AI 3603 Artificial Intelligence: Principles and Techniques

By: Dexuan He (521030910063)

HW#: 1

October 8, 2023

I. INTRODUCTION

In this homework, we are required to develop a path-planning framework for a service robot in a room using A* algorithm. A 2D global map which marks obstacles is given. We are required to design and implement the path planning algorithm according to the following task requirements.

1. Task1

In this task, we need to implement the basic A* algorithm. The robot's starting position, target position, and global map are all known. The robot can move forward, backward, left, and right. The 120m×120m world map is discretized into a grid map, where the value 0 represents a reachable point (blank) and 1 represents an obstacle (black).

2. Task2

To improve the performance of the A* planner, we are asked to formulate and incorporate the following factors into the A* algorithm:

1. Possibility of moving towards upper left, upper right, bottom left, bottom right
2. Consider the distance between the robot and the obstacle to avoid possible collisions
3. Adding the cost of steering to reduce unnecessary turns in the path

3. Task3

Due to the discretization of the map, the paths obtained by the methods in Task 2 and 3 are not smooth. However, smoothness is usually required to guarantee the comfort and energy effciency of self-driving cars. In this task, we are asked to improve the smoothness of the path. There are four recommended method:

- Polynormial interpolation
- Bezier curve
- Hybrid A* algorithm
- Lattice planner

A. Environment

- Python 3.9.6
- numpy 1.25.1
- matplotlib 3.5.1

II. THEOREM & METHOD

A. A* Algo

1. *Algorithm*

For any heuristic path planning algo, there is a universal pattern:

1. **Find reachable points (the openset) from reached points (the closeset)**
2. **Calculate the cost of each point in the openset**
3. **Move the point with the lowest cost to the closeset**

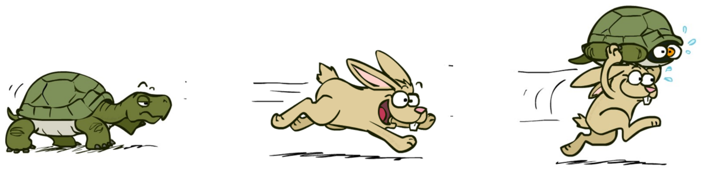


FIG. 1: UCS(l), Greedy BFS(m), and A*(r)

The key factor effected in the planning process is the design of the cost function, which is also the main differences between algos. The Uniform Cost Search focus on the known cost $g(n)$ which characterize the cost to the start, while the Greedy Best-first Search only consider the expected cost to the end, a heuristic function $h(n)$. The A* search combines them and use a cost function $f(n) = g(n) + h(n)$. Thanks to this unique cost function, A* is able to work with a great probability of finding an optimal path, while being highly time-efficient.

2. *Design of cost function*

Let's take a close look into the cost function of A* algo. The known cost $g(n)$ is the cost from the start to current point, which is definite since we have full knowledge of the closeset. On the other hand, the heuristic function $h(n)$ is of multiple choices. Because we don't know exactly where the obstacles are in the unexplored area, we concentrate on the distance from current point to the end in most cases. There're two popular distance used: the Manhattan Dis and the Euclidean Dis.

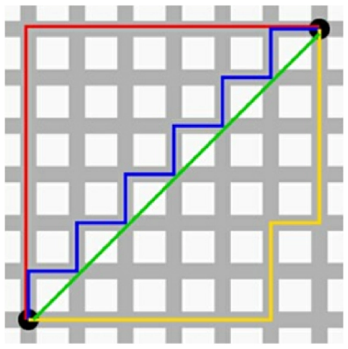


FIG. 2: Manhattan Dis(red, blue and yellow), Euclidean Dis(green)

$$\begin{gathered}
For\ point\ A(x_1, y_1),\ and\ B(x_2, y_2), \\
Manhattan\ Distance:\ \ Dis_M = |x_1 - x_2| + |y_1 - y_2| \\
Euclidean\ Distance:\ \ Dis_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}
\end{gathered}$$

From the fomulas we notice that Euclidean Dis provides a more nuanced picture of distance while Manhattan Dis has a much lower time complexity. Here is the trade off between accuracy and complexity. Besides, the Manhattan path usually goes in 2 directions while the Euclidean prefers to go straight.

Moreover, to improve A* with the consideration of the distance with obstacle and the cost of steering, we can simply add corresponding penalties into the cost function. In detail, we add a penal term to the cost which directly proportional to the number of obstacles in the field of view and add another penal term when the robot changes its direction. Go to the implement part for more concise information of these penalties.

B. Bezier Curve

In task3, we choose to use the Bezier curve to smoothes the path (as polynomial interpolation fail in repeated x and the other two methods have much higher complexity)

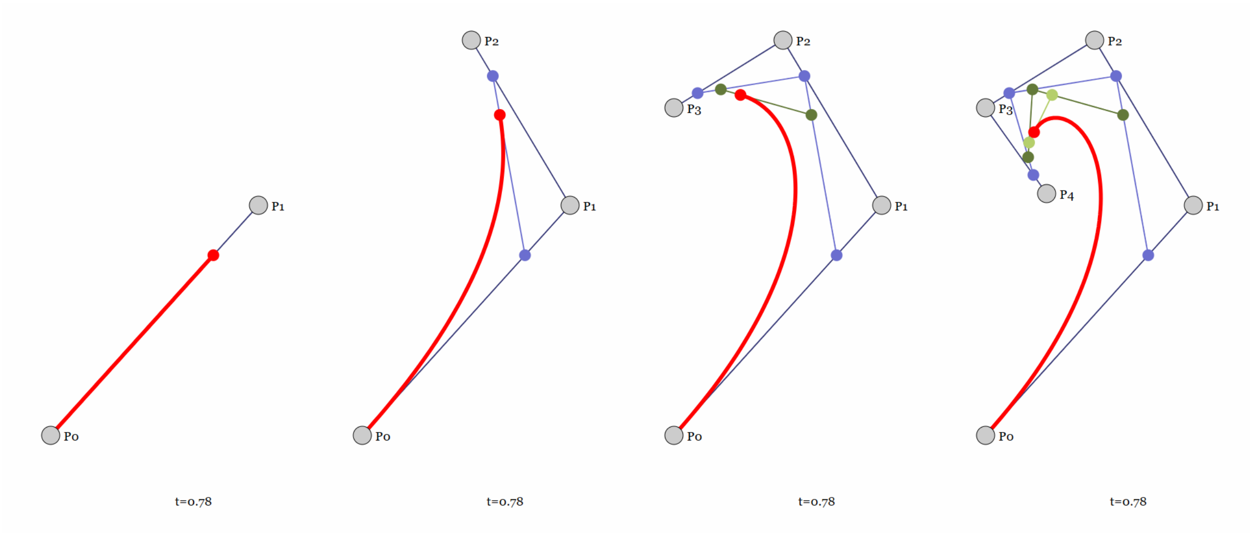


FIG. 3: Bezier curves of 2 to 5 control points

1. Definition and properties

A Bezier curve is a smooth curve generated by several control points. Unlike most popular fitting method, the Bezier curve doesn't have the tendency to pass through all the control points. Instead, these points act as a regulator of the shape of the curve (just as an obstacle or a turning point does). For n+1 control points $P_0, P_1, \ldots, P_n$ (i.e. the order of the curve = n), here's its fomula:

$$B(t) = \sum_{i=0}^{n} \binom{n}{i} P_i (1-t)^{n-i} t^i \tag{1}$$

Besides, Bezier curve has the following properties which makes it a nice choice for smoothes.

1. A Bezier curve can be generate recursively (De Casteljau): $B_{i,n}(t) = (1-t)B_{i,n-1}(t) + tB_{i-1,n-1}(t)$
2. The Bezier curve will always be in the least convex polygon that contains all the control points
3. The first and the last control point are the start and end of the curve, and the curve is tangent with line P_0P_1 and $P_{n-1}P_n$ at the start and end

In this task, we basically use Property 2 and 3.

2. Partial Bezier method

For a given path, now we consider how to use the Bezier curve to smoothes it. First, we notice that only the turning point in the path is not smooth. However, it's inappropriate to take all the turning points as control points to generate a Bezier curve, since the curve has no tendency to pass through them. Therefore, we propose an original method: the partial Bezier method.

This method can by explained in a word: **Divide the path into several parts (turning area and straight area) and apply Bezier to each turning area.**

Concisely, we set a parameter Turning_R as the (half) size of the turning area for each turning point in the origin path. If two turning area intersect, than take their union as a new turning area. Take the borders and turning points of the turning area as control points to generate a Bezier curve.

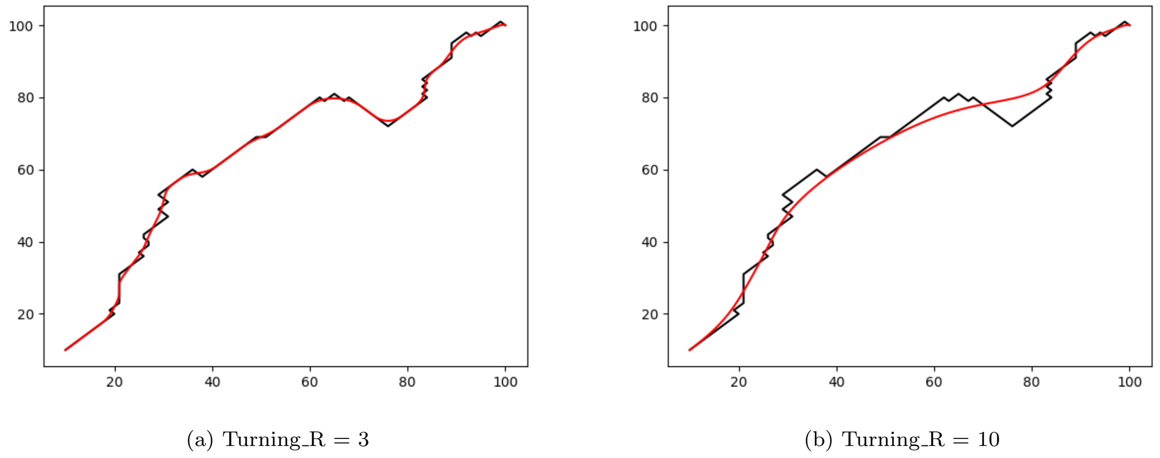


FIG. 4: Smaller radius generates finer curves while bigger radius avoid oscillation

Why it works? We look back to Property 2 and 3. In most cases, the origin path generated by A* turns because of obstacles, and Property 2 ensures that the smoothed path won't hit on these obstacles. At the same time, Property 3 ensures that the curves (or straight lines) of each area can meet smoothly, in other words, ensures the smoothness of the whole path.

Why partial? On one hand, as the number of control points increases, the force of each point declines, resulting in potential collisions from a global Bezier. On the other hand, due to the delicate balance between the intentions to avoid obstacles and reduce path cost, the path generated by A* may have highly frequent turnings, resulting in unwarranted oscillation, which can be avoided by applying Bezier in a relatively big area. Consequently, an appropriate Turning_R is significant to this approach.

III. IMPLEMENT & RESULT

A. Task1

In this task, we implement the original A* algo to get a viable path. We encapsulate the points on the map into a class Node to track its father node as well as to avoid repeating cost calculating. The key module of A* is the expansion of openset, which is partly done by the following function:

```python
def get_nearby(self, world_map, goal_pos):
    nearby = []
    checklist = [(self.x+1, self.y), (self.x-1, self.y), (self.x, self.y+1), (self.x
        , self.y-1)]
    for pos in checklist:
        if valid_pos(world_map, pos):
            nearby.append(Node(pos[0], pos[1], self.g+cost((self.x, self.y), pos), h
                (pos, goal_pos), self))
        else:
            continue
    return nearby
```

Considered that the robot can only move in four directions, we use the Manhattan Distance as $h(n)$. The core logic of A* is implemented as follow:

```python
#init
startN = Node(start_pos[0], start_pos[1], 0, 0, None)
closeset.append(startN)
nowN = startN

while [nowN.x, nowN.y] != goal_pos:
    #append the nearby points to the openset
    nearN = nowN.get_nearby(world_map, goal_pos)
    for newN in nearN:
        if newN in closeset:
            continue
        exist_in_openset = False
        for node in openset:
            if (node.x, node.y) == (newN.x, newN.y):
                exist_in_openset = True
                if node.g > newN.g:
                    node = newN
        if not exist_in_openset:
            openset.append(newN)

    #find the new point of the least cost
    now_index = np.argmin([i.g + i.h for i in openset])
    nowN = openset.pop(now_index)
    closeset.append(nowN)

#use attribute pre to get path
while (nowN.x, nowN.y) != (startN.x, startN.y):
    path.insert(0, [nowN.x, nowN.y])
    nowN = nowN.pre
path.insert(0, [nowN.x, nowN.y])
```

For more detail in Task1, please view the source code file. The result path is show in the following figure, which seems quite stiff, too close to the obstacles.

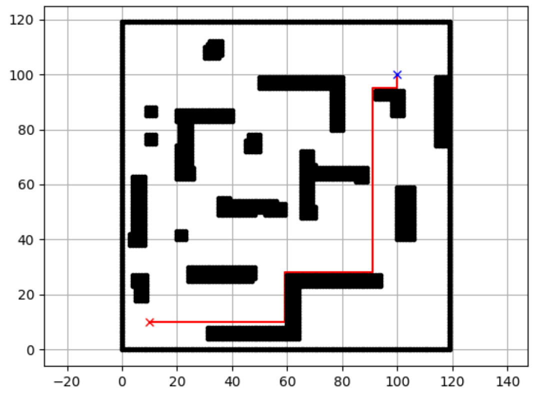


FIG. 5: Task1

B. Task2

We are making the robot moving in all directions, avoid too small distance with obstacles and rapidly changing directions in this task.

For the first requirement, it's far less from enough to just add four corner points in the class function get_nearby(self, world_map, goal_pos). For $g(n)$, the cost for new directions are $\sqrt{2}$, no longer 1. For $h(n)$, the Euclidean Distance should be applied (in theory). We can see that the Euclidean path is closer to the straight line connecting the start and end, resulting in a lower cost.

```
def Manhattan_Distance(now_pos, goal_pos):
    return abs(now_pos[0] - goal_pos[0]) + abs(now_pos[1] - goal_pos[1])

def Euclidean_Distance(now_pos, goal_pos):
    return ((now_pos[0] - goal_pos[0])**2 + (now_pos[1] - goal_pos[1])**2)**0.5
```

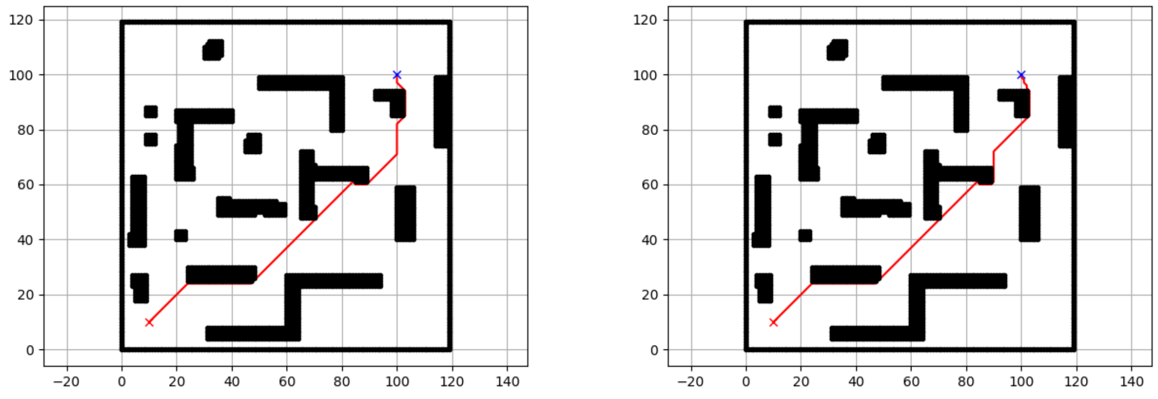

(a) Manhattan Dis (b) Euclidean Dis

FIG. 6: Comparison between path of different h(n)

However, we note Manhattan Distance is still used in the following part because of its lower time complexity.

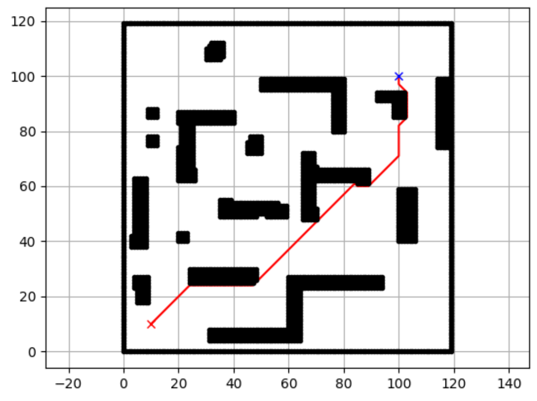
For the second and third requirement, we use the strategy of penalty. We use 3 parameters:

- Safe_Distance: the distance to sense obstacle, i.e. the distance to add penalty
- Penalty_Factor: the weight of obstacles in the field of view
- Steer_Factor: the additional cost of turning

```
#consider 8 dirs and the cost of turning, pos1 is the current position
def cost(pos0, pos1, pos2):
    s = 0
    vec1 = (pos1[0]-pos0[0], pos1[1]-pos0[1])
    vec2 = (pos2[0]-pos1[0], pos2[1]-pos1[1])
    if vec1 != vec2:
        s = Steer_Factor
    if pos1[0] == pos2[0] or pos1[1] == pos2[1]:
        return 1 + s
    else:
        return 1.4 + s

#penalty term of too close to obstacles
def obstacle_penalty(now_pos, world_map):
    p = 0
    dl = (now_pos[0]-Safe_Distance, now_pos[1]-Safe_Distance)
    dr = (now_pos[0]+Safe_Distance, now_pos[1]-Safe_Distance)
    ul = (now_pos[0]-Safe_Distance, now_pos[1]+Safe_Distance)
    ur = (now_pos[0]+Safe_Distance, now_pos[1]+Safe_Distance)
    if in_map(world_map, dl) and in_map(world_map, ur):
        p = Penalty_Factor * np.sum(map[dl[0]:dr[0]+1, dl[1]:ul[1]+1])
    return p

#expected to-goal cost function
def h(now_pos, goal_pos, world_map):
    return Manhattan_Distance(now_pos, goal_pos) + obstacle_penalty(now_pos,
        world_map)
```


(a) Without penalty

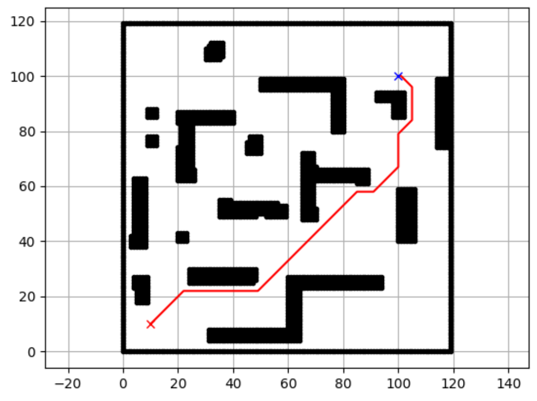

(b) With penalty

FIG. 7: The effect of penalty for obstacle and turning

C. Task3

In this task, we're required to smoothes the path generated in the tasks before. Here we use the partial Bezier method mentioned above, with the parameter Turning_R, and the number of interpolating points between control points.

First we encapsulate the Bezier curve into a class Bezier and calculate the interpolating points using the definition equation.

```python
class Bezier:
    def __init__(self, Points, InterpolationNum):
        self.order = Points.shape[0]-1
        self.num = InterpolationNum
        self.pointsNum = Points.shape[0]
        self.Points = Points

    # get all interpolation points of Bezeir curve
    def getBezierPoints(self):
        PB=np.zeros((self.pointsNum, 2))
        pis =[]
        for u in np.arange(0, 1 + 1/self.num, 1/self.num):
            for i in range(0, self.pointsNum):
                PB[i] = (factorial(self.order)/(factorial(i)*factorial(self.order -
                    i)))*(u**i)*(1-u)**(self.order - i)*self.Points[i]
            pi = sum(PB).tolist()
            pis.append(pi)
        return np.array(pis)
```

Then we use a function get all the turning points from the path

```python
#find the turning points in the path (as well as its index)
def get_turnings(path):
    turnings = []
    turnings_index = []
    for i in range(1, len(path)-1):
        if change_dir(path[i-1], path[i], path[i+1]):
            turnings_index.append(i)
            turnings.append(path[i])
    return turnings, turnings_index
```

The tricky part is how to divide the path into turning areas (and straight areas). Here we need to take the union of intersect areas. Rather than constructing elementary turning areas and detect their intersection, we make it out in a recursive way.

```python
#return a list of curves which consist of lists of control points, use an aux func
def control_points(path):
    turnings, turnings_index = get_turnings(path)
    #the index of current turning point in the turning point list
    index = 0
    result = []
    while index != len(turnings_index):
        cp_set, index = control_points_aux(path, turnings_index, [path[
            turnings_index[index]]], index, False)
        result.append(cp_set)
        index += 1
    return result
```

To get the control points in each area, we conduct forward search at the current turning point (radius = 2 * Turning_R) to detect intersection. If intersect, take the new turning point as current point and call this function recursively.

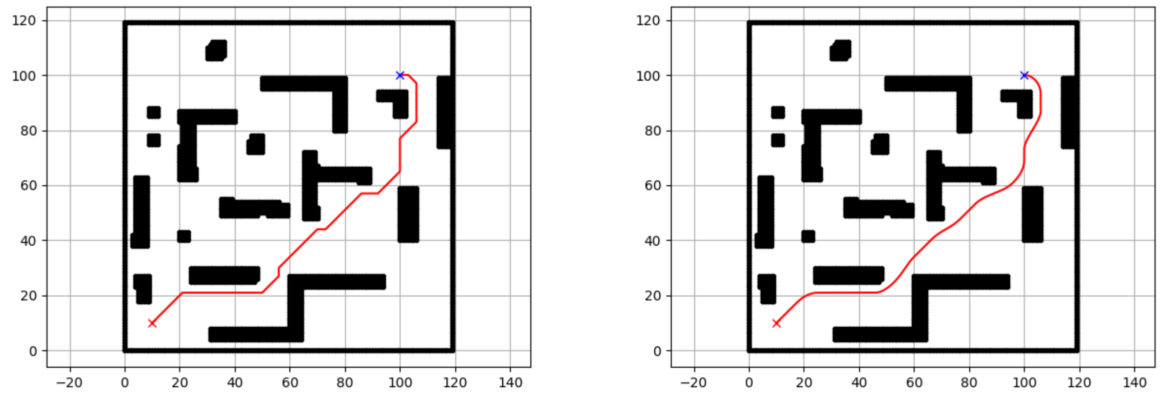
```python
#return a list of control points of current curve as well as the index of the
    current turning point to use a recursion
def control_points_aux(path, turnings_index, cp_set, index, extending):
    index_in_path = turnings_index[index]

    #boundary condition
    #left
    if extending == False:
        if index_in_path < Turning_R:
            cp_set.insert(0, path[0])
        else:
            cp_set.insert(0, path[index_in_path - Turning_R])
    #right
    if index_in_path + 2*Turning_R > len(path) - 1:
        #last turning point
        if index == len(turnings_index) - 1:
            if index_in_path + Turning_R > len(path) - 1:
                cp_set.append(path[-1])
            else:
                cp_set.append(path[index_in_path + Turning_R])
            return cp_set, index
        #not the last
        else:
            cp_set.append(path[turnings_index[index + 1]])
            index += 1
            return control_points_aux(path, turnings_index, cp_set, index, True)
    #the last turning point but not at the boundary
    if index == len(turnings_index) - 1:
        cp_set.append(path[index_in_path + Turning_R])
        return cp_set, index

    #if there's a new turning at this turning area, append it with its turning area
    if index_in_path + 2*Turning_R > turnings_index[index + 1]:
        cp_set.append(path[turnings_index[index + 1]])
        index += 1
        return control_points_aux(path, turnings_index, cp_set, index, True)
    #with no new turnings at current turning area, return
    else:
        cp_set.append(path[index_in_path + Turning_R])
        return cp_set, index
```

Finally, generate a Bezier curve for each turning area (cp_set) and bind them together.

```python
def path_smoother(path):
    curves = []
    cps = control_points(path)
    for cp in cps:
        points = np.array(cp)
        bz = Bezier(points,InterpolationNum)
        curves.append(bz.getBezierPoints())
    smooth_path = curves[0]
    for i in range(1, len(curves)):
        smooth_path = np.append(smooth_path, curves[i], axis=0)
    smooth_path = smooth_path.tolist()
    start_index = path.index(smooth_path[0])
    smooth_path = path[:start_index] + smooth_path
    return smooth_path
```



(a) Without smoothes

(b) With smoothes

FIG. 8: The partial Bezier method

As you can see, the smoother retained some efficient straight roads and wipes out unwanted oscillations while avoiding potential collisions.

IV. DISCUSSION & CONCLUSION

A. Smoothes in planning process

In our solution to smoothness, we apply partial Bezier to the thoroughly generated path after planning. However, after-planning smoothes add uncertainty to the path, which may increase the cost and the probability of collision. That arises the consideration to a smoothes in the planning process. We tried a compromise first, using polynormial fitting at known paths in each iteration and calculate the distance between the forecast point and the points in the openset to add a smooth penal term at the cost function, but this method is of extremely high time complexity and failed to get an optimal path.

A popular way to solve this problem is Hybrid A*, using gradient decent to optimize the following object function to get a smooth optimal path.

$$P = P_{obs} + P_{cur} + P_{smo} + P_{vor} \tag{2}$$

- P_{obs} the penal term to avoid too small distance to obstacles
- P_{cur} the penal term to limit curvature in order to avoid sudden turns
- P_{smo} the penal term to ensure smoothness
- P_{vor} another penal term to avoid obstacles using Voronoi Diagrams

B. Conclusion

In this homework, we complete three tasks about path planning, including basic A* algo, improved A* and path smoothness. We solve the previous two problem mainly by designing and improving the cost function, and propose the partial Bezier method to solve the last one. During the process, we deepen understanding of A* algo and path planning problems, and improve our coding capacity.