Project 2: Gridworld Policy Optimization with Dynamic Programming and Monte Carlo Algorithms

Student: Harshani Rathnayake
Departments: Mathematics and Statistics and Computer Science
Date: July 21, 2025
GitHub Repository: `https://github.com/Harshani-Rathnayake/Project-2-Gridworld-Policy-Optimization-with-Dynamic-Programming-and-Monte-Carlo-Algorithms.git`

Abstract

This study investigates reinforcement learning methods in a basic 5×5 gridworld setting. First, it analyzes the Bellman equations directly and evaluates the policy iteratively in order to estimate state value functions under a uniform random policy. Next, this study uses explicit solutions of the Bellman optimality formula, policy iteration, and value iteration to identify optimal policies. Moreover, it investigates Monte Carlo methods with and without investigating begins, and off-policy learning with importance sampling in an enlarged version of the environment containing terminal states. The results show which states have the highest expected returns and show how various algorithms converge to reliable answers.

Introduction

Training an agent to interact with an environment and make successive decisions that maximize cumulative rewards is a challenge that reinforcement learning attempts to solve. The gridworld environment is a common standard for researching and contrasting reinforcement learning algorithms because of its ease of use and interpretability. In this study, a 5×5 gridworld with special states, distinct reward dynamics, and transitions is analyzed. Using traditional dynamic programming techniques, this study determines optimal policies and estimates value functions under a uniform random policy. Furthermore, it uses off-policy learning with importance sampling and Monte Carlo methods to study learning in episodic settings with terminal states. This project shows and contrasts the convergence qualities and efficacy of different methods in identifying the best ways to make decisions.

Environment and Problem Setup

The 5×5 grid that makes up the environment produces 25 distinct states. The agent has four options for each state: move up, down, left, or right. Movements are controlled by the grid borders; attempting to step off the grid results in no change in position and a negative reward.

Part 1: Original gridworld

Initially, the grid contains four distinctive colored squares:

- **Blue square** - first row, second column: Any action results in an immediate jump to the red square and yields a reward of +5.
- **Green square** - first row, fifth column: Any action leads to a stochastic transition to either the red or yellow square, each with probability 0.5, yielding a reward of +2.5.
- **Red square** - fourth row, third column: A target state reached from the blue and green squares.
- **Yellow square** - fifth row, fifth column: Another target state is reachable from the green square.

All moves from white, red, or yellow squares earn 0; however, attempting to step outside the grid from a white or yellow square results in a penalty of -0.5. Future reward present values are calculated using a discount factor of $\gamma = 0.95$.

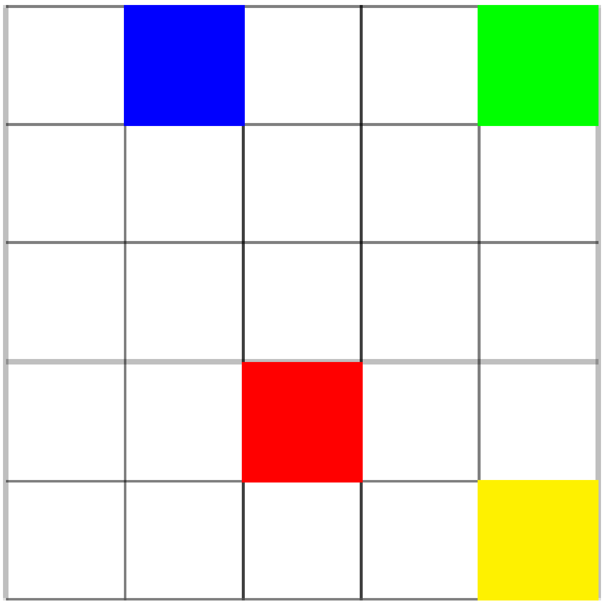

Figure 1: Original gridworld environment showing special colored squares

Part 2: Modified gridworld with terminal states

In the second part, the gridworld is modified as follows:

- The positions of the special squares are updated:
  - **Blue square** - first row, second column
  - **Green square** - first row, fifth column
  - **Red square** - fifth row, third column
  - **Yellow square** - fifth row, fifth column
- Three black terminal squares are added:
  - third row, first column

– third row, fifth column
– fifth row, first column

The episode terminates instantly when the agent enters any black terminal square.

Any move from a white, yellow, or red square to any other square in this configuration results in a reward of -0.2, while trying to leave the grid from these squares still results in a penalty of -0.5. The same $\gamma = 0.95$ discount factor is applied.

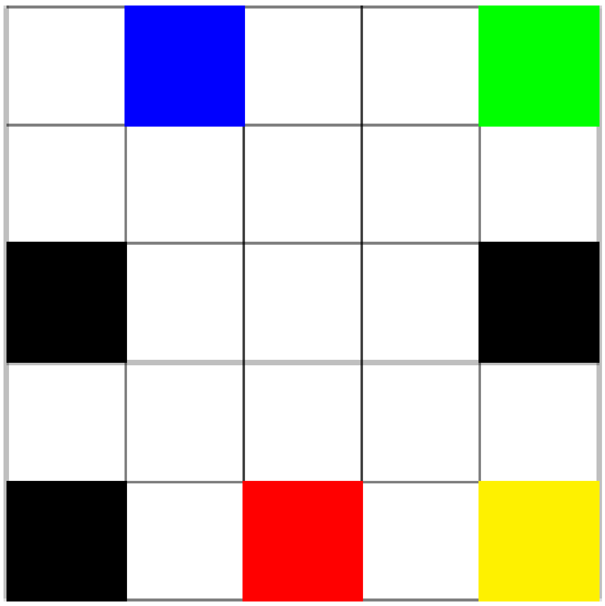

Figure 2: Modified gridworld environment showing special colored squares and black terminal squares

Part 1: Policy Evaluation and Optimal Policy Computation

This section initially examines the behavior of an agent in accordance with a uniform random policy, in which the four possible actions (up, down, left, and right) are all equally likely, in order to create a performance baseline. Under this policy, estimating the state value function reveals which states, even in the absence of learning, naturally produce higher long-term returns. This baseline is used as a point of comparison and evaluation for the efficacy of later, more complex policy optimization techniques.

Estimating the Value Function under a Uniform Random Policy

According to the uniform random policy, every action in every state has the same probability (0.25). The state value function was estimated using two methods under this policy:

Explicit Solution of Bellman Equations: The probabilities of policy action were used to weight transitions and rewards in order to create the expected transition probability matrix P_π and the expected reward vector R_π. The vector of the value function $\mathbf{V}$ was computed by solving the linear system:

$$\mathbf{V} = (I - \gamma P_\pi)^{-1} R_\pi,$$

where I is the identity matrix.

Iterative Policy Evaluation: The Bellman expectation backup was applied iteratively until convergence, updating each state value in the process described below, beginning with an initial value function of zeros:

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\big[R(s,a,s') + \gamma V(s')\big].$$

The process ended when the maximum change in state values was below a threshold (10^{-6}).

Results: The two approaches showed consistency as they converged to the same value estimates. The estimate of the state value function is shown in Table 1.

2.17	**4.73**	2.07	1.27	1.78
1.12	1.78	1.17	0.74	0.56
0.16	0.48	0.35	0.11	-0.19
-0.55	-0.28	-0.28	-0.44	-0.74
-1.11	-0.85	-0.81	-0.94	-1.24

Table 1: Estimated state value function under uniform random policy ($\gamma = 0.95$)

The blue square has the highest value (4.73), which is consistent with its deterministic jump to the red square and high immediate reward.

Deriving the Optimal Policy

Three methods were used to create optimal policies and value functions: value iteration, policy iteration, and explicit solution of the Bellman optimality equation mentioned above, all with the discount factor $\gamma = 0.95$.

Policy Iteration: In order to reach convergence, policy iteration alternates between policy evaluation and greedy policy improvement. The resulting value function and policy are shown in Tables 2 and 3.

Value Iteration: The value iteration selects the optimum action for each state and then iteratively applies the Bellman optimality update until convergence, at which point the optimal policy is retrieved.

Results: Consistent optimal value functions and policies were generated by both approaches, as demonstrated below.

21.00	22.10	21.00	19.95	18.38
19.95	21.00	19.95	18.95	18.00
18.95	19.95	18.95	18.00	17.10
18.00	18.95	18.00	17.10	16.25
17.10	18.00	17.10	16.25	15.43

Table 2: Optimal state value function ($\gamma = 0.95$) from policy and value iteration

3	0	2	2	0
0	0	0	0	2
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Table 3: Optimal policy actions (0: Up, 1: Down, 2: Left, 3: Right)

Policy Visualization: Figures 3 and 4 illustrate the derived optimal policies from policy iteration and value iteration, respectively.

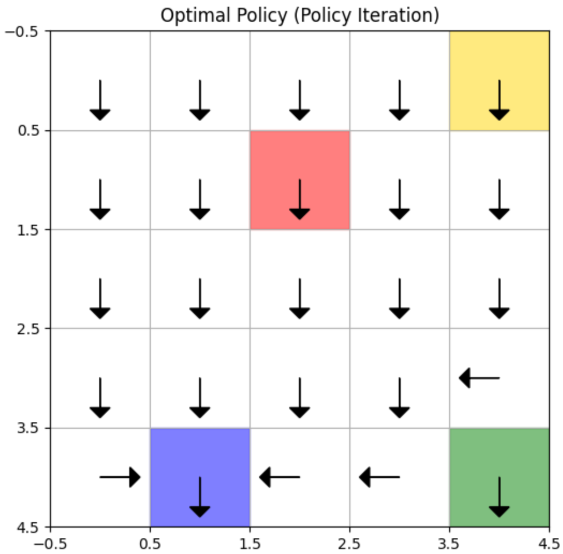

Figure 3: Optimal policy derived via policy iteration. Arrows represent recommended actions.

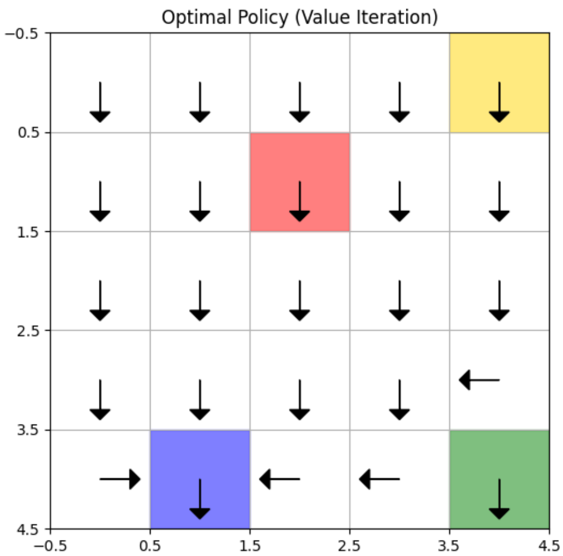

Figure 4: Optimal policy derived via value iteration. Arrows represent recommended actions.

Both methods converge to the same policy, confirming consistency under the defined environment.

Part 1: Discussion and Analysis

The outcomes of the policy optimization and evaluation process provide important new information on the structure of the gridworld environment and the efficacy of the reinforcement learning techniques used.

First, the blue square has the highest predicted return (4.73), according to the estimated state value function under the uniform random policy. This is consistent with intuition: the red square deterministically replaces the blue square, giving an instant reward of +5, which is then gradually discounted and spread to nearby states. In comparison, the green square has a lower expected return despite being a special state because of its stochastic nature: transitions divide between the red and yellow squares, effectively diluting the immediate reward to an average of +2.5.

The accuracy of the implementation and the coherence of the mathematical framework were demonstrated by the same value estimations obtained from the explicit solution of the Bellman equations and the iterative policy evaluation. The minor negative values in the lower rows represent states that, on average, are more likely to result in off-grid moves and penalties before reaching higher value states, resulting in slightly negative expected returns.

In the second part of the study, the optimal policies produced by policy iteration and value iteration converged to the same result, validating the theoretical hypothesis that both methods should provide the same optimal policy when the environment and discount factor remain fixed. The optimal value function yields values much higher than the uniform random policy, with a peak value of 22.10 in the blue square. This increase shows that the agent has the ability to exploit high-reward transitions repeatedly while avoiding penalties by adhering to the optimal policy.[1]

Moving toward the blue square and then using the deterministic jump to the red square is the main way that the optimal policy itself efficiently directs the agent towards the rewarding states. By prioritizing behaviors that result in immediate and frequent high rewards, the agent learns to maximize cumulative reward. This pattern demonstrates the usefulness of reinforcement learning algorithms in finding efficient strategies even in basic contexts.[1]

All things considered, the convergence of several approaches to the same optimal policy and value function demonstrates the adaptability of the reinforcement learning technique used. These findings also show that agent behavior is significantly influenced by incentive and environment design, resulting in policies that prioritize approaches with higher expected or immediate returns.

Part 2: Monte Carlo Methods in Episodic Gridworld

To estimate state value functions and learn policies, Monte Carlo methods were used to sample episodes in the same 5×5 gridworld environment, which is now considered episodic. After a certain number of steps or when the agent reaches a terminal state (special colored squares), an episode comes to an end.

On-Policy Monte Carlo Methods

Two on-policy methods have been examined below:

Exploring Starts: Each episode starts from a random state-action pair using the Monte Carlo with exploring starts method in order to promote varied exploration. The estimated value function and the learned policy were as follows after numerous episodes:

$$\begin{bmatrix} 2 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 3 & 1 \\ 0 & 2 & 3 & 3 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 2 & 1 & 1 & 0 \end{bmatrix}$$

Monte Carlo with exploring starts method - learned policy (0:Up, 1:Down, 2:Left, 3:Right)

$$\begin{bmatrix} -0.5 & -4.21 & -0.73 & -0.62 & -1.38 \\ -0.2 & -0.37 & -0.59 & -0.49 & -0.2 \\ 0.0 & -0.2 & -0.4 & -0.2 & 0.0 \\ -0.2 & -0.39 & -0.59 & -0.38 & -0.2 \\ 0.0 & -0.2 & -0.5 & -0.5 & -0.39 \end{bmatrix}$$

Value function from Monte Carlo with exploring starts method

Monte Carlo with ε-soft policy: This strategy encourages more extensive exploration by having the agent adopt a ε-soft policy, which is mostly greedy with occasional random actions. The learned policy and value function generated are shown below:

$$\begin{bmatrix} 3 & 1 & 2 & 2 & 1 \\ 0 & 3 & 0 & 2 & 2 \\ 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 2 \end{bmatrix}$$

Monte Carlo with ε-soft policy method - learned policy (0:Up, 1:Down, 2:Left, 3:Right)

$$\begin{bmatrix} 3.05 & 2.83 & 2.34 & 1.77 & 1.72 \\ 2.49 & 1.71 & 1.53 & 1.20 & 1.32 \\ 0.00 & 1.17 & 1.01 & 0.70 & 0.00 \\ 0.81 & 0.50 & 0.20 & 0.15 & 0.30 \\ 0.00 & 0.13 & -0.77 & -0.38 & 0.15 \end{bmatrix}$$

Value function from Monte Carlo with ε-soft method

Off-Policy Learning Using Importance Sampling

An off-policy Monte Carlo method was used to evaluate a deterministic target policy while following a different behavior policy, applying importance sampling to correct for the distribution mismatch. The resulting learned policy was:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Visualization and Comparison

Figure 5 displays the policies learned by each Monte Carlo method, with arrows representing the chosen actions and special squares marked with colors, while black squares represent terminal states.

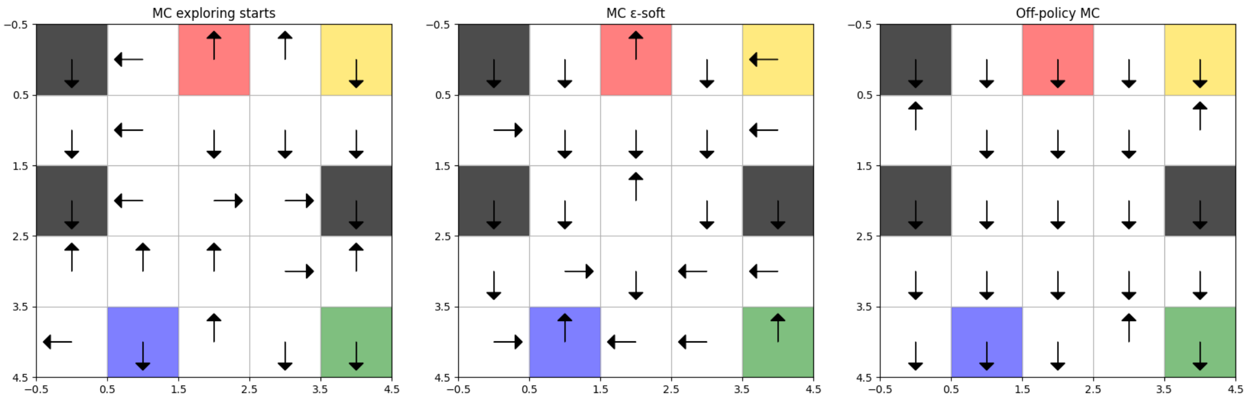

Figure 5: Learned policies from Monte Carlo methods: (Left) Exploring Starts; (Center) ε-soft; (Right) Off-policy with importance sampling.

Table 4 summarizes the estimated value functions from each method, indicating differences in learning quality and convergence behavior.

Method	Characteristics of Value Estimates
Exploring Starts	Due to high variance and sparse reward input, the values are primarily negative with scattered low values, suggesting unstable learning and poor convergence.
ε-soft Policy	Higher, smoother values indicate better exploration-exploitation balance and learning stability, particularly within close proximity of rewarding situations.
Off-Policy with Importance Sampling	Sparse and less reliable estimates; large variance in importance weights leads to unstable value estimation and fragmented policies.

Table 4: Summary of value function estimates from Monte Carlo methods

Part 2: Discussion and Interpretation

The Monte Carlo experiments highlight several important aspects of learning in episodic gridworld environments:

- **Exploring Starts** promotes extensive exploration by beginning each episode at random, but it has poor sample usage and large return variation, which leads to erratic and scattered policy and value estimations.
- **ε-soft policies** enhance learning by the combination of stochastic action selection and mostly greedy behavior. As a result, the value function converges more consistently, and state space coverage is improved. The agent is typically effectively guided toward high-reward states by the learned policies.

- **Off-policy Monte Carlo method**, Although theoretically strong, these face real-world difficulties in this setting because there are not many ideal action scenes. As a result, importance sampling ratios increase, resulting in unstable updates and large variance.

The observed variations among approaches highlight the need to establish a balance between exploration and exploitation as well as the challenges of off-policy evaluation using importance sampling, particularly in settings with lengthy episodes and few rewards.[2]

These results indicate that on-policy Monte Carlo methods with controlled exploration may yield more consistent and useful learning outcomes for episodic tasks with limited rewards. In contrast, off-policy methods might require the use of different algorithms or additional variance reduction strategies.

Conclusion

This study focused on both policy evaluation and optimal policy determination, demonstrating the application of multiple reinforcement learning techniques to a 5×5 gridworld setting. Iterative policy evaluation, policy iteration, value iteration, and explicit Bellman equation solving are examples of dynamic programming techniques that consistently generate stable and convergent value functions and policies by using the dynamics of the given environment.

Monte Carlo methods used in the episodic scenario provided useful insights into learning from the sampling experience. The significance of establishing a balance between exploration and exploitation is shown by the reasonable policy quality and smoother value estimations obtained by on-policy Monte Carlo methods with ε-soft exploration. On the other hand, off-policy Monte Carlo with importance sampling had challenges because of variance explosion, which limited its practical usefulness in this setting, whereas exploratory Monte Carlo experienced large variance and unstable learning.

Overall, the findings highlight the advantages and disadvantages of model-based versus model-free approaches: Monte Carlo methods provide flexible learning from experience but require careful exploration strategies and variance management, while dynamic programming performs best when fully aware of transition dynamics. Even in straightforward gridworld settings, this analysis highlights how important environment structure, reward design, and algorithmic selection are for reinforcement learning performance and stability.

Appendix

Part 1: Dynamic Programming in Gridworld

```

#Defined gridworld environment:
grid_size = 5  #5x5 grid
gamma = 0.95 #reward discount factor
actions = ['U', 'D', 'L', 'R']  #action space: Up, Down, Left, Right
action_prob = 0.25 #uniform random policy: equal probability for each action

#Helper: This convert (row, col) to state index
def to_state(row, col):
    return row * grid_size + col

#Special states defined by the problem statement
blue = (0,1)  #first row, second column
green = (0,4)  #first row, fifth column
red = (3,2)  #fourth row, third column
yellow = (4,4)   #fifth row, fifth column

#Converts the special states to indices:
blue_s = to_state(*blue)
green_s = to_state(*green)
red_s = to_state(*red)
yellow_s = to_state(*yellow)

#Total number of states
n_states = grid_size * grid_size

#Initialized transition probability (P) and reward (R) tensors
#Dimensions: [state, action, next_state]
P = np.zeros((n_states, len(actions), n_states))
R = np.zeros((n_states, len(actions), n_states))

#Build environment dynamics (transition + reward model)
for row in range(grid_size):
    for col in range(grid_size):
        s = to_state(row, col)
        for a_idx, a in enumerate(actions):
            #Special state: blue square always jumps to red with reward 5
            if s == blue_s:
                next_s = red_s
                P[s,a_idx,next_s] = 1.0
                R[s,a_idx,next_s] = 5
            #Special state: green square jumps randomly to yellow or red
   with reward 2.5
            elif s == green_s:
                P[s,a_idx,red_s] = 0.5
                P[s,a_idx,yellow_s] = 0.5
                R[s,a_idx,red_s] = 2.5
                R[s,a_idx,yellow_s] = 2.5
            else:
                #Regular square: compute next position after action
                if a == 'U':
                    next_row = max(row-1,0)
                    next_col = col
                elif a == 'D':
                    next_row = min(row+1,grid_size-1)
                    next_col = col
```

```
                elif a == 'L':
                    next_row = row
                    next_col = max(col-1,0)
                elif a == 'R':
                    next_row = row
                    next_col = min(col+1,grid_size-1)
                next_s = to_state(next_row, next_col)
                #Below checks if move hits border:
                if next_row == row and next_col == col and (
                    (row==0 and a=='U') or
                    (row==grid_size-1 and a=='D') or
                    (col==0 and a=='L') or
                    (col==grid_size-1 and a=='R')):
                    P[s,a_idx,next_s] = 1.0
                    R[s,a_idx,next_s] = -0.5  #penalty for trying to step off grid
                else:
                    P[s,a_idx,next_s] = 1.0
                    R[s,a_idx,next_s] = 0   #normal move with zero reward


#(1) Solve Bellman equations explicitly
#Compute expected transition matrix P_pi and expected reward R_pi under uniform policy
P_pi = np.zeros((n_states, n_states))
R_pi = np.zeros(n_states)

for s in range(n_states):
    for a_idx in range(len(actions)):
        for next_s in range(n_states):
            P_pi[s,next_s] += action_prob * P[s,a_idx,next_s]
            R_pi[s] += action_prob * P[s,a_idx,next_s] * R[s,a_idx,next_s]

#Solve linear system: V = (I - gamma * P_pi)^(-1) * R_pi
I = np.eye(n_states)
V_explicit = np.linalg.solve(I - gamma * P_pi, R_pi)
print("Value function by solving Bellman equations:")
print(np.round(V_explicit.reshape((grid_size,grid_size)),2))

#Find highest value states
max_val = np.max(V_explicit)
max_states = np.argwhere(np.isclose(V_explicit, max_val)).flatten()
positions = [(s // grid_size, s % grid_size) for s in max_states]
print(f"\nHighest value: {max_val:.2f}")
print("States with highest value (index):", max_states)
print("Positions (row,col):", positions)

#(2) Iterative policy evaluation under uniform random policy
#Goal: Estimate value function V(s) assuming the agent picks each action with equal probability
V = np.zeros(n_states) #Initialize value function: all zeros
theta = 1e-6  #Convergence threshold: stop when changes are very small
while True:
    delta = 0 #Track the maximum change in V across all states
    V_new = np.zeros_like(V)  #Temporary array to store new value estimates
    #Loop over all states
    for s in range(n_states):
        v = 0  #To accumulate expected return for state s under uniform random policy
        #Loop over all possible actions
```

```
        for a_idx in range(len(actions)):
            #Loop over all possible next states
            for next_s in range(n_states):
                #Expected contribution:
                #action_prob    transition probability    [reward + value of next state]
                v += action_prob * P[s, a_idx, next_s] * (R[s, a_idx, next_s] + gamma * V[next_s])
        V_new[s] = v  #Update value for state s
        #Update delta to track the largest difference from previous iteration
        delta = max(delta, abs(V_new[s] - V[s]))
    V = V_new  #Update value function for next iteration
    #Check for convergence: if max change is below threshold, stop
    if delta < theta:
        break
#Display final estimated value function
print("\nValue function by iterative policy evaluation:")
print(np.round(V.reshape((grid_size,grid_size)),2))

#(3) Policy iteration: alternating between policy evaluation and improvement
#Initialize policy: start by always taking action 0
policy = np.zeros(n_states, dtype=int)
#Initialize value function: start with all zeros
V_pi = np.zeros(n_states)
while True:
    #Step 1: Policy Evaluation
    #Compute value function V_pi for the current fixed policy until it converges
    while True:
        delta = 0   #Track maximum change to check convergence
        V_new = np.zeros_like(V_pi)  #Temporary array for updated values
        #Loop over each state to update value
        for s in range(n_states):
            a_idx = policy[s]  #action chosen by current policy in state s
            #Compute expected return:
            #sum over next states of P(s,a,next_s) * (reward +    * V[next_s])
            v = sum(
                P[s, a_idx, next_s] * (R[s, a_idx, next_s] + gamma * V_pi[next_s])
                for next_s in range(n_states)
            )
            V_new[s] = v
            #Update delta to track largest change
            delta = max(delta, abs(V_new[s] - V_pi[s]))
        V_pi = V_new  #update value function
        #Stop policy evaluation if value function converged (small changes only)
        if delta < theta:
            break

    #Step 2: Policy Improvement
    #Update policy to be greedy w.r.t. current value function V_pi
    policy_stable = True  #Flag to check if policy changes in this step
    for s in range(n_states):
        old_action = policy[s] #remember old action to compare later
        #Compute expected return for each possible action
        action_values = np.zeros(len(actions))
        for a_idx in range(len(actions)):
```

```python
            action_values[a_idx] = sum(
                P[s, a_idx, next_s] * (R[s, a_idx, next_s] + gamma * V_pi[next_s])
                for next_s in range(n_states)
            )
        #Choose the action with highest expected return
        best_action = np.argmax(action_values)
        policy[s] = best_action
        #If policy changed, mark as unstable so we keep iterating
        if old_action != best_action:
            policy_stable = False
    #If policy didn't change for any state, we have found the optimal policy
    if policy_stable:
        break
#Display final results
print("\nOptimal value function (policy iteration):")
print(np.round(V_pi.reshape((grid_size,grid_size)),2))
print("\nOptimal policy (0:U,1:D,2:L,3:R):")
print(policy.reshape((grid_size,grid_size)))

#(4) Value iteration to compute optimal value function and policy
#Initialize value function: start with all zeros
V_vi = np.zeros(n_states)
while True:
    delta = 0     #Tracks max change in value function to check for convergence
    V_new = np.zeros_like(V_vi)  #Temporary array to store new value estimates
    #Loop over each state to update its value
    for s in range(n_states):
        action_values = np.zeros(len(actions))   #Store expected return for each possible action
        #For each action, compute expected return:
        #sum over all possible next states: P(s,a,next_s) * [ R(s,a,next_s) +    * V(next_s) ]
        for a_idx in range(len(actions)):
            action_values[a_idx] = sum(
                P[s, a_idx, next_s] * (R[s, a_idx, next_s] + gamma * V_vi[next_s])
                for next_s in range(n_states)
            )
        #Choose the maximum expected return among actions     Bellman optimality update
        V_new[s] = np.max(action_values)
        #Update delta to see how much V changed
        delta = max(delta, abs(V_new[s] - V_vi[s]))
    #Update value function
    V_vi = V_new
    #Check for convergence: if changes are smaller than small threshold theta, stop
    if delta < theta:
        break

#Extract optimal policy from final value function
policy_vi = np.zeros(n_states, dtype=int)   #Array to store best action per state
for s in range(n_states):
    action_values = np.zeros(len(actions))
    #For each action, compute expected return as before
    for a_idx in range(len(actions)):
```

```
        action_values[a_idx] = sum(
            P[s, a_idx, next_s] * (R[s, a_idx, next_s] + gamma * V_vi[next_s])
            for next_s in range(n_states)
        )
    #Best action is the one with highest expected return
    policy_vi[s] = np.argmax(action_values)
#Display results
print("\nOptimal value function (value iteration):")
print(np.round(V_vi.reshape((grid_size,grid_size)),2))
print("\nOptimal policy (0:U,1:D,2:L,3:R):")
print(policy_vi.reshape((grid_size,grid_size)))

#Plot function to visualize policy as arrows
def plot_policy(policy, title):
    fig, ax = plt.subplots(figsize=(6,6))
    ax.set_xlim(-0.5, grid_size-0.5)
    ax.set_ylim(-0.5, grid_size-0.5)
    ax.set_xticks(np.arange(-0.5, grid_size, 1))
    ax.set_yticks(np.arange(-0.5, grid_size, 1))
    ax.grid(True)
    ax.set_title(title)
    #Draw special squares
    special_states = {
        'blue': (blue, 'blue'),
        'green': (green, 'green'),
        'red': (red, 'red'),
        'yellow': (yellow, 'gold')
    }
    for name, (pos, color) in special_states.items():
        row, col = pos
        ax.add_patch(plt.Rectangle((col-0.5, grid_size - row -1 -0.5), 1, 1, color=color, alpha=0.5))
    #Draw arrows for each state's chosen action
    action_to_delta = {
        0: (0, +0.3), #U
        1: (0, -0.3), #D
        2: (-0.3, 0), #L
        3: (+0.3, 0) #R
    }
    for row in range(grid_size):
        for col in range(grid_size):
            s = to_state(row,col)
            a = a = policy[s] if np.isscalar(policy[s]) else np.argmax(policy[s])
            dx, dy = action_to_delta[a]
            plot_row = grid_size - row -1
            ax.arrow(col, plot_row, dx, dy, head_width=0.2, head_length=0.1, fc='k', ec='k')
    plt.gca().invert_yaxis()
    plt.show()

#Plots the optimal policies found
plot_policy(policy, "Optimal Policy (Policy Iteration)")
plot_policy(policy_vi, "Optimal Policy (Value Iteration)")

#Compare policies: are they identical or not
same = np.all(policy == policy_vi)
print("Are policies from policy iteration and value iteration identical?", same)
```

```
#Show where they differ:
diff = (policy != policy_vi).reshape((grid_size, grid_size))
print("Differences between policies (True = different):")
print(diff)
```

Listing 1: Python Code for Value Iteration, Policy Iteration and Bellman Equations in Gridworld

Part 2: Monte Carlo Methods in Gridworld

```

#Define gridworld and parameters
grid_size = 5  #Gridworld is 5x5 cells
gamma = 0.95  #Discount factor: future rewards are worth 95% as much
actions = ['U', 'D', 'L', 'R']  #Action set: Up, Down, Left, Right
action_idx = {a:i for i,a in enumerate(actions)}  #Helper: map actions to
    indices for array indexing

#Define special colored squares (positions are (row, col) starting from 0)
blue = (0, 1) #first row, second column
green = (0, 4) #first row, fifth column
red = (4, 2) #last row, third column
yellow = (4, 4)  #last row, last column
#Terminal (black) squares where episode ends immediately when agent steps in
black_terminals = [
    (2, 0), #third row, first column
    (2, 4), #third row, last column
    (4, 0)  #last row, first column
]

#Convert positions (row, col) to state indices (flattened index)
blue_s = blue[0]*grid_size + blue[1]
green_s = green[0]*grid_size + green[1]
red_s = red[0]*grid_size + red[1]
yellow_s = yellow[0]*grid_size + yellow[1]
black_s = [r*grid_size + c for (r, c) in black_terminals]  #list of terminal
     state indices
n_states = grid_size * grid_size   #total number of states in grid

#Initialize starting policy: uniform random policy
#Each state: equal probability for each action
policy = np.ones((n_states, len(actions))) / len(actions)

#Function to simulate a single episode in the Gridworld environment under a
    given policy
#Parameters:
 #policy : current policy, mapping each state to a probability distribution
    over actions
 #exploring_starts : if True, start from a random non-terminal state and
    random action
 #max_steps : maximum number of steps to run in an episode to prevent
    infinite loops
 #episode  : list of (state, action, reward) tuples representing the
    trajectory
def generate_episode(policy, exploring_starts=False, max_steps=100):
    episode = []  #list to store the sequence of (state, action, reward)
    #Choose starting state and starting action
    if exploring_starts:
        #For exploring starts: start from a random non-terminal state and a
    random action
        s = random.choice([s for s in range(n_states) if s not in black_s])
```

```
        a = random.choice(range(len(actions)))
    else:
        #Otherwise: start from a random non-terminal state and select action according to the current policy
        s = random.choice([s for s in range(n_states) if s not in black_s])
        a = np.random.choice(len(actions), p=policy[s])
    #Loop to simulate the episode, step by step
    for _ in range(max_steps):
        #Convert flattened state index to (row, col) coordinates
        row, col = divmod(s, grid_size)
        #Check if current state is a terminal state (black square): if yes, episode ends
        if s in black_s:
            break
        #Check for special colored squares and apply special transitions and rewards
        if s == blue_s:
            reward = 5  #reward for blue square
            next_s = red_s   #teleport to red square
        elif s == green_s:
            reward = 2.5   #reward for green square
            #teleport randomly to red or yellow square
            next_s = np.random.choice([red_s, yellow_s])
        else:
            #For normal white squares: determine next position based on action
            if actions[a] == 'U': #move up
                next_row = max(row-1, 0)
                next_col = col
            elif actions[a] == 'D':  #move down
                next_row = min(row+1, grid_size-1)
                next_col = col
            elif actions[a] == 'L': #move left
                next_row = row
                next_col = max(col-1, 0)
            else:       #move right ('R')
                next_row = row
                next_col = min(col+1, grid_size-1)
            #Check if the agent attempted to step off the grid (hit the wall)
            if next_row == row and next_col == col and (
                (row==0 and actions[a]=='U') or
                (row==grid_size-1 and actions[a]=='D') or
                (col==0 and actions[a]=='L') or
                (col==grid_size-1 and actions[a]=='R')):
                reward = -0.5   # penalty for hitting the wall
            else:
                reward = -0.2 #penalty for moving from white square to white square
          #Convert next (row, col) to flattened state index
            next_s = next_row * grid_size + next_col
        #Add the current step to the episode list
        episode.append((s, a, reward))
        # Move to the next state
        s = next_s
        # Check again if the new state is a terminal state: if yes, episode ends
        if s in black_s:
            break
        #Choose next action in the new state according to the current policy
```

```python
        a = np.random.choice(len(actions), p=policy[s])
    #Return the full episode: list of (state, action, reward)
    return episode

#Helper for policy iteration (not Monte Carlo): build P_pi and R_pi under deterministic policy
def build_P_pi_and_R_pi(policy, P, R):
    P_pi = np.zeros((n_states, n_states))  #transition matrix under policy
    R_pi = np.zeros(n_states)   #expected reward vector under policy
    for s in range(n_states):
        a = policy[s]  #deterministic action chosen in state s
        P_pi[s, :] = P[s, a, :]  #probabilities of next states
        R_pi[s] = np.sum(P[s, a, :] * R[s, a, :])  #expected immediate reward
    return P_pi, R_pi

#Policy iteration using explicit matrix inversion (solve Bellman equations)
def policy_iteration_explicit(P, R, gamma, policy, max_iters=100):
    for i in range(max_iters):
        #Step 1: Policy evaluation
        P_pi, R_pi = build_P_pi_and_R_pi(policy, P, R)
        I = np.eye(n_states)
        V = np.linalg.solve(I - gamma * P_pi, R_pi)  #solve V = (I - gamma*P_pi)^(-1) * R_pi
        #Step 2: Policy improvement
        policy_stable = True
        for s in range(n_states):
            action_values = np.zeros(len(actions))
            for a in range(len(actions)):
                #expected return if we take action a in state s
                action_values[a] = np.sum(P[s, a, :] * (R[s, a, :] + gamma * V))
            best_action = np.argmax(action_values)
            if best_action != policy[s]:
                policy[s] = best_action
                policy_stable = False
        if policy_stable:
            print(f"Policy converged after {i+1} iterations")
            break
    return policy, V

#Same plot function as before, but with the black terminal squares
def plot_policy(policy, title):
    fig, ax = plt.subplots(figsize=(6,6))
    ax.set_xlim(-0.5, grid_size-0.5)
    ax.set_ylim(-0.5, grid_size-0.5)
    ax.set_xticks(np.arange(-0.5, grid_size, 1))
    ax.set_yticks(np.arange(-0.5, grid_size, 1))
    ax.grid(True)
    ax.set_title(title)
    #Draw special squares
    special_states = {
        'blue': (blue, 'blue'),
        'green': (green, 'green'),
        'red': (red, 'red'),
        'yellow': (yellow, 'gold')
    }
    for name, (pos, color) in special_states.items():
        row, col = pos
        ax.add_patch(plt.Rectangle((col-0.5, grid_size - row -1 -0.5), 1, 1,
```

```
    color=color, alpha=0.5))
    #NEW: Draw black terminal squares
    for (row, col) in black_terminals:
        ax.add_patch(plt.Rectangle((col-0.5, grid_size - row -1 -0.5), 1, 1,
    color='black', alpha=0.7))
    #Draw arrows for policy
    action_to_delta = {
        0: (0, +0.3), #U
        1: (0, -0.3), #D
        2: (-0.3, 0), #L
        3: (+0.3, 0) #R
    }
    for row in range(grid_size):
        for col in range(grid_size):
            s = row * grid_size + col
            a = a = policy[s] if np.isscalar(policy[s]) else np.argmax(
    policy[s])
            dx, dy = action_to_delta[a]
            plot_row = grid_size - row -1
            ax.arrow(col, plot_row, dx, dy, head_width=0.2, head_length=0.1,
    fc='k', ec='k')
    plt.gca().invert_yaxis()
    plt.show()

#(1) Monte Carlo control with exploring starts (start random state/action)
Q = np.zeros((n_states, len(actions)))       # Action-value function
returns_count = np.zeros((n_states, len(actions)))  # Count of visits per (s
    ,a)
for i in range(5000):   #number of episodes
    episode = generate_episode(policy, exploring_starts=True)
    G = 0   #return (discounted sum of rewards)
    visited = set()   #to only update first visit in each (s,a)
    for s, a, r in reversed(episode):
        G = gamma*G + r
        if (s,a) not in visited:
            returns_count[s,a] += 1
            #Incremental mean update
            Q[s,a] += (G - Q[s,a]) / returns_count[s,a]
            best_a = np.argmax(Q[s])
            #Update policy: greedy in Q
            policy[s] = np.eye(len(actions))[best_a]
            visited.add((s,a))
print("MC with exploring starts - learned policy:")
print(np.argmax(policy, axis=1).reshape((grid_size, grid_size)))
plot_policy(np.argmax(policy, axis=1), "MC exploring starts")
#Compute value function as max_a Q(s,a)
V_mc_es = np.max(Q, axis=1)
print("\nValue function from MC with exploring starts:")
print(np.round(V_mc_es.reshape((grid_size, grid_size)), 2))


#(2) Monte Carlo control with ϵ-soft policy
Q2 = np.zeros((n_states, len(actions)))
returns2 = np.zeros((n_states, len(actions)))
epsilon = 0.1
policy2 = np.ones((n_states, len(actions))) / len(actions)  #start
    equiprobable
for i in range(5000):
    episode = generate_episode(policy2)
    G = 0
```

```python
    visited = set()
    for s,a,r in reversed(episode):
        G = gamma*G + r
        if (s,a) not in visited:
            returns2[s,a] += 1
            Q2[s,a] += (G - Q2[s,a]) / returns2[s,a]
            best_a = np.argmax(Q2[s])
            #ϵ-soft policy update: mostly greedy, small prob for others
            for a_idx in range(len(actions)):
                if a_idx == best_a:
                    policy2[s,a_idx] = 1 - epsilon + epsilon/len(actions)
                else:
                    policy2[s,a_idx] = epsilon/len(actions)
            visited.add((s,a))

print("\nMC with ϵ-soft policy - learned policy:")
print(np.argmax(policy2, axis=1).reshape((grid_size, grid_size)))
plot_policy(np.argmax(policy2, axis=1), "MC ϵ-soft")
V_mc_soft = np.max(Q2, axis=1)
print("\nValue function from MC with ϵ-soft:")
print(np.round(V_mc_soft.reshape((grid_size, grid_size)),2))

#(3) Off-policy MC with importance sampling
target_policy = np.zeros((n_states, len(actions)))  #target policy we want to learn: greedy in Q
Q3 = np.zeros((n_states, len(actions)))  #action-value estimates
C = np.zeros((n_states, len(actions))) #cumulative sum of importance weights
#Initialize target policy as uniform
for s in range(n_states):
    target_policy[s] = np.ones(len(actions)) / len(actions)
for i in range(5000):
    episode = generate_episode(policy)  #generate episode using behavior policy (here, policy is uniform)
    G = 0
    W = 1
    for s,a,r in reversed(episode):
        G = gamma*G + r
        C[s,a] += W
        Q3[s,a] += (W/C[s,a])*(G - Q3[s,a])  #weighted update
        best_a = np.argmax(Q3[s])
        #Make target policy greedy in Q
        target_policy[s] = np.eye(len(actions))[best_a]
        if a != best_a:
            break   #stop if action taken is different from greedy: importance weight becomes 0
        W = W / policy[s,a]  #update importance weight
print("\nOff-policy MC with importance sampling - learned policy:")
print(np.argmax(target_policy, axis=1).reshape((grid_size, grid_size)))
plot_policy(np.argmax(target_policy, axis=1), "Off-policy MC")
```

Listing 2: Python Code for Monte Carlo with Exploring Starts, ε - soft Policy, and Off-Policy Learning

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, 2nd edition, 2018.

[2] Csaba Szepesvári. *Algorithms for Reinforcement Learning.* Morgan & Claypool Publishers, 2010.