AI 3603 Artificial Intelligence: Principles and Techniques

By: Chen Jingtao (523030910028)

HW#: 1

October 19, 2025

I. INTRODUCTION

This Lab aims to design an algorithm to guide from the start point to the target point avoiding obstacles based on the A* algorithm. The algorithm should be implemented in Python language. And it's required to generate a smooth trajectory based on the path found by the A* algorithm using cubic spline interpolation.

The Lab is divided into three steps:

- Implement basic A* algorithm to find a path from start to target avoiding obstacles
- Optimize the A* algorithm to achieve a better path
- Generate a smooth path based on the path found by the A* algorithm

II. TASKS

A. Task 1: Implement basic A* algorithm to find a path from start to target avoiding obstacles

The first task is to implement the basic A* algorithm to find a path from start to target avoiding obstacles. The A* algorithm is a popular pathfinding and graph traversal algorithm that is used in many applications, including games and robotics. The algorithm uses a heuristic function to estimate the cost of reaching the target from the current node, and it combines this with the cost of reaching the current node from the start node to determine the total cost of each node. The algorithm then explores the nodes with the lowest total cost first, until it reaches the target node.

As for the search strategy we pick up BFS (Breadth-First Search) to implement the A* algorithm.

The heuristiuc function we choose is the Euclidean distance between the current node and the target node. And the cost of reaching the current node from the start node is simply the number of steps taken to reach that node.

- $f(x) = g(x) + h(x)$
- $g(x) =$ cost from start to current node
- $h(x) =$ Euclidean distance from current node to target

The code implementation is as follow:

```python
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import time
import heapq
MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')


### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this
    ↪ block if necessary.

class Queue:
```

```
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)

    def pop(self):
        """
          Dequeue the earliest enqueued item still in the queue. This
          operation removes the item from the queue.
        """
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0

class PriorityQueue:
    """
      Implements a priority queue data structure. Each inserted item
      has a priority associated with it and the client is usually interested
      in quick retrieval of the lowest-priority item in the queue. This
      data structure allows O(1) access to the lowest-priority item.
    """
    def  __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        # If item already in priority queue with higher priority, update its priority
            ↪ and rebuild the heap.
        # If item already in priority queue with equal or lower priority, do nothing.
        # If item not in priority queue, do the same thing as self.push.
        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)

def heuristic(pos, goal_pos):
```

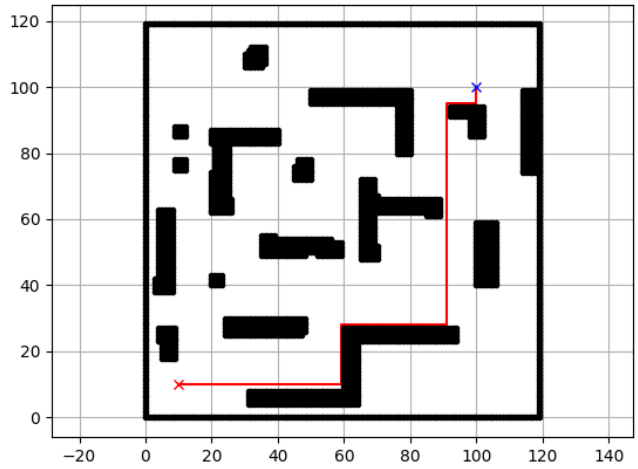
```
    return abs(pos[0] - goal_pos[0]) + abs(pos[1] - goal_pos[1])

###  END CODE HERE  ###


def A_star(world_map, start_pos, goal_pos):
    """
    Given map of the world, start position of the robot and the position of the goal,
    plan a path from start position to the goal using A* algorithm.

    Arguments:
    world_map -- A 120*120 array indicating current map, where 0 indicating
        ↪ traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """

    ### START CODE HERE ###

    fringe = PriorityQueue()

    fringe.push([start_pos,[start_pos], 0], 0 + heuristic(start_pos, goal_pos))

    best_g = dict()

    best_g[tuple(start_pos)] = 0

    while not fringe.isEmpty():
        state, path, cost = fringe.pop()
        if state == goal_pos:
            return path
        # Get successors
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1)]:
            successor = [state[0]+dx, state[1]+dy]
            if 0 <= successor[0] < 120 and 0 <= successor[1] < 120 and
                ↪ world_map[successor[0]][successor[1]] == 0:
                stepCost = 1
                new_cost = cost + stepCost
                if tuple(successor) not in best_g or new_cost <
                    ↪ best_g[tuple(successor)]:
                    best_g[tuple(successor)] = new_cost
                    f_cost = new_cost + heuristic(successor, goal_pos)
                    fringe.push([successor, path + [successor], new_cost], f_cost)

    raise Exception("No path found")


    ###  END CODE HERE  ###




if __name__ == '__main__':

    # Get the map of the world representing in a 120*120 array, where 0 indicating
```

```
        ↪ traversable and 1 indicating obstacles.
    map = np.load(MAP_PATH)

    # Define goal position of the exploration
    goal_pos = [100, 100]

    # Define start position of the robot.
    start_pos = [10, 10]

    # Plan a path based on map from start position of the robot to the goal.
    path = A_star(map, start_pos, goal_pos)

    # Visualize the map and path.
    obstacles_x, obstacles_y = [], []
    for i in range(120):
        for j in range(120):
            if map[i][j] == 1:
                obstacles_x.append(i)
                obstacles_y.append(j)

    path_x, path_y = [], []
    for path_node in path:
        path_x.append(path_node[0])
        path_y.append(path_node[1])

    plt.plot(path_x, path_y, "-r")
    plt.plot(start_pos[0], start_pos[1], "xr")
    plt.plot(goal_pos[0], goal_pos[1], "xb")
    plt.plot(obstacles_x, obstacles_y, ".k")
    plt.grid(True)
    plt.axis("equal")
    plt.show()
```

In the front of the code we implement the Queue and its descendant PriorityQueue to support the A* algorithm. Then we implement the AStarPlanner function to encapsulate the A* algorithm. The code here are reused from another project I did before.

The main function is the plan function which takes in the start and target positions and returns the path found by the algorithm. The function uses a priority queue to explore the nodes with the lowest total cost first, and it keeps track of the visited nodes to avoid cycles using a dictionary bestg.

Finally we get a path as the graph shown below:



B. Task 2: Optimize the A* algorithm to achieve a better path

Basing on Task-1, the assignment require to optimize our path according to following three rules:

- Be able to turn in 8 directions (N, NE, E, SE, S, SW, W, NW)
- Consider the distance between the current node and the wall
- Consider the cost of steering

Our algorithm framework is similar to Task-1, and we make some modifications to the searching-part and the heuristic function.

```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import time
import heapq
MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')


### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this
    ↪ block if necessary.

class Queue:
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)
```

```
21
22      def pop(self):
23          """
24            Dequeue the earliest enqueued item still in the queue. This
25            operation removes the item from the queue.
26          """
27          return self.list.pop()
28
29      def isEmpty(self):
30          "Returns true if the queue is empty"
31          return len(self.list) == 0
32
33  class PriorityQueue:
34      """
35        Implements a priority queue data structure. Each inserted item
36        has a priority associated with it and the client is usually interested
37        in quick retrieval of the lowest-priority item in the queue. This
38        data structure allows O(1) access to the lowest-priority item.
39      """
40      def  __init__(self):
41          self.heap = []
42          self.count = 0
43
44      def push(self, item, priority):
45          entry = (priority, self.count, item)
46          heapq.heappush(self.heap, entry)
47          self.count += 1
48
49      def pop(self):
50          (_, _, item) = heapq.heappop(self.heap)
51          return item
52
53      def isEmpty(self):
54          return len(self.heap) == 0
55
56      def update(self, item, priority):
57          # If item already in priority queue with higher priority, update its priority
                ↪ and rebuild the heap.
58          # If item already in priority queue with equal or lower priority, do nothing.
59          # If item not in priority queue, do the same thing as self.push.
60          for index, (p, c, i) in enumerate(self.heap):
61              if i == item:
62                  if p <= priority:
63                      break
64                  del self.heap[index]
65                  self.heap.append((priority, c, item))
66                  heapq.heapify(self.heap)
67                  break
68          else:
69              self.push(item, priority)
70
71  def Distance_to_Wall(world_map, position):
72      x, y = position
73      d = 1000.0
74      dist = 1
75      while True:
76          for dx in [-dist, 0, dist]:
77              for dy in [-dist, 0, dist]:
78                  if dx * dx + dy * dy <= dist * dist:
```

```
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < 120 and 0 <= ny < 120:
                        if world_map[nx][ny] == 1:
                            d = min(d, (dx * dx + dy * dy) ** 0.5)
        if d != 1000:
            return d
        dist += 1

def heuristic(world_map, pos, goal_pos, if_turn=0):
    return abs(pos[0] - goal_pos[0]) + abs(pos[1] - goal_pos[1]) - 3 *
        ↪ Distance_to_Wall(world_map, pos) + 10 * if_turn

###  END CODE HERE  ###


def Improved_A_star(world_map, start_pos, goal_pos):
    """
    Given map of the world, start position of the robot and the position of the goal,
    plan a path from start position to the goal using A* algorithm.

    Arguments:
    world_map -- A 120*120 array indicating current map, where 0 indicating
        ↪ traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """

    ### START CODE HERE ###

    fringe = PriorityQueue()

    fringe.push([start_pos,[start_pos], 0], 0 + heuristic(world_map, start_pos,
        ↪ goal_pos))

    best_g = dict()

    best_g[tuple(start_pos)] = 0

    while not fringe.isEmpty():
        state, path, cost = fringe.pop()
        if state == goal_pos:
            return path
        dx_prev = 2
        dy_prev = 2
        if len(path) >= 2:
            dx_prev = state[0] - path[-2][0]
            dy_prev = state[1] - path[-2][1]
        # Get successors
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1),(1,1),(1,-1),(-1,1),(-1,-1)]:
            successor = [state[0]+dx, state[1]+dy]
            if (dx == dx_prev and dy == dy_prev):
                if_turn = 0
            else:
                if_turn = 1
            if 0 <= successor[0] < 120 and 0 <= successor[1] < 120 and
                ↪ world_map[successor[0]][successor[1]] == 0:
```

```
                stepCost = 1.4 if abs(dx) + abs(dy) == 2 else 1
                new_cost = cost + stepCost
                if tuple(successor) not in best_g or new_cost <
                    ↪ best_g[tuple(successor)]:
                    best_g[tuple(successor)] = new_cost
                    f_cost = new_cost + heuristic(world_map, successor, goal_pos,
                        ↪ if_turn)
                    fringe.push([successor, path + [successor], new_cost], f_cost)

    raise Exception("No path found")


    ###  END CODE HERE  ###




if __name__ == '__main__':

    # Get the map of the world representing in a 120*120 array, where 0 indicating
        ↪ traversable and 1 indicating obstacles.
    map = np.load(MAP_PATH)

    # Define goal position of the exploration
    goal_pos = [100, 100]

    # Define start position of the robot.
    start_pos = [10, 10]

    # Plan a path based on map from start position of the robot to the goal.
    path = Improved_A_star(map, start_pos, goal_pos)

    # Visualize the map and path.
    obstacles_x, obstacles_y = [], []
    for i in range(120):
        for j in range(120):
            if map[i][j] == 1:
                obstacles_x.append(i)
                obstacles_y.append(j)

    path_x, path_y = [], []
    for path_node in path:
        path_x.append(path_node[0])
        path_y.append(path_node[1])

    plt.plot(path_x, path_y, "-r")
    plt.plot(start_pos[0], start_pos[1], "xr")
    plt.plot(goal_pos[0], goal_pos[1], "xb")
    plt.plot(obstacles_x, obstacles_y, ".k")
    plt.grid(True)
    plt.axis("equal")
    plt.show()
```

When generating the (dx, dy) pairs, we consider 8 directions instead of 4 directions in Task-1.

When calculating the heruistic function, we add two more factors: The shortest distance between the cureent node and the wall, and the cost of steering.

The shortest distance between the current node and the wall is calculated in a new function. It will be a negative value to the h-function since we want our robot to stay away from the wall.

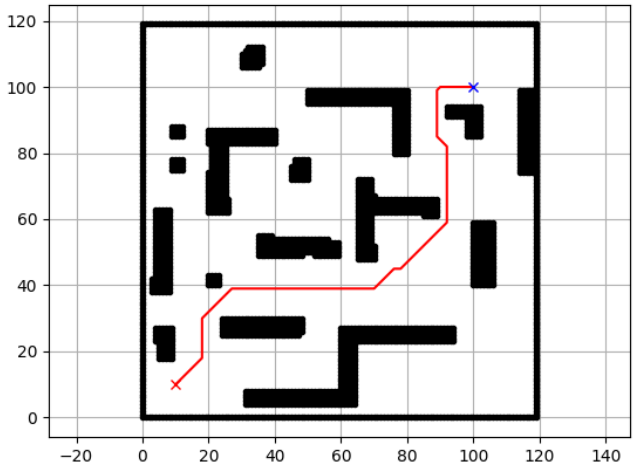
The cost of steering is calculated only if our current direction is different from the last direction. To implement this, we judge if we are changing direction and pass it to our h-function in the searching-part.

At last we assign a proper weight to each factor in the h-function to get a better path.

$$h(x) = \text{EuclideanDistance} - 3 \cdot \text{DistanceToWall} + 10 \cdot \text{SteeringCost} \tag{1}$$

$$\text{SteeringCost} = \begin{cases} 1, & \text{if changing direction} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

And here is the result path that:



Additionally, we compare our A-star algorithm in Task-1 and Task-2. For example, we calculate the time consumption of both Task-1 and Task-2. The result is as follow:

- Time consumption of Task-1: 0.231 s
- Time consumption of Task-2: 0.732 s

Task-2 apparently cost more time than Task-1, which is reasonable since we add more factors in the heuristic function and expand our searching directions, so our searching queue is larger than before.

And practically, the path in Task-2 is better than Task-1, since it avoids obstacles better. If we follow the path we found in Task-1, which is too close to the wall, our robot may collide with the wall due to some errors in movement. In Task-2, we consider the distance to the wall, so our robot can avoid obstacles better. Besides, we move towards eight directions instead of four, which gives us more flexibility in navigating the environment.

So in all, though Task-2 costs more time, it gives us a better path to avoid obstacles.

C. Task 3: Generate a smooth path based on the path found by the A* algorithm

To generate a smooth path, the assignment offer me three methods, and I choose to dig into the Polynomial Interpolation, not of any reasons, only because it seems most easy, lol.

My strategy is brutely simple:

- First, use our optimized A* algorithm in Task-2 to find the initial path
- Then, simplify the path by only recording the steering nodes.
- Finally, we use cubic spline to link every two neighbor nodes to form an overall smooth path, and sampling nodes equidistantly on the smooth path to plot our final path.(Here sampling 100 nodes)

To achieve the cubic spline interpolation, I use the CubicSpline function in the scipy.interpolate library. The code implementation is as follow:

```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import time
import heapq
import numpy as np
from scipy.interpolate import CubicSpline
MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')


### START CODE HERE ###
# This code block is optional. You can define your utility function and class in this
    ↪ block if necessary.

class Queue:
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)

    def pop(self):
        """
          Dequeue the earliest enqueued item still in the queue. This
          operation removes the item from the queue.
        """
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0

class PriorityQueue:
    """
      Implements a priority queue data structure. Each inserted item
      has a priority associated with it and the client is usually interested
      in quick retrieval of the lowest-priority item in the queue. This
      data structure allows O(1) access to the lowest-priority item.
```

```
    """
    def  __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        # If item already in priority queue with higher priority, update its priority
            ↪ and rebuild the heap.
        # If item already in priority queue with equal or lower priority, do nothing.
        # If item not in priority queue, do the same thing as self.push.
        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)

def Distance_to_Wall(world_map, position):
    x, y = position
    d = 1000.0
    dist = 1
    while True:
        for dx in [-dist, 0, dist]:
            for dy in [-dist, 0, dist]:
                if dx * dx + dy * dy <= dist * dist:
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < 120 and 0 <= ny < 120:
                        if world_map[nx][ny] == 1:
                            d = min(d, (dx * dx + dy * dy) ** 0.5)
        if d != 1000:
            return d
        dist += 1

def heuristic(world_map, pos, goal_pos, if_turn=0):
    return abs(pos[0] - goal_pos[0]) + abs(pos[1] - goal_pos[1]) - 3 *
        ↪ Distance_to_Wall(world_map, pos) + 10 * if_turn

def Improved_A_star(world_map, start_pos, goal_pos):
    """
    Given map of the world, start position of the robot and the position of the goal,
    plan a path from start position to the goal using A* algorithm.

    Arguments:
```

```
    world_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    path -- A N*2 array representing the planned path by A* algorithm.
    """

    fringe = PriorityQueue()

    fringe.push([start_pos,[start_pos], 0], 0 + heuristic(world_map, start_pos, goal_pos))

    best_g = dict()

    best_g[tuple(start_pos)] = 0

    while not fringe.isEmpty():
        state, path, cost = fringe.pop()
        if state == goal_pos:
            return path
        dx_prev = 2
        dy_prev = 2
        if len(path) >= 2:
            dx_prev = state[0] - path[-2][0]
            dy_prev = state[1] - path[-2][1]
        # Get successors
        for dx, dy in [(-1,0),(1,0),(0,-1),(0,1),(1,1),(1,-1),(-1,1),(-1,-1)]:
            successor = [state[0]+dx, state[1]+dy]
            if (dx == dx_prev and dy == dy_prev):
                if_turn = 0
            else:
                if_turn = 1
            if 0 <= successor[0] < 120 and 0 <= successor[1] < 120 and world_map[successor[0]][successor[1]] == 0:
                stepCost = 1.4 if abs(dx) + abs(dy) == 2 else 1
                new_cost = cost + stepCost
                if tuple(successor) not in best_g or new_cost < best_g[tuple(successor)]:
                    best_g[tuple(successor)] = new_cost
                    f_cost = new_cost + heuristic(world_map, successor, goal_pos, if_turn)
                    fringe.push([successor, path + [successor], new_cost], f_cost)

    raise Exception("No path found")

def simplify_path(world_map, path):
    simplified_path = [path[0]]
    current_direction = (path[1][0] - path[0][0], path[1][1] - path[0][1])

    for i in range(2, len(path)):
        prev_node = path[i - 1]
        current_node = path[i]
        direction = (current_node[0] - prev_node[0], current_node[1] - prev_node[1])

        if direction != current_direction:
            simplified_path.append(current_node)
            current_direction = direction
```

```
152
153     return simplified_path
154
155 def generate_smooth_trajectory(waypoints, num_samples=100):
156     waypoints = np.array(waypoints)
157
158     #
159     distances = np.cumsum([0] + [np.linalg.norm(waypoints[i+1] - waypoints[i])
160                                  for i in range(len(waypoints)-1)])
161
162     #
163     cs_x = CubicSpline(distances, waypoints[:, 0])
164     cs_y = CubicSpline(distances, waypoints[:, 1])
165
166     #
167     s_new = np.linspace(0, distances[-1], num_samples)
168     x_smooth = cs_x(s_new)
169     y_smooth = cs_y(s_new)
170
171     return list(zip(x_smooth, y_smooth))
172
173 ###  END CODE HERE  ###
174
175
176 def Self_driving_path_planner(world_map, start_pos, goal_pos):
177     """
178     Given map of the world, start position of the robot and the position of the goal,
179     plan a path from start position to the goal using A* algorithm.
180
181     Arguments:
182     world_map -- A 120*120 array indicating current map, where 0 indicating
            ↪ traversable and 1 indicating obstacles.
183     start_pos -- A 2D vector indicating the current position of the robot.
184     goal_pos -- A 2D vector indicating the position of the goal.
185
186     Return:
187     path -- A N*2 array representing the planned path by A* algorithm.
188     """
189
190     ### START CODE HERE ###
191
192     # A-star algorithm to plan a path from start position to the goal.
193
194     path = Improved_A_star(world_map, start_pos, goal_pos)
195
196     path = simplify_path(world_map, path)
197
198     path= generate_smooth_trajectory(path, num_samples=200)
199
200     ###  END CODE HERE  ###
201     return path
202
203
204 if __name__ == '__main__':
205
206     # Get the map of the world representing in a 120*120 array, where 0 indicating
            ↪ traversable and 1 indicating obstacles.
207     map = np.load(MAP_PATH)
208
```

```
    # Define goal position of the exploration
    goal_pos = [100, 100]

    # Define start position of the robot.
    start_pos = [10, 10]

    # Plan a path based on map from start position of the robot to the goal.
    path = Self_driving_path_planner(map, start_pos, goal_pos)

    # Visualize the map and path.
    obstacles_x, obstacles_y = [], []
    for i in range(120):
        for j in range(120):
            if map[i][j] == 1:
                obstacles_x.append(i)
                obstacles_y.append(j)

    path_x, path_y = [], []
    for path_node in path:
        path_x.append(path_node[0])
        path_y.append(path_node[1])

    plt.plot(path_x, path_y, "-r")
    plt.plot(start_pos[0], start_pos[1], "xr")
    plt.plot(goal_pos[0], goal_pos[1], "xb")
    plt.plot(obstacles_x, obstacles_y, ".k")
    plt.grid(True)
    plt.axis("equal")
    plt.show()
```

The result we get is quite satisfying:

It is more similar to the real-world robot path-planning, and it avoids sudden turns, which is better for the robot's movement.

III. CONCLUSION

In this report, we have explored the implementation of an optimized A* algorithm for robot path planning. We compared the performance of the algorithm in two different tasks, highlighting the improvements made in Task-2 by considering additional factors in the heuristic function and expanding the search directions. Although Task-2 required more computational time, it resulted in a more effective path that better avoided obstacles.

Furthermore, we applied cubic spline interpolation to smooth the path generated by the A* algorithm in Task-2. The resulting smooth path demonstrated improved characteristics for real-world robot navigation, minimizing sudden turns and enhancing overall movement efficiency.

In conclusion, the enhancements made in the A* algorithm and the subsequent path smoothing techniques have significantly contributed to the development of a more robust and effective robot path-planning solution.

And from this lab, I have gained a deeper understanding of path-planning algorithms and their practical applications in robotics. The experience has been invaluable in enhancing my skills in algorithm design and implementation.

Thanks for your patience in reading my lab report!