EXPERIMENT ASSESSMENT

ACADEMIC YEAR 2023-24

Course: Reinforcement Learning Lab

Course code: CSDOL8013

Year: BE **Sem:** VIII

Experiment No.: 3
Aim: - Implementing a basic grid-world environment as an MDP and applying policy iteration and value iteration algorithms to find optimal policies
Name:
Roll Number:
Date of Performance:
Date of Submission:

Evaluation

Performance Indicator	Max. Marks	Marks Obtained
Performance	5	
Understanding	5	
Journal work and timely submission.	10	
Total	**20**	

Performance Indicator	Exceed Expectations (EE)	Meet Expectations (ME)	Below Expectations (BE)
Performance	5	3	2
Understanding	5	3	2
Journal work and timely submission.	10	8	4

Checked by

Name of Faculty : Rujuta Vartak

Signature :

Date :

EXPERIMENT 3

Aim: Implementing a basic grid-world environment as an MDP and applying policy iteration and value iteration algorithms to find optimal policies.

Objective: The objective is to create a simplified grid-world environment, representing it as Markov Decision Process (MDP), and then employ policy iteration and value iteration algorithms to compute optimal policies within this environment, facilitating a fundamental understanding of dynamic programming techniques in reinforcement learning.

Theory:

Markov Decision Process (MDP):

- An MDP is a mathematical framework used to model decision-making in environments where outcomes are partially random and partially under the control of an agent.
- It consists of a set of states, a set of actions, transition probabilities, rewards, and a discount factor.
- In a grid-world environment, each cell represents a state, and the agent can take actions (move up, down, left, right) to transition between states.

Optimal Policies:

- An optimal policy specifies the best action to take in each state to maximize cumulative rewards over time.
- Finding optimal policies involves determining the best strategy for the agent to navigate the environment and achieve its goals.

Policy Iteration:

- Policy iteration is an iterative algorithm for finding the optimal policy in an MDP.
- It alternates between two steps: policy evaluation and policy improvement.
- Policy evaluation involves estimating the value function for a given policy, while policy improvement updates the policy based on the current value function.
- This process continues until the policy converges to an optimal policy.

Algorithm for Policy Iteration:

1. Policy Evaluation:

   - Initialize V(s) arbitrarily for all states

   - Repeat until $\Delta < \varepsilon$ (small positive number):

     - $\Delta \leftarrow 0$

- For each state s:

  - $v \leftarrow V(s)$

  - $V(s) \leftarrow \Sigma(P(s' \mid s, \pi(s)) * [R(s, \pi(s), s') + \gamma * V(s')])$

  - $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

2. Policy Improvement:

   - Policy_stable ← true

   - For each state s:

     - old_action ← $\pi(s)$

     - $\pi(s) \leftarrow \text{argmax}[a]\{\Sigma(P(s' \mid s, a) * [R(s, a, s') + \gamma * V(s')])\}$

     - If old_action ≠ $\pi(s)$, then Policy_stable ← false

   If Policy_stable, then stop and return optimal policy π^*

Value Iteration:

- Value iteration is another iterative algorithm used to find optimal policies in MDPs.
- It iteratively updates the value function for each state until it converges to the optimal value function.
- The optimal policy can then be derived from the optimal value function by selecting actions that maximize expected returns.

Algorithm of Value Iteration:

Initialize V(s) arbitrarily for all states

Repeat until $\Delta < \varepsilon$ (small positive number):

- $\Delta \leftarrow 0$

For each state s:

- $v \leftarrow V(s)$

- $V(s) \leftarrow \text{max}[a]\{\Sigma(P(s' \mid s, a) * [R(s, a, s') + \gamma * V(s')])\}$

- $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Return optimal policy π^* such that $\pi^*(s) = \text{argmax}[a]\{\Sigma(P(s' \mid s, a) * [R(s, a, s') + \gamma * V(s')])\}$

In these algorithms:

- **π** represents the policy, which specifies the action to be taken in each state.
- **V(s)** represents the value function, which estimates the expected cumulative reward starting from state **s** and following the current policy **π**.
- **P(s' | s, a)** represents the transition probability from state **s** to state **s'** under action **a**.
- **R(s, a, s')** represents the reward obtained when transitioning from state **s** to state **s'** under action **a**.
- **γ** is the discount factor, representing the importance of future rewards compared to immediate rewards.
- **Δ** is a small positive number used to determine convergence.

These algorithms iteratively update the value function and policy until convergence, where the policy either stabilizes (in policy iteration) or the value function converges (in value iteration). The resulting policy is then considered optimal for the given grid-world environment.

Code:

```
# Initialize environment
row = 3
col = 4

# Non-terminal state rewards
α = -0.01
γ = 0.99

# Action sequneces Down, Left, Up, Right
A = [(1, 0), (0, -1), (-1, 0), (0, 1)]
# Acton steps
Steps = 4

# Error Rate
Max_Err = 10**(-3)

# Define untility control
U = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]

# Construct a random policy
rand_policy = [[random.randint(0, 3) for j in range(col)] for i in
range(row)]

# Get the utility of the state reached by performing the given action from
the given state
def utility(U, r, c, action):
    dr, dc = A[action]
    newR, newC = r+dr, c+dc
```

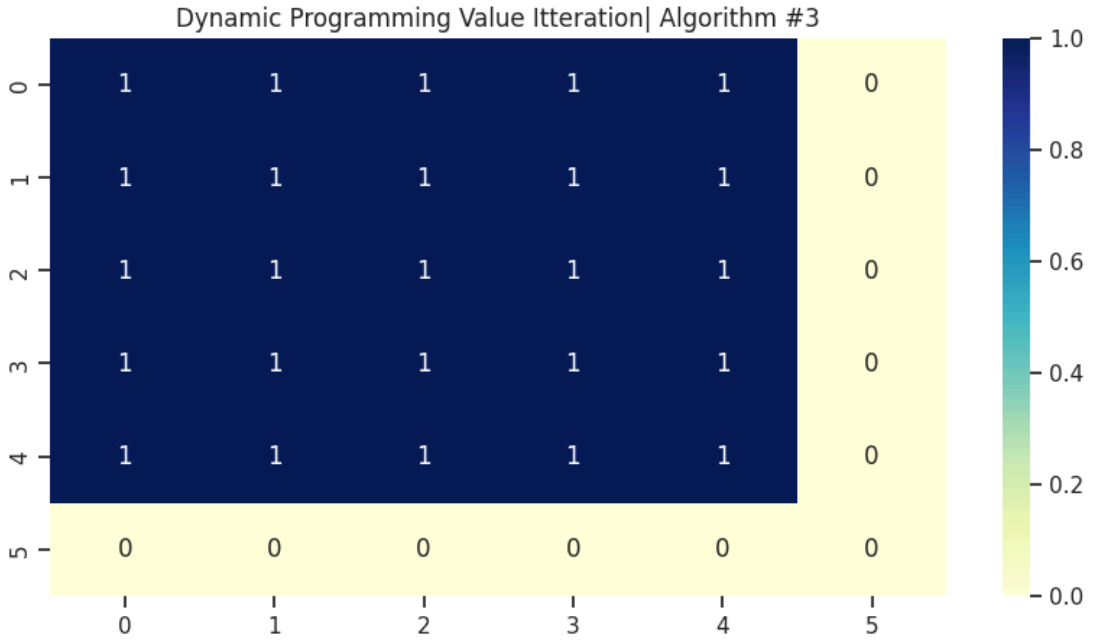
```
    if newR < 0 or newC < 0 or newR >= row or newC >= col or (newR == newC ==
1): # collide with the boundary or the wall
        return U[r][c]
    else:
        return U[newR][newC]

# Calculate the utility of a state per each given action
def calc_util(U, r, c, action):
    u = R
    u += 0.1 * γ * utility(U, r, c, (action-1)%4)
    u += 0.8 * γ * utility(U, r, c, action)
    u += 0.1 * γ * utility(U, r, c, (action+1)%4)
    return u

# value iteration steps to approximate the util
def evaluate_Policy(rand_policy, U):
    while True:
        nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]]
        error = 0
        for r in range(row):
            for c in range(col):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                nextU[r][c] = calc_util(U, r, c, rand_policy[r][c]) #
simplified Bellman update
                error = max(error, abs(nextU[r][c]-U[r][c]))
        U = nextU
        if error < Max_Err * (1-γ) / γ:
            break
    return U
# Actions during the policy iteration
def policy_itter(rand_policy, U):
    while True:
        U = evaluate_Policy(rand_policy, U)
        unchanged = True
        for r in range(row):
            for c in range(col):
                if (r <= 1 and c == 3) or (r == c == 1):
                    continue
                maxAction, maxU = None, -float("inf")
                for action in range(Steps):
                    u = calc_util(U, r, c, action)
                    if u > maxU:
                        maxAction, maxU = action, u
                if maxU > calc_util(U, r, c, rand_policy[r][c]):
                    rand_policy[r][c] = maxAction # the action that maximizes
the utility
                    unchanged = False
        if unchanged:
            break
    return rand_policy
print('Random policy after evaluation:', rand_policy, '\n')
print('Retrun U: ', U)
policy_table_3 = np.zeros((6, 6))
plt.title('Dynamic Programming Value Itteration| Algorithm #3')
```

```
for i in range(5):
    for j in range(5):
        policy_table_3[i][j]=np.max(U)
grd = sns.heatmap(policy_table_3, cmap="YlGnBu", annot=True)
grd
```

Output:



Conclusion:

1. **How does the value iteration algorithm differ from policy iteration, and what are its main steps in the context of finding optimal policies?**

   The value iteration algorithm and policy iteration algorithm are both fundamental methods for finding optimal policies in Markov Decision Processes (MDPs). Value iteration directly computes the optimal value function through iterative updates, while policy iteration alternates between policy evaluation and improvement steps until convergence. Value iteration is often computationally more efficient as it does not require separate evaluation and improvement steps, making it a preferred choice in many scenarios. Both algorithms aim to converge to the optimal policy by iteratively refining value estimates or policies until they no longer change significantly.