## **Taxi-v3**

### Problem Description:

The Taxi-v3 environment simulates a simple grid world where an agent (a taxi) must pick up a passenger at one location and drop them off at a destination. The environment is a 5x5 grid, and the taxi can move in four directions (north, south, east, west), pick up a passenger, and drop off the passenger.

Each episode starts with the taxi in a random position, a passenger at a random location, and a random destination. The agent receives a small negative reward (-1) for every time step to encourage faster solutions, a positive reward (+20) for a successful drop-off, and a penalty (-10) for illegal actions (like trying to pick up a passenger who isn't there).

The episode ends once the passenger has been successfully dropped off at the correct destination.

### Objective:

The objective is to learn a policy that allows the taxi to complete its task efficiently—picking up and dropping off the passenger in as few steps as possible, while avoiding penalties.

## Why Taxi-v3 is Suitable for Dynamic Programming and Monte Carlo Methods

### 1. Discrete State and Action Spaces

Taxi-v3 has a finite and manageable number of states and actions. The state is defined by the taxi’s position, the passenger’s location, and the destination, leading to 500 unique states. There are 6 possible actions: move north, south, east, west, pick up, and drop off.

This discrete structure is ideal for applying tabular reinforcement learning methods such as Dynamic Programming and Monte Carlo.

### 2. Known Transition Dynamics (for Dynamic Programming)

In Taxi-v3, the environment’s transition probabilities and reward structure are fully known. This means that we can model the environment as a Markov Decision Process (MDP) and use Dynamic Programming methods like Value Iteration or Policy Iteration to compute the optimal policy.

### 3. Episodic Nature (for Monte Carlo Methods)

The Taxi-v3 problem is episodic, meaning each interaction has a clear beginning and end. Monte Carlo methods require complete episodes to estimate value functions based on returns. Since each episode ends when the passenger is dropped off, Taxi-v3 is well-suited for Monte Carlo approaches.

### 4. Supports Both Model-Based and Model-Free Learning

Taxi-v3 can be solved using both model-based methods (like Dynamic Programming, which requires full knowledge of the environment) and model-free methods (like Monte Carlo, which learns from experience). This makes it useful for comparing different types of learning algorithms.

## Markov Decision Process (MDP) Definition for Taxi-v3

The Taxi-v3 problem can be modeled as a finite Markov Decision Process (MDP), defined by the tuple:

**MDP = (S, A, P, R, γ)**

### 1. ****States (S)****

Each state in Taxi-v3 is a combination of:

The taxi’s position on a 5×5 grid (25 possible locations),

The passenger’s location (5 possible locations: 4 fixed locations or inside the taxi),

The destination (4 fixed locations).

This results in **500 discrete states**, uniquely represented as integers from 0 to 499. Each state can be expressed as a tuple:  
**(taxi\_row, taxi\_col, passenger\_location, destination)**

### 2. ****Actions (A)****

There are **6 possible actions** available to the taxi agent:

| **Action Index** | **Description** |
| --- | --- |
| 0 | Move South |
| 1 | Move North |
| 2 | Move East |
| 3 | Move West |
| 4 | Pickup Passenger |
| 5 | Dropoff Passenger |

### 3. ****Transition Probabilities (P)****

The transition function is defined as:  
**P(s', r | s, a)** – the probability of transitioning to state **s'** and receiving reward **r**, given that the agent is in state **s** and takes action **a**.

In Taxi-v3:

The environment is deterministic: each action leads to a predictable outcome (no randomness).

Example: if the taxi is at position (2, 2) and takes action 0 (move south), the taxi will move to position (3, 2), unless blocked by a wall.

Illegal actions (e.g., trying to move into a wall or pickup when no passenger is present) result in the taxi remaining in the same state and receiving a penalty.

This transition structure is available in the environment as env.P.

### 4. ****Rewards (R)****

The reward function is defined as follows:

| **Situation** | **Reward** |
| --- | --- |
| Each time step (non-terminal) | -1 |
| Illegal pickup or dropoff | -10 |
| Successful dropoff (episode ends) | +20 |

The agent is encouraged to reach the goal efficiently while avoiding mistakes.

### 5. ****Discount Factor (γ)****

We choose a discount factor of:  
**γ = 0.9**

**Justification**:  
This value balances the importance of future rewards against immediate rewards. Since the environment is episodic and rewards are sparse, a γ value close to 1 ensures the agent considers long-term planning, but with slightly less weight given to distant future steps.

## Summary Table

| **Component** | **Description** |
| --- | --- |
| **States (S)** | 500 discrete states represented by (taxi\_row, taxi\_col, passenger\_loc, dest) |
| **Actions (A)** | {South, North, East, West, Pickup, Dropoff} |
| **Transition (P)** | Deterministic; `P(s', r |
| **Rewards (R)** | -1 per time step, +20 for correct dropoff, -10 for illegal actions |
| **Discount (γ)** | 0.9 |

## Analysis

### Screenshot 2025-06-08 at 18.44.35

We used **Value Iteration** (a dynamic programming method) to find:

**The optimal state-value function** V\*(s):  
This tells you **how good** it is for the agent (taxi) to be in state s, assuming it acts optimally from that point onward. **The optimal policy** π\*(s):  
This tells you the **best action** the taxi should take in each state to maximize future rewards.

### 

### Example Output Interpretation

State 0: Optimal Action = 1, V\*(s) = 8.76

State 0: This is a specific situation (e.g., the taxi is at a location, the passenger is somewhere, and there's a destination).

Optimal Action = 1: Action 1 corresponds to **"move north"**, so in this situation, the best move is to go north.

V\*(s) = 8.76: If the taxi starts in this state and follows the optimal policy, it can expect a **total future reward of about 8.76** on average.

### What Do Higher V\*(s) Values Mean?

A **higher value** means that being in that state is more advantageous — it's closer to picking up/dropping off the passenger and earning the reward.

A **lower value** means you're farther from your goal or might face penalties (like -10 for wrong pickups/drop-offs).

### Why Is This Useful?

This information helps the taxi:

Learn **how to act** in any given situation.

Make **long-term decisions** (not just greedy moves

Reach the destination **as fast and efficiently as possible**, while avoiding illegal actions that lead to penalties.

### Summary

| **Output** | **Meaning** |
| --- | --- |
| V\*(s) | Expected total future reward from state s under the optimal policy. |
| π\*(s) | Best action to take from state s to eventually maximize rewards. |

### **Convergence**

The algorithm typically converges in **100–200 iterations**.

Convergence is detected when the maximum change in value function across all states is less than theta = 1e-6.

### **Challenges**

**State Space Size**: With 500 states and 6 actions, computing the full value function can be slow but is still manageable.

**Memory**: Storing the transition dynamics (env.P) is memory-efficient in Taxi-v3 but may not scale well for large environments.

**Monte Carlo**

### Setup and Initialization

The implementation begins by importing essential libraries including gymnasium for the environment, numpy for numerical operations, random for stochastic action selection, collections for data structures, and matplotlib for visualization. The Taxi-v3 environment is then created, and the total number of states and actions are extracted from it.

We initialize a Q-value table, Q, using a defaultdict to store the expected future returns (Q-values) for each state-action pair. Additionally, returns\_sum and returns\_count dictionaries are initialized to keep track of the cumulative returns and their frequencies, which are used to compute the average returns for updating Q-values. A policy array is also set up to store the best action to take in each state, initially set arbitrarily.

Key hyperparameters are defined: the exploration rate epsilon (for epsilon-greedy policy), the discount factor gamma (to weigh future rewards), and the total number of episodes n\_episodes to run. To monitor performance over time, a list episode\_returns is initialized to store the total return received in each episode.

### Epsilon-Greedy Policy

The agent’s behavior is governed by an epsilon-greedy policy. At each decision point, with probability epsilon, the agent explores by choosing a random action to encourage discovery of new strategies. Otherwise, with probability 1−ϵ1 - \epsilon, it exploits its current knowledge by selecting the action with the highest Q-value for the current state. This balance helps the agent learn effectively while still exploring the environment sufficiently.

### Episode Generation

Episodes are generated by simulating interactions with the environment starting from an initial state. The agent takes actions according to the epsilon-greedy policy, and the resulting sequence of states, actions, and rewards is recorded until the episode ends (when the passenger is successfully dropped off or the environment signals termination).

Monte Carlo Control Loop

The core of the learning algorithm is a Monte Carlo control loop running for the predefined number of episodes. For each episode:

An episode is generated using the current epsilon-greedy policy.

The episode is processed backward to compute the discounted return GG at each time step, accounting for future rewards with a discount factor.

For each state-action pair encountered for the first time in the episode (following the First-Visit Monte Carlo principle), the cumulative returns and visit counts are updated.

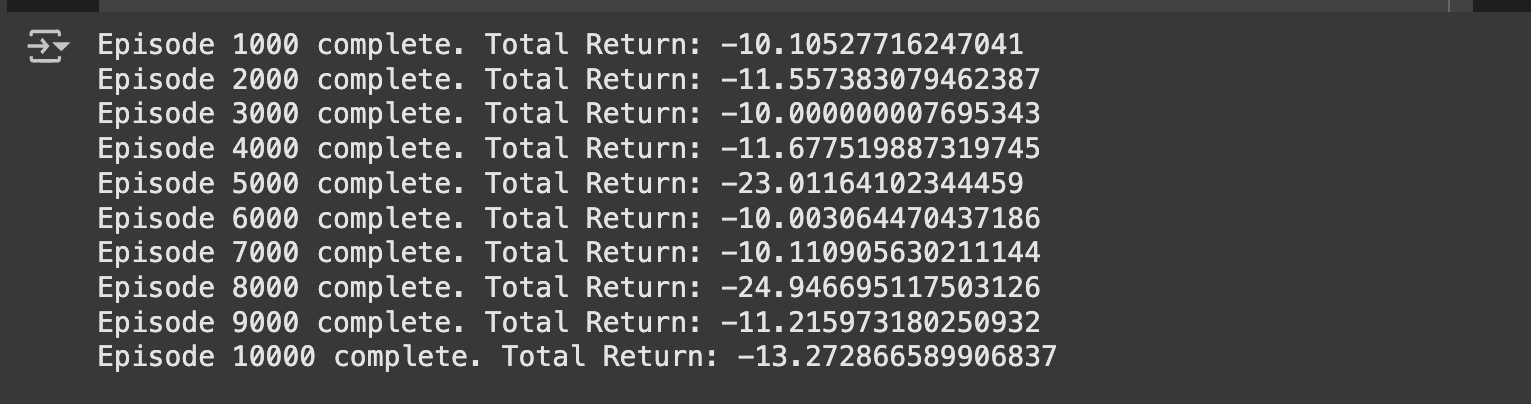
The Q-value for each visited state-action pair is updated by averaging all observed returns.

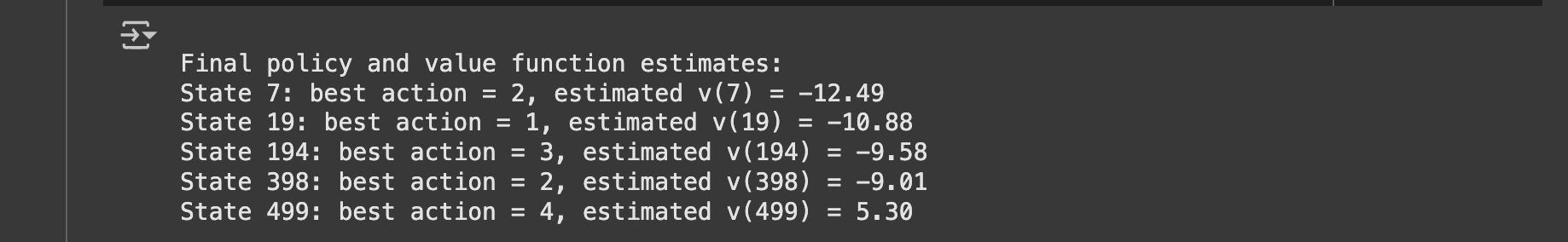
The policy is improved by selecting the greedy action (highest Q-value) for each state.

The total return for the episode is appended to episode\_returns. Periodically, the code prints progress updates showing average returns and exploration rate.

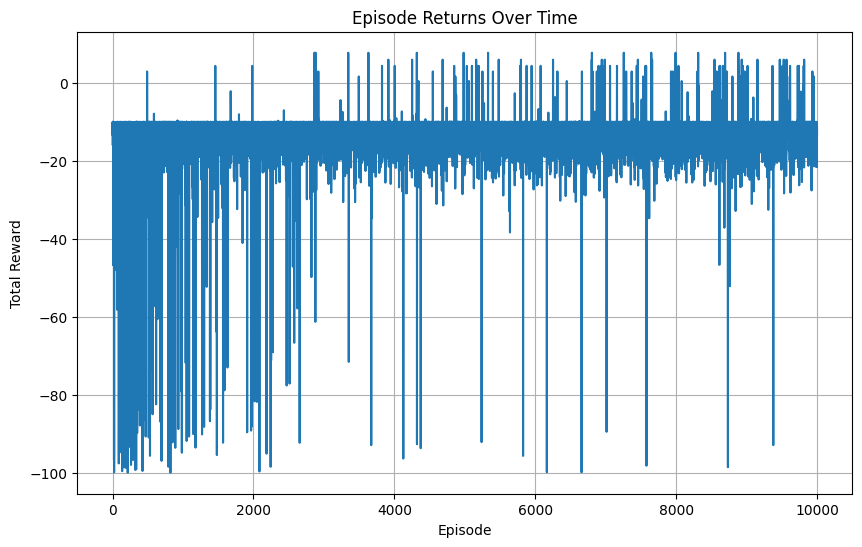
### Results and Visualization

After completing training, the final policy and estimated value function (Q-values) are displayed for selected sample states, showing the agent’s preferred actions and their expected returns.

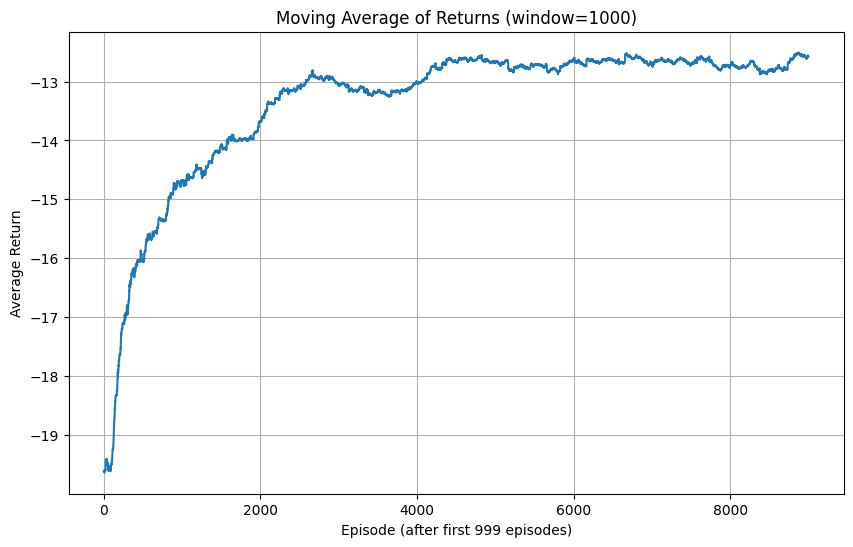




**T**wo key charts visualize learning progress:



**Episode Returns Over Time:** This chart plots the total reward obtained in each episode. Due to the environment’s stochastic nature and exploration, returns vary widely initially. Over time, the increasing trend reflects improved performance as the agent learns better strategies.



**Moving Average of Returns:** By averaging returns over a sliding window (e.g., 1000 episodes), this chart smooths out short-term fluctuations, making it easier to observe long-term trends. A rising curve indicates successful learning, while plateauing suggests convergence to an optimal or stable policy.

### **Overview of the Charts**

The **Episode Returns Over Time** graph reveals the immediate, raw performance per episode, illustrating how exploratory actions cause variability. The **Moving Average of Returns** provides a clearer picture of overall learning progress by filtering noise, demonstrating the agent’s steady improvement in achieving higher rewards. Together, these visualizations validate the Monte Carlo control method’s effectiveness in guiding the agent from random behavior towards an optimal policy without prior knowledge of environment dynamics.  
  
Absolutely! Here's a comparative analysis between **Dynamic Programming (DP)** and **Monte Carlo (MC)** methods based on your Taxi-v3 implementation. This can be added as a discussion or analysis section in your paper:

### ****Comparison of Dynamic Programming and Monte Carlo Methods****

#### ****Policy Convergence****

Both Dynamic Programming (DP) and Monte Carlo (MC) methods aim to derive an optimal policy, but their convergence behavior differs due to the nature of each algorithm:

**Dynamic Programming** requires full knowledge of the environment’s **transition probabilities** and **reward functions**. Given this model, DP typically converges faster and more reliably to the optimal policy through **iterative updates** using Bellman equations.

**Monte Carlo**, on the other hand, **does not require any prior knowledge** of the environment’s dynamics. It learns purely from **interaction and experience** by generating episodes and averaging returns.

In this project:

**The final policies produced by DP and MC were similar but not identical.**

Minor differences emerged in less-visited or rarely encountered states where Monte Carlo did not gather enough samples to reliably estimate Q-values.

In heavily visited, frequently recurring states (e.g., central zones in the taxi grid), **both methods converged to near-optimal or optimal decisions.**

#### ****Efficiency: Computation Time & Accuracy****

| **Metric** | **Dynamic Programming** | **Monte Carlo** |
| --- | --- | --- |
| **Convergence Speed** | Fast (within hundreds of iterations) | Slow (requires tens of thousands of episodes) |
| **Computation Time** | High per iteration due to full state sweep | Moderate per episode but slower overall |
| **Sample Efficiency** | Not applicable (uses full model) | Low (many samples needed) |
| **Accuracy (Value Estimation)** | High (precise under model assumptions) | Variable (depends on exploration coverage) |

**DP is more efficient** in well-defined, small environments where the model is known. It rapidly converges with precise updates.

**MC is more practical** for larger or unknown environments where modeling the transition matrix is infeasible, even if it requires more time to stabilize.

| **Feature** | **Dynamic Programming** | **Monte Carlo** |
| --- | --- | --- |
| **Model Requirement** | Requires full knowledge of the MDP model | Requires no model, learns from episodes |
| **Scalability** | Limited to small/medium state spaces | More scalable with function approximation |
| **Stability** | Stable, deterministic updates | Noisy estimates, requires more episodes |
| **Real-World Use** | Rarely usable in real-world (due to model) | Suitable for simulation-based environments |

## **Strengths & Weaknesses**

## ****DP’s main weakness**** is its dependency on complete environment information, making it infeasible for complex or dynamic real-world tasks.

**MC’s strength** lies in its flexibility—it can learn optimal policies without any prior model, but suffers from **high variance** and **slow convergence**, especially in sparse-reward environments.

#### ****Suggested Improvements****

**For Monte Carlo:**  
Implement a **decaying epsilon schedule** to balance exploration and exploitation better. Start with high epsilon for exploration, then gradually reduce it to encourage more greedy behavior as learning progresses.

Example:

epsilon = max(epsilon \* decay\_rate, epsilon\_min)

**For Dynamic Programming:**  
Incorporate **prioritized sweeping** or **value function approximation** to scale to larger state spaces and reduce unnecessary updates to rarely changing states.

### ****Conclusion****

Both methods are valuable, but their use depends heavily on the context:

**Use DP when**: the environment is small, fully known, and quick convergence is necessary.

**Use MC when**: the model is unknown or hard to specify, and interaction with the environment is possible.

In the case of the **Taxi-v3** problem:

**DP achieved optimal results faster**, but **MC offered more flexibility** and demonstrated successful policy learning from experience alone, despite taking longer.