AI 3603 Artificial Intelligence: Principles and Techniques

By: Benhao Huang (521030910073)

HW#: 1

October 5, 2023

Contents

1 Introduction

In this lab, students are asked to implement A* algorithm, which is quite important in Path Planning Field. In addition to complete basic A^*, students are also asked to improve A^* algorithm, respectively in aspects of:

- Possibility of moving towards upper left, upper right, bottom left, bottom right.
- Proper distance to obstacles to avoid possible collision.
- Decreased unnecessary turns to save energy.
- Smoother path to guarantee the comfort and energy efficiency of self-driving cars.

These requirements are divided into three tasks, and this report will arrange the content of the article in three tasks.

2 Task1: Implementation of basic A^*

In this section, we will implement basic A^* algorithm, and give together its description as well as formulation in pseudo-code.

2.1 Description

A* is a graph traversal and path search algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency. It's worst-case time complexity is $O(|E|\log|V| = O(b^m)$, and it's worst-case space complexity is $O(|V|) = O(b^m)$. **Here b is *branching factor* and m is *maximum depth.***

Besides, it is also an informed search algorithm, meaning that it is formulated in terms of weighted graphs: starting from a starting node, it aims to find a path to the given goal node with smallest cost. The cost here is designed specifically for different purpose. It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.

At each iteration, A* needs to determine which of its paths to extend by selecting the path that minimizes: $f(n) = g(n) + h(n)$ where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that **estimates** the cost of the cheapest path from current node n to the goal.

More details will be demonstrated in following Formulation Section.

2.2 Formulation

We render its pseudo-code here to demonstrate basic A^* algorithm's formulation. We first illustrate some symbol used in the pseudo-code.

Illustration

- $g(n)$ **function** stores cost of the path from StartNode to the current node n
- $h(n)$ **function** stores the heuristic function map, here we apply Euclidean Distance. This can actually be replaced with L_1-Norm (Manhattan distance) and other possible criterion.
- **openList** stores nodes discovered but remained to be extended; **closedList** stores nodes that have been extended and reach its smallest cost currently.
- **parent(n)** is literally the parent node of n, for backtracking the path when the goalNode is reached.
- The **neighbor of** n refers to the four adjacent points on the top, bottom, left, and right of n. In Task1, we don't add the Possibility of moving towards upper left, upper right, bottom left, bottom right.

Algorithm 1 A* algorithm

```
Initialize Parameters
Initialize Map for A* to run on, openList ← [StartNode], closedList ← [ ], path ← [ ]
Initialize parent(StartNode) ← StartNode, g(StartNode) ← 0

Calculate heuristic function map
for Node in Map do
  if Node is obstacle then
    h(Node) = Infinity
  else
    h(Node) = Euclidean Distance between Node and GoalNode.
  end if
end for

Get A* Path
while openList is not empty do
  n ← None
  for openNode in openList do
    if n is None OR g(openNode) + h(openNode) < g(n) + h(n) then
      n ← openNode
    end if
  end for
  if n is None then
    return No Possible Path
  end if
  if n is goalNode then
    while parent(n) is not startNode do
      append n to path
    end while
    append startNode to path
    return Path found
  end if
  for neighbour of n do
    if neighbour is neither in openList nor in closedList then
      append neighbour to openlist
      parent(neighbour) ← n
      g(neighbour) = g(n)+ Euclidean Distance between n and neighbour
    else
      if g(neighbour) > g(n)+Euclidean Distance between n and neighbour then
        g(neighbour) = g(n)+ Euclidean Distance between n and neihgbour
        parent(neighbour) ← n
        if neighbour in closedList then
          remove neighbour from closedList
          append neighbour to openList
        end if
      end if
    end if
  end for
  remove n from openList
  append n to closedList
end while
```

2.3 Implementation

```python
class A_Star_Map:
    def __init__(self, world_map, start_pos=(10, 10), goal_pos=(100, 100)):
        self.world_map = world_map
        self.map_x, self.map_y = np.shape(world_map)  # 120, 120

        if type(start_pos) is not tuple or type(goal_pos) is not tuple:
            print('start_pos and goal_pos should be tuple')
            self.start_pos = (start_pos[0], start_pos[1])
            self.goal_pos = (goal_pos[0], goal_pos[1])
        else:
            self.start_pos = start_pos
            self.goal_pos = goal_pos

        self.open_list = [self.start_pos]  # start_pos (x,y)
        self.closed_list = []

        self.g = dict()  # dict((x,y): value), cost of the path from start to the current node
        self.g[self.start_pos] = 0  # g(start) = 0

        self.h = dict()  # dict((x,y): value), heuristic function
        self.init_h()

        self.parent_nodes = dict()  # dict((x,y): (x1,y1))
        self.parent_nodes[self.start_pos] = self.start_pos
        self.path = []

    def get_4_neighbours(self, pos):
        neighbours = []
        for i in [-1, 1]:
            neighbour = (pos[0] + i, pos[1])
            if neighbour[0] < 0 or neighbour[0] >= 120 or neighbour[1] < 0 or neighbour[1] >= 120:
                continue
            if self.world_map[neighbour[0]][neighbour[1]] == 1:
                continue
            neighbours.append(neighbour)
        for j in [-1, 1]:
            neighbour = (pos[0], pos[1] + j)
            if neighbour[0] < 0 or neighbour[0] >= 120 or neighbour[1] < 0 or neighbour[1] >= 120:
                continue
            if self.world_map[neighbour[0]][neighbour[1]] == 1:
                continue
            neighbours.append(neighbour)
        return neighbours

    def get_path(self):
        return self.path

    def get_euclidean_distance(self, pos1, pos2):
        return np.sqrt((pos1[0] - pos2[0]) ** 2 + (pos1[1] - pos2[1]) ** 2)

    def get_manhattan_distance(self, pos1, pos2):
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])

    def init_h(self):
        for i in range(self.map_x):
            for j in range(self.map_y):
                if self.world_map[i][j] == 1:
                    continue
                self.h[(i, j)] = self.get_euclidean_distance((i, j), self.goal_pos)

    def A_star(self):
        while (len(self.open_list)) > 0:
            n = None  # current node
            for pos in self.open_list:
                if n is None or self.g[pos] + self.h[pos] < self.g[n] + self.h[n]:
                    n = pos

            if n is None:
                print('no path found')
                return False

            if n == self.goal_pos:
                self.path = []
                while self.parent_nodes[n] != self.start_pos:  # until reach the start position
                    self.path.append(n)
                    n = self.parent_nodes[n]
                self.path.append(self.start_pos)
                self.path.reverse()
                return True

            neighbours = self.get_4_neighbours(n)
            for neighbour in neighbours:
                if neighbour not in self.open_list and neighbour not in self.closed_list:
                    self.open_list.append(neighbour)
                    self.parent_nodes[neighbour] = n
                    self.g[neighbour] = self.g[n] + self.get_euclidean_distance(n, neighbour)
                else:
                    if self.g[neighbour] > self.g[n] + self.get_euclidean_distance(n, neighbour):
                        self.g[neighbour] = self.g[n] + self.get_euclidean_distance(n, neighbour)
                        self.parent_nodes[neighbour] = n
                        if neighbour in self.closed_list:
                            self.closed_list.remove(neighbour)
                            self.open_list.append(neighbour)

            self.open_list.remove(n)
            self.closed_list.append(n)
```

The main code used to implement A^* is shown above. I have referred to the tutorial here [1].

2.4 Results

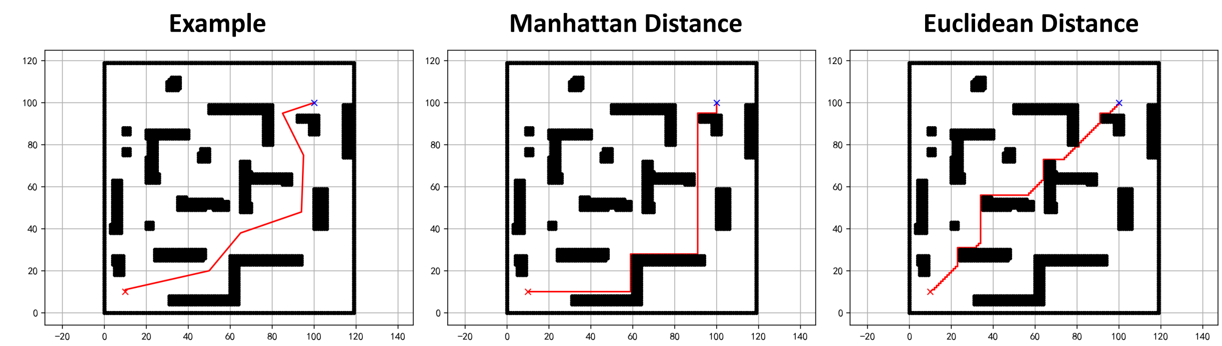

Figure 1: Task 1 results with different methods

In the figure above, "Example" is the demo provided by TA, using fixed points on map. The second figure uses Manhattan Distance as heuristic function while the last figure uses Euclidean Distance.

As we can see, when applying Manhattan Distance, the trajectory tends to be horizontal, flat, and vertical; when applying Euclidean Distance, the trajectory tends to go diagonally, with a lot of turns, but the distance is shorter. This makes sense when we check its mathematical formula:

$$d = \sqrt{(x-x_0)^2 + (y-y_0)^2} \quad \text{Euclidean Distance}$$
$$d = |x-x_0| + |y-y_0| \quad \text{Manhattan Distance}$$

Besides, we notice that the track produced by this naive A^* has some obvious drawbacks:

- the track is too close to the obstacle, which brings great possibility of collision in real scenario.
- there are too many turns in the track produced by Euclidean Distance heuristic function
- the track is hardly smooth enough, with many sharp turns.

3 Task2: Improve A^* with Proper Distance to Obstacles and Less Unnecessary Turns

In this section, we will improve basic A^* algorithm, including:

- Possibility of moving towards upper left, upper right, bottom left, bottom right.
- Proper distance to obstacles to avoid possible collision.
- Decreased unnecessary turns to save energy.

Since the main structure of A^* remains the same, we would only cover some corresponding changes of function and code here, instead of rendering the pesudo-code again.

3.1 Formulation and Implementation

3.1.1 Diagnoal Direction

To add the possibility of moving diagonally, we could just modify the way we get neighbours of a node n.

```python
# new neighbour function, considering diagnoal neighbour
def get_neighbours(self, pos):
    neighbours = []
    for i in range(-1, 2):
        for j in range(-1, 2):
            if i == j == 0:
                continue
            neighbour = (pos[0] + i, pos[1] + j)
            if neighbour[0] < 0 or neighbour[0] >= 120 or neighbour[1] < 0 or neighbour[1] >= 120:
                continue
            if self.world_map[neighbour[0]][neighbour[1]] == 1:
                continue
            neighbours.append(neighbour)
    return neighbours
```

3.1.2 Proper Distance to Obstacles

In order to prevent the track being too close to the obstacle, we could add an extra cost item named **close to obstacle punishment** when initializing heuristic function:

```python
def init_h(self):
    for i in range(self.map_x):
        for j in range(self.map_y):
            if self.world_map[i][j] == 1:
                continue
            self.h[(i, j)] = self.get_euclidean_distance((i, j), self.goal_pos)
            if self.is_close_to_obstacle((i,j)):
                self.h[(i,j)] += self.close_to_obstacle_punishment # punish those near to obstacles

def is_close_to_obstacle(self, pos,threshold = 3): # threshold means cover range or "radius"
    for i in range(-threshold, threshold+1):
        for j in range(-threshold, threshold+1):
            if pos[0] + i < 0 or pos[0] + i >= 120 or pos[1] + j < 0 or pos[1] + j >= 120:
                continue
            if self.world_map[pos[0] + i][pos[1] + j] == 1:
                return True
    return False
```

3.1.3 Decrease Unnecessary Turns

So as to decrease those unnecessary turns made in the track, we could also add an extra cost called **turning punishment**. Details are as follows:

```python
# some changes in A_star(self) function
def A_star(self):
...
neighbours = self.get_neighbours(n)
for neighbour in neighbours:
if neighbour not in self.open_list and neighbour not in self.closed_list:
    self.open_list.append(neighbour)
    self.parent_nodes[neighbour] = n
    self.g[neighbour] = self.g[n] + self.get_euclidean_distance(n, neighbour)
else:
    if self.g[neighbour] > self.g[n] + self.get_euclidean_distance(n, neighbour):
        self.g[neighbour] = self.g[n] + self.get_euclidean_distance(n, neighbour)
        self.parent_nodes[neighbour] = n
        angle = self.turningAngle(n, neighbour)
        self.h[neighbour] += self.turning_punishment * angle / (2 * np.pi)
        if neighbour in self.closed_list:
            self.closed_list.remove(neighbour)
            self.open_list.append(neighbour)


self.open_list.remove(n)
self.closed_list.append(n)
...

def turningAngle(self, pos1, pos2):
    parent = self.parent_nodes[pos1]
    vector1 = (pos1[0] - parent[0], pos1[1] - parent[1])
    vector2 = (pos2[0] - pos1[0], pos2[1] - pos1[1])
    if vector1[0] == vector2[0] and vector1[1] == vector2[1]:
        return 0
    angle = np.arccos((vector1[0] * vector2[0] + vector1[1] * vector2[1]) / (
                np.sqrt(vector1[0] ** 2 + vector1[1] ** 2) * np.sqrt(vector2[0] ** 2 + vector2[1] ** 2)))
    return angle
```

turningAngle(self,pos1,pos2) function will calculate the angle between vector $\overrightarrow{pos1} - \overrightarrow{parent(pos1)}$ and $\overrightarrow{pos2} - \overrightarrow{pos1}$. We set turning punishment according to the magnitude of the angle, that is, punishment = turn punishment ratio $* \, angle/2\pi$.

3.2 Results

The results of improved A^* are displayed as follows:

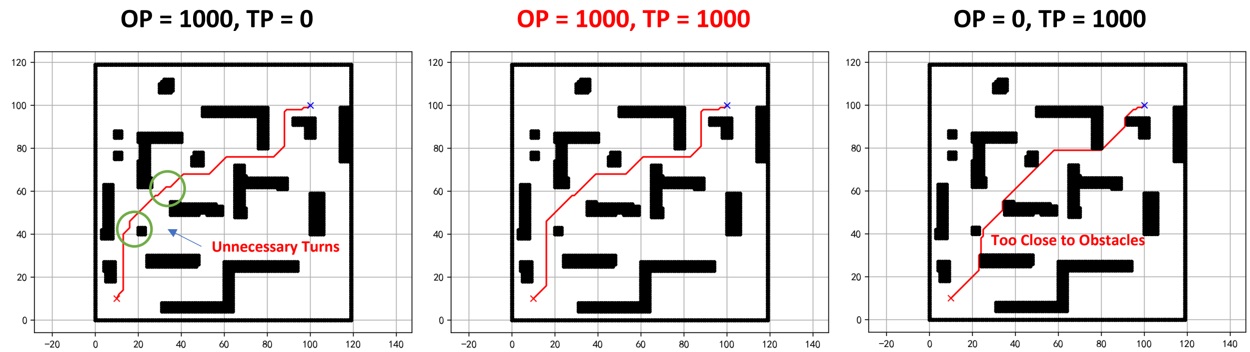

Figure 2: Task 2 results for proper obstacle distance and decreased unnecessary turns

In the figure 2, **OP** stands for close to obstacles punishment, while **TP** stands for turning punishment. Besides, in this section, we apply Euclidean Distance, because it can derive diagonal track, which is comparably shorter in length.

It can be seen from figures that when TP is set to zero, there is a increased number of unnecessary turns. When OP is set to zero, the track derived seemed to be too closed to the obstacles.

However, if we set OP = 1000, TP = 1000, then the track can satisfy all the requirements of task 2: possibility to go diagonally, proper distance from obstacles, together with decreased unnecessary turns.

3.3 Comparison Between Basic A^* and Improved A^*

Considering the changes we make in improved A^*:

1. we add a turning punishment, which is directly added to the heuristic function of a certain node.

2. we add a close to obstacle punishment, which is directly added to the heuristic function of a certain node.

Meanwhile, the function for calculating the turning angle is at most $O(1)$, and also the function for checking whether a node is near obstacles takes constant steps with fixed threshold, thus the time and space complexity should be the same as the basic A^*, which is $O(b^m)$, where b is *branching factor* and m is *maximum depth.*

Besides, by following the same **blocking methods** demonstrated in the lecture 1, the optimality of basic A^* won't be influenced after being improved. (optimality is described on definite heuristic function) . It could be inferred that improved A^* is also complete.

A heuristic h is **admissible** if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal. When we use Euclidean Distance in basic A^*, we ignore the influence of obstacles and simply apply straight line distance, which obvious means $0 \leq h_{basic}(n) \leq h^*(n)$.

And in the case of improved A^*, we add extra cost to $h_{basic}(n)$ to get $h_{improved}(n)$. However, the corresponding $h^*(n)$ should also be perceived differently: for those points that are close to obstacle and those unnecessary turns, its true cost should be set to infinity and such behavior should always be excluded. From this point of view, the heuristic function of improved A^* is still admissible.

According to figure 2, we can clearly see that basic A^* is neither safe nor energy efficient, because the track is too close to obstacles and has many unnecessary turn. Improved A^* is comparably safe and energy efficient.

Table 1: The comparison between basic A^* and improved A^*

Algorithm	Time	Space	Safety	Complete	Optimality	Admissible	Energy Efficient
basic A^*	$O(b^m)$	$O(b^m)$	×	✓	✓	✓	×
Improved A^*	$O(b^m)$	$O(b^m)$	✓	✓	✓	✓	✓

4 Task 4: Improve the Smoothness of the Path

In this section, we will further improve the smoothness of the path to make it more practical to real scenario. The algorithm we use here is Hybrid A^* Algorithm [2].

4.1 Description

Hybrid A* is an extension of the classical A* algorithm designed to take into account the **non-holonomic**[1] nature of a car-like vehicle. So basically, a car model is quite necessary here.

4.2 Modeling Process

To implement this algorithm, we should build a **car model** and learn **ReedsShepp Path** first.

4.2.1 Car Model

Car model could be built as follows, with all the parameters that will be used marked out in the figure.

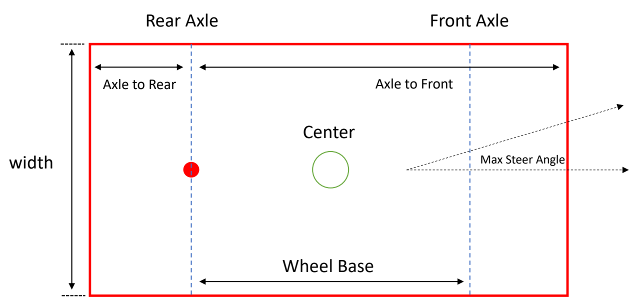

Figure 3: Car Model

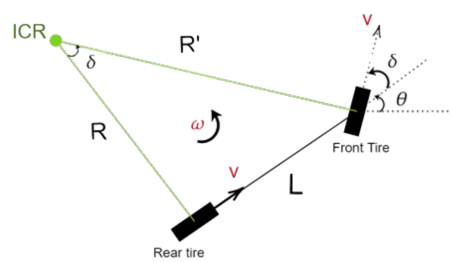

Figure 4: Two-wheeled Bicycle Model

Figure 4 comes from web. In our implementation, we set those parameters as follows:

Table 2: Car Model Parameters

Car Model Parameters	maxSteerAngle	wheelBase	axleToFront	axleToRear	width
Value	0.4 rad	3.5 m	4.5 m	1 m	3 m

[1] Non-holonomic systems are mechanical systems with constraints on their velocity that are not derivable from position constraints

Besides, we should also consider the scenario where car is making turns. The Ackman chassis car can be reduced to a two-wheeled bicycle model as shown in figure 4:

$$\dot{x} = v\cos\theta$$
$$\dot{y} = v\sin\theta$$
$$\dot{\theta} = \frac{v}{L}\tan\delta$$

Where, $v$is the linear speed of the vehicle, θ is the orientation Angle of the vehicle, δ is the steering Angle of the vehicle, L is the wheel base (wb) of the vehicle. Detailed derivation is not the main point here, please check the link at footnote for details [2].

4.2.2 Reeds-Shepp Path

Reeds-Shepp algorithm is put forward by J.A. Reeds and L.A.Shepp, 1990 (optimal path for a car that goes both forward and backwards). This method improved Dubins algorithm, adding the reverse motion (the car is allowed to back up) to the planning, which makes it possible to obtain better solutions than the Dubins curve in certain cases.

Table 3: Notation Table of Reeds-Shepp Path

Base word	Sequences of motion primitives
$C\|C\|C$	$(L^+R^-L^+)\,(L^-R^+L^-)\,(R^+L^-R^+)\,(R^-L^+R^-)$
$CC \mid C$	$(L^+R^+L^-)\,(L^-R^-L^+)\,(R^+L^+R^-)\,(R^-L^-R^+)$
$C \mid CC$	$(L^+R^-L^-)\,(L^-R^+L^+)\,(R^+L^-R^-)\,(R^-L^+R^+)$
CSC	$(L^+S^+L^+)\,(L^-S^-L^-)\,(R^+S^+R^+)\,(R^-S^-R^-)$ $(L^+S^+R^+)\,(L^-S^-R^-)\,(R^+S^+L^+)\,(R^-S^-L^-)$
$CC_\beta \mid C_\beta C$	$\left(L^+R^+_\beta L^-_\beta R^-\right)\left(L^-R^-_\beta L^+_\beta R^+\right)\left(R^+L^+_\beta R^-_\beta L^-\right)\left(R^-L^-_\beta R^+_\beta L^+\right)$
$C\,\|C_\beta C_\beta\|\,C$	$\left(L^+R^-_\beta L^-_\beta R^+\right)\left(L^-R^+_\beta L^+_\beta R^-\right)\left(R^+L^-_\beta R^-_\beta L^+\right)\left(R^-L^+_\beta R^+_\beta L^-\right)$
$C \mid C_{\pi/2}SC$	$\left(L^+R^-_{\pi/2}S^-R^-\right)\left(L^-R^+_{\pi/2}S^+R^+\right)\left(R^+L^-_{\pi/2}S^-L^-\right)\left(R^-L^+_{\pi/2}S^+L^+\right)$ $\left(L^+R^-_{\pi/2}S^-L^-\right)\left(L^-R^+_{\pi/2}S^+L^+\right)\left(R^+L^-_{\pi/2}S^-R^-\right)\left(R^-L^+_{\pi/2}S^+R^+\right)$
$CSC_{\pi/2} \mid C$	$\left(L^+S^+L^+_{\pi/2}R^-\right)\left(L^-S^-L^-_{\pi/2}R^+\right)\left(R^+S^+R^+_{\pi/2}L^-\right)\left(R^-S^-R^-_{\pi/2}L^+\right)$ $\left(R^+S^+L^+_{\pi/2}R^-\right)\left(R^-S^-L^-_{\pi/2}R^+\right)\left(L^+S^+R^+_{\pi/2}L^-\right)\left(L^-S^-R^-_{\pi/2}L^+\right)$
$C\,\|C_{\pi/2}SC_{\pi/2}\|\,C$	$\left(L^+R^-_{\pi/2}S^-L^-_{\pi/2}R^+\right)\left(L^-R^+_{\pi/2}S^+L^+_{\pi/2}R^-\right)$ $\left(R^+L^-_{\pi/2}S^-R^-_{\pi/2}L^+\right)\left(R^-L^+_{\pi/2}S^+R^+_{\pi/2}L^-\right)$

A GIF demo of Reeds-Shepp Path is shown here. And we have referred to this blog for more specific theory details.

4.3 Formulation and Implementation

For the sake of brevity, we only present the pseudo-code form here, for details please refer to the submitted source code. Besides, here we intend to dismiss some details considering the implementation of Reeds-Shepp Path. For the code structures, I have referred to the code here [3].

[2] Ackerman structural chassis

Illustration

- **holonomicMove** means move for holonomic robot, which has only 8-Directions like in Task 2.
- $h(n)$ **function** stores the heuristic function map, here we apply real path length from goalNode to the current node, not merely Euclidean Distance like in Task 2, thus is more complex.
- **openList** stores nodes discovered but remained to be extended; **closedList** stores nodes that have been extended and reach its smallest cost currently.
- **reedsSheppNode** trys to find a reedsSheppPath start from currentNode and end at the goalNode. If found, then return such reedsSheppPath.
- **simulatedNode** is aimed to calculate a possible smooth path from currentNode to its neighbour, whose position is also integer. In this process, car will actually pass through several points whose position could be float, in order to render a smooth path. It simulate the real scenario where car could make turn with different steer angle. This corresponds to the turning model in figure 4. Implementation details are included in source code.

Algorithm 2 Hybrid A* algorithm

Initialize Parameters
Initialize Map for A* to run on, openList ← {startNode index : startNode}, closedList ← [ ],
Initialize costQueue ← priorityQueue()

Calculate heuristic function map
nodeQueue ← priorityQueue((cost of goalNode, goalNode index))
openList ← goalNode index : goalNode, closedList ← { },
while true do
  if openList is empty **then**
    break
  end if
  currentIndex ← nodeQueue.pop()
  currentNode ← openList[currentIndex]
  move currentNode from openList to closedList
  for move in **holonomicMove do**
    neighbourNode ← currentNode make move to
    if neighbourNode is not valid **then**
      continue
    end if
    get neighbourNode index
    if neighbourNode not in closedList **then**
      if neighbourNode in openList **then**
        if neighbourNode.cost < openList[neighbourNode index].cost **then**
          openList[neighbourNode index].cost = neighbourNode.cost
          openList[neighbourNode index].parent = neighbourNode.parent
        end if
      else
        openList[neighbourNode index] = neighbourNode
        nodeQueue.push((neighbourNode.cost, neighbourNodeIndex))
      end if
    end if
  end for
end while

```
Initialize holonomicCost with infinity for obstacles
for node in closedList do
  holonomicCost[node] = node.cost
end for
return holonomicCost
```

Get A* Path

```
openList ← startNode index : goalNode, closedList ← { },
costQueue ← priorityQueue()
costQueue[startNode index] ← startNode.cost + hybridWeight * h[startNode]
while true do
  if openList is empty then
    return None
  end if
  currentNode index ← costQueue.pop()
  currentNode = openList[currentNode index]
  Move currentNode from openList to closedList
  rsNode ← reedsSheppNode(currentNode, goalNode, mapParameters)
  if reedsSheppPath is found then
    closedList[rsNode index] = rsNode
    break
  end if
  if currentNode is goalNode then
    break
  end if
  for move in possible Car Moves do
    simulatedNode ← simulatedNode(currentNode, move, mapParameters)
  end for
  if simulatedNode is not valid then
    continue
  end if
  get simulatedNode index
  if simulatedNode not in closedList then
    if simulatedNode not in openList then
      openList[simulatedNode index] = simulatedNode
      costQueue[simulatedNode index] = simulatedNode.cost + Cost.hybridWeight * h[simulatedNode]
    else
      if simulatedNode.cost < openList[simulatedNodeIndex].cost then
        openList[simulatedNodeIndex] ← simulatedNode
        costQueue[simulatedNodeIndex] ← simulatedNode.cost + Cost.hybridWeight * h[simulatedNode]
      end if
    end if
  end if
end while
path ← backTrack(closedList)
return Path
```

Notes

Expanding nodes when getting heuristic function map is quite time-consuming with original map size. To accelerate the calculation process of heuristic function map, here introduce **"Map Resolution"**. In the implementation code, we set Map Resolution of x-y axis to 4. In this case, every position (x, y) on the map is scaled to $(x/4, y/4)$, which could both guarantee a smooth path as well as a relatively ideal running time in our experiment process.

4.4 Results

By comparing the path derived in Task 1, 2 and 3, we could easily see that the Hybrid A^* algorithm has produced a track that is smooth and safe.

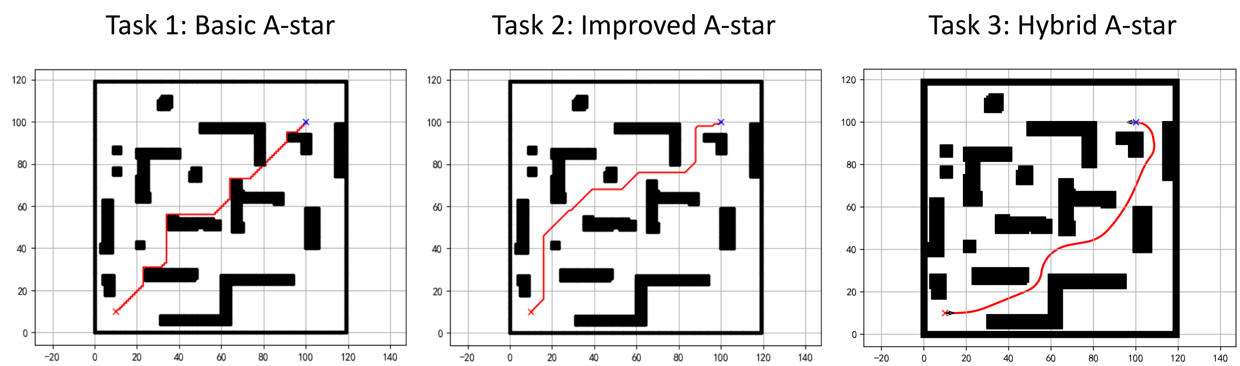

Figure 5: Task 3 results with Hybrid A^*

A GIF demo of the driving car is available here. The car in the GIF may seem a bit close to the obstacle, but there is no collision. The red plot displayed in figure 5 is the red center point of car's rear axle as shown in figure 3.

References

[1] Dimitrije Stamenic David Landup. A* search algorithm. https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/a-star-search-algorithm/.

[2] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo, and James Diebel. Practical search techniques in path planning for autonomous driving. *Ann Arbor*, 1001(48105):18–80, 2008.

[3] RajPShinde. Hybrid-a-star. https://github.com/RajPShinde/Hybrid-A-Star.