Experiment 10 : Solving a Markov Decision Process (MDP)

Total Marks: 100

1. Learning Objectives

Upon successful completion of this assignment, students will be able to:

- Define and understand the core components of a Markov Decision Process (MDP):
  - **States (S)**
  - **Actions (A)**
  - **Transition Model (T)**
  - **Reward Function (R)**
  - **Discount Factor (gamma)**
- Translate a "grid world" problem into a formal MDP structure.
- Implement the **Value Iteration** algorithm from scratch using only NumPy.
- Understand and apply the **Bellman Optimality Equation** to calculate state values.
- **Extract an optimal policy** (the best action for each state) from a converged value function.
- Visualize and interpret the resulting value function and policy.
- Analyze how hyperparameters (gamma and "living penalty") affect the agent's final behavior.

2. Introduction

This assignment is your first step into Reinforcement Learning. We will focus on the "planning" problem, where we have a perfect *model* of the environment (the MDP) and want to find the *best possible plan* (the optimal policy) before the agent even takes a step.

You will be implementing **Value Iteration**, a classic algorithm that repeatedly applies the Bellman equation to find the true "value" of being in every state.1 Once we know the value of all states, figuring out the best action is easy: just move to the state with the highest value!

Experiment 10 : Solving a Markov Decision Process MDP1

You will solve the "GridWorld" problem, a 3x4 grid with a goal, a "pit" (danger), and walls. Your agent must learn the shortest, safest path to the goal.

3. Prerequisites

Ensure your Python environment has the following libraries installed:

```
pip install numpy matplotlib seaborn
```

4. Experiment Tasks

Task 1: Define the GridWorld (The MDP) (30 Marks)

First, we must define the "rules of the game." You will not use any existing RL libraries (like `gym`). You will define the world yourself.

The world is a 3x4 grid:

- **States (S):** The grid cells. `(0,0)`, `(0,1)`, `(0,2)`, `(0,3)`, etc.
- **Walls:** There is a wall at `(1,1)`. The agent cannot move into this state.
- **Terminal States:**
  - **Goal:** `(0,3)` (e.g., a gem)
  - **Pit:** `(1,3)` (e.g., a fire pit)
- **Actions (A):** The agent can try to move `['up', 'down', 'left', 'right']`.

1. **Define States:** Create a list or set of all valid states (all `(row, col)` tuples, *except* the wall at `(1,1)`).
2. **Define Rewards (R):** Create a dictionary or function that defines the reward `R(s)` for *being in* a state `s`.
   - **Goal `(0,3)`:** `+1`
   - **Pit `(1,3)`:** `1`

   All other states: `0.04` (This is a "living penalty" to encourage the agent to find the *shortest* path).

3. **Define Discount Factor:** Use `gamma = 0.99` .
4. **Define Transition Model (T):** This is the most important part. You must create a function that defines the *probabilities* of moving. The world is **stochastic** (unpredictable).

If the agent chooses an action (e.g., `'up'`):

80% chance it goes in the intended direction (e.g., `'up'`).

10% chance it slips and goes 90 degrees to the **left** (e.g., `'left'`).

10% chance it slips and goes 90 degrees to the **right** (e.g., `'right'`).

Handling Walls/Boundaries: If a move (intended or slipped) would land the agent in a wall (like `(1,1)`) or off the grid, the agent **stays in its current state**.2

Terminal States: Once the agent enters a terminal state (Goal or Pit), it stays there and receives no further rewards. (For Value Iteration, we can simplify this: terminal states have a value of 0 and no actions leading out).3

To Implement: Create a helper function `get_next_states(s, a)` that, given a state s and action a, returns a list of `(probability, next_state)` tuples. For example, from `(0,0)`:

`get_next_states((0,0), 'right')` might return:

`[(0.8, (0,1)), (0.1, (0,0)), (0.1, (1,0))]`

(0.8 for 'right', 0.1 for 'up' (slips left, hits wall, stays at (0,0)), 0.1 for 'down' (slips right)).

Task 2: Value Iteration Algorithm (From Scratch) (40 Marks)

Now you will implement the algorithm to *solve* the MDP.

1. **Initialize Value Function:** Create your main data structure, `V` , which will be a dictionary or a 2D NumPy array. `V[s]` stores the current estimated value of being in state `s` . Initialize the value of all states to **0.0**.
2. **Implement Value Iteration:**

   You will loop until the `V` function converges.

   Convergence is when the maximum change in `V` for any state in a single iteration is very small.

   Use a threshold `theta = 0.0001` .
3. **Run:** Run your `value_iteration` function. It should return the final, converged `V` table.

Task 3: Policy Extraction (From Scratch) (15 Marks)

The `V` table tells you how *good* each state is, but not *what to do*. Now, you must extract the optimal policy (`Pi`) from `V`.

1. **Create Policy Table:** Create a new table `Pi` (dictionary or 2D array) to store the best action for each state.

2. **One-Step Lookahead:** For each state `s`:

   Calculate the expected value (`Q(s,a)`) for all four actions, *just like you did in Task 2*.

   Find the action `a` that gives the **maximum** `Q(s,a)` value.

   Store this best action (e.g., `'up'`) in `Pi[s]`.

3. **Return:** Return the final `Pi` table. This `Pi` is the optimal policy!

Task 4: Visualization and Analysis (15 Marks)

1. **Visualize Value Function:** Write a simple function to print your `V` table in a 3x4 grid format. Use `seaborn.heatmap` for a much better visualization.
2. **Visualize Policy:** Write a simple function to print your `Pi` table in a 3x4 grid, using arrows (`^`, `v`, `<`, `>`) to represent the actions.
3. **Analyze:**

   Question 1: Run your full pipeline. Print the final `V` table and the final `Pi` table. Does the policy make sense? Does it correctly avoid the pit and find the goal?

   Question 2: Change the "living penalty" `R(s)` from `0.04` to `0.0`. Rerun. Does the policy change? Why or why not?

   Question 3: Change the "living penalty" `R(s)` from `0.04` to `0.5` (a high penalty). Rerun. What happens to the policy? Does the agent take a different path? Why?

5. Submission Guidelines

Submit a single `.zip` archive containing:

1. **Source Code:** A single Jupyter Notebook (`.ipynb`) containing all your code, outputs, and visualizations.
2. **PDF Report:** A formal report (`StudentID_Report.pdf`) that includes:

   Code Snippets: Your implementation of the `value_iteration` loop (from Task 2) and your policy_extraction function (from Task 3).

Final Results: The final visualized **Value Table** (as a heatmap) and **Policy Table** (as arrows) for the default parameters (R=-0.04).

Analysis: Your written answers to the three analysis questions in Task 4, explaining the changes in the agent's behavior.