ME5418 - Machine Learning in Robotics

National University of Singapore

College of Design and Engineering

Homework 3: Planning

Author:
Wu Rui (ID: A0304636H)

Date: April 3, 2025

Contents

1 Task 1: Global Path Planning

1.1 Algorithm Principles

This section implements and compares three commonly used path planning algorithms:

1.1.1 A* Algorithm

The A* algorithm is a heuristic search algorithm that combines the ”shortest path first” of Dijkstra’s algorithm and the ”goal-oriented” feature of greedy best-first search.
Evaluation function: $f(n) = g(n) + h(n)$
$g(n)$: The actual cost from the start to node n
$h(n)$: The estimated cost from node n to the goal (heuristic function)
Characteristics: A* guarantees an optimal solution when $h(n)$ is an admissible heuristic (i.e., it does not overestimate the actual cost).

1.1.2 Dijkstra’s Algorithm

Dijkstra’s algorithm is a graph search algorithm used to calculate the shortest path from a single source.
Evaluation function: $f(n) = g(n)$, which only considers the cost from the start to the current node.
Characteristics: This is equivalent to the A* algorithm with the heuristic function $h(n) = 0$, and it guarantees the shortest path.

1.1.3 Greedy Best-First Search

Greedy Best-First Search is a purely heuristic algorithm that only considers the estimated values.
Evaluation function: $f(n) = h(n)$, which only considers the estimated cost from the current node to the goal.
Characteristics: It is fast to compute but does not guarantee an optimal solution.

1.2 Algorithm Implementation Details

1.2.1 Common Implementation Framework

All three algorithms use the same basic framework:

```
neighbors = [(0,1),(0,-1),(1,0),(-1,0),(-1,1),(1,1),(-1,-1)
    ,(1,-1)]
neighbor_costs = [0.2,0.2,0.2,0.2,0.282,0.282,0.282,0.282]
open_set = []   # Use priority queue to manage the open set
closed_set = set()
```

1.2.2 A* Algorithm Specific Implementation

A* algorithm's key feature is the evaluation function $f(n) = g(n) + h(n)$:

```
# Heuristic function definition
def heuristic(a, b):
    return map_resolution * np.sqrt((a[0] - b[0])**2 + (a[1] -
       b[1])**2)

# Initialize open set
heapq.heappush(open_set, (heuristic(start_node, goal_node), 0,
   start_node))

# Update neighbor nodes
tentative_g = g_score[current] + neighbor_costs[i]
f_score = tentative_g + heuristic(neighbor, goal_node)
heapq.heappush(open_set, (f_score, tentative_g, neighbor))
```

1.2.3 Dijkstra's Algorithm Specific Implementation

Dijkstra's algorithm does not use a heuristic function:

```
# Priority based only on g value
heapq.heappush(open_set, (0, 0, start_node))

# Update neighbor nodes
tentative_g = g_score[current] + neighbor_costs[i]
priority = tentative_g  # No heuristic value added
heapq.heappush(open_set, (priority, tentative_g, neighbor))
```

1.2.4 Greedy Best-First Search Specific Implementation

Greedy Best-First Search algorithm uses only the heuristic function to decide the priority:

```
# Priority based only on the heuristic value
heapq.heappush(open_set, (heuristic(start_node, goal_node), 0,
   start_node))

# Update neighbor nodes
tentative_g = g_score[current] + neighbor_costs[i]
priority = heuristic(neighbor, goal_node)
# Priority based only on heuristic value
heapq.heappush(open_set, (priority, tentative_g, neighbor))
```

1.2.5 Algorithm Difference Analysis

The main differences between the three algorithms lie in the composition of their evaluation functions:

Search Behavior Differences:

Algorithm	Evaluation Function	Advantages	Disadvantages
A*	$f(n) = g(n) + h(n)$	Balances efficiency and optimality	Dependent on heuristic function
Dijkstra	$f(n) = g(n)$	Guarantees optimal solution	Explores many nodes
Greedy Best-First Search	$f(n) = h(n)$	Fast computation	Does not guarantee optimal solution

Table 1: Comparison of Three Path Planning Algorithms

- **Dijkstra's Algorithm:** Expands outward uniformly in a ”wave-like” manner.
- **Greedy Best-First Search:** Chases the goal in a straight line, ignoring potential obstacles.
- **A* Algorithm:** Balances the two behaviors, leaning towards the goal but also considering the path cost.

1.3 Experimental Results

1.3.1 Performance Comparison

Algorithm	Computation Time (seconds)	Number of Nodes Visited	Path Length (meters)	Shortest Path
A*	26.7399	105514	470.86	start → snacks → movie → food → store
Dijkstra	51.8386	315695	470.86	start → snacks → movie → food → store
Greedy Best-First	14.6835	7261	505.58	start → snacks → store → food → movie

Table 2: Task 1: Performance Comparison of Open TSP Algorithms

The table in Appendix I records the shortest distances between each pair of locations.

1.3.2 Path Visualization

The exploration patterns and final paths of the algorithms show significant differences:

- **A* Algorithm:** Explores towards the goal direction but still considers multiple potential paths.
- **Dijkstra's Algorithm:** Expands uniformly in all directions until the goal is found.
- **Greedy Algorithm:** Directly explores towards the goal direction, visiting the fewest cells.

The corresponding figures for the search behaviors of each algorithm are provided in Appendix II.

1.4 Challenges and Solutions during the Experiment

1.4.1 Handling Diagonal Movement

Challenge: Diagonal movement may pass through obstacles.
Solution: Check if the adjacent cells on either side of the diagonal are free space.

```
if i >= 4:  # Diagonal movement
    if (grid_map[current[0] + dx, current[1]] == 0 or
        grid_map[current[0], current[1] + dy] == 0):
        continue
```

1.4.2 Path Reconstruction

Challenge: After finding the target, A* needs to reconstruct the full path.
Solution: Maintain a record of parent nodes and trace back to reconstruct the path.

2 Task 2: Traveling Salesman Problem

2.1 Problem Modeling

Task 2 is a classical Traveling Salesman Problem (TSP): starting from an initial point, visiting all four locations exactly once, and returning to the starting point while minimizing the total travel distance.

2.1.1 Mathematical Definition

Given a complete weighted graph $G = (V, E)$, where:

- $V = \{v_0, v_1, \dots, v_n\}$ is the set of vertices (locations),
- E is the set of edges (paths),
- $d(v_i, v_j)$ represents the distance between v_i and v_j.

The objective is to find a closed-loop path $\pi = (v_0, v_{\pi(1)}, \dots, v_{\pi(n)}, v_0)$ that minimizes the total distance:

$$\min_{\pi} \sum_{i=0}^{n} d\left(v_{\pi(i)}, v_{\pi((i+1) \mod (n+1))}\right)$$

2.1.2 Solution Strategies

Considering the problem size (5 locations), I implemented two approaches:

- **Brute-force Enumeration**: Enumerates all possible paths (4! = 24 possibilities), computes the total distance for each path, and selects the shortest one.
- **Nearest Neighbor Algorithm**: A greedy approach that starts from the initial point, iteratively selects the nearest unvisited node, and finally returns to the starting point.

2.2 Method implementation

Due to space limitations, the specific code implementation of the brute force method and the nearest neighbor algorithm can be found in the Appendix II.

Method	Total Distance (m)	Computation Time (ms)	Path
Brute-force Enumeration	669.29	1.325	start → store → food → movie → snacks → start
Nearest Neighbor Algorithm	700.60	0.019	start → snacks → store → food → movie → start

Table 3: Comparison of TSP Algorithms in Task 2

2.3 Results Comparison

Table3 shows the comparison between two algorithms' results while using GBFS algorithm in task1.

2.4 Observations and Findings

2.4.1 Method Comparison

- The brute-force method guarantees finding the optimal solution, with a total distance of 669.29 meters.
- The nearest neighbor algorithm produces a slightly suboptimal solution with a distance of 700.60 meters (approximately 4.7% longer).
- Due to the small problem size (5 locations), the computation time difference between the two methods is negligible.

2.4.2 Algorithm Characteristics

- The brute-force method is suitable for small-scale problems but has a complexity of $O(n!)$.
- The nearest neighbor algorithm is computationally efficient ($O(n^2)$) but does not guarantee an optimal solution.
- In this case, the brute-force method's result is very close to the optimal solution.

2.4.3 Path Analysis

- The brute-force method's path is: **start** → **store** → **food** → **movie** → **snacks** → **start**.
- The nearest neighbor algorithm's path is: **start** → **snacks** → **store** → **food** → **movie** → **start**.
- The difference between the two solutions is only in the order of deciding snacks.

2.5 Final Shortest Path

The final shortest path is obtained using the brute-force enumeration method:

- **Path**: start → store → food → movie → snacks → start
- **Total Distance**: 669.29 meters

A Appendix I: The shortest distances between each pair of locations

Distance	start	snacks	store	movie	food
start	0.00	141.35	154.55	178.44	219.34
snacks	141.35	0.00	114.41	106.65	130.71
store	154.55	114.41	0.00	208.47	110.79
movie	178.44	106.65	208.47	0.00	112.06
food	219.34	130.71	110.79	112.06	0.00

Table 4: Shortest Distance Matrix under A* Algorithm (Unit: meters)

Distance	start	snacks	store	movie	food
start	0.00	141.35	154.55	178.44	219.34
snacks	141.35	0.00	114.41	106.65	130.71
store	154.55	114.41	0.00	208.47	110.79
movie	178.44	106.65	208.47	0.00	112.06
food	219.34	130.71	110.79	112.06	0.00

Table 5: Shortest Distance Matrix under Dijkstra Algorithm (Unit: meters)

Distance	start	snacks	store	movie	food
start	0.00	145.56	162.99	183.61	236.91
snacks	160.76	0.00	122.56	142.01	137.65
store	173.44	120.46	0.00	255.02	118.25
movie	195.02	108.09	241.26	0.00	204.71
food	249.04	182.14	131.66	119.20	0.00

Table 6: Shortest Distance Matrix under GBFS Algorithm (Unit: meters)

B Appendix II: Visualization of the most efficient route in task1

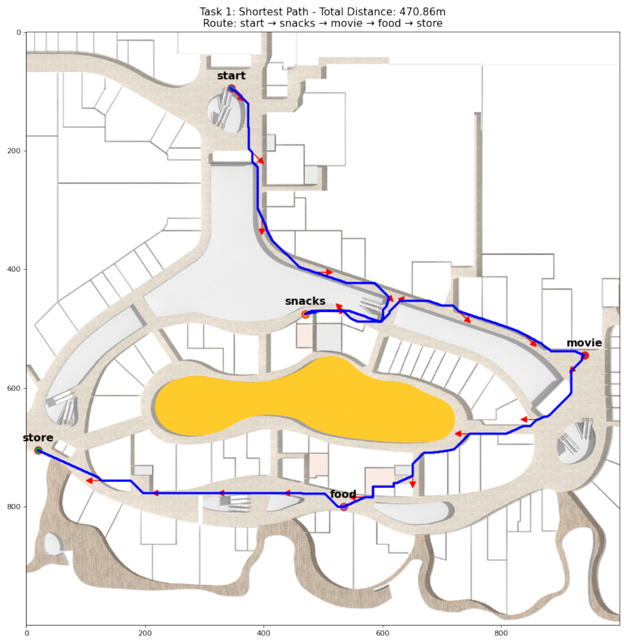

Figure 1: The most efficient route in task1 using A* algorithm

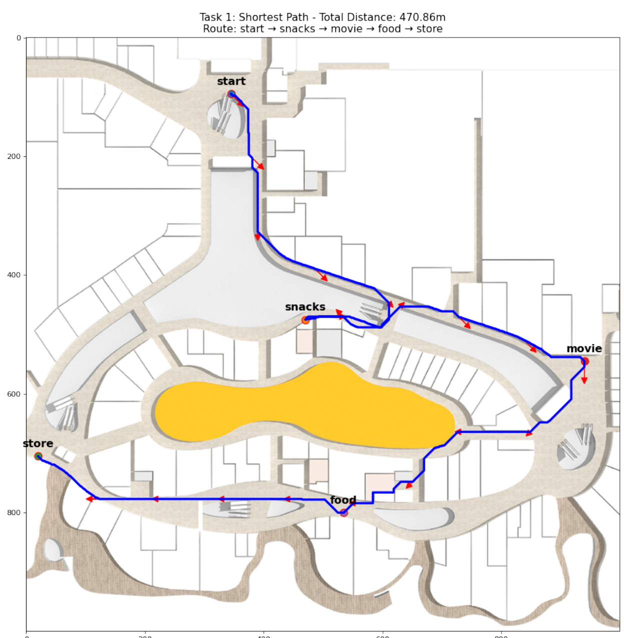

Figure 2: The most efficient route in task1 using Dijkstra algorithm

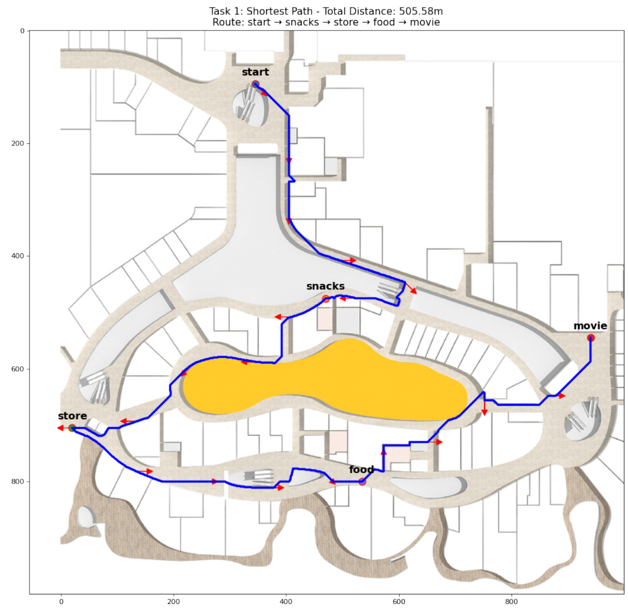

Figure 3: The most efficient route in task1 using GBFS algorithm

C Appendix III: TSP Algorithms

C.1 Brute Force Algorithm for TSP

```
def solve_tsp_bruteforce_task2(distance_matrix, start_index=0):
    n = distance_matrix.shape[0]
    other_indices = [i for i in range(n) if i != start_index]
    best_route = None
    best_distance = float('inf')

    # Enumerate all permutations
    for perm in itertools.permutations(other_indices):
        route = [start_index] + list(perm) + [start_index]
        distance = sum(distance_matrix[route[j], route[j+1]]
                       for j in range(len(route) - 1))

        if distance < best_distance:
            best_distance = distance
            best_route = route

    return best_route, best_distance
```

C.2 Nearest Neighbor Algorithm for TSP

```
def solve_tsp_nearest_neighbor_task2(distance_matrix,
   start_index=0):
    n = distance_matrix.shape[0]
    unvisited = set(range(n))
    unvisited.remove(start_index)

    route = [start_index]
    current = start_index
    total_distance = 0

    # Greedy selection of the nearest node
    while unvisited:
        next_point = min(unvisited, key=lambda x:
           distance_matrix[current, x])
        route.append(next_point)
        total_distance += distance_matrix[current, next_point]
        unvisited.remove(next_point)
        current = next_point
```

```
    # Return to start
    route.append(start_index)
    total_distance += distance_matrix[current, start_index]

    return route, total_distance
```

D Appendix IV: Visualization of the most efficient route in task2

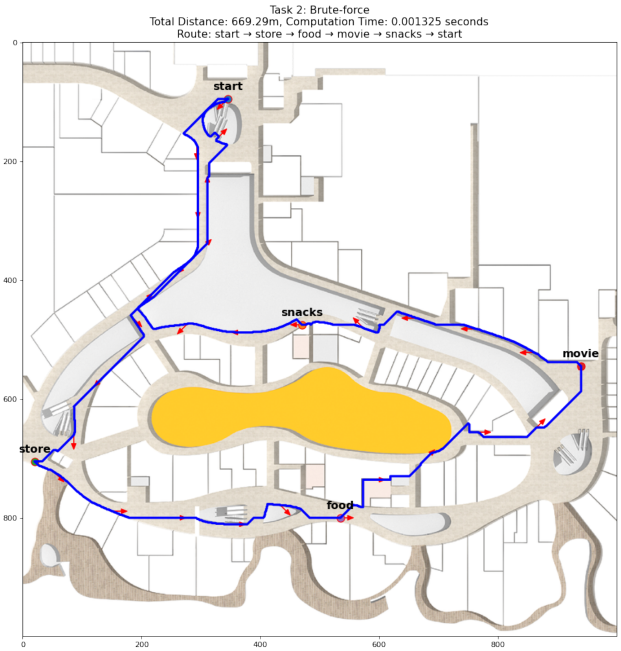

Figure 4: The most efficient route in task2 using Brute Force Algorithm

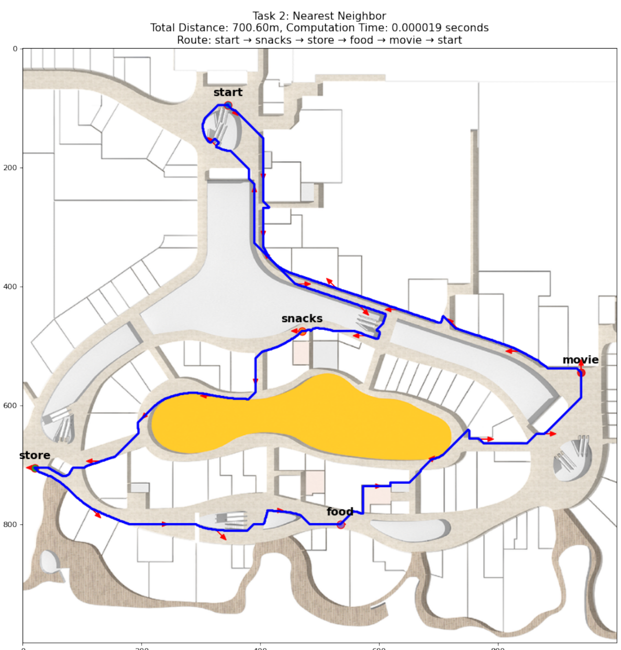

Figure 5: The most efficient route in task2 using The nearest neighbor Algorithm

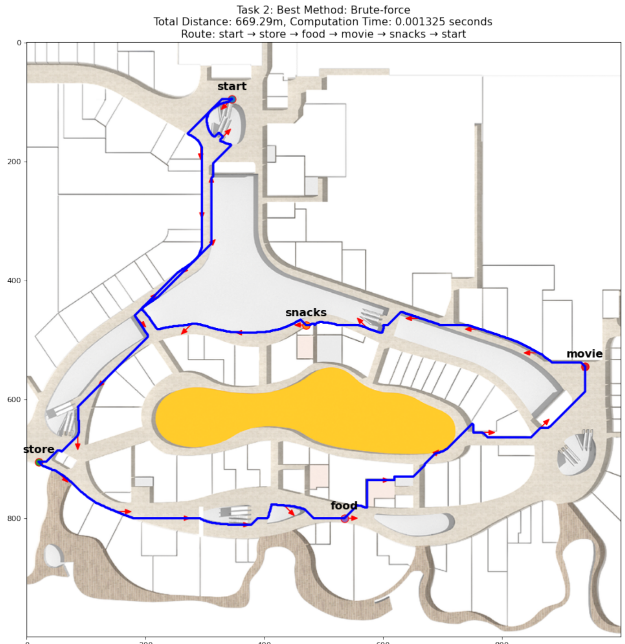

Figure 6: The most efficient route in task2 (Brute Force Algorithm)