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.

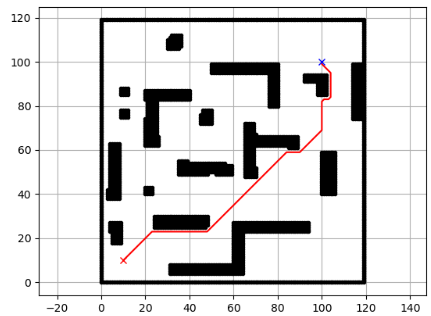
Complete code:

```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt

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 N():
    def __init__(self,x,y,parent=None) -> None:
        self.x =x
        self.y =y
        self.parent = parent
        if [x,y] == start_pos:
            self.parent = None
            self.g = 0
        else:
            self.g = self.parent.g + abs(self.x - self.parent.x) + abs(self.y - self.parent.y)
        self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])
        self.f =self.h+self.g

class A:
    def __init__(self,map,start,goal) -> None:
        self.map =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)

    def in_close(self,x,y):
        for i in self.closeset:
            if x == i.x and y==i.y:
                return True
        return False

    def in_open(self,x,y):
        for i in self.openset:
            if x == i.x and y==i.y:
                return True
        return False

    def iterate(self,x,y,parent):
        if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
            return
        if self.map[x][y] == 1:
```

```python
            return
        if self.in_close(x,y):
            return
        if not self.in_open(x,y):
            n = N(x,y,parent)
            self.openset.append(n)

    def sel_min(self):
        min_cost = 1000000
        num =0
        for n in self.openset:
            if n.f <=min_cost:
                min_cost = n.f
                sel = num
            num +=1
        return sel

    def res(self,n):
        path = []
        while True:
            n=n.parent
            if n:
                path.insert(0,[n.x,n.y])
            if (n.x == self.currentN.x) and (n.y == self.currentN.y):
                return path

    def algorithm(self):
        while True:
            i = self.sel_min()
            n =self.openset[i]
            self.closeset.append(n)
            del self.openset[i]
            if(n.x == self.goal[0] and n.y == self.goal[1]):
                path = self.res(n)

                return path
            self.iterate(n.x-1,n.y,n)
            self.iterate(n.x,n.y-1,n)
            self.iterate(n.x+1,n.y,n)
            self.iterate(n.x,n.y+1,n)



###  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 map, where 0 indicating traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the start 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 ###
    a = A(world_map,start_pos,goal_pos)
    path = a.algorithm()
    ###  END CODE HERE  ###
    return path




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()
```

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 $\sqrt{2}$.

```
 if  abs(self.x_change) + abs(self.y_change) == 1:
                self.g = self.parent.g + 1
            else:
                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):
        obstacle_map = self.map.copy()
        for i in range(1, 119):
            for j in range(1, 119):
                if self.map[i][j] + self.map[i - 1][j] + \
```

```
                    self.map[i+1][j] + self.map[i][j-1] + \
                    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:
                     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.

```
self.x_change = self.x - self.parent.x
self.y_change = self.y - self.parent.y

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:
    self.s=1
```

Complete code:

```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt

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.
import math
class N():
    def __init__(self,x,y,parent=None) -> None:
        self.x =x
        self.y =y
        self.parent = parent

        self.s = 0
        if [x,y] == start_pos:
            self.parent = None
            self.g = 0
        else:
            self.x_change = self.x - self.parent.x
            self.y_change = self.y - self.parent.y
            if  abs(self.x_change) + abs(self.y_change) == 1:
                self.g = self.parent.g + 1
            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:
                    self.s=1
        self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])

        self.f = self.g + self.h + self.s

class A:
    def __init__(self,map,start,goal) -> None:
        self.map =map

        obstacle_map = self.map.copy()
        for i in range(1, 119):
            for j in range(1, 119):
                if self.map[i][j] + self.map[i - 1][j] + \
                    self.map[i+1][j] + self.map[i][j-1] + \
                    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:
                     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)


    def in_close(self,x,y):
        for i in self.closeset:
            if x == i.x and y==i.y:
                return True
        return False

    def in_open(self,x,y):
        for i in self.openset:
            if x == i.x and y==i.y:
                return True
        return False

    def iterate(self,x,y,parent):
```

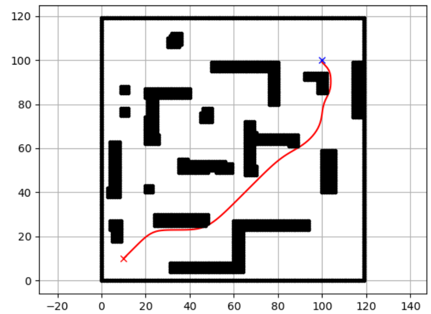
```
        if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
            return
        if self.map[x][y] == 1:
            return
        if self.in_close(x,y):
            return
        if not self.in_open(x,y):
            n = N(x,y,parent)

            self.openset.append(n)

    def sel_min(self):
        min_cost = 1000000
        num =0
        for n in self.openset:
            if n.f <=min_cost:
                min_cost = n.f
                sel = num
            num +=1
        return sel

    def res(self,n):
        path = []
        while True:
            n=n.parent
            if n:
                path.insert(0,[n.x,n.y])
            if (n.x == self.currentN.x) and (n.y == self.currentN.y):
                return path

    def algorithm(self):
        while True:
            i = self.sel_min()
            n =self.openset[i]
            self.closeset.append(n)
            del self.openset[i]
            if(n.x == self.goal[0] and n.y == self.goal[1]):
                path = self.res(n)

                return path
            self.iterate(n.x+1,n.y+1,n)
            self.iterate(n.x-1,n.y+1,n)
            self.iterate(n.x+1,n.y-1,n)
            self.iterate(n.x-1,n.y-1,n)
            self.iterate(n.x-1,n.y,n)
            self.iterate(n.x,n.y-1,n)
            self.iterate(n.x+1,n.y,n)
            self.iterate(n.x,n.y+1,n)

###  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 improved A* algorithm.

    Arguments:
    world_map -- A 120*120 array indicating map, where 0 indicating traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the start 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 ###

    a = A(world_map,start_pos,goal_pos)
    path = a.algorithm()


    ###  END CODE HERE  ###
    return path




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()
```

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:

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

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

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

```python
def bernstein_poly(self,i, n, t):
        return np.math.comb(n, i) * (1 - t)**(n - i) * t**i

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))

    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:

```python
import sys
import os
import numpy as np
import matplotlib.pyplot as plt

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.
import math
class N():
    def __init__(self,x,y,parent=None) -> None:
        self.x =x
        self.y =y
        self.parent = parent
        self.s = 0
        if [x,y] == start_pos:
            self.parent = None
            self.g = 0
        else:
            self.x_change = self.x - self.parent.x
            self.y_change = self.y - self.parent.y
            if  abs(self.x_change) + abs(self.y_change) == 1:
                self.g = self.parent.g + 1
            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:
                    self.s=1
        self.h = abs(self.x - goal_pos[0]) + abs(self.y - goal_pos[1])


        self.f = self.g + self.h + self.s

class A:
    def __init__(self,map,start,goal) -> None:
        self.map =map

        self.gen_obstacle_map()
        self.gen_obstacle_map()
        self.gen_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)

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


    def in_close(self,x,y):
        for i in self.closeset:
            if x == i.x and y==i.y:
                return True
        return False

    def in_open(self,x,y):
        for i in self.openset:
            if x == i.x and y==i.y:
                return True
```

```
            return False

    def iterate(self,x,y,parent):
        if (x < 0) or (x >= 120) or (y < 0) or (y >= 120):
            return
        if self.map[x][y] == 1:
            return
        if self.in_close(x,y):
            return
        if not self.in_open(x,y):
            n = N(x,y,parent)

            self.openset.append(n)

    def sel_min(self):
        min_cost = 1000000
        num =0
        for n in self.openset:
            if n.f <=min_cost:
                min_cost = n.f
                sel = num
            num +=1
        return sel

    def bernstein_poly(self,i, n, t):
        return np.math.comb(n, i) * (1 - t)**(n - i) * t**i

    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))

        return curve_points

    def res(self,n):
        path = []
        while True:
            n=n.parent
            if n:
                path.insert(0,[n.x,n.y])
            if (n.x == self.currentN.x) and (n.y == self.currentN.y):
                return self.bezier_curve(path)
                #return path

    def algorithm(self):
        while True:
            i = self.sel_min()
            n =self.openset[i]
            self.closeset.append(n)
            del self.openset[i]
            if(n.x == self.goal[0] and n.y == self.goal[1]):
                path = self.res(n)

                return path
            self.iterate(n.x+1,n.y+1,n)
            self.iterate(n.x-1,n.y+1,n)
            self.iterate(n.x+1,n.y-1,n)
            self.iterate(n.x-1,n.y-1,n)
            self.iterate(n.x-1,n.y,n)
            self.iterate(n.x,n.y-1,n)
            self.iterate(n.x+1,n.y,n)
            self.iterate(n.x,n.y+1,n)


###  END CODE HERE  ###


def self_driving_path_planner(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.

    Arguments:
    world_map -- A 120*120 array indicating map, where 0 indicating traversable and 1 indicating obstacles.
    start_pos -- A 2D vector indicating the start 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.
    """

    ### START CODE HERE ###

    a = A(world_map,start_pos,goal_pos)
    path = a.algorithm()


    ###  END CODE HERE  ###
    return path



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
    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()
```