

AI 3603 HW1

By: WuShuwen (521030910087)

HW#: 1

October 9, 2023

I. TASK 1: BASIC A* ALGORITHM



(a) Output of Basic A* Algorithm

FIG. 1

A. Implementation

I implemented the basic A* algorithm by defining a class of nodes, named N, and a class of the algorithm itself, named A. Evaluation function, calculated by adding cost function and heuristic function, is included in the class of nodes.

Complete code:

```
1 import sys
2 import os
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')
7
8 ### START CODE HERE ###
9 # This code block is optional. You can define your utility function and class in this block if necessary.
10 class N():
11     def __init__(self, x, y, parent=None) -> None:
12         self.x = x
13         self.y = y
14         self.parent = parent
15         if [x, y] == start_pos:
16             self.parent = None
17             self.g = 0
18         else:
19             self.g = self.parent.g + abs(self.x - self.parent.x) + abs(self.y - self.parent.y)
20             self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])
21             self.f = self.g + self.h
22
23 class A:
24     def __init__(self, map, start, goal) -> None:
25         self.map = map
26         self.start = start
27         self.goal = goal
28         self.openset = []
29         self.closeset = []
30         self.currentN = N(self.start[0], self.start[1])
31         self.openset.append(self.currentN)
32
33     def in_close(self, x, y):
34         for i in self.closeset:
35             if x == i.x and y == i.y:
36                 return True
37         return False
38
39     def in_open(self, x, y):
40         for i in self.openset:
41             if x == i.x and y == i.y:
42                 return True
43         return False
44
45     def iterate(self, x, y, parent):
46         if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
47             return
48         if self.map[x][y] == 1:
```

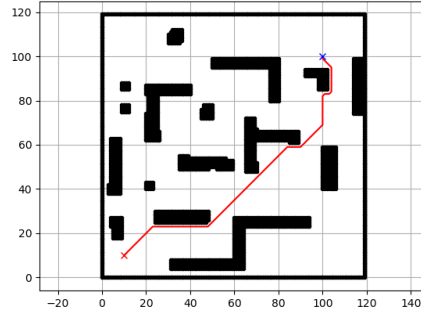
```

49         return
50     if self.in_close(x,y):
51         return
52     if not self.in_open(x,y):
53         n = N(x,y,parent)
54         self.openset.append(n)
55
56     def sel_min(self):
57         min_cost = 1000000
58         num = 0
59         for n in self.openset:
60             if n.f <= min_cost:
61                 min_cost = n.f
62                 sel = n
63         num += 1
64         return sel
65
66     def res(self,n):
67         path = []
68         while True:
69             n=n.parent
70             if n:
71                 path.insert(0,[n.x,n.y])
72                 if (n.x == self.currentN.x) and (n.y == self.currentN.y):
73                     return path
74
75     def algorithm(self):
76         while True:
77             i = self.sel_min()
78             n = self.openset[i]
79             self.closeset.append(n)
80             del self.openset[i]
81             if (n.x == self.goal[0] and n.y == self.goal[1]):
82                 path = self.res(n)
83
84             return path
85             self.iterate(n.x-1,n.y,n)
86             self.iterate(n.x,n.y-1,n)
87             self.iterate(n.x+1,n.y,n)
88             self.iterate(n.x,n.y+1,n)
89
90
91     ### END CODE HERE ###
92
93
94     def A_star(world_map, start_pos, goal_pos):
95         """
96         Given map of the world, start position of the robot and the position of the goal,
97         plan a path from start position to the goal using A* algorithm.
98
99         Arguments:
100         world_map: A 120*120 array indicating map, where 0 indicating traversable and 1 indicating obstacles.
101         start_pos: A 2D vector indicating the start position of the robot.
102         goal_pos: A 2D vector indicating the position of the goal.
103
104         Return:
105         path: A N*2 array representing the planned path by A* algorithm.
106         """
107
108         ### START CODE HERE ###
109         a = A(world_map,start_pos,goal_pos)
110         path = a.algorithm()
111         ### END CODE HERE ###
112         return path
113
114
115
116
117
118
119 if __name__ == '__main__':
120
121     # Get the map of the world representing in a 120*120 array, where 0 indicating traversable and 1 indicating
122     # obstacles.
123     map = np.load(MAP_PATH)
124
125     # Define goal position of the exploration
126     goal_pos = [100, 100]
127
128     # Define start position of the robot.
129     start_pos = [10, 10]
130
131     # Plan a path based on map from start position of the robot to the goal.
132     path = A_star(map, start_pos, goal_pos)
133
134     # Visualize the map and path.
135     obstacles_x, obstacles_y = [], []
136     for i in range(120):
137         for j in range(120):
138             if map[i][j] == 1:
139                 obstacles_x.append(i)
140                 obstacles_y.append(j)
141
142     path_x, path_y = [], []
143     for path_node in path:
144         path_x.append(path_node[0])
145         path_y.append(path_node[1])
146
147     plt.plot(path_x, path_y, "-r")
148     plt.plot(start_pos[0], start_pos[1], "xr")
149     plt.plot(goal_pos[0], goal_pos[1], "xb")
150     plt.plot(obstacles_x, obstacles_y, ".k")

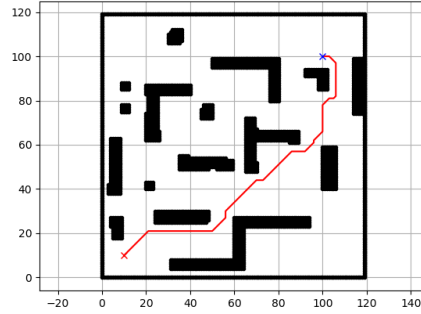
```

```
151 plt.grid(True)
    plt.axis("equal")
    plt.show()
```

II. TASK 2: IMPROVED A* ALGORITHM



(a) Output of improved A* algorithm when lower limit for the distance to obstacle set to 1



(b) Output of improved A* algorithm when lower limit for the distance to obstacle set to 3

FIG. 2

A. Implementation

The implementation of improved A* algorithm is based on that of basic A* algorithm. Compared to the basic one, the path is enabled to upper left, upper right, bottom left, bottom right. The cost of doing so is set to be $\sqrt{2}$.

```

2   if abs(self.x_change) + abs(self.y_change) == 1:
3       self.g = self.parent.g + 1
4       else:
5           self.g = self.parent.g + math.sqrt(2)

```

To avoid collision, I simply set a lower limit for the distance to obstacle, by creating a new map where the blocks considered dangerous for being too close to obstacles are also abandoned.

```

def gen_obstacle_map(self):
2   obstacle_map = self.map.copy()
3   for i in range(1, 119):
4       for j in range(1, 119):
5           if self.map[i][j] + self.map[i - 1][j] + \

```

```

6         self.map[i+1][j] + self.map[i][j-1] + \
8         self.map[i-1][j-1] + self.map[i+1][j-1] + \
        self.map[i][j+1] + self.map[i-1][j+1] + \
        self.map[i+1][j+1] > 0:
10             obstacle_map[i][j] = 1
        self.map = obstacle_map

```

To avoid steering, I added a steering cost to the evaluation, so that turns of acute angles would be punished. I implemented this with a second time difference of the path, that is, by assessing the difference of the direction from the parent node to the current node and that from the current node to the next node.

```

1 self.x_change = self.x - self.parent.x
  self.y_change = self.y - self.parent.y
3
  if [self.parent.x, self.parent.y] != start_pos and abs(self.x_change-self.parent.x_change)+abs(self.y_change-self.
    parent.y_change)>1:
5      self.s=1

```

Complete code:

```

1 import sys
  import os
3 import numpy as np
  import matplotlib.pyplot as plt
5
  MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')
7
  ### START CODE HERE ###
9 # This code block is optional. You can define your utility function and class in this block if necessary.
  import math
11 class N():
    def __init__(self, x, y, parent=None) -> None:
13         self.x = x
          self.y = y
15         self.parent = parent
17
          self.s = 0
          if [x, y] == start_pos:
19             self.parent = None
              self.g = 0
21         else:
          self.x_change = self.x - self.parent.x
          self.y_change = self.y - self.parent.y
23             if abs(self.x_change) + abs(self.y_change) == 1:
                self.g = self.parent.g + 1
25             else:
          self.g = self.parent.g + math.sqrt(2)
          if [self.parent.x, self.parent.y] != start_pos and abs(self.x_change-self.parent.x_change)+abs(self.
            y_change-self.parent.y_change)>1:
27                 self.s=1
29         self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])
31         self.f = self.g + self.h + self.s
33
  class A:
    def __init__(self, map, start, goal) -> None:
35         self.map = map
37
          obstacle_map = self.map.copy()
          for i in range(1, 119):
39              for j in range(1, 119):
                  if self.map[i][j] + self.map[i-1][j] + \
41                     self.map[i+1][j] + self.map[i][j-1] + \
43                     self.map[i-1][j-1] + self.map[i+1][j-1] + \
45                     self.map[i][j+1] + self.map[i-1][j+1] + \
47                     self.map[i+1][j+1] > 0:
                        obstacle_map[i][j] = 1
          self.map = obstacle_map
          self.start = start
          self.goal = goal
          self.openset = []
          self.closeset = []
          self.currentN = N(self.start[0], self.start[1])
          self.openset.append(self.currentN)
53
55
          def in_close(self, x, y):
              for i in self.closeset:
57                  if x == i.x and y==i.y:
                      return True
              return False
61
          def in_open(self, x, y):
              for i in self.openset:
63                  if x == i.x and y==i.y:
                      return True
              return False
65
67
          def iterate(self, x, y, parent):

```

```

69         if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
70             return
71         if self.map[x][y] == 1:
72             return
73         if self.in_close(x,y):
74             return
75         if not self.in_open(x,y):
76             n = N(x,y,parent)
77
78         self.openset.append(n)
79
80     def sel_min(self):
81         min_cost = 1000000
82         num = 0
83         for n in self.openset:
84             if n.f <= min_cost:
85                 min_cost = n.f
86                 sel = n
87         num += 1
88         return sel
89
90     def res(self,n):
91         path = []
92         while True:
93             n=n.parent
94             if n:
95                 path.insert(0,[n.x,n.y])
96             if (n.x == self.currentN.x) and (n.y == self.currentN.y):
97                 return path
98
99     def algorithm(self):
100         while True:
101             i = self.sel_min()
102             n = self.openset[i]
103             self.closeset.append(n)
104             del self.openset[i]
105             if (n.x == self.goal[0] and n.y == self.goal[1]):
106                 path = self.res(n)
107
108             return path
109             self.iterate(n.x+1,n.y+1,n)
110             self.iterate(n.x-1,n.y+1,n)
111             self.iterate(n.x+1,n.y-1,n)
112             self.iterate(n.x-1,n.y-1,n)
113             self.iterate(n.x-1,n.y,n)
114             self.iterate(n.x,n.y-1,n)
115             self.iterate(n.x+1,n.y,n)
116             self.iterate(n.x,n.y+1,n)
117
118     ### END CODE HERE ###
119
120
121 def Improved_A_star(world_map, start_pos, goal_pos):
122     """
123     Given map of the world, start position of the robot and the position of the goal,
124     plan a path from start position to the goal using improved A* algorithm.
125
126     Arguments:
127     world_map: A 120*120 array indicating map, where 0 indicating traversable and 1 indicating obstacles.
128     start_pos: A 2D vector indicating the start position of the robot.
129     goal_pos: A 2D vector indicating the position of the goal.
130
131     Return:
132     path: A N*2 array representing the planned path by improved A* algorithm.
133     """
134
135     ### START CODE HERE ###
136
137     a = A(world_map,start_pos,goal_pos)
138     path = a.algorithm()
139
140     ### END CODE HERE ###
141     return path
142
143
144
145
146
147 if __name__ == '__main__':
148
149     # Get the map of the world representing in a 120*120 array, where 0 indicating traversable and 1 indicating
150     # obstacles.
151     map = np.load(MAP_PATH)
152
153     # Define goal position of the exploration
154     goal_pos = [100, 100]
155
156     # Define start position of the robot.
157     start_pos = [10, 10]
158
159     # Plan a path based on map from start position of the robot to the goal.
160     path = Improved_A_star(map, start_pos, goal_pos)
161
162     # Visualize the map and path.
163     obstacles_x, obstacles_y = [], []
164     for i in range(120):
165         for j in range(120):
166             if map[i][j] == 1:
167                 obstacles_x.append(i)
168                 obstacles_y.append(j)
169

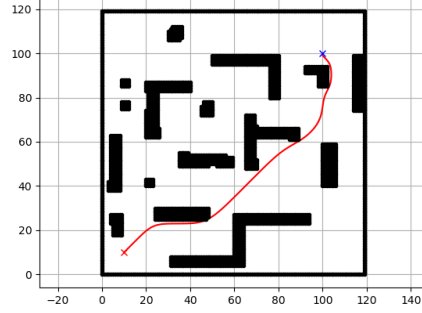
```

```
171     path_x, path_y = [], []
172     for path_node in path:
173         path_x.append(path_node[0])
174         path_y.append(path_node[1])
175
176     plt.plot(path_x, path_y, "-r")
177     plt.plot(start_pos[0], start_pos[1], "xr")
178     plt.plot(goal_pos[0], goal_pos[1], "xb")
179     plt.plot(obstacles_x, obstacles_y, ".k")
180     plt.grid(True)
181     plt.axis("equal")
182     plt.show()
```

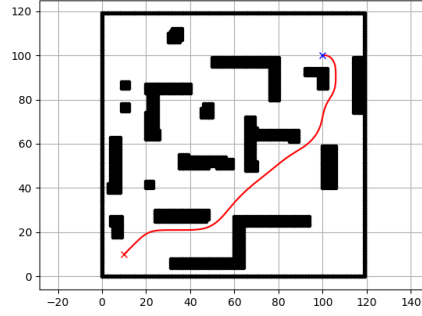

B. Comparison

Compared to the basic one, the improved A* algorithm keeps a sufficient distance from obstacles, thus guarantees safety. In respect of optimality, the improved algorithm does better in this map, though it is restricted by the anti-collision limits, because it can go upper left, upper right, bottom left, bottom right.

III. TASK 3: PATH PLANNING FOR SELF-DRIVING



(a) Output of improved A* algorithm with Bezier curve when lower limit of distance to obstacle set to 1



(b) Output of improved A* algorithm with Bezier curve when lower limit of distance to obstacle set to 3

FIG. 3

A. Implementation

To smooth the path, I simply use the path produced by the improved A* algorithm as control points of Bezier curve to form a smooth path.

I implemented Bezier curve according to its definition:

$$C_n(u) = \sum_{i=0}^n P_i b_{n,i}(u) \quad (1)$$

Where, $b_{n,i}$ represents the Bernstein polynomial:

$$b_{n,i}(u) = C_n^i t^i (1-t)^{(n-i)} \quad (2)$$

```

2 def bernstein_poly(self, i, n, t):
    return np.math.comb(n, i) * (1 - t)**(n - i) * t**i
4 def bezier_curve(self, control_points):
    num_points = 100
    t = np.linspace(0, 1, num_points)
    n = len(control_points) - 1
    curve_points = []
    for i in range(num_points):
        x = sum(self.bernstein_poly(j, n, t[i]) * control_points[j][0] for j in range(n + 1))
        y = sum(self.bernstein_poly(j, n, t[i]) * control_points[j][1] for j in range(n + 1))
        curve_points.append((x, y))
14 return curve_points

```

As is shown in the pictures, the smoothed path whose lower limit is set to 1 turns out to be too close to obstacles, while the one with lower limit set to 3 still performs well.

Complete code:

```

import sys
2 import os
import numpy as np
4 import matplotlib.pyplot as plt
6 MAP_PATH = os.path.join(os.path.dirname(os.path.abspath(__file__)), '3-map/map.npy')
8
9 ### START CODE HERE ###
10 # This code block is optional. You can define your utility function and class in this block if necessary.
import math
12 class N():
    def __init__(self, x, y, parent=None) -> None:
14         self.x = x
15         self.y = y
16         self.parent = parent
17         self.s = 0
18         if [x, y] == start_pos:
19             self.parent = None
20             self.g = 0
21         else:
22             self.x_change = self.x - self.parent.x
23             self.y_change = self.y - self.parent.y
24             if abs(self.x_change) + abs(self.y_change) == 1:
25                 self.g = self.parent.g + 1
26             else:
27                 self.g = self.parent.g + math.sqrt(2)
28             if [self.parent.x, self.parent.y] != start_pos and abs(self.x_change - self.parent.x_change) + abs(self.y_change - self.parent.y_change) > 1:
29                 self.s = 1
30         self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])
31
32         self.f = self.g + self.h + self.s
34
35 class A:
36     def __init__(self, map, start, goal) -> None:
37         self.map = map
38
39         self.gen_obstacle_map()
40         self.gen_obstacle_map()
41         self.gen_obstacle_map()
42
43         self.start = start
44         self.goal = goal
45         self.openset = []
46         self.closeset = []
47         self.currentN = N(self.start[0], self.start[1])
48         self.openset.append(self.currentN)
49
50     def gen_obstacle_map(self):
51         obstacle_map = self.map.copy()
52         for i in range(1, 119):
53             for j in range(1, 119):
54                 if self.map[i][j] + self.map[i - 1][j] + \
55                    self.map[i + 1][j] + self.map[i][j - 1] + \
56                    self.map[i - 1][j - 1] + self.map[i + 1][j - 1] + \
57                    self.map[i][j + 1] + self.map[i - 1][j + 1] + \
58                    self.map[i + 1][j + 1] > 0:
59                     obstacle_map[i][j] = 1
60         self.map = obstacle_map
61
62     def in_close(self, x, y):
63         for i in self.closeset:
64             if x == i.x and y == i.y:
65                 return True
66         return False
67
68     def in_open(self, x, y):
69         for i in self.openset:
70             if x == i.x and y == i.y:
71                 return True
72

```

```

74         return False
75
76     def iterate(self, x, y, parent):
77         if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
78             return
79         if self.map[x][y] == 1:
80             return
81         if self.in_close(x, y):
82             return
83         if not self.in_open(x, y):
84             n = N(x, y, parent)
85
86             self.openset.append(n)
87
88     def sel_min(self):
89         min_cost = 1000000
90         num = 0
91         for n in self.openset:
92             if n.f <= min_cost:
93                 min_cost = n.f
94                 sel = num
95             num += 1
96         return sel
97
98     def bernstein_poly(self, i, n, t):
99         return np.math.comb(n, i) * (1 - t)**(n - i) * t**i
100
101     def bezier_curve(self, control_points):
102         num_points = 100
103         t = np.linspace(0, 1, num_points)
104         n = len(control_points) - 1
105         curve_points = []
106         for i in range(num_points):
107             x = sum(self.bernstein_poly(j, n, t[i]) * control_points[j][0] for j in range(n + 1))
108             y = sum(self.bernstein_poly(j, n, t[i]) * control_points[j][1] for j in range(n + 1))
109             curve_points.append((x, y))
110
111         return curve_points
112
113     def res(self, n):
114         path = []
115         while True:
116             n = n.parent
117             if n:
118                 path.insert(0, [n.x, n.y])
119             if (n.x == self.currentN.x) and (n.y == self.currentN.y):
120                 return self.bezier_curve(path)
121             #return path
122
123     def algorithm(self):
124         while True:
125             i = self.sel_min()
126             n = self.openset[i]
127             self.closeset.append(n)
128             del self.openset[i]
129             if (n.x == self.goal[0] and n.y == self.goal[1]):
130                 path = self.res(n)
131
132                 return path
133                 self.iterate(n.x + 1, n.y + 1, n)
134                 self.iterate(n.x - 1, n.y + 1, n)
135                 self.iterate(n.x + 1, n.y - 1, n)
136                 self.iterate(n.x - 1, n.y - 1, n)
137                 self.iterate(n.x - 1, n.y, n)
138                 self.iterate(n.x, n.y - 1, n)
139                 self.iterate(n.x + 1, n.y, n)
140                 self.iterate(n.x, n.y + 1, n)
141
142     ### END CODE HERE ###
143
144
145 def self_driving_path_planner(world_map, start_pos, goal_pos):
146     """
147     Given map of the world, start position of the robot and the position of the goal,
148     plan a path from start position to the goal.
149
150     Arguments:
151     world_map: A 120*120 array indicating map, where 0 indicating traversable and 1 indicating obstacles.
152     start_pos: A 2D vector indicating the start position of the robot.
153     goal_pos: A 2D vector indicating the position of the goal.
154
155     Return:
156     path: A N*2 array representing the planned path.
157     """
158
159     ### START CODE HERE ###
160
161     a = A(world_map, start_pos, goal_pos)
162     path = a.algorithm()
163
164     ### END CODE HERE ###
165     return path
166
167
168
169
170
171 if __name__ == '__main__':
172     # 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)
176
# Define goal position
178 goal_pos = [100, 100]
180
# Define start position of the robot.
start_pos = [10, 10]
182
# Plan a path based on map from start position of the robot to the goal.
184 path = self-driving-path-planner(map, start_pos, goal_pos)
186
# Visualize the map and path.
obstacles_x, obstacles_y = [], []
188 for i in range(120):
    for j in range(120):
190         if map[i][j] == 1:
192             obstacles_x.append(i)
            obstacles_y.append(j)
194
path_x, path_y = [], []
196 for path_node in path:
    path_x.append(path_node[0])
    path_y.append(path_node[1])
198
plt.plot(path_x, path_y, "-r")
200 plt.plot(start_pos[0], start_pos[1], "xr")
plt.plot(goal_pos[0], goal_pos[1], "xb")
202 plt.plot(obstacles_x, obstacles_y, ".k")
plt.grid(True)
204 plt.axis("equal")
plt.show()

```