Reinforcement Learning

Ansuman Patra

COPS Summer of Code 2025

Intelligence Guild

Club of Programmers, IIT (BHU) Varanasi

Official IG Website: `https://cops-iitbhu.github.io/IG-website`

June 10, 2025

Contents

1 Optimal Bellman Equations for Value and Action-Value Functions

1.1 What is Optimality?

In Markov Decision Processes (MDPs), **optimality** refers to finding a policy π that maximizes the expected cumulative discounted reward.

A policy π is optimal if:

$$V^{\pi}(s) \geq V^{\pi'}(s), \quad \forall s \in \mathcal{S},\ \forall \pi'$$

where $V^{\pi}(s)$ is the value function, representing the expected return starting from state s under policy π.

The **optimal value function** and **optimal action-value function** are:

$$V^*(s) = \max_{\pi} V^{\pi}(s), \quad Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

Bellman optimality equations characterize V^* and Q^* recursively, enabling dynamic programming methods such as *value iteration* and *policy iteration*.

1.2 Expectation and Its Role in Bellman Equations

Expectation ($\mathbb{E}$) refers to the weighted average of all possible outcomes of a random variable.

For a discrete random variable X with outcomes x_i and probabilities $P(x_i)$:

$$\mathbb{E}[X] = \sum_i x_i \cdot P(x_i)$$

In MDPs, expectations appear because:

- Transitions are stochastic: $s' \sim P(s' \mid s, a)$
- Rewards may also be stochastic: $r \sim R(s, a, s')$

Role in Bellman Equations:
Expectation is used to handle uncertainty in:

- the next state (s')
- the immediate reward (r)

This gives rise to the recursive Bellman expectation equations:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[r + \gamma V^{\pi}(s') \mid s]$$

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[r + \gamma Q^{\pi}(s', a') \mid s, a]$$

***Optimality equations extend this by replacing the expectation over actions with a maximization.**

1.3 Derivation of Bellman Optimality Equations

Let an MDP be defined by:

- States $s \in \mathcal{S}$
- Actions $a \in \mathcal{A}$
- Transition function $P(s' \mid s, a)$
- Reward function $r(s, a, s')$
- Discount factor $\gamma \in [0, 1)$

1. Bellman Optimality Equation for $V^*(s)$

$$V^*(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[r(s, a, s') + \gamma V^*(s') \right]$$

2. Bellman Optimality Equation for $Q^*(s, a)$

$$Q^*(s, a) = \sum_{s'} P(s' \mid s, a) \left[r(s, a, s') + \gamma \max_{a'} Q^*(s', a') \right]$$

1.4 Solutions to Exercises 3.25–3.29 from Sutton and Barto

Exercise 3.25: v_* in terms of q_*

$$v_*(s) = \max_a q_*(s, a)$$

Exercise 3.26: q_* in terms of v_* and p

$$q_*(s, a) = \sum_{s'} p(s' \mid s, a)[r + \gamma v_*(s')]$$

Exercise 3.27: π_* in terms of q_*

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_a q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

Exercise 3.28: π_* in terms of v_* and p

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_a \sum_{s'} p(s'|s,a)[r + \gamma v_*(s')] \\ 0 & \text{otherwise} \end{cases}$$

Exercise 3.29: Bellman equations in terms of p and r

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_\pi(s')]$$

$$v_*(s) = \max_a \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma v_*(s')]$$

$$q_\pi(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')]$$

$$q_*(s,a) = \sum_{s'} p(s'|s,a)[r(s,a,s') + \gamma \max_{a'} q_*(s',a')]$$

2 Policy and Value Iteration

2.1 Prove Convergence of Policy and Value Iterations

Policy Iteration (PI)

*P*olicy iteration alternates between policy evaluation and policy improvement, ensuring that each step improves or maintains the policy quality.

Policy Improvement:

Reference - yt/Mutual Information

- **Given:** A value function $V_\pi(s)$, determine a better policy than π.
- If π is deterministic: $a = \pi(s)$.
- For an optimal policy π_*: $\pi_*(s) = \arg\max_a q_*(s,a)$.

Define a new policy π':

$$\pi'(s) = \arg\max_a q_\pi(s,a)$$

Result: Due to the *Policy Improvement Theorem*:

$$V_\pi(s) \leq V_{\pi'}(s) \quad \forall s \in \mathcal{S}$$

Mechanism of Convergence:

1. **Policy Evaluation:** For a given policy π_k, the value function V_{π_k} is computed iteratively until it converges to the true value function. Each iteration reduces the error between the estimated and actual values.

2. **Policy Improvement:** The policy π_k is updated greedily based on the value function V_{π_k}. This ensures that the updated policy π_{k+1} is at least as good as π_k, and strictly better unless π_k is already optimal.

Convergence Guarantee: Since there are only a finite number of deterministic policies in a Markov Decision Process (MDP), the algorithm converges in a finite number of iterations when the policy stops changing (reaches the optimal policy π_*).

Value Iteration (VI)

*V*alue iteration directly computes the optimal value function V_* by iteratively applying the Bellman optimality operator.

Mechanism of Convergence:

1. **Bellman Optimality Operator:** The value function is updated as:

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V_k(s') \right]$$

2. **Contraction Mapping:** The Bellman operator reduces the difference between successive value functions by a fixed fraction γ (the discount factor).

3. **Convergence:** As $k \to \infty$, V_k converges to the unique fixed point V_*, which satisfies the Bellman optimality equation.

Convergence Guarantee:

- The contraction property ensures that the error $|V_{k+1} - V_k| < \epsilon$ for any $\epsilon > 0$ decreases exponentially with each iteration.

- The algorithm stops when the difference between successive value functions is smaller than a predefined threshold ϵ, indicating convergence to an approximately optimal value function.

2.2 What Actually is GPI?

According to Sutton and Barto:

> "GPI refers to the general idea of letting policy evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI."

GPI is a framework where policy evaluation and policy improvement interact iteratively until they converge to an optimal policy.

- **Policy Evaluation:** Computes the value function V^π or Q^π for the current policy π.
- **Policy Improvement:** Updates the policy to be greedy (or ϵ-greedy) with respect to the current value estimates.

Working:

- **Loop:**
  - Evaluate $\pi \rightarrow$ Get V^π.
  - Improve $\pi \rightarrow$ New policy π' greedy w.r.t. V^π.
- **Termination:** Stops when no further improvement is possible (optimal policy π^* is found).

Reason for Convergence:

- **Policy Evaluation** contracts to V^π.
- **Policy Improvement** ensures $V^{\pi_{k+1}} \geq V^{\pi_k}$.
- Finite MDPs guarantee convergence to π^*.

2.3 What is the Complete and Rigorous Proof of Their Convergence?

Note - Used help of DeepSeek for this part

Policy Iteration Convergence:

Policy Improvement Theorem: If policy π' is greedy with respect to V_π, then

$$V_{\pi'}(s) \geq V_\pi(s) \quad \forall s$$

with strict inequality if π is suboptimal.

Proof:

$$V_\pi(s) \leq Q_\pi(s, \pi'(s)) = T_{\pi'} V_\pi(s) \leq (T_{\pi'})^k V_\pi(s) \xrightarrow[k\to\infty]{} V_{\pi'}(s)$$

Termination: Finite policies (since $|\Pi| = |A|^{|S|}$) and monotonic improvement imply convergence to optimal policy π^*.

Value Iteration Convergence:

The Bellman optimality operator T^* is a γ-contraction in the $\|\cdot\|_\infty$ norm:

$$\|T^* V_1 - T^* V_2\|_\infty \leq \gamma \|V_1 - V_2\|_\infty$$

Proof:

$$\max_s \left| \max_a [T_a V_1(s)] - \max_a [T_a V_2(s)] \right| \leq \max_s \max_a |T_a V_1(s) - T_a V_2(s)| \leq \gamma \|V_1 - V_2\|_\infty$$

By the Banach Fixed-Point Theorem, iterating

$$V_{k+1} = T^* V_k$$

converges to the optimal value function V^*.

2.4 What is the Time and Memory Complexity for Each of the Algorithms? Where are Each of Them Particularly Useful?

Note- Used help of AI for this part

Policy Iteration Complexity

Time Complexity:

- Each iteration of Policy Iteration consists of:
  1. **Policy Evaluation:** Compute V_π for the current policy π (solving a system of linear equations or iterative evaluation until convergence). For $|S|$ states, this is $O(|S|^2)$ per evaluation.
  2. **Policy Improvement:** Update the policy π greedily with respect to V_π, $O(|S||A|)$.
- Let K be the number of policy iterations until convergence.

$$O\left(K \times \left(|S|^2 + |S||A|\right)\right)$$

Space Complexity:

- Storing the value function: $O(|S|)$
- Storing the policy: $O(|S|)$
- Storing the transition model: $O(|S|^2|A|)$ (dense)

$$O(|S|^2|A|)$$

Value Iteration Complexity

Time Complexity:

- Each iteration updates the value for each state by taking the maximum over actions, involving a sum over successor states: $O(|S|^2|A|)$ per iteration.
- Number of iterations: $O\left(\frac{\log(1/\epsilon)}{1-\gamma}\right)$, where ϵ is the precision.

$$O\left(|S|^2|A| \times \frac{\log(1/\epsilon)}{1-\gamma}\right)$$

Space Complexity:

- Storing the value function: $O(|S|)$
- Storing the transition model: $O(|S|^2|A|)$

$$O(|S|^2|A|)$$

3 Implementing DP algorithms on Frozen Lake from Gymnasium

3.1 Implementing policy iteration on the original frozen lake environment

1. Importing Libraries

```
[title={Import Libraries}]
import numpy as np
import gymnasium as gym
```

Here, `numpy` is used for numerical operations and array handling, while `gymnasium` provides the FrozenLake environment.

2. Defining the Policy Iteration Function

```
[title={Policy Iteration Function}]
def policy_iteration(env, gamma=0.99, theta=1e-8):
    nS = env.observation_space.n
    nA = env.action_space.n
    policy = np.zeros(nS, dtype=int)
    V = np.zeros(nS)
    policy_eval_iterations = 0
    policy_improvement_steps = 0

    while True:
        # Policy Evaluation
        while True:
            delta = 0
            policy_eval_iterations += 1
            for s in range(nS):
                a = policy[s]
                v = sum(p * (r + gamma * V[s_]) for p, s_, r, d in env.P[
                    s][a])
                delta = max(delta, abs(v - V[s]))
                V[s] = v
            if delta < theta:
                break

        # Policy Improvement
        policy_stable = True
        policy_improvement_steps += 1
        for s in range(nS):
            old_action = policy[s]
            action_values = np.zeros(nA)
            for a in range(nA):
                action_values[a] = sum(p * (r + gamma * V[s_]) for p, s_,
                     r, d in env.P[s][a])
            best_action = np.argmax(action_values)
            policy[s] = best_action
            if best_action != old_action:
                policy_stable = False

        if policy_stable:
            break

    return policy, V, policy_eval_iterations, policy_improvement_steps
```

Explanation:

- `nS` and `nA` represent the number of states and actions respectively.
- `policy` is initialized to always take action 0.
- `V` stores the value function for each state.
- **Policy Evaluation**: Iteratively updates the value function for the current policy until convergence (controlled by `theta`).
- **Policy Improvement**: For each state, computes action values and updates the policy greedily.
- The loop continues until the policy is stable (no changes).

Note - Used help of AI for this part mainly for conceptually understanding the process of implementing policy iteration in python. Also used help of - Medium.Nan - Markov decision process: policy iteration with code implementation

3. Running the Learned Policy

```
[title={Run Policy Function}]
def run_policy(env, policy, episodes=10, render=True):
    episode_lengths = []
    rewards = []
    successes = 0

    for ep in range(episodes):
        obs, _ = env.reset()
        done = False
        total_reward = 0
        steps = 0
        if render:
            print(f"\nEpisode {ep+1}")
            env.render()
        while not done:
            action = policy[obs]
            obs, reward, done, _, _ = env.step(action)
            total_reward += reward
            steps += 1
            if render:
                env.render()
        episode_lengths.append(steps)
        rewards.append(total_reward)
        if total_reward > 0:
            successes += 1

    avg_length = np.mean(episode_lengths)
    success_rate = successes / episodes
    avg_reward = np.mean(rewards)
    var_length = np.var(episode_lengths)

    print("\n--- Metrics ---")
    print(f"Average Episode Length: {avg_length:.2f}")
    print(f"Success Rate: {success_rate:.2%}")
    print(f"Average Reward per Episode: {avg_reward:.2f}")
    print(f"Variance in Episode Length: {var_length:.2f}")

    return avg_length, success_rate, avg_reward, var_length
```

This function runs the environment using the learned policy for a specified number of episodes, rendering each step and printing the total reward per episode.

4. Main Execution Block

```
[title={Main Block}]
if __name__ == "__main__":
    env = gym.make("FrozenLake-v1", is_slippery=True, render_mode="human"
        )
    env = env.unwrapped  # Unwrap environment to access transition
        probabilities

    policy, V = policy_iteration(env)

    print("Optimal Policy:")
    print(policy.reshape((4, 4)))

    run_policy(env, policy)
```

Details:

- The FrozenLake environment is created with slippery (stochastic) dynamics.
- `env.unwrapped` is used to access the transition model `env.P`, which is necessary for policy iteration.
- After computing the optimal policy and value function, the policy is reshaped and printed in a 4x4 grid corresponding to the lake layout.
- Finally, the policy is executed and rendered.

4 Implementing and Explaining Custom and Expanded FrozenLake Environments

4.1 1. Imports and Setup

```
[title={Imports and Setup}]
import numpy as np
import gymnasium as gym
from gymnasium import spaces
from gymnasium.envs.registration import register
import pygame
import time
```

Explanation:

- `numpy`: For numerical operations and random choices.
- `gymnasium` and `spaces`: For defining and registering custom RL environments.
- `pygame`: For rendering the environment visually.
- `time`: For timing policy iteration and controlling animation speed.

4.2 2.1 Custom FrozenLake Environment (8x8)

```
[title={Custom 8x8 FrozenLake Environment}]
class CustomFrozenLake(gym.Env):
    metadata = {"render_modes": ["human"], "render_fps": 4}

    def __init__(self, render_mode=None):
        super().__init__()
        self.map = [
            "SFFFFFFF",
            "FFFFFFFF",
            "FFFHFFFF",
            "FFFFFHFF",
            "FFFHFFFF",
            "FHHFFFHF",
            "FHFFHFHF",
            "FFFHFFFG"
        ]
        self.setup_environment()
        self.render_mode = render_mode
        self.window = None
        self.clock = None
        self.window_size = 512
        self.cell_size = self.window_size // self.nrow
```

Explanation:

- Defines an 8x8 custom FrozenLake grid with start (`S`), goal (`G`), frozen (`F`), and hole (`H`) tiles.
- Sets up rendering parameters for visualization.

2.2 Environment Setup and State Handling

```
[title={Environment Setup and State Handling}]
    def setup_environment(self):
        self.nrow = len(self.map)
        self.ncol = len(self.map[0])
        self.nS = self.nrow * self.ncol
        self.nA = 4
        self.observation_space = spaces.Discrete(self.nS)
        self.action_space = spaces.Discrete(self.nA)
        self.P = self._build_transition_model()
        self.state = self._to_state(0, 0)

    def _to_state(self, r, c):
        return r * self.ncol + c

    def _to_pos(self, s):
        return divmod(s, self.ncol)
```

Explanation:

- Initializes environment dimensions, state/action spaces, and builds the transition model.
- Helper functions convert between (row, col) and flat state indices.

2.3 Transition Model with Slippery Dynamics

```
[title={Transition Model with Slippery Dynamics}]
    def _build_transition_model(self):
        P = {s: {a: [] for a in range(self.nA)} for s in range(self.nS)}
        slip_probs = [0.1, 0.8, 0.1]
        directions = {0: (0, -1), 1: (1, 0), 2: (0, 1), 3: (-1, 0)}

        for r in range(self.nrow):
            for c in range(self.ncol):
                s = self._to_state(r, c)
                cell = self.map[r][c]
                if cell in "GH":
                    for a in range(self.nA):
                        P[s][a] = [(1.0, s, 0, True)]
                    continue

                for a in range(self.nA):
                    transitions = []
                    for i, offset in enumerate([-1, 0, 1]):
                        a_eff = (a + offset) % 4
                        dr, dc = directions[a_eff]
                        nr, nc = r + dr, c + dc
                        if 0 <= nr < self.nrow and 0 <= nc < self.ncol:
                            ns = self._to_state(nr, nc)
                            next_cell = self.map[nr][nc]
                        else:
                            ns = s
                            next_cell = cell

                        reward = 1.0 if next_cell == 'G' else 0.0
                        done = next_cell in "GH"
                        transitions.append((slip_probs[i], ns, reward,
                           done))
                    P[s][a] = transitions
        return P
```

Explanation:

- For each state and action, computes possible transitions considering stochastic "slippery" movement (intended direction 80%, left/right 10% each).
- If the agent steps into a hole or the goal, the episode ends.

2.4 Reset, Step, and Render Methods

```
[title={Reset, Step, and Render Methods}]
    def reset(self, seed=None, options=None):
        self.state = self._to_state(0, 0)
        return self.state, {}

    def step(self, action):
        transitions = self.P[self.state][action]
        i = np.random.choice(len(transitions), p=[t[0] for t in
           transitions])
        _, s_prime, reward, done = transitions[i]
        self.state = s_prime
        return s_prime, reward, done, False, {}

    def render(self):
        if self.window is None:
            pygame.init()
            self.window = pygame.display.set_mode((self.window_size, self
               .window_size))
        if self.clock is None:
            self.clock = pygame.time.Clock()

        colors = {'S': (0, 255, 0), 'F': (180, 180, 255), 'H': (0, 0, 0),
            'G': (255, 215, 0)}
        self.window.fill((255, 255, 255))
        for r in range(self.nrow):
            for c in range(self.ncol):
                cell = self.map[r][c]
                color = colors.get(cell, (200, 200, 200))
                pygame.draw.rect(self.window, color, pygame.Rect(
                    c * self.cell_size, r * self.cell_size, self.
                       cell_size, self.cell_size))

        r, c = self._to_pos(self.state)
        pygame.draw.circle(self.window, (255, 0, 0), (
            c * self.cell_size + self.cell_size // 2, r * self.cell_size
               + self.cell_size // 2), self.cell_size // 3)

        pygame.display.update()
        self.clock.tick(self.metadata["render_fps"])

    def close(self):
        if self.window:
            pygame.quit()
```

Explanation:

- `reset`: Resets to the starting state.
- `step`: Samples the next state according to the transition probabilities.
- `render`: Uses Pygame to visually display the board and agent.

4.3 3. Expanded FrozenLake Environment (10x10)

```
[title={Custom 10x10 FrozenLake Environment}]
class CustomFrozenLakeExpanded(CustomFrozenLake):
    def __init__(self, render_mode=None):
        super().__init__(render_mode)
        self.map = [
            "SFFFFFFFFF",
            "FFFFFFFFFF",
            "FFFHFFFFFF",
            "FFFFFHFFFF",
            "FFFHFFFFFH",
            "FHHFFFFFHF",
            "FHFFFHFHFF",
            "FFFFHFFFFF",
            "FFFFFHFFFF",
            "FFFFFFFFFG"
        ]
        self.setup_environment()
        self.cell_size = self.window_size // self.nrow
```

Explanation:

- Inherits from the 8x8 class but uses a 10x10 grid with more holes and a longer path to the goal.

4.4 4. Environment Registration

```
[title={Environment Registration}]
register(id='CustomFrozenLake8x8-v0', entry_point=__name__ + ':
   CustomFrozenLake', kwargs={'render_mode': None})
register(id='CustomFrozenLake10x10-v0', entry_point=__name__ + ':
   CustomFrozenLakeExpanded', kwargs={'render_mode': None})
```

4.5 5. Policy Iteration (with Timing)

```
[title={Policy Iteration with Timing}]
def timed_policy_iteration(env, gamma=0.99, theta=1e-8):
    nS, nA = env.observation_space.n, env.action_space.n
    policy = np.zeros(nS, dtype=int)
    V = np.zeros(nS)
    iterations = 0
    start_time = time.time()

    while True:
        # Policy Evaluation
        while True:
            delta = 0
            for s in range(nS):
                a = policy[s]
                v = V[s]
                V[s] = sum(p * (r + gamma * V[s_]) for p, s_, r, d in env
                    .P[s][a])
                delta = max(delta, abs(v - V[s]))
            if delta < theta:
                break

        # Policy Improvement
        policy_stable = True
        for s in range(nS):
            old_action = policy[s]
            action_values = np.array([
                sum(p * (r + gamma * V[s_]) for p, s_, r, d in env.P[s][a
                    ])
                for a in range(nA)
            ])
            best_action = np.argmax(action_values)
            policy[s] = best_action
            if best_action != old_action:
                policy_stable = False

        iterations += 1
        if policy_stable:
            break

    duration = time.time() - start_time
    return policy, V, iterations, duration
```

Explanation:

- Implements policy iteration: alternates between policy evaluation and improvement until convergence.
- Tracks the number of improvement steps and wall-clock time for convergence.

4.6 6. Policy Evaluation Function

```
[title={Policy Evaluation Function}]
def evaluate_policy(env, policy, episodes=100):
    total_rewards = []
    steps = []

    for _ in range(episodes):
        obs, _ = env.reset()
        done = False
        total_reward = 0
        step_count = 0

        while not done:
            action = policy[obs]
            obs, reward, done, _, _ = env.step(action)
            total_reward += reward
            step_count += 1

        total_rewards.append(total_reward)
        steps.append(step_count)

    avg_reward = np.mean(total_rewards)
    avg_length = np.mean(steps)
    success_rate = np.mean(total_rewards) * 100

    return {
        "Average Reward": avg_reward,
        "Average Episode Length": avg_length,
        "Success Rate (%)": success_rate
    }
```

Explanation:

- Runs the learned policy for multiple episodes.

- Tracks total rewards, steps per episode, and computes metrics: average reward, average episode length, and percentage of successful episodes.

4.7 7. Visual Playback Function

```
[title={Visual Playback Function}]
def run_visual_policy(env, policy, delay=0.5):
    obs, _ = env.reset()
    done = False
    while not done:
        env.render()
        action = policy[obs]
        obs, reward, done, _, _ = env.step(action)
        time.sleep(delay)
    env.render()
    time.sleep(2)
    env.close()
```

Explanation:

- Visually runs the agent through the environment using the learned policy, with a delay for human viewing.
- Renders each step and pauses at the end before closing the window.

4.8 8. Main Execution: Running and Comparing Both Environments

```
[title={Main Execution Block}]
if __name__ == "__main__":
    print("8x8 Frozen Lake")
    env8 = CustomFrozenLake(render_mode="human")
    policy8, V8, iter8, dur8 = timed_policy_iteration(env8)
    print(f"Policy Iteration on 8x8: {iter8} improvement steps, {dur8:.3f
       } seconds")
    eval_metrics8 = evaluate_policy(env8, policy8)
    print("Evaluation Metrics (8x8):")
    for k, v in eval_metrics8.items():
        print(f"{k}: {v}")
    run_visual_policy(env8, policy8)

    print("\n10x10 Frozen Lake")
    env10 = CustomFrozenLakeExpanded(render_mode="human")
    policy10, V10, iter10, dur10 = timed_policy_iteration(env10)
    print(f"Policy Iteration on 10x10: {iter10} improvement steps, {dur10
       :.3f} seconds")
    eval_metrics10 = evaluate_policy(env10, policy10)
    print("Evaluation Metrics (10x10):")
    for k, v in eval_metrics10.items():
        print(f"{k}: {v}")
    run_visual_policy(env10, policy10)
```

Explanation:

- Runs policy iteration and evaluation on both the 8x8 and 10x10 custom FrozenLake environments.
- Prints convergence metrics (steps, time), evaluation metrics (reward, episode length, success rate), and visually displays the agent's optimal path for each environment.

5 Comparison of Metrics Across FrozenLake Environments

Metric	Original (4x4)	Custom (8x8)	Expanded (10x10)
Policy Improvement Steps	Low (5–15)	Moderate (15–40)	High (40–100)
Convergence Time (s)	<1	1–3	3–10
Average Episode Length	10–20	20–50	40–80
Success Rate (%)	70–95	30–60	10–40
Average Reward per Episode	0.7–0.95	0.3–0.6	0.1–0.4
Variance in Episode Length	Low	Moderate	High

Metric Explanations

- **Policy Improvement Steps:**
  The number of times the policy is updated before reaching stability. More complex environments require more improvement steps as the agent needs to discover and refine better paths.

- **Convergence Time (s):**
  The actual wall-clock time for policy iteration to converge. Larger grids with more states and transitions take longer to process.

- **Average Episode Length:**
  The mean number of steps per episode when following the learned policy. In more complex or larger environments, it generally takes more steps to reach the goal (if reached at all).

- **Success Rate (%):**
  The percentage of episodes where the agent successfully reaches the goal. This typically decreases as the environment becomes larger and more hazardous (more holes, longer paths).

- **Average Reward per Episode:**
  The mean reward collected per episode. Since reward is usually only given for reaching the goal, this metric closely tracks the success rate.

- **Variance in Episode Length:**
  Indicates how consistent the agent's paths are. Low variance means the agent reliably finds similar-length paths; high variance suggests frequent failures or wandering.

Summary:

As the FrozenLake environment increases in size and complexity (from 4x4 to 10x10), all metrics indicate greater difficulty: convergence takes longer, the agent requires more steps to reach the goal, and the likelihood of success drops significantly.