National University of Singapore

ME5413 Autonomous Mobile Robot

AY24/25 Semester 2

Homework 3: Planning

NI HAOZHAN (A0305097A)

Contents

1 | Task 1: Implement A^* Planning Algorithm

The A^* (A-Star) algorithm is a heuristic search algorithm widely used in path planning and graph search problems. A^* combines the shortest path search of Dijkstra's algorithm with the heuristic guidance of Greedy Best First Search, enabling efficient computation of the optimal path from the start to the goal. A^* algorithm sorts the frontier cells in the priority queue based on the sum of cost-so-far $g(n)$ and cost-to-go $h(n)$: $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost from the start node to the current node n, $h(n)$ is the heuristic estimate of the cost from the current node to the goal node, $f(n)$ is the total estimated cost. At each iteration, A^* always picks the frontier with the minimal $f(x)$.

1.1 | Configuration Space

Since a real robot has size, shape, and orientation, direct path planning in its original environment can be complicated. Instead, we use C-Space to simplify the problem:

- The robot is shrunk to a point;
- Obstacles are expanded by the size of the robot.

In this task, I use morphological dilation function in OpenCV to generate the Configuration Space (C-Space) from a given 2D grid cells map (Figure 1.1).

```python
MAP_RES = 0.2 # each cell represents a 0.2m x 0.2m square in reality
ROB_RAD = 0.3 # Robot's radius is 0.3m
kernel_size = int(np.ceil(ROB_RAD / MAP_RES)) # Convert to grid cells
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (kernel_size * ...
    2, kernel_size * 2))
cspace_map = cv2.dilate((grid_map_img == 0).astype(np.uint8), kernel, ...
    iterations=1) # Expand obstacles
cspace_map = (1 - cspace_map) * 255  # Convert back to free space format
grid_map = cspace_map.transpose()
```

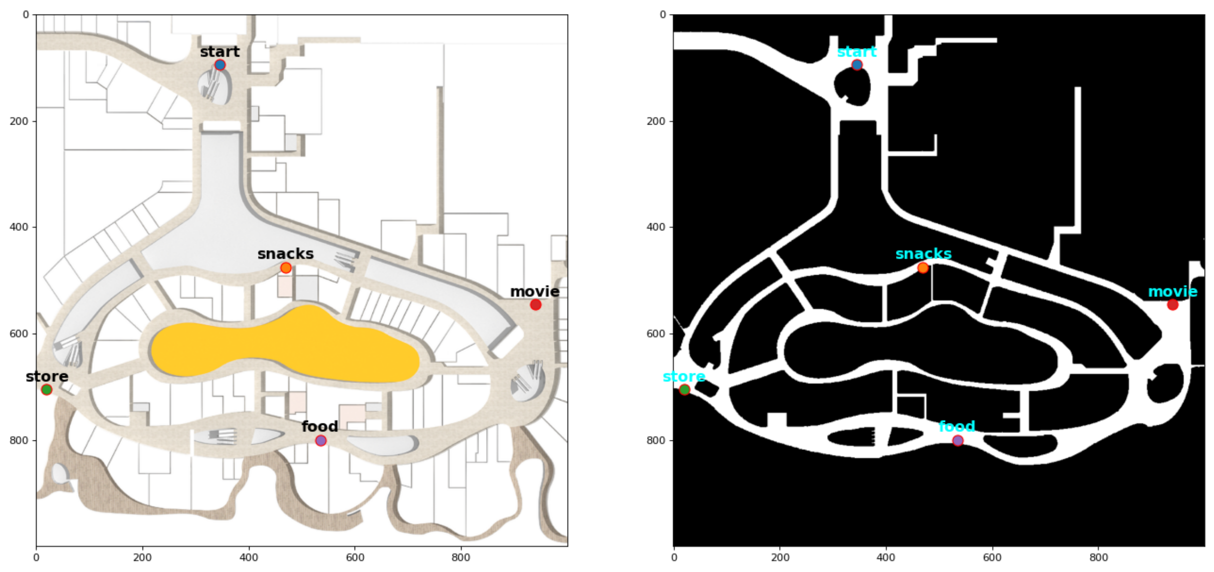


Figure 1.1: Right: original environment; Left: configuration Space

1.2 | A^* Planning Algorithm

The A^* algorithm finds the shortest path from a start position to a goal position in a grid map. Here's the step-by-step explanation:

- Initialize priority queue and cost dictionaries;

```
open_set = []
heappush(open_set, (0, tuple(start)))
came_from = {}
g_score = {tuple(start): 0}
f_score = {tuple(start): heuristic(start, goal)}
visited_cells = set()  # Store all visited cells
```

- Define 8-connected neighbors with cost values;

```
directions =
[(-1, 0, 0.2), (1, 0, 0.2),  # left, right
(0, -1, 0.2), (0, 1, 0.2),  # up, dowm
(-1, -1, 0.282), (-1, 1, 0.282), # lower left, lower right
(1, -1, 0.282), (1, 1, 0.282)]   # top left, top right
```

- Process frontiers with the minimal $f(n)$ from priority queue;

```
while open_set:
    _, current = heappop(open_set)
    visited_cells.add(current)
    for dx, dy, move_cost in directions:
        neighbor = (current[0] + dx, current[1] + dy)
        if grid_map[neighbor] == 255:
            tentative_g_score = g_score[current] + move_cost
            if neighbor not in g_score or tentative_g_score < ...
                g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g_score
                f_score[neighbor] = tentative_g_score + ...
                    heuristic(neighbor, goal)
                heappush(open_set, (f_score[neighbor], neighbor))
```

- If the goal is reached, return the reconstructed path.

```
if current == tuple(goal):
    path = []
    while current in came_from:
        path.append(current)
        current = came_from[current]
    path.append(tuple(start))
    path.reverse()
    return path, visited_cells
```

1.3 | Heuristic functions

Here are common heuristic functions used in A^*:

1.3.1 | Euclidean Distance: for continuous space or movement in any direction

$$h(n) = \sqrt{(x_n - x_g)^2 + (y_n - y_g)^2} \tag{1.1}$$

```python
# Define heuristic function using euclidean distance
def heuristic(a, b):
    return np.linalg.norm(np.array(a) - np.array(b))
```

1.3.2 | Manhattan Distance: for 4-connected grids

;

$$h(n) = |x_n - x_g| + |y_n - y_g| \tag{1.2}$$

```python
# Define heuristic function using Manhattan Distance
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

1.3.3 | Chebyshev Distance: for 8-connected grids with the same cost value

$$h(n) = \max(|x_n - x_g|, |y_n - y_g|) \tag{1.3}$$

```python
# Define heuristic function using Chebyshev Distance
def heuristic(a, b):
    return max(abs(a[0] - b[0]), abs(a[1] - b[1]))
```

1.3.4 | Diagonal Distance: for 8-connected grids with different diagonal cost value

$$h(n) = D \cdot (dx + dy) + (D_{\text{diagonal}} - 2D) \cdot \min(dx, dy) \tag{1.4}$$

where $dx = |x_n - x_g|, \quad dy = |y_n - y_g|$;

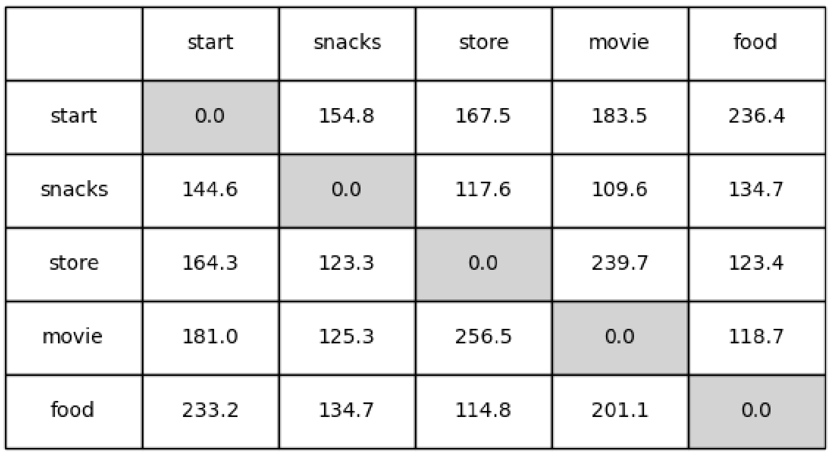
```python
# Define heuristic function using Diagonal Distance (for 8-connected ...
    grid)
def heuristic(a, b):
    D, D2 = 0.2, 0.282
    dx = abs(a[0] - b[0])
    dy = abs(a[1] - b[1])
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

1.4 | Comparisons of different heuristic functions

The mission of this task is to plan a path/trajectory between each pair of a start and end points, and find the most efficient route to visit all the four locations and return to the start. To compare the performance of different heuristic functions, I apply the exhaustive search method to solve this 'Traveling Shopper Problem' (TSP), which will be further discussed in task 2. The critical performance indicators can be classified into three types: the total traveled distance in meters, the number of all the cells visited by the A^* algorithm, and the total run time of the A^* algorithm.

1.4.1 | Euclidean Distance performance indicators

- Best Route : ['start', 'store', 'food', 'movie', 'snacks', 'start']
  Total Distance: 662.27 meters



	start	snacks	store	movie	food
start	0.0	154.8	167.5	183.5	236.4
snacks	144.6	0.0	117.6	109.6	134.7
store	164.3	123.3	0.0	239.7	123.4
movie	181.0	125.3	256.5	0.0	118.7
food	233.2	134.7	114.8	201.1	0.0

Figure 1.2: The shortest distances between each pair of locations through **Euclidean Distance**-based heuristic function.

- The number of all the cells visited by the A^* algorithm: 42700

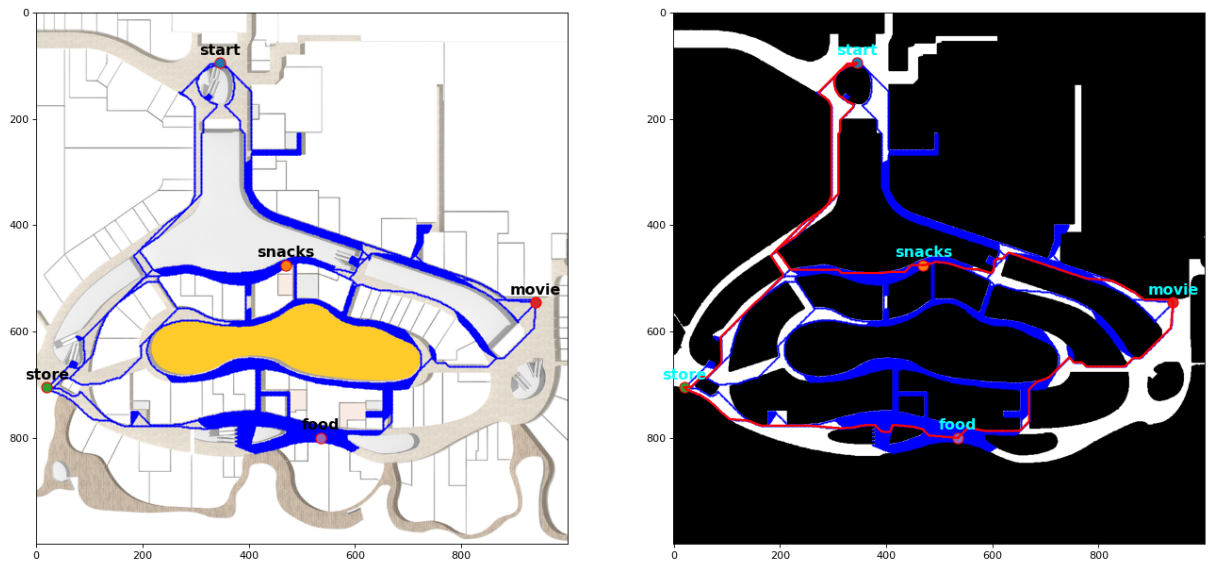


Figure 1.3: Left: the visited cells (blue area); Right: the most efficient route to visit all the four locations and return to the start through A^* algorithm with **Euclidean Distance**-based heuristic function and exhaustive search TSP algorithm.

- The total run time of the A^* algorithm: 3.7336 seconds

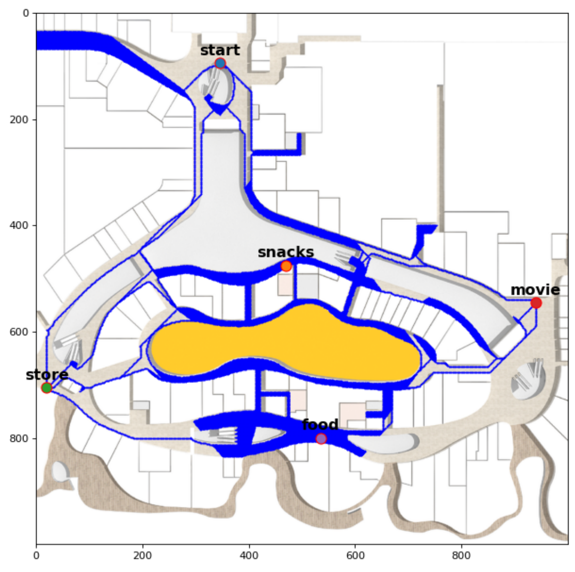
1.4.2 | Manhattan Distance performance indicators

- Best Route : ['start', 'store', 'food', 'movie', 'snacks', 'start']
  Total Distance: 652.91 meters

	start	snacks	store	movie	food
start	0.0	152.4	165.6	183.0	232.0
snacks	144.2	0.0	119.1	108.3	134.7
store	158.9	124.1	0.0	218.8	132.7
movie	209.5	141.9	251.7	0.0	118.5
food	230.7	135.2	114.8	117.6	0.0

Figure 1.4: The shortest distances between each pair of locations through **Manhattan Distance**-based heuristic functions

- The number of all the cells visited by the A^* algorithm: 53691



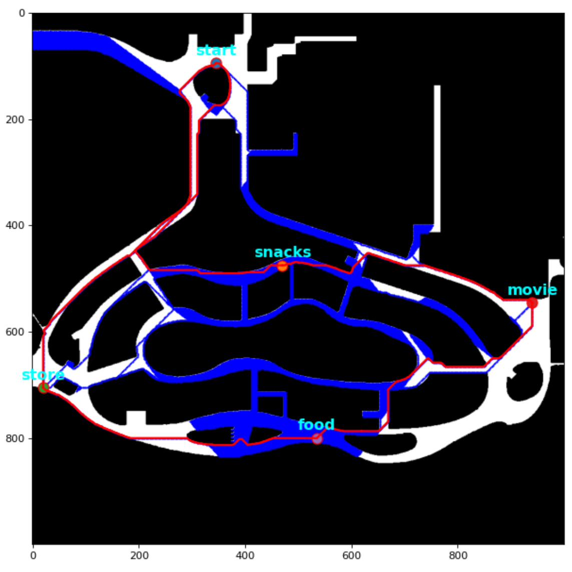
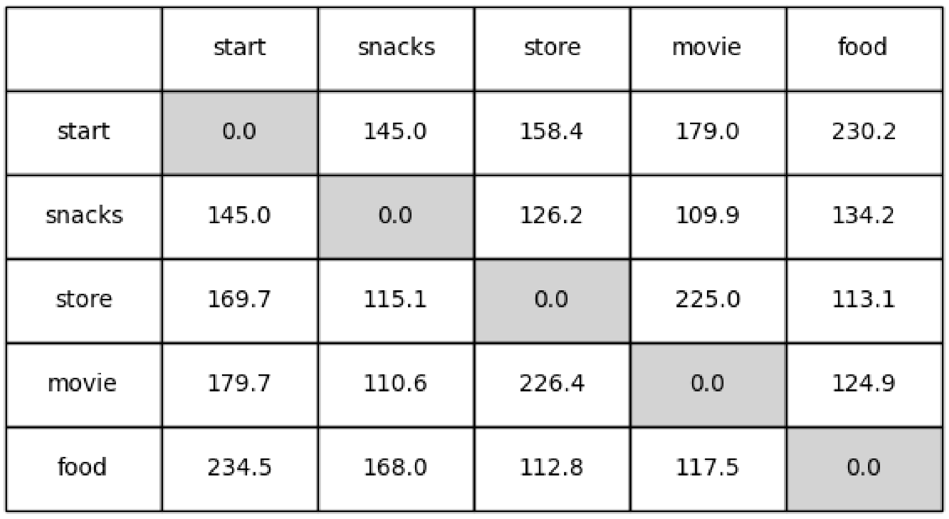


Figure 1.5: Left: the visited cells (blue area); Right: the most efficient route to visit all the four locations and return to the start through A^* algorithm with **Manhattan Distance**-based heuristic function and exhaustive search TSP algorithm.

- The total run time of the A^* algorithm: 7.9803 seconds

1.4.3 | Chebyshev Distance performance indicators

- Best Route : ['start', 'snacks', 'movie', 'food', 'store', 'start']
  Total Distance: 644.67 meters



	start	snacks	store	movie	food
start	0.0	145.0	158.4	179.0	230.2
snacks	145.0	0.0	126.2	109.9	134.2
store	169.7	115.1	0.0	225.0	113.1
movie	179.7	110.6	226.4	0.0	124.9
food	234.5	168.0	112.8	117.5	0.0

Figure 1.6: The shortest distances between each pair of locations through **Chebyshev Distance**-based heuristic functions

- The number of all the cells visited by the A^* algorithm: 62536

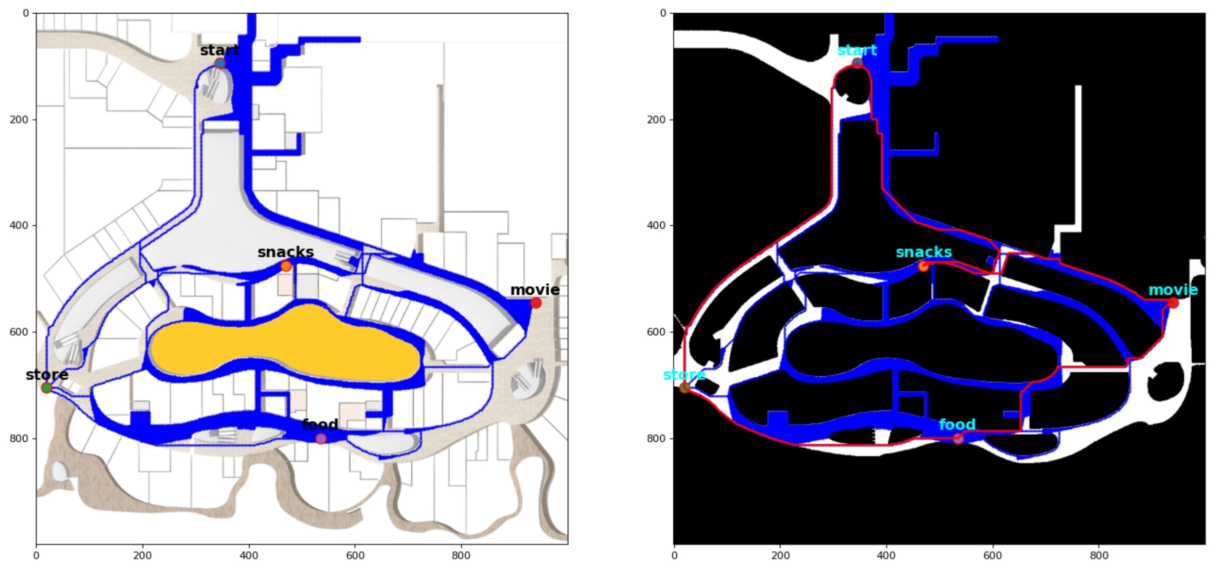
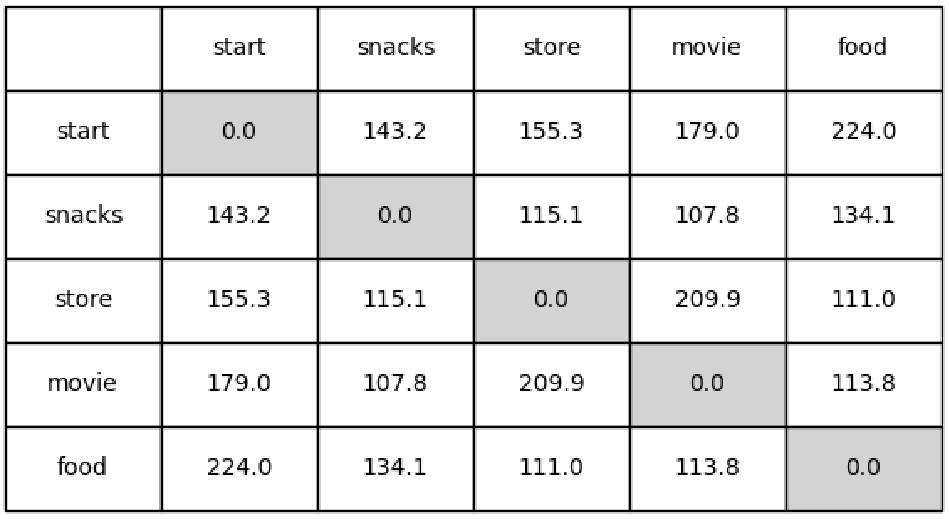


Figure 1.7: Left: the visited cells (blue area); Right: the most efficient route to visit all the four locations and return to the start through A^* algorithm with **Chebyshev Distance**-based heuristic function and exhaustive search TSP algorithm.

- The total run time of the A^* algorithm: 12.2235 seconds

1.4.4 | Diagonal Distance performance indicators

- Best Route : ['start', 'snacks', 'movie', 'food', 'store', 'start']
  Total Distance: 631.09 meters



	start	snacks	store	movie	food
start	0.0	143.2	155.3	179.0	224.0
snacks	143.2	0.0	115.1	107.8	134.1
store	155.3	115.1	0.0	209.9	111.0
movie	179.0	107.8	209.9	0.0	113.8
food	224.0	134.1	111.0	113.8	0.0

Figure 1.8: The shortest distances between each pair of locations through **Diagonal Distance**-based heuristic functions

- The number of all the cells visited by the A^* algorithm: 128457

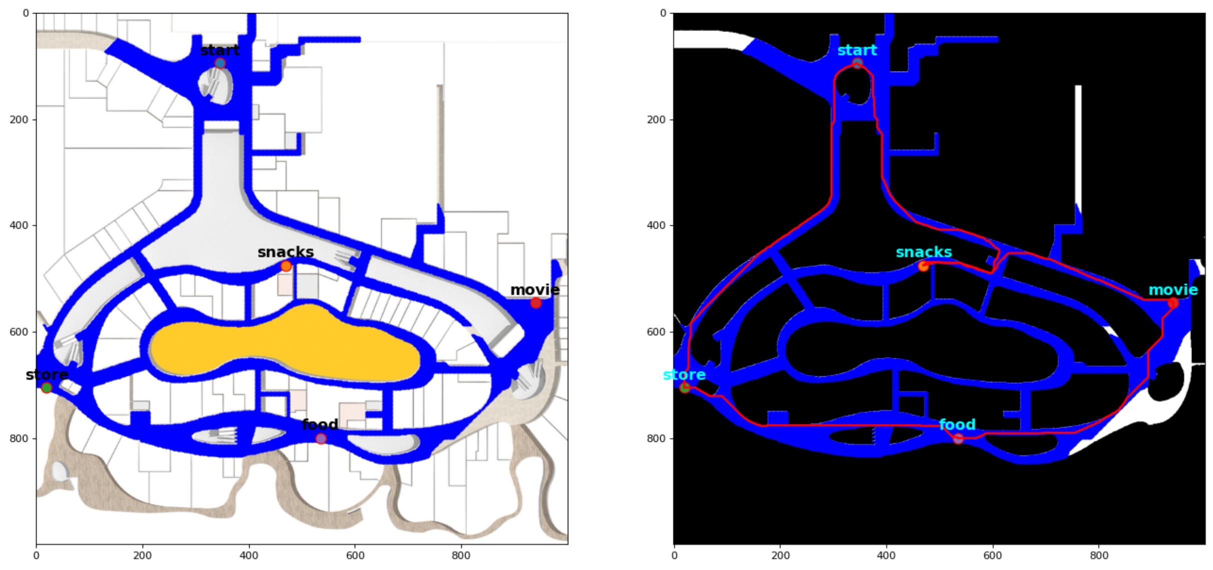


Figure 1.9: Left: the visited cells (blue area); Right: the most efficient route to visit all the four locations and return to the start through A^* algorithm with **Diagonal Distance**-based heuristic function and exhaustive search TSP algorithm.

- The total run time of the A^* algorithm: 8.1011 seconds

1.4.5 | Conclusion

Viewing from the perspective of the total traveled distance, Diagonal Distance-based A^* algorithm has the best performance, Euclidean Distance-based method has the worst performance. While from the perspective of the number of all the cells visited and the total run time of the algorithm, Euclidean Distance-based method has the best performance, Diagonal Distance-based has the worst performance. In conclusion, in this 8-connected neighbors with different cost values scenario, Diagonal Distance-based A^* algorithm have a better performance in generating optimal planning path, while with a relatively long running time; Euclidean Distance-based A^* algorithm have a slightly worse performance in generating optimal planning path, while with a shorter running time. Therefore, when the environment is huge, we'd better apply the Euclidean Distance-based A^* algorithm for path planning. If the environment is relatively small, like this task, Diagonal Distance-based A^* algorithm is a better choice.

1.5 | Implement the RRT algorithm

Rapidly-Exploring Random Tree (RRT) is a sampling-based algorithm commonly used for path planning in robotics. Below is the step-by-step explanation of how RRT works:

- Initialize the Tree: The algorithm starts with an initial node (the robot's start position);

```python
class Node:
    def __init__(self, pos, parent=None):
        self.pos = pos
        self.parent = parent
# Initialize the Tree
tree = [Node(tuple(start))]
```

- Random Sampling: RRT randomly samples a point in the configuration space;

```python
# Generate a random free space point
def get_random_point(grid_map):
    while True:
        pos = (random.randint(0, 999), random.randint(0, 999))
        if grid_map[pos] == 255:
            return pos
```

- Nearest Node Selection: RRT finds the closest node in the tree to the sampled point;

```python
# Find nearest node in the tree
def get_nearest_node(tree, point):
    return min(tree, key=lambda node: ...
        euclidean_distance(node.pos, point))
```

- Steering: A new node is created by moving a fixed step size from the nearest node toward the sampled point;

```python
def steer(from_node, to_point, step_size):
    vector = np.array(to_point) - np.array(from_node.pos)
    dist = np.linalg.norm(vector)
    if dist < step_size:
        new_point = to_point
    else:
        new_point = tuple((np.array(from_node.pos) + vector / ...
            dist * step_size).astype(int))
    return new_point
```

- Collision Check: If the new node is in a collision-free space, it is added to the tree;

```
# Check if the path from from_node to to_point is collision-free
def check_collision(grid_map, from_node, to_point):
    line = np.linspace(from_node.pos, to_point, num=20).astype(int)
    return all(grid_map[tuple(pos)] == 255 for pos in line)
```

- Iterate the above steps until a node reaches the goal region or the maximum number of iterations is reached.

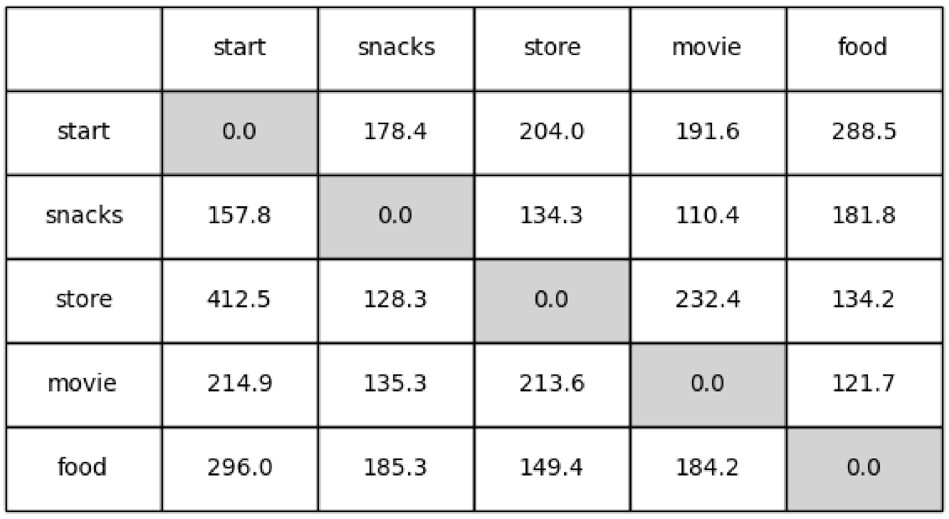
```
# Run the RRT algorithm from start to goal
def rrt_search(grid_map, start, goal):
    visited_cells = set()
    path = []
    tree = [Node(tuple(start))]
    for _ in range(max_iterations):
        rand_point = tuple(goal) if random.random() < 0.1 else ...
            get_random_point(grid_map)
        nearest_node = get_nearest_node(tree, rand_point)
        new_point = steer(nearest_node, rand_point, step_size)

        if grid_map[new_point] == 255 and ...
            check_collision(grid_map, nearest_node, new_point):
            visited_cells.add(new_point)
            node = Node(new_point, nearest_node)
            tree.append(node)
            if euclidean_distance(new_point, tuple(goal)) < ...
                step_size:
                path.append(tuple(goal))
                while node:
                    path.append(node.pos)
                    node = node.parent
                path.append(tuple(start))
                path.reverse()
                return path, visited_cells
    return [], visited_cells
```

1.6 | Comparisons of A^* and RRT algorithms

1.6.1 | RRT algorithm performance indicators

- Best Route : ['start', 'snacks', 'store', 'food', 'movie', 'start']
  Total Distance: 748.74 meters
- The number of all the cells visited by the A^* algorithm: 8095



	start	snacks	store	movie	food
start	0.0	178.4	204.0	191.6	288.5
snacks	157.8	0.0	134.3	110.4	181.8
store	412.5	128.3	0.0	232.4	134.2
movie	214.9	135.3	213.6	0.0	121.7
food	296.0	185.3	149.4	184.2	0.0

Figure 1.10: The shortest distances between each pair of locations through **RRT** planning algorithm

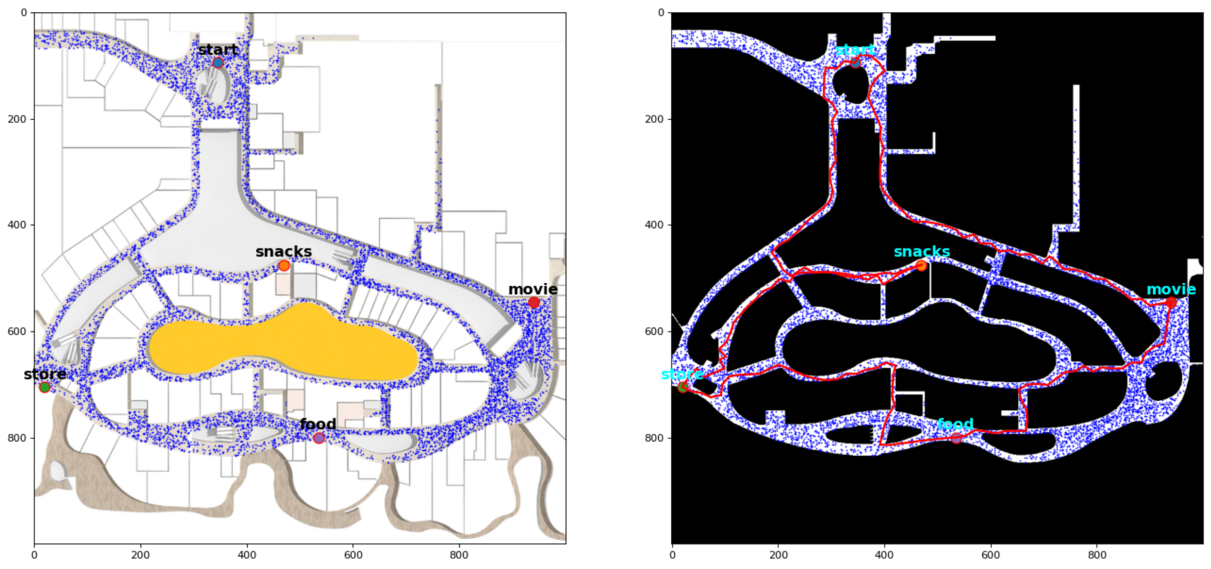


Figure 1.11: Left: the visited cells (blue area); Right: the most efficient route to visit all the four locations and return to the start through RRT with exhaustive search TSP algorithm.

- The total run time of the RRT algorithm: 11.3980 seconds

1.6.2 | Conclusion

Viewing from the perspective of the total traveled distance and the total run time of the algorithm, A^* planning algorithm (especially with Euclidean Distance and Diagonal Distance-based heuristic functions) has a better performance. Although the RRT algorithm visits fewer number of cells, the complex computations involved, such as Nearest Node Selection, Steering, and Collision Check, result in an increased computation time rather than a reduction compared to the A^* algorithm. At the same time, I found that the A* algorithm produces deterministic results, meaning that each run generates the same outcome. In contrast, the RRT algorithm, due to its random generation of collision-free path points, produces different results in each run and cannot guarantee path planning completion within a given limited number of iterations. Therefore, we can conclude that in the 2D grid cells environment similar to this task, the A^* algorithm provides a better path planning solution. However, A^* algorithm also has its drawbacks. In scenarios involving dynamic obstacles where the configuration space cannot be explicitly constructed, the A^* algorithm loses its applicability, whereas the RRT algorithm can effectively adapt to these complex environments.

2 | Task 2: The Traveling Shopper Problem

Traveling Shopper Problem (TSP) focuses on efficiently visiting a set of stores to purchase specific items while minimizing travel cost or time. The mission of this task is to find the most efficient route to visit all the four locations $['snacks']$, $['store']$, $['movie']$, $['food']$ and return to the $['start']$, which is a Traveling Shopper Problem. Here, I tested two different TSP algorithms: list heuristic **Exhaustive Search TSP** and greedy heuristic **Nearest Neighbor TSP**.

2.1 | Exhaustive Search TSP

Exhaustive search TSP, also known as brute-force search TSP, is a method of solving TSP by evaluating all possible permutations of location visit sequences to find the shortest possible route.

```python
# Solve Travelling Shopper Problem (TSP) through trying all possible ...
    orderings of the locations
def tsp_exhaustive_search(path_cache_dist, keys):
    start_time = time.perf_counter()
    shortest_distance = float('inf')
    best_route = []

    for perm in permutations(keys[1:]):
        route = ['start'] + list(perm) + ['start']
        total_distance = sum(path_cache_dist[(route[i], route[i+1])] ...
            for i in range(len(route) - 1))
        if total_distance < shortest_distance:
            shortest_distance = total_distance
            best_route = route

    end_time = time.perf_counter()
    tsp_execution_time = end_time - start_time
    return best_route, shortest_distance, tsp_execution_time
```

2.2 | Nearest Neighbor TSP

The Nearest Neighbor TSP algorithm is a greedy heuristic method to solve the Traveling Salesman Problem (TSP). It builds the optimal route by always selecting the nearest unvisited location at each step, continuing until all locations are visited.

```python
# Solve Travelling Shopper Problem (TSP) through taking the nearest ...
    neighbor from current
def tsp_nearest_neighbor(path_cache_dist, keys):
    start_time = time.perf_counter()
    unvisited = set(keys)
    unvisited.remove('start')
    best_route = ['start']
    shortest_distance = 0
    current = 'start'

    while unvisited:
        next = min(unvisited, key=lambda pos: ...
            path_cache_dist[(current, pos)])
        shortest_distance += path_cache_dist[(current, next)]
        best_route.append(next)
        unvisited.remove(next)
        current = next

    shortest_distance += path_cache_dist[(current, 'start')]
    best_route.append('start')

    end_time = time.perf_counter()
    tsp_execution_time = end_time - start_time
    return best_route, shortest_distance, tsp_execution_time
```

2.3 | Conclusion

Exhaustive Search TSP guarantees the optimal solution by evaluating all possible routes, but its $O(N!)$ time complexity makes it impractical for large-scale problems, limiting its use to small cases ($N < 10$) or as a benchmark for other algorithms. In contrast, Nearest Neighbor TSP is a greedy heuristic with $O(N^2)$ complexity, making it significantly faster and scalable for large datasets, though it often produces suboptimal paths and is sensitive to the starting city. While Exhaustive Search is ideal for small-scale exact optimization problems, Nearest Neighbor is more suitable for real-time applications like logistics, vehicle routing, and robotics, or as an initial solution for more advanced optimization techniques.

3 | Task 3: Submitted as a GitHub Repository

URL: https://github.com/nihaozhan/ME5413_Homework3_planning