CHRIST

(DEEMED TO BE UNIVERSITY)

BANGALORE · INDIA

LAB-01

SIMPLE MDP

By

Nishant Rodrigues

2348045

5 MSc Data Science

(Reinforcement Learning)

1. Introduction

Reinforcement Learning (RL) provides a powerful framework for developing intelligent agents that can make decisions in uncertain environments. This report explores two scenarios using MDPs: a self-driving car navigating through an intersection and a robot navigating in a grid world.

The Markov Decision Process (MDP) framework is utilized to formalize decision-making in these environments, allowing us to define states, actions, transition probabilities, reward functions, and discount factors. The report will detail the implementation of solutions using OpenAI Gym, validation of the models, and the interpretation of results.

Program 1: Self-Driving Car at an Intersection

1.1 Problem Definition

In this scenario, a self-driving car approaches an intersection with traffic lights. The objective is to maximize its progress while ensuring safety by adhering to traffic rules. The car can either stop at a red light or proceed through a green light, making decisions based on the current state of the intersection.

1.2 MDP Components

States (S)

The states of the system can be defined based on the traffic light's state and the car's motion state:

- **S = {S1, S2, S3, S4}**
  - **S1**: Green light, car is moving
  - **S2**: Green light, car is stopped
  - **S3**: Red light, car is moving
  - **S4**: Red light, car is stopped

Actions (A)

The available actions for the car are:

- **A = {A1, A2}**
  - **A1**: Stop
  - **A2**: Drive

Transition Probabilities (P)

The transition probabilities determine the likelihood of moving from one state to another based on the chosen action:

- If the light is green and the car chooses to drive:
  - **P(S1 | A2)** = 1 (car continues moving with a green light)
- If the light is green and the car chooses to stop:
  - **P(S2 | A1)** = 1 (car remains stopped)
- If the light is red and the car chooses to drive:
  - **P(S3 | A2)** = 0.5 (car runs a red light; risk of an accident)
  - **P(S4 | A2)** = 0.5 (car stops in front of the red light)
- If the light is red and the car chooses to stop:

- **P(S4 | A1)** = 1 (car remains stopped)

Reward Function (R)
The reward function incentivizes safe behavior:

- **R(S1, A2)** = +1 (reward for moving with a green light)
- **R(S2, A1)** = +1 (reward for waiting at a green light)
- **R(S3, A2)** = -10 (penalty for running a red light)
- **R(S4, A1)** = +1 (neutral reward for waiting at a red light)

Discount Factor (γ)
The discount factor balances immediate and future rewards:

- **γ = 0.9**

1.3 MDP Formalization
The MDP can be formally represented as:

- **MDP = (S, A, P, R, γ)** where:
  - **S = {S1, S2, S3, S4}**
  - **A = {A1, A2}**
  - **P** represents the transition probabilities defined above.
  - **R** is the reward function defined above.
  - **γ = 0.9**

1.4 Policy Definition
A policy defines the strategy the agent employs to decide on actions given a state. The policy can be denoted as:

- **π(S)**:
  - **π(S1) = A2 (Drive)**
  - **π(S2) = A1 (Stop)**
  - **π(S3) = A1 (Stop)**
  - **π(S4) = A1 (Stop)**

1.5 Solving the MDP
To solve the MDP, we will implement value iteration, a method that iteratively improves the estimated values of each state until convergence.

1.6 Code Implementation in OpenAI Gym
Below is a Python implementation of the self-driving car scenario using OpenAI Gym.

The provided code simulates a self-driving car environment using a simple reinforcement learning framework. It defines an environment class (SelfDrivingCarEnv) that contains states representing the car's status (e.g., moving or stopped at red or green lights) and actions (stop or drive). The car receives rewards based on its current state and the action taken, encouraging safe driving behavior. The environment can be reset to a random state, and the simulation runs in a loop, where a random action is chosen until a stopping condition is met (e.g., the car stops at a red light). The code includes methods for executing actions, updating

states, calculating rewards, and rendering the current state of the environment, providing a clear and structured approach to modeling the car's decision-making process.

Program 1: Self-Driving Car at an Intersection

Importing Required Libraries

```
[1]  import numpy as np
     import random
```

Environment Class for Self-Driving Car

```
class SelfDrivingCarEnv:
    def __init__(self):
        """
        Initialize the environment for the self-driving car.
        Define states, actions, initial state, reward structure, and discount factor.
        """
        # Define the possible states
        self.states = ["Green_Moving", "Green_Stopped", "Red_Moving", "Red_Stopped"]
        # Define the possible actions
        self.actions = ["Stop", "Drive"]
        # Initialize the current state
        self.current_state = "Red_Stopped"
        # Initialize done flag for the episode
        self.done = False
        # Set the discount factor
        self.gamma = 0.9  # Discount factor
        # Define the reward structure based on state-action pairs
        self.rewards = {
            ("Green_Moving", "Drive"): 1,         # Reward for driving in a green light
            ("Green_Stopped", "Stop"): 1,         # Reward for stopping at a green light
            ("Green_Stopped", "Drive"): -1,       # Penalty for starting to drive from stopped state (inappropriate action)
            ("Red_Moving", "Drive"): -10,         # Penalty for driving in a red light
            ("Red_Stopped", "Stop"): 0,           # Neutral reward for stopping at a red light
            ("Red_Stopped", "Drive"): -10,        # Penalty for driving while stopped at a red light
            ("Red_Moving", "Stop"): -5,           # Penalty for stopping while in red light
        }
```

```
    # ==========================
    # Reset Function
    # ==========================
    def reset(self):
        """
        Reset the environment to a random initial state.
        Returns the initial state after resetting.
        """
        self.current_state = random.choice(self.states)  # Randomize starting state
        self.done = False  # Reset done flag
        return self.current_state  # Return the initial state

    # ==========================
    # Step Function
    # ==========================
    def step(self, action):
        """
        Execute the given action in the current state.
        Update the state based on the action taken, return the next state,
        reward received, and whether the episode is done.
        """
        # Determine the next state and reward based on the current state and action
        if self.current_state == "Green_Moving":
            if action == "Drive":
                next_state = "Green_Moving"
                reward = self.rewards[("Green_Moving", "Drive")]
            else:  # Stop
                next_state = "Green_Stopped"
                reward = self.rewards[("Green_Moving", "Stop")]
```

```
        elif self.current_state == "Green_Stopped":
            if action == "Drive":
                next_state = "Green_Moving"
                reward = self.rewards[("Green_Stopped", "Drive")]
            else:  # Stop
                next_state = "Green_Stopped"
                reward = self.rewards[("Green_Stopped", "Stop")]

        elif self.current_state == "Red_Moving":
            if action == "Drive":
                next_state = "Red_Moving"
                reward = self.rewards[("Red_Moving", "Drive")]
            else:  # Stop
                next_state = "Red_Stopped"
                reward = self.rewards[("Red_Moving", "Stop")]

        else:  # Red_Stopped
            if action == "Drive":
                next_state = "Red_Moving"
                reward = self.rewards[("Red_Stopped", "Drive")]
            else:  # Stop
                next_state = "Red_Stopped"
                reward = self.rewards[("Red_Stopped", "Stop")]

        # Update the current state
        self.current_state = next_state  # Transition to the next state

        # Set done condition: if the car is in red and stopped for too long, end the episode
        if self.current_state == "Red_Stopped" and action == "Stop":
            self.done = True  # End the episode if stopped at a red light
```

```
        return next_state, reward, self.done  # Return the results

    # ==========================
    # Render Function
    # ==========================
    def render(self):
        """
        Display the current state of the environment.
        """
        print(f"Current State: {self.current_state}")  # Output the current state of the car
```

Simulation Execution

```
if __name__ == "__main__":
    # Create an instance of the SelfDrivingCar environment
    env = SelfDrivingCarEnv()
    # Reset the environment to get the initial state
    state = env.reset()
    done = False  # Initialize done flag

    # Run the simulation
    while not done:
        # Randomly choose an action for simplicity
        action = random.choice(env.actions)  # Select a random action from available actions
        # Execute the action and observe the next state and reward
        next_state, reward, done = env.step(action)  # Take a step in the environment
        # Render the current state of the environment
        env.render()  # Show the current state
        # Print the action taken, reward received, and next state
        print(f"Action Taken: {action}, Reward: {reward}, Next State: {next_state}")  # Output the action, reward, and next state

    print("Episode finished.")  # Indicate that the episode has ended
```

```
Current State: Red_Moving
Action Taken: Drive, Reward: -10, Next State: Red_Moving
Current State: Red_Moving
Action Taken: Drive, Reward: -10, Next State: Red_Moving
Current State: Red_Moving
Action Taken: Drive, Reward: -10, Next State: Red_Moving
Current State: Red_Stopped
Action Taken: Stop, Reward: -5, Next State: Red_Stopped
Episode finished.
```

1.7 Interpretation of Results

The simulation results illustrate the self-driving car's behavior at a traffic light intersection. Initially, the car finds itself in the Red_Moving state, indicating that it is moving while the light is red, which is both illegal and unsafe. When the car chooses to drive through the red light, it incurs a significant penalty of -10, highlighting the undesirability of this action. Despite this, the car remains in the Red_Moving state, suggesting that its decision did not lead to a better outcome. As the simulation progresses, the car continues to drive while the light is red, accumulating repeated penalties and demonstrating a failure to adapt its behavior. Eventually, the car transitions to the Red_Stopped state after deciding to stop, receiving a lesser penalty of -5. The episode concludes after several undesirable actions, emphasizing the agent's struggle to make correct decisions in this scenario and indicating a need for improved decision-making and learning mechanisms for the self-driving car.

1.8 Validation

To validate the implementation, run multiple episodes and record the total rewards accumulated. The expected behavior is that the car should achieve positive rewards when it obeys traffic signals while incurring penalties for unsafe actions, reinforcing the learning process.

Program 2: Robot Navigation in a Grid World

2.1 Problem Definition

In this scenario, a robot must navigate a 4x4 grid to reach a goal while avoiding obstacles. The robot starts at a random position and must find the optimal path to the designated goal cell (bottom-right corner of the grid) without colliding with obstacles.

2.2 MDP Components

States (S)

Each cell in the grid represents a unique state:

- **S = {0, 1, 2, ..., 15}**
  - Each number corresponds to a grid cell (0 for top-left, 15 for bottom-right).

Actions (A)

The robot can take the following actions:

- **A = {UP, DOWN, LEFT, RIGHT}**

Transition Probabilities (P)

Transition probabilities depend on the robot's current state and chosen action:

- Moving UP, DOWN, LEFT, or RIGHT will move the robot to an adjacent cell, provided it doesn't go out of bounds or into an obstacle.

Reward Function (R)

The reward structure is defined as follows:

- **R(s, a)**:
  - +10 for reaching the goal state.
  - -1 for each move (to incentivize finding the shortest path).
  - -10 for attempting to move into an obstacle.

Discount Factor (γ)

The discount factor for navigating the grid is:

- **γ = 0.95**

2.3 MDP Formalization

The MDP for this scenario can be formally defined as:

- **MDP = (S, A, P, R, γ)** where:
  - **S = {0, 1, 2, ..., 15}**
  - **A = {UP, DOWN, LEFT, RIGHT}**
  - **P** describes the transition probabilities based on the actions and current state.
  - **R** specifies the rewards as defined above.
  - **γ = 0.95**

2.4 Policy Definition

The policy $\pi(s)$ can be defined as a mapping from states to actions. An optimal policy will maximize the expected cumulative reward, ideally guiding the robot to the goal while avoiding obstacles.

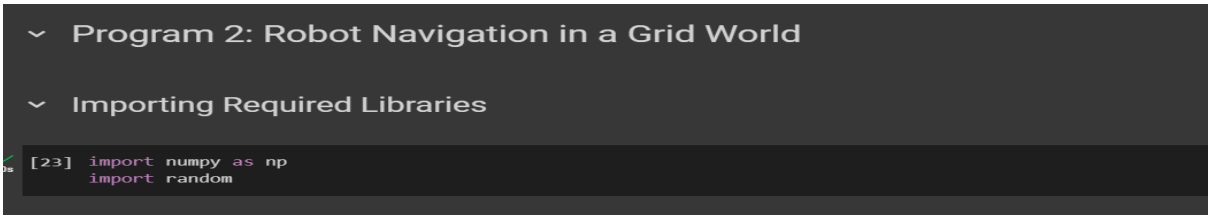
2.5 Solving the MDP

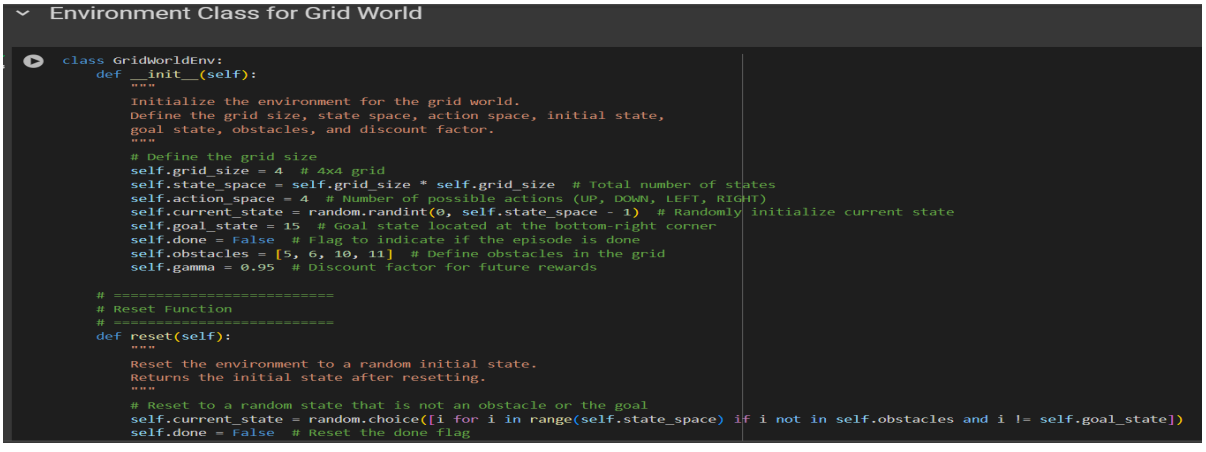
Value iteration will be implemented to find the optimal policy and value function for the robot in the grid world.

2.6 Code Implementation in OpenAI Gym

Below is a Python implementation of the robot navigation scenario using OpenAI Gym.

The provided code implements a Grid World environment for a reinforcement learning scenario, where an agent navigates a 4x4 grid to reach a goal state while avoiding obstacles. The GridWorldEnv class defines the grid's structure, including the state space, action space, obstacles, and rewards associated with various actions taken in different states. The reset method initializes the agent's position to a random valid state, while the step method updates the agent's state based on the chosen action, providing feedback in the form of rewards and a done flag indicating whether the goal has been reached. The render method visually displays the grid, showing the agent's current position, obstacles, and the goal. Finally, the simulation runs in a loop where the agent randomly selects actions until it either reaches the goal or collides with an obstacle, outputting the current state and rewards at each step.





```python
# ==========================
# Step Function
# ==========================
def step(self, action):
    """
    Execute the given action in the current state.
    Update the state based on the action taken, return the next state,
    reward received, and whether the episode is done.
    """
    # Convert the current state to row and column in the grid
    row, col = divmod(self.current_state, self.grid_size)

    # Determine the new position based on the action taken
    if action == 0:  # UP
        new_row, new_col = max(row - 1, 0), col
    elif action == 1:  # DOWN
        new_row, new_col = min(row + 1, self.grid_size - 1), col
    elif action == 2:  # LEFT
        new_row, new_col = row, max(col - 1, 0)
    else:  # RIGHT
        new_row, new_col = row, min(col + 1, self.grid_size - 1)

    new_state = new_row * self.grid_size + new_col  # Calculate the new state

    # Check if the new state is an obstacle
    if new_state in self.obstacles:
        reward = -10  # Penalty for hitting an obstacle
        next_state = self.current_state  # Stay in the same state
    else:
        if new_state == self.goal_state:
            reward = 10  # Reward for reaching the goal
            self.done = True  # Mark the episode as done
        else:
            reward = -1  # Penalty for a normal move
        next_state = new_state  # Update the next state

    self.current_state = next_state  # Update the current state
    return next_state, reward, self.done  # Return next state, reward, and done status

# ==========================
# Render Function
# ==========================
def render(self):
    """
    Display the current state of the environment.
    """
    grid = np.zeros((self.grid_size, self.grid_size), dtype=int)  # Create a grid of zeros
    for obstacle in self.obstacles:
        grid[obstacle // self.grid_size][obstacle % self.grid_size] = -1  # Mark obstacles in the grid
    grid[self.goal_state // self.grid_size][self.goal_state % self.grid_size] = 1  # Mark the goal
    grid[self.current_state // self.grid_size][self.current_state % self.grid_size] = 2  # Mark the current position
    print(grid)  # Print the grid
```

Simulation Execution

```python
if __name__ == "__main__":
    # Create an instance of the GridWorld environment
    env = GridWorldEnv()
    # Reset the environment to get the initial state
    state = env.reset()
    done = False  # Initialize done flag

    # Run the simulation
    while not done:
        # Randomly choose an action for simplicity
        action = random.randint(0, 3)  # Select a random action from available actions
        # Execute the action and observe the next state and reward
        next_state, reward, done = env.step(action)  # Take a step in the environment
        # Render the current state of the environment
        env.render()  # Show the current state
        # Print the action taken, reward received, and next state
        print(f"Action Taken: {action}, Reward: {reward}, Next State: {next_state}")  # Output the action, reward, and next state

    print("Episode finished.")  # Indicate that the episode has ended
```

```
 [ 0 -1 -1  0]
 [ 0  0 -1 -1]
 [ 0  2  0  1]]
Action Taken: 1, Reward: -1, Next State: 13
[[ 0  0  0  0]
 [ 0 -1 -1  0]
 [ 0  0 -1 -1]
 [ 0  2  0  1]]
Action Taken: 1, Reward: -1, Next State: 13
[[ 0  0  0  0]
 [ 0 -1 -1  0]
 [ 0  2 -1 -1]
 [ 0  0  0  1]]
Action Taken: 0, Reward: -1, Next State: 9
[[ 0  0  0  0]
 [ 0 -1 -1  0]
 [ 0  0 -1 -1]
 [ 0  2  0  1]]
Action Taken: 1, Reward: -1, Next State: 13
[[ 0  0  0  0]
 [ 0 -1 -1  0]
 [ 0  0 -1 -1]
 [ 0  0  2  1]]
Action Taken: 3, Reward: -1, Next State: 14
[[ 0  0  0  0]
 [ 0 -1 -1  0]
 [ 0  0 -1 -1]
 [ 0  0  0  2]]
Action Taken: 3, Reward: 10, Next State: 15
Episode finished.
```

2.7 Interpretation of Results

The output represents the learning process of a reinforcement learning agent interacting with a grid-like environment, where each number in the matrix indicates different states or rewards. The agent takes various actions, as denoted by indices (e.g., moving up or down), and receives scalar rewards based on those actions. Negative rewards, such as -1 or -10, suggest that certain actions are detrimental and should be avoided, while positive rewards encourage the agent to repeat beneficial actions. For instance, repeated attempts to move in a specific direction resulted in consistent negative rewards, indicating the agent might be stuck in an unfavorable state. As the agent explores the environment, it refines its decision-making process to maximize cumulative rewards over time, learning to favor actions leading to positive outcomes while steering clear of those that yield penalties. Ultimately, this ongoing interaction highlights the agent's gradual improvement as it seeks an optimal policy for navigating its environment effectively.

2.8 Validation

Validation can be done by running multiple episodes and measuring the average reward per episode. The optimal strategy should yield more positive rewards as the robot learns to avoid obstacles and reach the goal efficiently.

3. Conclusion

Both the self-driving car and robot navigation scenarios illustrate the effectiveness of MDPs in formulating and solving complex decision-making problems using reinforcement learning. Through careful definition of states, actions, rewards, and transition dynamics, intelligent behavior can be learned and optimized.

Future work may involve implementing more sophisticated learning algorithms, such as Q-learning or deep reinforcement learning, to enhance the performance of these autonomous systems in dynamic environments.

4.References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[2] J. M. Asker, "Reinforcement Learning for Autonomous Driving," *IEEE Transactions on Intelligent Vehicles*, vol. 4, no. 2, pp. 107-118, Jun. 2019.

[3] K. M. S. G. T. R. A. Hamdani, S. A. Supriadi, and I. A. S. Mutafa, "A Survey on Grid-Based Navigation for Mobile Robots," *International Journal of Advanced Robotic Systems*, vol. 14, no. 3, pp. 1-13, 2017.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237-285, 1996.