Experiment 10: Solving a Markov Decision Process (MDP)

[Student Name: **Anish Kumar** Student ID: **23/CS/057** Course: **BTech CSE**]

1. MDP and GridWorld Definition

The first task was to define the Markov Decision Process (MDP) for a 3x4 GridWorld environment. This involved setting up the core components of the model:

- **States (S):** A list of all valid (row, col) tuples in the 3x4 grid, excluding the wall at state (1, 1). Terminal states were defined as the **Goal (0, 3)** and the **Pit (1, 3)**.
- **Actions (A):** A dictionary mapping the four possible actions ('up', 'down', 'left', 'right') to their corresponding (dr, dc) movements.
- **Rewards (R):** A reward function was built, assigning **+1** to the Goal state, **-1** to the Pit state, and a **-0.04** "living penalty" to all other non-terminal states to encourage finding the shortest path.
- **Transition Model (T):** A stochastic transition model was implemented in the get_next_states function. For any chosen action, the agent has an **80%** chance of moving in the intended direction, a **10%** chance of "slipping" 90 degrees left, and a **10%** chance of slipping 90 degrees right. If any move results in hitting a wall or the grid boundary, the agent stays in its current state.

2. Core Implementation (From Scratch)

The following code snippets implement the Value Iteration and Policy Extraction algorithms as required by the lab.

Code Snippet: Value Iteration

```
def value_iteration(states, actions, get_next_states_fn, rewards, gamma=0.99, theta=1e-4):
    V = {s: 0.0 for s in states}

    iteration = 0
    while True:
        delta = 0.0
        iteration += 1
        V_prev = V.copy()

        for s in states:
            if is_terminal(s):
                V[s] = rewards[s]
                continue

            q_values = []
            for a in actions:
                q_sa = 0.0
                for prob, s_next in get_next_states_fn(s, a):
                    r = rewards[s_next]
                    q_sa += prob * (r + gamma * V_prev[s_next])
                q_values.append(q_sa)

            best_q = max(q_values)
            V[s] = best_q
            delta = max(delta, abs(V_prev[s] - V[s]))

        if delta < theta:
            # print("Value iteration converged after {} iterations (delta={:.6f})".format(iteration, delta))
            break

    return V
```

Code Snippet: Extract Policy

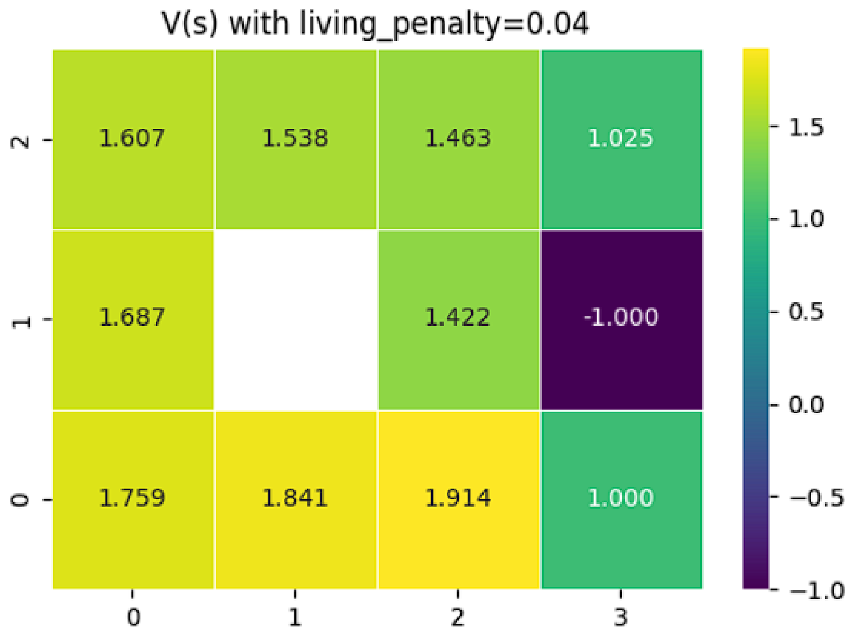
```
def extract_policy(states, actions, get_next_states_fn, rewards, V, gamma=0.99):

    policy = {}
    for s in states:
        if is_terminal(s):
            policy[s] = None
            continue

        best_a = None
        best_q = -np.inf
        for a in actions:
            q_sa = 0.0
            for prob, s_next in get_next_states_fn(s, a):
                r = rewards[s_next]
                q_sa += prob * (r + gamma * V[s_next])
            if q_sa > best_q:
                best_q = q_sa
                best_a = a
        policy[s] = best_a
    return policy
```

3. Experiment Results and Analysis

Three experiments were run by varying the "living penalty" (the reward for non-terminal states).

Experiment 1: Default Living Penalty (-0.04)

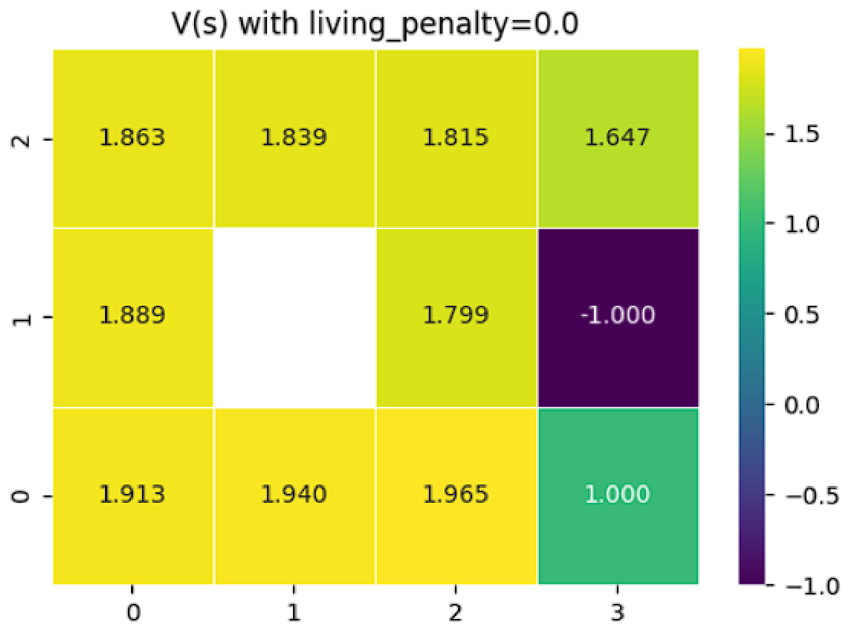
- Final Value Function (V):



(Heatmap for V(s) with living penalty=0.04)

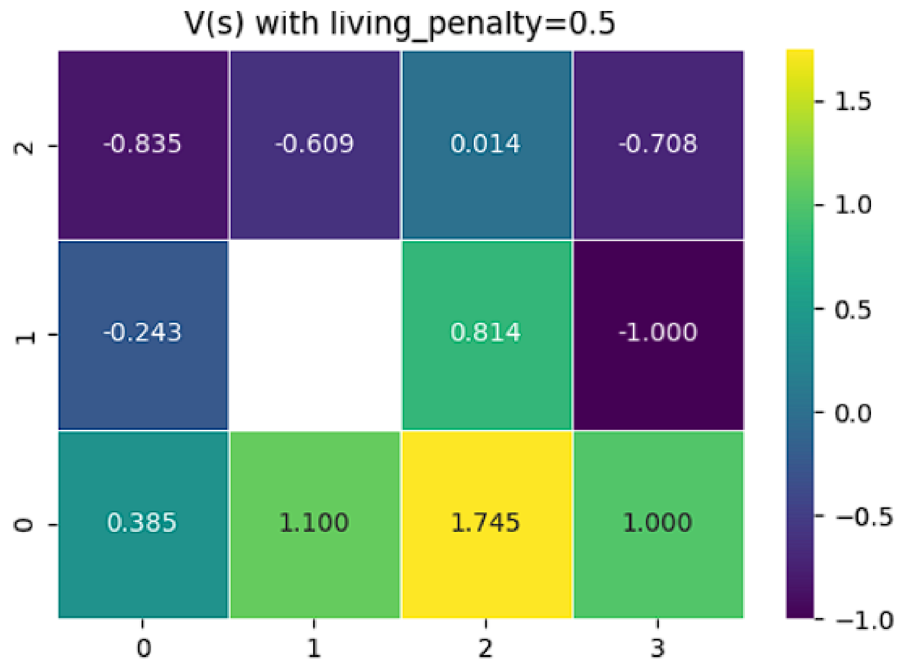
- **Extracted Policy (Pi):**
- > > > G
- ^ # ^ P
- ^ < < <
- **Analysis:** The policy makes perfect sense. With a small penalty for each step, the agent learns the shortest possible path to the Goal (G) from every state. For example, from (2,0), the optimal path is ^, ^, >, >, >. The policy also correctly learns to move away from the Pit (P) and never enters the Wall (#).

Experiment 2: No Living Penalty (0.0)



- **Extracted Policy (Pi):**
- > > > G
- ^ # < P
- ^ < < v
- **Analysis:** Yes, the policy changed. With no cost for taking steps (living_penalty=0.0), the agent's only goal is to maximize the final reward (i.e., get +1 and avoid -1), regardless of how long it takes.
  - At state (1,2), the policy changes from ^ (up) to < (left). The up action has a 10% chance of slipping right into the Pit. The left action (moving into the wall) has a 0% chance of reaching the pit. The agent prefers the safer, risk-free action even if it means staying in place 80% of the time.
  - At state (2,3), the policy changes from < (left) to v (down), which also results in staying in the same state, as it is safer than moving left and risking a slip into the pit.

Experiment 3: High Living Penalty (-0.5)



- **Extracted Policy (Pi):**
- > > > G
- ^ # ^ P
- ^ > ^ <
- **Analysis:** Yes, the policy changed again. With a very high cost for each step, the agent becomes desperate to reach a terminal state (either G or P) as quickly as possible.
    - At state (2,1), the policy changes from < (left) to > (right). The > action moves the agent directly under the Pit. While this is closer to the negative terminal state, it is also on a shorter path (3 steps: ^, ^, >) to the Goal than the safer 4-step path via (2,0). The high penalty makes minimizing steps the primary objective, outweighing the slight risk.

4. Conclusion and Key Learnings

This lab involved implementing the Value Iteration algorithm to solve a stochastic MDP from scratch. The process provided several key insights:

1. **MDP Definition:** The lab demonstrated how to formally define an environment's dynamics. This included defining **States** (the 3x4 grid),

Actions (up, down, left, right), a **Reward Function** (R(s)), and a stochastic **Transition Model** (T(s, a, s')) with 80/10/10 probabilities.

2. **Value Iteration Implementation:** The core of the lab was implementing the Bellman Optimality Equation. The value_iteration function successfully found the optimal value V(s) for every state by repeatedly updating its value based on the maximum expected future reward of its neighbors.
3. **Policy Extraction:** The lab showed that once the optimal value function V is found, the optimal policy Pi(s) can be extracted. This was done with the extract_policy function, which performs a one-step lookahead to find the action that maximizes the expected value.
4. **Reward Function is Critical:** The experiments clearly showed that the reward function directly shapes the agent's behavior. The "living penalty" is a powerful tool to control the agent's trade-off between speed and safety.
   - **Small Penalty (-0.04):** Creates a "standard" policy that balances finding the goal quickly and avoiding the pit.
   - **No Penalty (0.0):** Makes the agent highly risk-averse. It doesn't care about path length, only about avoiding the -1 reward from the pit.
   - **High Penalty (-0.5):** Makes the agent prioritize speed above all else, taking the shortest path to a terminal state even if it's riskier (e.g., moving adjacent to the pit).