



**EAST WEST UNIVERSITY**

**Department of Computer Science and Engineering  
A/2, Jahurul Islam Avenue, Jahurul Islam City, Aftabnagar,  
Dhaka-1212**

**Spring 2024 Semester**

**Report on  
“Assignment-1: Enhanced Dynamic  
Robot Movement Simulation”**

**Course Code : CSE366  
Course Title : Artificial Intelligence  
Section : 03**

**Submitted To:**

**Instructor:**

**Dr. Mohammad Rifat Ahmmad Rashid  
Assistant Professor,  
Department of Computer Science & Engineering**

**Submitted By:**

**Name: Md Maruf Hasan  
ID : 2021-3-60-101**

# Assignment -1 Title: Enhanced Dynamic Robot Movement Simulation

**Objective:** Designing and implementation of an advanced simulation environment for a robot navigating through a dynamically created grid. This project aims to deepen understanding of basic programming concepts, object-oriented programming (OOP), algorithms for navigation and pathfinding, task optimization, safety, and energy management strategies.

**Overview:** I have developed a simulation for a robot moving through a grid-based environment considering task optimization strategies and safety to ensure efficient travel, collision avoidance, and effective energy management. The simulation should manage the robot's energy levels and battery status, incorporating the necessity of recharging.

## Detailed Requirements:

### 1. Environment Setup

- Implement a class Environment that generates a 10x10 grid. Dynamically place obstacles, a start position, and an end position within the grid.

#### CODE:

```
# Environment Class represents the grid and handles state transitions.
class Environment:
    def __init__(self, grid, start, goal):
        self.grid = grid # The grid layout where 1 represents an obstacle and 0 is free
        space.
        self.initial = start # Starting position of the agent.
        self.goal = goal # Goal position the agent aims to reach.

    # Returns the possible actions from a given state.
    def actions(self, state):
        possible_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        x, y = state

        # Remove impossible actions based on grid boundaries and obstacles.
        if x == 0 or self.grid[x - 1][y] == 1:
            possible_actions.remove('UP')
```

```

if x == len(self.grid) - 1 or self.grid[x + 1][y] == 1:
    possible_actions.remove('DOWN')
if y == 0 or self.grid[x][y - 1] == 1:
    possible_actions.remove('LEFT')
if y == len(self.grid[0]) - 1 or self.grid[x][y + 1] == 1:
    possible_actions.remove('RIGHT')

return possible_actions

# Returns the state resulting from taking a given action at a given state.
def result(self, state, action):
    x, y = state
    if action == 'UP':
        return (x - 1, y)
    if action == 'DOWN':
        return (x + 1, y)
    if action == 'LEFT':
        return (x, y - 1)
    if action == 'RIGHT':
        return (x, y + 1)

# Checks if the goal has been reached.
def is_goal(self, state):
    return state == self.goal

```

## 2. Robot

- Implement a class Agent with movement capabilities and tracking of its current position. Include methods to manage the robot's energy levels and battery status, incorporating task optimization and safety for efficient and safe navigation.

- **Battery Management:** The robot starts with a battery level of 100%. For each move from one block to another, the battery level decreases by 10%. If the battery level reaches 0%, the robot must recharge to 100% before continuing.

## 3. Simulation

- Simulate the robot's movement through the 10x10 grid with randomly placed obstacles, accounting for energy consumption and managing energy levels to complete tasks.

## 4. Pathfinding Algorithms

- Students are required to implement two pathfinding algorithms: Uniform Cost

Search (UCS) and A\* (A Star). Evaluate these algorithms based on the number of times the robot needs to recharge its battery while traversing the path to the goal. This will determine the best algorithm for this environment.

## CODE:

```
class Agent:
    def __init__(self, env):
        self.env = env # The environment in which the agent operates.
        self.battery = 100 # int((np.random.randint(100))/10)*10 Initial battery level

    #battery section
    def recharge(self):
        self.battery = 100

    def current_charge(self):
        return self.battery

    # Performs UCS search to find a path from the initial state to the goal.
    def uniform_cost_search(self):
        frontier = PriorityQueue() # Priority queue for UCS.
        frontier.put(Node(self.env.initial, path_cost=0), 0)
        came_from = {self.env.initial: None}
        cost_so_far = {self.env.initial: 0}

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

            if self.env.is_goal(current_node.state):
                return self.reconstruct_path(came_from, current_node.state)

            for action in self.env.actions(current_node.state):
                new_state = self.env.result(current_node.state, action)
                new_cost = cost_so_far[current_node.state] + 1 # Assuming uniform cost for simplicity; adjust if varying costs.
                if new_state not in cost_so_far or new_cost < cost_so_far[new_state]:
                    cost_so_far[new_state] = new_cost
                    priority = new_cost
                    frontier.put(Node(new_state, current_node, action, new_cost), priority)
                    came_from[new_state] = current_node.state

        return []

    # This is a_star_search
```

```

def a_star_search(self):
    #self.recharge()
    # The start node is created with a path cost of 0.
    start_node = Node(self.env.initial, path_cost=0)
    frontier = PriorityQueue()
    frontier.put(start_node, 0) # Priority is f-cost, initially the heuristic cost from
start to goal
    came_from = {self.env.initial: None} # Tracks the best path to a node
    cost_so_far = {self.env.initial: 0} # Tracks the g-cost (cost so far to reach a
node)

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

        if self.env.is_goal(current_node.state):
            return self.reconstruct_path(came_from, current_node.state)

        for action in self.env.actions(current_node.state):
            new_state = self.env.result(current_node.state, action)
            new_cost = cost_so_far[current_node.state] + 1 # Assuming uniform cost
for simplicity
            if new_state not in cost_so_far or new_cost < cost_so_far[new_state]:
                cost_so_far[new_state] = new_cost
                priority = new_cost + heuristic(new_state, self.env.goal) # f-cost = g-
cost + h-cost
                frontier.put(Node(new_state, current_node, action, new_cost), priority)
                came_from[new_state] = current_node.state

    return []

# Performs Uniform Cost Search to find the lowest cost path from the initial
state to the goal.
def uniform_cost_search(self):
    #self.recharge()
    frontier = PriorityQueue() # Priority queue for UCS.
    frontier.put(Node(self.env.initial, path_cost=0), 0)
    came_from = {self.env.initial: None}
    cost_so_far = {self.env.initial: 0}

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

        if self.env.is_goal(current_node.state):
            return self.reconstruct_path(came_from, current_node.state)

```

```

        for action in self.env.actions(current_node.state):
            new_state = self.env.result(current_node.state, action)
            new_cost = cost_so_far[current_node.state] + 1 # Assuming uniform cost
for simplicity; adjust if varying costs.
            if new_state not in cost_so_far or new_cost < cost_so_far[new_state]:
                cost_so_far[new_state] = new_cost
                priority = new_cost
                frontier.put(Node(new_state, current_node, action, new_cost), priority)
                came_from[new_state] = current_node.state

    return []

# Reconstructs the path from start to goal by following the came_from links.
def reconstruct_path(self, came_from, current):
    path = []
    while current in came_from:
        path.append(current)
        current = came_from[current]

    path.append(self.env.initial) # Start is not included in the came_from map.
    path.reverse() # Reverse to get the path from start to goal.
    return path

# Battery Charging point and management
def battery_manager(self, path):
    self.recharge()
    charge = []

    for i in range(len(path)):
        if self.current_charge() <= 10:
            charge.append(path[i])
            self.recharge()
            pass
        elif path[i] != (0, 0):
            self.battery = self.current_charge() - 10
            pass
        print(f"{path[i]} = {self.current_charge()}")

    return charge

```

```

class PriorityQueue:
    def __init__(self):
        self.elements = []

```

```

def empty(self):
    return len(self.elements) == 0

def put(self, item, priority):
    heapq.heappush(self.elements, (priority, item))

def get(self):
    return heapq.heappop(self.elements)[1]
def heuristic(a, b):
    """
    Calculate the Manhattan distance between two points a and b.

    Parameters:
    - a: Tuple representing the x and y coordinates of point a (e.g., (x1, y1))
    - b: Tuple representing the x and y coordinates of point b (e.g., (x2, y2))

    Returns:
    - The Manhattan distance between points a and b.
    """
    (x1, y1) = a
    (x2, y2) = b
    return abs(x1 - x2) + abs(y1 - y2)

```

## 5. Visualization

- Use libraries like matplotlib to visualize the grid, obstacles, paths, and the robot's energy levels over time.

## 6. Task Optimization and Safety

- Implement strategies to minimize travel time, energy consumption, and ensure robots can detect and avoid potential collisions, maintaining safety.

### CODE:

```

# Visualization Function plots the grid and the found path.
def visualize_grid_and_path(grid, path, charge):
    grid_array = np.array(grid) # Convert grid to numpy array for easy plotting.
    fig, ax = plt.subplots()
    ax.imshow(grid_array, cmap='flag', alpha=0.3) # Grid background.
    start = path[0]
    goal = path[-1]
    ax.plot(goal[1], goal[0], 'bs', markersize=10) # goal position in blue.

```

```

ax.plot(start[1], start[0], 'ks', markersize=10) # start position in black.
if (charge is not None) :
    xc, yc = zip(*charge)
    ax.plot(yc, xc, 'gs', markersize=10)# charging position are in green.

xs, ys = zip(*path) # Extract X and Y coordinates of the path.
ax.plot(ys, xs, 'r-', linewidth=2) # Plot the path in red.
ax.set_xticks(np.arange(-.5, len(grid[0]), 1), minor=True)
ax.set_yticks(np.arange(-.5, len(grid), 1), minor=True)

ax.grid(which="minor", color="b", linestyle='-', linewidth=1)
ax.tick_params(which="minor", size=0)
ax.tick_params(which="major", bottom=False, left=False,
labelbottom=False, labelleft=False)
plt.show()

```

```

def generate_random_grid(size, obstacle_probability):
    return np.random.choice([0, 1], size=(size, size), p=[1-obstacle_probability,
obstacle_probability])

# Define the size of the grid and the probability of an obstacle in each cell
grid_size = 10
obstacle_probability = 0.28 # 28% chance of being an obstacle

# Generate a random grid
grid = generate_random_grid(grid_size, obstacle_probability)

# Define start and goal positions
start = (0, 0)
goal = (grid_size - 1, grid_size - 1)

# Ensure start and goal are not obstacles
grid[start] = 0
grid[goal] = 0

# Create the environment and agent
environment = Environment(grid, start, goal)
agent = Agent(environment)

```

## Using UCS Algorithm:

```

# Solve the problem with the UCS algorithm
print("-----Uniform Cost Search algorithm Solution Path-----")
print("-----Discharging and Recharging States with Charge Level-----")
solution_path_ucs = agent.uniform_cost_search()

```

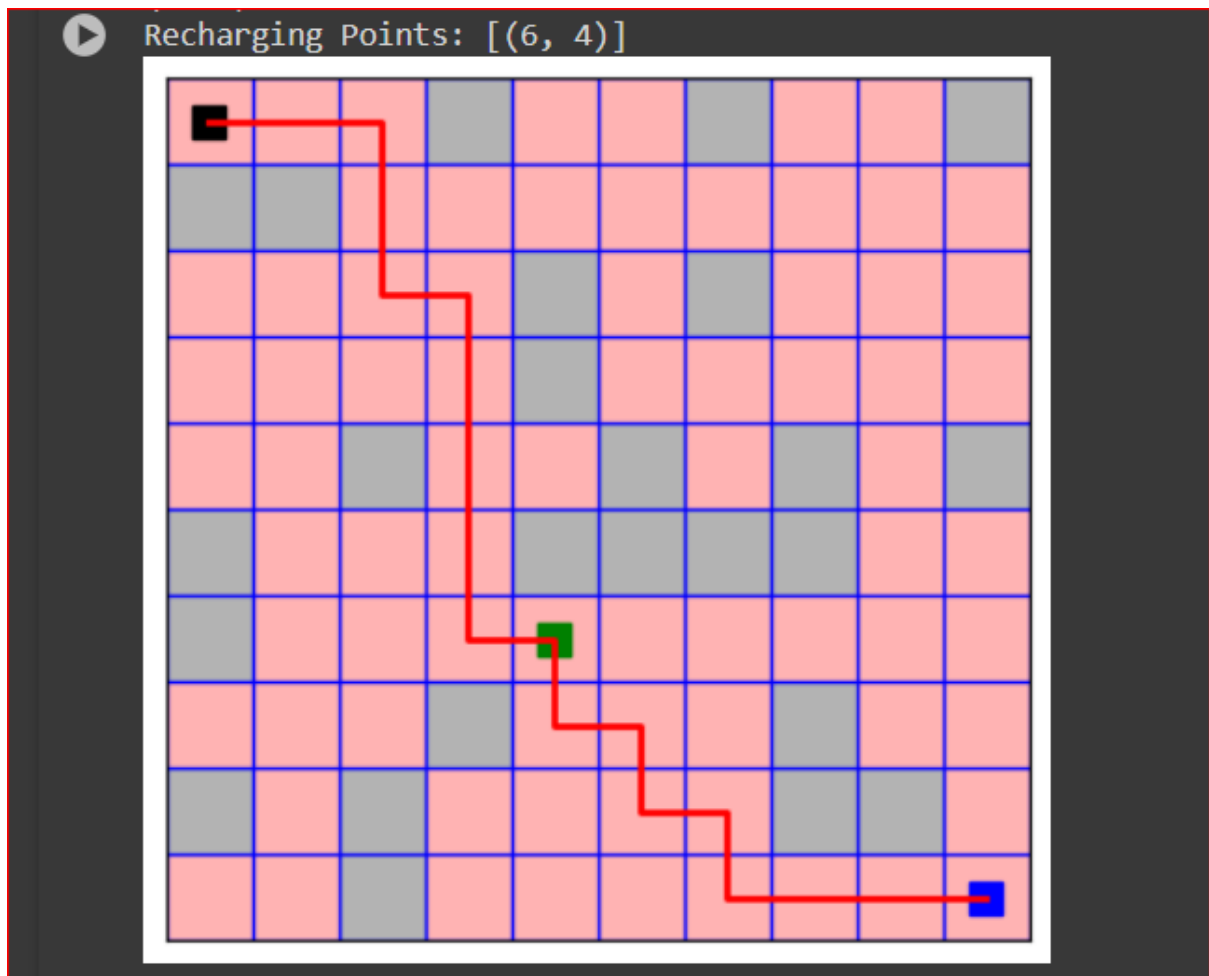


```
charging_point = agent.battery_manager(solution_path_ucs)
print("Recharging Points:", charging_point)

# Visualize the solution
visualize_grid_and_path(grid, solution_path_ucs, charging_point)
```

## OUTPUT:

```
-----Uniform Cost Search algorithm Solution Path-----
-----Discharging and Recharging States with Charge Level-----
(0, 0) = 100
(0, 0) = 100
(0, 1) = 90
(0, 2) = 80
(1, 2) = 70
(2, 2) = 60
(2, 3) = 50
(3, 3) = 40
(4, 3) = 30
(5, 3) = 20
(6, 3) = 10
(6, 4) = 100
(7, 4) = 90
(7, 5) = 80
(8, 5) = 70
(8, 6) = 60
(9, 6) = 50
(9, 7) = 40
(9, 8) = 30
(9, 9) = 20
Recharging Points: [(6, 4)]
```



## Using A\* algorithm:

```
# Solve the problem with the A* algorithm
print("-----A* Search algorithm Solution Path-----")
print("-----Discharging and Recharging States with Charge Level-----")
solution_path_a_star = agent.a_star_search()
charging_point = agent.battery_manager(solution_path_a_star)
print("Recharging Points:", charging_point)

# Visualize the solution
visualize_grid_and_path(grid, solution_path_a_star, charging_point)
```

## OUTPUT:

```
-----A* Search algorithm Solution Path-----
-----Discharging and Recharging States with Charge Level-----
(0, 0) = 100
(0, 0) = 100
(0, 1) = 90
(0, 2) = 80
(1, 2) = 70
(2, 2) = 60
(3, 2) = 50
(3, 3) = 40
(4, 3) = 30
(5, 3) = 20
(6, 3) = 10
(6, 4) = 100
(7, 4) = 90
(8, 4) = 80
(8, 5) = 70
(9, 5) = 60
(9, 6) = 50
(9, 7) = 40
(9, 8) = 30
(9, 9) = 20
Recharging Points: [(6, 4)]
```

