National University of Singapore

Faculty of Mechanical Engineering

ME5413: Autonomous Mobile Robotics

Homework 3: Planning

Group 15

Student Name	Student ID
Chen Ziao	A0285086U
Fan Xiuqi	A0285060J
Wang Sijie	A0284859E

https://github.com/LandoFan/ME5413_Planning_HW3

Supervisor:

Prof. Marcelo H Ang Jr

Submission Date : 03/04/2024

1 Global Planning

1.1 Problem Statement

The path planning algorithms are applied to the map of Vivocity level 2, where the visitor plans to visit the five key locations ("start", "snacks", "store", "movie" and "food") with the optimal path.

1.2 Dijkstra's Algorithm

The implementation of Dijkstra's algorithm starts with the source node, gradually expanding the set of vertices until all vertices in the graph are included. At each step, the algorithm selects the vertex with the smallest "current shortest distance" (represented by the cost-so-far function $g(n)$ to add to the solved set, and then updates the shortest distance estimates for other vertices based on this newly added vertex.

Key Steps

1. Initialization: Set the initial node as s, and initialize all nodes' shortest distance estimation $d[v]$ to infinity (∞), except for $d[s] = 0$.
2. Selection of the Vertex with the Minimum Distance: Select a vertex u from the set of vertices Q whose shortest paths are yet to be determined, with the minimum distance from the source node, i.e., $u = argmin(d[v] : v \in Q)$
3. Updating the Distance of Adjacent Vertices: For each adjacent vertex v of u, update the shortest distance $d[v] = 0$ from v to the source node s. The updating formula is as follows: $d[v] = min(d[v], d[u] + g(u, v))$. Here, $g(u, v)$ is the cost of the edge (u, v). This step checks if there exists a shorter path to v via the newly added vertex s to the set. If so, $d[v]$ is updated accordingly.
4. Repeat Selection and Updating Steps: Repeat steps 2 and 3 until all vertices have been processed, i.e., the set Q is empty.
5. Result: Ultimately, $d[v]$ will represent the shortest distance from the source node s to all other vertices.

1.3 Greedy Best First Search Algorithm

The Greedy Best First Search algorithm prioritizes finding a path by utilizing a heuristic function to estimate the cost from the current node to the target, which is the cost-to-go function $h(n)$. The efficiency of this algorithm stems from its focus on rapidly approaching the target, despite not guaranteeing the shortest path. The key steps are similar to the Dijkstra except for the heuristic cost $h(n)$.

1.4 A* Algorithm

The A* algorithm emphasizes a balance between exploring and advancing towards the goal. The key idea is the cost function $f(n) = g(n) + h(n)$, where the cost-so-far $g(n)$ to the start state and cost-to-go $h(n)$ to the goal state are combined. The key steps are similar to the Dijkstra except for the heuristic cost $f(n)$.

1.5 Heuristic Function

The choice of the heuristic function $h(n)$ has a significant impact on the efficiency and effectiveness of the algorithm. It must be admissible (i.e., it will not overestimate the actual minimum cost) to ensure the algorithm can find the shortest path. Common heuristic functions include the Manhattan distance, Euclidean

distance, etc., with the specific choice depending on the nature of the problem. Based on the requirement, we select the diagonal distance due to 8 directions. According to the Equation 1, where x_{target} and y_{target} denote the coordinates of the target node, and x_n and y_n represent the coordinates of the current node.

1.6 Code Explanation

The path planning algorithms are implemented through 'Dijkstra', 'greedy', and 'a_star_search' functions, where the 'Dijkstra' function utilizes Dijkstra's algorithm focusing on global optimum paths at the expense of time, while the 'greedy' function employs a heuristic-based greedy search aiming for quick, local optima which may not guarantee the shortest path, and 'a_star_search' function combines the best of both, using a heuristic to efficiently balance between cost and distance to find optimal paths swiftly. These functions illustrate a range of approaches for path planning, from thorough exploration to heuristic-driven shortcuts.

In addition to the path-planning functions, the 'obs_map' function processes the grid map to identify and record the locations of obstacles to ensure that path-planning algorithms avoid these areas. The 'heuristic(a, b)' function defines a heuristic function to estimate the distance between two points, which combines the concepts of Manhattan distance and diagonal distance to apply to straight and diagonal movement in the grid. The 'get_neighbors' function identifies all potential neighboring nodes of a specific node, considering obstacles, collision possibilities, own size, and boundary constraints to improve the accuracy of path planning. The 'reconstruct_path' function reconstructs the path from the start to the goal using the given predecessor node mapping and calculates the total distance of the path.

1.7 Evaluation of Algorithm Performance

Comparing the three algorithms, all the results regarding distance, searching time, and visited cells are detailed in the appendix. The shortest path distance between the five key locations by Dijkstra, Greedy, and A* are recorded in Tab. 1, Tab. 4, Tab. 7 respectively, while the searching time is recorded in Tab. 2, Tab. 5, Tab. 8, and the number of grids searched is recorded in Tab. 3, Tab. 6, Tab. 9.

The searching time of the Greedy algorithm is the least among the three ways because it lays great emphasis on local optimum, resulting in the minimal number of cells being visited. Because of the tendency to fall into local optimal solutions, distances tend to vary when we travel back and forward from one place to another. Dijkstra's algorithm, targeting the global optimum, tends to slow down its process. A* algorithm merges advantages by maintaining the search for the global optimum while accelerating the runtime, proving to be the most effective approach.

1.8 Problem Faced and Further Improvement

1. Time-consuming problem: If we read the map grid by grid, judging whether this grid is free to go or not, it will be very time-consuming. Thus, we define a function called 'obs_map():' to identify the obstacle. By reading this set, the runtime can be improved to a great extent.

2. Further Improvement: In our algorithm, we judge the obstacle and circular footprint in the function 'get_neighbors', if we can deal with this problem when pre-processing the image information, the runtime can be further improved.

2 The "Traveling Shopper" Problem

2.1 Problem Statement

Given the shortest path distance in Tab. 1, Tab. 4, Tab. 7, the visitor wishes to visit all the locations and return to the start location with the optimal route. One way is to enumerate all possible routes, or some planning algorithms can be used. In this problem, for scenarios involving just four locations, there are 24 potential permutations with certain positions. A planning model can be built to find the shortest path by using dynamic programming methods or the direct traversal method.

2.2 Direct Traversal Method

2.2.1 Key Steps and Code Explanation

1. Listing All Possible Routes: Initially, find all potential paths that start and end at the original location, ensuring each destination is visited precisely once are generated. The permutation of all positions needs to be considered, as the order of each location is random.

2. Calculating Total Distance for Each Route: Utilizing the distance table obtained from Task 1, the total distance for each route is calculated, including the distance from the last location back to the start.

3. Identifying the Shortest Route: By evaluating the total distances of all possible routes, the one with the shortest distance is identified as the best sequence for visitation.

2.3 Particle Swarm Optimization

The Particle Swarm Optimization (PSO) algorithm is a collective intelligence optimization technique. Inspired by the foraging behavior of birds, it simulates a group of particles (or "individuals") searching for the optimal solution in the solution space. Each particle represents a potential solution within the solution space, and it finds the optimal solution by tracking and adjusting its position.

2.3.1 Algorithm Principle

In the PSO algorithm, each particle possesses two primary attributes: position and velocity. The position represents the coordinates of a potential solution, while the velocity determines the direction and distance of the particle's movement. The movement of particles is influenced by two optimal values: Personal Best (pbest): The best position found by the particle thus far. Global Best (gbest): The best position found by the entire swarm of particles.

2.3.2 Algorithm Steps and Code Explanation

1. Initialization: Randomly generate a group of particles with positions and velocities in the function 'initialize_population()'.

2. Evaluation: Calculate the fitness (i.e., the quality of the solution) of each particle in the function 'calculate_distance()'.

3. Update Personal Best and Global Best: Update each particle's personal best and the global best positions based on the newly calculated fitness in the function 'update_global_best_particle()'

4. Update Velocity and Position: Adjust the velocity and subsequent position of each particle in the function 'update_particle_velocity()' according to the following formulas: Velocity update formula as shown in Equation. 2 and

position update formula as shown in Equation. 3. Here, w is the inertia, c_1 and c_2 are learning factors, $rand_1()$ and $rand_2()$ are random numbers within the range [0,1].

5. Iteration: Repeat steps 2 to 4 until a stopping criterion is met (reaching the maximum number of iterations or the fitness meeting a minimum standard).

2.4 Result

Leveraging the distance matrix obtained from global planning, both algorithms perform the same path route and achieve the shortest distance. The sequence traversed is 'start'-'snacks'-'movie'-'food'-'store'-'start'(Fig. 4), covering a total distance of 670 meters. Given the scenario involves only five locations, the direct traversal approach is faster than PSO as illustrated in Fig. 5. This discrepancy arises because PSO requires time to identify the optimal sequence. Nonetheless, when dealing with more complex problems, PSO may be more efficient than the direct method.

3 Path tracking

3.1 Problem Statement

The robot is required to follow the given Figure 8 track with algorithms including PID, Pure Pursuit, and LQR. The evaluation of these algorithms is based on metrics including RMSE position, RMSE heading, and RMSE speed.

3.2 Pure Pursuit

The Pure Pursuit algorithm is based on geometric methods, functioning by identifying the target point on the path and steering the robot towards it. The essence of the algorithm lies in calculating and adjusting the robot's steering angle to maintain its movement toward the "pursuit point" on the predefined path.

Algorithm Steps and Code Explanation

1. Parameter Configuration: Key parameters including 'PID_Kp', 'PID_Ki, 'PID_Kd', 'Speed_Target', and 'lookahead_distance' are dynamically configured through the 'dynamicParamCallback' function, which can be updated during runtime to adapt to changes in the environment.

2. Monitoring and Feedback: The 'PathTrackerNode' subscribes to the robot's odometry information (/gazebo/ground_truth/state) and planned local path (/me5413_world/planning/local_path), while applying callback functions 'robotOdomCallback' and 'localPathCallback' to handle the current state information from the robot and the planned path.

3. Computing the Lookahead Target Point: All points on the given path are iterated to find a point that is at least 'lookahead_distance' away from the robot's current position as the 'lookahead_point', towards which the algorithm will strive to guide the robot.

4. Generating Control Commands: The 'ControlOutput' function first calculates the linear speed based on a PID controller to maintain 'SPEED_TARGET'. Then, it calculates the angle from the robot's current position to the 'lookahead_point', and adjusts the angle by the current orientation angle to determine the yaw angle error. Finally, using a proportional controller (with 'Kp' as the proportional coefficient), it calculates the angular velocity control command based on the yaw angle error.

3.3 LQR

The Linear Quadratic Regulator algorithm designs controllers by minimizing a quadratic cost function to optimize system performance. The cost function includes the squared deviations of system states and the squares of control inputs, aiming to balance the relationship between control effort and system performance.

3.3.1 Algorithm Principle

1. Quadratic Cost Function: In the linear system, the cost function can be defined as the weights for system state deviations and control inputs, taking the form according to Equation. 4, where x is the system state deviation, u is the control input, and Q and R are weight matrices that adjust the relative importance of state deviations and control inputs in the cost function.

2. Optimization Objective: The goal of LQR design is to find a control law u that minimizes the cost function J. This typically involves solving a Riccati equation to obtain the optimal feedback gain K.

3. Feedback Control: The optimal control law is often in the form of state feedback, i.e., $u = -Kx$, where K is the optimal feedback gain obtained from solving the Riccati equation.

3.3.2 Code Explanation

In the algorithm, the system matrices A,B are defined in function 'gain' as well as the control matrices Q,R. The state is a 4x1 matrix including the error of position x, position y, heading error, and the velocity error. The input of the system is the linear velocity and the angular velocity, which is a 2x1 matrix. We use a function called the 'riccati_equation' to solve the cost function and get the gain K.

3.4 Result and Further Improvement

Compared with the three algorithms on this planning project, the pure pursuit performs the best. It shows the fastest and stablest response following the trace in global error with the least absolute position error. When it comes to the LQR control, it shows great control in the absolute speed error with the minimum fluctuation. It responds very fast and keeps stable in speed error. The global position error is greater than the origin method but big waves are seen in absolute position error.

Moreover, two 360-degree rotations happen when tracing the trajectory, which greatly influences the global heading error. However, the absolute error shows smaller at other moments compared with other methods. This may be caused by the input of the angular velocity, which does not control well. The states and inputs can be optimized to achieve a better performance.

4 Appendix

4.1 Equation

$$\begin{aligned} h(n) = &|x_{\text{target}} - x_n| + |y_{\text{target}} - y_n| \\ &+ (\sqrt{2} - 2)(\min(|x_{\text{target}} - x_n|, |y_{\text{target}} - y_n|)) \end{aligned} \tag{1}$$

$$v_i^{(t+1)} = w \cdot v_i^{(t)} + c_1 \cdot \text{rand}_1() \cdot (pbest_i - x_i^{(t)}) + c_2 \cdot \text{rand}_2() \cdot (gbest - x_i^{(t)}) \tag{2}$$

$$x_i^{(t+1)} = x_i^{(t)} + v_i^{(t+1)} \tag{3}$$

$$J = \int_0^\infty (x^T Q x + u^T R u)\, dt \tag{4}$$

4.2 Table

To \ From	start	snacks	store	movie	food
start	0.00	141.04	154.42	178.41	218.87
snacks	141.04	0.00	114.32	106.53	129.69
store	154.42	114.32	0.00	208.51	110.87
movie	178.41	106.54	208.51	0.00	111.37
food	218.87	129.69	110.87	111.37	0.00

Table 1: Path distance between key locations (m) by Dijkstra

To \ From	start	snacks	store	movie	food
start	0.00	1.840	2.192	2.747	3.931
snacks	3.571	0.00	2.189	1.705	3.024
store	2.306	1.471	0.00	3.602	1.467
movie	3.00	1.545	3.852	0.00	1.593
food	3.615	2.552	2.096	2.115	0.00

Table 2: Searching Time between key locations (s) by Dijkstra

To \ From	start	snacks	store	movie	food
start	0	82516	98338	121180	171942
snacks	152638	0	99609	79533	134505
store	107239	68218	0	169257	66560
movie	131418	67612	165847	0	70304
food	156932	111187	87958	88761	0

Table 3: Number of visited cells between key locations by Dijkstra

To \ From	start	snacks	store	movie	food
start	0	0.114	0.016	0.054	0.088
snacks	0.132	0	0.011	0.019	0.036
store	0.019	0.012	0	0.113	0.012
movie	0.032	0.015	0.022	0	0.042
food	0.218	0.318	0.021	0.018	0

Table 5: Searching Time between key locations (s) by Greedy Search

To \ From	start	snacks	store	movie	food
start	0	5284	715	2558	3483
snacks	5855	0	528	938	1674
store	913	537	0	1817	574
movie	1494	697	1078	0	1963
food	10331	14939	1039	968	0

Table 6: Number of visited cells between key locations by Greedy Search

To \ From	start	snacks	store	movie	food
start	0.00	161.01	173.85	195.65	284.77
snacks	145.64	0.00	120.29	108.75	187.64
store	163.32	122.25	0.00	240.76	131.44
movie	211.30	141.60	254.82	0.00	118.87
food	236.47	135.86	118.37	204.34	0.00

Table 4: Path distance between key locations (m) by Greedy Search

To \ From	start	snacks	store	movie	food
start	0.00	141.042	154.42	178.41	218.87
snacks	141.042	0.00	1140.32	106.53	129.69
store	154.42	114.32	0.00	208.51	110.87
movie	178.41	106.53	208.51	0.00	111.37
food	218.87	129.69	110.87	111.37	0.00

Table 7: Path distance between key locations (m) by Astar

From / To	start	snacks	store	movie	food
start	0.00	1.719	2.237	2.699	3.650
snacks	3.345	0.00	2.208	1.705	2.956
store	2.323	1.516	0.00	3.670	1.432
movie	3.188	1.578	3.900	0.00	1.605
food	3.595	2.479	2.047	2.081	0.00

Table 8: Searching Time between key locations (s) by Astar

From / To	start	snacks	store	movie	food
start	0	82516	98338	121180	171942
snacks	152638	0	99609	79533	134505
store	107239	68218	0	169257	66560
movie	131418	67612	165847	0	70304
food	156932	111187	87958	88761	0

Table 9: Number of visited cells between key locations by Astar

4.3 Figure

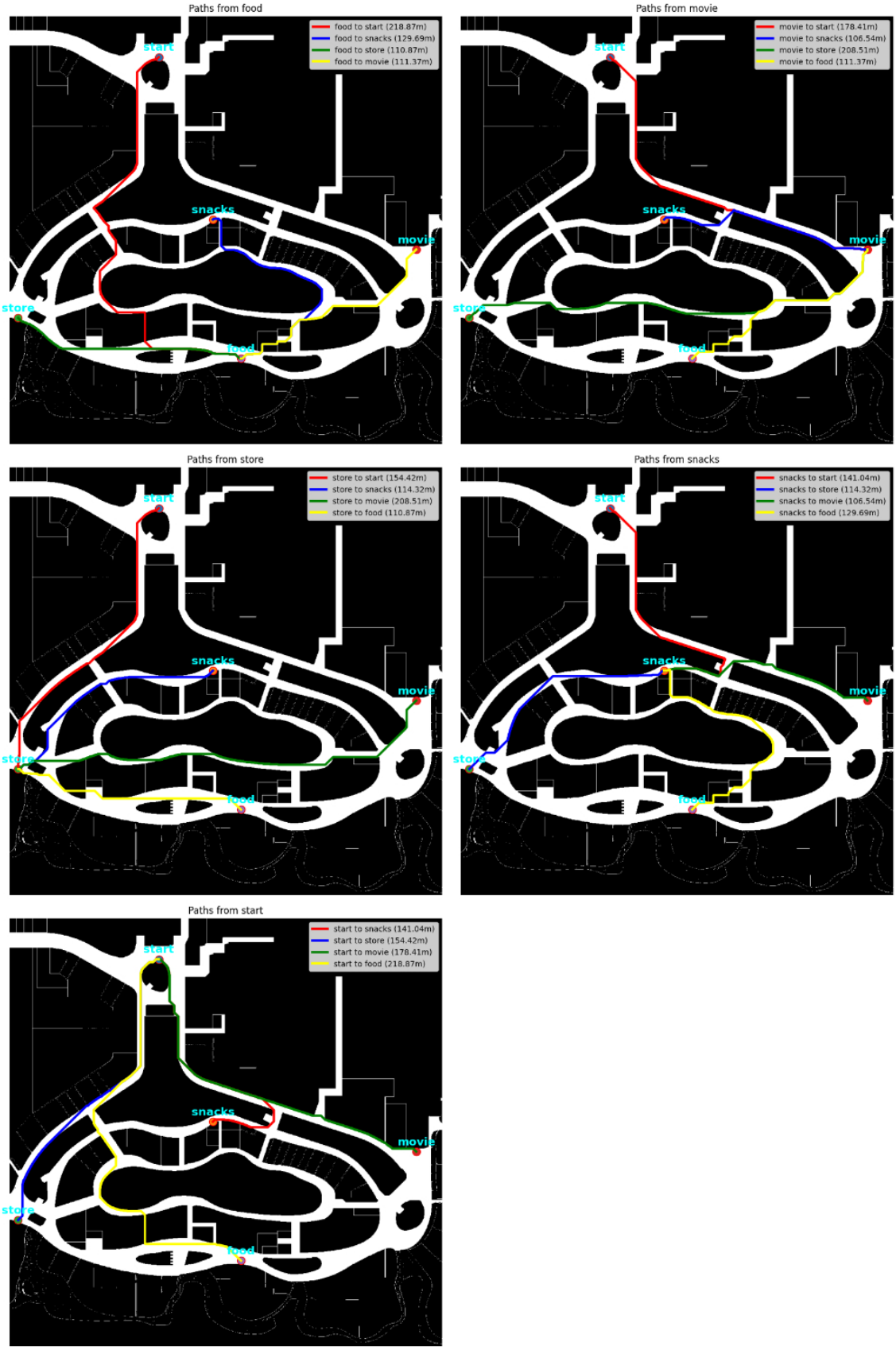

Figure 1: Different starts to different ends based on Dijkstra

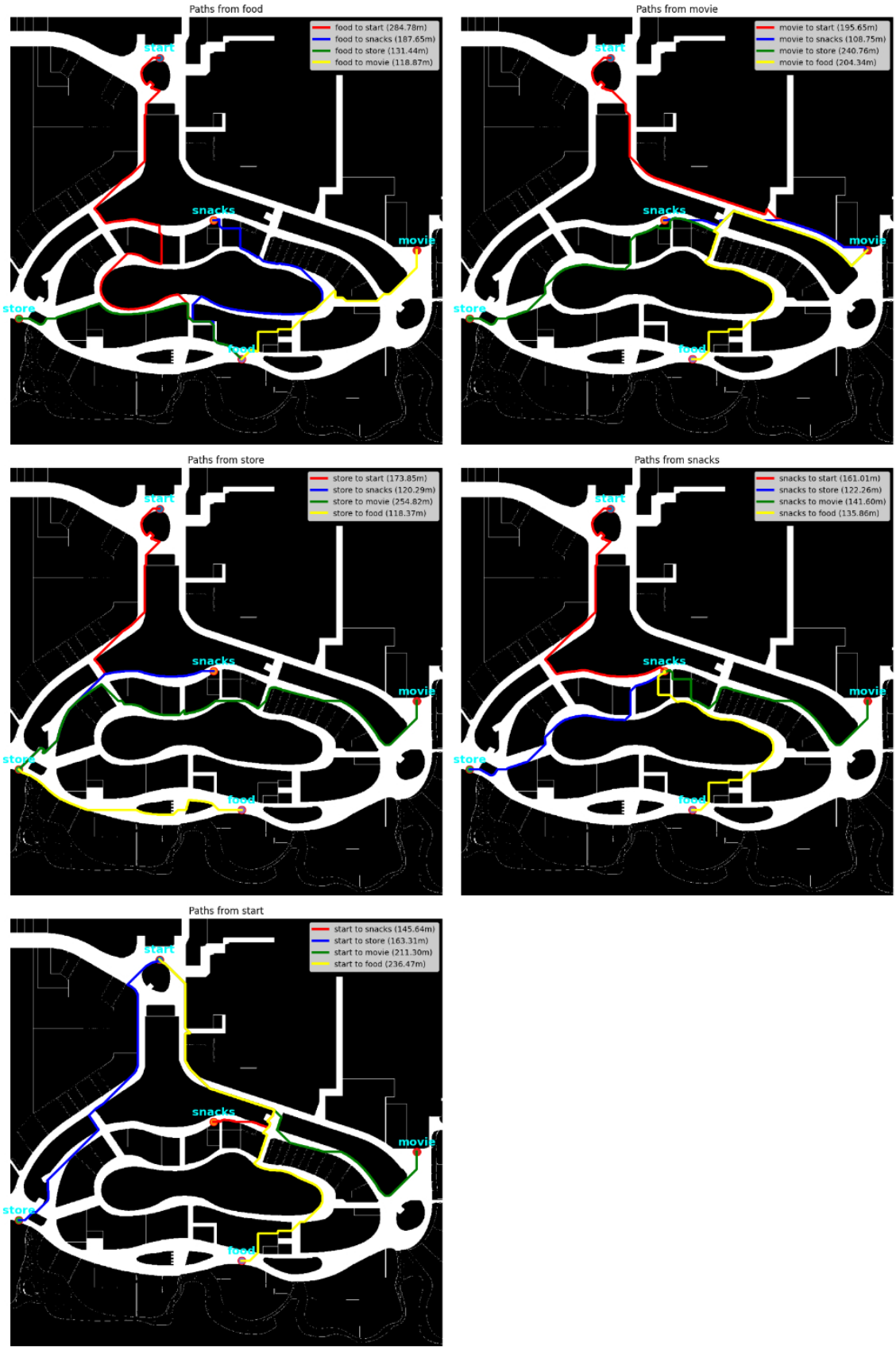

Figure 2: Different starts to different ends based on Greedy Search

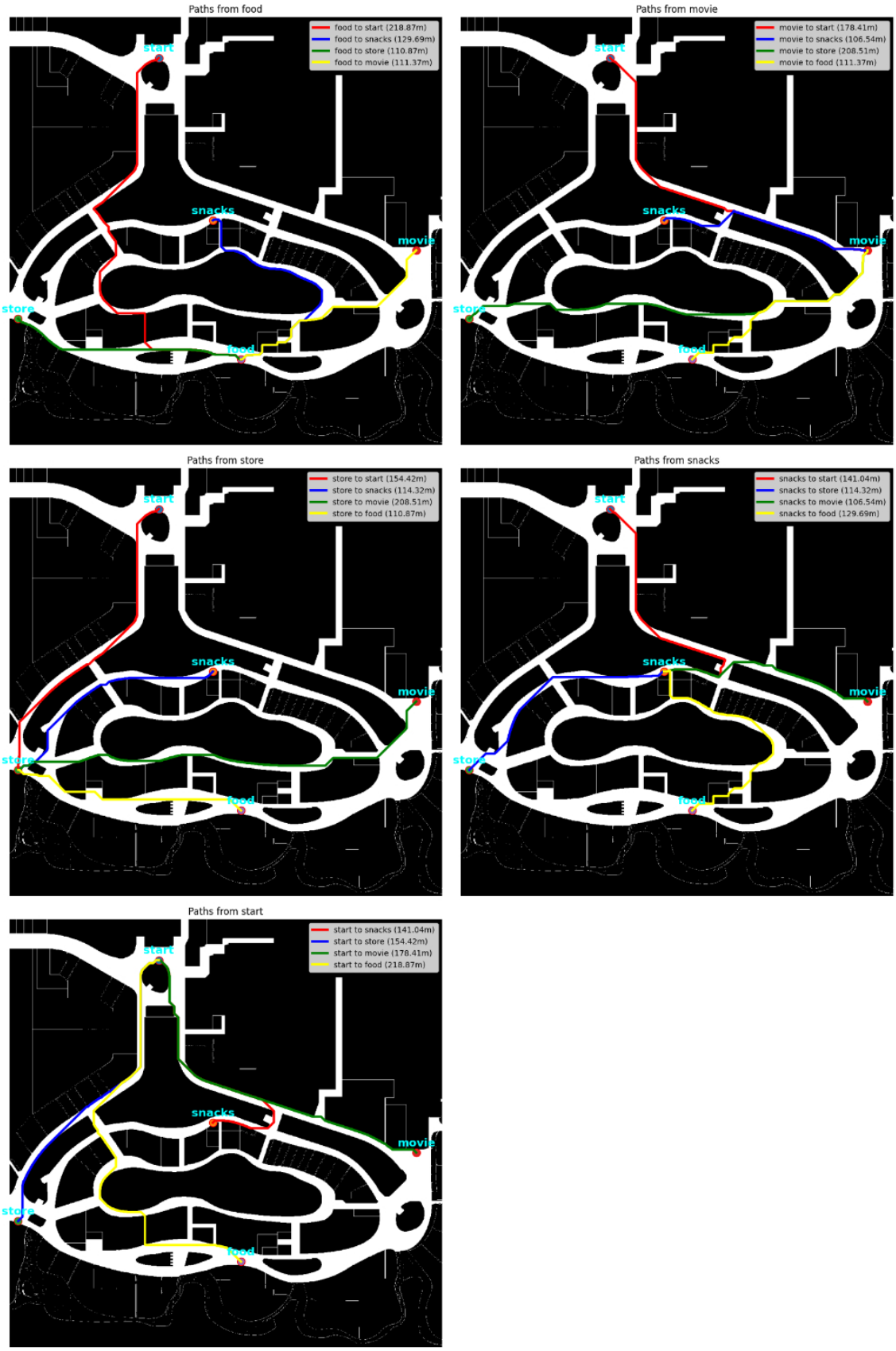

Figure 3: Different starts to different ends based on Astar

Figure 4: The route of the map

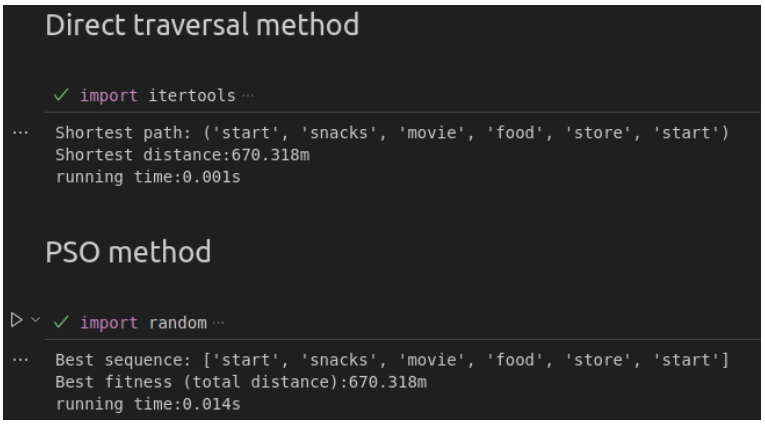

Figure 5: The route evaluation

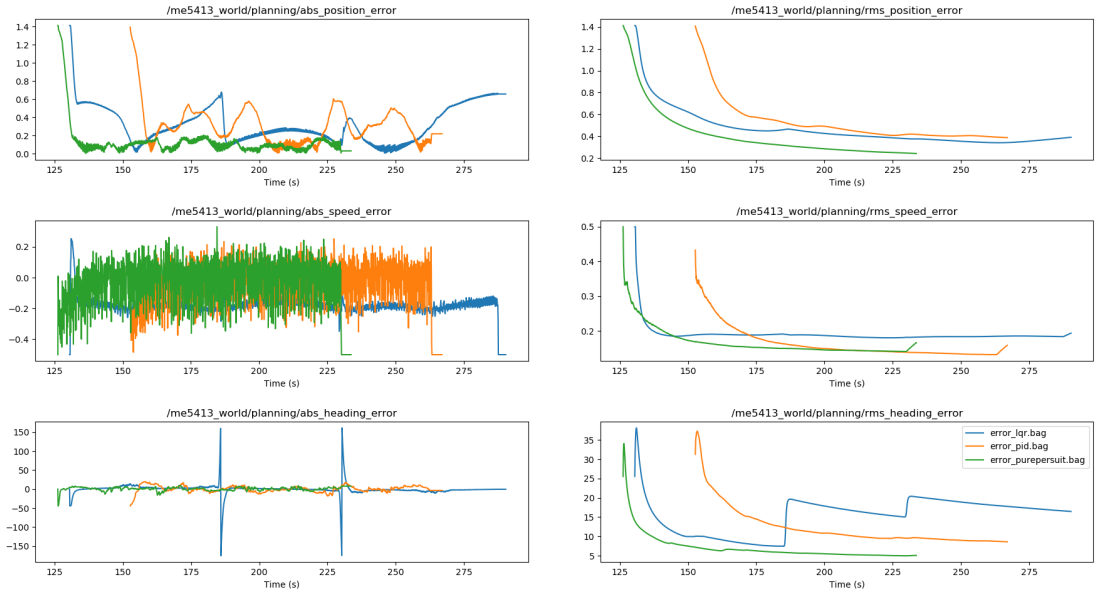

Figure 6: The evaluation of the tracing