heuristic estimate" that gives an estimation of the best route that goes towards target node. It visits the nodes in order of the weighted sum of cost and heuristic estimation $f(x) = g(x) + w * h(x)$. The core to its success is that it combines the pieces of information that Dijkstra's algorithm uses and information that Best-First Search uses[1]. The process of the algorithm is illustrated in Alg. 2.

1.1.4 RRT*

The RRT* algorithm is an efficient path planning method that explores paths from a start to a goal point by randomly sampling in the configuration space and constructing a tree structure. It optimizes path quality through best parent selection and rewiring mechanisms, achieving incremental optimality. While RRT* excels in enhancing path efficiency and quality, there's a trade-off between performance and path quality, particularly as the number of nodes increases, which can affect convergence speed [2]. The process of the algorithm is illustrated in Alg. 4 and the results are shown in Table 7

1.2 Improvments

1.2.1 Heuristic Function

In this experiment, four heuristic functions were tried: Manhattan, Euclidean, Chebyshev, and Analytic for the grid map. The corresponding formulas are shown as Equation 1, 2, 3 and 4.

1.2.2 Heuristic Ratio

The heuristic ratio is a key factor in heuristic search algorithms, balancing the heuristic's guidance towards the goal with the actual cost of the path taken. It fine-tunes the algorithm, optimizing search efficiency and ensuring a balance between exploring new paths and exploiting known ones. This balance makes the algorithm adaptable to various problems, reducing the search space and computational resources needed, thus playing a crucial role in the algorithm's performance and effectiveness.

1.2.3 Tie-breaking Strategy

In graph-searching algorithms, particularly within the realm of heuristic search and optimization, tie-breaking strategy serves as a deterministic arbitrage to resolve equivalence in evaluative metrics across competing nodes or paths. This strategy is indispensable for ensuring algorithmic consistency and optimality, especially in scenarios where multiple candidates exhibit identical heuristic or cost values. The implementation of a tie-breaking protocol can markedly influence the efficiency and computational trajectory of algorithms like A*, by systematically privileging certain nodes or paths within the search space, thereby mitigating the computational burden associated with equivocal decision nodes.

1.3 Comparison and Analysis

From the results of the distance and routes shown in Table 3,4,5,6,7, several conclusions can be drawn, including:

- The performance comparison among different search strategies in Figure 1 revealed that the Greedy algorithm is the fastest strategy but can miss the optimal path due to its short-term focus. A* algorithm outperforms Greedy by efficiently finding the shortest paths, considering both current and future costs. Dijkstra's algorithm is reliable like A* but slower because it meticulously compares all point costs. In summary, taking both the path optimally and computation complexity into consideration, A* is the best choice.

- The results in Figure 4 and 5 suggest that when the searching radius of RRT* is set to 10, there is a notable acceleration in computation speed accompanied by a significant increment in the overall distance error. When the searching radius is set to 30, some individual errors are particularly large. An optimal balance is observed when the search radius is adjusted to 20, contributing to the occurrence of considerable individual errors due to the intrinsic stochasticity of the RRT* algorithm. It is important to note that with an increase in step size, the resultant path lengthens, albeit with a reduction in computational time.

- The comparison in Figure 2 indicate that though the analytic method incurs a greater computational time, it unfailingly produces the shortest path. Among the alternative strategies, the Chebyshev heuristic notably leads to longer routes. Hence, the analytic approach is deemed the most efficacious for path optimization.

- The Heuristic Ratio shown in Figure 3impacts only the search's temporal efficiency, not the path's optimality. Setting this ratio to 1 minimizes search time, whereas 0.5 extends it the most. A ratio of 0.75 slightly outperforms 0.25 in improving search speed.

- According to Table 6, A* search with and without tie-breaking yields the same optimal path length. However, the tie-breaking strategy slightly reduces computation time by resolving nodes with equal f-scores, thereby expanding fewer nodes. Although the improvement is limited, tie-breaking can enhance efficiency in scenarios where minor gains are desired.

2 Task2

The problem of visiting all given locations and returning to the starting point is typically classified 'Travelling Salesman Problem (TSP)'. The objective of the Travelling Salesman Problem is to find the shortest possible route that allows a salesman to visit each city once and return to the starting point. For this particular problem, we can conceptualize it as an instance of the TSP, with the elevator serving as the initial and final location.

2.1 Greedy Search

Greedy Search represents an heuristic approach for finding approximate solutions by sequentially selecting the nearest unvisited city from the current location until all cities are covered, culminating in a return to the starting point. Although characterized by simplicity in implementation and rapid computation, this algorithm typically falls short of guaranteeing optimal solutions, particularly in scenarios involving extensive problem scales.

2.2 Branch and Bound

The branch-and-bound framework is a fundamental and widely-used methodology for producing exact solutions to NP-hard optimization problems. The technique, which was first proposed by Land and

Doig, is often referred to as an algorithm; however, it is perhaps more appropriate to say that the methodology encapsulates a family of algorithms that all share a common core solution procedure. This procedure implicitly enumerates all possible solutions to the problem under consideration, by storing partial solutions called subproblems in a tree structure. Unexplored nodes in the tree generate children by partitioning the solution space into smaller regions that can be solved recursively (i.e., branching), and rules are used to prune off regions of the search space that are provably suboptimal (i.e., bounding). Once the entire tree has been explored, the best solution found in the search is returned. [3]

2.3 Held-Karp Algorithm

Held and Karp have proposed, in the early 1970s, a relaxation for the Traveling Salesman Problem (TSP) as well as a branch-and-bound procedure that can solve small to modest-size instances to optimality. [4] The Held-Karp algorithm is a notable example of applying Dynamic Programming, which seeks the shortest possible route that visits each city exactly once and returns to the origin city. The Held-Karp algorithm computes an exact solution to the TSP by considering all subsets of the cities to be visited and calculating the shortest path that visits each subset, eventually encompassing all cities. Despite its ability to provide an exact solution, the Held-Karp algorithm has a computational complexity, making it impractical for large-scale problems due to the exponential growth in computations as the number of cities increases.

2.4 Results and Analysis

In this task, the best visit route is: ['start', 'store', 'food', 'movie', 'snacks', 'start'], with the shortest distance: 629.72m. The final shortest route is depicted in Figure 6b.

The Greedy Search, Branch and Bound, and Held-Karp algorithms all found the optimal solution of 629.72 meters for the given TSP instance, refering to Table 8. However, their execution times varied, with Greedy Search being the fastest at 0.02 ms, followed by Held-Karp at 0.048 ms, and Branch and Bound taking the longest at 0.08 ms. These results highlight the trade-offs between computation time and solution quality. The choice of algorithm should consider the problem scale, desired optimality, and acceptable computational cost.

3 Task3

Several algorithms are commonly used for vehicle path tracking, each with its own advantages and applications. The most popular ones are Proportional-Integral-Derivative (PID) control, Model Predictive Control (MPC), and Linear Quadratic Regulator (LQR). PID generates the control input based on the error signal, which does not require the system model and is thus insensitive to system disturbances. However, it may lack smoothness and accuracy in high-dynamic cases or when the controlled object large inertia. LQR guarantees closed-loop stability for the system dynamics but can only handle linear systems, whereas the vehicle is a typical non-linear system. MPC combines the advantages of feedback control, optimal control, and kinetic model, making it powerful for such problems, with the cost of high computational complexity. In this task, MPC is implemented and tested.

3.1 System Model

In this task, the MPC controller is employed to control a differential-drive car. The state of the car is represented by three variables: the x-coordinate (x), the y-coordinate (y), and the orientation (θ). The input variables are the linear velocity (v) and the angular velocity (ω) of the car.

The discrete state space equation is derived as Equation 5, where k denotes the current time step, $k+1$ represents the next time step, and Δt is the time interval between the steps. The next state $[x_{k+1}, y_{k+1}, \theta_{k+1}]^T$ is computed based on the current state $[x_k, y_k, \theta_k]^T$ and the input variables v_k and ω_k.

At each timestamp t, the MPC receives a state measurement and calculates the optimal control action sequence that drives the predicted system output to the desired reference. The optimal control problem is converted into a nonlinear programming (NLP) problem using the direct multiple shooting method [5], as shown in Equation 6. Here, $||\cdot||_{-}\chi$ represents the quadratic form, and $\chi \in R$ is a positive semi-definite diagonal matrix. $\mathbf{x}^r(t)$ denotes the reference state to be tracked. Constraint (a) represents the system dynamics differential equation derived from Equation 5, constraint (b) indicates the initial condition with x_0 being the real-time measured state value, and constraints $(c) - (d)$ define physical limitations.

3.2 Implementation

In this implementation, CasADi [6] is utilized as the MPC code formulation framework, and IPOPT [7] is utilized as the underlying non-linear optimization solver. Our code has been pushed to the Github [1].

One major challenge encountered during the implementation is the change in θ angle. When the turning movement of the vehicle exceeds π or $-\pi$, the θ angle changes from π to $-\pi$ or from $-\pi$ to π due to the property of $atan2$, causing the vehicle to turn around. To address this issue, the state error between θ angles is remapped to $[-\pi, \pi]$. Another challenge with the MPC controller is that, the speed control is relatively difficult. When the controller receives the reference path, the distance between way points are fixed. However, when the MPC controller tracks a different velocity, the time for the vehicle to travel the same distance in the reference path changes. To compensate for this error, the dt parameter in the MPC controller should also change inversely proportional to the speed.

3.3 Results and Analysis

As shown in Figure 9 and the comparisons presented in Table 10, the MPC controller achieves higher accuracy than the provided PID and Stanley controllers. Throughout the entire process, the MPC-controlled vehicle follows the track closely and performs well in most metrics. After changing the shape of the path, the MPC controller still outperform the PID controller in most metrics.

[1] https://github.com/zyaqcl/ME5413_Planning_Project/tree/main

References

[1] A. Goyal, P. Mogha, R. Luthra, and N. Sangwan, "Path finding: A* or dijkstra's?" *International Journal in IT & Engineering*, vol. 2, no. 1, pp. 1–15, 2014.

[2] I. Noreen, A. Khan, and Z. Habib, "Optimal path planning using rrt* based approaches: a survey and future directions," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 11, 2016.

[3] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell, "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning," *Discrete Optimization*, vol. 19, pp. 79–102, 2016.

[4] P. Benchimol, J.-C. Régin, L.-M. Rousseau, M. Rueher, and W.-J. Van Hoeve, "Improving the held and karp approach with constraint programming," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer, 2010, pp. 40–44.

[5] C. Kirches, *The Direct Multiple Shooting Method for Optimal Control*. Wiesbaden: Vieweg+Teubner Verlag, 2011, pp. 13–29. [Online]. Available: https://doi.org/10.1007/978-3-8348-8202-8_2

[6] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, 2018.

[7] A. Wächter and L. Biegler, "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming," *Mathematical Programming*, vol. 106, no. 1, pp. 25–57, 2006. [Online]. Available: https://doi.org/10.1007/s10107-004-0559-y

A Task1

Algorithm 1 Dijkstra's Algorithm

Input: Graph G, source vertex s
Output: Shortest distance to all vertices from s

Create vertex set Q
for each vertex v in G **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow undefined$
    Add v to Q
$dist[s] \leftarrow 0$
while Q is not empty **do**
    $u \leftarrow$ vertex in Q with min $dist[u]$
    Remove u from Q
    for each neighbor v of u **do**
        $alt \leftarrow dist[u] + length(u, v)$
        if $alt < dist[v]$ **then**
            $dist[v] \leftarrow alt$
            $prev[v] \leftarrow u$

Algorithm 2 A* Search Algorithm

Input: Graph G, start $start$, goal $goal$
Output: Shortest path from $start$ to $goal$

Initialize open list with $start$
Initialize closed list to empty
while open list is not empty **do**
    $current \leftarrow$ node in open list with lowest f_{cost}
    if $current = goal$ **then**
        return reconstructed path
    Remove $current$ from open list
    Add $current$ to closed list
    for each neighbor of $current$ **do**
        if neighbor in closed list **then**
            continue to next neighbor
        Calculate f_{cost} for neighbor
        if neighbor not in open list **then**
            Add neighbor to open list

Algorithm 3 Greedy Best-First Search

Input: Graph G, start $start$, goal $goal$
Output: Path from $start$ to $goal$ if exists

Initialize open list with $start$
Initialize closed list to empty
while open list is not empty **do**
    $current \leftarrow$ node in open list with lowest heuristic cost to $goal$
    if $current = goal$ **then**
        return path reconstructed from $start$ to $goal$
    Remove $current$ from open list
    Add $current$ to closed list
    for each neighbor of $current$ **do**
        if neighbor is not in open or closed list **then**
            Add neighbor to open list

Algorithm 4 RRT* Algorithm

Input: Search space S, initial configuration q_{init}, goal region Q_{goal}, step size ϵ, cost threshold C_{max}
Output: Tree T approximating the shortest path from q_{init} to Q_{goal}

Initialize T with q_{init}
while goal not reached and within computation budget **do**
    $q_{rand} \leftarrow$ random configuration in S
    $q_{nearest} \leftarrow$ nearest node in T to q_{rand}
    $q_{new} \leftarrow$ steer from $q_{nearest}$ towards q_{rand} by ϵ
    if path from $q_{nearest}$ to q_{new} is collision-free **then**
        Find neighbors of q_{new} within a radius
        $q_{min} \leftarrow q_{nearest}$
        Connect q_{new} to q_{min} in T
        Try to rewire the tree with q_{new} if it provides a lower cost path
return T

$$\text{Manhattan}(p, q) = |x_2 - x_1| + |y_2 - y_1| \tag{1}$$

$$\text{Euclidean}(p, q) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{2}$$

$$\text{Chebyshev}(p, q) = \max(|x_2 - x_1|, |y_2 - y_1|) \tag{3}$$

$$\text{Analytic}(p, q) = 0.2 \times (|y_2 - y_1| + |x_2 - x_1| + (2 - 2) \times \min(|x_2 - x_1|, |y_2 - y_1|)) \tag{4}$$

Table 1: Shortest distances in Task 1.

From \ To	start	snacks	store	movie	food
start	0.0	142.48	155.13	178.89	223.32
snacks	142.48	0.0	114.79	107.51	133.43
store	155.13	114.79	0.0	209.42	110.87
movie	178.89	107.51	209.41	0.0	113.72
food	223.32	133.43	110.87	113.72	0.0

Table 2: The number of visited cells in Task 1.

From \ To	start	snacks	store	movie	food
start	0.0	36019	16075	23753	73965
snacks	20022	0.0	10011	10929	25711
store	20673	18249	0.0	42601	8535
movie	21626	9521	24697	0.0	22658
food	63305	34364	9483	20303	0.0

Table 3: Performance comparison among different search strategies.

Algorithm	Metrics	Snacks	Store	Movie	Food
A*	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	0.87	0.45	0.56	1.90
Dijkstra's	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	1.38	1.58	1.87	2.76
GBFS	Distance (m)	145.45	164.21	210.77	238.79
	Time (s)	0.14	0.01	0.06	0.11

Table 4: Performance comparison among different heuristic functions.

Heuristic	Metrics	Snacks	Store	Movie	Food
Analytic	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	0.90	0.47	0.57	1.94
Chebyshev	Distance (m)	147.05	170.40	185.92	237.49
	Time (s)	0.14	0.16	0.47	0.20
Euclidean	Distance (m)	144.54	163.93	181.79	238.46
	Time (s)	0.15	0.01	0.05	0.15
Manhattan	Distance (m)	143.97	159.27	209.13	232.74
	Time (s)	0.17	0.34	0.14	0.41

Table 5: Performance comparison among different heuristic ratios.

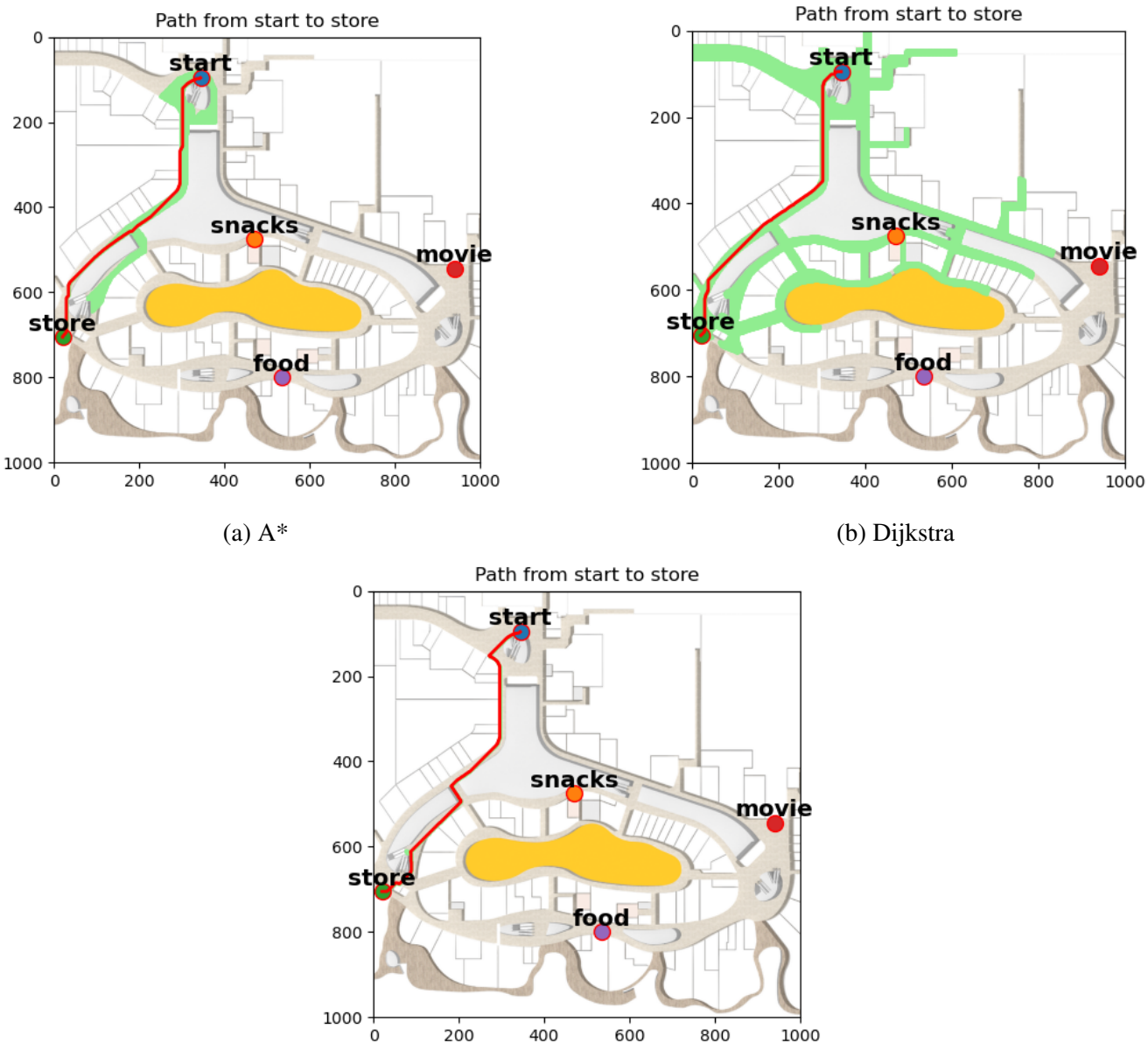
Ratio	Metrics	Snacks	Store	Movie	Food
0.25	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	1.11	1.29	1.50	2.74
0.5	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	1.26	1.31	1.29	2.83
0.75	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	1.05	1.06	0.96	2.33
1.0	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	0.96	0.46	0.57	1.96

Table 6: Performance comparison in terms of whether use tie-breaking mechanism.

Tie-breaking	Metrics	Snacks	Store	Movie	Food
True	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	0.95	0.47	0.59	1.99
False	Distance (m)	142.48	155.13	178.89	223.32
	Time (s)	0.94	0.49	0.61	2.05

Table 7: Performance comparison among different parameters for RRT*.

Parameter	Metrics	Snacks	Store	Movie	Food
Radius=10	Distance (m)	523.58	272.58	229.23	306.22
	Time (s)	2.37	0.06	0.04	0.08
Radius=20	Distance (m)	146.13	171.59	192.22	283.79
	Time (s)	0.36	0.47	0.04	0.04
Radius=30	Distance (m)	136.99	413.60	213.68	347.24
	Time (s)	0.71	0.13	0.02	0.20
Radius=40	Distance (m)	172.60	152.21	172.61	295.55
	Time (s)	0.07	0.30	0.01	0.08
Step=5	Distance (m)	167.45	257.53	170.59	259.35
	Time (s)	2.78	1.68	0.26	0.53
Step=10	Distance (m)	154.26	370.02	173.43	261.40
	Time (s)	0.61	1.36	0.08	0.10
Step=20	Distance (m)	469.72	396.83	181.97	288.88
	Time (s)	0.42	0.12	0.11	0.03
Step=30	Distance (m)	198.53	432.52	185.50	286.16
	Time (s)	0.44	0.20	0.01	0.02

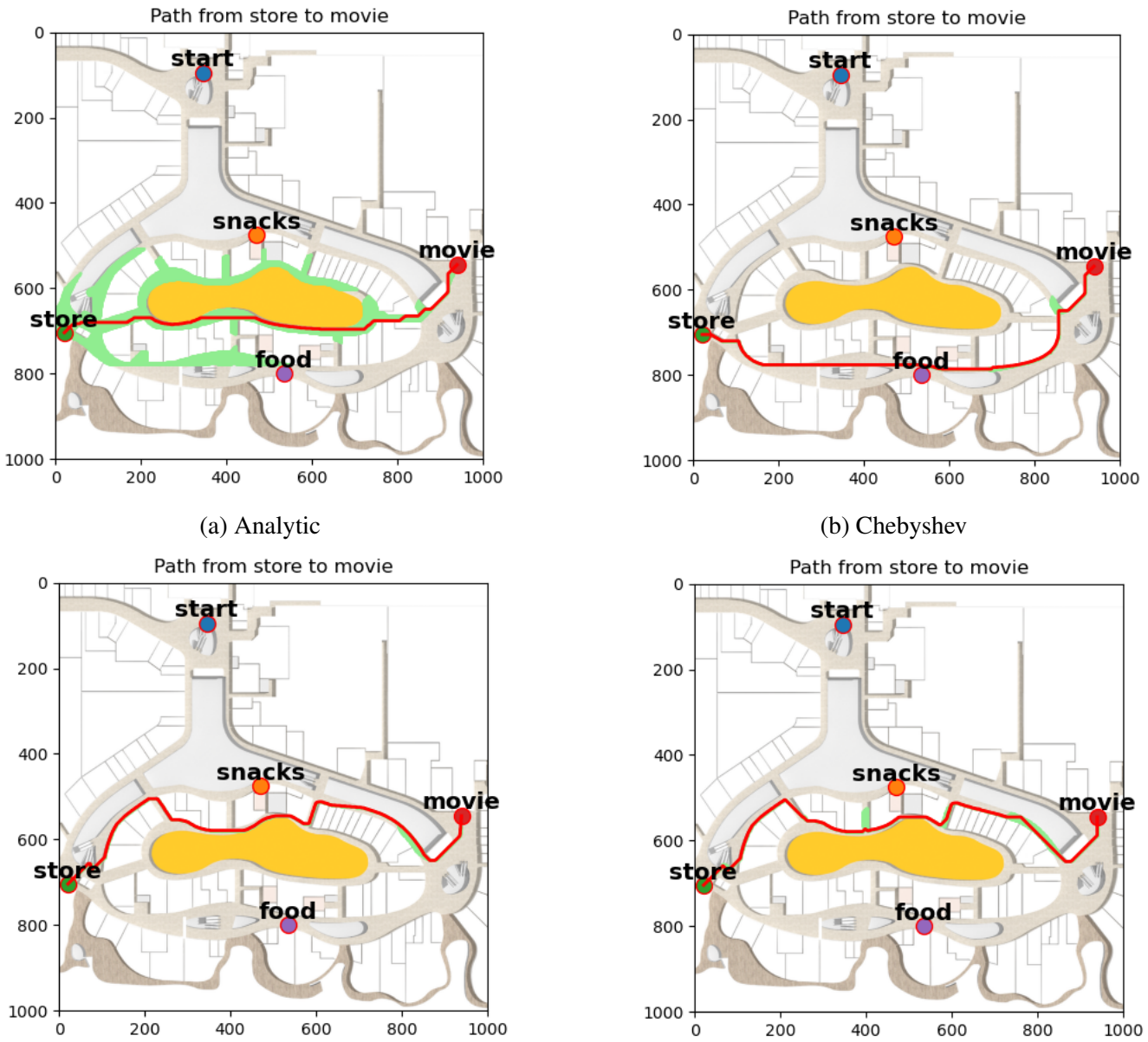


(a) A*

(b) Dijkstra

(c) Greedy Best First Search

Figure 1: Path searching comparison among different search strategies. The green pixels indicates the visited cells



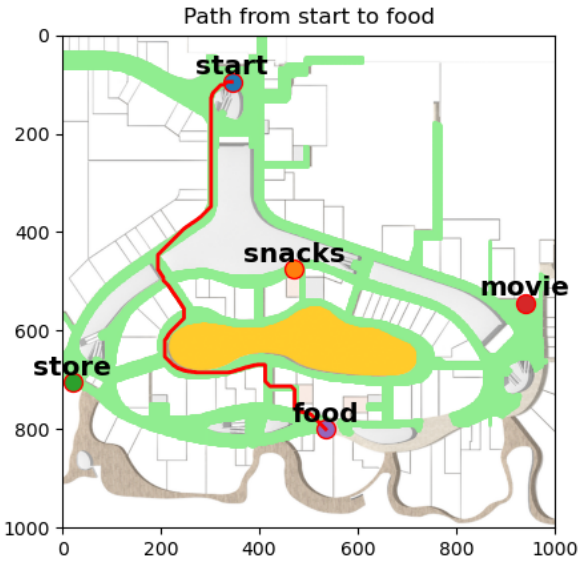
(a) Analytic

(b) Chebyshev

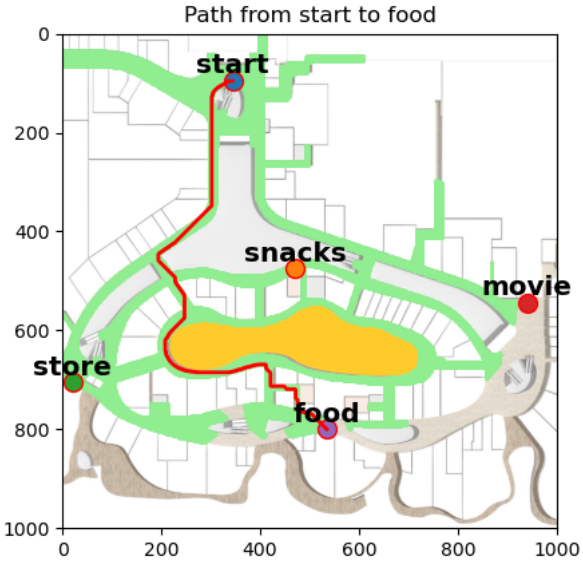
(c) Euclidean

(d) Manhattan

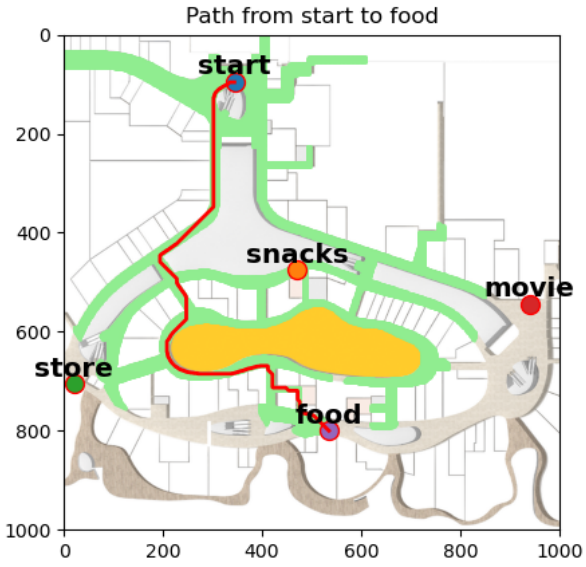
Figure 2: Path searching comparison among different heuristic functions in A*.



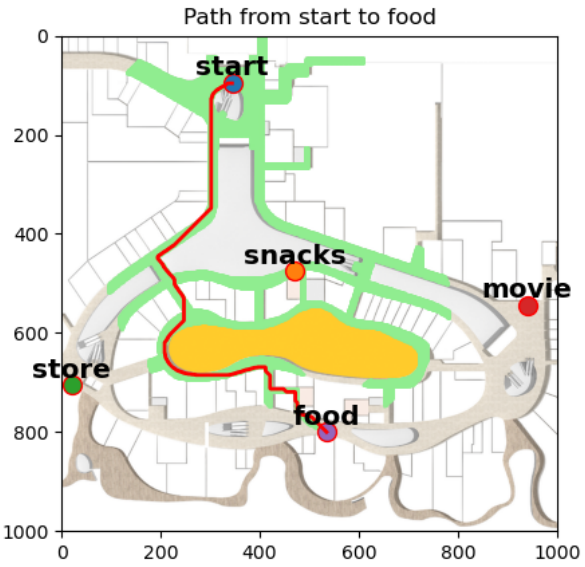
(a) Heuristic Ratio=0.25



(b) Heuristic Ratio=0.5

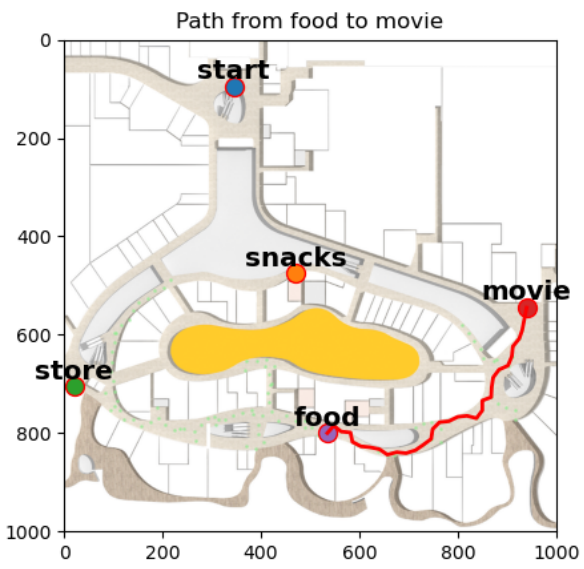


(c) Heuristic Ratio=0.75

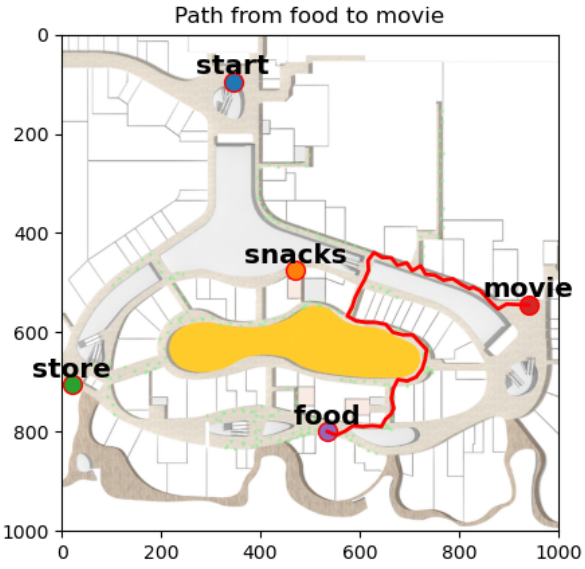


(d) Heuristic Ratio=1.0

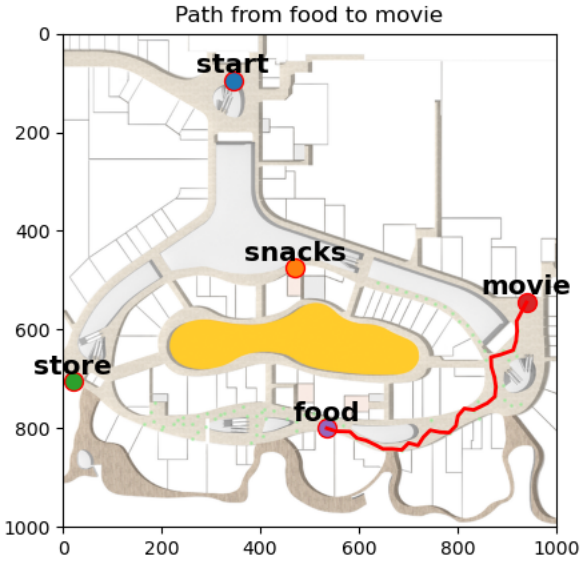
Figure 3: Path searching comparison among different heuristic ratios in A*.



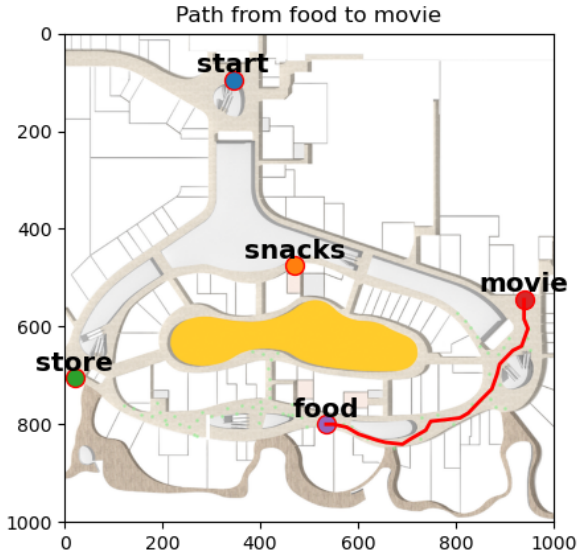
(a) Radius=10



(b) Radius=20



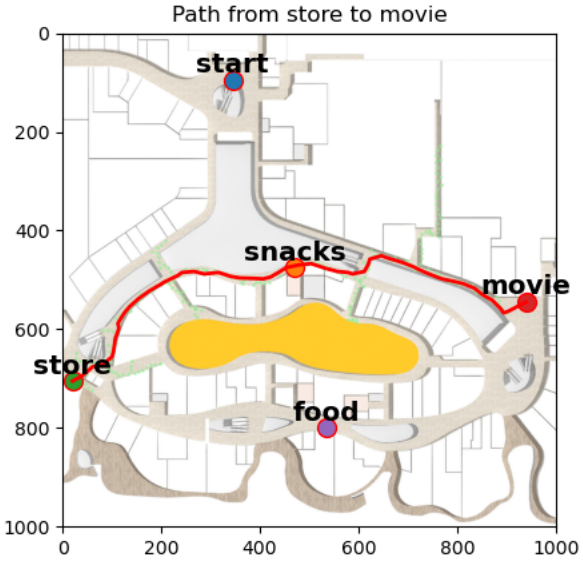
(c) Radius=30



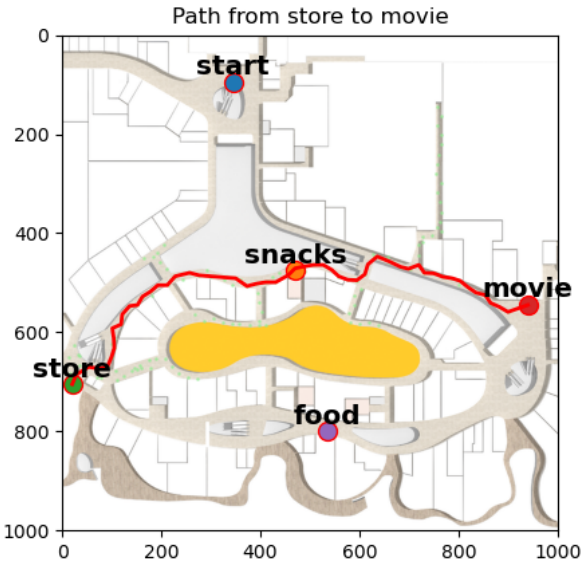
(d) Radius=40

Figure 4: Path searching comparison among different search radius of rewiring process in RRT*

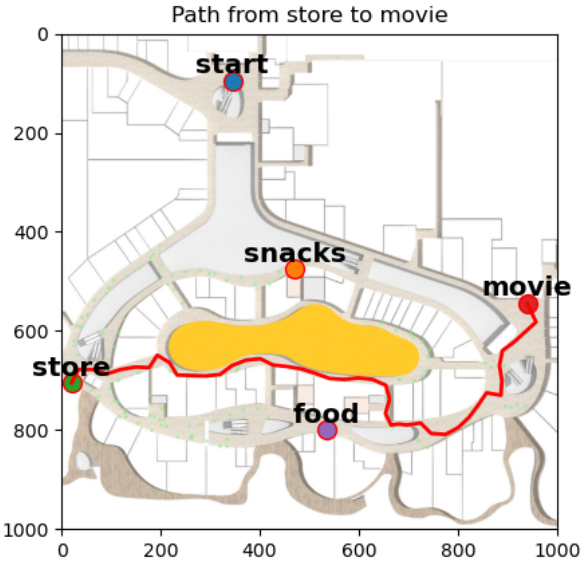
(a) Step=5



(b) Step=10



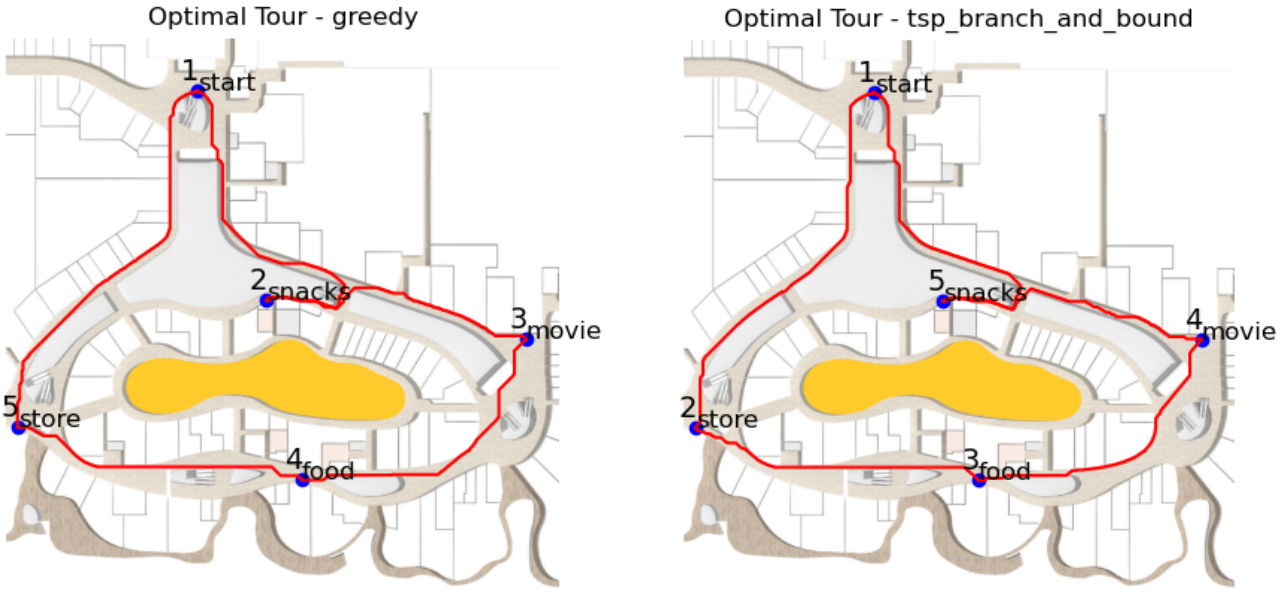
(c) Step=20



(d) Step=30

Figure 5: Path searching comparison among different steps in RRT*

B Task2



(a) Greedy Search

(b) Branch-and-Bound (Held-Karp)

Figure 6: Route results of Task 2.

Table 8: Performance comparison of Task2

Algorithm	Total Distance (m)	Time (ms)
Greedy	629.721	0.02
Branch_and_Bound	629.721	0.08
Held-Karp	629.721	0.048

C Task3

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k + v_k \Delta t \cos(\theta_k) \\ y_k + v_k \Delta t \sin(\theta_k) \\ \theta_k + \omega_k \Delta t \end{bmatrix} \tag{5}$$

$$
\begin{aligned}
\min_{\mathrm{x,u}} \quad & \int_{t}^{t+T} \left(\|\mathrm{x}(t) - \mathrm{x}^{r}(t)\|_{Q} + \|\mathrm{u}(t)\|_{R} \right) dt \\
s.t. \quad & \dot{\mathrm{x}} = f(\mathrm{x}, \mathrm{u}) && (a) \\
& \mathrm{x}(0) = \mathrm{x}_0 && (b) \\
& |\mathrm{x}_i| \leq c_{1,i}, i \in \{1, 2, 3, 4\} && (c) \\
& |\mathrm{u}| \leq c_2 && (d)
\end{aligned}
\tag{6}
$$

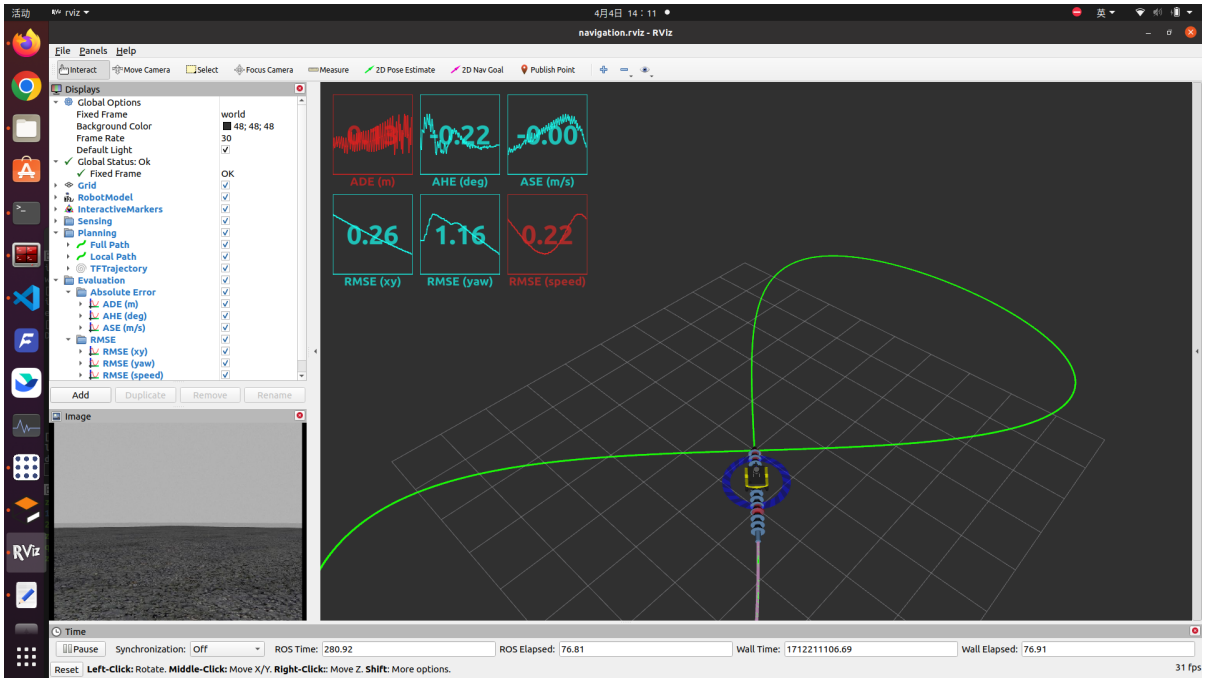


Figure 7: MPC tracking effect.

Table 9: Performance comparison of MPC and PID with original path.

Algorithm	ADE	ASE	AHE	RMSE(xy)	RMSE(yaw)	RMSE(speed)
Given PID	0.31	2.00	-1.00	0.44	5.41	0.32
Tuned PID	0.30	1.70	-1.00	**0.22**	3.38	0.32
MPC	**0.13**	**-0.22**	**0.00**	0.26	**1.16**	**0.22**

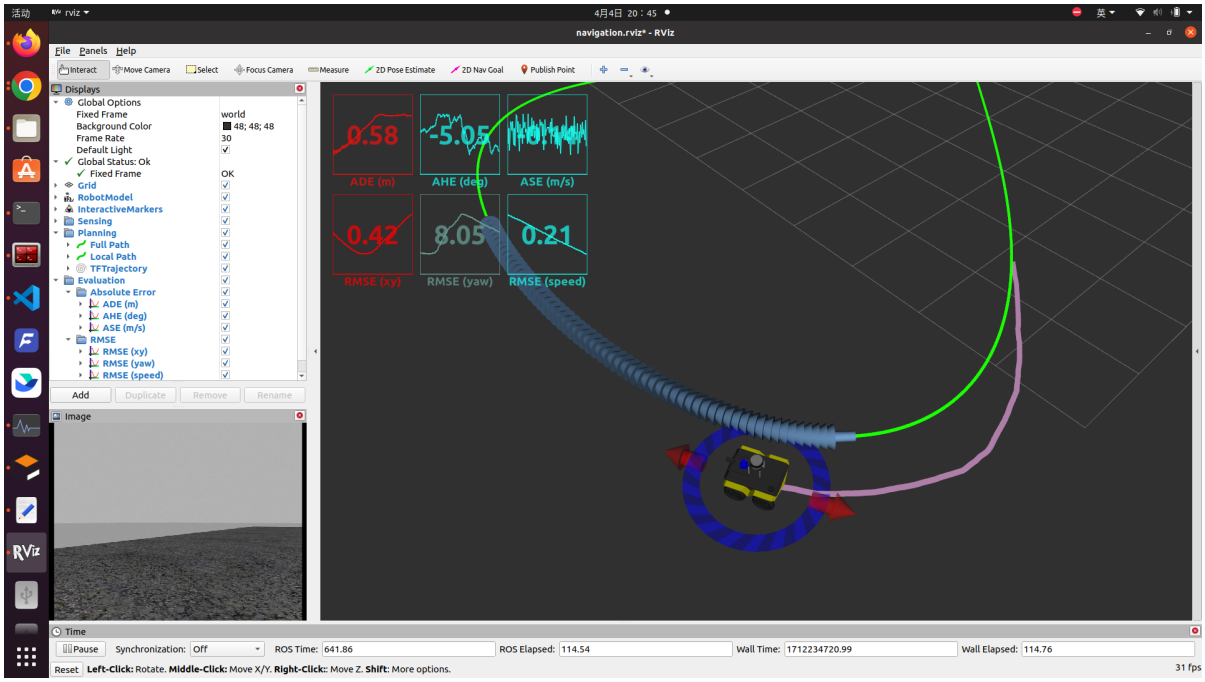


Figure 8: PID tracking effect.

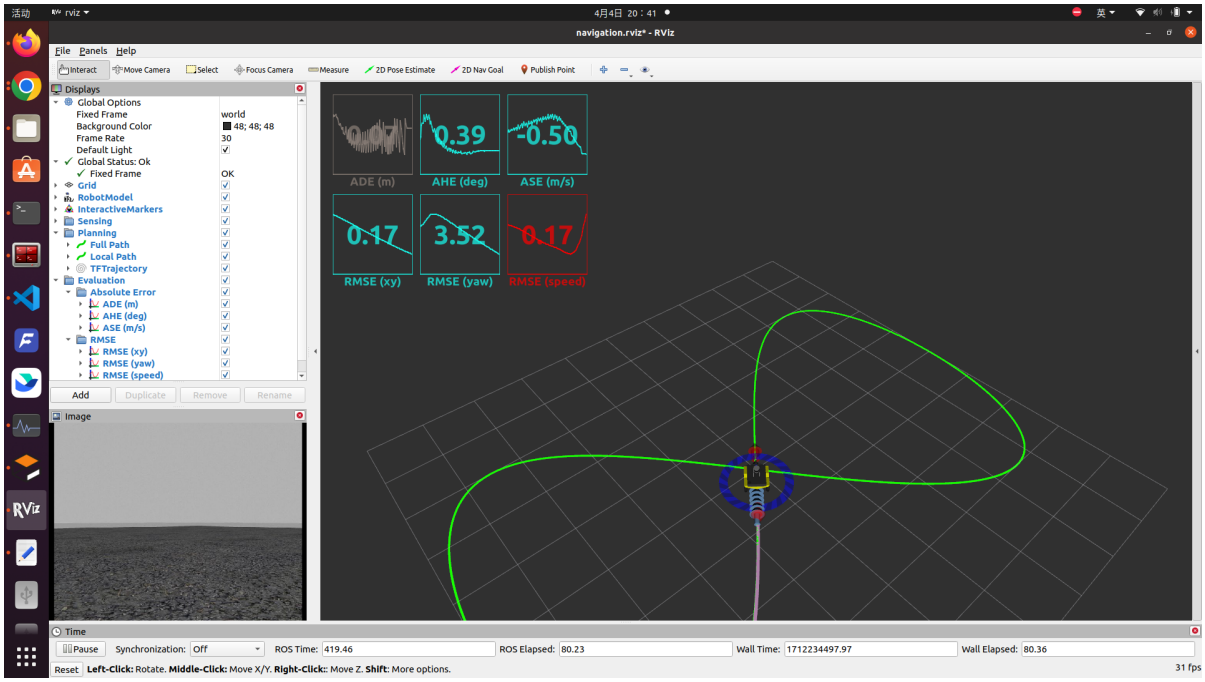


Figure 9: MPC tracking effect on the trajectory of differnet shape.

Table 10: Performance comparison of MPC and PID on the trajectory of different shape.

Algorithm	ADE	ASE	AHE	RMSE(xy)	RMSE(yaw)	RMSE(speed)
Given PID	0.31	3.42	-0.50	0.41	8.97	0.50
Tuned PID	0.30	1.70	-1.00	0.22	**3.38**	0.32
MPC	**0.07**	**0.39**	**-0.50**	**0.17**	3.52	**0.17**