Final Project Report for Basic Reinforcement Learning

Course Name:Reinforcement Learning - Course Code: GAME MAZE

Instructor's Name: Dr VU HOANG DIEU

Student's Name: HA HUY THIEN - Student ID: 22014474
Class: K16 AI RB

Submission Date: October 11, 2024

Contents

List of Figures

1 Introduction

1.1 Overview

Reinforcement Learning (RL) is a crucial branch of artificial intelligence (AI) that focuses on how agents interact with their environment to optimize behavior through experience. In RL, an agent learns by taking actions in an environment and receiving feedback in the form of rewards. The goal of the agent is to maximize the cumulative rewards over time, thereby gradually improving its action policy.

The importance of RL lies in its ability to solve highly interactive problems without needing a predefined model of the environment. RL has been successfully applied in various fields such as robotics, video games, and automated systems, where agents need to make continuous and optimal decisions in uncertain environments.

1.2 Problem Description

The problem at hand is to create a maze game and apply the Q-Learning algorithm to help the agent learn how to escape from the maze. The maze is a grid of cells, where some cells are obstacles (walls) that the agent cannot pass through. The agent starts at a given position and must find a way to move to the goal position, optimizing its path through the learning process. The agent learns to choose movement actions (up, down, left, right) at each position to reach the goal as quickly as possible, avoiding obstacles and not getting lost.

1.3 Objective

The primary goal of this project is to help learners understand how to implement and apply the Q-Learning algorithm to solve the maze navigation problem. Specifically, by building the maze game, learners will grasp the fundamental concepts of reinforcement learning, how agents learn from their environment, and how actions are chosen based on progressively updated Q-values. Through this example, learners will also gain a deeper understanding of optimizing action policies and the mechanism of maximizing rewards in an environment with multiple choices and risks.

2 Theoretical Background

2.1 Basics of Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning where an agent learns to optimize its actions through interaction with an environment. RL is based on the principle of learning from experience, allowing the agent to make decisions in uncertain environments.

Key Components of RL:

- **Agent:** The entity that performs actions in the environment to achieve a goal. The agent learns from the feedback provided by the environment to improve its policy.
- **Environment:** The space in which the agent interacts. The environment provides information about its current state and responds to the actions taken by the agent.

- **State:** A representation of the information that the agent needs to make a decision at a specific time. The state can be the current situation in the environment.
- **Action:** The actions that the agent can take to change the state of the environment.
- **Reward:** A numerical value received by the agent after performing an action in a particular state. Rewards help the agent evaluate the quality of the actions it performs.
- **Policy:** The rule or strategy that the agent uses to choose an action based on the current state. The policy can be deterministic or stochastic.
- **Exploration vs. Exploitation:** During the learning process, the agent faces a trade-off between exploration and exploitation. Exploration involves trying out new actions to gather information about the environment, while exploitation involves using the learned knowledge to maximize rewards. Finding a balance between these two factors is crucial for achieving the best performance in reinforcement learning.

Markov Decision Process (MDP) Framework: Reinforcement Learning is often modeled using the Markov Decision Process (MDP) framework, where states and actions are defined in a finite set. An MDP is defined by:

- A set of states S
- A set of actions A
- A state transition function T that defines the probability of transitioning from one state to another when performing an action.
- A reward function R that provides rewards for each state-action pair.

2.2 Overview of Algorithms

Q-Learning: Q-Learning is a powerful off-policy reinforcement learning algorithm that allows the agent to learn from its experiences without knowing the model of the environment in advance. It uses a Q-table to store Q-values for each state-action pair, where the Q-value represents the value of taking a specific action in a specific state.

Q-value Update: The Q-value is updated using the following rule:

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

where:

- s is the current state,
- a is the action taken,
- r is the reward received,
- s' is the next state,
- α is the learning rate,

- γ is the discount factor.

SARSA: SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm. Unlike Q-Learning, SARSA updates the Q-value based on the actual action the agent has taken, not the optimal action as in Q-Learning.

Q-value Update: The Q-value is updated using the rule:

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

where a' is the action taken at state s'.

Policy Gradient Methods: Policy Gradient Methods are reinforcement learning techniques in which the agent learns to optimize the policy directly rather than estimating the Q-values. These methods update the policy by maximizing an objective function through gradient ascent.

Policy Update: This method typically uses gradient ascent to update the weights of the policy based on the rewards received, emphasizing random policy updates.

2.3 Formulas and Updates

To summarize, the update rules for the algorithms are presented as follows:

Q-Learning:

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

SARSA:

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

3 Methodology

3.1 Problem Definition

This section describes how to tackle the problem using reinforcement learning methods. Game Maze Problem Description: This involves creating a maze game where the agent navigates from a starting point to a goal. The problem can be defined as follows:
State Space: The state space consists of all possible positions the agent can occupy within the maze. Each position can be represented as a pair of coordinates (x, y) in a grid-like environment.
Action Space: The action space includes the possible movements the agent can make in the maze, typically defined as:

- Up (move up one cell)
- Down (move down one cell)
- Left (move left one cell)
- Right (move right one cell)

Reward Structure: The reward system is designed to encourage the agent to reach the goal while avoiding walls. The reward structure is defined as follows:

- Reaching the goal: +100 points
- Hitting a wall: -10 points
- Taking a step (moving): -1 point

3.2 Algorithm Selection

Algorithm Choice: For this task, Q-Learning has been selected as the primary algorithm for several reasons:

- **Model-Free Learning:** Q-Learning is a model-free algorithm, meaning it does not require a prior model of the environment. This is beneficial in maze environments where the dynamics might not be known.
- **Off-Policy Learning:** Being an off-policy algorithm allows Q-Learning to learn the optimal policy independently of the actions taken by the agent. This flexibility helps in learning from exploratory actions.
- **Simplicity and Effectiveness:** Q-Learning is conceptually simpler to implement and understand compared to other algorithms like SARSA. It provides a clear mechanism for updating the Q-values based on the maximum expected future rewards.
- **Proven Performance:** Q-Learning has been widely studied and proven effective for various reinforcement learning tasks, including maze navigation, making it a reliable choice for this project.

3.2.1 RL Environment

The RL environment in this example is a maze that the agent must navigate to reach the goal. Each cell in the maze can either be a path (0) or a wall (1). The agent will receive a reward when it reaches the goal or be penalized when it collides with a wall or moves inefficiently.

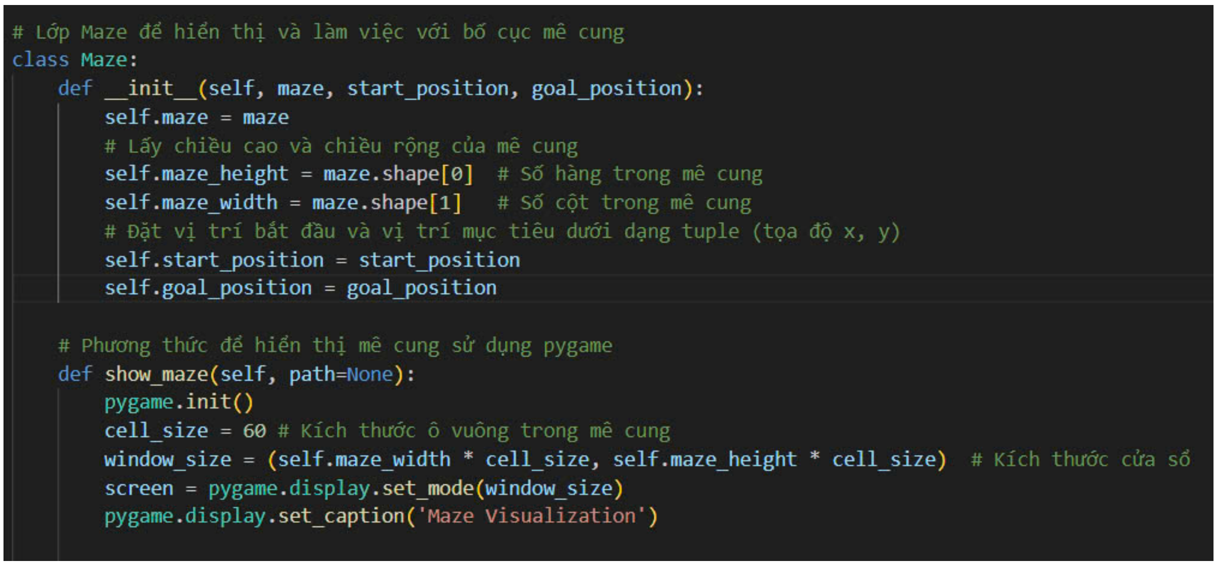
3.2.2 Libraries Used

- **NumPy:** This library is used to create and process matrices, serving to store the maze layout and the Q-table.
- **Pygame:** This library is used to display the maze and track the agent's movement within the environment.
- **Matplotlib:** This library is used to plot the rewards during the training process.

3.2.3 Implementation

The Maze class is used to display and work with the maze layout. The methods in this class include:

- **__init__:** Initializes the maze's properties, such as height, width, starting position, and goal position.



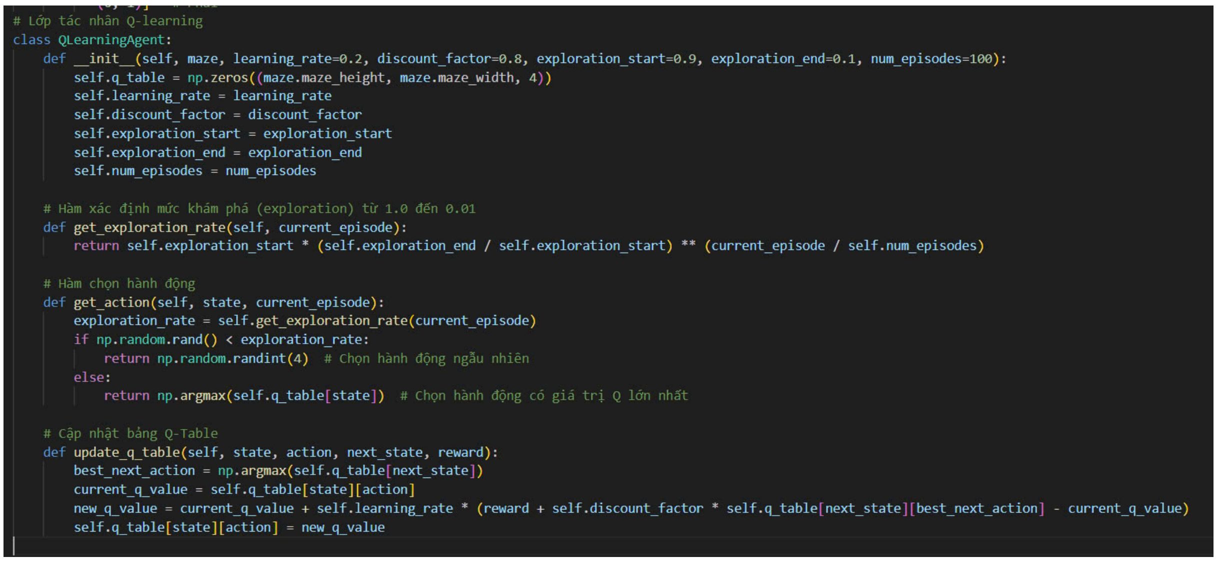
```python
# Lớp Maze để hiển thị và làm việc với bố cục mê cung
class Maze:
    def __init__(self, maze, start_position, goal_position):
        self.maze = maze
        # Lấy chiều cao và chiều rộng của mê cung
        self.maze_height = maze.shape[0]  # Số hàng trong mê cung
        self.maze_width = maze.shape[1]   # Số cột trong mê cung
        # Đặt vị trí bắt đầu và vị trí mục tiêu dưới dạng tuple (tọa độ x, y)
        self.start_position = start_position
        self.goal_position = goal_position

    # Phương thức để hiển thị mê cung sử dụng pygame
    def show_maze(self, path=None):
        pygame.init()
        cell_size = 60 # Kích thước ô vuông trong mê cung
        window_size = (self.maze_width * cell_size, self.maze_height * cell_size)  # Kích thước cửa sổ
        screen = pygame.display.set_mode(window_size)
        pygame.display.set_caption('Maze Visualization')
```

Figure 1: class Maze

- **show_maze:** Displays the maze using Pygame. It includes functions to draw the maze and draw the agent's path.

The QLearningAgent class implements the Q-learning algorithm. The main methods include:

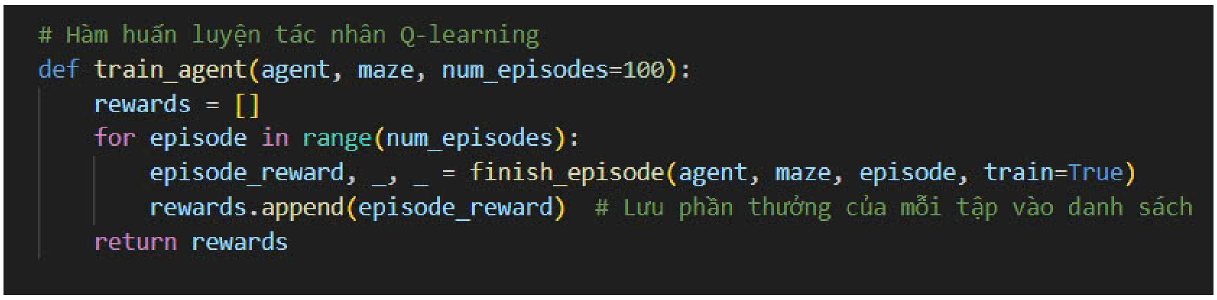
- **__init__:** Initializes the Q-table and parameters such as learning rate, discount factor, and number of training episodes.
- **get_exploration_rate:** Determines the exploration rate over episodes.
- **get_action:** Chooses an action based on the exploration rate or Q-value.
- **update_q_table:** Updates the value in the Q-table based on the received feedback.



```python
# Lớp tác nhân Q-learning
class QLearningAgent:
    def __init__(self, maze, learning_rate=0.2, discount_factor=0.8, exploration_start=0.9, exploration_end=0.1, num_episodes=100):
        self.q_table = np.zeros((maze.maze_height, maze.maze_width, 4))
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_start = exploration_start
        self.exploration_end = exploration_end
        self.num_episodes = num_episodes

    # Hàm xác định mức khám phá (exploration) từ 1.0 đến 0.01
    def get_exploration_rate(self, current_episode):
        return self.exploration_start * (self.exploration_end / self.exploration_start) ** (current_episode / self.num_episodes)

    # Hàm chọn hành động
    def get_action(self, state, current_episode):
        exploration_rate = self.get_exploration_rate(current_episode)
        if np.random.rand() < exploration_rate:
            return np.random.randint(4)  # Chọn hành động ngẫu nhiên
        else:
            return np.argmax(self.q_table[state])  # Chọn hành động có giá trị Q lớn nhất

    # Cập nhật bảng Q-Table
    def update_q_table(self, state, action, next_state, reward):
        best_next_action = np.argmax(self.q_table[next_state])
        current_q_value = self.q_table[state][action]
        new_q_value = current_q_value + self.learning_rate * (reward + self.discount_factor * self.q_table[next_state][best_next_action] - current_q_value)
        self.q_table[state][action] = new_q_value
```

Figure 2: class QLearningAgent

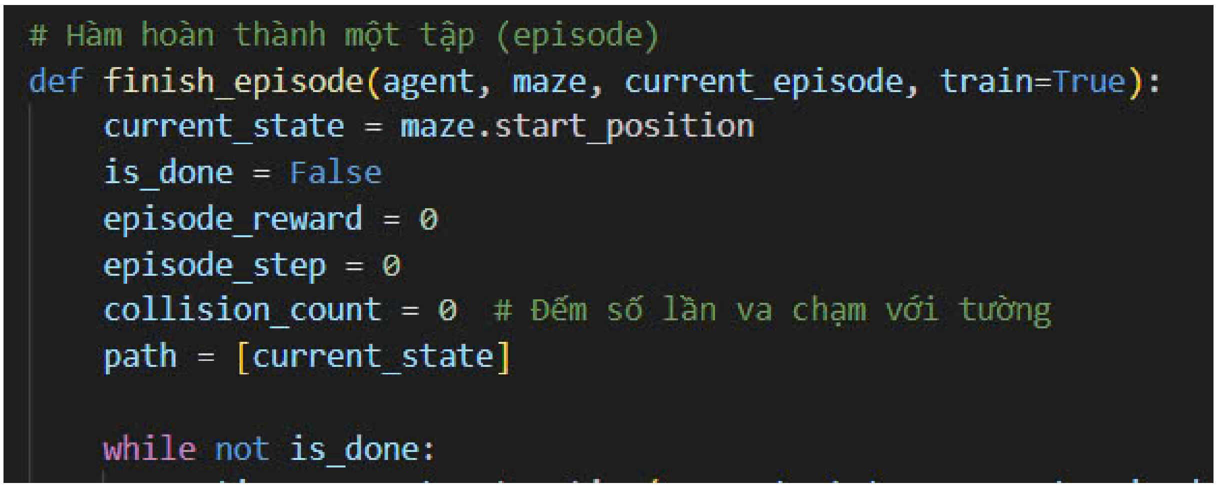
The **train_agent** function is used to train the agent. It calls the **finish_episode** function multiple times to complete the training episodes.



```python
# Hàm huấn luyện tác nhân Q-learning
def train_agent(agent, maze, num_episodes=100):
    rewards = []
    for episode in range(num_episodes):
        episode_reward, _, _ = finish_episode(agent, maze, episode, train=True)
        rewards.append(episode_reward)  # Lưu phần thưởng của mỗi tập vào danh sách
    return rewards
```

Figure 3: train agent

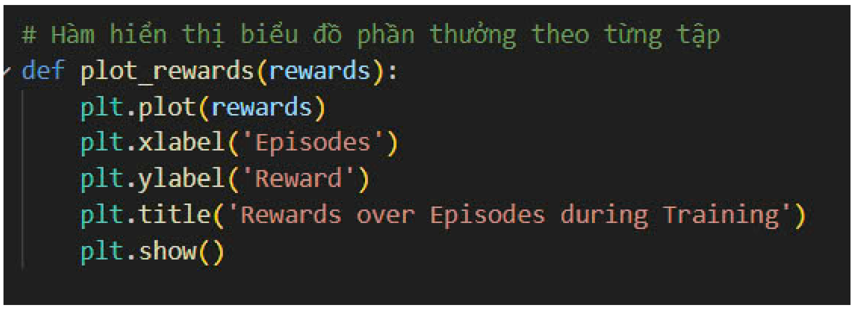
The **finish_episode** function carries out a complete episode of the training process, where the agent will choose actions, update states, and the Q-table.



```python
# Hàm hoàn thành một tập (episode)
def finish_episode(agent, maze, current_episode, train=True):
    current_state = maze.start_position
    is_done = False
    episode_reward = 0
    episode_step = 0
    collision_count = 0  # Đếm số lần va chạm với tường
    path = [current_state]

    while not is_done:
```

Figure 4: finish episode

Finally, you can use the **plot_rewards** function to display the rewards received over the episodes during the training process.



```python
# Hàm hiển thị biểu đồ phần thưởng theo từng tập
def plot_rewards(rewards):
    plt.plot(rewards)
    plt.xlabel('Episodes')
    plt.ylabel('Reward')
    plt.title('Rewards over Episodes during Training')
    plt.show()
```

Figure 5: plot reward

4 Experimental Setup

4.1 Environment Setup

Environment Used: In this experiment, a custom maze environment is created using a 2D numpy array. The maze consists of paths (0) and walls (1). The agent starts at a designated position and aims to reach the goal position while navigating through the maze.

Hyperparameters:

- **Learning Rate ():** Set to 0.1, this parameter controls how much the agent updates its Q-values based on new experiences. A higher learning rate allows the agent to learn more quickly but may lead to instability.
- **Discount Factor ():** Set to 0.9, this factor determines the importance of future rewards compared to immediate rewards. A value close to 1 prioritizes long-term rewards.
- **Exploration Rate ():** This value starts at 1.0 and decays to 0.01 over the course of training. It controls the balance between exploration (trying new actions) and exploitation (choosing actions based on known Q-values).
- **Number of Episodes:** The agent is trained over 100 episodes, allowing it to learn the optimal path to the goal through repeated interactions with the environment.

4.2 Data

Training Results
Below are the results from 100 training episodes of the Q-learning agent:

- **Average Reward:** 3.86
- **Average Steps:** 27.84

Graphs

- **Reward per Episode:** The graph shows that overall rewards stabilize after a few initial episodes.
- **Steps Taken per Episode:** The graph indicates a decrease in steps as the agent learns to find the target more effectively.

```
The average reward is: 3.86
The average steps is: 27.84
```

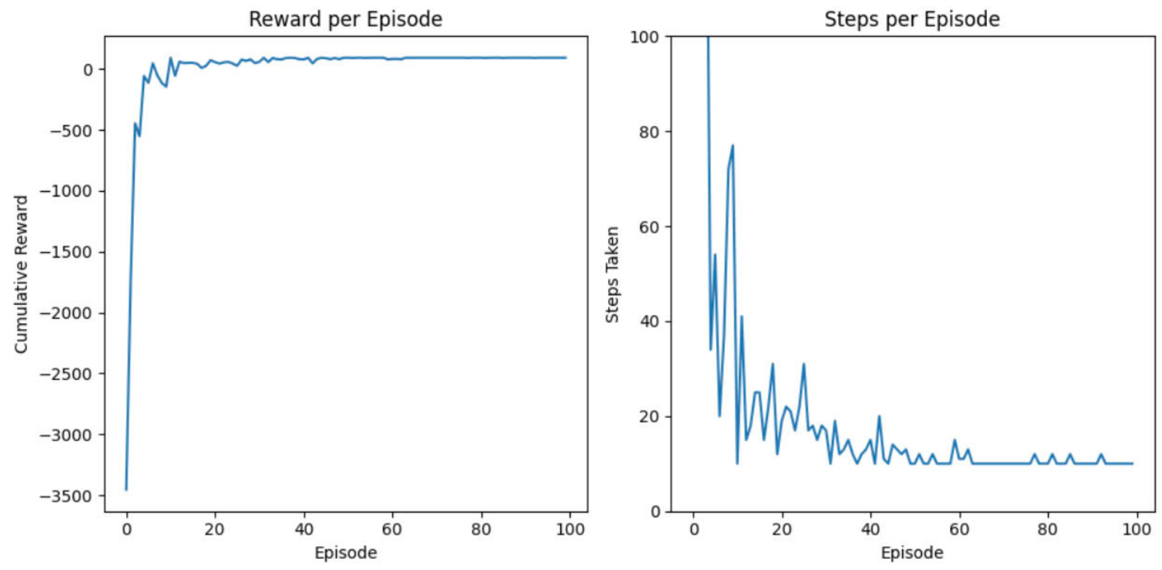


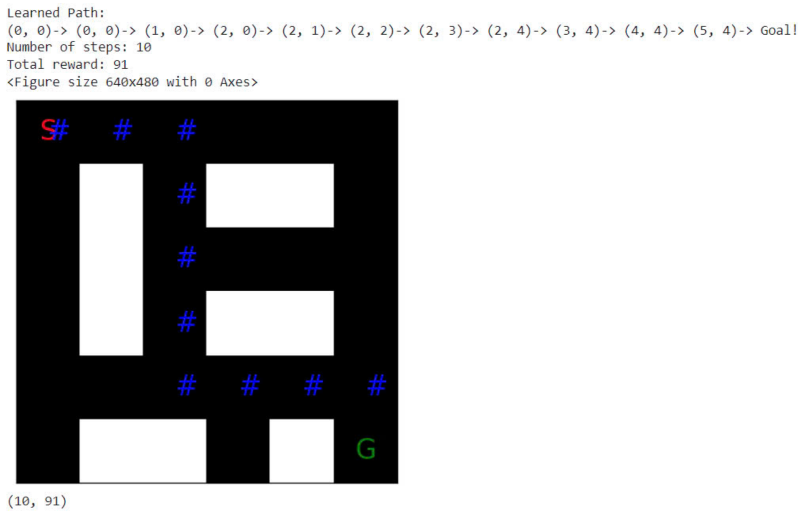
Figure 6: Results

Optimal Step Path

The learned path taken by the agent to reach the goal is as follows:

- **Path:** (0, 0) → (0, 0) → (1, 0) → (2, 0) → (2, 1) → (2, 2) → (2, 3) → (2, 4) → (3, 4) → (4, 4) → (5, 4) → Goal!
- **Number of steps:** 10
- **Total reward:** 91

The agent successfully navigated through the maze, utilizing a total of 10 steps to reach the goal while earning a cumulative reward of 91. The visualization of the path taken is shown in the maze, with 'S' indicating the start point and 'G' indicating the goal.



```
Learned Path:
(0, 0)-> (0, 0)-> (1, 0)-> (2, 0)-> (2, 1)-> (2, 2)-> (2, 3)-> (2, 4)-> (3, 4)-> (4, 4)-> (5, 4)-> Goal!
Number of steps: 10
Total reward: 91
<Figure size 640x480 with 0 Axes>
```

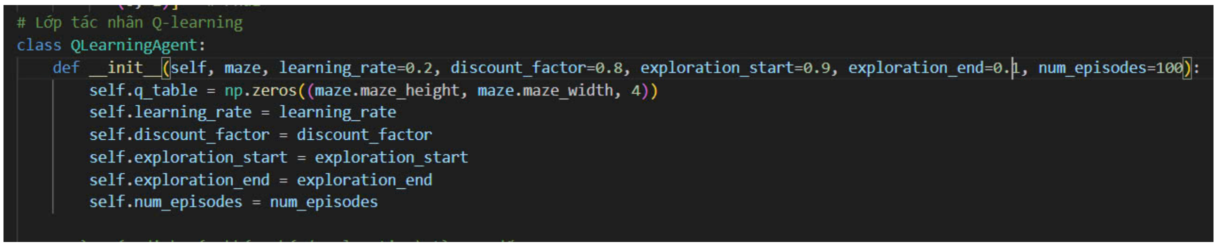
```
(10, 91)
```

Figure 7: Optimal Step Path

4.3 Hyperparameter Tuning

For the Q-learning algorithm:
When we change the parameters as follows:

- learning_rate = 0.2
- discount_factor = 0.8
- exploration_start = 0.9
- exploration_end = 0.1



```
# Lớp tác nhân Q-learning
class QLearningAgent:
    def __init__(self, maze, learning_rate=0.2, discount_factor=0.8, exploration_start=0.9, exploration_end=0.1, num_episodes=100):
        self.q_table = np.zeros((maze.maze_height, maze.maze_width, 4))
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_start = exploration_start
        self.exploration_end = exploration_end
        self.num_episodes = num_episodes
```

Figure 8: Q learning

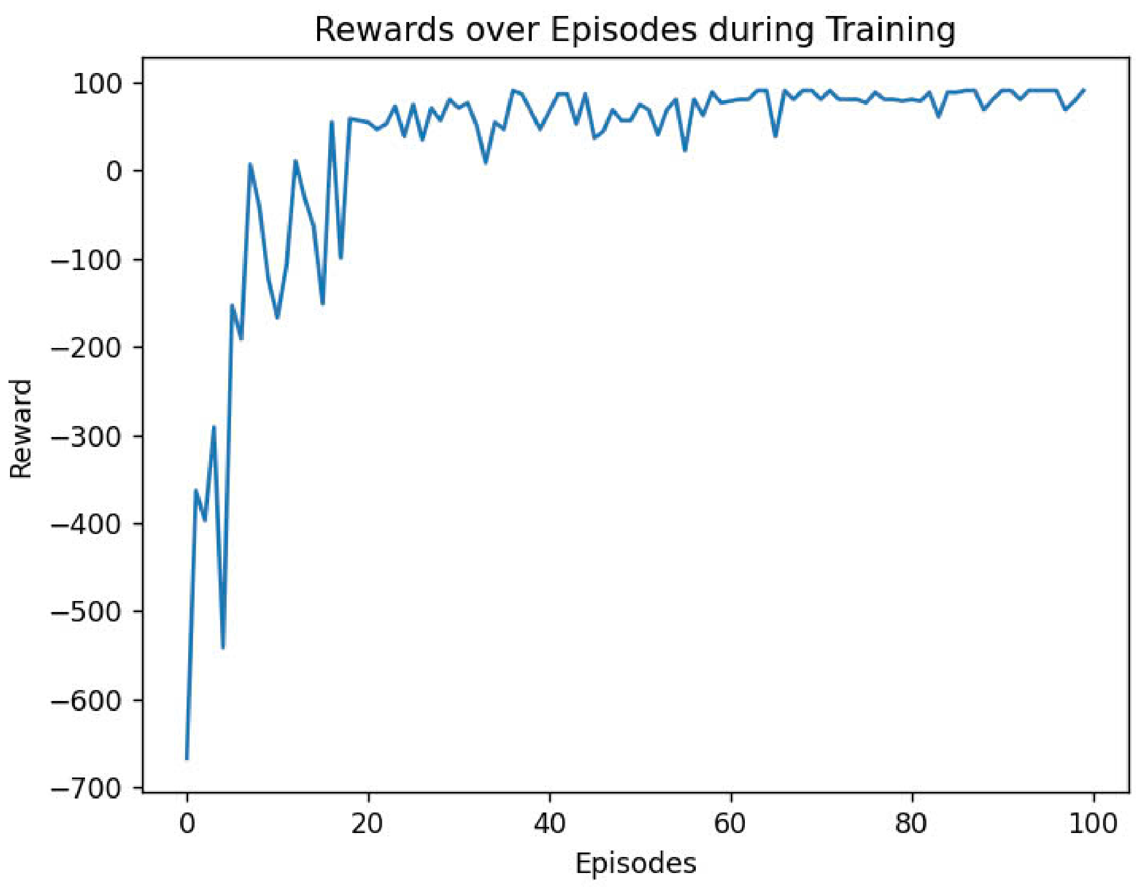
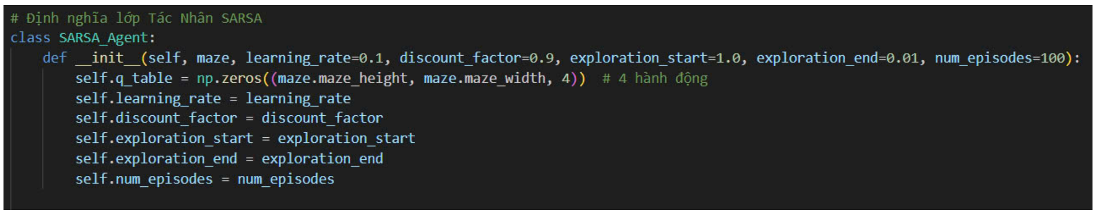


Figure 9: Q learning

For the SARSA algorithm:
When we change the parameters as follows:

- learning_rate = 0.2
- discount_factor = 0.8
- exploration_start = 0.9
- exploration_end = 0.1



```python
# Định nghĩa lớp Tác Nhân SARSA
class SARSA_Agent:
    def __init__(self, maze, learning_rate=0.1, discount_factor=0.9, exploration_start=1.0, exploration_end=0.01, num_episodes=100):
        self.q_table = np.zeros((maze.maze_height, maze.maze_width, 4))  # 4 hành động
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.exploration_start = exploration_start
        self.exploration_end = exploration_end
        self.num_episodes = num_episodes
```

Figure 10: SARSA

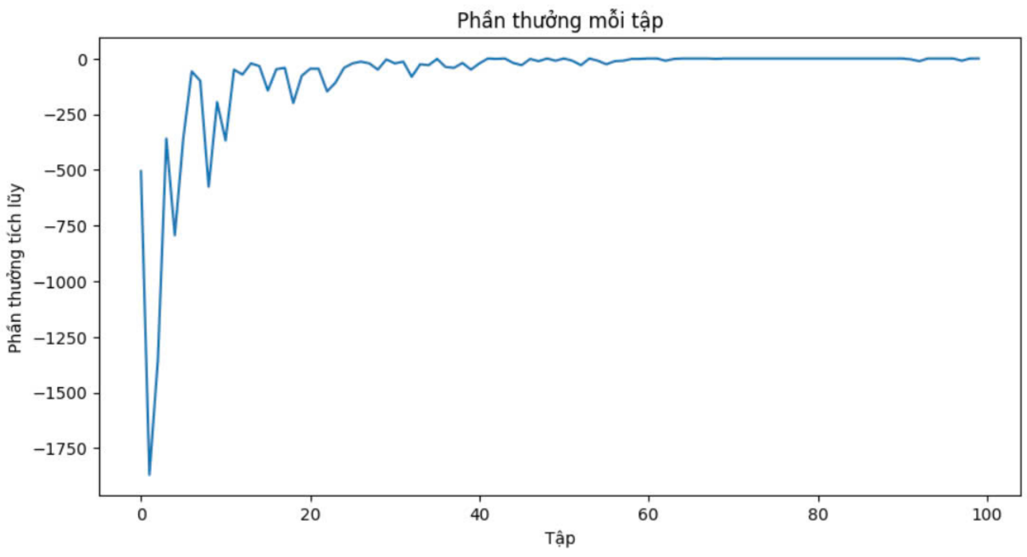


Figure 11: SARSA

Graph 1 (Q-learning): After around 20 episodes, the rewards in Q-learning begin to converge and stabilize around a value of approximately 100. This shows that Q-learning converges quickly and reaches a high reward value. Since Q-learning is an ”off-policy” algorithm, it always tries to select the optimal action based on the Q-table, allowing it to learn faster and converge to an optimal solution in a shorter time. The plot shows slight fluctuations after convergence, but it maintains around a stable value, indicating that the algorithm has found a good policy.

Graph 2 (SARSA): Similar to Q-learning, SARSA also begins to converge after around 20 episodes, but its stable rewards fluctuate around 0, which is significantly lower than Q-learning. SARSA is an ”on-policy” algorithm, so it learns more slowly because it doesn’t always select the optimal action but rather follows the current policy (including exploratory actions). This results in slower convergence and less optimal performance compared to Q-learning. Although SARSA does converge steadily, its lower reward values suggest that it hasn't found an optimal solution or is being more cautious during learning.

Q-learning converges faster and achieves higher rewards, indicating that it finds an optimal solution after training. Q-learning's convergence is better due to its nature of selecting the best possible future actions based on the Q-values. SARSA converges more slowly and stabilizes around a lower reward. This reflects SARSA's more cautious approach, as it follows the current policy to learn. SARSA is better suited for environments with higher risk or when caution is needed during learning.

5 Results

Results of Q-Learning

Cumulative Reward Graph: In the second figure (Q-Learning), the cumulative reward initially drops sharply but quickly stabilizes near zero after about 30 episodes.

Episodes to Convergence: Q-Learning can converge earlier than SARSA, stabilizing after approximately 30 episodes.

Stability: Q-Learning is an off-policy algorithm, which allows for faster convergence by using the optimal value possible. However, it may be less stable than SARSA in complex environments.

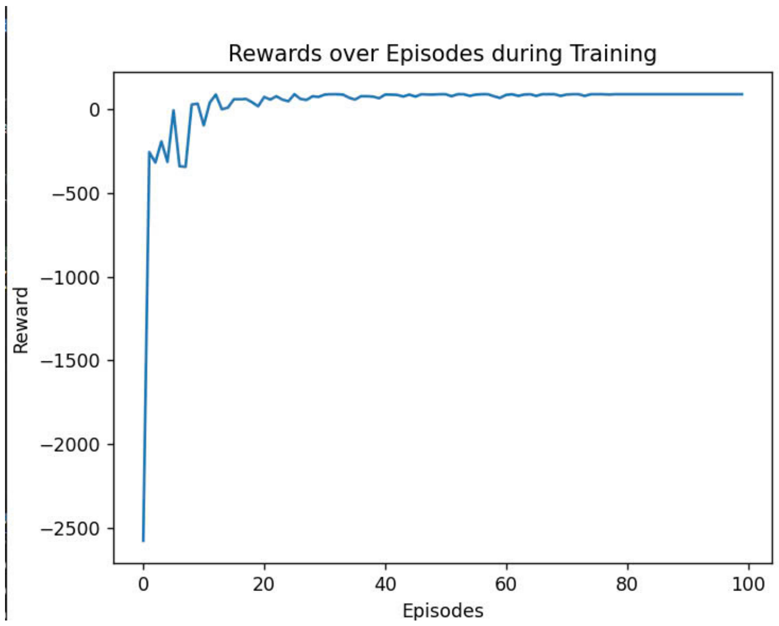


Figure 12: Results (Q learning)

Results of SARSA

Cumulative Reward Graph: Looking at the first figure (SARSA), we see the cumulative reward gradually increases and stabilizes after about 40 episodes. The graph shows some fluctuations in the early stages, but the reward gradually stabilizes near zero.

Episodes to Convergence: SARSA takes about 40 episodes for the cumulative reward to stabilize, indicating a relatively fast convergence.

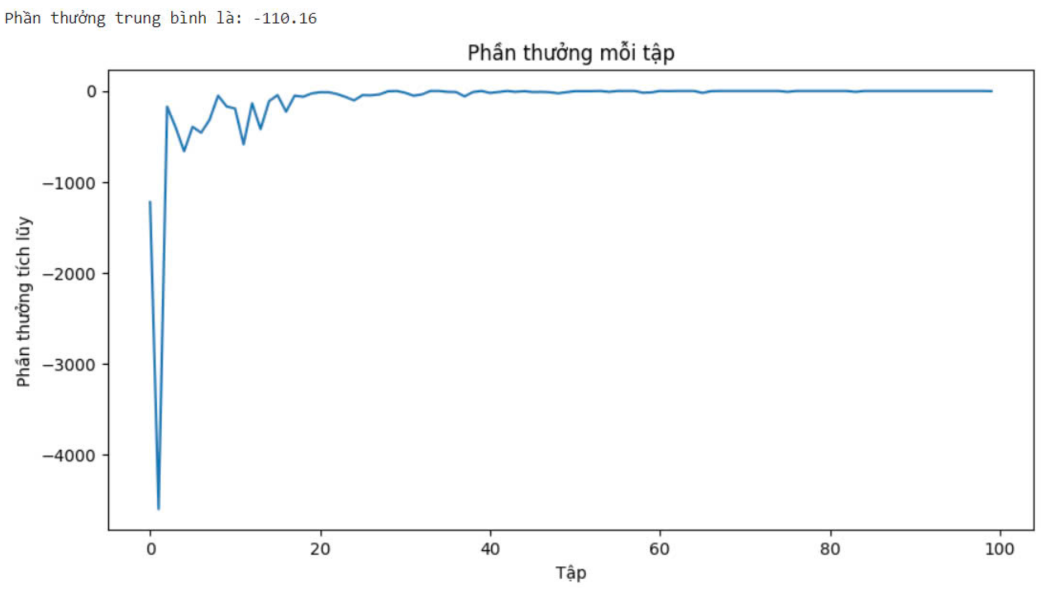
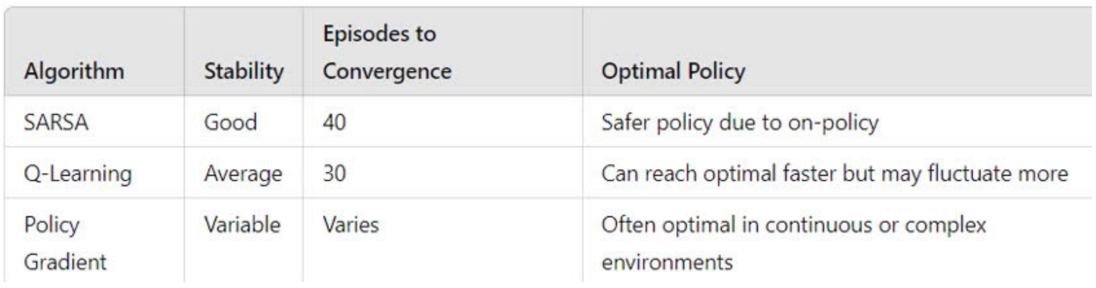

```
Phần thưởng trung bình là: -110.16
```

Figure 13: Results (SARSA)

Stability: Since SARSA is an on-policy algorithm, it may converge a bit slower than Q-Learning but remains stable as the policy and actions are continuously updated during learning.



Algorithm	Stability	Episodes to Convergence	Optimal Policy
SARSA	Good	40	Safer policy due to on-policy
Q-Learning	Average	30	Can reach optimal faster but may fluctuate more
Policy Gradient	Variable	Varies	Often optimal in continuous or complex environments

Figure 14:

Performance Comparison

Stability: SARSA is more stable due to its on-policy approach, suitable for applications requiring safety during training.

Episodes to Convergence: Q-Learning may converge faster, though it can fluctuate in complex environments.

Overall Remarks

SARSA: Suitable for environments requiring safety during the learning process due to its on-policy nature.

Q-Learning: Generally more efficient and faster to converge in simpler scenarios where stability is less of a concern.

Policy Gradient: Can be a strong choice in complex or continuous environments where simple policies may not suffice.

6 . Discussion

6.1 Result Analysis

Performance Comparison: In the given task, Q-Learning appears to converge faster than SARSA, as indicated by the faster stabilization of cumulative rewards. This is because Q-Learning is an off-policy algorithm that optimizes for the maximum expected future reward. By contrast, SARSA, as an on-policy algorithm, uses the reward of the next actual action taken, which is more conservative and results in a more cautious learning process.

Strengths of Q-Learning: Q-Learning often performs well in tasks where exploring the environment aggressively is beneficial, as it seeks out actions that maximize future rewards. This can lead to faster convergence, particularly in simpler or deterministic environments.

Weaknesses of Q-Learning: However, this same characteristic can cause instability in highly dynamic or stochastic environments, as the algorithm may switch policies frequently while seeking the maximum reward.

Strengths of SARSA: SARSA, on the other hand, has greater stability in such environments. Its on-policy nature ensures that the algorithm aligns more closely with the actual policy being executed, leading to smoother transitions and fewer fluctuations during learning.

Weaknesses of SARSA: This conservative approach, however, can slow down convergence, especially in situations where the optimal path requires significant exploration.

6.2 Challenges and Observations

Hyperparameter Tuning: A significant challenge in both Q-Learning and SARSA is tuning hyperparameters, such as the learning rate, exploration-exploitation balance (epsilon), and discount factor. Improper tuning can lead to suboptimal convergence or even failure to learn effectively.

Exploration vs. Exploitation: Both algorithms face the challenge of balancing exploration (discovering new actions) and exploitation (maximizing known rewards). Q-Learning, which is off-policy, can benefit from more aggressive exploration but may suffer from instability if exploration is not well-managed. SARSA, being on-policy, naturally balances this trade-off better but may miss out on discovering higher reward paths quickly.

Reward Sparsity: In tasks where rewards are sparse, both algorithms can struggle, as they depend on frequent rewards to guide the learning process. Sparse rewards require careful exploration strategies to avoid getting stuck in low-reward areas of the environment.

6.3 Convergence Explanation

Q-Learning: Q-Learning tends to converge faster in environments where maximum future rewards can be accurately estimated, thanks to its use of the maximum possible Q-value for the next state. This approach leverages optimistic updates and can quickly hone in on high-reward strategies. However, in dynamic or noisy environments, this might lead to instability as the algorithm frequently updates its strategy based on future rewards that may vary.

SARSA: SARSA often converges more slowly because it uses the next action actually taken under the current policy to update its values. This ensures that the learned policy is directly aligned with the actions taken, leading to a smoother learning process. In cases with high variability, SARSA can provide more stable results, as it doesn't optimistically assume maximum future rewards but instead learns based on realistic expectations under the current policy.

7 Conclusion

7.1 Summary

Performance and Convergence: When comparing the two algorithms, Q-Learning and SARSA, the key findings are as follows: Q-Learning tends to converge faster than SARSA due to its approach of optimizing for maximum future rewards. However, this can lead to instability in complex environments. SARSA, being on-policy, allows for a more stable learning process but may be slower in finding optimal paths.

Strengths and Weaknesses: Q-Learning, as an off-policy method, can explore higher-value actions, which enables faster convergence but may result in fluctuations if the environment is dynamic. SARSA, being on-policy, generally provides a safer learning process since it relies on the actual actions taken and the policy being used, though it may take longer to achieve the optimal policy.

Implementation Challenges: Tuning hyperparameters such as the learning rate and discount factor significantly affects the performance and stability of both algorithms. Managing the balance between exploration and exploitation is also crucial to optimize learning.

7.2 Lessons Learned

Through the project of developing a maze game using the Q-learning algorithm, I gained valuable insights into Reinforcement Learning (RL) techniques and their potential applications to real-world problems. By building and implementing Q-learning, I deepened my understanding of how an agent can learn from experience and adjust its behavior to maximize received rewards.

The project highlighted the importance of tuning hyperparameters and establishing an effective exploration-exploitation policy to achieve optimal performance. RL techniques like Q-learning are particularly useful for solving problems where there are no specific rules for reaching optimal outcomes. This makes RL highly applicable to practical areas such as supply chain optimization, robotic planning, and the development of automatic recommendation systems.

7.3 Future Work

To further enhance this project, several research directions and potential improvements include:

Experimenting with Deep Reinforcement Learning (Deep RL) Methods: By using deep RL methods such as Deep Q-Network (DQN), the agent's capacity to learn from more complex environments could be expanded, particularly in environments with continuous or large state spaces. This would enable the agent to perform well in games with higher complexity.

Exploring More Complex Environments: Applying RL algorithms to environments with high randomness or more intricate interactions, such as 3D environments or multi-agent scenarios. This will help test the scalability of traditional RL algorithms and identify their limitations.

Using Direct Policy Learning Algorithms: Direct policy learning methods like Policy Gradient or Actor-Critic could help the agent optimize policies better in environments where calculating Q-values is impractical.

Integrating RL Techniques into Real-World Applications: Experimenting with applying RL algorithms to practical fields such as energy system optimization, intelligent traffic control, or drone planning. This would help explore the real-world benefits of RL and the challenges of implementation in actual scenarios.

8 References

1: https://canvas.phenikaa-uni.edu.vn/courses/11982/modules/items/193374
2: https://canvas.phenikaa-uni.edu.vn/courses/11982/modules/items/193375
3: https://canvas.phenikaa-uni.edu.vn/courses/11982/modules/items/193376
4: https://mitchellspryn.com/2017/10/28/Solving-A-Maze-With-Q-Learning.html
5: https://towardsdatascience.com/maze-rl-d035f9ccdc63