ME5413 Report Homework 3: Planning (Group 1)

Zhao Yimin A0285282X
Chen Ruiyang A0285289J
Luo Yuanchun A0285275U

1 Task 1: Global Planning

1.1 Implementation details

In our quest for the most efficient route, we initially opted for Dijkstra's algorithm, revered for its comprehensive search capability that guarantees the optimal solution. However, the algorithm is not without drawbacks, chief among them being the high computational cost associated with its implementation. This is particularly pronounced when dealing with expansive graphs where the number of vertices and edges makes for a considerable processing load.

To address this concern, we are considering the adoption of the A* (A-star) algorithm, which strikes a compelling balance between finding an excellent solution and maintaining manageable implementation costs. The A* algorithm achieves this balance by employing heuristics to guide the search—effectively narrowing the search space and reducing the number of computations required.

Here's an overview of the implementation details for both Dijkstra's and A* algorithms:
Dijkstra's algorithm: The core idea of Dijkstra's algorithm is to find the shortest paths from a starting vertex to all other vertices in a weighted graph by calculating the "cost" of each path, where the cost is the length of the path from the start point to a given vertex. The **pseudocode for Dijkstra's Algorithm** is shown in the Appendix.
A* algorithm: This is an efficient pathfinding algorithm that combines the accuracy of Dijkstra's algorithm with the heuristic approach of Greedy Best First Search to optimize the search process. By managing the nodes to be explored with a priority queue and sorting them according to the cost estimation function f(x), the algorithm can effectively determine the shortest path from the start to the endpoint. This cost estimation function consists of two parts: g(x), which is the actual path cost from the starting point to the current node, and h(x), which is the estimated cost from the current node to the endpoint, serving as a heuristic assessment to help predict the remaining path cost. At each step, the A* algorithm always selects the node with the lowest f(x) value for exploration, thereby not only speeding up the process toward the goal but also ensuring the optimality of the path.
Fulfil footprint requirement According to the requirements, the planned path needs to maintain a circular footprint with a minimum radius of 0.3 meters. In other words, This means that adjacent grid cells to the planned path should remain unoccupied. Hence, an additional check is performed when inspecting the 8-connected neighbors to ensure they meet the footprint requirement. Nevertheless, this examination would significantly increase the running time of the program. One action to optimize the code is duplicating the map and marking examined cells to avoid redundant checks. This action could help achieve a reduction in runtime by over 50%.

1.2 Results

Tables 1 to 4 list the distances calculated using Dijkstra's algorithm and the A* algorithm paired with three different heuristic functions: Euclidean distance, Manhattan distance, and Chebyshev distance. Each table represents a distance matrix among nodes within a graph, where each row and column corresponds to a specific node (**Start,**

Snacks, Store, Movie, Food), and the intersections represent the calculated distances between these nodes.

Table 1: Distance (Dijkstra's Algorithm)

0	142.318	154.904	178.674	223.05
142.318	0	114.642	107.382	133.278
154.904	114.642	0	209.246	110.79
178.674	107.382	209.246	0	113.592
223.05	133.278	110.79	113.592	0

Table 2: Distance (A* with Euclidean dist.)

0	143.702	163.504	180.832	231.976
154.306	0	122.892	125.208	133.934
166.626	117.01	0	255.17	114.732
182.8	108.962	239.162	0	200.294
235.742	133.442	122.912	118	0

Table 3: Distance (A* with Manhattan dist.)

0	143.302	158.676	208.784	229.464
151.558	0	123.712	141.13	134.426
164.534	118.414	0	250.378	114.562
182.118	107.546	218.594	0	117.226
230.784	133.442	132.334	117.554	0

Table 4: Distance (A* with Chebyshev dist.)

0	144.206	168.878	179.618	233.986
144.324	0	114.642	110.332	167.476
158.326	125.526	0	226.098	112.796
178.674	109.624	224.266	0	117.604
229.658	133.396	112.876	124.668	0

Tables 5 to 8 list the number of cells visited by different algorithms in the process of searching for the target. These numbers can be considered as the cost of the search. Generally speaking, the larger the number, the more the algorithm has explored invalid points and the more time it has spent in the process of finding the target.

Table 5: Visited Cells (Dijkstra's Algorithm)

0	78938	92766	108876	159315
135713	0	82444	67912	118792
98215	61166	0	154989	59630
126333	60812	159318	0	63763
142391	102395	76680	80515	0

Table 6: Visited Cells (A* with Euclidean dist.)

0	20214	848	12231	29223
15169	0	532	4736	38007
1854	805	0	5340	556
9060	1561	1017	0	33195
67562	131263	2891	1409	0

Table 7: Visited Cells (A* with Manhattan dist.)

0	23599	84755	14330	133740
163850	0	582	6710	56187
39209	932	0	10103	1237
11114	2769	4611	0	2895
184828	248005	2887	6335	0

Table 8: Visited Cells (A* with Chebyshev dist.)

0	51964	91605	490473	29472
10721	0	720	532	45947
731	175439	0	3484	534
994	1556	2276	0	90170
32523	680503	519	3028	0

1.3 Discussion

(1) Comparison of Algorithm Efficiency:

- **Dijkstra's Algorithm**: Among various algorithms, Dijkstra's algorithm tends to visit a larger number of nodes, showcasing its exhaustive nature as a non-heuristic search method in pursuit of the shortest path. Although this algorithm ensures the discovery of the optimal path, its efficiency is somewhat lacking. As shown in **Table 9**, using Dijkstra's algorithm for pathfinding between two locations takes approximately 2 seconds.
- **A* Algorithm with Euclidean Distance Heuristic**: Significantly reduces the number of nodes that need to be visited, demonstrating that this heuristic can effectively guide the search direction and reduce futile attempts in many scenarios, thereby enhancing search efficiency. Compared to Table 9, **Table 10** illustrates that after introducing the A* algorithm, the search time is significantly reduced, with most searches completing in less than 0.5 seconds.
- **A* Algorithm with Manhattan Distance Heuristic**: Its performance varies across different scenarios. In certain contexts, such as pathfinding from a start point to specific targets, the number of nodes visited may be close to or slightly higher than that with the Euclidean distance, possibly reflecting the Manhattan distance's better suitability in certain graph structures.

- **A* Algorithm with Chebyshev Distance Heuristic**: In some situations, this combination has visited an exceptionally high number of nodes, especially in paths that require frequent diagonal movements, suggesting that the Chebyshev distance heuristic may not be sufficiently precise in these cases, leading to excessive exploration.

Table 9: Running Time (Dijkstra's Algorithm)

0	1.77755	1.99308	2.34490	3.47810
2.94939	0	1.83519	1.51041	2.61195
2.14185	1.33794	0	3.40929	1.30510
2.83562	1.41696	3.66124	0	1.49624
3.34755	2.42006	1.81199	1.88764	0

Table 10: Running Time (A* with Euclidean dist.)

0	0.35675	0.03559	0.27540	0.49952
0.29086	0	0.02254	0.09327	0.61786
0.05160	0.02772	0	0.11883	0.02289
0.16754	0.03787	0.04359	0	0.56579
1.08359	2.15654	0.05956	0.03797	0

(2) Efficiency and Accuracy: The A* algorithm combined with the Euclidean distance heuristic performs best in terms of both efficiency and path accuracy. It not only maintains the optimality of the path (comparable to the results of Dijkstra's algorithm) but also significantly reduces the search cost (by reducing the number of nodes visited).
(3) Choice of Heuristic Function: The selection of the heuristic function has a decisive impact on the performance of the A* algorithm. The Euclidean distance is a more universal choice because it performs well in various situations, while the Manhattan and Chebyshev distances may offer more advantages in specific scenarios.
(4) Applicability of Algorithms: Dijkstra's algorithm retains its value in processing small-scale graphs due to its comprehensiveness. However, in the case of large-scale graphs, heuristic search algorithms like the A* algorithm show higher efficiency, especially when the heuristic function is chosen correctly. The above analysis reveals that the choice of algorithms and heuristic functions requires careful consideration based on the characteristics and needs of the specific application scenario in order to achieve optimal search efficiency and path accuracy.

2 Task 2: The "Travelling Shopper" Problem

The "Travelling Shopper" problem, a variation of the classic Travelling Salesperson Problem (TSP), requires finding an optimal route that visits all specified locations and returns to the starting point.

Dynamic Programming Approach. To model this problem, we consider each location as a node in a graph and the distances between them as weighted edges. The optimal route minimizes the total distance traveled while visiting each node exactly once before returning to the start. To solve this problem, we propose using a dynamic programming approach that can exhaustively explore all possible routes and select the shortest one. Below is the **pseudocode**:

1: **for** each node $u \in$ nodes **do**

- distance$[u] \leftarrow \infty$
- predecessor$[u] \leftarrow$ null

2: distance[start] $\leftarrow 0$

3: **for** mask = 1 **to** $2^n - 1$ **do**

- **for** each u where $(1 \ll u)$ & mask $\neq 0$ **do**
- **for** each $v \neq u$ where $(1 \ll v)$ & mask $\neq 0$ **do**
- new_distance $\leftarrow$ distance$[v]$ + cost(v, u)
- **if** new_distance $<$ distance$[u]$ **then**

- distance[u] ← new_distance
- predecessor[u] ← v

4: path ← Reconstruct path using predecessor[]

5: total_distance ← Calculate total path distance

6: **return** path, total_distance

Permutation-Based Heuristic Approach. Following task 1, which involved finding the optimal route using Dijkstra's algorithm. To solve the Travelling Shopper Problem was based on generating permutations of the sequence of locations, excluding the starting point. This approach considered every possible sequence in which the locations could be visited and calculated the total distance for each sequence, identifying the one with the minimum travel distance.

1: **for** each permutation of locations **excluding 'start' do**

- Calculate the total distance of the permutation, including the return to 'start.'
- Keep track of the minimum distance and route.

2: Identify the permutation with the minimum distance as the optimal route.

3: **return** the optimal route and the minimum distance.

In Task 1, we utilized Dijkstra's Algorithm to meticulously calculate the shortest distances between nodes, forming the foundational dataset for the subsequent approaches in Task 2. Both the heuristic method and dynamic programming (DP) leveraged these pre-calculated distances to ensure the integrity of their computations. As a result, each method arrived at an identical conclusion, pinpointing the minimum distance of **628.986 meters** and delineating an optimal route that commenced at **'start,' proceeded through 'snacks,' 'movie,' 'food,' 'store,' and circled back to 'start'**.

The heuristic method operates on the principle of permutation, exploring the gamut of possible node sequences to discover the most efficient path. This approach scales factorially with the number of nodes, thus described by the time complexity of O(n!).

Dynamic programming adopts an exhaustive search paradigm, meticulously evaluating all subsets of the node-set. Its time complexity stands at $O(n^2 \cdot 2^n)$, which suggests an exponential increase in computational effort as the number of nodes, n, escalates. Despite this, DP is heralded for its unwavering accuracy and the guaranteed procurement of the optimal solution. It systematically eliminates sub-optimal paths through a principled approach that synthesizes the shortest routes to subsets of nodes before culminating in the ultimate itinerary.

3 Bonus Task

3.1 Introduction: In this project, we developed and optimized a path-tracking system that makes the robot car move along a predetermined "8"-shaped trajectory. The system uses an integrated control scheme that includes a precise PID controller and a pure pursuit algorithm to control the robot's speed and heading.

3.2 Experimental Setup: In order to enable real-time control of important parameters while the path tracing task is running, a dynamic reconfiguration tool has also been added. These parameters include look_ahead_distance and yaw_error, which control the path-tracking flexibility of angular velocity. The PID control coefficients (PID_Kp, PID_Ki and PID_Kd) affect the accuracy of the line speed. Modification of parameters ensures the robot's ability to respond to trajectory changes.

3.3 Implementation Details: PID controller is used to gradually close the gap between the system's present and desired states by modifying the control inputs. They are deployed based on this difference. Proportional (P), integral (I), and derivative (D), the three parts of PID, cooperate as follows:

- Proportion (P) adjusts based on the current error with a response proportional to the error.
- Integral (I) focuses on the cumulative effect of errors, offsetting long-term steady-state deviations.
- Derivative (D) Monitors the rate of error change to predict trends and reduce system overshoot.

The application of **pure pursuit algorithm** in path tracking would help find an ideal target point in front of the robot's current position based on the preset forward distance and calculate the required heading angle adjustment accordingly. The algorithm includes the following steps:

- Target Point Selection: A target point is determined on the path based on the set look-ahead distance.
- Calculation of Steering Angle: The steering angle is calculated based on the geometric relationship between the robot and the target point.
- Control Command Generation: Turning instructions are generated from the calculated results to guide the movement of the robot.

The combined use of a PID controller and a pure pursuit algorithm forms a control strategy. This method uses a PID controller to fine-tune the speed to match the target speed and uses a pure pursuit algorithm to correct the course, ensuring that the robot can accurately follow complex paths while maintaining the set speed.

3.4 Error Analysis and Performance Evaluation: The performance of the path tracking system was evaluated by analyzing the root mean square error (RMSE). As shown in Figures 1 to 3 in the Appendix, specific analyses of heading error (rms_heading_error), position error (rms_position_error), and speed error (rms_speed_error) led to the following **conclusions:**

- Heading Error: The initial error revealed the challenges faced by the robot in adapting to the initial segment of the trajectory. However, over time, the control system successfully reduces the heading deviation, leading to rapid convergence and stabilization of the error at a low level.
- Position Error: The trend of position error reduction was similar to that of heading error, with initially high values quickly reducing and maintaining at a low level, keeping the robot on the pre-set trajectory.
- Speed Error: Initial speed error fluctuations reflect the PID controller's sensitive response to speed changes. Despite slight fluctuations in the mid-term, errors remained low, indicating that the robot was able to travel at a speed close to its target.

As shown in Figures 1 to 3 in the Appendix, the designed PID and Pure Pursuit control strategies successfully guided the robot along the pre-determined '8' figure path. Despite facing initial errors, the control system was able to quickly adjust and maintain a low error state, confirming its efficiency and robustness. **The final parameters obtained were: speed_target= 0.5 m/s, PID_Kp= 0.5, PID_Ki= 0.2, PID_Kd= 0.2, look_ahead_distance= 1.5m, ensuring the robot's precise traversal along the complex trajectory.**

3.5 GitHub link of Bonus Task: https://github.com/ztony0712/ME5413_Homework3_bonus

4 Appendix

Pseudocode for Dijkstra's Algorithm:

1: Notation: Graph G, Vertex set V, Edge cost function $\text{cost}(u, v)$, Start vertex start, Goal vertex goal, Array distance[], Array predecessor[]

2: Initialize:

- distance$[v] \leftarrow \infty$ for all $v \in V$
- distance[start] $\leftarrow 0$
- $Q \leftarrow V$ sorted by distance values

3: While Q is not empty, do

a. $u \leftarrow \text{ExtractMin}(Q)$

b. If u = goal then
   - break

c. For each neighbor v of u do
   - new_distance $\leftarrow$ distance$[u] + \text{cost}(u, v)$
   - If new_distance $<$ distance$[v]$ then
     - distance$[v] \leftarrow$ new_distance
     - predecessor$[v] \leftarrow u$
     - $\text{DecreaseKey}(Q, v, \text{new\_distance})$

d. $Q \leftarrow Q \setminus \{u\}$

4: Reconstruct path from goal to start using predecessor array

5: Output the shortest path and its length

- return predecessor[], distance[]

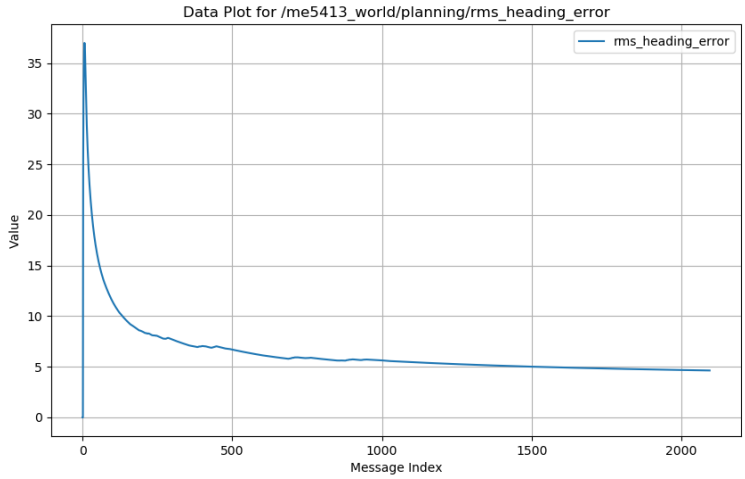

Figure 1: RMS Heading Error

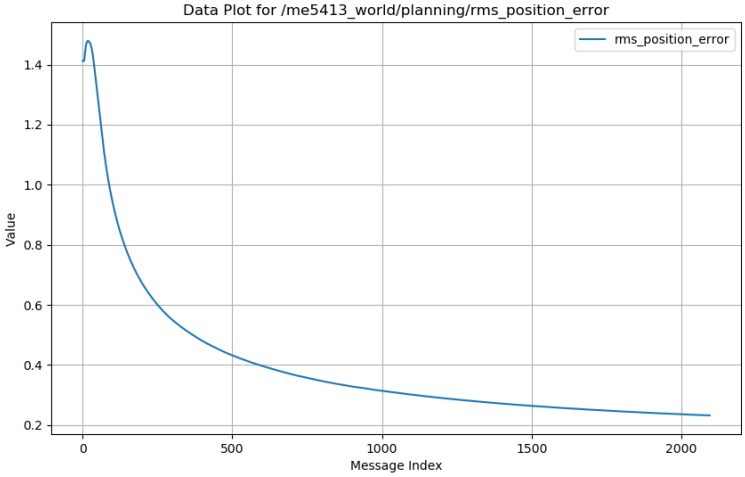

Figure 2: RMS Position Error

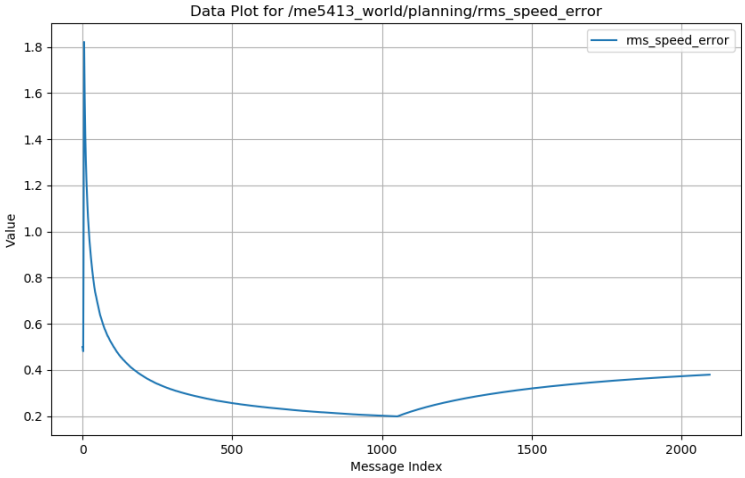

Figure 3: RMS Speed Error