import numpy as np

import matplotlib.pyplot as plt

from queue import PriorityQueue

# FIRST PART

# Define the dimensions of the grid, 12 rows and 12 columns

rows = 12

cols = 12

# Create a grid with all cells initially empty

grid = np.zeros((rows, cols))

# Define states as specific grid cells that the robot can occupy

grid[11, 1] = 2

grid[1, 2:10] = 1 # Define obstacles

grid[2, 8:10] = 1 # Define obstacles

grid[0, 2:10] = 1 # Define obstacles

grid[1, 3:10] = 1 # Define obstacles

grid[2, 2:4] = 1 # Define obstacles

grid[11, 4:8] = 1 # Define obstacles

grid[10, 4:8] = 1 # Define obstacles

grid[4, 0:3] = 1 # Define obstacles

grid[5:8, 0:2] = 1 # Define obstacles

grid[7, 2] = 1 # Define obstacles

grid[4:8, 9:12] = 1 # Define obstacles

grid[5:7, 5:7] = 1 # Define obstacles

grid[11, 0] = 3 # Define charging station

grid[11, 2] = 4 # Define charging station

# Plot the grid-based map

plt.figure(figsize=(10, 4))

plt.subplot2grid((3, 4), (0, 0), colspan=2, rowspan=3)

plt.imshow(grid, cmap='hot', origin='upper', extent=[0, cols, 0, rows])

plt.title('Grid-based Map')

plt.xlabel('Columns')

plt.ylabel('Rows')

plt.grid(True, which='both', color='white', linestyle='-', linewidth=1)

plt.xticks(np.arange(0, cols+1, 1))

plt.yticks(np.arange(0, rows+1, 1))

plt.gca().invert\_yaxis()

# Plot legend for states

plt.subplot2grid((3, 4), (0, 2), colspan=1, rowspan=3)

plt.axis('off')

legend\_elements\_states = [

plt.Line2D([0], [0], marker='s', color='w', markerfacecolor='black', markersize=10, label='Dirty'),

plt.Line2D([0], [0], marker='s', color='w', markerfacecolor='red', markersize=10, label='Obstacle'),

plt.Line2D([0], [0], marker='s', color='w', markerfacecolor='white', markersize=10, label='Cleaned'),

plt.Line2D([0], [0], marker='s', color='w', markerfacecolor='yellow', markersize=10, label='Charging station'),

plt.Line2D([0], [0], marker='s', color='w', markerfacecolor='orange', markersize=10, label='Visited'),

]

plt.legend(handles=legend\_elements\_states, loc='center', title='States')

# Plot legend for actions

plt.subplot2grid((3, 4), (2, 3), colspan=1, rowspan=1)

plt.axis('off')

legend\_elements\_actions = [

plt.Line2D([0], [0], marker='', color='w', markerfacecolor='black', markersize=10, label='Move forward'),

plt.Line2D([0], [0], marker='', color='w', markerfacecolor='red', markersize=10, label='Turn left'),

plt.Line2D([0], [0], marker='', color='w', markerfacecolor='white', markersize=10, label='Turn right'),

plt.Line2D([0], [0], marker='', color='w', markerfacecolor='white', markersize=10, label='Move backward'),

plt.Line2D([0], [0], color='w', markerfacecolor='white', markersize=10, label='Clean')

]

plt.legend(handles=legend\_elements\_actions, loc='center', title='Robot Actions')

plt.show()

# SECOND PART

# Define a cost function that assigns a cost value to each action the robot takes.

def cost\_function(action): # Name of the function is cost\_function

if action == 'Move forward': # If the action is move forward then the cost is 3

return 3

elif action =='Turn left': # If the action is turn left then the cost is 5

return 5

elif action == 'Turn right': # If the action is turn right then the cost is 5

return 5

elif action == 'Move backward': # If the action is move backward then the cost is 7

return 7

elif action == 'Clean': # If the action is clean then the cost is 10

return 10

else:

return 0 # If there is no action then the cost is 0

print("COST FOR SPECIFIC ROBOT ACTION:")

action = 'Move forward'

cost = cost\_function(action)

print("Cost for action", action, ":", cost)

action = 'Turn left'

cost = cost\_function(action)

print("Cost for action", action, ":", cost)

action = 'Turn right'

cost = cost\_function(action)

print("Cost for action", action, ":", cost)

action = 'Move backward'

cost = cost\_function(action)

print("Cost for action", action, ":", cost)

action = 'Clean'

cost = cost\_function(action)

print("Cost for action", action, ":", cost)

# Define a heuristic function that calculates the estimated remaining distance

# So we implemented a function called heuristic, the function takes in two paremeters (current\_position, target\_position)

def heuristic(current\_position, target\_position):

return abs(current\_position[0] - target\_position[0]) + abs(current\_position[1] - target\_position[1])

# Thie above line calculates the manhattan distance between two positions represented as '(x, y)' coordinates

# current\_position[0] represents the x-coordinate of the current position.

# current\_position[1] represents the y-coordinate of the current position.

# target\_position[0] represents the x-coordinate of the target position.

# target\_position[1] represents the y-coordinate of the target position.

# So, in simpler terms, the code calculates how many steps you need to move horizontally and vertically to get from

# current\_position to target\_position

def get\_successors(position, grid): # Here we creates a function called get\_successors, which takes in two parameters (position and grid)

# where position is the current position of the robot and grid is the 2D grind-based map

x, y = position

successors = [] #This function initialises an empty list called successors to store the successor positions

# Define possible movements: up, down, left, right

movements = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# This line It defines the possible movements that can be made from the current

# position. These movements include going up, down, left, or right. Each movement

# is represented as a tuple (dx, dy), where dx represents the change in the row

# index (x) and dy represents the change in the column index (y).

for dx, dy in movements:

new\_x, new\_y = x + dx, y + dy

# This code under checks if the new position is within the grid boundaries and not an obstacle

# For each movement, it calculates the new position (new\_x, new\_y) by adding the corresponding

# changes to the current position (x, y). It then checks if the new position is within the boundaries

# of the grid and if it's not an obstacle (i.e., if the value in the grid at the new position is not 1).

if 0 <= new\_x < len(grid) and 0 <= new\_y < len(grid[0]) and grid[new\_x, new\_y] != 1:

successors.append((new\_x, new\_y))

return successors # If both conditions are met, the new position is considered a valid successor, and it is appended to the 'successors' list

# THIRD PART

def astar\_search(grid, start, goal, charging\_station): # For this are we implemented a function called astar\_search with 4 parameters

# grid, start, goal and charging position

open\_list = PriorityQueue()

open\_list.put((0, start))

came\_from = {}

g\_score = {node: float("inf") for row in grid for node in row}

g\_score[start] = 0

f\_score = {node: float("inf") for row in grid for node in row}

f\_score[start] = heuristic(start, goal)

while not open\_list.empty():

current = open\_list.get()[1]

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(charging\_station)

return path[::-1]

for next\_pos in get\_successors(current, grid):

tentative\_g\_score = g\_score[current] + cost\_function('Clean')

if tentative\_g\_score < g\_score.get(next\_pos, float("inf")):

came\_from[next\_pos] = current

g\_score[next\_pos] = tentative\_g\_score

f\_score[next\_pos] = tentative\_g\_score + heuristic(next\_pos, goal)

open\_list.put((f\_score[next\_pos], next\_pos))

return None

# Call the A\* search function to find the optimal path

start = (4, 6) # Starting position of the robot

goal = (1, 11) # Goal position to clean every cell

charging\_station = (11, 0) # Charging station position

optimal\_path = astar\_search(grid, start, goal, charging\_station)

print("Optimal Path:", optimal\_path)