Maze-Solving Problem with Basic Reinforcement Learning

Course: Basis Reinforcement Learning - EEE703116-1-1-24(N01)

Instructor: Vu Hoang Dieu

Student: Nguyen Tien Minh

October 10, 2024

Contents

Chapter 1

Introduction

1.1 Overview

Reinforcement Learning (RL) is a branch of machine learning where an agent learns how to interact with an environment by taking actions and observing the results. Unlike supervised learning, where labeled data is provided, the agent in RL learns to optimize its behavior by receiving rewards or penalties from the environment. This makes RL a powerful tool in decision-making systems such as robotics, game AI, and autonomous systems. The importance of RL lies in its ability to solve complex problems through trial and error, helping to find the best strategy to achieve a desired goal.

1.2 Problem Description

In this project, I focus on solving a classic RL problem: the maze-solving problem. The agent is placed in a maze and must find its way out by moving through square cells step by step. The agent's goal is to find the shortest path from the start position to the goal. The maze environment contain walls , and the agent must learn how to navigate through different states to achieve its objective.

1.3 Objective

The primary objective of this project is to implement and compare various traditional RL algorithms, such as Q-Learning, SARSA, Policy Iteration, Value Iteration, and Monte Carlo, to solve the maze-solving problem. I will evaluate these algorithms based on their performance in solving the problem, convergence speed, and the quality of the optimal path.

1.4 Structure of Report

This report is structured as follows:

- Theoretical background, introducing core RL concepts and algorithms.
- Methodology, where we describe the implementation details of the RL problem.

- Experimental setup, detailing the environment and parameters.
- Results, presenting the performance of the algorithms.
- Discussion, analyzing the results and challenges encountered.
- Conclusion, summarizing key findings and suggestions for future work.

Chapter 2

Theoretical Background

2.1 Core Concepts of Reinforcement Learning

Reinforcement Learning (RL) is built on the interaction between an **agent** and an **environment**. The agent takes **actions** based on its current **state**, aiming to maximize the cumulative **reward** it receives from the environment over time. The **policy** defines the agent's behavior, mapping states to actions. RL operates under the framework of a **Markov Decision Process (MDP)**, which assumes that the future state depends only on the current state and action, not on previous states or actions.

- **Agent**: The decision-maker that interacts with the environment.
- **Environment**: The system the agent operates within, where states, rewards, and transitions are defined.
- **State**: A representation of the environment's current situation.
- **Action**: A choice the agent can make at each state.
- **Reward**: A scalar value given to the agent after each action, indicating how good or bad that action was.
- **Policy**: A strategy used by the agent to decide which action to take at each state.

A critical concept in RL is the trade-off between **exploration** and **exploitation**. The agent needs to explore the environment to discover better actions, but also exploit the knowledge it has gained to maximize rewards. This balance is key to learning an optimal policy.

2.2 Overview of Algorithms

2.2.1 Q-Learning

Q-Learning is a **model-free, off-policy** RL algorithm. It aims to learn the optimal action-value function $Q(s, a)$, which represents the expected future rewards when taking action a in state s and following the optimal policy thereafter. Q-Learning is called off-policy because it learns the value of the optimal policy, regardless of the actions taken by the current policy. The Q-value update rule for Q-Learning is given by:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

where:

- α is the learning rate,
- γ is the discount factor,
- r is the reward received after taking action a from state s,
- s' is the next state,
- a' is the next action chosen.

2.2.2 SARSA

SARSA (State-Action-Reward-State-Action) is an **on-policy** RL algorithm, meaning that it learns the value of the policy that the agent is currently following. Unlike Q-Learning, which uses the best possible future action for updates, SARSA updates its Q-values based on the actual action taken by the policy. The update rule for SARSA is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right]$$

Here, a' is the action chosen by the current policy for the next state s', not necessarily the optimal action. This makes SARSA more conservative compared to Q-Learning but also more stable in some environments.

2.2.3 Policy Iteration

Policy Iteration is a **model-based** RL algorithm where the policy is explicitly learned and improved iteratively. It involves two main steps:

1. **Policy Evaluation**: Calculate the value function $V^\pi(s)$ for a given policy π, which is the expected cumulative reward starting from state s and following policy π.
2. **Policy Improvement**: Update the policy π by selecting actions that maximize the expected value $V^\pi(s)$ for each state.

The process repeats until the policy converges to the optimal policy.

2.2.4 Value Iteration

Value Iteration is similar to Policy Iteration but combines the policy evaluation and improvement steps into a single update. Rather than explicitly calculating the value function for a fixed policy, Value Iteration updates the value of each state using the Bellman optimality equation:

$$V(s) \leftarrow \max_a \left[r + \gamma \sum_{s'} P(s'|s,a) V(s') \right]$$

where $P(s'|s,a)$ is the transition probability from state s to state s' after taking action a. Value Iteration is faster than Policy Iteration in some cases but requires more frequent updates.

2.2.5 Monte Carlo Methods

Monte Carlo methods are **model-free** algorithms that estimate value functions based on actual returns (rewards) from complete episodes of interaction with the environment. Instead of updating after every action, Monte Carlo methods wait until an episode finishes and then update the value of states or actions based on the total rewards accumulated during the episode. Monte Carlo methods can work in environments where the transition probabilities are not known.

The update rule for Monte Carlo is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[G_t - Q(s,a)\right]$$

where G_t is the return (total reward) from time step t to the end of the episode. The return is computed as:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$$

Here, γ is the discount factor and r_{t+k+1} is the reward received at each subsequent time step.

2.3 Equations and Updates

2.3.1 Q-Learning Update Rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$$

2.3.2 SARSA Update Rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a)\right]$$

2.3.3 Policy Iteration

1. **Policy Evaluation**:

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

2. **Policy Improvement**:

$$\pi(s) = \arg\max_{a} \sum_{s'} P(s'|s,a) \left[r(s,a,s') + \gamma V(s')\right]$$

2.3.4 Value Iteration

$$V(s) \leftarrow \max_{a} \sum_{s'} P(s'|s,a) \left[r(s,a,s') + \gamma V(s')\right]$$

2.3.5 Monte Carlo Update

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[G_t - Q(s,a)\right]$$

where G_t is the return (total reward) from time step t to the end of the episode.

Chapter 3

Methodology

3.1 Problem Definition

The problem we are tackling is the **maze-solving problem**, where an agent navigates through a maze to find the exit. The agent must learn an optimal path using reinforcement learning (RL). The maze is represented as a grid, where each cell is either a passable space or a wall. The agent starts from a defined position and must reach the goal (exit) while receiving rewards for correct actions and penalties for incorrect ones. The main components of the problem are:

- **State space**: Each state represents a specific location within the maze, denoted by nodes on the maze's adjacency matrix. When the agent is at a specific node, that location becomes the current state. The initial state of the agent is the maze's startNode, while the goal state is defined by the goal or sinkerNode.
- **Action space**: Actions correspond to the agent's movements from its current state (node) to adjacent states. The valid actions for each node are determined by the values in the maze's adjacency matrix. If there is a value greater than 0 between nodes, it indicates a valid action or a connection between those nodes. The get_possible_actions function returns the available actions from the current node, allowing the agent to move to connected neighboring nodes.
- **Reward structure**:The agent receives a negative reward (-0.1) for each movement, encouraging it to find the shortest path to the goal. If the agent performs an invalid action (choosing an action that has no connection from the current node), it receives a penalty of -1 to discourage invalid moves. If the agent moves more than the specified number of steps, the reward received is -10. Upon reaching the goal (sinkerNode), the agent receives a reward of 10, marking the completion of the objective.

The goal of the agent is to find an optimal policy that minimizes the number of steps needed to reach the goal.

3.2 Algorithm Selection

Several RL algorithms were considered for solving this problem, including Q-Learning, SARSA, Policy Iteration, Value Iteration, and Monte Carlo methods. The reasons for choosing these algorithms are:

- **Q-Learning**: It is a simple and efficient model-free, off-policy method suitable for environments like mazes, where the optimal policy may differ from the agent's current exploration policy.
- **SARSA**: This algorithm, being on-policy, is useful for environments where we want the agent to follow a consistent strategy during learning.
- **Policy Iteration and Value Iteration**: These model-based algorithms are chosen for their ability to explicitly compute and refine policies and value functions in structured environments.
- **Monte Carlo methods**: Useful for estimating values based on complete episodes, they are effective when the environment is deterministic, as is the case with the maze problem.

3.3 Implementation

The maze-solving problem was implemented by creating a custom environment using the Randomized Depth-First Search (DFS) algorithm to generate the maze. The steps to create the RL environment and implement the algorithms are outlined below:

3.3.1 Step-by-Step Process

1. **Maze generation**: A maze is generated using the Randomized DFS algorithm, where the grid is initialized as a series of walls, and the agent carves out paths through the grid by visiting cells and marking them as visited.
2. **Environment creation**: The maze is then used as the environment for the RL agent. The state space consists of all possible positions on the grid. Each state represents a specific location within the maze, denoted by nodes on the maze's adjacency matrix. When the agent is at a specific node, that location becomes the current state. The initial state of the agent is the maze's startNode, while the goal state is defined by the goal or sinkerNode. Actions correspond to the agent's movements from its current state (node) to adjacent states. The valid actions for each node are determined by the values in the maze's adjacency matrix. If there is a value greater than 0 between nodes, it indicates a valid action or a connection between those nodes. The get_possible_actions function returns the available actions from the current node, allowing the agent to move to connected neighboring nodes. The agent receives a negative reward (-0.1) for each movement, encouraging it to find the shortest path to the goal. If the agent performs an invalid action (choosing an action that has no connection from the current node), it receives a penalty of -1 to discourage invalid moves. If the agent moves more than the specified number of steps, the reward received is -10. Upon reaching the goal (sinkerNode), the agent receives a reward of 10, marking the completion of the objective.
3. **Algorithm implementation**: Q-Learning, SARSA, Policy Iteration, Value Iteration, and Monte Carlo methods are implemented to train the agent in the envi-

ronment. Libraries such as NumPy are used for matrix operations, and Python is used for general implementation.

4. **Training**: The agent is trained to navigate the maze using the selected algorithms, and its performance is measured in terms of the time and number of steps it takes to reach the goal.

3.3.2 Pseudo-code

The pseudo-code for the key algorithms is presented below:

Q-Learning Pseudo-code

```
Initialize Q(s, a) arbitrarily
for each episode:
    Initialize state s
    while not done:
        Choose action a using epsilon-greedy
        Take action a, observe reward r and next state s'
␣␣␣␣␣␣␣␣Q(s,␣a)␣=␣Q(s,␣a)␣+␣alpha␣*␣(r␣+␣gamma␣*␣max(Q(s', a'))␣-␣Q(s,␣a))
␣␣␣␣␣␣␣␣s␣=␣s'
```

SARSA Pseudo-code

```
Initialize Q(s, a) arbitrarily
for each episode:
    Initialize state s
    Choose action a using epsilon-greedy
    while not done:
        Take action a, observe reward r and next state s'
␣␣␣␣␣␣␣␣Choose␣next␣action␣a' using epsilon-greedy
        Q(s, a) = Q(s, a) + alpha * (r + gamma * Q(s',␣a') - Q(s, a))
        s = s'
␣␣␣␣␣␣␣␣a␣=␣a'
```

Policy Iteration Pseudo-code

```
Initialize policy and value function V(s)
repeat:
    Policy Evaluation: Compute V(s) for current policy
    Policy Improvement: For each state, update policy to choose action that
until policy converges
```

Value Iteration Pseudo-code

```
Initialize value function V(s)
for each state s:
    V(s) = max_a [r + gamma * sum P(s'|s,a) * V(s')]
    Update policy to choose action that maximizes expected return
until value function converges
```

Monte Carlo Pseudo-code

```
Initialize Q(s, a) arbitrarily
for each episode:
    Generate episode using current policy
    for each state-action pair (s, a) in episode:
        G = total reward from that point to the end of episode
        Q(s, a) = Q(s, a) + alpha * (G - Q(s, a))
```

The implementations of these algorithms allow the agent to learn optimal policies for solving the maze problem by balancing exploration and exploitation strategies and refining its policy over time.

Chapter 4

Experimental Setup

4.1 Environment Setup

The environment used in this project is a maze generated using the Randomized Depth-First Search algorithm. The agent navigates through the maze by selecting actions to reach the goal. Several key hyperparameters were used across different algorithms to train the agent in the maze environment:

- **Learning rate** (α): 0.1. This defines how quickly the agent updates its knowledge of the environment.
- **Discount factor** (γ): 0.99. This determines the importance of future rewards compared to immediate rewards.
- **Exploration rate** (ϵ): Initially set to 0.1 and decayed over time. This controls the balance between exploration (choosing random actions) and exploitation (choosing the best-known action).
- **Number of episodes**: 1000 episodes were run to allow sufficient training time for the agent to learn the optimal policy.

These hyperparameters were selected after several trials, with the aim of achieving fast convergence while maintaining the balance between exploration and exploitation.

4.2 Data

The maze environment used in this experiment was simulated. Each maze was generated randomly using the **Randomized Depth-First Search** algorithm. The maze consists of a grid of size $n \times n$, where $n = 4, 8, 16 and 32$ in my experiments. Each cell in the grid represents either a passable path or a wall, and the agent starts at a fixed position and must reach the goal.

4.3 Hyperparameter Tuning

Hyperparameter tuning was conducted for each algorithm to determine the optimal settings for training the agent efficiently in the maze environment. The key hyperparameters adjusted include the learning rate (α), the discount factor (γ), and the exploration rate (ϵ).

4.3.1 Q-Learning

For the Q-Learning algorithm, the learning rate (α) was fixed at 0.1, while the exploration rate (ϵ) was varied across different values to observe its impact on learning efficiency. The discount factor (γ) was fixed at 0.99 to prioritize long-term rewards. The specific values tested were:

- $\alpha = 0.1$
- $\epsilon = \{0.1, 0.5, 0.9\}$
- $\gamma = 0.99$

The exploration rate played a key role in determining how often the agent chose random actions versus exploiting the best-known actions, with higher values of ϵ promoting more exploration.

4.3.2 SARSA

Similar to Q-Learning, the learning rate (α) was fixed at 0.1, and the exploration rate (ϵ) was varied. The discount factor (γ) was also fixed at 0.99. The exploration-exploitation balance was tested with different values of ϵ to compare how SARSA, as an on-policy algorithm, handled exploration:

- $\alpha = 0.1$
- $\epsilon = \{0.1, 0.5, 0.9\}$
- $\gamma = 0.99$

4.3.3 Monte Carlo Methods

For the Monte Carlo method, the learning rate (α) was fixed at 0.1, and the exploration rate (ϵ) was fixed at 0.5 for a balanced exploration-exploitation trade-off. In contrast to Q-Learning and SARSA, the discount factor (γ) was varied to test its impact on learning performance:

- $\alpha = 0.1$
- $\epsilon = 0.5$
- $\gamma = \{0.1, 0.9\}$

The different values of γ allowed the agent to either prioritize immediate rewards or give more weight to long-term rewards, testing the impact on overall convergence and performance.

4.3.4 Policy Iteration and Value Iteration

For Policy Iteration and Value Iteration, no explicit learning rate or exploration rate is needed, as these algorithms compute exact policies. The discount factor (γ) was varied between [0.1, 0.9, 0.99], with $\gamma = 0.99$ yielding the best performance.

Overall, each algorithm was tested under various configurations to determine the optimal set of hyperparameters, based on convergence speed and overall performance.

Chapter 5

Results

In this section, I present the results for each algorithm, including Q-Learning, SARSA, Monte Carlo, Policy Iteration, and Value Iteration. I utilize tables, charts, and graphs to display performance metrics such as cumulative rewards, the number of episodes required to converge, and the success rate. The results of algorithms are calculated base on the 8x8 maze.

5.1 Q-Learning

The Q-Learning algorithm demonstrated a gradual improvement in performance as the Q-values updated over time. The following graph illustrates the cumulative reward per episode, showing how the agent converged to an optimal policy. The values of ϵ (0.1, 0.5, and 0.9) affected the exploration-exploitation balance, with lower values favoring quicker convergence but slower exploration of the environment.

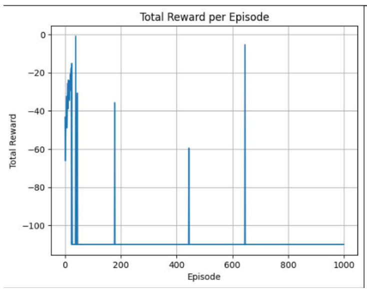

Figure 5.1: $\epsilon = 0.1$

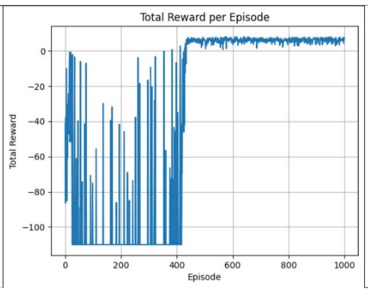

Figure 5.2: $\epsilon = 0.5$

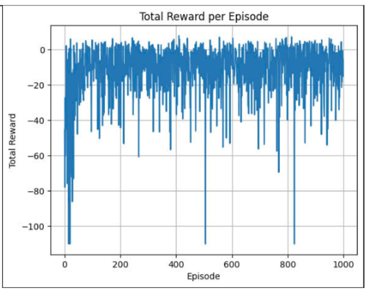

Figure 5.3: $\epsilon = 0.9$

Figure 5.4: Q-Learning with varying ϵ

Epsilon	Running Time (seconds)
0.1	2999.6
0.5	3271.5
0.9	3412.3

Table 5.1: Algorithm Running Time by Epsilon Value

5.2 SARSA

SARSA, being an on-policy algorithm, showed more stable performance compared to Q-Learning. This is attributed to its reliance on the current policy for updates, leading to oscillations in performance.

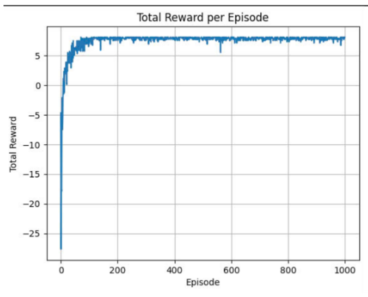

Figure 5.5: $\epsilon = 0.1$

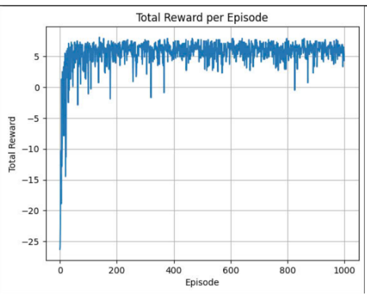

Figure 5.6: $\epsilon = 0.5$

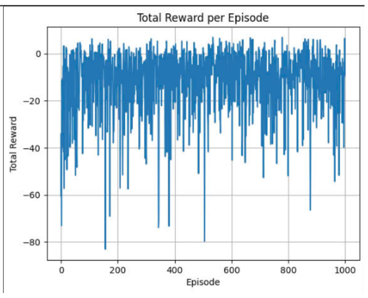

Figure 5.7: $\epsilon = 0.9$

Figure 5.8: SARSA with varying ϵ

Epsilon	Running Time (seconds)
0.1	11.5
0.5	19.3
0.9	87.8

Table 5.2: Algorithm Running Time by Epsilon Value

5.3 Monte Carlo

The Monte Carlo algorithm, using a fixed $\epsilon = 0.5$ and varying γ, showed a slower convergence rate compared to Q-Learning and SARSA. However, it performed well in environments where episodes could be simulated until completion. The results highlight how different discount factors influenced the learning process. However, the performance of this algorithm is quite poor when applied to large-sized maze environments.

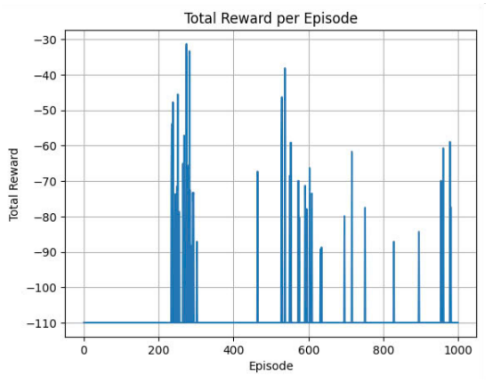

Figure 5.9: $\gamma = 0.1$

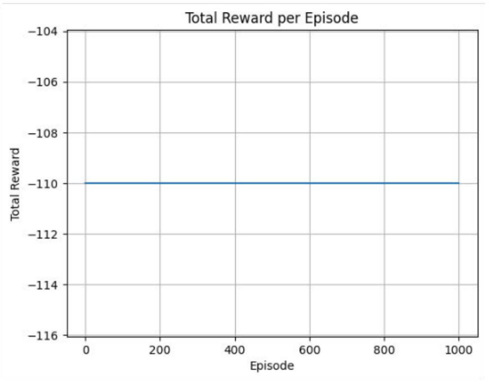

Figure 5.10: $\gamma = 0.9$

Figure 5.11: Monte Carlo with varying γ

Gamma	**Running Time (seconds)**
0.1	4510.5
0.9	1791.4

Table 5.3: Algorithm Running Time by Gamma Value

5.4 Policy Iteration

Policy Iteration showed fast convergence compared to the other methods due to its iterative approach, alternating between policy evaluation and policy improvement. It successfully found the optimal policy in fewer iterations than Q-Learning or SARSA. The result is only $\gamma = 0.99$ which gives the optimal policy.

Gamma	**Running Time (seconds)**
0.1	1.6
0.9	6.4
0.99	37.4

Table 5.4: Algorithm Running Time by Gamma Value

5.5 Value Iteration

Value Iteration, like Policy Iteration, converged quickly by iteratively updating the value function until convergence. It provided a more straightforward implementation for finding optimal policies but required careful consideration of the stopping criteria. The result is only $\gamma = 0.99$ which gives the optimal policy.

Gamma	**Running Time (seconds)**
0.1	1.71
0.9	9.5
0.99	79.3

Table 5.5: Algorithm Running Time by Gamma Value

5.6 Performance Comparison

The following table summarizes the performance of each algorithm in terms of stability, speed of convergence, and the optimal policy found.

Algorithm	**Stability**	**Speed of Convergence**	**Optimal Policy Found**
Q-Learning	Medium	Medium	Yes
SARSA	High	Fast	Yes
Monte Carlo	Low	Slow	Yes
Policy Iteration	High	Very Fast	Yes
Value Iteration	High	Very Fast	Yes

Table 5.6: Performance Comparison of RL Algorithms with 4x4 matrix

Algorithm	**Stability**	**Speed of Convergence**	**Optimal Policy Found**
Q-Learning	Medium	Medium	Yes
SARSA	High	Fast	Yes
Monte Carlo	Low	Slow	No
Policy Iteration	Low	High	Yes
Value Iteration	Low	High	Yes

Table 5.7: Performance Comparison of RL Algorithms with 8x8 matrix

Algorithm	**Stability**	**Speed of Convergence**	**Optimal Policy Found**
Q-Learning	Medium	Medium	Yes
SARSA	High	Fast	Yes
Monte Carlo	Low	Slow	No
Policy Iteration	Low	Slow	Yes
Value Iteration	Low	Slow	Yes

Table 5.8: Performance Comparison of RL Algorithms with 16x16 matrix

Algorithm	**Stability**	**Speed of Convergence**	**Optimal Policy Found**
Q-Learning	Medium	Medium	No
SARSA	High	Fast	Yes
Monte Carlo	Low	Slow	No
Policy Iteration	Low	Slow	No
Value Iteration	Low	Slow	No

Table 5.9: Performance Comparison of RL Algorithms with 32x32 matrix

Chapter 6

Discussion

6.1 Analysis of Results

The experimental results indicate that SARSA outperformed the other algorithms in terms of both runtime and convergence speed. SARSA's on-policy nature, where it updates based on the actual actions taken, led to faster convergence and more stable learning in this environment. Q-Learning, while capable of finding the optimal policy, converged more slowly as it updates based on the maximum future reward, which caused it to explore more extensively. Monte Carlo methods, on the other hand, were the slowest both in terms of convergence and computation time, requiring a large number of episodes to produce reliable value estimates.

Policy iteration and Value iteration showed impressive performance with small maze sizes, converging rapidly due to their full environment modeling and sweeping updates. However, as the maze size increased, the computational cost of these algorithms became prohibitively high, leading to significantly longer runtimes. This slowdown can be attributed to the increased complexity of the state space and the need to update values across all states in each iteration.

6.2 Challenges and Observations

One notable challenge was tuning the exploration rate (ϵ) to balance between exploration and exploitation. SARSA benefited from moderate exploration rates, while Q-Learning's performance was more sensitive to the exploration rate, leading to slower convergence when ϵ was too high. Monte Carlo methods, due to their reliance on full episode returns, required significantly more computation time and episodes to converge, especially when the maze was complex.

For Policy iteration and Value iteration, their performance was heavily dependent on the size of the maze. While these algorithms worked exceptionally well in small environments, larger mazes introduced exponential growth in computational requirements, slowing down convergence drastically. This presents a trade-off between algorithmic complexity and environment size that needs to be considered when selecting these methods.

The custom environment generated with the Randomized Depth-First Search algorithm presented additional challenges in ensuring consistent exploration across episodes. Penalty management for invalid actions also needed careful adjustment to prevent the agent from being overly penalized during exploration.

6.3 Interpretation of Convergence

The faster convergence of SARSA can be explained by its use of actual actions for updating values, allowing it to learn more quickly from the agent's immediate experience. Q-Learning's slower convergence stems from its reliance on the maximum future reward, which drives exploration but results in a longer time to refine the optimal policy. Monte Carlo methods were particularly slow due to the need to complete entire episodes before updating values, which made the process computationally expensive and time-consuming.

For Policy iteration and Value iteration, their fast convergence with small mazes comes from the ability to evaluate and improve policies across the entire state space simultaneously. However, their performance degrades significantly as the state space grows, leading to longer computation times and slower convergence in larger mazes.

Chapter 7

Conclusion

7.1 Summary

This project implemented and compared several traditional Reinforcement Learning (RL) algorithms, including Q-Learning, SARSA, Monte Carlo, Policy Iteration, and Value Iteration, in the context of solving maze navigation problems. SARSA demonstrated the fastest convergence and execution time, attributed to its on-policy nature. Q-Learning showed slower convergence due to its exploration-driven behavior, and Monte Carlo had the slowest performance both in terms of computation time and convergence rate. Policy Iteration and Value Iteration performed very efficiently in small maze environments but encountered significant slowdowns in larger mazes due to the increasing computational complexity of evaluating all states. Overall, each algorithm displayed unique strengths and limitations, making them suitable for different types of environments and tasks.

7.2 Lessons Learned

Throughout this project, several key insights were gained. First, the importance of balancing exploration and exploitation became evident, especially in SARSA and Q-Learning. Effective tuning of the exploration rate (ϵ) played a critical role in achieving faster convergence. The challenges associated with larger state spaces, as encountered with Policy Iteration and Value Iteration, highlighted the need for more scalable RL techniques in complex environments. Additionally, the computational costs of Monte Carlo methods emphasized the necessity of efficient data utilization in learning from full episodes.

The project also underscored the versatility of RL methods, which can be applied to a wide variety of tasks. The custom maze environment created using the Randomized Depth-First Search algorithm provided a flexible and dynamic setting for experimenting with different RL algorithms, revealing how RL techniques can adapt to various real-world problems where the state space and reward structures may be intricate.

7.3 Future Work

There are several potential avenues for further research and improvement. One direction would be to experiment with Deep Reinforcement Learning (DRL) methods, such as Deep

Q-Networks (DQN), to handle larger and more complex environments where traditional RL methods struggle. DRL could mitigate the computational challenges encountered with large state spaces by using neural networks for function approximation. Another way is to use quantum reinforcement learning algorithms to improve computational power.

Another potential area of improvement is to extend the maze-solving environment by introducing more complex dynamics, such as stochastic transitions, time-varying rewards, or multi-agent interactions, to explore the performance of the algorithms in more challenging scenarios. Lastly, future work could investigate the hybridization of RL algorithms, combining the strengths of different approaches to create more robust and efficient solutions for complex, real-world problems.

Bibliography

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.

[2] C. J. C. H. Watkins and P. Dayan, "Q-Learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279-292, 1992.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2009.

[4] R. E. Bellman, "A Markovian Decision Process," *Journal of Mathematics and Mechanics*, vol. 6, no. 5, pp. 679–684, 1957.

[5] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, "Randomized Depth-First Search for Maze Generation," in *Computational Geometry: Algorithms and Applications*, 3rd ed., Springer, 2008, pp. 339-343.

[6] R. E. Bellman, "Dynamic Programming," *Science*, vol. 153, pp. 34–37, Jul. 1966.