I SYED AZIM
3122 21 3002 110
ECE – B

VALUE ITERATION

Exp No: 1

Date: 18.09.2024

AIM:

To implement the dynamic Value Iteration programming algorithm to find the optimal policy and value function for a Markov Decision Process (MDP) in python.

ALGORITHM:

Input:

- A set of states S.
- A set of actions A.
- Transition probabilities P(s'|s,a) — the probability of reaching state s' from state s by taking action a.
- Reward function R(s,a) — the reward obtained by taking action a in state s.
- Discount factor γ (gamma) — used to weigh the importance of future rewards.
- Convergence threshold θ (theta) — used to determine when to stop iterations.

Output:

- Optimal value function V — the maximum expected reward for each state.
- Optimal policy π — the best action to take in each state.

Steps:

1. **Initialization:**
    - Initialize the value function V(s) for all states to 0.
    - Set a convergence threshold θ.
2. **Iterative Update:**
    - Repeat until the value function V converges:
        - Set Δ = 0, which tracks the maximum change in the value function across all states.
        - For each state s:
            - Store the old value v = V(s).

        - For each action a, calculate the action-value Q(s,a) by summing over all possible next states s': Q(s,a)=∑s′P(s′|s,a)×[R(s,a)+γ×V(s′)]Q(s,a) = \sum_{s'} P(s'|s,a) \times \left[R(s,a) + \gamma \times V(s') \right]Q(s,a)=s′∑P(s′|s,a)×[R(s,a)+γ×V(s′)]
        - Update the value function V(s) with the maximum action-value: V(s)=max⁡aQ(s,a)V(s) = \max_a Q(s,a)V(s)=amaxQ(s,a)
        - Compute the difference between the new value and the old value: Δ=max⁡(Δ,|v−V(s)|)Δ = \max(Δ, |v - V(s)|) Δ=max(Δ,|v−V(s)|)
    - If the maximum change Δ is less than the threshold θ, exit the loop, as the values have converged.

3. **Policy Extraction:**
    - For each state s, compute the action-value Q(s,a) for each action a and derive the optimal action: π(s)=arg⁡max⁡aQ(s,a)π(s) = \arg\max_a Q(s,a)π(s)=argamaxQ(s,a)
    - The optimal policy π(s) assigns the best action for each state.
4. **Return Results:**
    - Return the final value function V and the optimal policy π.

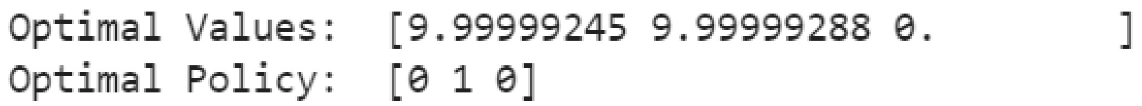
SOURCE CODE:

```
import numpy as np


def value_iteration(states, actions, transition_prob, rewards, gamma=0.9, theta=1e-6):
    """ Performs Value Iteration for a Markov Decision Process (MDP)
    :param states: List of states
    :param actions: List of actions
    :param transition_prob: Transition probability (P(s'|s,a)) 3D array of shape (states, actions, states)
    :param rewards: Rewards as a 2D array of shape (states, actions)
    :param gamma: Discount factor
    :param theta: Convergence threshold
    :return: Optimal values (V) and optimal policy (pi)  """

    # Initialize value function for each state
    V = np.zeros(len(states))
```

```
while True:
    delta = 0
    # Update each state's value
    for s in range(len(states)):
        v = V[s]
        Q_sa = np.zeros(len(actions))

        # Compute Q-value for each action
        for a in range(len(actions)):
            Q_sa[a] = sum([transition_prob[s, a, s_prime] * (rewards[s, a] + gamma * V[s_prime])
                           for s_prime in range(len(states))])

        # Update the value of the state with the best Q-value
        V[s] = max(Q_sa)
        delta = max(delta, abs(v - V[s]))

    # If the values converge, stop the iteration
    if delta < theta:
        break

# Derive the optimal policy
policy = np.zeros(len(states), dtype=int)
for s in range(len(states)):
    Q_sa = np.zeros(len(actions))
    for a in range(len(actions)):
        Q_sa[a] = sum([transition_prob[s, a, s_prime] * (rewards[s, a] + gamma * V[s_prime])
                       for s_prime in range(len(states))])
    policy[s] = np.argmax(Q_sa)
return V, policy
```

```
# Example usage:
states = [0, 1, 2]
actions = [0, 1]

# Transition probabilities (P(s'|s,a)): for each (state, action, next_state)
transition_prob = np.array([
    [[0.8, 0.2, 0], [0.1, 0.9, 0]],  # Transitions from state 0
    [[0.5, 0.5, 0], [0.2, 0.8, 0]],  # Transitions from state 1
    [[0.0, 0.0, 1], [0.0, 0.0, 1]]   # Transitions from state 2])

# Rewards for each (state, action)
rewards = np.array([
    [1, -1],  # Rewards for state 0
    [-1, 1],  # Rewards for state 1
    [0, 0]    # Rewards for state 2])

# Perform value iteration
V, policy = value_iteration(states, actions, transition_prob, rewards)
print("Optimal Values: ", V)
print("Optimal Policy: ", policy)
```

OUTPUT:



```
Optimal Values:  [9.99999245 9.99999288 0.        ]
Optimal Policy:  [0 1 0]
```

RESULT:

Thus we have successfully implemented the dynamic Value Iteration programming algorithm to find the optimal policy and value function for a Markov Decision Process (MDP) in python.

I SYED AZIM
3122 21 3002 110
ECE – B

VALUE ITERATION FOR BRIDGE GRID ENVIRONMENT

Exp No: 2a

Date: 18.09.2024

AIM:

To implement the solution for One Grid Environment (OGE) using value iteration in python.

ALGORITHM:

Inputs:

- **Grid Size**: (rows,columns)
- **Bridge Location**: Starting point of the bridge (row,col)
- **Bridge Length**: Length of the bridge
- **Goal State**: Target state (row,col)
- **Fall Penalty**: Reward for falling off the bridge (negative value)
- **Goal Reward**: Reward for reaching the goal (positive value)
- **Discount Factor**: γ ($0 < \gamma < 1$)
- **Convergence Threshold**: θ (small positive value)

Outputs:

- **Optimal Value Function**: V for each state
- **Optimal Policy**: π for each state

Steps:

1. **Initialize Environment**:
    - Create transition probability matrix P of shape (nstates,nactions,nstates).
    - Create reward matrix R of shape (nstates,nactions).
    - Set all state values V(s)=0.

2. **Build Environment**:
    - For each state (row,col):
        - For each action a:
            - If the current state is the goal state, set reward and transition.
            - Determine the next state based on the action:
                - If moving off the bridge, apply fall penalty.
                - Update the transition probabilities and rewards accordingly.

3. **Value Iteration Loop**:
    - Repeat until convergence:
        - Set $\delta=0$ (maximum change in value).
        - For each state s:
            - Store the current value: $v=V(s)$.
            - Calculate Q-values for all actions a: $Q(s,a)=s'\sum P(s,a,s')\cdot(R(s,a)+\gamma\cdot V(s'))$
            - Update the value of state s: $V(s)=a\max Q(s,a)$
            - Update δ: $\delta=\max(\delta,|v-V(s)|)$
        - If $\delta<\theta$, stop the iteration.
4. **Extract Optimal Policy**:
    - For each state s:
        - Calculate Q-values again for all actions.
        - Set policy: $\pi(s)=\text{arg}a\max Q(s,a)$
5. **Return**:
    - The optimal value function V and the optimal policy π.

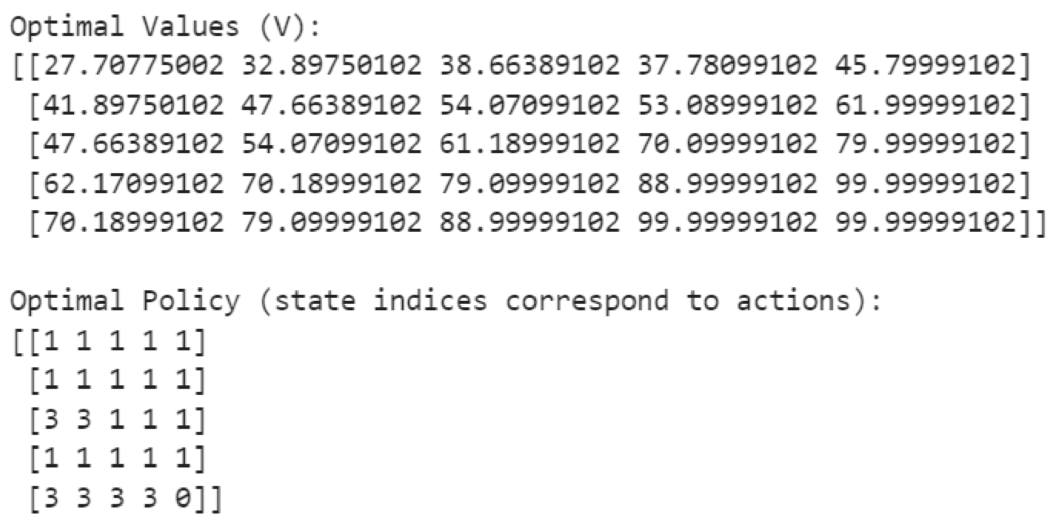
SOURCE CODE:

```
import numpy as np

class BridgeGridEnv:
    def __init__(self, grid_size=(5, 5), bridge_location=(2, 0), bridge_length=3, goal_state=(4, 4),
fall_penalty=-10, goal_reward=10):
        self.grid_size = grid_size
        self.bridge_location = bridge_location
        self.bridge_length = bridge_length
        self.goal_state = goal_state
        self.fall_penalty = fall_penalty
        self.goal_reward = goal_reward
        self.actions = ['up', 'down', 'left', 'right']
        self.n_states = grid_size[0] * grid_size[1]
        self.n_actions = len(self.actions)

        self.transitions, self.rewards = self.build_environment()

    def build_environment(self):
        """Build transition and reward matrices for the environment."""
        transition_prob = np.zeros((self.n_states, self.n_actions, self.n_states))
        rewards = np.full((self.n_states, self.n_actions), -1)  # Default reward for all actions is -1 (to
encourage faster completion)

        def state_to_index(row, col):
            return row * self.grid_size[1] + col

        def is_off_bridge(row, col):
            return (row != self.bridge_location[0] or col >= self.bridge_location[1] + self.bridge_length)
and row < self.grid_size[0] - 1

        for row in range(self.grid_size[0]):
            for col in range(self.grid_size[1]):
                state = state_to_index(row, col)
                for action_idx, action in enumerate(self.actions):
                    if (row, col) == self.goal_state:
                        rewards[state, action_idx] = self.goal_reward
                        transition_prob[state, action_idx, state] = 1.0
                        continue

                    next_row, next_col = row, col
                    if action == 'up' and row > 0:
                        next_row = row - 1
                    elif action == 'down' and row < self.grid_size[0] - 1:
```

```
                next_row = row + 1
            elif action == 'left' and col > 0:
                next_col = col - 1
            elif action == 'right' and col < self.grid_size[1] - 1:
                next_col = col + 1

            next_state = state_to_index(next_row, next_col)

            if (next_row, next_col) == self.goal_state:
                rewards[state, action_idx] = self.goal_reward
            elif is_off_bridge(next_row, next_col):
                rewards[state, action_idx] = self.fall_penalty

            transition_prob[state, action_idx, next_state] = 1.0

        return transition_prob, rewards

    def state_index_to_coordinates(self, state_index):
        """Convert state index back to row, col coordinates."""
        row = state_index // self.grid_size[1]
        col = state_index % self.grid_size[1]
        return row, col

def value_iteration_bridge(env, gamma=0.9, theta=1e-6):
    """   Perform value iteration to solve the Bridge Grid environment.
    """
    V = np.zeros(env.n_states)

    while True:
        delta = 0
        for s in range(env.n_states):
            v = V[s]
            Q_sa = np.zeros(env.n_actions)

            # Calculate Q-values for all actions in state s
            for a in range(env.n_actions):
                Q_sa[a] = sum([env.transitions[s, a, s_prime] * (env.rewards[s, a] + gamma * V[s_prime])
                               for s_prime in range(env.n_states)])

            # Update state value with the maximum Q-value
            V[s] = max(Q_sa)
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    # Derive policy from optimal state values
    policy = np.zeros(env.n_states, dtype=int)
    for s in range(env.n_states):
        Q_sa = np.zeros(env.n_actions)
        for a in range(env.n_actions):
            Q_sa[a] = sum([env.transitions[s, a, s_prime] * (env.rewards[s, a] + gamma * V[s_prime])
                           for s_prime in range(env.n_states)])
        policy[s] = np.argmax(Q_sa)

    return V, policy

# Create the bridge grid environment
env = BridgeGridEnv()
```

```
# Run value iteration
V, policy = value_iteration_bridge(env)

# Display results
print("Optimal Values (V):")
print(V.reshape(env.grid_size))

print("\nOptimal Policy (state indices correspond to actions):")
print(policy.reshape(env.grid_size))
```

OUTPUT:



```
Optimal Values (V):
[[27.70775002 32.89750102 38.66389102 37.78099102 45.79999102]
 [41.89750102 47.66389102 54.07099102 53.08999102 61.99999102]
 [47.66389102 54.07099102 61.18999102 70.09999102 79.99999102]
 [62.17099102 70.18999102 79.09999102 88.99999102 99.99999102]
 [70.18999102 79.09999102 88.99999102 99.99999102 99.99999102]]

Optimal Policy (state indices correspond to actions):
[[1 1 1 1 1]
 [1 1 1 1 1]
 [3 3 1 1 1]
 [1 1 1 1 1]
 [3 3 3 3 0]]
```

RESULT:

Thus, the solution for One Grid Environment (OGE) using value iteration is implemented in Python.

I SYED AZIM
3122 21 3002 110
ECE – B

VALUE ITERATION FOR TWO GRID ENVIRONMENT

Exp No: 2b

Date: 18.09.2024

AIM:

To implement the solution for Two Grid Environment (TGE) using value iteration in Python.

ALGORITHM:

1. Initialize Environment (BridgeGrid):

- Define a 5x5 grid.
- Set the goal position at (0, 4) (top-right corner).
- Set the start position at (4, 0) (bottom-left corner).
- Define a bridge of safe cells at positions (2, 1), (2, 2), (2, 3) on the middle row.
- Define possible actions as ['up', 'down', 'left', 'right'].
- Implement the reset, step, and render functions for the environment.

2. Environment Dynamics (step function):

- **Actions**: Move the agent based on the chosen action (up, down, left, right) with boundary checks.
- **Rewards**:
    - Reward of +1 for reaching the goal.
    - Reward of 0 for being on the bridge.
    - Penalty of -1 for falling off the bridge into the water (outside the bridge in the middle row).
    - Small penalty of -0.1 for regular moves to encourage faster learning.
- Update the agent's state after the action.

3. Initialize Q-Learning Agent (QLearningAgent):

- Set learning rate $\alpha = 0.1$, discount factor $\gamma = 0.9$, and exploration rate $\varepsilon = 0.1$.
- Use a Q-table (implemented as a dictionary) to store Q-values for state-action pairs.

4. Q-Learning Algorithm:

- **Q-value update formula**: Q(s,a)←Q(s,a)+α[r+γmax⁡aQ(s′,a)−Q(s,a)]Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_a Q(s', a) - Q(s, a) \right]Q(s,a)←Q(s,a)+α[r+γamax Q(s′,a)−Q(s,a)]
- For each state-action pair, update the Q-value using the above formula, where:
  - s is the current state.
  - a is the current action.
  - r is the reward received.
  - s' is the next state after taking action a.
  - α is the learning rate, and γ is the discount factor.

5. Choose Action (choose_action function):

- With probability ε, choose a random action (exploration).
- With probability 1 - ε, choose the action that has the highest Q-value for the current state (exploitation).
- In case of ties (multiple actions with the same Q-value), randomly choose one of the tied actions.

6. Training Loop (train function):

- For each episode:
  . **Reset** the environment to the initial state.
  . Loop until the episode terminates (done = True):
    - Choose an action using the ε-greedy strategy.
    - Take the action and observe the next state and reward.
    - Update the Q-value for the state-action pair.
    - Set the next state as the current state.

  . If the episode is a multiple of 100, print progress.
- Repeat for the specified number of episodes (e.g., 1000).

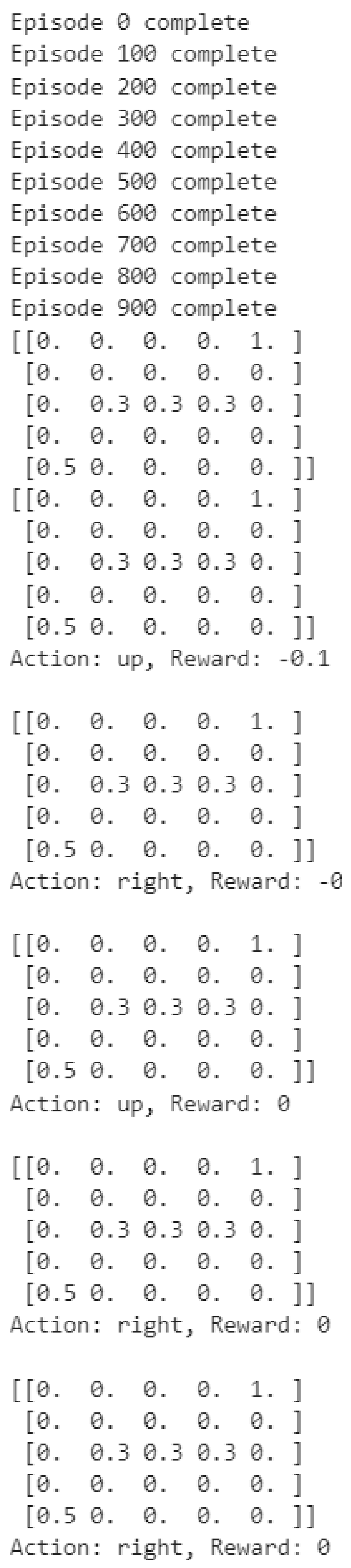
7. Testing (after training):

- Reset the environment to the starting state.
- Choose actions based on the learned Q-values (exploitation).
- Render the grid after each step to visualize the agent's movement.
- Output the action taken and the corresponding reward after each step.

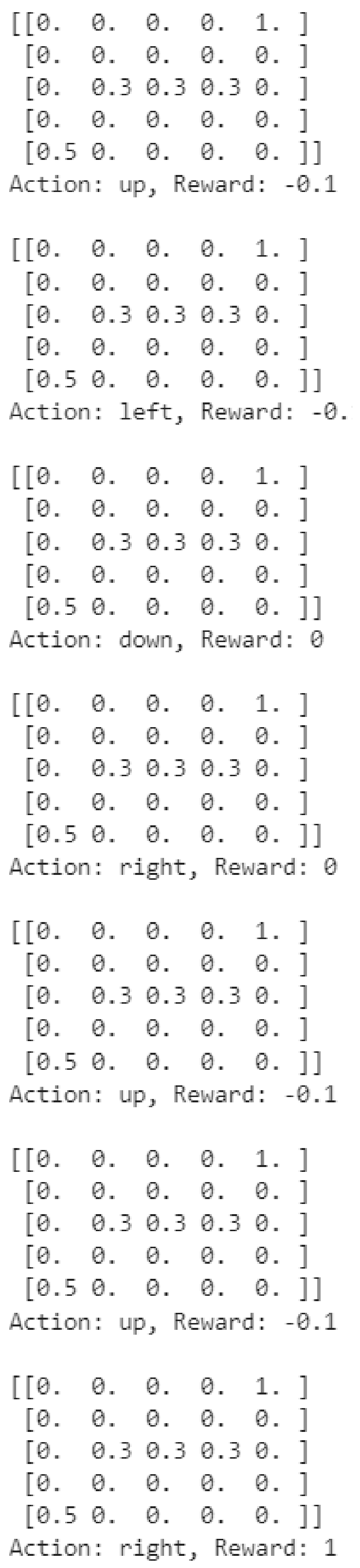
SOURCE CODE:

```
import numpy as np
import random


# Environment Definition: Bridge Grid
class BridgeGrid:
    def __init__(self):
        self.grid = np.zeros((5, 5))
        self.goal = (0, 4)  # Goal is at the top right corner
        self.start = (4, 0)  # Start is at the bottom left corner
        self.bridge = [(2, 1), (2, 2), (2, 3)]  # Narrow bridge in the middle row
        self.state = self.start
        self.actions = ['up', 'down', 'left', 'right']

    def reset(self):
        self.state = self.start
        return self.state

    def step(self, action):
        row, col = self.state

        if action == 'up':
            row = max(0, row - 1)
        elif action == 'down':
            row = min(4, row + 1)
        elif action == 'left':
            col = max(0, col - 1)
        elif action == 'right':
            col = min(4, col + 1)
        next_state = (row, col)

        # Define reward
        if next_state == self.goal:
            reward = 1  # Reward for reaching the goal
            done = True
        elif next_state in self.bridge:
            reward = 0  # Reward for being on the bridge
            done = False
        elif row == 2 and col not in [1, 2, 3]:
            reward = -1  # Falling off the bridge (penalty)
            done = True
        else:
            reward = -0.1  # Small penalty for each move to encourage faster learning
            done = False

        self.state = next_state
        return next_state, reward, done

    def render(self):
        grid_copy = np.copy(self.grid)
        grid_copy[self.goal] = 1
        grid_copy[self.start] = 0.5
        for bridge_part in self.bridge:
            grid_copy[bridge_part] = 0.3
        print(grid_copy)

# Q-Learning Agent
class QLearningAgent:
    def __init__(self, env, alpha=0.1, gamma=0.9, epsilon=0.1):
        self.env = env
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.epsilon = epsilon  # Exploration rate
        self.q_table = {}  # Q-values dictionary

    def get_q_value(self, state, action):
        return self.q_table.get((state, action), 0.0)

    def update_q_value(self, state, action, reward, next_state):
        max_next_q_value = max([self.get_q_value(next_state, a) for a in self.env.actions])
        current_q_value = self.get_q_value(state, action)
        new_q_value = current_q_value + self.alpha * (reward + self.gamma * max_next_q_value - 
current_q_value)
        self.q_table[(state, action)] = new_q_value

    def choose_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(self.env.actions)  # Explore
        else:
            # Exploit: choose action with the highest Q-value
            q_values = [self.get_q_value(state, action) for action in self.env.actions]
            max_q = max(q_values)
            max_actions = [action for action, q_value in zip(self.env.actions, q_values) if q_value == 
max_q]
            return random.choice(max_actions)

    def train(self, episodes=1000):
        for episode in range(episodes):
            state = self.env.reset()
        done = False
        while not done:
            action = self.choose_action(state)
            next_state, reward, done = self.env.step(action)
            self.update_q_value(state, action, reward, next_state)
            state = next_state
        if episode % 100 == 0:
            print(f"Episode {episode} complete")

# Create the environment
env = BridgeGrid()

# Create the Q-Learning agent
agent = QLearningAgent(env)

# Train the agent
agent.train(episodes=1000)

# Test the agent
state = env.reset()
env.render()

done = False
while not done:
    action = agent.choose_action(state)
    next_state, reward, done = env.step(action)
    state = next_state
    env.render()
    print(f"Action: {action}, Reward: {reward}\n")
```

OUTPUT:



```
Episode 0 complete
Episode 100 complete
Episode 200 complete
Episode 300 complete
Episode 400 complete
Episode 500 complete
Episode 600 complete
Episode 700 complete
Episode 800 complete
Episode 900 complete
[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: up, Reward: -0.1

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: right, Reward: -0

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: up, Reward: 0

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: right, Reward: 0

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: right, Reward: 0
```



```
[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: up, Reward: -0.1

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: left, Reward: -0.

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: down, Reward: 0

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: right, Reward: 0

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: up, Reward: -0.1

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: up, Reward: -0.1

[[0.  0.  0.  0.  1. ]
 [0.  0.  0.  0.  0. ]
 [0.  0.3 0.3 0.3 0. ]
 [0.  0.  0.  0.  0. ]
 [0.5 0.  0.  0.  0. ]]
Action: right, Reward: 1
```

RESULT:

Thus we implement the solution for Two Grid Environment (TGE) using value iteration in Python.

I SYED AZIM
3122 21 3002 110
ECE – B

VALUE ITERATION FOR DISCOUNT GRID ENVIRONMENT

Exp No: 3

Date: 18.09.2024

AIM:

To implement the solution for Discount Grid Environment using value iteration in python.

ALGORITHM

1. Initialize the Discount Grid Environment (DiscountGridEnv):

- Create a grid environment with size (rows, cols) (default is 4x4).
- Set a goal state and assign a high reward for reaching the goal.
- Set penalty for each step (negative reward), and discount factor gamma for future rewards.
- Specify obstacle states where the agent cannot move.
- Define possible actions: up, down, left, right.

2. Define Environment Dynamics (build_environment):

- Build transition probabilities and reward structures for each state-action pair.
- For each state (row, col):
  - If it is the goal state, set the reward and transition to itself.
  - For other states, calculate the next state based on the action.
  - Ensure the agent doesn't move into obstacles or out of bounds.
- Return the transition probability matrix P(s, a, s') and reward function R(s, a).

3. Value Iteration Algorithm (value_iteration_discount_grid):

- **Initialize** the value function V(s) for all states as 0.
- **Convergence criteria**: Use a small threshold theta to determine when to stop.

4. Iteration Steps:

- For each state s, calculate the Q-value for all actions a:
  Q(s,a)=∑s′P(s,a,s′)×(R(s,a)+γ×V(s′))Q(s, a) = \sum_{s'} P(s, a, s') \times \left(R(s, a) + \gamma \times V(s') \right)Q(s,a)=s′∑P(s,a,s′)×(R(s,a)+γ×V(s′))
- Update the value V(s) of state s as the maximum Q-value across all actions:
  V(s)←max⁡aQ(s,a)V(s) \leftarrow \max_{a} Q(s, a)V(s)←amaxQ(s,a)

- Track the maximum difference between the old and new value functions (delta) to check convergence.
- If delta is smaller than the threshold theta, stop the iteration.

5. Derive Optimal Policy:

- After convergence of the value function, derive the optimal policy:
    - For each state s, select the action that maximizes the Q-value:
      π(s)=argmaxaQ(s,a)\pi(s) = \arg\max_{a} Q(s, a)π(s)=argamaxQ(s,a)

6. Return the Optimal Value Function and Policy:

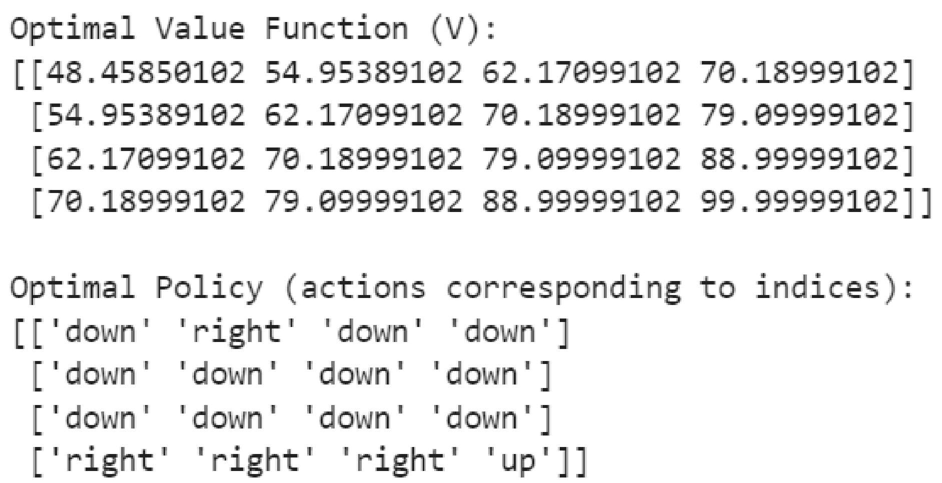
- Output the optimal value function V(s) and the optimal policy π(s).

SOURCE CODE:

```
import numpy as np


class DiscountGridEnv:
    def __init__(self, grid_size=(4, 4), goal_state=(3, 3), discount_factor=0.9, obstacle_states=[], goal_reward=10, step_penalty=-1):
        """
        Initializes the Discount Grid Environment.
        :param grid_size: The size of the grid (rows, cols)
        :param goal_state: The coordinates of the goal state
        :param discount_factor: Discount factor (gamma) for future rewards
        :param obstacle_states: List of obstacle coordinates (rows, cols) that the agent cannot pass through
        :param goal_reward: Reward for reaching the goal
        :param step_penalty: Penalty for taking a step (negative reward)
        """
        self.grid_size = grid_size
        self.goal_state = goal_state
        self.discount_factor = discount_factor
        self.obstacle_states = obstacle_states
        self.goal_reward = goal_reward
        self.step_penalty = step_penalty
        self.actions = ['up', 'down', 'left', 'right']
        self.n_states = grid_size[0] * grid_size[1]
        self.n_actions = len(self.actions)
        self.transitions, self.rewards = self.build_environment()

    def build_environment(self):
        """Build transition probabilities and rewards for each state-action pair.
        """
        transition_prob = np.zeros((self.n_states, self.n_actions, self.n_states))
        rewards = np.full((self.n_states, self.n_actions), self.step_penalty)  # Default penalty for every
step

        def state_to_index(row, col):
            return row * self.grid_size[1] + col

        def is_valid_state(row, col):
            return (0 <= row < self.grid_size[0]) and (0 <= col < self.grid_size[1]) and (row, col) not in
self.obstacle_states

        for row in range(self.grid_size[0]):
            for col in range(self.grid_size[1]):
                state = state_to_index(row, col)

                for action_idx, action in enumerate(self.actions):
                    if (row, col) == self.goal_state:
                        rewards[state, action_idx] = self.goal_reward
                        transition_prob[state, action_idx, state] = 1.0
                        continue

                    next_row, next_col = row, col
                    if action == 'up' and is_valid_state(row - 1, col):
                        next_row = row - 1
                    elif action == 'down' and is_valid_state(row + 1, col):
                        next_row = row + 1
                    elif action == 'left' and is_valid_state(row, col - 1):
                        next_col = col - 1
                    elif action == 'right' and is_valid_state(row, col + 1):
                        next_col = col + 1

                    next_state = state_to_index(next_row, next_col)

                    transition_prob[state, action_idx, next_state] = 1.0

        return transition_prob, rewards

    def state_index_to_coordinates(self, state_index):
        """Convert state index back to (row, col) coordinates."""
        row = state_index // self.grid_size[1]
        col = state_index % self.grid_size[1]
        return row, col

def value_iteration_discount_grid(env, gamma=0.9, theta=1e-6):
    "" Perform Value Iteration for the Discount Grid environment.
    :param env: The DiscountGridEnv environment
    :param gamma: Discount factor
    :param theta: Convergence threshold for value iteration
    :return: Optimal value function (V) and optimal policy
    """
    V = np.zeros(env.n_states)
    while True:
        delta = 0
        for s in range(env.n_states):
            v = V[s]
            Q_sa = np.zeros(env.n_actions)

            # Compute Q-values for all actions in state s
            for a in range(env.n_actions):
                Q_sa[a] = sum([env.transitions[s, a, s_prime] * (env.rewards[s, a] + gamma * V[s_prime])
                               for s_prime in range(env.n_states)])

            # Update the value of state s with the maximum Q-value
            V[s] = max(Q_sa)
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    # Derive policy from optimal state values
    policy = np.zeros(env.n_states, dtype=int)
    for s in range(env.n_states):
        Q_sa = np.zeros(env.n_actions)
        for a in range(env.n_actions):
            Q_sa[a] = sum([env.transitions[s, a, s_prime] * (env.rewards[s, a] + gamma * V[s_prime])
                           for s_prime in range(env.n_states)])
        policy[s] = np.argmax(Q_sa)
    return V, policy

# Example usage
```

```
env = DiscountGridEnv(grid_size=(4, 4), goal_state=(3, 3), discount_factor=0.9,
obstacle_states=[(1, 1)], goal_reward=10, step_penalty=-1)


# Run value iteration on the environment

V, policy = value_iteration_discount_grid(env, gamma=0.9)


# Display results

print("Optimal Value Function (V):")

print(V.reshape(env.grid_size))


print("\nOptimal Policy (actions corresponding to indices):")

policy_grid = np.array(env.actions)[policy].reshape(env.grid_size)

print(policy_grid)
```

OUTPUT:



```
Optimal Value Function (V):
[[48.45850102 54.95389102 62.17099102 70.18999102]
 [54.95389102 62.17099102 70.18999102 79.09999102]
 [62.17099102 70.18999102 79.09999102 88.99999102]
 [70.18999102 79.09999102 88.99999102 99.99999102]]

Optimal Policy (actions corresponding to indices):
[['down' 'right' 'down' 'down']
 ['down' 'down' 'down' 'down']
 ['down' 'down' 'down' 'down']
 ['right' 'right' 'right' 'up']]
```

RESULT:

Thus, we have implemented the solution for Discount Grid Environment (DGE) using value iteration in Python.

I SYED AZIM
3122 21 3002 110
ECE – B

ACTION SELECTION METHOD

Exp No: 4

Date: 18.09.2024

AIM:

To implement various action-selection strategies used in Reinforcement Learning (RL) to choose an action based on Q-values (which represent the estimated value of taking an action in a given state) for each state-action pair. To implement greedy, epsilon-greedy, softmax, and Upper Confidence Bound (UCB) strategies.

ALGORITHM

1. Greedy Action Selection

- **Input**: Q (2D array of state-action values), current state
- **Output**: Action with the highest value for the given state
- **Steps**:
    . For the current state, look up the Q-values of all possible actions.
    . Select the action that has the maximum Q-value.

2. Epsilon-Greedy Action Selection

- **Input**: Q (2D array of state-action values), current state, epsilon (exploration probability)
- **Output**: Either a random action (with probability epsilon) or the greedy action (with probability 1 - epsilon)
- **Steps**:
    . Generate a random number between 0 and 1.
    . If the number is less than epsilon, select a random action (exploration).
    . Otherwise, select the greedy action (action with the highest Q-value in the given state).

3. Softmax Action Selection

- **Input**: Q (2D array of state-action values), current state, tau (temperature parameter controlling exploration)
- **Output**: Action selected probabilistically based on Q-values
- **Steps**:

. For the current state, look up the Q-values of all possible actions.

. Apply the softmax function to convert Q-values into a probability distribution over actions.

. Select an action based on these probabilities (higher Q-values lead to higher probabilities, especially for lower tau).

4. Upper Confidence Bound (UCB) Action Selection

- **Input**: Q (2D array of state-action values), current state, action counts, total counts, exploration parameter ccc
- **Output**: Action selected based on both Q-value and exploration potential
- **Steps**:

  . For each action in the given state:

  - If the action has never been taken, choose it immediately.
  - Otherwise, calculate the UCB value for the action using the formula: UCB(a)=Q(state,a)+c×log⁡(total counts)action counts(state,a)\text{UCB}(a) = Q(\text{state}, a) + c \times \sqrt{\frac{\log(\text{total counts})}{\text{action counts}(\text{state}, a)}}UCB(a)=Q(state,a)+c×action counts(state,a)log(total counts)

  . Select the action with the highest UCB value.

SOURCE CODE:

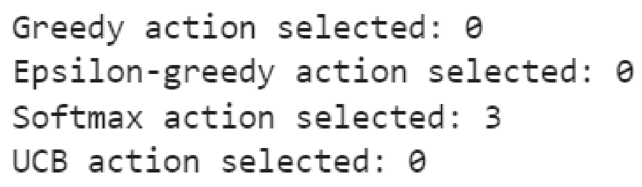
```
import numpy as np

import random


def greedy_action_selection(Q, state):
    """
    Select the action with the highest value (greedy).
    :param Q: A 2D array where Q[s, a] is the estimated value of action 'a' in state 's'
    :param state: Current state
    :return: Selected action (greedy)
    """
    return np.argmax(Q[state])
```

```
def epsilon_greedy_action_selection(Q, state, epsilon=0.1):
    """
    Select an action using the epsilon-greedy strategy.
    :param Q: A 2D array where Q[s, a] is the estimated value of action 'a' in state 's'
    :param state: Current state
    :param epsilon: Probability of selecting a random action (exploration)
    :return: Selected action
    """
    if random.uniform(0, 1) < epsilon:
        # Exploration: Choose a random action
        return random.choice(range(Q.shape[1]))
    else:
        # Exploitation: Choose the action with the highest Q-value
        return np.argmax(Q[state])

def softmax_action_selection(Q, state, tau=1.0):
    """ Select an action using the softmax strategy.
    :param Q: A 2D array where Q[s, a] is the estimated value of action 'a' in state 's'
    :param state: Current state
    :param tau: Temperature parameter controlling exploration; high tau means more exploration
    :return: Selected action
    """
    q_values = Q[state]
    # Apply the softmax transformation to the Q-values
    exp_q = np.exp(q_values / tau)
    action_probabilities = exp_q / np.sum(exp_q)

    # Choose an action based on the computed probabilities
    return np.random.choice(len(q_values), p=action_probabilities)
```

```
def ucb_action_selection(Q, state, action_counts, total_counts, c=1.0):
    """ Select an action using the Upper Confidence Bound (UCB) strategy.
    :param Q: A 2D array where Q[s, a] is the estimated value of action 'a' in state 's'
    :param state: Current state
    :param action_counts: A 2D array tracking how many times each action has been taken in each
state
    :param total_counts: Total number of actions taken so far
    :param c: Exploration parameter (higher means more exploration)
    :return: Selected action
    """
    ucb_values = np.zeros(Q.shape[1])

    for a in range(Q.shape[1]):
        if action_counts[state, a] == 0:
            return a  # If action has never been taken, choose it
        else:
            ucb_values[a] = Q[state, a] + c * np.sqrt(np.log(total_counts) / (action_counts[state, a] + 1e-
5))

    return np.argmax(ucb_values)

# Example Q-values for a simple environment with 3 states and 4 actions per state
Q = np.array([
    [1.0, 0.5, 0.2, 0.8],  # Q-values for state 0
    [0.1, 2.0, 0.3, 0.4],  # Q-values for state 1
    [0.5, 0.4, 3.0, 1.0]   # Q-values for state 2])

# Initialize action counts for UCB
action_counts = np.zeros(Q.shape)  # Keeps track of how many times each action has been chosen
total_counts = 1  # Total action selections (initially 1 to avoid division by zero)
```

```
# Current state
state = 0

# Greedy action selection
action_greedy = greedy_action_selection(Q, state)
print(f"Greedy action selected: {action_greedy}")

# Epsilon-greedy action selection
action_epsilon_greedy = epsilon_greedy_action_selection(Q, state, epsilon=0.1)
print(f"Epsilon-greedy action selected: {action_epsilon_greedy}")

# Softmax action selection
action_softmax = softmax_action_selection(Q, state, tau=1.0)
print(f"Softmax action selected: {action_softmax}")

# UCB action selection
action_ucb = ucb_action_selection(Q, state, action_counts, total_counts, c=2.0)
print(f"UCB action selected: {action_ucb}")
```

OUTPUT:



```
Greedy action selected: 0
Epsilon-greedy action selected: 0
Softmax action selected: 3
UCB action selected: 0
```

RESULT:

Thus, we have successfully implemented various action-selection strategies used in Reinforcement Learning (RL) to choose an action based on Q-values (which represent the estimated value of taking an action in a given state) for each state-action pair. We have successfully implemented greedy, epsilon-greedy, softmax, and Upper Confidence Bound (UCB) strategies.

I SYED AZIM
3122 21 3002 110
ECE – B

Q-LEARNING ALGORITHM

Exp No: 5

Date: 18.09.2024

AIM:

To implement a Q-learning algorithm in a grid world environment where an agent learns to navigate from the start to a goal while avoiding traps, using an epsilon-greedy strategy to balance exploration and exploitation which trains the agent by updating Q-values based on rewards and tests the learned policy to evaluate performance.

ALGORITHM:

1. Initialize the Environment (GridworldEnv Class):

- **Gridworld Environment Setup:**
    - Grid size: Defined by grid_size (default 4x4).
    - Goal state: Reaching this state gives a positive reward (goal_reward).
    - Trap states: Falling into these gives a penalty (trap_penalty).
    - Step penalty: Every move has a negative reward (step_penalty).
- **Agent Actions:** The agent can move up, down, left, or right.
- **Initial State:** The agent starts at the top-left corner (0, 0).
- **Step Function:** Moves the agent based on the action and checks if the agent has reached the goal, a trap, or just a normal position. Rewards and done flag are updated accordingly.

2. Q-Learning Algorithm:

- **Initialize Q-Table:**
    - Q-table is a 3D array of size grid_size[0] x grid_size[1] x n_actions. Initially, all Q-values are set to zero.
- **For each episode:**
    - **Reset the Environment:** The agent starts at the initial position (0, 0).
    - **While the episode is not done:**
        - **Epsilon-Greedy Action Selection:**
            - With probability epsilon, choose a random action (exploration).

        - With probability 1 - epsilon, choose the action with the highest Q-value for the current state (exploitation).
    - **Take Action:** Apply the chosen action, observe the next state, reward, and whether the episode has ended (goal or trap reached).
    - **Q-Value Update:**
        - Update the Q-value for the current state-action pair using the Q-learning update rule: Q[s,a]=(1−α)Q[s,a]+α(r+γmax⁡(Q[s′,a′]))Q[s, a] = (1 - \alpha) Q[s, a] + \alpha \left(r + \gamma \max(Q[s', a']) \right)Q[s,a]=(1−α)Q[s,a]+α(r+γmax(Q[s′,a′]))
        - Where:
            - Q[s,a]Q[s, a]Q[s,a] is the current Q-value for the state-action pair.
            - α\alphaα is the learning rate.
            - rrr is the observed reward.
            - γ\gammaγ is the discount factor for future rewards.
            - max⁡(Q[s′,a′])\max(Q[s', a'])max(Q[s′,a′]) is the highest Q-value for the next state s′s's′.
    - **Move to Next State:** Update the current state to the next state and repeat.
- **End of Episode:** Once the episode ends, move to the next episode and repeat the process.

3. Test the Learned Policy:

- **Using the Trained Q-Table:**
    - Start from the initial state (0, 0).
    - Always choose the action with the highest Q-value (greedy).
    - Execute the actions until the agent reaches the goal or a trap.
    - Output the steps taken, actions performed, and the total reward accumulated during the test.

SOURCE CODE:

```
import numpy as np

import random
```

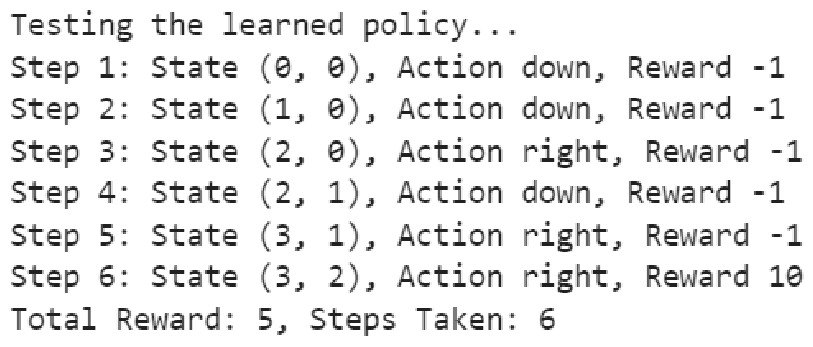
```
class GridworldEnv:
    def __init__(self, grid_size=(4, 4), goal_state=(3, 3), trap_states=[], goal_reward=10,
trap_penalty=-10, step_penalty=-1):
        """Initializes the Gridworld environment.
        :param grid_size: The size of the grid (rows, cols)
        :param goal_state: The coordinates of the goal state
        :param trap_states: List of trap coordinates (rows, cols) that give negative rewards
        :param goal_reward: Reward for reaching the goal
        :param trap_penalty: Penalty for falling into traps
        :param step_penalty: Penalty for taking a step (negative reward)
"""
        self.grid_size = grid_size
        self.goal_state = goal_state
        self.trap_states = trap_states
        self.goal_reward = goal_reward
        self.trap_penalty = trap_penalty
        self.step_penalty = step_penalty
        self.actions = ['up', 'down', 'left', 'right']
        self.n_actions = len(self.actions)

    def reset(self):
        """Reset the environment to the initial state (top-left corner)."""
        self.agent_position = (0, 0)
        return self.agent_position

    def step(self, action):
        """Take a step in the environment according to the action."""
        row, col = self.agent_position

        if action == 0:  # up
```

```
            next_position = (max(row - 1, 0), col)
        elif action == 1:  # down
            next_position = (min(row + 1, self.grid_size[0] - 1), col)
        elif action == 2:  # left
            next_position = (row, max(col - 1, 0))
        elif action == 3:  # right
            next_position = (row, min(col + 1, self.grid_size[1] - 1))

        reward = self.step_penalty
        done = False

        if next_position == self.goal_state:
            reward = self.goal_reward
            done = True
        elif next_position in self.trap_states:
            reward = self.trap_penalty
            done = True

        self.agent_position = next_position
        return next_position, reward, done

    def get_state(self):
        """Returns the current state (row, col)."""
        return self.agent_position

    def action_space(self):
        """Returns the number of available actions."""
        return self.n_actions

def q_learning(env, episodes=1000, alpha=0.1, gamma=0.9, epsilon=0.1):
```

```
"""Q-Learning algorithm implementation.
:param env: The environment
:param episodes: Number of episodes to run
:param alpha: Learning rate
:param gamma: Discount factor
:param epsilon: Exploration rate
:return: Learned Q-values
"""
# Initialize the Q-table (states: grid_size, actions: up, down, left, right)
Q = np.zeros((env.grid_size[0], env.grid_size[1], env.n_actions))

for episode in range(episodes):
    state = env.reset()
    done = False

    while not done:
        row, col = state

        # Epsilon-greedy action selection
        if random.uniform(0, 1) < epsilon:
            action = random.choice(range(env.n_actions))
        else:
            action = np.argmax(Q[row, col])

        # Take action, observe reward and next state
        next_state, reward, done = env.step(action)
        next_row, next_col = next_state

        # Update Q-value using the Q-learning update rule
        best_next_action = np.argmax(Q[next_row, next_col])
```

```
            td_target = reward + gamma * Q[next_row, next_col, best_next_action]
            Q[row, col, action] = (1 - alpha) * Q[row, col, action] + alpha * td_target

            state = next_state

    return Q


def test_q_learning(Q, env):
    """Test the learned Q-values by running the agent through the environment.
    :param Q: The learned Q-values
    :param env: The environment
    """
    state = env.reset()
    done = False
    steps = 0
    total_reward = 0

    print("Testing the learned policy...")

    while not done:
        row, col = state
        action = np.argmax(Q[row, col])  # Exploit the learned Q-values (greedy)

        next_state, reward, done = env.step(action)
        total_reward += reward
        steps += 1

        print(f"Step {steps}: State {state}, Action {env.actions[action]}, Reward {reward}")
        state = next_state
    print(f"Total Reward: {total_reward}, Steps Taken: {steps}")


# Create the Gridworld environment
env = GridworldEnv(grid_size=(4, 4), goal_state=(3, 3), trap_states=[(1, 1)], goal_reward=10, trap_penalty=-10)


# Run Q-Learning
Q = q_learning(env, episodes=1000, alpha=0.1, gamma=0.9, epsilon=0.1)


# Test the learned Q-values
test_q_learning(Q, env)
```

OUTPUT:



```
Testing the learned policy...
Step 1: State (0, 0), Action down, Reward -1
Step 2: State (1, 0), Action down, Reward -1
Step 3: State (2, 0), Action right, Reward -1
Step 4: State (2, 1), Action down, Reward -1
Step 5: State (3, 1), Action right, Reward -1
Step 6: State (3, 2), Action right, Reward 10
Total Reward: 5, Steps Taken: 6
```

RESULT:

Thus, we have successfully implemented a Q-learning algorithm in a grid world environment where an agent learns to navigate from the start to a goal while avoiding traps, using an epsilon-greedy strategy to balance exploration and exploitation which trains the agent by updating Q-values based on rewards and tests the learned policy to evaluate performance.