Practical Task: Understanding and Applying Markov Decision Processes (MDPs) in Reinforcement Learning

Objective:
By the end of this 3-hour session, you will understand the basic concepts of MDPs and implement a simple MDP environment. You will also create a basic policy to solve the MDP using Python.

Task Outline

Part 1: Theory Review and Setup (30 minutes)

1. **Quick Recap (10 minutes)**
   - Review the core components of MDPs:
     - States (S)
     - Actions (A)
     - Transition probabilities (P)
     - Rewards (R)
     - Discount factor ()
   - Explain how these components are used in RL to model decision-making problems.
   - Discuss the Bellman equation briefly.
2. **Environment Setup (20 minutes)**
   - Guide students to set up their Python environment with the following libraries:
     - gym or gymnasium (for environment simulation)
     - numpy (for matrix operations)

Part 2: Implementing a Simple MDP (60 minutes)

1. **Define the MDP (30 minutes)**

Students create a simple custom environment:

- **Scenario:** A robot navigating a 3x3 grid world to reach a goal state while avoiding a danger zone.
  - States: 9 cells of the grid.
  - Actions: {Up, Down, Left, Right}.
  - Transition Probabilities:
    - 80% chance of moving in the intended direction.
    - 20% chance of moving in a random direction.
  - Rewards:
    - +10 for reaching the goal.

- -5 for entering the danger zone.
- 0 otherwise.
- Discount factor: 0.9.

Instructions:

- Define the grid world as a 2D array.
- Write functions for:
  - Transition dynamics (given a state and action, determine the next state).
  - Reward calculation.

2. **Simulate the Environment (30 minutes)**

- Simulate random episodes in the environment.
- Visualize the robot's movement in the grid.

Part 3: Implementing a Policy (60 minutes)

1. **Create a Simple Policy (30 minutes)**

- Students write a policy to maximize rewards (e.g., greedy policy).
- Test the policy on the environment and calculate the total reward per episode.

2. **Enhance the Policy (Optional - 30 minutes)**

- Implement Value Iteration or Policy Iteration.
- Compare the results with the initial greedy policy.

Deliverables

1. Python script defining the custom MDP environment.
2. Visualization of the agent's trajectory in the grid world for random and policy-driven behavior.
3. Documentation or comments explaining each step.

Evaluation Criteria

1. Correct implementation of the MDP environment (30%).
2. Simulation of the agent's behavior in the environment (30%).
3. Implementation and evaluation of the policy (40%).

```
import numpy as np
import random

class GridWorldMDP:
    def __init__(self, grid_size=3, goal_state=(2, 2), danger_state=(1, 1)):
        self.grid_size = grid_size
        self.goal_state = goal_state
        self.danger_state = danger_state
        self.states = [(i, j) for i in range(grid_size) for j in range(grid_size)]
        self.actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
        self.reward_map = self.create_reward_map()
        self.transition_prob = 0.8
        self.discount_factor = 0.9

    def create_reward_map(self):
        """Initialize the grid with rewards for goal and danger states."""
        reward_map = np.zeros((self.grid_size, self.grid_size))
        reward_map[self.goal_state] = 10
        reward_map[self.danger_state] = -5
        return reward_map

    def is_valid_state(self, state):
        """Check if a state is within the grid boundaries."""
        x, y = state
        return 0 <= x < self.grid_size and 0 <= y < self.grid_size

    def get_next_state(self, state, action):
        """Compute the next state based on the action taken."""
        x, y = state
        if action == 'UP':
            next_state = (x - 1, y)
        elif action == 'DOWN':
            next_state = (x + 1, y)
        elif action == 'LEFT':
            next_state = (x, y - 1)
        elif action == 'RIGHT':
            next_state = (x, y + 1)
        else:
            next_state = state
        return next_state if self.is_valid_state(next_state) else state

    def step(self, state, action):
        """
        Simulate one step in the environment:
        - 80% chance of following the intended action.
        - 20% chance of transitioning randomly.
```

```
"""
if random.random() < self.transition_prob:
    next_state = self.get_next_state(state, action)
else:
    rand_action = random.choice(self.actions)
    next_state = self.get_next_state(state, rand_action)
reward = self.reward_map[next_state]
done = (next_state == self.goal_state)
return next_state, reward, done
```

Value Iteration Equation

For each state s, the Value Iteration algorithm updates the value function $V(s)$ using:

$$V(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} P(s'|s,a) \left[R(s,a,s') + \gamma V(s') \right]$$

Where:

- $V(s)$: Value of state s (expected long-term reward starting from s).
- a: Action chosen from the set of possible actions $\mathcal{A}(s)$.
- $P(s'|s,a)$: Transition probability of reaching state s' from state s by taking action a.
- $R(s,a,s')$: Immediate reward for taking action a in state s and ending up in state s'.
- γ: Discount factor ($0 \leq \gamma \leq 1$), which determines the importance of future rewards.

How you can use in the code

The value function update is essentially:

$$V(s) \leftarrow \max_{a \in \mathcal{A}(s)} \left(0.8 \cdot \left[R(s,a,s') + \gamma \cdot V(s') \right] + \sum_{a' \in \mathcal{A}(s)} \frac{0.2}{|\mathcal{A}(s)|} \cdot \left[R(s,a',s'') + \gamma \cdot V(s'') \right] \right)$$

1. **Intended vs. Random Transitions**:
   - 80% of the time, the agent transitions to the expected next state s'.
   - 20% of the time, the agent randomly transitions to another state s''.
2. **Dynamic Programming**:
   - The algorithm iteratively computes $V(s)$ for all states by considering all possible actions and transitions.
3. **Optimal Policy**:
   - After convergence, the policy chooses the action a that yields the highest expected return from state s.

```
def main():
    env = GridWorldMDP()
    print("\n--- Value Iteration (Student Task) ---")
    # Students call value_iteration here and print results
    # V, optimal_policy = value_iteration(env)

    print("\n--- Simulate Policy Episode (Student Task) ---")
    # Students call simulate_policy_episode here to test their policy
    # simulate_policy_episode(env, optimal_policy, max_steps=20)


if __name__ == "__main__":
    random.seed(0)
    np.random.seed(0)
    main()
```