Reinforcement Learning for Sokoban Game

Le Duc Hoa
Student ID: 22010984, Class: K16 AIRB
Course: Sokoban Reinforcement Learning
Instructor: Hoang-Dieu Vu
11/10/2024

Abstract

Reinforcement Learning (RL) is a key subfield of machine learning that focuses on training agents to make decisions by interacting with an environment and maximizing cumulative rewards. In contrast to supervised learning, where models learn from labeled data, RL emphasizes learning through trial and error, allowing agents to discover optimal policies by exploring and exploiting available information. Central concepts in RL include states, actions, rewards, and policies, often formalized through a Markov Decision Process (MDP).

RL has made significant contributions across various domains, including gaming, robotics, and finance. Notably, RL has achieved superhuman performance in complex games such as Chess, Go, and StarCraft by discovering novel strategies beyond human capabilities. In robotics, RL enables machines to adapt to dynamic environments, facilitating tasks such as autonomous navigation and manipulation. In finance, RL is used in algorithmic trading and portfolio management to optimize decision-making under uncertain market conditions.

1 Introduction

Reinforcement Learning (RL) and Artificial Intelligence (AI) have experienced a remarkable surge in recent years, as demonstrated by the emergence of humanoid robots, autonomous vehicles, and other cutting-edge technologies. The critical role of RL in shaping the future is particularly evident in the quest to develop versatile, multi-functional robots. While current robots offer various capabilities designed to address numerous challenges, integrating these functions into a cohesive system remains a significant hurdle.
The paper "Review of Reinforcement Learning Research" by Jingkai Jia and Wenlin Wang (2020)[1] reviews current research on reinforcement learning. The authors present key algorithms in this field, including dynamic programming, Monte Carlo methods, TD-learning,

Q-learning, and Sarsa. The paper also discusses the advances and challenges in the application of reinforcement learning, as well as optimization methods and future research directions. The paper emphasizes the importance of improving algorithms and techniques to enhance the performance of automatic learning systems.

Machine Learning (ML) and RL have emerged as powerful tools for tackling this complexity, enabling robots to compute and select optimal actions instead of relying on exhaustive programming for every potential scenario. For example, programming a robot to press a button is relatively simple, but enabling it to handle a broader set of tasks—such as pressing buttons, toggling switches, and turning knobs—becomes far more complex. This is where RL excels, offering decision-making capabilities across a variety of situations.
The paper "An Empirical Study of On-Policy and Off-Policy Actor-Critic Algorithms in the Context of Exploration-Exploitation Dilemma" (2023)[2] by Supriya Seshagiri and Prema K. V. studies the differences between on-policy and off-policy actor-critic algorithms in the context of the exploration-exploitation problem. The authors conduct empirical experiments to compare the performance of these algorithms, focusing on how factors such as entropy affect optimization in reinforcement learning. The results show that the choice between on-policy and off-policy can significantly affect the performance and exploration ability of reinforcement learning models. This study draws on foundational works in reinforcement learning. Our goal is to explore fundamental knowledge that applies directly to our robotic system, aiming to gain a deep understanding of the agent's (robot's) behavior during interactions with its environment. This understanding forms the foundation for selecting and optimizing algorithms to achieve the most efficient and adaptive behavior The core focus of this paper lies in analyzing and optimizing RL algorithms based on theoretical frameworks, with simulations serving as test environments for various functions and scenarios. Our contributions to this research include the following key areas:

- **Reward and Punishment Optimization for Dynamic Tasks:** We focus on identifying the optimal balance of rewards and punishments for dynamic push and pull tasks, significantly improving the learning process for mobile robots.
- **Specialized RL Algorithms for Push and Pull Operations:** To address the unique challenges of push and pull tasks, we propose specialized RL algorithms.
- **Behavioral Patterns for Adaptive Robotic Systems:** By analyzing the learned behaviors of mobile robots in dynamic environments, we uncover key patterns crucial for building effective and adaptable robotic systems.
- **Practical Implications for RL in Robotics:** Our findings shed light on the practical applications of RL in dynamic push and pull tasks.

2 Related Work

In the context of developing reinforcement learning methods,The paper of Garcia (2001) [3] presents a method that emphasizes the management of exceptions during the learning process. This approach aims to enhance the system's learning capabilities by enabling it to respond more effectively to unusual situations.
The paper "Proximal Policy Optimization Algorithms" by John Schulman et al. (2017) [5] introduces the Proximal Policy Optimization (PPO) method, which aims to improve both the stability and performance of reinforcement learning algorithms. PPO utilizes a new loss function that facilitates effective policy updates without necessitating extensive information about the model. The results indicate that PPO achieves outstanding performance across various complex tasks, making it a widely adopted approach in the reinforcement learning community.
The paper "Push and Pull Robot with Reinforcement Learning Algorithms" by Vu Hoang-Dieu et al. (2024)[6] presents a mobile robot system capable of performing push and pull operations in a Sokoban environment. The research focuses on applying reinforcement learning algorithms, including Monte Carlo methods and heuristic algorithms, to improve the performance and stability of the robot in completing complex tasks. The results show that the use of these algorithms can significantly enhance the automation capability in the field of service robots.
The paper "Simultaneously Learning at Different Levels of Abstraction" by Wurm and Lingnau (2015)[7] investigates the brain's ability to learn simultaneously at different levels of abstraction. The authors analyze how the neural system can integrate information from different levels to optimize learning. The study uses neuroimaging methods to demonstrate that parallel learning at multiple levels of abstraction not only enhances cognitive performance but also improves the ability to perform complex tasks in real-world environments.
Reinforcement learning is a subset of machine learning that stands out due to its unique approach to learning. Unlike other learning types, reinforcement learning evaluates the actions taken based on feedback from training data. The agent is not explicitly instructed on which actions to take or avoid; instead, it seeks to identify the actions that yield the highest rewards. This process relies on a trial-and-error mechanism, learning through continuous experimentation and feedback. Key algorithms in this domain include dynamic programming, Monte Carlo methods, Q-Learning, TD-Learning, and the Sarsa algorithm.
The paper "Dynamically Interrupting Deadlocks in Game Learning Using Multisampling Multiarmed Bandits" by Rendong Chen and Fa Wu (2023)[8] proposes a novel approach to solving the deadlock problem in game learning, especially in games like Sokoban. The authors use multisampling multiarmed bandits to determine the appropriate time to intervene and break the deadlock. The study focuses on cost analysis and optimization of the recovery process, and provides a more efficient learning approach by minimizing the learning loss.
The paper "Deep Reinforcement Learning with Experience Replay Based on SARSA"

(2016)[9] by Dongbin Zhao and co-authors studies a deep learning method in reinforcement learning, combined with experience replay storage based on the SARSA algorithm. The author introduces how the experience collected from previous interactions is used to improve the learning performance of the model. The study shows that the use of experience replay not only improves the stability but also enhances the exploitation ability of the SARSA algorithm. Experimental results show that this method can achieve better performance than traditional methods in complex reinforcement learning problems.
The paper "Comparison of Deep Reinforcement Learning Approaches for Intelligent Game Playing" by Anoop Jeerige, Doina Bein, and Abhishek Verma (2019)[10] compares Deep Reinforcement Learning (DRL) approaches for developing intelligent agents in games. The authors analyze and evaluate the performance of various DRL approaches, including Policy Gradient, Deep Q-Learning Network, and Asynchronous Advantage Actor-Critic. This study uses the OpenAI Gym environment for testing and shows that each approach has its own advantages and limitations, and provides important insights into choosing the appropriate approach for game applications. After going through many algorithms of the above authors, I am trying to test all the algorithms of dynamic programming for Sokoban environment. In order to optimize the agent's moves, with the paper of "Deep Reinforcement Learning with Experience Replay Based on SARSA" (2016) mentioned above, we will try to optimize the sarsa algorithm for Sokoban environment.

3 Background

3.1 Learning in Sokoban

Reinforcement Learning (RL) involves an agent that interacts with an environment to maximize cumulative rewards. Key components of RL include:

- **Agent:** The learner or decision maker that interacts with the environment, which in this case is the Sokoban game.
- **Environment:** The setting in which the agent operates, represented by the Sokoban game board.
- **State (s):** A representation of the environment at a given time, indicating the positions of the player and the boxes on the board.
- **Action (a):** The choices available to the agent, including moving or pushing boxes, which affect the state of the environment.
- **Reward (r):** Feedback received after taking an action in a state, guiding the agent's learning process.

- **Policy ():** A strategy that defines the agent's behavior, mapping states to actions.

A critical concept in RL is the **exploration-exploitation trade-off**, where the agent must balance between exploring new actions (exploration) and leveraging known rewarding actions (exploitation). This is often modeled using **Markov Decision Processes (MDP)**, which provide a formal framework for decision-making where outcomes are partly random and partly under the control of the decision maker.

3.2 Overview of Algorithms

3.2.1 Q-Learning

Q-Learning is an off-policy, model-free algorithm. It learns the value of action-reward pairs (Q-values) without requiring a model of the environment. The core update rule for Q-values is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1}$$

where α is the learning rate, γ is the discount factor, and r is the reward received after transitioning from state s to state s' by taking action a.

3.2.2 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy algorithm, which means it updates the Q-values based on the actions actually taken by the agent. The update rule differs from Q-Learning as it considers the next action chosen by the policy:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right] \tag{2}$$

3.2.3 Monte Carlo

Monte Carlo is a reinforcement learning algorithm that updates the Q-values based on the actual returns obtained from full episodes. Unlike temporal-difference methods, which update the Q-values step-by-step, Monte Carlo waits until the entire episode finishes and then computes the cumulative reward (return) for each state-action pair encountered in the episode. The Q-values are updated by averaging the returns across multiple episodes:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[G_t - Q(s,a) \right], \tag{3}$$

where G_t is the return (total accumulated reward) starting from time step t.

The main distinction between SARSA and Q-Learning is that SARSA uses the action selected by the policy, while Q-Learning uses the action that maximizes the Q-value.

4 Methodology

4.1 Problem Definition

The Sokoban problem is framed as a grid-world environment where an agent (the player) pushes boxes to designated target locations (goals). The environment consists of the following components:

- **State Space:** Each state represents a unique configuration of the grid, including the position of the agent and the boxes, the positions of the traps, and the goals. With a grid whose rows are states and whose columns are actions, the state space consists of 100 states, encoded based on the positions of the agent and the boxes.
- **Action Space:** The agent can perform a set of actions:

Action Index	Action Description
1	No action
2	Move Up
3	Move Down
4	Move Left
5	Move Right
6	Push Up
7	Push Down
8	Push Left
9	Push Right

Table 1: Action for Sokoban Environment

4.2 Algorithm Implementation

The RL algorithms are implemented using Python, employing libraries such as NumPy for numerical operations and Matplotlib for visualization. The structure of the Sokoban environment is defined as follows: The state transition function determines how the environment responds to the agent's actions. The reward structure is defined such that the agent receives positive rewards for moving boxes to their goals and negative rewards for invalid actions or getting stuck.

4.3 Simulation and Results

Simulations are conducted over multiple episodes, allowing the agent to explore the environment and update its Q-values or policy based on the observed rewards. The performance is

evaluated by tracking the agent's learning progress, including the number of steps taken to complete the game and the convergence of the Q-values. Reinforcement Learning provides a robust framework for training agents to solve complex problems such as the Sokoban game. By implementing various algorithms like Q-Learning, SARSA, and Monte Carlo, we can explore their effectiveness in navigating and optimizing performance in dynamic environments. Future work includes enhancing the reward structure and integrating more advanced techniques such as deep reinforcement learning to address larger state spaces and more intricate tasks.

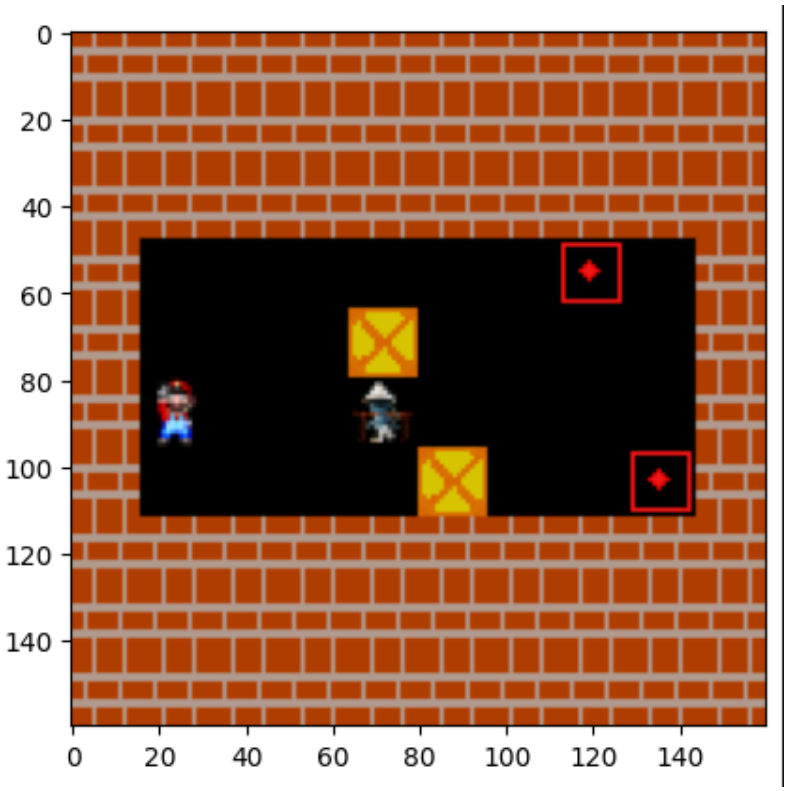

Figure 1: The Sokoban Environment.

Note: *mushroom man is a trap.*

- **Perform Step** (-0.1): Small penalty for each move, encouraging the agent to complete the game in the fewest steps possible.
- **Push Box on Target** (1.0): High reward for pushing a box onto a target, incentivizing the main goal achievement.
- **Push Box off Target** (-1.0): Penalty for pushing a box off a target, discouraging

ineffective actions.

- **Push all boxes on targets** (10.0): Large reward for pushing all boxes onto targets, the primary objective of completing the game.
- **Move into Trap** (-5.0): Penalty for moving into a trap position, guiding the agent to avoid dangerous locations.
- **Push Box into Trap** (-10.0): Severe penalty for pushing a box into a trap, encouraging the avoidance of actions leading to failure.

These reward settings are tailored to optimize the agent's learning process in Sokoban, ensuring efficient and effective policy discovery towards achieving optimal outcomes.

4.4 Method

For this project, I have chosen to implement the SARSA algorithm for solving the Sokoban problem due to its on-policy nature, which allows for more stable learning in dynamic environments.

The reasons for selecting SARSA over other methods include:

- Exploration: SARSA inherently encourages exploration, as it updates Q-values based on the actions taken by the agent. This can help the agent learn more robust strategies in the Sokoban environment, where optimal paths may require exploring multiple routes.
- Training: The SARSA algorithm performs well, along with the Q-Learning and Monte Carlo algorithms, when the total reward obtained is close to 10.
- Performs: Here we use the gym library from open Ai, a powerful library for solving the Sokoban game Sokoban Gym is a grid-world environment, in which the actions of the agent take place in a square space like a chessboard. Each Sokoban Gym is a grid environment, in which the agent's actions take place in a square space like a chessboard. Each] x square on the grid can contain an object such as: agent, box or wall, ... and each object will return the coordinate position in the form of an image.

  Each component will be represented by a symbol below. each coordinate will present for each action:

  - **Action Coordinates:**
    - $0 : (-1, 0)$ (Move Up)
    - $1 : (1, 0)$ (Move Down)
    - $2 : (0, -1)$ (Move Left)
    - $3 : (0, 1)$ (Move Right)

Action	Description
0	Move Up
1	Move Down
2	Move Left
3	Move Right

Table 2: Sokoban Game Actions

Symbol	Description
#	Wall
@	Player
$	Box
.	Goal
T	Special (Trap)

Table 3: Sokoban Game Symbols

The environment is set up so that each state can be controlled and adjusted easily when needed. Coordinates follow the position of the row and column in the field. Suppose the origin coordinate of the agent is defined as $[0, 0]$. In this coordinate system, when the agent moves to the right, its updated position becomes $[0, 1]$. Here, the first value remains 0 (indicating the row), while the second value changes to 1 (indicating the column). In this study, we utilize the Sokoban environment, a popular setting in reinforcement learning where the player must move boxes to target locations within a maze. This environment is structured as a matrix, with states representing the positions of the player and the boxes.

The hyperparameters used in this environment include:

- Learning Rate (α): 0.85
- Discount Factor (γ): 0.90
- Exploration Rate (ϵ): 0.2, 0.5, 0.8
- Number of Episodes: 5000

4.5 Hyperparameters in Sokoban Environment

In this study, the Sokoban environment serves as a fundamental setting within the realm of reinforcement learning, presenting a challenging scenario where the player navigates through a maze to push boxes onto designated target locations. This classic puzzle game is represented as a grid or matrix, where each state encapsulates the positions of the player and the boxes.

The Sokoban environment exemplifies the complexity inherent in many real-world problems by requiring the agent to strategize movement and action sequences to achieve specific objectives.

4.6 Hyperparameter Tuning

Hyperparameter tuning is crucial for optimizing the performance of reinforcement learning algorithms. In the Sokoban environment, the tuning process is essential to help the agent learn effective strategies and improve its decision-making. For each algorithm, we experiment with different values for key parameters such as the learning rate, discount factor, and exploration rate to identify combinations that yield the best outcomes. The learning rate, for instance, affects how quickly the agent updates its understanding of the environment, balancing between fast adaptation and stability. The discount factor influences how much the agent values future rewards, which is critical in a game like Sokoban where rewards are often delayed and require careful planning. Lastly, the exploration rate plays a role in the agent's behavior, determining how much it explores unknown actions versus choosing the best-known actions.

The tuning process involves running multiple training sessions, during which we monitor how quickly the agent's policy converges and how stable its decisions remain over time. By observing these patterns, we can refine our parameter ranges iteratively, focusing on values that consistently lead to effective learning. This iterative approach allows us to make small adjustments, moving toward the optimal settings for better performance in the Sokoban environment. The ultimate goal is to enhance the agent's ability to navigate and interact with the environment with increasing efficiency, learning robust policies that can handle the challenges of Sokoban more effectively.

5 Results of aglorithm

5.1 SARSA Algorithm

I will introduce the SARSA algorithm, which is useful for the Sokoban environment. It not only helps the agent learn effectively but also optimizes the average reward, reducing the time cost and improving performance.

Initialize $Q(s, a)$ arbitrarily for all state-action pairs
for each episode **do**
  Initialize state S
  repeat
    Choose an action A based on epsilon-greedy policy
    Take action A and observe the next state S' and reward R
    Update $Q(S, A)$ using the SARSA update rule:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A) \right]$$

Set $S \leftarrow S'$ and $A \leftarrow A'$
if S is a goal state **then**
End episode
end if
until episode ends
Decay epsilon for exploration control
end for

The SARSA algorithm outperforms the other algorithms in terms of convergence speed and average rewards. After running 5000 iterations, the average reward for the SARSA algorithm reaches 10.98, while the Q-learning and Monte Carlo methods yield average rewards of 2.73 and 0.23, respectively.

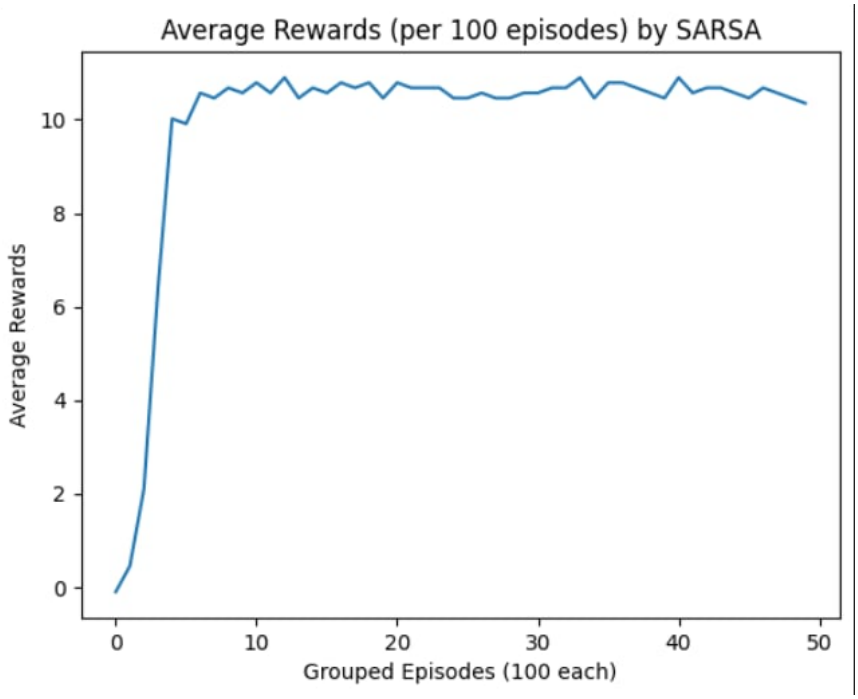

Figure 2: SARSA Algorithm Performance

The SARSA algorithm performs effectively in the Sokoban environment, demonstrating steady convergence and reliable learning progress. After approximately 200 iterations, the algorithm begins to show signs of convergence, gradually achieving an average reward close to 10 by the end of 5000 iterations. This progression suggests that the SARSA agent adapts well to the challenges of Sokoban, learning to navigate complex layouts and interact strategically with obstacles. SARSA's on-policy nature,

which updates based on the agent's chosen actions, enables it to handle the environment's unpredictability with consistency. The algorithm's stable performance implies that it successfully balances exploration and exploitation, allowing the agent to make calculated moves while optimizing for reward. This stability and adaptability make SARSA particularly well-suited for Sokoban, where effective navigation and obstacle management are essential for sustained progress.

5.2 Q-learning Algorithm

Q-learning is one of the most common algorithms in reinforcement learning. in this environment , we also try do opptimizer Q learning table by using this algorithm, which can bring many benefit for trainning agent effeciency.

Initialize $Q(s, a)$ arbitrarily for all state-action pairs
for each episode **do**
  Initialize state S
  repeat
    Choose an action A based on epsilon-greedy policy
    Take action A and observe the next state S' and reward R
    Update $Q(S, A)$ using the Q-learning update rule:

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

    Set $S \leftarrow S'$
    if S is a goal state **then**
      End episode
    end if
  until episode ends
  Decay epsilon for exploration control
end for

The Q-learning algorithm shows decent performance, achieving an average reward of 2.73 after 5000 iterations. Although it runs well, it converges more slowly than the SARSA algorithm, taking around 400 epochs to begin its convergence process.

This line given graph illustrates the average rewards obtained by the Q-learning algorithm in the Sokoban environment, with rewards averaged over groups of 100 episodes. Initially, the rewards are negative, indicating a learning phase where the agent struggles to find optimal moves and incurs penalties. However, from around the second group of episodes onward, there is a sharp increase in rewards, demonstrating

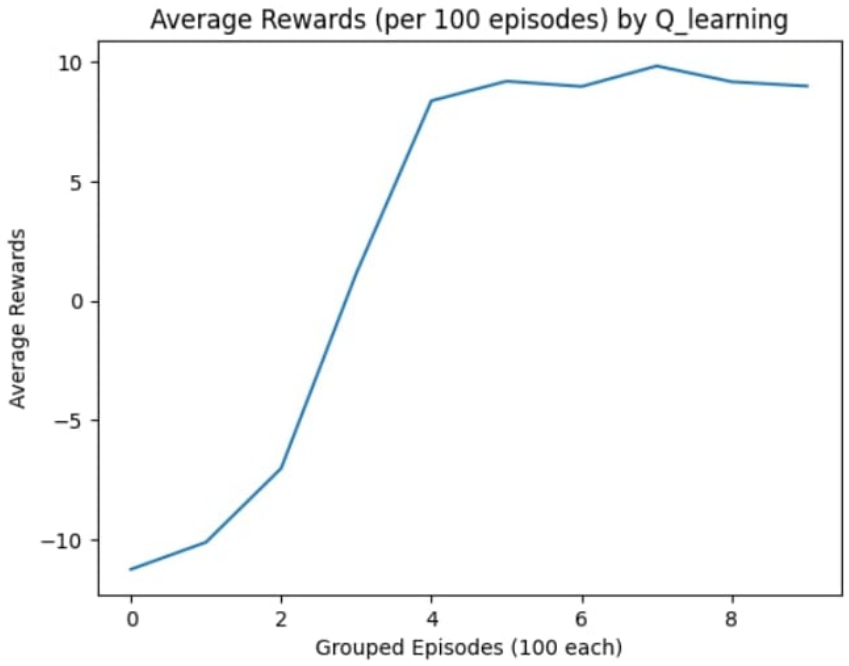

Figure 3: Q-learning Algorithm Performance

significant learning progress as the agent begins to adapt and improve its actions within the environment. By the fourth group of episodes, the rewards stabilize near the maximum, reaching values close to 10, suggesting that the agent has effectively learned to perform well in Sokoban. The slight fluctuations in the latter part of the graph indicate minor variations in performance but overall reflect a stable convergence of the Q-learning algorithm. This trend shows that Q-learning can achieve high performance with sufficient training, though its initial learning phase is slower.

5.3 Monte Carlo Algorithm

After experiencing both algorithms mentioned above, we found that the Monte Carlo method was the preferred approach because we believe it optimizes the average reward and reduces training time

---

Initialize $Q(s, a)$ arbitrarily for all state-action pairs
for each episode **do**
  Initialize state S

Initialize empty list of rewards for the episode
repeat
Choose an action A based on epsilon-greedy policy
Take action A and observe the next state S' and reward R
Append reward R to the list of rewards
Set $S \leftarrow S'$
if S is a goal state **then**
End episode
end if
until episode ends
Compute return G from the rewards list
for each state-action pair (S_t, A_t) in the episode history **do**
Update $Q(S_t, A_t)$ using the Monte Carlo update rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[G - Q(S_t, A_t)\right]$$

end for
Decay epsilon for exploration control
end for

Overall with figure 4, as the result, The Monte Carlo method has the lowest average reward, When this algorithm alway have up and donwn trend through process with average reward be received just around -10 rewar. This indicates that it struggled to converge effectively in the Sokoban environment.

After 5000 episodes, the average reward achieved by the Monte Carlo agent remains low, around -9.5, indicating that the agent struggles to learn an optimal policy under this approach. One of the main challenges with Monte Carlo in this context is its episodic nature; the algorithm updates its policies only after completing an episode, making learning slower, especially when immediate feedback could benefit the agent's learning process. Additionally, because Sokoban's rewards often depend on achieving multiple steps without immediate gains, the delayed reward signal makes it difficult for Monte Carlo to effectively optimize actions, leading to an overall poorer performance compared to other algorithms. This outcome highlights the need for alternative algorithms that update more frequently or can better handle sparse rewards, improving the agent's efficiency and learning speed in complex, multi-step environments like Sokoban.

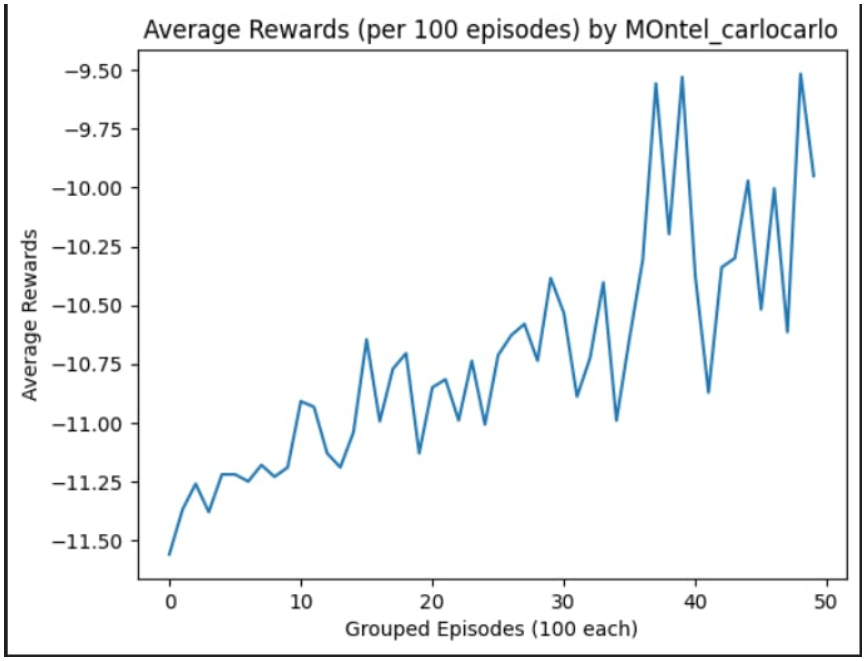

Figure 4: Monte Carlo Algorithm Performance

5.4 Experience with SARSA

The SARSA algorithm is the best-performing algorithm in this environment. To further evaluate its performance, I decided to conduct another episode using different epsilon values: 0.2, 0.5, and 0.8. The convergence speeds across these values are similar.

Epsilon	**Average Reward**
0.2	9.85
0.5	9.95
0.8	9.921

Table 4: Rewards of SARSA with Different Epsilon Values

Moreover, I also obtained the best convergence results for the three epsilon values in the SARSA algorithm. The agent consistently demonstrates efficiency, ensuring a balance between exploration and exploitation.

Figure 5 demonstrates that the SARSA algorithm performs effectively with the use of epsilon-greedy exploration, which balances exploration and exploitation. As depicted, the rewards received by the agent show a smooth convergence around iteration 500,

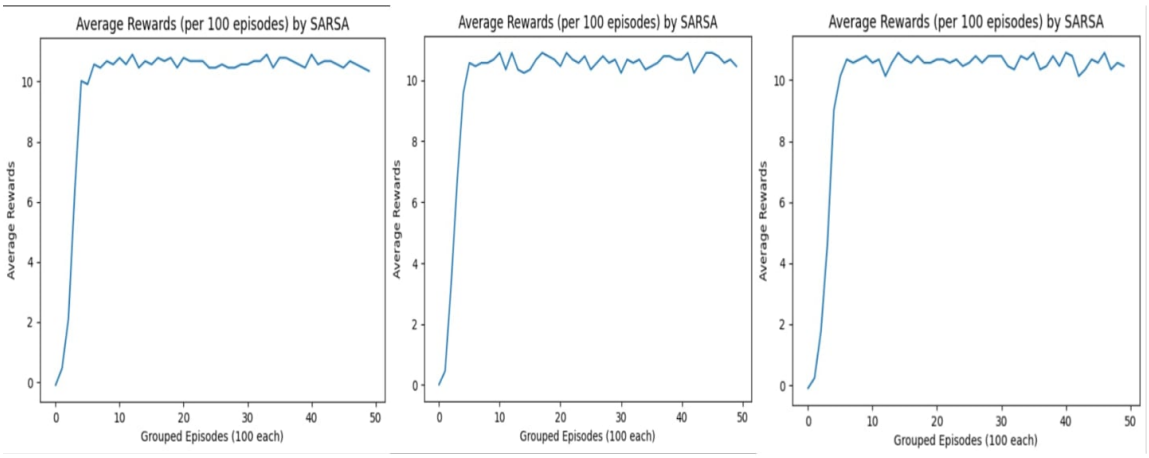

Figure 5: SARSA Performance with Different Epsilon Values

indicating that the algorithm is successfully learning and optimizing its policy. The steady increase in rewards suggests that the agent is effectively navigating the environment and making informed decisions based on past experiences. This behavior reflects SARSA's ability to adapt its strategy over time, reinforcing beneficial actions while reducing less effective ones, ultimately leading to improved performance in the Sokoban environment.

5.5 Experience with Q learning

Q learning is also the best algorithm work well with Sokoban environment but all of the reward be recieved is not hight like Sarsa algorithm working. Be here , i also set up with 3 epsilon value is 0.2, 0.5, and 0.8

Epsilon	**Average Reward**
0.2	2.3
0.5	-2.6
0.8	-0.39

Table 5: Rewards of Q-learning with Different Epsilon Values

For an epsilon value of 0.2, the agent primarily exploits its learned knowledge, resulting in good performance within this environment. However, setting epsilon to 0.8

leads to an overemphasis on exploration, as evidenced by a reward of only approximately −0.39. This indicates that excessive exploration is not suitable for the agent in this context.

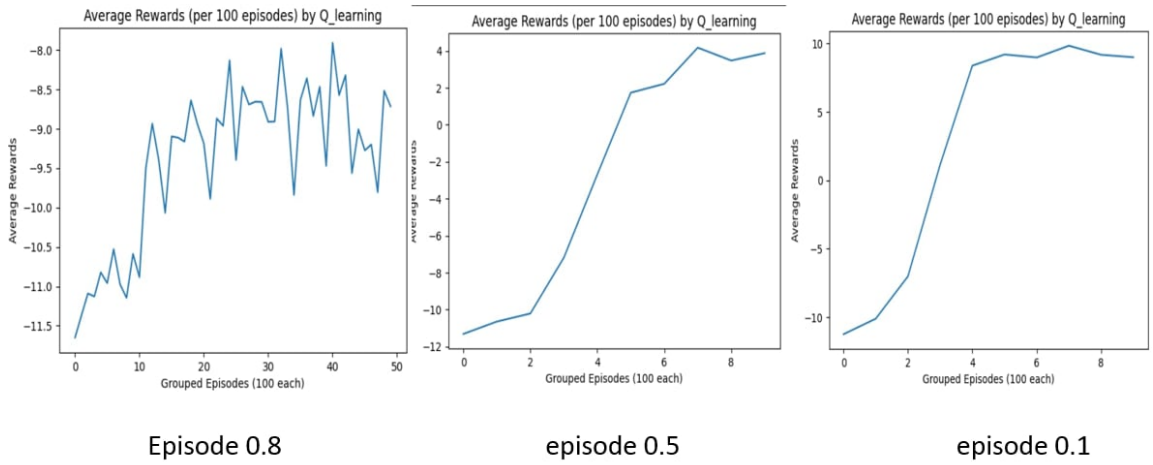

Figure 6: Q-learning Performance with Different Epsilon Values

Figure 6 illustrates that Q-learning functions more effectively with a focus on exploitation rather than exploration. At epsilon 0.8, the reward lines fluctuate consistently, indicating that the algorithm struggles to identify better moves.

5.6 Comparison of Algorithms

In this section, we provide a comparison of the Sarsa, Q-learning, and Monte Carlo algorithms based on three key factors: stability, convergence speed, and the optimal policy discovered. Sarsa demonstrates higher stability compared to Q-learning and Monte Carlo in the Sokoban environment, attributed to its on-policy nature, which allows for smoother adjustments of the Q-values during training. Furthermore, Sarsa exhibits faster convergence, achieving an average reward of 10.98 after 5000 iterations, whereas Q-learning reaches only 2.73, and Monte Carlo achieves approximately 0.23. Additionally, Sarsa finds optimal policies more effectively in Sokoban, particularly in complex state scenarios. This superiority is evident in situations with similar states or multiple actions leading to the same outcome, where Sarsa can adjust Q-values based on the actual actions taken, thereby maintaining a more optimal policy. In contrast, Q-learning may converge to a suboptimal policy in such cases due to updates not reflecting the actual actions performed. Overall, Sarsa stands out as a robust algorithm

in the Sokoban environment, characterized by high stability, rapid convergence, and the ability to discover effective optimal policies compared to other algorithms.Besides the strengths mentioned above, another important factor that makes SARSA stand out from Q-learning and Monte Carlo in the Sokoban environment is its ability to deal with risky states or states that lead to dead ends. (dead-end). Because SARSA updates the Q value based on the actual actions the agent performs, it helps the agent recognize dangerous or ineffective actions in a specific context, thereby making reasonable adjustments during the process. Conversely, Q-learning may not accurately reflect the risk level of actual actions, leading to unstable or ineffective learning of policies. This is especially important in highly complex problems like Sokoban, where carefulness in each move can greatly influence the final result. SARSA demonstrated a good balance between exploration and exploitation, with different epsilon values showing improved performance during the learning process. In contrast, Q-learning, while also showing effectiveness, struggled with convergence, particularly at higher epsilon values, resulting in lower average rewards.

Overall, this study confirms that SARSA is the superior choice for the Sokoban problem, while Q-learning may require further tuning of parameters to achieve optimal performance.

6 Challenge and Improvement

In the Sokoban environment, I have added a feature to receive rewards when approaching the box as quickly as possible. By receiving a reward of +1, the agent is encouraged to adjust its actions to reach the box faster. However, the experimental results indicate that this feature does not significantly improve performance. Training is relatively slow, and after 5000 epochs, the agent only achieves an average reward of 0.82. This suggests that while the idea of rewarding proximity to the box may seem beneficial, it does not sufficiently accelerate the learning process in this context. Because 1 box will have 4 faces corresponding to 4 positions. And I am planning to set up the agent to be able to reach the coordinates of the 2 faces in front of the agent. In order to help the agent reach the box as quickly as possible. Minimize the situation of getting lost and not reaching the box. If multiple boxes are close together, the agent should assess the surroundings to determine which box is the most accessible. It needs to scan for obstacles such as walls, other boxes, or even another agent. By expanding its search around each box, the agent can evaluate the number of free spaces adjacent to it, preferring boxes with more open surroundings for easier movement. Additionally, the agent should be aware of narrow corridors or dead-ends where pushing a box could become difficult or impossible. To further enhance its decision-making, the agent may adjust its epsilon value to favor exploration, allowing it to gather more information about potential obstacles and paths.

this figure 7 show that performance of algorithm not well when appyling the new feature of reaching the fastest crate to the agent. The rewards it receives are almost constantly fluctuating and uneven, the rewards are also only around 0.8. It seems that the agent is having difficulty finding new paths to reach the crate as quickly as possible, combined with having to avoid traps by experimenting. And when tested, the agent gets stuck after completing 1 crate, The training time is also quite expensive for applying the feature.

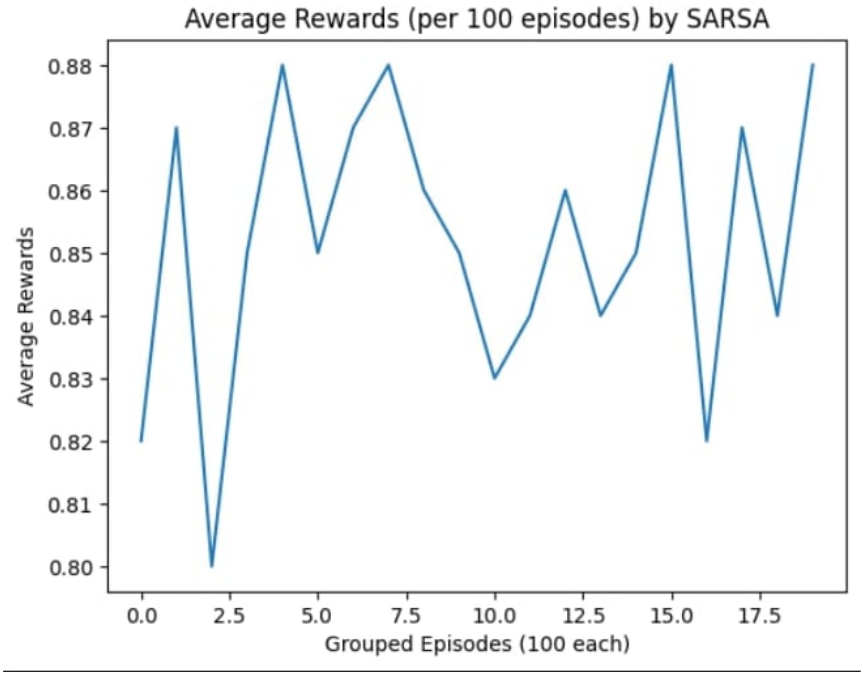

Figure 7: Performance of Sarsa Algorithm in Sokoban Environment

7 Summary

In this project, we compared the performance of traditional reinforcement learning algorithms such as Sarsa, Q-learning, and Monte Carlo in the Sokoban environment. Our findings highlight that Sarsa outperforms the other algorithms in terms of stability and convergence speed, achieving an average reward of 10.98 after 5000 iterations. While Q-learning also ensure converges to the average and the training time is also quite short. Monte Carlo algorithms show slower convergence, they may still have potential in different or less complex environments.

7.1 Lessons Learned

Throughout the project, we gained valuable insights into the selection and application of reinforcement learning algorithms tailored to specific tasks. Our study focused on the Sokoban environment, where the player must strategically move boxes to designated targets. We found that Sarsa, with its on-policy approach, consistently outperformed other algorithms in this dynamic setting. This success highlighted the importance of maintaining policy control during learning, allowing the agent to adjust its actions based on its current policy and achieve stable convergence.

Furthermore, our experiments underscored the critical role of reward structure design in shaping the agent's learning trajectory. By carefully crafting rewards, we observed significant improvements in learning efficiency and final performance. This aspect of reinforcement learning reaffirmed the delicate balance between exploration and exploitation strategies. Our exploration rates (ϵ) of 0.2, 0.5, and 0.8 enabled the agent to explore new actions

7.2 Future Work

For future work, we suggest exploring the use of deep reinforcement learning (DRL) methods such as Deep Q-Networks (DQN) or Proximal Policy Optimization (PPO) to handle more complex environments. These methods can potentially scale better with high-dimensional state spaces and continuous action spaces that are common in dynamic environments. Additionally, testing these traditional RL algorithms in dynamic and multi-agent environments may reveal new insights. Improving reward functions and fine-tuning hyperparameters, such as exploration-exploitation trade-offs, discount factors, and learning rates, may further enhance agent performance in the Sokoban environment.

We also plan to implement a feature that allows two agents to collaborate within the environment. This will enable them to communicate about their positions, the locations of boxes, and any obstacles they encounter. By coordinating their actions, they can maximize efficiency in tasks like pushing boxes. For example, if one agent is blocked by an obstacle, the other can navigate around to assist or clear the path. This communication will enhance their problem-solving abilities, making the agents more effective in exploring and interacting with the environment together. Overall, this feature aims to create a more dynamic and cooperative experience.

Finally, exploring transfer learning between Sokoban and other puzzle-solving environments could provide valuable insights. By pre-training agents on one type of environment and fine-tuning them for Sokoban, we might discover patterns or strategies that can be transferred across similar tasks, enhancing the adaptability and robustness of the agents.

References

[1] Jia, J., & Wang, W. (2020). Review of reinforcement learning research. In *2020 35th Youth Academic Annual Conference of Chinese Association of Automation (YAC)* (pp. 186-191). IEEE. https://doi.org/10.1109/YAC51587.2020.9337653

[2] Seshagiri, S., & V, P. K. (2023). An empirical study of on-policy and off-policy actor-critic algorithms in the context of exploration-exploitation dilemma. In *2023 International Conference on Emerging Techniques in Computational Intelligence (ICETCI)* (pp. 238-243). IEEE. https://doi.org/10.1109/ICETCI58599.2023.10331400

[3] Garcia, P. (2001). Exception-based reinforcement learning. In *IECON'01. 27th Annual Conference of the IEEE Industrial Electronics Society (Cat. No.37243)* (Vol. 3, pp. 2074-2079). doi:10.1109/IECON.2001.975612

[4] Peng, S. (2020). Overview of Meta-Reinforcement Learning Research. In *2020 2nd International Conference on Information Technology and Computer Application (ITCA)* (pp. 54-57). doi:10.1109/ITCA52113.2020.00019

[5] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. Retrieved from https://arxiv.org/abs/1707.06347

[6] Vu, H.-D., Cong, P.-K. P., Hoang, N.-M., Tran, K., Ngo, D. M., & Pham, D.-D. (2024). Push and Pull Robot with Reinforcement Learning Algorithms. In *2024 9th International Conference on Integrated Circuits, Design, and Verification (ICDV)* (pp. 219-224). IEEE. https://doi.org/10.1109/ICDV61346.2024.10617147

[7] Wurm, M. F., & Lingnau, A. (2015). Simultaneously learning at different levels of abstraction. *Journal of Neuroscience*, 35, 7727–7735.

[8] Chen, R., & Wu, F. (2023). Dynamically interrupting deadlocks in game learning using multisampling multiarmed bandits. *IEEE Transactions on Games*, 15(3), 360-367. https://doi.org/10.1109/TG.2022.3177598

[9] Zhao, D., Wang, H., Shao, K., & Zhu, Y. (2016). Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 1-6). IEEE. https://doi.org/10.1109/SSCI.2016.7849837

[10] Jeerige, A., Bein, D., & Verma, A. (2019). Comparison of deep reinforcement learning approaches for intelligent game playing. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* (pp. 0366-0371). https://doi.org/10.1109/CCWC.2019.8666545