ME5413 Homework 3: Planning

Cao Chenyu
Mechanical Engineering
National University of Singapore
Singapore, Singapore
A0285295N

Li Zhangjin
Mechanical Engineering
National University of Singapore
Singapore, Singapore
A0285091B

Zhao Xu
Mechanical Engineering
National University of Singapore
Singapore, Singapore
A0285284U

***Abstract*— This report presents our group's work on implementing planning algorithms for autonomous mobile robots. We applied the A* algorithm and its variants for global path planning on a map of VivoCity Level 2. The Traveling Shopper Problem of finding the optimal route to visit multiple locations was modeled and solved using two approaches. As a bonus, a path tracking controller was developed to follow a figure-8 reference trajectory. The methodology, results, and insights gained from each task are discussed.**

***Index terms*—Path planning, A* algorithm, Traveling Salesman Problem, Path tracking control**

I. Introduction

Planning algorithms are a fundamental component of autonomous mobile robots, enabling them to find efficient paths in complex environments and optimize routes for multi-goal missions.[1] In this homework, we implemented and analyzed several key planning techniques, including the A*, Dijkstra, and Greedy BFS algorithms for global path planning, solutions to the Traveling Salesman Problem (TSP) for multi-goal path optimization, and path tracking controllers for precise trajectory following.

II. Task 1: Graph Search Algorithms

A. Problem Formulation

The first task involved implementing the A* algorithm [2] and its variants (Dijkstra [3], and Greedy BFS [4]) to find the shortest path between different locations on a map of VivoCity Level 2. The goal was to compute the distances, cells visited, and computation time for each algorithm, comparing their performance and optimality.

B. Methodology

We implemented the A* algorithm with the following key components and extensions:

- 8-connected neighborhood with 0.2m (straight) and 0.282m (diagonal) step costs
- Euclidean distance heuristic function
- Try switch the start and goal positions to get the whole path maps
- Dijkstra's algorithm (A* with zero heuristic)
- Greedy Best-First Search (A* with heuristic only)

C. Implementation Details

1) *Base Planner Class:*

The `Planner` clase serves as the base class for all search algorithms. It initializes rhe planner with the obstacle map, grid resolution, and rabot radius. The class also defines the `Node` inner class, which represents a node in the search space with attributes such as position, cost, and parent indes.

The `get_motion_model` method defines the robot's movement model, allowing it to move to any of the 8 surrounding cells in each step. The `calc_position`, `calc_xy_index`, and `calc_index` methods are utility functions for converting between grid indices and actual positions. The `verify_node` method checks if a node is valid by ensuring it is sithin the map boundaries and not too close to obstacles based on the robot's radius. It uses a brute-force approach to check the surrounding cells within the robot's radius. The `calc_final_path` method method reconstructs the final path by tracing back the parent indices from the goal node to the start node. The `calculate_total_distance` method calculates the total distance of the path based on the grid resolution.

2) *A* Algorithm:*

The `AStar` class inherits from the `Planner` class and implements the A* algorithm. It uses the `Heuristic` class to calculate the heuristic value, which estimates the cost from a node to the goal. The default heuristic is the Euclidean distance, and Manhattan distance is also provided as an option.

The `planning_astar` method performs the A* path planning. It maintains an open set of nodes to be explored and a closed set of visited nodes. The algorithm starts with the start node and iteratively expands the node with the lowest f-score (cost + heuristic) until the goal node is reached or the open set is empty.

At each iteration, the current node is removed from the open set and added to the closed set. The algorithm generates the neighbors of the current node using the motion model and calculates their costs and heuristic values. If a neighbor is not in the closed set and is valid, it is added to the open set or updated if a better path is found. The algorithm continues until the goal is reached or no path is found. Finally, the `calc_final_path method` is called to reconstruct the optimal path.

3) *Dijkstra's Algorithm:*

The `Dijkstra` class implements Dijkstra's algorithm, which is a special case of A* with a zero heuristic. The `planning_dijkstra` method is similar to the planning_astar method but only considers the cost (g-score) when selecting the next node to expand.

Dijkstra's algorithm guarantees the optimal path but may explore more nodes than A* due to the lack of a heuristic to guide the search.

4) *Greedy Best-First Search Algorithm:*

The `GBFS` class implements the Greedy Best-First Search algorithm, which is a variant of A* that only considers the heuristic value (h-score) when selecting the next node to expand. The `planning_gbfs` method is similar to the `planning_astar` method but uses the heuristic value as the sole criterion for node selection.

GBFS can find a suboptimal solution quickly but may get stuck in local minima. It does not guarantee the optimal path but can be faster than A* in some cases.

5) *Relationship and Comparison of the A* Variants:*

All three algorithms use the same `calc_final_path` method to reconstruct the path and the `calculate_total_distance` method to compute the total path distance. The main function loads the obstacle map, sets the grid size and robot radius, creates instances of the planners, and calls the plot_all_paths function to visualize the planned paths between all pairs of locations using the respective algorithms.

Table 1 summarizes the key differences between A*, Dijkstra, and Greedy BFS in terms of node selection criterion, optimality, and completeness.

Algorithm	**Node Selection Criterion**	**Optimality**	**Completeness**
A*	Cost + Heuristic	Yes	Yes
Dijkstra	Cost	Yes	Yes
GBFS	Heuristic	No	No

Table 1: Comparison of A* Variants

D. Results and Discusion

1) *Shortest Path Visualization:*

Figure 1 shows the shortest paths between each pair of locations computed by the A* algorithm. The paths are visualized on the map of VivoCity Level 2, highlighting the optimal routes from the different starting locations (start, snacks, store, movie, food) to the other locations.

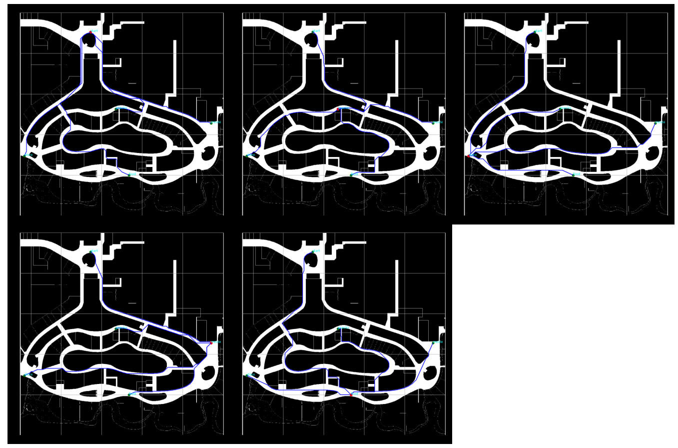

Figure 1: Shortest paths between locations using A* algorithm

Table 2 summarizes the distances between each pair of locations computed by our A* planner (or Dijkstra planner), which is the optimal path length for each pair of locations. The results show that the distances vary depending on the locations and the obstacles in the map, so it could be further used to be the basis for the TSP problem in Task 2.

	start	**snacks**	**store**	**movie**	**food**
start	0.0	141.97	154.66	178.66	221.77
snacks	141.97	0.0	114.56	106.94	132.68
store	154.66	114.56	0.0	209.09	110.87
movie	178.66	106.94	209.09	0.0	113.72
food	221.77	132.68	110.87	113.72	0.0

Table 2: The distances between each pair of locations

2) *Comparison of The A* Variants:*

We have also compared the performance of the A*, Dijkstra, and Greedy BFS algorithms in terms of the paths, nodes visited, and computation time when we fixed the start and goal positions to be the start and food locations. The results are summarized in Table 3.

From the results, we can observe the following key points:

- For each goal position, A* and Dijkstra find the same optimal path length, while GBFS finds a suboptimal path with a slightly longer distance.
- A* visits fewer nodes than Dijkstra, indicating the effectiveness of the heuristic in guiding the search.
- A* has a longer computation time than Dijkstra due to the additional heuristic computation (Euclidean distance).
- GBFS visits the fewest nodes and has the shortest computation time but sacrifices optimality for speed.
- As the goal position gets farther from the start (in the order of snacks, store, movie, food), the path length, nodes visited, and computation time generally increase for all

algorithms. This is because the search space expands as the goal gets farther away.

Algorithm	Goal Position	Path Length (m)	Computation Time (s)	Nodes Visited
A*	snacks	**141.97**	7.19	39161
Dijkstra		**141.97**	3.50	76048
GBFS		161.38	**0.42**	**4580**
A*	store	**154.66**	10.43	31528
Dijkstra		**154.66**	4.08	90099
GBFS		163.30	**0.07**	**740**
A*	movie	**178.66**	6.38	34370
Dijkstra		**178.66**	5.00	107307
GBFS		183.61	**0.16**	**1999**
A*	food	**221.77**	18.90	85380
Dijkstra		**221.77**	7.61	159774
GBFS		241.05	**0.48**	**4094**

Table 3: Comparison of A* Variants when start is fixed to `start`

III. Task 2: The "Travelling Shopper" Problem

A. Introduction

The Traveling Salesman Problem (TSP) is a well-known optimization problem in computer science and operations research. Given a set of cities and the distances between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city [5]. In this task, we implemented and compared three algorithms to solve the TSP: Brute Force, Dynamic Programming, and Genetic Algorithm.

B. Methodology

1) *Brute Force:* The Brute Force algorithm generates all possible permutations of the cities and calculates the total distance for each permutation. It then selects the permutation with the minimum total distance as the optimal solution. The time complexity of this approach is $O(n!)$, where n is the number of cities [6].

2) *Dynamic Programming:* The Dynamic Programming algorithm uses a bottom-up approach to solve the TSP. It builds a table of subproblems, where each subproblem represents the shortest path for a subset of cities ending at a particular city. The algorithm fills the table iteratively and constructs the optimal solution by backtracking from the final subproblem [7]. The time complexity of this approach is $O(2^n \times n^2)$.

3) *Genetic Algorithm:* The Genetic Algorithm is a metaheuristic approach inspired by the process of natural selection. It starts with a population of randomly generated solutions and evolves them over multiple generations using selection, crossover, and mutation operations. The fitness of each solution is evaluated based on the total distance of the TSP tour. The algorithm continues until a termination condition is met, such as reaching a maximum number of generations or finding a satisfactory solution [8].

C. Results and Discussion

Table 4 compares the execution time and total distance obtained by each TSP algorithm. The results show that all three algorithms found the optimal path with a total distance of 628.17m. However, there are notable differences in their execution times.

The Dynamic Programming (DP) algorithm was the fastest, with an execution time of 1.11×10^{-4} clock cycles. This efficiency can be attributed to the relatively small number of nodes in the TSP problem, which allows the DP algorithm to solve it effectively.

The Brute Force (BF) algorithm had the second-fastest execution time of 9.77×10^{-4} clock cycles. Although it guarantees finding the optimal solution by exhaustively exploring all possible paths, its execution time is slightly longer than the DP algorithm due to its exponential time complexity.

The Genetic Algorithm (GA) had the longest execution time of 6.77×10^{-2} clock cycles. This is because the GA requires more time to converge towards the optimal solution through an iterative process of selection, crossover, and mutation. Despite its longer execution time, the GA still managed to find the optimal path in this specific TSP problem.

These results demonstrate the trade-offs between execution time and solution quality among the three algorithms. While the DP algorithm excels in solving small-scale TSP problems efficiently, the BF algorithm provides a reliable approach for finding the optimal solution, albeit with a slightly longer execution time. The GA, on the other hand, offers a balance between exploration and exploitation, making it suitable for larger and more complex TSP instances where finding the exact optimal solution may be computationally expensive.

Algorithm	Execution Time (clock cycles)	Total Distance (m)
BF	9.77×10^{-4}	628.17
DP	$\mathbf{1.11 \times 10^{-4}}$	
GA	6.77×10^{-2}	

Table 4: Comparison of TSP Algorithms

Figure 2 visualizes the optimal TSP path found by the algorithms. The path starts from the `start` location, visits all other locations exactly once, and returns to the `start` location. (start $\rightarrow$ store $\rightarrow$ food $\rightarrow$ movie $\rightarrow$ snacks $\rightarrow$ start)

Figure 2: Optimal TSP path found by the algorithms

IV. Task 3 (Bonus): Path Tracking

A. Introduction

The goal of this task was to control the robot to follow a given figure-8 track. We were provided with a template code that included a PID controller for throttle and a Stanley controller for steering. However, both controllers were not properly configured or tuned. In this report, we present our approach to improving the path tracking performance.

Github Repo: ME5413_Planning_Project_Group20

B. Methodology

1) *Orginal: PID Stanley Controller:* We started by dynamically tuning the parameters of the original PID Stanley controller to improve tracking performance. We adjusted the PID gains and to achieve better tracking results. The dynamic reconfigure GUI provided in the template code allowed us to tune these parameters in real-time.

The Stanley controller uses the cross-track error (CTE) and heading error to calculate the steering angle. The PID controller adjusts the throttle based on the speed error to maintain the desired speed. The intuitive control law comes as,

$$\delta(t) = \theta_{e(t)} + \tan^{-1}\left(k\frac{e_{\text{fa}(t)}}{v(t)}\right) \tag{1}$$

Where,

θ_e = Heading error (rad)
e_{fa} = Lateral offset (m)
δ = Steering angle (rad)
k = Stanley Gain
c_x, c_y = Nearest point in path
v = Velocity of vehicle (m/s)

2) *Improvment of Velocity-based Stanley Gain Adjustment:* To further improve the given Stanley method, we adjusted the `stanley_k` parameter based on the robot's current velocity. This adjustment allows the robot to have a higher Stanley gain at lower speeds for better tracking accuracy and a lower gain at higher speeds for smoother steering.

3) *PID + Pure Pursuit Controller:*

The final control strategy implemented in this work is a combination of the PID and Pure Pursuit methods, inspired by [9]. The Pure Pursuit algorithm calculates the goal pose by identifying the closest path point to the robot that is a fixed distance ahead, defined as the `purePursuit_DistanceAhead` parameter. This concept is illustrated in detail in Figure 3 [10]. The computed goal pose serves as the target for the PID controller, which generates the appropriate throttle and steering commands to guide the robot towards the desired path.

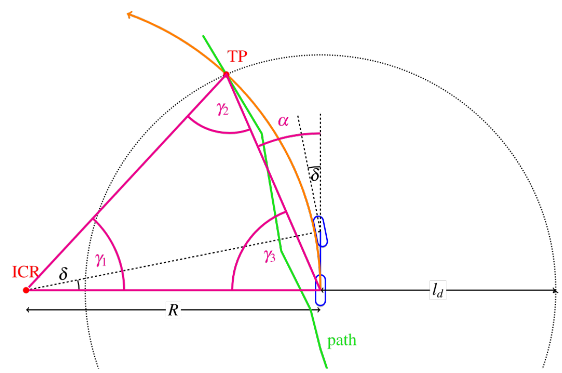

Figure 3: Pure Pursuit Method

C. Results and Discussion

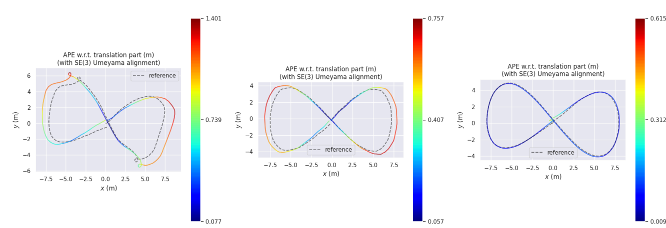

Figure 4: Path Tracking Visualization (Original PID Stanley vs. Improved PID Stanley vs. PID + Pure Pursuit)

Figure 4 shows the path tracking results of the original PID Stanley controller and the PID + Pure Pursuit controller. The visualization demonstrates the improved tracking performance of the PID + Pure Pursuit method, which follows the figure-8 trajectory more accurately.

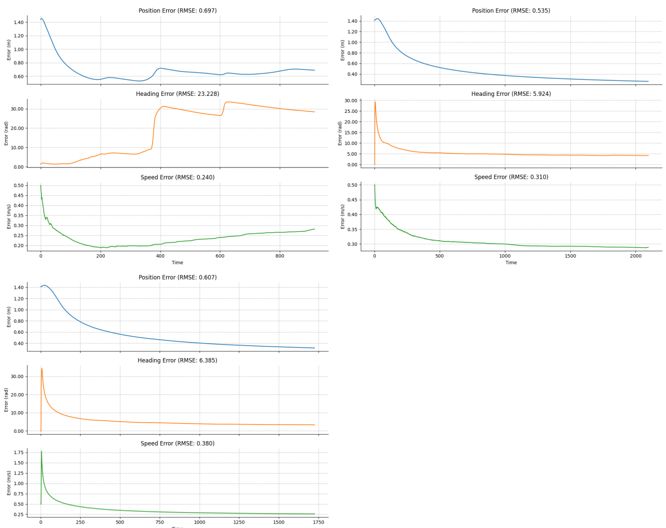

Figure 5: Errors of Path Tracking Methods (Original PID Stanley vs. Improved PID Stanley vs. PID + Pure Pursuit)

Method	RMS Position Error (m)	RMS Heading Error (deg)	RMS Speed Error (m/s)
Original PID Stanley	0.697	23.23	0.240
Improved PID Stanley	**0.535**	**5.924**	**0.310**
PID Pure Pursuit	0.607	6.385	0.380

Table 5: Comparison of Path Tracking Methods (RMS Errors)

Table 5 summarizes the root mean square (RMS) errors for position, heading, and speed obtained by the original PID Stanley controller, the improved Stanley controller, and the PID + Pure Pursuit controller. The results show that the improved PID Stanley method achieved the best tracking accuracy, with significantly reduced RMS errors compared to the original implementation.

D. Conclusion

In this task, we successfully improved the path tracking performance of the robot by dynamically tuning the parameters of the PID Stanley controller, adjusting the Stanley gain based on velocity, and implementing a combination of PID and Pure Pursuit methods. The results show that the PID + Pure Pursuit approach achieved the best tracking accuracy, with significantly reduced RMS errors compared to the original implementation. Future work could explore more advanced path tracking algorithms and further optimization of the controller parameters.

References

[1] S. M. LaValle, *Planning algorithms*. Cambridge university press, 2006.

[2] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," in *IEEE transactions on Systems Science and Cybernetics*, 1968, pp. 100–107.

[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[4] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.

[5] W. J. Cook, D. L. Applegate, R. E. Bixby, and V. Chvátal, *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[7] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *Journal of the ACM (JACM)*, vol. 9, no. 1, pp. 61–63, 1962.

[8] K. Deb, "Genetic algorithm in search and optimization: the technique and applications," in *Proceedings of International Workshop on Soft Computing and Intelligent Systems,(ISI, Calcutta, India)*, 1998, pp. 58–87.

[9] R. C. Coulter and others, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[10] M. Althoff, "Pure Pursuit Controller." 2023.