Maze-Solving Problem with Basic Reinforcement Learning

Tien-Minh Nguyen
Student ID: 22010759, Class: K16 AIRB
Course : Basic Reinforcement Learning
Instructor: Hoang-DIeu Vu
11/10/2024

Abstract—**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.**

Index Terms—**Reinforcement Learning, Q-Learning, SARSA, Monte Carlo, Value iteration, Policy iteration, Maze, Artificial Intelligence.**

I. Introduction

In this project, we address a classic problem in Reinforcement Learning (RL): solving a maze. The maze-solving problem involves placing an agent at a specified starting point within a maze environment. The agent's objective is to navigate from the start to a designated goal location while maneuvering through square cells, step by step. Successfully solving this problem requires the agent to learn a strategy, or policy, that enables it to find the shortest possible path to the goal without taking unnecessary detours. This scenario showcases fundamental RL concepts, such as state exploration, policy optimization, and reward-based learning through continuous interaction with the environment [1].

Each cell in the maze represents a unique state, while walls act as barriers that limit the agent's available actions. As the agent transitions between cells, it receives rewards or penalties based on its proximity to the goal. To succeed, the agent must discover an efficient strategy to navigate the maze, optimizing its route to reach the goal as quickly as possible [2]. The inherent complexity lies in balancing exploration (testing new paths) with exploitation (choosing known shorter routes) to efficiently find the solution.

The primary aim of this project is to implement and compare several foundational RL algorithms, including Q-Learning, SARSA, Policy Iteration, Value Iteration, and Monte Carlo. Each algorithm offers a distinct approach to RL problem-solving, with varying techniques for updating values, learning policies, and managing exploration-exploitation trade-offs [2]. By contrasting these methods, we aim to identify which algorithm is best suited for solving the maze problem, especially in terms of convergence speed and the ability to produce an optimal path.

This report is organized as follows. First, the Theoretical Background section outlines essential RL concepts, including policies, rewards, value functions, and the exploration-exploitation dilemma. We also introduce each of the algorithms employed in the project, describing their underlying mechanisms, unique characteristics, and typical use cases in RL contexts.

In the Methodology section, the maze-solving problem is formalized within an RL framework, where states, actions, rewards, and transitions are clearly defined. The section covers the implementation details of each RL algorithm, describing how they operate within the maze environment, update values, and make decisions. Specific adaptations made to align each algorithm with the maze problem, such as parameter settings, are discussed.

The Experimental Setup section describes the maze layout, including the start and goal positions and wall distribution, and details the parameters for each algorithm, like learning rates, discount factors, and exploration strategies. These setup choices are essential to understanding the conditions and constraints under which each algorithm was evaluated.

The Results section presents the performance of each algorithm, with quantitative data on metrics like convergence speed, optimal path length, and the number of episodes required for solution stability. Visuals, including graphs and tables, highlight the comparative effectiveness of each algorithm under similar experimental conditions.

The Discussion section offers a more in-depth analysis of the results, highlighting the strengths and weaknesses of each algorithm in the context of the maze problem. Key factors influencing algorithm performance, such as parameter tuning and maze complexity, are examined. Challenges encountered during implementation, such as handling dead-ends and achieving an optimal balance between exploration and exploitation, are also discussed.

Finally, the Conclusion summarizes key findings, pointing out which algorithms proved most effective for maze-solving and why. It also suggests potential improvements for future work, such as applying advanced RL techniques, experimenting with larger and more complex mazes, or adjusting the environment for more challenging tasks. By evaluating these traditional RL algorithms, this project sheds light on their respective strengths and limitations, contributing to a broader understanding of their effectiveness in discrete state-space environments.

II. Theoretical Background

Reinforcement Learning (RL) is a subfield of machine learning centered around how an agent can learn to make effective decisions within a dynamic environment to achieve specific goals. The interaction between the agent and environment is fundamental to RL, as it forms a feedback loop that allows the agent to improve over time. At each step, the agent finds itself in a particular state, which reflects the current status or configuration of the environment. Based on this state, the agent takes an action as directed by its policy, which is a strategy mapping states to actions to maximize rewards. Each action receives a reward from the environment, which guides the agent's learning process. The goal of RL is to enable the agent to learn an optimal policy that consistently maximizes cumulative rewards across various states and interactions [1] [3].

The theoretical framework often used in RL is a Markov Decision Process (MDP), a mathematical model for decision-making in situations where outcomes are partly under the agent's control and partly random. An MDP has the essential property called the Markov property, which assumes that the future state depends only on the current state and the action taken—not on any previous states or actions. This assumption simplifies the learning process, making it possible to use algorithms like Q-Learning, SARSA, and Policy Iteration to find optimal policies efficiently. By modeling RL problems as MDPs, we obtain a structured way to tackle complex decision-making scenarios, from robotic navigation to playing strategic games [4] [2].

In RL, the agent is the core decision-maker, tasked with learning which actions lead to the highest rewards across various states. As it interacts with the environment, it explores possible strategies and refines its policy, gradually improving its choices. For example, in a maze-solving task, the agent needs to learn to avoid walls and obstacles while moving towards a target in the shortest path possible. The agent continuously adapts its actions in response to the feedback it receives from the environment, which is a primary component defining how it learns [3].

The environment in RL defines the context in which the agent operates. It provides feedback in the form of rewards or penalties, influencing the agent's decisions and strategies. Each state within the environment represents a distinct situation that the agent encounters. For instance, in a maze-solving scenario, each position the agent occupies could be considered a unique state, and every movement represents a transition to a new state. By exploring the environment, the agent gathers information about which actions yield positive or negative outcomes, a process that is essential to building a robust strategy [1].

The reward signal is fundamental in RL, providing feedback after each action to indicate the desirability of that action. Positive rewards reinforce actions that help the agent reach its goals, while negative rewards discourage less effective actions. The agent's objective is to maximize these rewards over time, leading it to balance between short-term gains and long-term goals. In our maze example, reaching the goal could be associated with a high positive reward, while actions leading to dead ends or obstacles might incur penalties [4].

The agent's policy represents its strategy for choosing actions across different states. A strong policy enables the agent to consistently perform well by selecting the most effective actions in each situation. Policies can be deterministic, meaning they produce the same action for a given state, or probabilistic, where actions may vary based on probability distributions derived from past experiences. Refining the policy is central to RL, as it defines the agent's behavior and, ultimately, its success in achieving the desired objectives [1] [2].

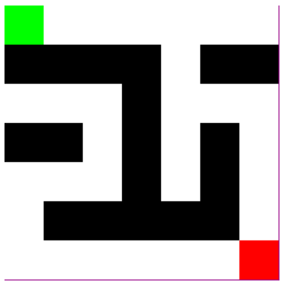
A significant challenge in RL is finding the right balance between exploration and exploitation. The agent needs to explore the environment to discover new actions and their associated rewards, but it also must exploit its current knowledge to make decisions that yield high rewards based on past experiences. This trade-off is crucial for the agent's success; excessive exploration might delay reaching the goal, while too much exploitation may prevent the agent from discovering potentially superior strategies. Balancing exploration and exploitation is essential in developing an optimal policy, as it enables the agent to efficiently navigate complex environments [3][2].

III. Maze Environment

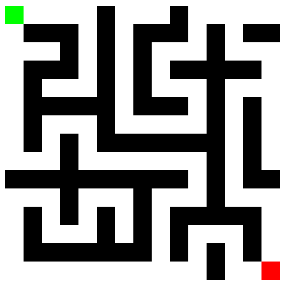
In this project, a custom environment was developed for the maze-solving problem, with the maze generated using the Randomized Depth-First Search (DFS) algorithm [5]. This algorithm begins with a grid fully populated with walls, representing an initial closed maze. Through recursive backtracking, it carves out paths to form a solvable maze. Starting from a random cell, the agent selects neighboring cells to visit, marking them as visited and creating open paths. This method is effective in generating mazes that have a unique solution, making it ideal for testing RL algorithms [6].

The created maze serves as the RL environment for the agent (Fig. 1). In this environment, the agent's goal is to navigate through the maze to reach a designated goal state, typically represented as a target cell within the grid. The state space for the agent comprises all possible grid positions, where each state corresponds to a specific cell in the maze [1]. These states are represented as nodes within an adjacency matrix, allowing the agent to determine valid moves based on available paths. Each move represents an action in the RL context, moving the agent from one cell to another according to the grid's open paths. In this environment, the agent interacts with three main components: the state space, action space, and reward structure.

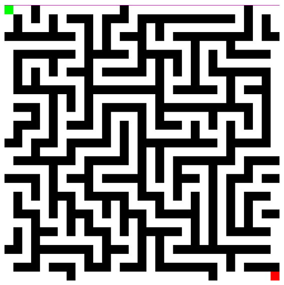
In the maze environment, the state space represents all possible positions within the maze, with each state corresponding to a particular cell. These cells are nodes on the maze's adjacency matrix, where each node represents a specific location the agent can occupy. The initial state is the agent's starting point, and the goal state is the destination



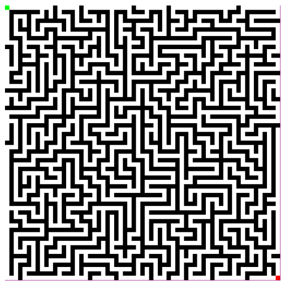
(a) 4x4 Maze



(b) 8x8 Maze



(c) 16x16 Maze



(d) 32x32 Maze

Fig. 1: Mazes are generated with different sizes

the agent seeks to reach. Various maze structures, including smaller and larger mazes (e.g., 4x4, 8x8, 16x16, 32x32 grids), were tested to analyze the performance of different algorithms in varying levels of complexity. Visualizations of these maze grids provide insight into how the agent perceives and interacts with each unique environment.

The action space defines the available movements the agent can make at each state. Each action corresponds to a move between adjacent cells, governed by the adjacency matrix of the maze. The adjacency matrix specifies connections between cells, with a value greater than zero between two nodes indicating a valid path. The agent's action choices at any position depend on these connections, ensuring that moves are constrained to valid, traversable paths within the maze. The get_possible_actions function dynamically retrieves the possible moves from the agent's current location, guiding it towards connected neighboring cells. This allows the agent to explore the maze efficiently and avoid paths that lead to dead ends or unnecessary detours.

The reward structure motivates the agent to find the shortest path to the goal. Each movement incurs a small negative reward (-0.1) to incentivize efficient navigation. Actions that attempt to move the agent to an unconnected cell, representing an invalid move, result in a penalty of -1, discouraging exploration of non-viable paths. Furthermore, if the agent exceeds a predetermined maximum number of steps, it incurs a larger penalty of -10 to encourage more optimal path-finding strategies. The agent receives a positive reward of 10 upon reaching the goal (sinkerNode), signaling successful completion of the objective. Table I outlines this reward structure, emphasizing penalties for inefficiency and rewarding goal achievement.

TABLE I: Reward system for the maze-solving agent

Action	**Reward**
Valid movement (step)	-0.1
Invalid action (no connection between nodes)	-1
Exceeding maximum steps	-10
Reaching the goal (sinkerNode)	10

IV. Methodology

Several RL algorithms were considered for solving this problem, including Q-Learning, SARSA, Policy Iteration, Value Iteration, and Monte Carlo methods.

A. Q-Learning

Q-Learning [2] is a model-free, off-policy algorithm where the agent learns a Q-value function, $Q(s,a)$, representing the expected future reward for each action a in each state s. This approach allows the agent to select actions that maximize the expected cumulative reward over time, even if the agent does not follow the optimal policy during training.

The Q-value update rule is:

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

where: - s and s' are the current and next states, - a and a' are the current and next actions, - r is the reward for transitioning from s to s' by taking action a, - α is the learning rate, - γ is the discount factor.

The algorithm iteratively updates Q-values until they converge to an optimal Q-value function, which the agent can use to derive the optimal policy.

Algorithm 1 Q-Learning Algorithm

```
1: Initialize $Q(s,a)$ arbitrarily for all states $s$ and actions $a$
2: for each episode do
3:     Initialize state $s$
4:     repeat
5:         Choose action $a$ using epsilon-greedy policy based on $Q(s,a)$
6:         Take action $a$, observe reward $r$ and next state $s'$
7:         Update Q-value:
8:             $Q(s,a) \leftarrow Q(s,a)+\alpha \left[ r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$
9:         $s \leftarrow s'$
10:    until $s$ is terminal
11: end for
```

B. SARSA

SARSA [7] (State-Action-Reward-State-Action) is an on-policy algorithm that updates the Q-values based on the action the agent actually takes according to its policy, rather than the action that would yield the maximum future reward. SARSA follows the actual policy to update its Q-values, making it an on-policy method. The update rule is:

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

Algorithm 2 SARSA Algorithm

```
1: Initialize $Q(s,a)$ arbitrarily for all states $s$ and actions $a$
2: for each episode do
3:     Initialize state $s$
4:     Choose action $a$ using epsilon-greedy policy based on $Q(s,a)$
5:     repeat
6:         Take action $a$, observe reward $r$ and next state $s'$
7:         Choose next action $a'$ using epsilon-greedy policy based on $Q(s',a')$
8:         Update Q-value:
9:             $Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r + \gamma Q(s',a') - Q(s,a) \right]$
10:        $s \leftarrow s'$
11:        $a \leftarrow a'$
12:    until $s$ is terminal
13: end for
```

where s, a, s', a' refer to the current state, action, next state, and next action.

C. Policy Iteration

Policy Iteration [8] is a model-based reinforcement learning algorithm where the agent learns a policy explicitly by alternating between evaluating the policy and improving it. This iterative approach is typically applied in environments with known dynamics and is particularly useful for small, finite environments.

Policy Iteration consists of two steps:

1) **Policy Evaluation**: Calculate the value function $V^\pi(s)$ for a given policy π.
2) **Policy Improvement**: Update the policy π by selecting actions that maximize $V^\pi(s)$.

This process repeats until convergence to the optimal policy.

Algorithm 3 Policy Iteration Algorithm

```
1: Initialize policy $\pi$ arbitrarily
2: repeat
3:     Policy Evaluation: For each state $s$, calculate $V^\pi(s)$ by solving:
4:         $V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[r(s,a,s') + \gamma V^\pi(s')]$
5:     Policy Improvement: Update policy $\pi$ to maximize expected value:
6:         $\pi(s) \leftarrow \arg\max_a \sum_{s'} P(s'|s,a)[r(s,a,s') + \gamma V^\pi(s')]$
7: until policy $\pi$ converges to optimal policy
```

D. Value Iteration

Value Iteration [9] is a variation of Policy Iteration that combines policy evaluation and improvement in a single update step. Instead of iteratively computing $V(s)$ for a fixed policy, Value Iteration uses the Bellman optimality equation to update $V(s)$, aiming to approximate the optimal value function more efficiently.

The update rule for Value Iteration is:

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

where $P(s'|s,a)$ represents the transition probabilities.

Algorithm 4 Value Iteration Algorithm

```
1: Initialize $V(s)$ arbitrarily for all states $s$
2: repeat
3:     for each state $s$ do
4:         Update value for state $s$:
5:             $V(s) \leftarrow \max_a \sum_{s'} P(s'|s,a)[r(s,a,s') + \gamma V(s')]$
6:     end for
7: until $V(s)$ converges for all $s$
8: Extract policy $\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)[r(s,a,s') + \gamma V(s')]$
```

E. Monte Carlo Methods

Monte Carlo methods [10] are model-free algorithms that estimate value functions by calculating the average return from complete episodes. Monte Carlo methods are particularly effective when the environment's dynamics are unknown, as they rely on actual observed returns rather than theoretical probabilities.

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 represents the cumulative reward from time step t to the end of the episode. This return is computed as:

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

V. Experimental Setup

A. 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.

Algorithm 5 Monte Carlo Algorithm

```
1: Initialize $Q(s, a)$ arbitrarily for all states $s$ and actions $a$
2: for each episode do
3:     Initialize empty list to store episode history
4:     Initialize state $s$
5:     repeat
6:         Choose action $a$ using epsilon-greedy policy based on $Q(s, a)$
7:         Take action $a$, observe reward $r$ and next state $s'$
8:         Append $(s, a, r)$ to episode history
9:         $s \leftarrow s'$
10:    until $s$ is terminal
11:    for each state-action pair $(s, a)$ in the episode do
12:        Calculate return $G_t$ from time step $t$ onward
13:        Update Q-value:
14:            $Q(s, a) \leftarrow Q(s, a) + \alpha [G_t - Q(s, a)]$
15:    end for
16: end for
```

- **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.

B. 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.

C. 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).

For the Q-Learning, SARSA and Monte Carlo 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.

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.

VI. Results

In this section, I present the results for each algorithm, including Q-Learning, SARSA, Monte Carlo, Policy Iteration, and Value Iteration. I use tables and graphs to show performance metrics such as cumulative rewards, epsilons required to converge, and algorithm running time. The results of the algorithms are computed based on an 8x8 maze.

A. Q-Learning

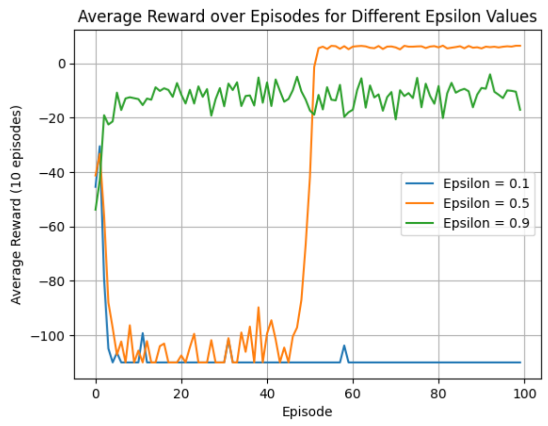


Fig. 2: Q-Learning with varying ϵ

The Q-Learning algorithm demonstrated a gradual improvement in performance as the Q-values updated over time. The graph (Fig. 2) 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. We can see that the algorithm performs best with a value of ϵ of 0.5 and worst with a value of ϵ of 0.1

TABLE II: Running time and best total reward for different ϵ values of Q-Learning algorithm

Epsilon	Running Time (seconds)	Best Total Reward
0.1	2999.6	6.9
0.5	3271.5	8.1
0.9	3412.3	6.9

Table II shows the algorithm running time and the best total reward corresponding to different ϵ values.We see that with ϵ equal to 0.5, the highest total reward is 8.1 corresponding to a running time of 3271.5 seconds.

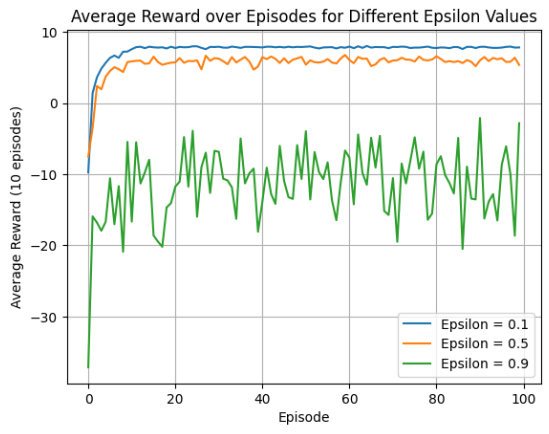


Fig. 3: SARSA with varying ϵ

B. 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.The results in Figure 3 show that the algorithm converges best with ϵ equal to 0.1 followed by 0.5, with ϵ equal to 0.9 the algorithm does not achieve the best result.

TABLE III: Running time and best total reward for different ϵ values of SARSA algorithm

Epsilon	Running Time (seconds)	Best Total Reward
0.1	11.5	8.1
0.5	19.3	8.1
0.9	87.8	6.7

Table III shows the algorithm running time and the best total reward corresponding to different ϵ values.We see that with ϵ equal to 0.1 and 0.5, the highest total reward is 8.1 corresponding to a running time of 11.5 and 19.3 seconds.

C. Monte Carlo

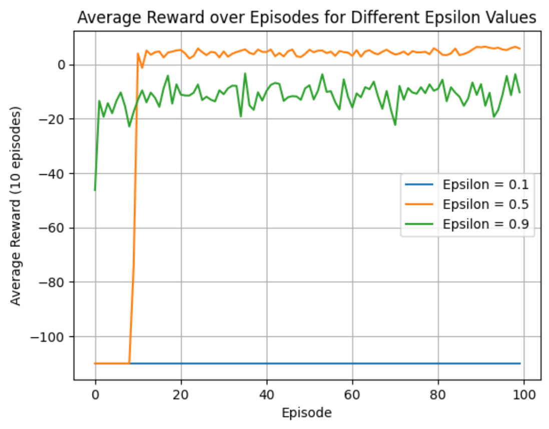


Fig. 4: Monte Carlo with varying ϵ

The Monte Carlo algorithm shows slower convergence than SARSA. However, it works well in environments where episodes can be simulated to completion.The results in Figure 4 show that the algorithm converges best with ϵ equal to 0.5 followed by 0.9, with ϵ equal to 0.1 the algorithm does not achieve the best result.

TABLE IV: Running time and best total reward for different ϵ values of Monte Carlo algorithm

Epsilon	Running Time (seconds)	Best Total Reward
0.1	6314.6	-110
0.5	45	8.1
0.9	53.7	6.7

Table IV shows the algorithm running time and the best total reward corresponding to different ϵ values.We see that with ϵ equal to 0.5, the highest total reward is 8.1 corresponding to a running time of 45 seconds.

D. 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.

TABLE V: Algorithm Running Time by Gamma Value

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

E. 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.

TABLE VI: Algorithm Running Time by Gamma Value

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

F. Performance Comparison

In figure 5, SARSA demonstrates the fastest convergence, quickly achieving stable positive rewards within just a few episodes. This efficient learning can be attributed to SARSA's on-policy approach, which aligns value updates closely with the current policy. Monte Carlo also stabilizes at a positive reward but takes slightly longer to converge due to its reliance on episodic returns for learning. Q-Learning, on the other hand, shows the slowest convergence, with significant fluctuations and negative rewards initially before gradually stabilizing. This slower convergence reflects Q-Learning's off-policy nature,

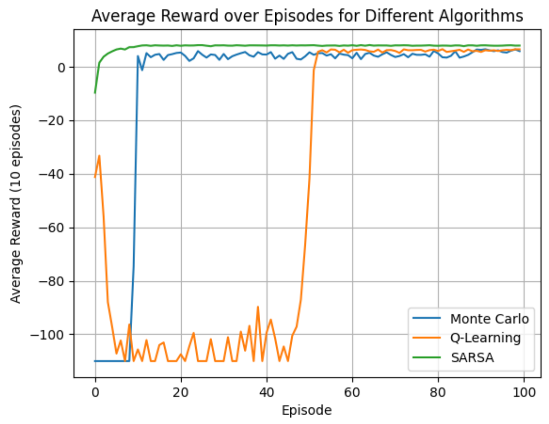


Fig. 5: Comparison of Rewards Across Q-Learning, SARSA, and Monte Carlo Algorithms

which drives more extensive exploration. Overall, SARSA proves to be the most efficient in this context, followed by Monte Carlo and then Q-Learning.

TABLE VII: Performance Comparison of RL Algorithms with 4x4 maze

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

TABLE VIII: Performance Comparison of RL Algorithms with 8x8 maze

Algorithm	Stability	Speed	Optimal Policy Found
Q-Learning	Medium	Medium	Yes
SARSA	High	Fast	Yes
Monte Carlo	Low	Fast	Yes
Policy Iteration	Low	High	Yes
Value Iteration	Low	High	Yes

TABLE IX: Performance Comparison of RL Algorithms with 16x16 maze

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

TABLE X: Performance Comparison of RL Algorithms with 32x32 maze

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

The following table (Table VII,VIII,IX,X) summarizes the performance of each algorithm in terms of stability, speed of convergence, and the optimal policy found for different sizes of mazes.

VII. Conclusion

This project implemented and compared several traditional Reinforcement Learning (RL) algorithms—SARSA, Monte Carlo, Q-Learning, Policy Iteration, and Value Iteration—for solving maze navigation problems. SARSA demonstrated the fastest convergence rate, benefiting from its on-policy nature, which allowed for stable and efficient learning. Monte Carlo displayed a moderate convergence rate, leveraging episodic rewards to build reliable value estimates but requiring complete episodes for updates. Q-Learning, while highly exploratory, showed the slowest convergence due to its off-policy approach, often exploring less optimal paths before improving its Q-values. Policy Iteration and Value Iteration performed efficiently in smaller mazes but faced significant slowdowns in larger ones due to the increased computational demands of evaluating all states. Each algorithm presented distinct advantages and limitations, making them suitable for various environment complexities and navigation tasks.

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. 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.

Acknowledgment

I would like to thank Dr.Vu Hoang Dieu for the guidance and support throughout this project.

References

[1] R. S. Sutton, "Reinforcement learning: An introduction," *A Bradford Book*, 2018.

[2] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[3] S. J. Russell and A. Zimdars, "Q-decomposition for reinforcement learning agents," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 656–663.

[4] R. Bellman, "A markovian decision process," *Journal of mathematics and mechanics*, pp. 679–684, 1957.

[5] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[6] N. Dalla Pozza, L. Buffoni, S. Martina, and F. Caruso, "Quantum reinforcement learning: the maze problem," *Quantum Machine Intelligence*, vol. 4, no. 1, p. 11, 2022.

[7] N. Sprague and D. Ballard, "Multiple-goal reinforcement learning with modular sarsa (0)," 2003.

[8] S. P. Meyn, "The policy iteration algorithm for average reward markov decision processes with general state space," *IEEE Transactions on Automatic Control*, vol. 42, no. 12, pp. 1663–1680, 1997.

[9] K. Chatterjee and T. A. Henzinger, "Value iteration," in *25 Years of Model Checking: History, Achievements, Perspectives*. Springer, 2008, pp. 107–138.

[10] Y. Iba, "Population monte carlo algorithms," *Transactions of the Japanese Society for Artificial Intelligence*, vol. 16, no. 2, pp. 279–286, 2001.