AI 3603 Artificial Intelligence: Principles and Techniques

By: LiuQiaoan (520030910220)

HW#: 1

October 5, 2022

Contents

I. INTRODUCTION

In AI3603 class, we students have studied A* algorithm, which can be applied to a search problem. In this homework, I try to develop a path planning framework for a service robot in the unknown environment using A* algorithm. There are 3 tasks,

- Task1 is a naive inplementation of A* algorithm, we control the robot move forward, backward, left and right.
- Task2 is an improved inplementation of A* algorithm, we can move upper left, upper right, bottom left, bottom right. What's more, we need to add steer cost and collision into consideration.
- Task3 is a self-driving car, I try to use Bézier curve to improve performance of Task2.

Now we are going to complete each task step by step.

II. TASK1: BASIC A* ALGORITHM

A. Model and Pseudocode

As we learned in the class, A* algorithm is a combination of UCS algorithm (dijkstra) (with a cost function g) and a heuristic function (h) modeling the remained cost from current position to the goal.

We model that the movement from one point to an adjacent point costs 1, since we only move the robot forward, backward, left and right in this task, we then use Manhattan distance from curret position to the goal as heuristic function.

First, I try to write down the pseudocode. I construct a data structure named `node`, which can represent the point at each position. See in Figure 1.

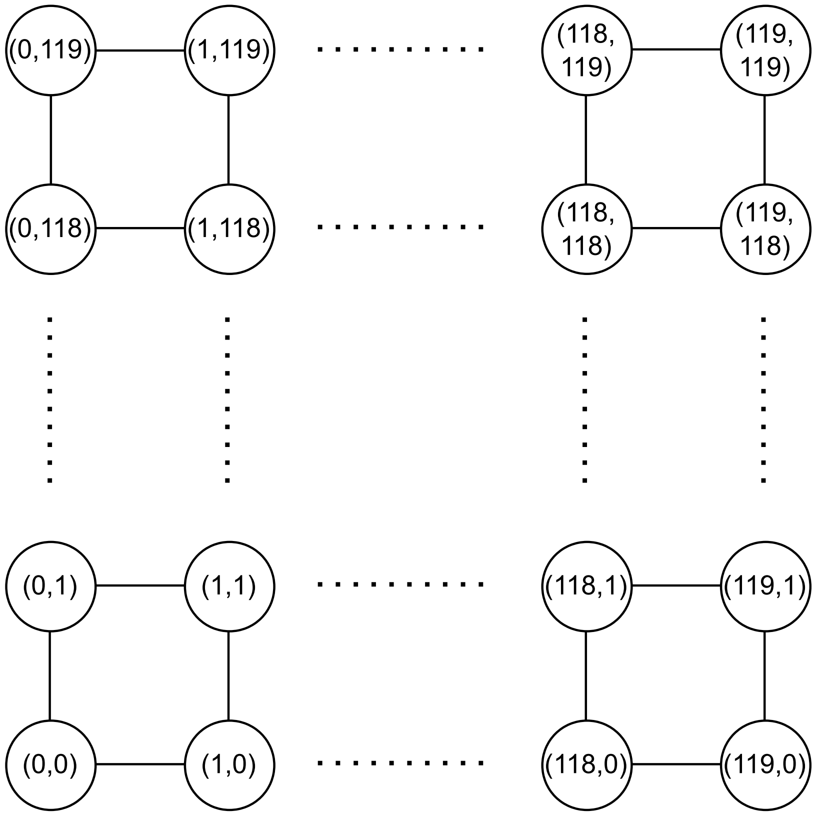

FIG. 1: Represent each point as a node.

Algorithm 1 BasicAStarAlgorithm

Input: *goal_pos*: position of goal, *current_map*: current known map, *current_pos*: current position.
Outcome: A *path* to move.

initialize a 2D array *distance* to store the graph, we view each point on the grid as a node in the graph
source_node $\leftarrow$ the node corresponding to *current_pos*
initialize g of *source_node* as 0
initialize $PQ \leftarrow \emptyset$ as a priority queue
add *source_node* to PQ
while PQ is not empty **do**
    pop *current_node* out
    if *current_node* is in the goal area **then**
        break
    end if
    if *current_node* has been visited **then**
        continue
    end if
    explore(*current_node*) $\triangleright$ This algorithm will be introduced later.
    Set *current_node* as visited
end while
From the lastly visited node, we trace back to get the complete *path* from *current_pos* to *goal_pos*
return *path*

Algorithm 2 explore

Input: A node named *node*.
Outcome: Nothing.

get current x-axis and y-axis position
get x-axis and y-axis position of ancester of *node*
for *next_node*: each point to move (forward, backward, left, right) **do**
    if This point is out of the map **then**
        break
    end if
    if This point is an obstacle **then**
        break
    end if
    if $g(node) + 1 < g(next\_node)$ **then**
        $g(next\_node) = g(node) + 1$
        Set the ancester of *next_node* as *node*
        PQ.push($next\_node$)
    end if
end for

B. Inplementation

First, I import packages and construct data structure `node`. Note that I add a `steer_cost` attribute to `node`, which can deal with the problem of too many turns easily.

```python
import DR20API
import numpy as np
import random
from queue import PriorityQueue
from matplotlib import pyplot as plt

MAX_G_COST = 1e3
q = PriorityQueue()
Manhattan_or_Euclidean = 1  # 1 means Manhattan and 2 means Euclidean
random.seed("AI3603")

def h(pos, goal_pos):
    return np.linalg.norm(np.array(pos)-np.array(goal_pos),ord=Manhattan_or_Euclidean)

class node:
    def __init__(self, pos, goal_pos):
        self.pos = pos
        self.is_visited = False
        self.ancester = pos
        self.steer_cost = 0
        self.h_cost = h(pos, goal_pos)+random.uniform(0, 3603e-6)
        self.g_cost = MAX_G_COST

    def cost(self):
        return self.h_cost+self.g_cost

    def __call__(self):
        self.f = self.h_cost+self.g_cost+self.steer_cost
        return self.f, selfPython code
```

Second, the inplementation of A* algorithm (corresponding to BasicAStarAlgorithm):

```python
def A_star(current_map, current_pos, goal_pos):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating
        obstacles.
    current_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 ###
    global distance
    distance = [[node([x,y],[100,100]) for y in range(0,120)] for x in range(0,120)]
    source_node = distance[current_pos[0]][current_pos[1]]
    source_node.g_cost = 0
```

```
    q.put(source_node())

    while not q.empty():
        current_node = q.get()[1]
        if np.abs(current_node.pos[0] - goal_pos[0]) < 3 and np.abs(current_node.pos[1] - goal_pos[1]) < 3:
            break
        if current_node.is_visited:
            continue
        explore(current_node)
        current_node.is_visited = True

    x,y = current_node.pos
    path=[[x,y]]
    while not (x == current_pos[0] and y == current_pos[1]):
        x,y = distance[x][y].ancester
        path.append([x,y])
    path = path[-3::-1] if len(path) < 32 else path[-3::-1][:32]

    ###  END CODE HERE  ###
    print(f"path = {path}")
    return path
```

Third, the inplementation of `explore` (corresponding to explore).

```
def explore(node):
    current_x, current_y = node.pos
    ances_x, ances_y = node.ancester
    direc = [(0,1), (0,-1), (-1,0), (1,0)]
    for move_x, move_y in direc:
        if not (0 <= current_x+move_x <= 119 and 0 <= current_y+move_y <= 119):
            break
        if current_map[current_x+move_x][current_y+move_y]:
            break
        next_node = distance[current_x+move_x][current_y+move_y]
        if node.g_cost + 1 < next_node.g_cost:
            next_node.g_cost = node.g_cost + 1
            next_node.steer_cost = np.linalg.norm([current_x-ances_x-move_x, current_y-ances_y-move_y], ord=1) * 4
            next_node.ancester = node.pos
            q.put(next_node())
```

Forth, fill the `reach_goal` function.

```
def explore(node):
    current_x, current_y = node.pos
    ances_x, ances_y = node.ancester
    direc = [(0,1), (0,-1), (-1,0), (1,0)]
    for move_x, move_y in direc:
        if not (0 <= current_x+move_x <= 119 and 0 <= current_y+move_y <= 119):
            break
        if current_map[current_x+move_x][current_y+move_y]:
            break
        next_node = distance[current_x+move_x][current_y+move_y]
        if node.g_cost + 1 < next_node.g_cost:
            next_node.g_cost = node.g_cost + 1
            next_node.steer_cost = np.linalg.norm([current_x-ances_x-move_x, current_y-ances_y-move_y], ord=1) * 4
            next_node.ancester = node.pos
            q.put(next_node())def reach_goal(current_pos, goal_pos):
    """
    Given current position of the robot,
    check whether the robot has reached the goal.

    Arguments:
    current_pos -- A 2D vector indicating the current position of the robot.
    goal_pos -- A 2D vector indicating the position of the goal.

    Return:
    is_reached -- A bool variable indicating whether the robot has reached the goal, where True indicating reached.
    """

    ### START CODE HERE ###
    if np.abs(current_pos[0] - goal_pos[0]) < 20 and np.abs(current_pos[1] - goal_pos[1]) < 20:
        return True
    else:
        return False
    ###  END CODE HERE  ###
    return is_reached
```

Lastly, add some code to `main` and plot the trace of robot.

```
if __name__ == '__main__':
    # Define goal position of the exploration, shown as the gray block in the scene.
    goal_pos = [100, 100]
    controller = DR20API.Controller()

    # Initialize the position of the robot and the map of the world.
    current_pos = controller.get_robot_pos()
    print(f"current_pos = {current_pos}")
    current_map = controller.update_map()
    total_path = []
    # Plan-Move-Perceive-Update-Replan loop until the robot reaches the goal.
    while not reach_goal(current_pos, goal_pos):
        # Plan a path based on current map from current position of the robot to the goal.
        path = A_star(current_map, current_pos, goal_pos)
        total_path.extend(path[0:len(path)-3])
        # Move the robot along the path to a certain distance.
        controller.move_robot(path)
        # Get current position of the robot.
        current_pos = controller.get_robot_pos()
```

```
        print(f"current_pos = {current_pos}")
        # Update the map based on the current information of laser scanner and get the updated map.
        current_map = controller.update_map()

    # Plot and Stop the simulation.
    controller.stop_simulation()

    obstacles_x, obstacles_y = [], []
    for i in range(120):
        for j in range(120):
            if current_map[i][j] == 1:
                obstacles_x.append(i)
                obstacles_y.append(j)

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

    plt.plot(path_x, path_y, "-r")
    plt.plot(current_pos[0], current_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.savefig("task1.pdf")
```

C. Result

If we run the code without add `steer_cost` in $f = g + h$, i.e. we write `node.__call__` function as follows:

```
def __call__(self):
        self.f = self.h_cost+self.g_cost   #   no self.steer_cost
        return self.f, selfPython code
```

After run the code, I get the trace of robot as shown in Figure 2. We can see that there are too many turns, which will cause much time waste.

After adding steer cost, we can see that the trace of robot is much more straight then that before, shown in Figure 3.

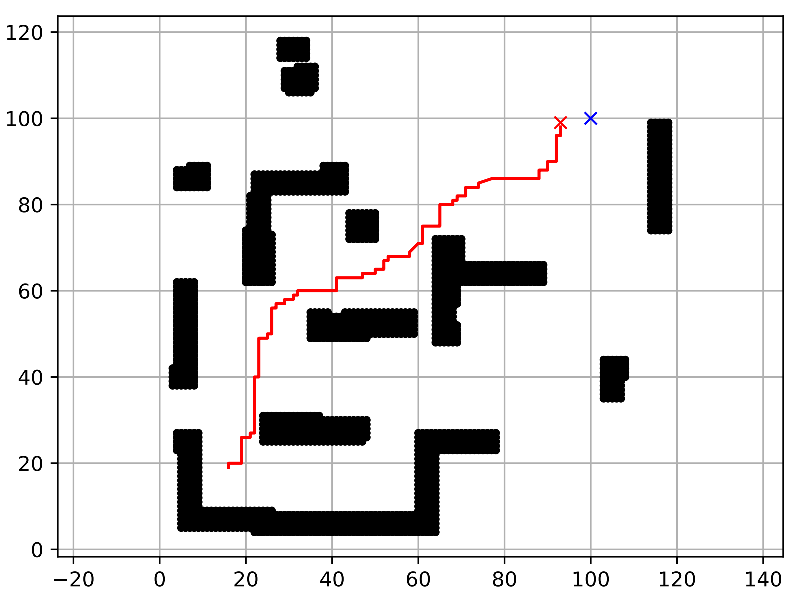

FIG. 2: The trace of robot in Task1 without steer cost.

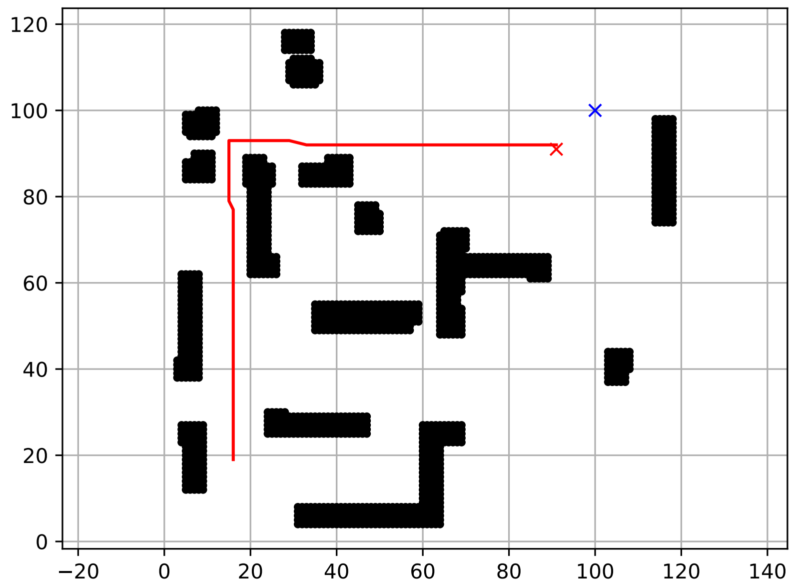

FIG. 3: The trace of robot in Task1 after applying steer cost.

III. TASK2: IMPROVED A* ALGORITHM

A. Model

In Task2, we want to improve the performance of A* planner written in Task1. Specifically, I will do that in three perspective:

- Add four more directions into consideration (upper left, upper right, bottom left, bottom right).
- Add a penalty term when there are some obstacles along each direction.
- Add a penalty term for steering in each position.

First, we need to add four more directions in `explore` function. As we model the cost from one point to its adjacent point to be 1, the cost for this four new movement should be $\sqrt{2}$. It's reasonable since the amount of fuel can be considered decreasing linearly with the distance moved.

What's more, since now we can move with a line to the goal with an angle of 45°, we can not use Manhattan distance as heuristic function any more, since h may be **larger than** h^* in some point, which will lead to wrong choice of path. So we use **Euclidean distance** instead (ℓ_2-norm). Euclidean distance can always be smaller than h^* since the shortest distance from current position to goal is a straight line between them, so this choice makes sense.

Second, I add an `obstc_cost` in `node`, which calculate the number of obstacles in a direction. As shown in Figure 4, I sum over a 5*5 area for each direction (Blue boxes represent `current_node`, gray boxes represent obstacles. There are 8 types of relationships between 5*5 summing area and `current_node`).

Third, I use the relationship between `current_node` and its ancester node with direction, to calculate the steer cost. Since it's nearly impossible to move back to former position, we just use ℓ_1-norm to calculate how many 45° to turn. An example is shown in Figure 5. The position is relative to `current_node`. The

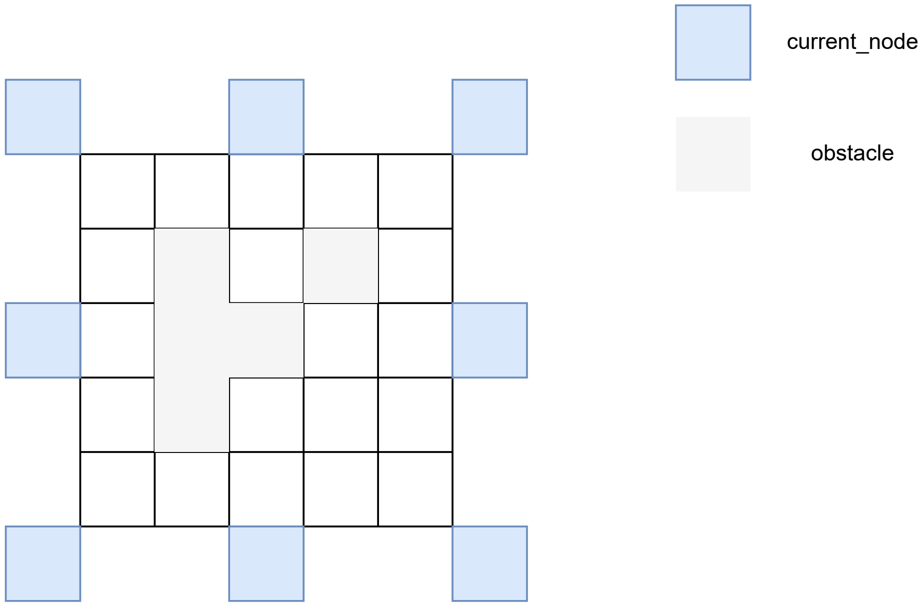

FIG. 4: Calculate penalty term for obstacles.

steer angle

$$\theta = \{|(next\_node.x - current\_node.x) - (current\_node.x - ancester\_node.x)| \\ +|(next\_node.y - current\_node.y) - (current\_node.y - ancester\_node.y)|\} * 45^\circ$$

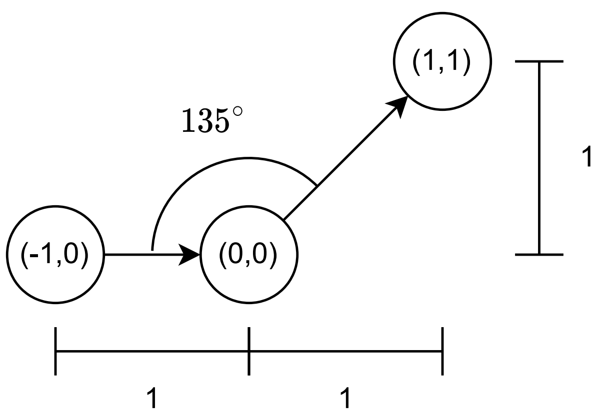

FIG. 5: An example to calculate steer angle.

B. Inplementation

First, I modify the definition of `node`, change heuristic function to be Euclidean, add a function named `obstacle_cost` to calculate penalty term of collision.

```python
MAX_G_COST = 1e3
q = PriorityQueue()
Manhattan_or_Euclidean = 2 # 1 means Manhattan and 2 means Euclidean
random.seed("AI3603")
def h(pos, goal_pos):
    return np.linalg.norm(np.array(pos)-np.array(goal_pos),ord=Manhattan_or_Euclidean)
def obstacles_cost(current_map, current_pos, direc):
    """
    Given current map of the world, direction of the robot, calculate the obstacles cost for each of 8 directions.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating obstacles.
    current_pos -- A 2D vector indicating the current position of the robot.
    direc -- A 2*1 array indicationg the pos after one of [U,UR,R,DR,D,DL,L,UL] movement.

    Returns:
    A double
    """
    distance = 3
    width = 5
    left_right = direc[0]
    up_down = direc[1]
    center = [current_pos[0]+distance*left_right,current_pos[1]+distance*up_down]
    return 1/32*np.sum(current_map[center[0]-width:center[0]+width+1,center[1]-width:center[1]+width+1])

class node:
    def __init__(self, pos, goal_pos):
        self.pos = pos
        self.is_visited = False
        self.ancester = pos
        self.h_cost = h(pos, goal_pos)+random.uniform(0, 3603e-6)
        self.g_cost = MAX_G_COST

    def cost(self):
        return self.h_cost+self.g_cost

    def __call__(self):
        self.f = self.h_cost+self.g_cost
        return self.f, self
```

Second, I modify A* algorithm.

```python
def Improved_A_star(current_map, current_pos, goal_pos):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using improved A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating obstacles.
    current_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 improved A* algorithm.
    """

    ### START CODE HERE ###
    global distance
    distance = [[node([x,y],[100,100]) for y in range(0,120)] for x in range(0,120)]
    source_node = distance[current_pos[0]][current_pos[1]]
    source_node.g_cost = 0
    q.put(source_node())

    while not q.empty():
        current_node = q.get()[1]
        if np.abs(current_node.pos[0] - goal_pos[0]) < 3 and np.abs(current_node.pos[1] - goal_pos[1]) < 3:
            break
        if current_node.is_visited:
            continue
        explore(current_node)
        current_node.is_visited = True

    x,y = current_node.pos
    path=[[x,y]]
    while x != current_pos[0] or y != current_pos[1]:
        x,y = distance[x][y].ancester
        path.append([x,y])
    path = path[-3::-1] if len(path) < 22 else path[-3::-1][:22]
    print(f"path = {path}")
    ### END CODE HERE ###
    return path
```

Third, I modify `explore` function.

```python
def explore(node):
    current_x, current_y = node.pos
    ances_x, ances_y = node.ancester
    direc = [(0,1),(1,1),(1,0),(1,-1),(0,-1),(-1,-1),(-1,0),(-1,1)]
    for i,(move_x, move_y) in enumerate(direc):
        if not (0 <= current_x+move_x <= 119 and 0 <= current_y+move_y <= 119):
            break
        if current_map[current_x+move_x][current_y+move_y]:
            break
        next_node = distance[current_x+move_x][current_y+move_y]
        steer_cost = np.linalg.norm([current_x-ances_x-move_x,current_y-ances_y-move_y],ord=1) * 4
        obstc_cost = obstacles_cost(current_map, node.pos , direc[i])

```

```
        if i % 2 == 0:
            if node.g_cost + 1 + steer_cost + obstc_cost < next_node.g_cost:
                next_node.g_cost = node.g_cost + 1 + steer_cost + obstc_cost
                next_node.ancester = node.pos
                q.put(next_node())
        else:
            if node.g_cost + np.sqrt(2) + steer_cost + obstc_cost < next_node.g_cost:
                next_node.g_cost = node.g_cost + np.sqrt(2) + steer_cost + obstc_cost
                next_node.ancester = node.pos
                q.put(next_node())
```

Other codes nearly remain the same as Task1.

C. Result and Comparison

In Task2, I got a much better result than Task1 shown in Figure 6. Here are some comparisons.

- The computational time of Task2 is larger than Task1. However, since this map is not very large, the increase of time is acceptable.
- The possibility of collision is reduced (although Figure 6 seems to have some collisions, they are mainly due to plot function), in most cases, robot is more away from obstacles.
- The path in Task2 is more close to optimal path, since the total distance or time is lower than Task1.

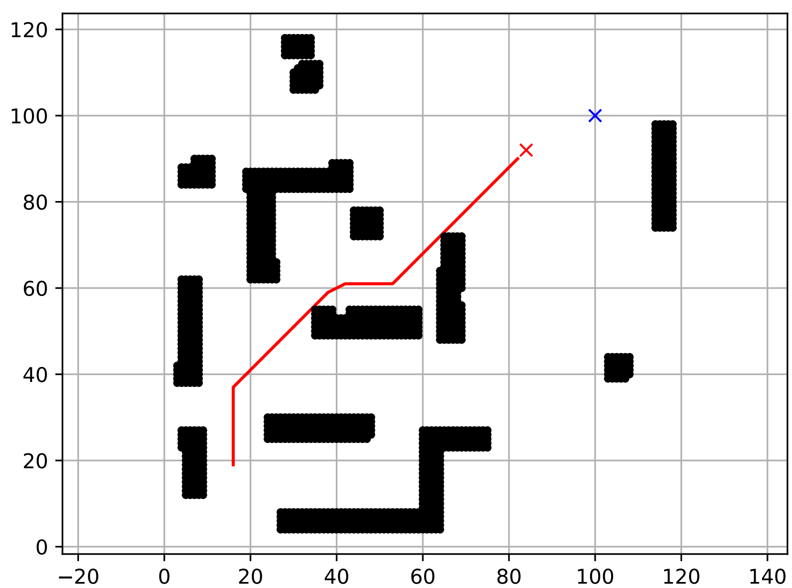

FIG. 6: The trace of robot in Task2.

IV. TASK3: PATH PLANNING FOR SELF-DRIVING

A. Model

In this Task, I modify my code from Task2, and use **Bézier curve** to improve performance, and get a smoother path.

As we know, Bézier curve is widely used in computer science, here is an introduction to it's application in robotics [1].

> *Bézier curves can be used in robotics to produce trajectories of an end-effector due to the virtue of the control polygon's ability to give a clear indication of whether the path is colliding with any nearby obstacle or object. Furthermore, joint space trajectories can be accurately differentiated using Bézier curves. Consequently, the derivatives of joint space trajectories are used in the calculation of the dynamics and control effort (torque profiles) of the robotic manipulator.*

Here is an example [1].

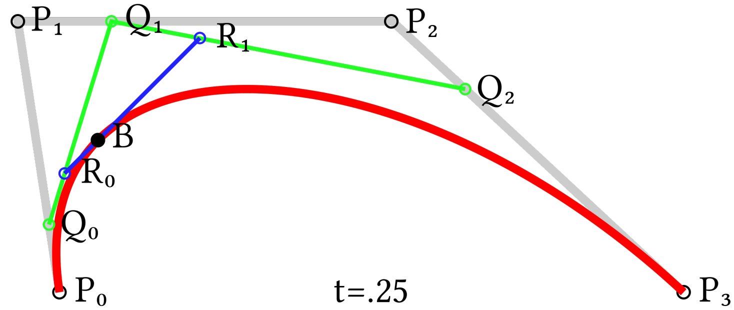

FIG. 7: An example of Bézier curve [1]

B. Inplementation

We need to modify the `Improved_A_Star` function to get a new function `planner`.

```
def planner(current_map, current_pos, goal_pos):
    """
    Given current map of the world, current position of the robot and the position of the goal,
    plan a path from current position to the goal using improved A* algorithm.

    Arguments:
    current_map -- A 120*120 array indicating current map, where 0 indicating traversable and 1 indicating obstacles.
    current_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 improved A* algorithm.
    """

    ### START CODE HERE ###
    global distance
    distance = [[node([x,y],[100,100]) for y in range(0,120)] for x in range(0,120)]
    source_node = distance[current_pos[0]][current_pos[1]]
    source_node.g_cost = 0
    q.put(source_node())

    while not q.empty():
        current_node = q.get()[1]
        if np.abs(current_node.pos[0] - goal_pos[0]) < 3 and np.abs(current_node.pos[1] - goal_pos[1]) < 3:
            break
        if current_node.is_visited:
            continue
        explore(current_node)
        current_node.is_visited = True

    x,y = current_node.pos
    path=[[x,y]]
    while x != current_pos[0] or y != current_pos[1]:
        x,y = distance[x][y].ancester
        path.append([x,y])
    path = path[-3::-1] if len(path) < 22 else path[-3::-1][:22]
    path = np.array(path)

    def bernstein_poly(n, i, t):
        return scipy.special.comb(n, i) * t ** i * (1 - t) ** (n - i)

    def bezier(t, control_points):
        n = len(control_points) - 1
        return np.sum([bernstein_poly(n, i, t) * control_points[i] for i in range(n + 1)], axis=0)
    traj = []
    n_points = 220
```

```
    for t in np.linspace(0, 1, n_points):
        traj.append(bezier(t, path))
    path = traj
    ###  END CODE HERE  ###
    return path
```

C. Result and Comparison

Now the path from start to goal is shown in Figure 8. The most progress of this task is the smoothness. We avoid sharp turn in this task, which can make the speed retain and have more flexibility to avoid collision.

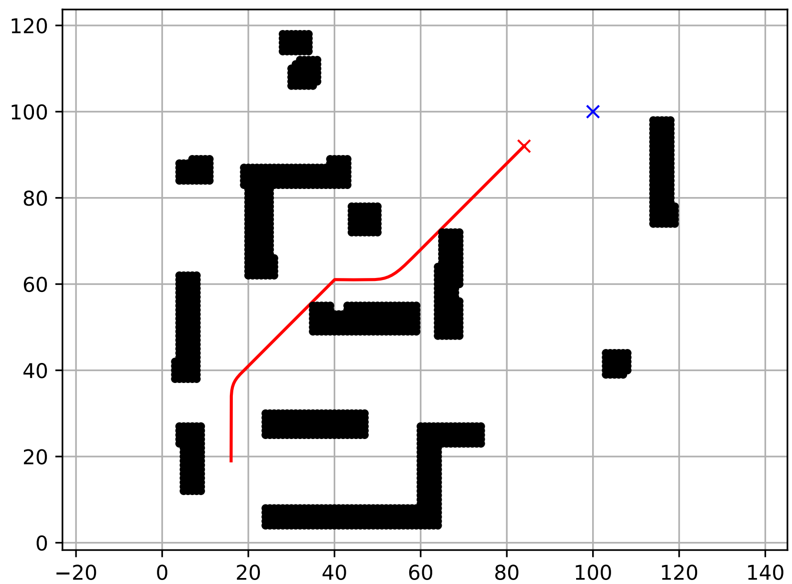

FIG. 8: The trace of robot in Task3

V. CONCLUSION

In this homework, I combine the knowledge from the class and Python code, and build a self-driving car in virtual world. From this homework, I not only learn how to apply AI to a real task, but I get into know how to do optimization, how to analyze, how to model, those are things required for researching.

VI. ACKNOWLEDGEMENT

I thank for Cheng Lei, since he communicate with me and share me some good and useful inspirations (No code).

[1] https://en.wikipedia.org/wiki/B%C3%A9zier_curve