```
# Description: This is a Python script that implements the A* algorithm for path planning in a 2D grid map.

# Author: Hoang Pham & Fazil Mammadli

# Last modified: 2024-03-25

# Python version: 3.8

# Usage: python a_star_hoang_pham.py

# Notes: This script requires the OpenCV library to be installed.

#       The script will prompt you to enter the start and goal node coordinates, as well as the step size for the robot.

#       The script will display the map, explored nodes, and the optimal path using OpenCV.

#       The script will also print the execution time, number of nodes explored, and path length after finding the optimal path.



from queue import PriorityQueue

import numpy as np

import math

import cv2

import time


################################################################################################


# Step 1: Define the Node class and actions


################################################################################################

class Node:

  def __init__(self, state, parent=None, cost_to_come=0, cost_to_go=0, cost=float('inf')):

    self.state = state
```

```
        self.parent = parent
        self.cost_to_come = cost_to_come
        self.cost_to_go = cost_to_go
        self.cost = cost
        self.x = int(round(state[0]))  # Store the rounded integer value of x
        self.y = int(round(state[1]))  # Store the rounded integer value of y
        self.theta = state[2] % 360    # Store the theta value

    def __lt__(self, other):
        return self.cost < other.cost

    def __eq__(self, other):
        if isinstance(other, Node):
            return self.state == other.state
        return False

    def __hash__(self):
        return hash(self.state)

def normalize_theta(theta):
    theta = theta % 360  # Ensure theta is within 0 to 360 degrees
    theta = int(round(theta / 30)) * 30  # Round theta to the nearest multiple of 30 degrees
    return theta

# Define the action functions
def move_straight(node, step_size):
    x, y, theta = node.state
    new_x = max(0, min(x + step_size * math.cos(math.radians(theta)), 1200 - 1))
    new_y = max(0, min(y + step_size * math.sin(math.radians(theta)), 500 - 1))
```

```
    new_theta = normalize_theta(theta)
    return (new_x, new_y, new_theta, step_size)

def move_30_left(node, step_size):
    x, y, theta = node.state
    new_x = max(0, min(x + step_size * math.cos(math.radians(theta + 30)), 1200 - 1))
    new_y = max(0, min(y + step_size * math.sin(math.radians(theta + 30)), 500 - 1))
    new_theta = normalize_theta(theta + 30)
    return (new_x, new_y, new_theta, step_size + 1)

def move_60_left(node, step_size):
    x, y, theta = node.state
    new_x = max(0, min(x + step_size * math.cos(math.radians(theta + 60)), 1200 - 1))
    new_y = max(0, min(y + step_size * math.sin(math.radians(theta + 60)), 500 - 1))
    new_theta = normalize_theta(theta + 60)
    return (new_x, new_y, new_theta, step_size + 2)

def move_30_right(node, step_size):
    x, y, theta = node.state
    new_x = max(0, min(x + step_size * math.cos(math.radians(theta - 30)), 1200 - 1))
    new_y = max(0, min(y + step_size * math.sin(math.radians(theta - 30)), 500 - 1))
    new_theta = normalize_theta(theta - 30)
    return (new_x, new_y, new_theta, step_size + 1)

def move_60_right(node, step_size):
    x, y, theta = node.state
    new_x = max(0, min(x + step_size * math.cos(math.radians(theta - 60)), 1200 - 1))
    new_y = max(0, min(y + step_size * math.sin(math.radians(theta - 60)), 500 - 1))
    new_theta = normalize_theta(theta - 60)
```

```
    return (new_x, new_y, new_theta, step_size + 2)



################################################################################
##############

# Step 2: Define the configuration space with obstacles using mathematical equations


################################################################################
##############
def create_map(height=500, width=1200, border_thickness=5):
    # Pre-compute obstacle positions
    obstacle_positions = set()
    border_positions = set()

    # Add border obstacle
    for y in range(height):
        for x in range(width):
            if x < border_thickness or x >= width - border_thickness or y < border_thickness or y >= height -
border_thickness:
                border_positions.add((x, y))

    # Rectangle obstacle 1
    for y in range(100 - 5, 500):
        for x in range(100 - 5, 175 + 5):
            if 100 <= x < 175 and 100 <= y < 500:
                obstacle_positions.add((x, y))
            else:
                border_positions.add((x, y))
```

```
# Rectangle obstacle 2
for y in range(0, 400 + 5):
    for x in range(275 - 5, 350 + 5):
        if 275 <= x < 350 and 0 <= y < 400:
            obstacle_positions.add((x, y))
        else:
            border_positions.add((x, y))

# Polygonal obstacle with 6 sides (hexagon)
for y in range(100 - 5, 400 + 6):
    for x in range(510 - 5, 790 + 6):
        if y - 0.615384615385 * x <= 5 and \
            y + 0.576923076923 * x - 775 <= 5 and \
            x <= 785 and \
            y - 0.576923076923 * x + 275 >= -5 and \
            y + 0.576923076923 * x - 475 >= -5 and \
            x >= 510:
                obstacle_positions.add((x, y))

# Obstacle made from rectangles on the far right
for y in range(50-5, 125 + 5):
    for x in range(900 - 5, 1100 + 5):
        if 900 <= x < 1100 and 50 <= y < 125:
            obstacle_positions.add((x, y))
        else:
            border_positions.add((x, y))
for y in range(375 - 5, 450 + 5):
    for x in range(900 - 5, 1100 + 5):
        if 900 <= x < 1100 and 375 <= y < 450:
```

```
            obstacle_positions.add((x, y))
        else:
            border_positions.add((x, y))
for y in range(50 - 5, 450 + 5):
    for x in range(1020 - 5, 1100 + 5):
        if 1020 <= x < 1100 and 125 <= y < 450:
            obstacle_positions.add((x, y))
        else:
            border_positions.add((x, y))

# Create an empty canvas
canvas = np.zeros((height, width), dtype=np.uint8)

# Create a boolean matrix to represent obstacle positions
obstacle_matrix = np.zeros((height, width), dtype=bool)

# Draw obstacles on the canvas and update the obstacle matrix
for y in range(height):
    for x in range(width):
        if (x, height - 1 - y) in obstacle_positions:
            canvas[y, x] = 255  # White represents obstacles
            obstacle_matrix[y, x] = True  # Update the obstacle matrix
        elif (x, height - 1 - y) in border_positions:
            canvas[y, x] = 128  # Gray represents borders
            obstacle_matrix[y, x] = True  # Update the obstacle matrix

# Obstacles made from borders
for coord in border_positions:
    obstacle_positions.add(coord)
```

```
    return canvas, height, obstacle_matrix

################################################################################
##############

# Step 3: Implement the A* algorithm to generate the graph and find the optimal path

################################################################################
##############
# Define the heuristic function
def euclidean_distance(state1, state2):
    x1, y1, _ = state1
    x2, y2, _ = state2
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

# Define the Diagonal distance heuristic function
def diagonal_distance(state1, state2):
    dx = abs(state2[0] - state1[0])
    dy = abs(state2[1] - state1[1])
    return (dx + dy) + (math.sqrt(2) - 2) * min(dx, dy)

# Define the goal node check function
def is_goal_node(current_node, goal_node, threshold=1.5):
    return euclidean_distance(current_node.state, goal_node.state) <= threshold

# Define the obstacle check function
def is_obstacle(node, obstacle_matrix, height):
    y, x = int(round(height - 1 - node.y)), int(round(node.x))
```

```
    if 0 <= y < obstacle_matrix.shape[0] and 0 <= x < obstacle_matrix.shape[1]:
        return obstacle_matrix[y, x] # Return True if the node is an obstacle
    else:
        return True # Return True if the node is out of bounds

def is_duplicate_node(cost_grid, node):
    x, y, theta = node.state
    theta = normalize_theta(theta)
    i = int(round(y / 2))
    j = int(round(x / 2))
    k = int(round(theta / 30))

    return cost_grid[i][j][k] != float('inf')

def update_node_cost(cost_grid, node):
    x, y, theta = node.state
    theta = normalize_theta(theta)
    i = int(round(y / 2))
    j = int(round(x / 2))
    k = int(round(theta / 30))

    cost_grid[i][j][k] = node.cost

def a_star(start_node, goal_node, step_size, is_obstacle):
    open_list = PriorityQueue()
    closed_list = set()

    cost_grid = np.full((500, 1200, 12), float('inf'))  # Cost grid to store the cost of reaching each node
```

```
start_node.cost_to_come = 0
start_node.cost_to_go = diagonal_distance(start_node.state, goal_node.state)
start_node.cost = start_node.cost_to_come + start_node.cost_to_go
open_list.put(start_node)
update_node_cost(cost_grid, start_node)

while not open_list.empty():
  current_node = open_list.get()

  if is_goal_node(current_node, goal_node):
    return "Success", current_node, closed_list

  closed_list.add(current_node)

  for action in actions:
    new_node = Node((0, 0, 0))
    x, y, theta, action_cost = action(current_node, step_size)
    new_node.state = (x, y, theta)
    new_node.x = int(round(x))
    new_node.y = int(round(y))
    new_node.parent = current_node
    new_node.cost_to_come = current_node.cost_to_come + action_cost
    new_node.cost_to_go = diagonal_distance(new_node.state, goal_node.state)
    new_node.cost = new_node.cost_to_come + new_node.cost_to_go

    if not is_obstacle(new_node) and not is_duplicate_node(cost_grid, new_node):
      open_list.put(new_node)
      update_node_cost(cost_grid, new_node)
```

```
    return "Failure", None, closed_list

################################################################################
##############

# Step 4: Implement the backtracking function to find the optimal path

################################################################################
##############
# Define the backtracking function
def backtrack(goal_node):
    path = []
    current_node = goal_node
    while current_node is not None:
        path.append(current_node)
        current_node = current_node.parent
    path.reverse()
    return path

################################################################################
##############

# Step 5: Implement the visualization function to display the map, explored nodes, and the optimal
path

################################################################################
##############
def visualize_start_and_goal_nodes(canvas, start_node, goal_node):
    # Drawing the start node as a green circle
    cv2.circle(canvas, (start_node.x, 500 - start_node.y), radius=5, color=(0, 255, 0),
thickness=cv2.FILLED)
```

```
    # Drawing the goal node as a red circle
    cv2.circle(canvas, (goal_node.x, 500 - goal_node.y), radius=5, color=(0, 0, 255), thickness=cv2.FILLED)

def visualize_obstacles(canvas, obstacle_matrix):
    # Draw obstacles on the canvas
    for y in range(obstacle_matrix.shape[0]):
        for x in range(obstacle_matrix.shape[1]):
            if obstacle_matrix[y][x]:
                canvas[y, x] = (255, 255, 255)  # White color for obstacles

def visualize_explored_nodes(canvas, explored_nodes, step_size):
    for i, node in enumerate(explored_nodes):
        if i % 500 == 0:
            x = node.x
            y = node.y
            for action in actions:
                new_x, new_y, new_theta, _ = action(node, step_size)
                new_x = int(new_x)
                new_y = int(new_y)
                cv2.arrowedLine(canvas, (int(x), int(500 - y)), (new_x, int(500 - new_y)), (255, 0, 255), thickness=1, tipLength=0.2)
            cv2.imshow("Path Planning", canvas)
            cv2.waitKey(1)

def visualize_optimal_path(canvas, optimal_path, step_size):
    print(f"Visualizing path with {len(optimal_path)} steps.")

    for idx in range(len(optimal_path) - 1):
```

```
        current_node = optimal_path[idx]
        next_node = optimal_path[idx + 1]
        x1, y1 = current_node.x, current_node.y
        x2, y2 = next_node.x, next_node.y
        cv2.arrowedLine(canvas, (x1, 500 - y1), (x2, 500 - y2), (0, 255, 255), thickness=1, tipLength=0.5)
        cv2.imshow("Path Planning", canvas)
        cv2.waitKey(10)

    cv2.waitKey(0)
    cv2.destroyAllWindows()


def visualize_path(canvas, path_nodes, closed_list, obstacle_matrix, start_node, goal_node,
step_size):
    # Create a canvas copy with three channels for RGB
    canvas_copy = np.zeros((canvas.shape[0], canvas.shape[1], 3), dtype=np.uint8)

    # Draw free space, obstacles, and borders onto the canvas
    for y in range(canvas.shape[0]):
        for x in range(canvas.shape[1]):
            if canvas[y, x] == 0:
                canvas_copy[y, x] = (0, 0, 0)  # Black for free space
            elif canvas[y, x] == 255:
                canvas_copy[y, x] = (255, 255, 255)  # White for obstacles
            elif canvas[y, x] == 128:
                canvas_copy[y, x] = (128, 128, 128)  # Gray for borders

    # Visualize start and goal nodes
    visualize_start_and_goal_nodes(canvas_copy, start_node, goal_node)
```

```
    # Visualize explored nodes
    visualize_explored_nodes(canvas_copy, closed_list, step_size)

    # Visualize the optimal path
    visualize_optimal_path(canvas_copy, path_nodes, step_size)

    cv2.waitKey(0)
    cv2.destroyAllWindows()

# Function to get start and goal nodes from user input
def get_start_and_goal_nodes(obstacle_matrix, height):
    while True:
        start_x, start_y, start_theta = map(int, input("Enter the start node coordinates (x y theta): ").split())
        if not is_obstacle(Node((start_x, start_y, start_theta)), obstacle_matrix, height) and start_theta % 30 == 0:
            break
        else:
            print("Start node coordinates are within an obstacle or theta is not a multiple of 30 degrees. Please choose different coordinates.")

    while True:
        goal_x, goal_y, goal_theta = map(int, input("Enter the goal node coordinates (x y theta): ").split())
        if not is_obstacle(Node((goal_x, goal_y, goal_theta)), obstacle_matrix, height) and goal_theta % 30 == 0:
            break
        else:
            print("Goal node coordinates are within an obstacle or theta is not a multiple of 30 degrees. Please choose different coordinates.")
```

```
    start_node = Node((start_x, start_y, start_theta))
    goal_node = Node((goal_x, goal_y, goal_theta))

    return start_node, goal_node

# Function to get the step size from user input
def get_step_size():
    while True:
        step_size = int(input("Enter the step size for the robot (from 1 to 10): "))
        if step_size >= 1 and step_size <= 10:
            return step_size
        else:
            print("Step size must be within 1 and 10.")

def evaluate_performance(start_time, end_time, closed_list, path_nodes):
    execution_time = end_time - start_time
    nodes_explored = len(closed_list)
    path_length = len(path_nodes)

    print("Performance Evaluation:")
    print(f"Execution Time: {execution_time:.4f} seconds")
    print(f"Nodes Explored: {nodes_explored}")
    print(f"Path Length: {path_length}")

################################################################################
##############

# Step 6: Main function to run the path planning algorithm
```

```
################################################################################
##############

if __name__ == "__main__":

    # Define the action set
    actions = [
        move_straight,
        move_30_left,
        move_60_left,
        move_30_right,
        move_60_right
    ]

    # Create the map
    canvas, height, obstacle_matrix  = create_map()

    ############################################ (Uncomment to preview the map using
OpenCV)
    # cv2.imshow("Map", canvas)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()
    ############################################

    # Get user input for start and goal nodes
    start_node, goal_node = get_start_and_goal_nodes(obstacle_matrix, height)

    # Get user input for the step size
    step_size = get_step_size()
```

```
    # Run the A* algorithm
    start_time = time.time()
    result, goal_node, closed_list = a_star(start_node, goal_node, step_size, lambda node:
is_obstacle(node, obstacle_matrix, height))
    end_time = time.time()

    if result == "Success":
        print("Path found!")
        path_nodes = backtrack(goal_node)
        evaluate_performance(start_time, end_time, closed_list, path_nodes)
        visualize_path(canvas, path_nodes, closed_list, obstacle_matrix, start_node, goal_node,
step_size)

    else:
        print("No path found.")
```