NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ARTIFICIAL INTELLIGENCE LAB

NAME	Ayesha Imran
Class	CS-A
Lab	01
Course	Artificial Intelligence
Date	1-October-25
Submitted To	Lec. Ijlal Haider

IN LAB TASKS

Task 1:

```
Implement a Simple Reflex Agent for a Grid-Based Environment with the following
Requirements:
1. The agent should suck the dirt if the current square is dirty. This is the
primary action
the agent performs when it detects dirt in its current square.
2. If the square is clean, the agent should move randomly in one of the available
directions (north, south, west, east). However, the agent should avoid moving
into
walls (obstacles). The agent is provided with bumper sensor data indicating
whether
there is a wall in each direction.
3. The agent cannot directly access any variables in the environment. Instead, it
only
receives:
• Bumper Sensor Data: A dictionary that indicates whether there is a wall
(bumper) in each of the four directions (north, south, west, east).
• Dirt Sensor: A Boolean value that indicates whether the current square is
dirty.
4. The simulation stops when either:
• All squares in the grid are cleaned, or
• The agent reaches the maximum number of steps.
```

Solution:

```python
import random

class SimpleReflexAgent:
    def __init__(self, grid, max_steps=20):
        self.grid = grid
        self.rows = len(grid)
        self.cols = len(grid[0])
        self.position = (0, 0)      # agent starts at top-left
        self.max_steps = max_steps
        self.steps = 0

    # Check if current cell is dirty
    def dirt_sensor(self):
        r, c = self.position
        return self.grid[r][c] == 1

    # Check where the walls are
    def bumper_sensor(self):
```

```
        r, c = self.position
        return {
            "north": r == 0,
            "south": r == self.rows - 1,
            "west": c == 0,
            "east": c == self.cols - 1
        }

    # Clean the current cell
    def suck(self):
        r, c = self.position
        self.grid[r][c] = 0
        print(f"Step {self.steps}: Cleaned at {self.position}")

    # Move randomly to a free direction
    def move(self):
        walls = self.bumper_sensor()
        directions = [d for d, is_wall in walls.items() if not is_wall]  # only
valid moves

        direction = random.choice(directions)
        r, c = self.position

        if direction == "north": r -= 1
        elif direction == "south": r += 1
        elif direction == "west": c -= 1
        elif direction == "east": c += 1

        self.position = (r, c)
        print(f"Step {self.steps}: Moved {direction} to {self.position}")

    # Check if all cells are clean
    def all_clean(self):
        for row in self.grid:
            if 1 in row:
                return False
        return True

    # Run the agent
    def run(self):
        while self.steps < self.max_steps:
            self.steps += 1

            if self.dirt_sensor():
                self.suck()
```

```
            else:
                self.move()

            if self.all_clean():
                print("✅ All clean! Done.")
                return
        print("⚠️ Max steps reached. Stopping.")


# ----------------------------
# MAIN PROGRAM
# ----------------------------
grid = [
    [1, 0, 1],
    [1, 1, 0],
    [0, 1, 1]
]

agent = SimpleReflexAgent(grid, max_steps=20)
agent.run()
```

Output:

```
Step 1: Cleaned at (0, 0)
Step 2: Moved east to (0, 1)
Step 3: Moved south to (1, 1)
Step 4: Cleaned at (1, 1)
Step 5: Moved east to (1, 2)
Step 6: Moved west to (1, 1)
Step 7: Moved south to (2, 1)
Step 8: Cleaned at (2, 1)
Step 9: Moved north to (1, 1)
Step 10: Moved south to (2, 1)
Step 11: Moved north to (1, 1)
```

```
Step 12: Moved west to (1, 0)
Step 13: Cleaned at (1, 0)
Step 14: Moved north to (0, 0)
Step 15: Moved south to (1, 0)
Step 16: Moved east to (1, 1)
Step 17: Moved east to (1, 2)
Step 18: Moved north to (0, 2)
Step 19: Cleaned at (0, 2)
Step 20: Moved west to (0, 1)
Max steps reached. Stopping.
```

Task 2:

```
Implement a Model-Based Reflex Agent for a Grid-Based Environment. The agent's
decisions are based on both its percepts (sensor readings) and its internal
state. The agent
needs to track its environment's cleanliness and avoid revisiting locations that
it has already
cleaned.
1. If the agent detects that the current square is dirty, it should clean the
square.
2. The agent should move to an adjacent square (north, south, west, or east) if
there are
no bumpers (walls) in that direction and the square has not already been visited.
3. The agent should maintain an internal map of cleanliness and a record of
visited
positions to avoid revisiting and cleaning the same square more than once.
4. The agent has access to bumper sensor data which tells it if there are walls
in each of
the four directions (north, south, west, east).
5. The agent will move and clean the environment until:
• All squares are cleaned,
• The agent reaches the maximum number of steps allowed.
```

Solution:

```python
import numpy as np

class ModelBasedReflexAgent:
    def __init__(self, grid_shape):
        self.visited = set()
        self.grid_shape = grid_shape

    def next_action(self, pos, bumpers, dirty):
        self.visited.add(tuple(pos))
        if dirty:
            return "suck"
        # Find unvisited, reachable neighbors
        moves = []
        directions = {
            "north": (-1, 0),
            "south": (1, 0),
            "west": (0, -1),
            "east": (0, 1)
        }
        for d, (dr, dc) in directions.items():
            if not bumpers[d]:
                new_r, new_c = pos[0] + dr, pos[1] + dc
                if (0 <= new_r < self.grid_shape[0] and
                    0 <= new_c < self.grid_shape[1] and
                    (new_r, new_c) not in self.visited):
                    moves.append(d)
        if moves:
            return np.random.choice(moves)
        # If all neighbors visited, move randomly to any open direction
        open_moves = [d for d in directions if not bumpers[d]]
        return np.random.choice(open_moves) if open_moves else None

def environment_model_based(agent, grid, max_steps=50):
    pos = [0, 0]
    cleaned = 0
    steps = 0
    while steps < max_steps and np.any(grid == 1):
        r, c = pos
        dirty = grid[r, c] == 1
        bumpers = {
            "north": r == 0,
            "south": r == grid.shape[0] - 1,
            "west": c == 0,
```

```
            "east": c == grid.shape[1] - 1
        }
        action = agent.next_action(pos, bumpers, dirty)
        print(f"Step {steps+1}: Pos {pos}, Dirty: {dirty}, Action: {action}")
        if action == "suck":
            grid[r, c] = 0
            cleaned += 1
        elif action == "north" and not bumpers["north"]:
            pos[0] -= 1
        elif action == "south" and not bumpers["south"]:
            pos[0] += 1
        elif action == "west" and not bumpers["west"]:
            pos[1] -= 1
        elif action == "east" and not bumpers["east"]:
            pos[1] += 1
        steps += 1
    print("Final grid:\n", grid)
    print("Total cleaned squares:", cleaned)

# Example grid
grid = np.array([
    [1, 0, 1],
    [0, 1, 0],
    [1, 0, 1]
])

agent = ModelBasedReflexAgent(grid.shape)
environment_model_based(agent, grid, max_steps=20)
```

Output:

```
Step 1: Pos [0, 0], Dirty: True, Action: suck
Step 2: Pos [0, 0], Dirty: False, Action: east
Step 3: Pos [0, 1], Dirty: False, Action: south
Step 4: Pos [1, 1], Dirty: True, Action: suck
Step 5: Pos [1, 1], Dirty: False, Action: south
Step 6: Pos [2, 1], Dirty: False, Action: east
Step 7: Pos [2, 2], Dirty: True, Action: suck
```

```
Step 8: Pos [2, 2], Dirty: False, Action: north
Step 9: Pos [1, 2], Dirty: False, Action: north
Step 10: Pos [0, 2], Dirty: True, Action: suck
Step 11: Pos [0, 2], Dirty: False, Action: south
Step 12: Pos [1, 2], Dirty: False, Action: north
Step 13: Pos [0, 2], Dirty: False, Action: west
Step 14: Pos [0, 1], Dirty: False, Action: east
Step 15: Pos [0, 2], Dirty: False, Action: south
Step 16: Pos [1, 2], Dirty: False, Action: south
Step 17: Pos [2, 2], Dirty: False, Action: west
Step 18: Pos [2, 1], Dirty: False, Action: west
Step 19: Pos [2, 0], Dirty: True, Action: suck
Final grid:
 [[0 0 0]
 [0 0 0]
 [0 0 0]]
Total cleaned squares: 5
```

Post Lab Tasks

Task:

Implement a Simple Reflex Agent that monitors the temperature in a residential area using sensors.Implement a program in Python where the agent gets temperature readings from multiple sensors (assume 9 sensors are installed in different locations). Each sensor measures the temperature in Celsius.

- The agent should calculate the average temperature for all the sensors and convert it into Fahrenheit.
- Implement simple condition-action rules for deciding the actions based on the temperature range.

o If the average temperature is above 85°F, the agent should activate a cooling mechanism
o if average temperature is below 60°F, the agent should activate heating.
o If the temperature is within the comfortable range (60°F to 85°F), no action should be taken.
Visualize the sensor readings and the actions taken by the agent. Use a simple plot
(e.g., using Matplotlib) to:
o Display the temperature data for each sensor.
o Show the corresponding actions (cooling, heating, or no action).

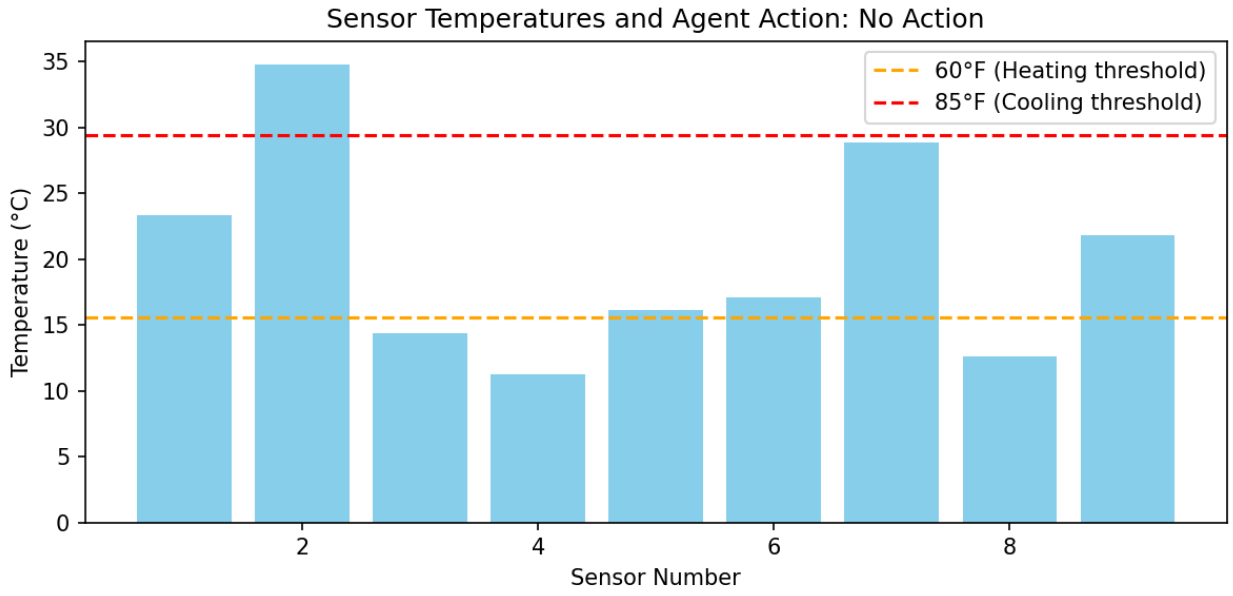
Solution :

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

# Generate random Celsius readings for 9 sensors (e.g., between 10°C and 35°C)
sensor_readings_c = np.random.uniform(10, 35, 9)

# Calculate average in Celsius and convert to Fahrenheit
avg_c = np.mean(sensor_readings_c)
avg_f = avg_c * 9/5 + 32

# Decide action based on average Fahrenheit temperature
if avg_f > 85:
    action = "Cooling Activated"
elif avg_f < 60:
    action = "Heating Activated"
else:
    action = "No Action"

# Print results
print("Sensor readings (°C):", np.round(sensor_readings_c, 2))
print(f"Average temperature: {avg_c:.2f}°C / {avg_f:.2f}°F")
print("Action:", action)

# Visualization
plt.figure(figsize=(8,4))
plt.bar(range(1, 10), sensor_readings_c, color='skyblue')
plt.axhline((60-32)*5/9, color='orange', linestyle='--', label='60°F (Heating threshold)')
plt.axhline((85-32)*5/9, color='red', linestyle='--', label='85°F (Cooling threshold)')
plt.title(f"Sensor Temperatures and Agent Action: {action}")
plt.xlabel("Sensor Number")
plt.ylabel("Temperature (°C)")
```

```
plt.legend()
plt.tight_layout()
plt.show()
```

Output:

```
Sensor readings (°C): [23.3  34.8  14.38 11.29 16.14 17.12 28.83 12.6  21.79]
Average temperature: 20.03°C / 68.05°F
Action: No Action
```



Task 4:

After successfully implementing a reflex agent, extend it to a Model-Based Agent. In your implementation, the agent should:

- ❖ Maintain an internal model of the environment (e.g., temperature history).
- ❖ Use this model to predict the future state of the environment and make
- ❖ decisions based on the predicted state, not just the current percept.

Solution :

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

class ModelBasedTempAgent:
    def __init__(self):
```

```
        self.history = []  # Store temperature history

    def update_history(self, readings_c):
        self.history.append(readings_c.copy())

    def predict_next_avg(self):
        # Simple prediction: average of last 2 readings, or current if only one
        if len(self.history) >= 2:
            last = np.mean(self.history[-1])
            prev = np.mean(self.history[-2])
            return (last + prev) / 2
        elif self.history:
            return np.mean(self.history[-1])
        else:
            return None

    def decide_action(self, predicted_avg_f):
        if predicted_avg_f > 85:
            return "Cooling Activated"
        elif predicted_avg_f < 60:
            return "Heating Activated"
        else:
            return "No Action"

# Simulate readings for 9 sensors over 10 time steps
agent = ModelBasedTempAgent()
actions = []
predicted_avgs_f = []

for t in range(10):
    readings_c = np.random.uniform(10, 35, 9)
    agent.update_history(readings_c)
    predicted_avg_c = agent.predict_next_avg()
    predicted_avg_f = predicted_avg_c * 9/5 + 32
    action = agent.decide_action(predicted_avg_f)
    actions.append(action)
    predicted_avgs_f.append(predicted_avg_f)
    print(f"Time {t+1}: Readings (°C): {np.round(readings_c,2)} | Predicted Avg:
{predicted_avg_c:.2f}°C / {predicted_avg_f:.2f}°F | Action: {action}")

# Visualization
plt.figure(figsize=(10,5))
for i in range(9):
    plt.plot([agent.history[t][i] for t in range(10)], label=f"Sensor {i+1}")
```

```
plt.axhline((60-32)*5/9, color='orange', linestyle='--', label='60°F (Heating
threshold)')
plt.axhline((85-32)*5/9, color='red', linestyle='--', label='85°F (Cooling
threshold)')
plt.title("Sensor Temperatures Over Time")
plt.xlabel("Time Step")
plt.ylabel("Temperature (°C)")
plt.legend(loc='upper left', bbox_to_anchor=(1,1))
plt.tight_layout()
plt.show()

plt.figure(figsize=(8,3))
plt.plot(predicted_avgs_f, marker='o', label='Predicted Avg (°F)')
plt.title("Predicted Average Temperature (°F) and Actions")
plt.xlabel("Time Step")
plt.ylabel("Predicted Avg Temp (°F)")
for i, act in enumerate(actions):
    plt.text(i, predicted_avgs_f[i]+1, act, ha='center', fontsize=8)
plt.axhline(60, color='orange', linestyle='--')
plt.axhline(85, color='red', linestyle='--')
plt.tight_layout()
plt.show()
```

Output:

```
Time 1: Readings (°C): [27.07 29.9  30.31 21.82 32.6  26.21 14.75
16.06 24.14] | Predicted Avg: 24.76°C / 76.57°F | Action: No Action

Time 2: Readings (°C): [33.87 30.83 33.02 24.03 22.81 23.18 25.54
10.39 17.94] | Predicted Avg: 24.69°C / 76.45°F | Action: No Action

Time 3: Readings (°C): [15.36 29.8  16.76 29.58 23.23 32.26 30.43
30.36 33.49] | Predicted Avg: 25.72°C / 78.29°F | Action: No Action

Time 4: Readings (°C): [25.   12.9  16.95 29.02 24.47 26.88 29.29
29.37 17.35] | Predicted Avg: 25.14°C / 77.25°F | Action: No Action

Time 5: Readings (°C): [34.82 28.97 10.14 18.55 23.36 12.99 12.82
16.41 12.02] | Predicted Avg: 21.18°C / 70.13°F | Action: No Action

Time 6: Readings (°C): [26.21 16.91 22.85 17.27 12.24 29.53 25.87
21.79 12.79] | Predicted Avg: 19.75°C / 67.55°F | Action: No Action
```

Time 7: Readings (°C): [11.45 15.99 10.43 13.46 10.96 21.66 25.03 21.17 19.55] | Predicted Avg: 18.62°C / 65.52°F | Action: No Action

Time 8: Readings (°C): [33.22 27.41 20.96 20.73 19.04 15.46 21.59 22.66 16.08] | Predicted Avg: 19.27°C / 66.69°F | Action: No Action

Time 9: Readings (°C): [26.74 11.98 25.09 26.64 28.87 31.57 23.33 22.81 30.86] | Predicted Avg: 23.61°C / 74.50°F | Action: No Action

Time 10: Readings (°C): [27.63 29.23 20.09 22.89 30.65 15.97 19.64 24.01 32.14] | Predicted Avg: 25.01°C / 77.01°F | Action: No Action