Test 1

> 刘嘉奇 522030910009

```
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
import time
import math
import heapq

for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        MAP_PATH = os.path.join(dirname, filename)

class AStar:
    """AStar set the cost + heuristics as the priority
    """
    def __init__(self, s_start, s_goal, heuristic_type):
        self.s_start = tuple(s_start)
        self.s_goal = tuple(s_goal)
        self.heuristic_type = heuristic_type

        self.u_set = [(-1, 0), (-1, 1), (0, 1), (1, 1),
                        (1, 0), (1, -1), (0, -1), (-1, -1)]   # feasible input
set

        self.obs = set()  # position of obstacles

        self.OPEN = []  # priority queue / OPEN set
        self.CLOSED = []  # CLOSED set / VISITED order
        self.PARENT = dict()  # recorded parent
        self.g = dict()  # cost to come
        self.count = 0

    def searching(self):
        """
        A_star Searching.
        :return: path, visited order
        """

        self.PARENT[self.s_start] = self.s_start
        self.g[self.s_start] = 0
        self.g[self.s_goal] = math.inf
        heapq.heappush(self.OPEN,
                       (self.f_value(self.s_start), self.s_start))

        while self.OPEN:
            _, s = heapq.heappop(self.OPEN)
            if s == self.s_goal:  # stop condition
                break
            self.CLOSED.append(s)
```

```
            for s_n in self.get_neighbor(s):
                if s_n in self.PARENT:
                    new_cost = self.g[s] + self.cost(s, s_n, self.PARENT[s_n])
                else:
                    new_cost = self.g[s] + self.cost(s, s_n)

                if s_n not in self.g:
                    self.g[s_n] = math.inf

                if new_cost < self.g[s_n]:  # conditions for updating Cost
                    self.g[s_n] = new_cost
                    self.PARENT[s_n] = s
                    heapq.heappush(self.OPEN, (self.f_value(s_n), s_n))

        return self.extract_path(self.PARENT), self.CLOSED

    def cost(self, s_start, s_goal, s_last=None):
        """
        Calculate Cost for this motion, with added smoothness penalty.
        :param s_start: starting node (current node)
        :param s_goal: end node (neighbor node)
        :param s_last: the previous node (parent node) for turning angle calculation
        :return: Cost for this motion with turning angle penalty
        """
        if self.is_collision(s_start, s_goal):
            return math.inf

        # Basic length cost (distance cost)
        length = math.hypot(s_goal[0] - s_start[0], s_goal[1] - s_start[1])

        # Smoothness cost (turning angle penalty)
        angle_penalty = 0
        if s_last is not None:
            # Calculate the vectors for current direction and previous direction
            vec1 = (s_start[0] - s_last[0], s_start[1] - s_last[1])  # Direction from s_last to s_start
            vec2 = (s_goal[0] - s_start[0], s_goal[1] - s_start[1])  # Direction from s_start to s_goal

            # Calculate turning angle between vec1 and vec2
            norm1 = math.hypot(vec1[0], vec1[1])  # Magnitude of vec1
            norm2 = math.hypot(vec2[0], vec2[1])  # Magnitude of vec2

            if norm1 > 0 and norm2 > 0:  # Avoid zero division
                # Calculate cosine of the angle between vec1 and vec2
                cos_angle = (vec1[0] * vec2[0] + vec1[1] * vec2[1]) / (norm1 * norm2)
                cos_angle = max(-1.0, min(1.0, cos_angle))  # Clamp the value to avoid numerical issues

                # Calculate the actual angle in radians
                angle = math.acos(cos_angle)

                # Add a penalty for sharp turns
                angle_penalty = abs(angle)  # or use `angle ** 2` for a stronger penalty
```

```
        # Total cost: distance + smoothness penalty (adjust the weight as
needed)
        smooth_weight = 0.01  # Adjust this weight to control the influence of
turning angle
        total_cost = length + smooth_weight * angle_penalty

        self.count += 1
        if self.count % 1000 == 0 :
            print("length:",length)
            print("extra:",smooth_weight * angle_penalty)

        return total_cost

    def heuristic(self, s):
        """
        Calculate heuristic.
        :param s: current node (state)
        :return: heuristic function value
        """

        return math.hypot(self.s_goal[0] - s[0], self.s_goal[1] - s[1])

    def f_value(self, s):
        """
        f = g + h. (g: Cost to come, h: heuristic value)
        :param s: current state
        :return: f
        """
        return self.g[s] + self.heuristic(s)



    def get_neighbor(self, s):
        """
        find neighbors of state s that not in obstacles.
        :param s: state
        :return: neighbors
        """

        return [(s[0] + u[0], s[1] + u[1]) for u in self.u_set]

    def is_collision(self, s_start, s_end):
        """
        check if the line segment (s_start, s_end) is collision.
        :param s_start: start node
        :param s_end: end node
        :return: True: is collision / False: not collision
        """

        if s_start in self.obs or s_end in self.obs:
            return True

        if s_start[0] != s_end[0] and s_start[1] != s_end[1]:
            if s_end[0] - s_start[0] == s_start[1] - s_end[1]:
                s1 = (min(s_start[0], s_end[0]), min(s_start[1], s_end[1]))
                s2 = (max(s_start[0], s_end[0]), max(s_start[1], s_end[1]))
            else:
                s1 = (min(s_start[0], s_end[0]), max(s_start[1], s_end[1]))
```

```python
                s2 = (max(s_start[0], s_end[0]), min(s_start[1], s_end[1]))

            if s1 in self.obs or s2 in self.obs:
                return True

        return False

    def extract_path(self, PARENT):
        """
        Extract the path based on the PARENT set.
        :return: The planning path
        """

        path = [self.s_goal]
        s = self.s_goal

        while True:
            s = PARENT[s]
            path.append(s)

            if s == self.s_start:
                break

        return list(path)

def get_obstacles(map):
    obstacles = set()
    for i in range(120):
        for j in range(120):
            if map[i][j] == 1:
                obstacles.add((i,j))
    return obstacles

def A_star(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:
    map -- A 120*120 array indicating current map, where 0 indicating traversable
and 1 indicating obstacles.
    start_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.
    """
    astar = AStar(start_pos, goal_pos, "euclidean")
    astar.obs = get_obstacles(map)
    path, _ = astar.searching()
    path.reverse()

    return path

if __name__ == '__main__':
```

```python
    # 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()
```