Digital Object Identifier

Final Project: Sokoban Solving using Reinforcement Learning

NGO MANH DUY-21010298[1], HOANG NHAT MINH-21013299[1], TRAN VAN KHAI-21012331 [1],
[1]Falcuty of Electrical and Electronic Engineering, Phenikaa University

ABSTRACT Sokoban, a classic puzzle game crafted by Hiroyuki Imabayashi, serves as an intricate testbed for Reinforcement Learning (RL) algorithms. This deceptively simple game demands foresight, strategic planning, and meticulous execution. RL, a subset of machine learning, empowers intelligent agents to make decisions that maximize rewards. In the context of Sokoban, RL offers a promising approach for devising effective strategies to navigate obstacles, move boxes to target locations, and address dead-ends. This study explores the application of five RL algorithms—Value Iteration, Policy Iteration, Monte Carlo, Q-learning, and SARSA—to solve Sokoban puzzles. The evaluation of these algorithms within the Gym Sokoban environment, along with tailored modifications, unveils their unique strengths and limitations. Notably, the analysis encompasses both traditional Sokoban maps and custom-created challenges, such as a unique box that grants the agent the ability to pull boxes backward. The results demonstrate the promise of SARSA, Q-learning, and Monte Carlo in effectively solving Sokoban maps, underlining their adaptability to diverse RL and game-solving challenges. The study concludes with a commitment to explore advanced RL techniques to further enhance problem-solving in this game. Link to the GitHub repository: https://github.com/ngoManhDUY/Sokoban-Solver-using-Reinforcement-Learning.git.

INDEX TERMS
Sokoban, Reinforcement Learning(RL), Dynamic Programming (DP), Policy iteration, Value iteration, Monte Carlo, Q-learning, SARSA.

I. INTRODUCTION

Sokoban, a classic puzzle game, originated from the creative mind of Hiroyuki Imabayashi. This game presents an environment that necessitates a delicate interplay of foresight, meticulous planning, and strategic acumen. With its grid-based layout, discrete action space, and inherent complexity due to sparsity, Sokoban poses a formidable challenge, making it an ideal testbed for the application of Reinforcement Learning (RL) algorithms.

Sokoban's premise is deceptively simple: the player must maneuver boxes to designated target locations within a confined space. However, beneath its straightforward rules lies a profound complexity that renders it an enticing problem for exploration within the realm of reinforcement learning (RL). Reinforcement Learning, a subfield of machine learning, is dedicated to training intelligent agents to make sequences of decisions that maximize cumulative rewards. In the context of Sokoban, RL offers a promising avenue for enabling agents to acquire effective strategies for pushing boxes to their designated positions while deftly navigating around obstacles and potential dead-ends.

The primary objective of this project is to assess the efficacy of five distinctive RL algorithms—Dynamic Programming(Value Iteration, Policy Iteration), Monte Carlo, Q-learning, and SARSA—in solving the Sokoban puzzle, conducted within the Gym Sokoban environment, with some tailored modifications. We aim to illuminate the unique strengths and limitations of each algorithm and, by extension, provide insights into their adaptability to diverse RL and game-solving challenges. Notably, our exploration doesn't merely encompass solving traditional Sokoban maps; we venture into the realm of custom-created maps, featuring unique elements such as a special box that, when occupied by the agent, confers the ability to pull boxes backward.

This paper is organized as follows: Section 2 introduces our Sokoban environment. Section 3 provides an overview of the five algorithms we employed to solve the problem. Section 4 presents the experiments conducted to evaluate the performance of these algorithms. Section 5 concludes the paper, summarizing key findings and suggesting future research directions.

’0’	no operation		
’1’	push up	’7’	move left
’2’	push down	’8’	move right
’3’	push left	’9’	pull up
’4’	push right	’10’	pull down
’5’	move up	’11’	pull left
’6’	move down	’12’	pull right

TABLE 1. Actions.

II. SOKOBAN ENVIRONMENT

Before the agent can learn very complex algorithms, it needs to have a place to stand and interact. The closest environment we can compare is Gym-Sokoban [5]. Although the gym’s environment is pretty good it still has a lot of issues we have to deal with to improve flexibility and increase performance when just using pure reinforcement learning algorithms. The number of actions is quite a lot which counts to 13 actions 1. And one more important thing is the reward the agent gets whenever the action is done:

Action	Reward
Each step	-0.1
Off target	-1
On target	+1
Finished	+10

The first is to have better flexibility. We are setting and generating the environment based on Google-Deepmind’s Sokoban [6] which each puzzle is a 10 by 10 ASCII string that uses the following encoding:

’#’	wall
’@’	player
’$’	box
’.’	goal
’?’	special

So the map and environment are more flexible to adjust. Throw that flow, we created 6 signature maps that are very iconic in Figure 1. Not just how the map looks, we have to create a special game mode in which the agent can get one more pull time whenever he "eats" the mushroom. Firstly we create the pull function in a simple concept: When the agent pulls, both the box and the agent will move in the same direction. Next, we create a variable "num_pull" that stores how many times the agent can pull, if it is zero, the agent can’t pull.

Our final adjustment to the environment is to get better performance while training by decreasing the number of states and changing how the agent stores information after interacting with the environment, especially since the observation is encoded to a number, not an RGB array in gym-sokoban 5. We encoded the state by predicting every possible state in a movable zone by calculating permutation ${}_nP_k$ (in which n is are movable zone and $k = 1 + num\_boxes$, 1 are agent number) and replaced the position of agent and boxes. Finally, we encode with just a simple concept: $state = index(all\_state)$.

III. REINFORCEMENT LEARNING ALGORITHM

A. MARKOV’S DECISION PROCESS

1) Definition MDP

A **Markov Decision Process (MDP)** is a mathematical framework used in artificial intelligence and operations research to model decision-making processes [1]. It provides a mathematical framework for modeling decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. MDPs are useful for studying optimization problems solved via dynamic programming. At each time step, the process is in some state, and the decision maker may choose any action that is available in the state. The process responds at the next time step by randomly moving into a new state and giving the decision-maker a corresponding reward. The probability that the process moves into its new state is influenced by the chosen action.

An MDP is defined by a 4-tuple $(S, A, P, R,)$, with $\gamma \in [0,1]$, where:
- S is the set of all states.
- A is the set of all actions available.
- P is the transition probability from one state to another g
- R is the reward function obtained from transitioning from one state to another.
- γ is the discount factor.

$R : S * A \longrightarrow R$ is the reward function.

Figure 2 represents the iteration between the agent and environment. At any given time step t, the agent must base their action, a_t, on the current state of the environment, s_{t+1}, and the agent receives a reward, R_t, from the environment. This cycle of events yields a trajectory (a sequence of states, actions, and rewards) akin to the one depicted in **Figure 3**.

2) Sokoban MDP

To frame Sokoban as a MDP problem, we need to identify all possible states (S), all possible actions (A), the transition function (T), and the reward function (R). This implementation of a Sokoban gym[1] environment also relies on the maximum number of steps the agent is allowed to take, with this parameter being an alternate ending condition of the simulation.
The role of Markov decision process (MDP) in Sokoban game is to model the decision-making of the player (*agent*) who tries to push the boxes (*object*) to the goal locations (*target*) in a grid-like environment. MDP can help the agent learn the optimal policy (*strategy*) to solve the game by maximizing the expected reward (*score*) over time. (in **Sokoban environment**)

B. VALUE ITERATION

The Value Iteration Algorithm is a **dynamic programming algorithm** that can be used to find the optimal policy for

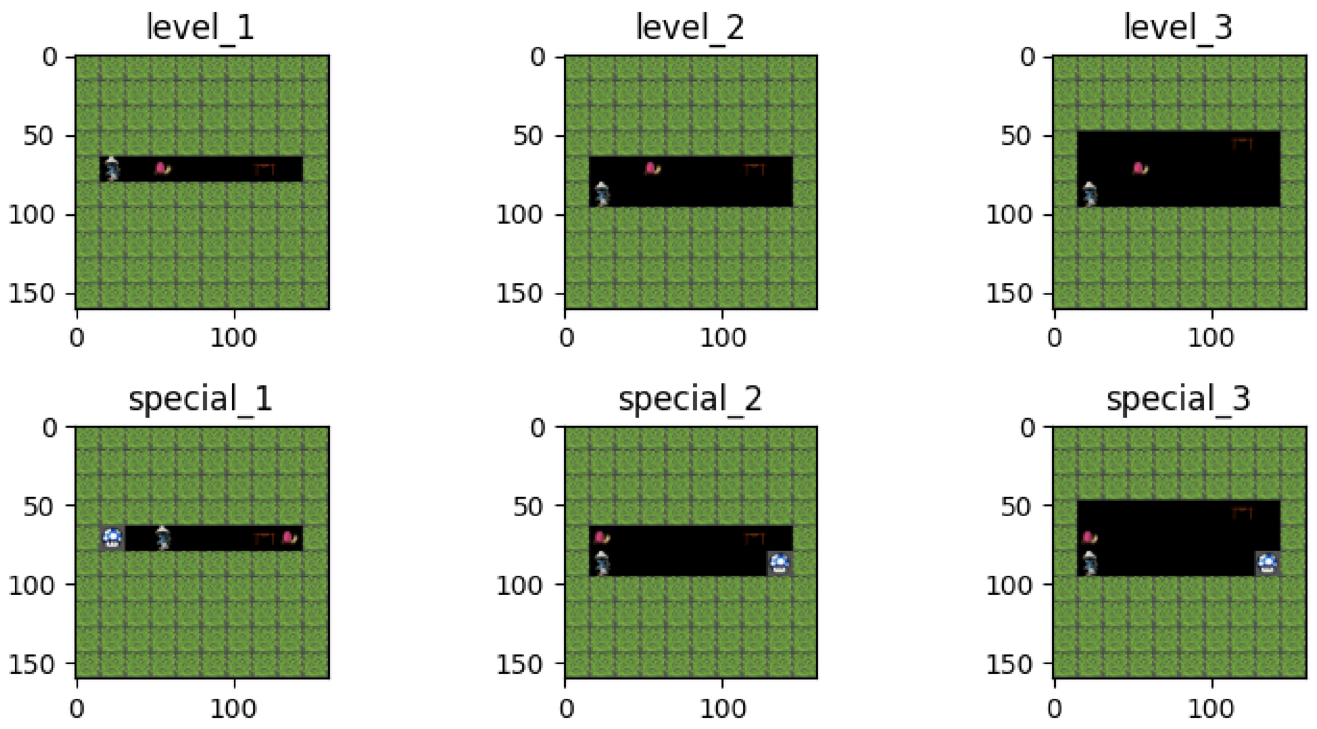

FIGURE 1. All map

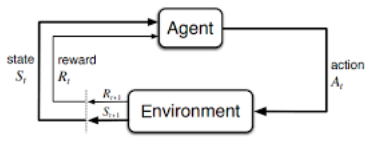

FIGURE 2. MDP flowchart cycle

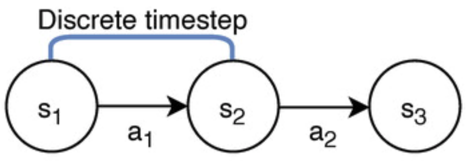

FIGURE 3. MDP trajectory

Algorithm 1 Value Iteration Algorithm, for estimating $\pi = \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in S^+$, arbitrarily except that V (terminal)=0
loop
    $\Delta \leftarrow 0$
    loop
        each $s \in S$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow max_a \Sigma_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
    end loop
    until $\delta < \theta$
end loop
Output a deterministic policy, $\pi \approx \pi_*$,such that
$\pi(s) = argmax_a \Sigma_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
```

a Markov Decision Process (MDP). In Sokoban game, the algorithm can be used to find the optimal policy for the game by finding the optimal value function for each state in the game. It also determines which moves will lead to the highest reward and which moves will lead to lower rewards.
This **psedocode** below shows a complete algorithm with this kind of termination condition (**Algorithm 1**)

C. POLICY ITERATION

1) Definition

The Policy Iteration Algorithm (combination of Policy Evaluation and Policy Improvement) is another dynamic programming algorithm that can be used to find the optimal policy . In Sokoban game, the algorithm can be used to find the optimal policy for the game by iteratively improving the value function and policy until convergence.
Policy Evaluation is the process of determining the value function for a given policy. The value function is the expected

long-term reward for each state under a given policy.
Policy Improvement is the process of improving the current policy by making it greedy with respect to the value function. This means that at each state, we choose the action that has the highest expected long-term reward according to the current value function.

2) Formula

Once a policy, π, has been improved using v_π to yield a better policy, π' , we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} v_* \tag{1}$$

where $\xrightarrow{E}$ denotes a *policy evaluation* and $\xrightarrow{I}$ denotes a *policy improvement*
A complete algorithm is **pseudocode** given below (**Algorithm 2**)

Algorithm 2 Policy Iteration Algorithm

```
1. Initialization
$V(s) \in R$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
$V(terminal) = 0$
2. Policy Evaluation
loop
    $\Delta \longleftarrow 0$
    Loop for each $s \in S$ :
    loop
        $v \leftarrow V(s)$
        $V(s) \leftarrow \Sigma_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
    end loop
    until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
end loop
3. Policy Improvement
policy-stable $\leftarrow$ true
For each $s \in S$:
loop
    old-action $\leftarrow \pi(s)$
    $\pi(s) \leftarrow argmax_a \Sigma_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    If old-action $\neq \pi(s)$, then policy-stable $\leftarrow$ false
end loop
If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

D. FIRST VISIT MONTE CARLO

We define the Monte Carlo method in this project as stochastic. Monte Carlo is a model-free method for learning the state-value function. Under the term “model-free method”, we understand a method that does not require a piece of prior information about the state transition probabilities. The value of each state is the total amount of the reward an agent can expect to accumulate over the future, starting from that state. The First-Visit Monte Carlo method estimates this value function by averaging the returns following all first visits to a state. The state-value function is defined as:

$$v_\pi(s) = E_\pi[G_t|S_t = s] \tag{2}$$

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-1} R_T. \tag{3}$$

We have:
π: policy.
G_t: weighted return.
S_t: state at the time step t.
s: state.
$\gamma \in [0; 1]$: discount rate.
R_t: reward obtained at time step t.
So we can simplify this equation to:

$$V(s) \leftarrow V(s) + \frac{1}{N(s)}(G_t - V(s)) \tag{4}$$

Which $N(s)$ is the number of times the agent visits state s. So, after looping in a very large amount of time, we can extract the optimal policy π so the agent can complete the puzzle.

E. Q-LEARNING ALGORITHM

Q-Value Update Rule: The Q-value for a particular state-action pair is updated based on the Q-learning formula:

$$Q(s, a) = (1-\alpha)\cdot Q(s, a)+\alpha\cdot(R + \gamma \cdot \max(Q(s', a'))) \tag{5}$$

where:
$Q(s, a)$: Q-value for state s and action a.
α: Learning rate, determining the influence of new information on the current estimate.
R: Immediate reward obtained after taking action a in state s.
γ: Discount factor, reflecting the agent’s preference for immediate rewards over delayed ones.
$Q(s', a')$: The maximum Q-value for the next state s' among all possible actions a'.
Epsilon-Greedy Policy: Q-learning employs an epsilon-greedy exploration policy. With probability ϵ, the agent explores by selecting a random action, and with probability $(1 - \epsilon)$, it exploits its current knowledge by selecting the action with the highest Q-value.
In summary, Q-learning aims to iteratively update Q-values for state-action pairs based on the rewards received and the maximum expected future rewards. Over time, these updates converge to the optimal Q-values that guide the agent toward making the best decisions in the given environment. The epsilon-greedy strategy balances exploration (trying new actions) and exploitation (choosing known best actions) to gradually improve the policy.

F. SARSA

SARSA is an on-policy reinforcement learning algorithm that aims to estimate the optimal action-value function Q(s, a), just like Q-learning. The key difference is in how it updates Q-values based on the interaction of the agent with the environment. Here is the SARSA algorithm formula:

$$Q(s,a) = (1-\alpha)\cdot Q(s,a)+\alpha\cdot(R+\gamma\cdot\max(Q(s',a'))) \quad (6)$$

where:

$Q(s,a)$: Q-value for state s and action a.

α: Learning rate, controlling the step size of the Q-value updates.

R: Immediate reward obtained after taking action a in state s.

γ: Discount factor, emphasizing the importance of future rewards.

$Q(s',a')$: Q-value for the next state-action pair (s',a').

Key Differences between SARSA and Q-learning:

1. **On-Policy vs. Off-Policy:** - SARSA is an on-policy algorithm, meaning it learns the Q-values for the policy it follows during exploration. - Q-learning is an off-policy algorithm, which learns Q-values for an optimal policy but often explores using an epsilon-greedy policy.

2. **Q-Value Update:** - SARSA updates the Q-value based on the action taken in the next state according to the exploration policy. - Q-learning updates the Q-value based on the action that maximizes the Q-value in the next state regardless of the actual action taken during exploration.

3. **Exploration vs. Exploitation:** - SARSA uses the same epsilon-greedy exploration policy for both action selection and Q-value updates. - Q-learning typically explores using epsilon-greedy but exploits the maximum Q-value during Q-value updates.

4. **Stability and Convergence:** - SARSA tends to be more conservative and may converge to a more stable policy. - Q-learning can sometimes be more aggressive in exploration, which may lead to instability in learning but can result in faster convergence to an optimal policy.

In summary, SARSA and Q-learning are both popular reinforcement learning algorithms that estimate optimal action-value functions. The key difference lies in their approach to exploration and how they update Q-values, with SARSA following the policy it's learning about and Q-learning learning an optimal policy while exploring using a different strategy.

IV. EXPRERIMENT AND EVAULATION

In this section, we embark on the implementation of all five algorithms to tackle the Sokoban game with two maps: level 3 and special 3. Our evaluation metrics include tracking the Q-value differences across training episodes for Monte Carlo and monitoring the return values for each training episode with varying algorithm epsilon parameters. For both Q-learning and sarsa, we just set the learning rate equal to 0.1 and the discount factor equal to 0.9. Regrettably, we have excluded the results of Value Iteration and Policy Iteration due to their less favorable performance in this context.

This section is dedicated to the practical implementation and analysis of our selected reinforcement learning algorithms, offering insights into their efficacy in addressing the Sokoban challenge. Our focus lies in understanding the behavior of these algorithms through Q-value comparisons and return value assessments, all while acknowledging the limitations observed in the Value Iteration and Policy Iteration methods. Here are some results:

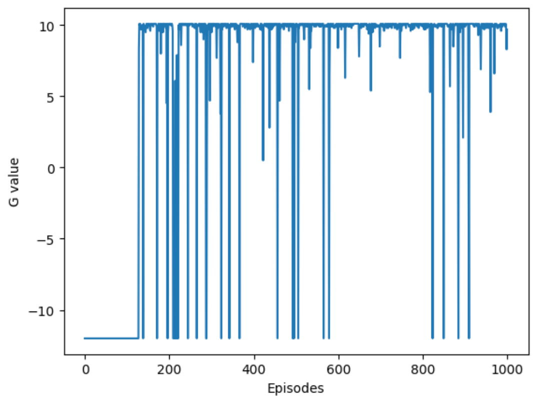

FIGURE 4. **Monte Carlo | Level-3 | epsilon = 0.1**
SS

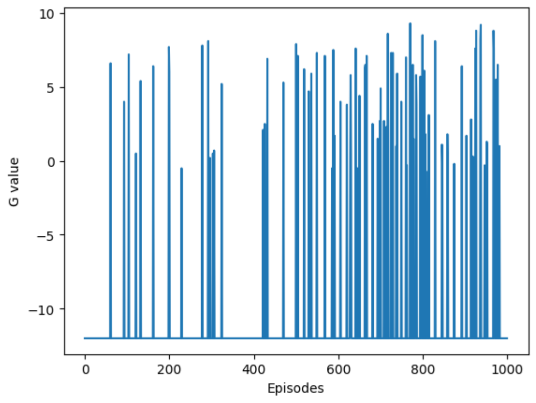

FIGURE 5. **Monte Carlo | Level-3 | epsilon = 0.9**

Based on all the training result, we can easily see that the algorithm show the best performance is the Sarsa algorithm while the result of Monte Carlo and Q-learning show promising if we increase the number of training episodes. Here is the link to the video in which we used the Sarsa algorithm with 3000 training episodes to solve all the Sokoban environment maps: https://www.youtube.com/shorts/LjkShr2jGhs.

V. CONCLUSION

In this study, we have successfully implemented all five fundamental Reinforcement Learning algorithms: Value Iteration, Policy Iteration, Monte Carlo, Q-learning, and SARSA.

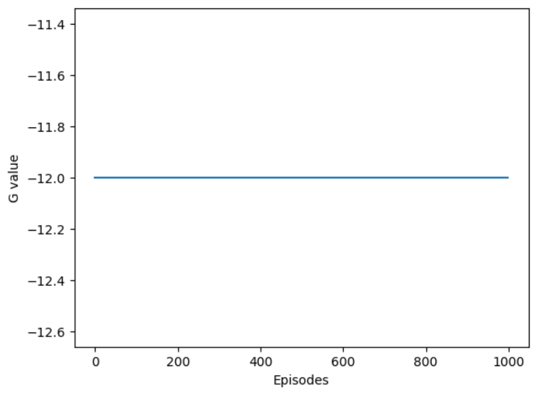

FIGURE 6. **Monte Carlo | Special-3 | epsilon = 0.1**

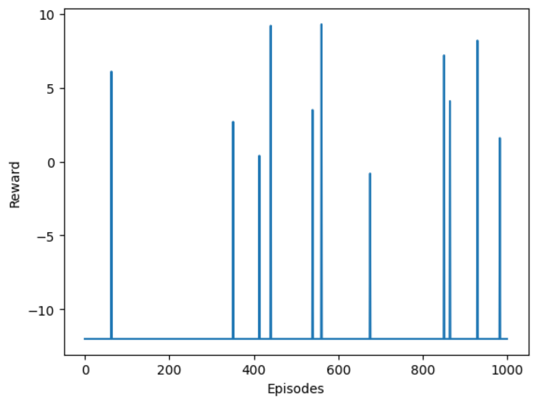

FIGURE 9. **Q-learning | Level-3 | epsilon = 0.9**

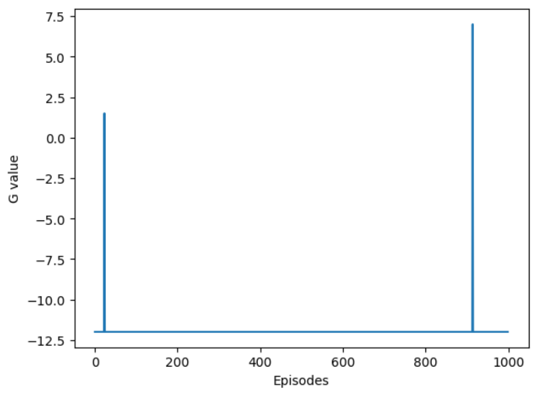

FIGURE 7. **Monte Carlo | Special-3 | epsilon = 0.9**

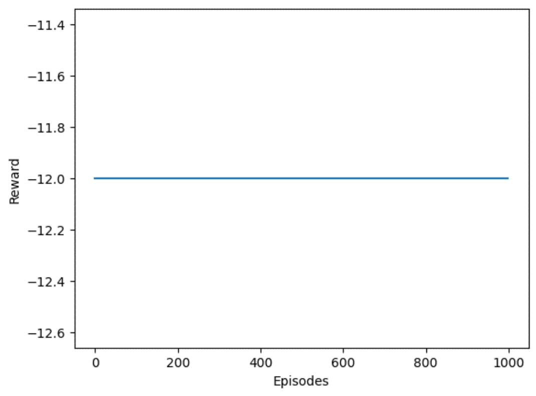

FIGURE 10. **Q-learning | Special-3 | epsilon = 0.1**

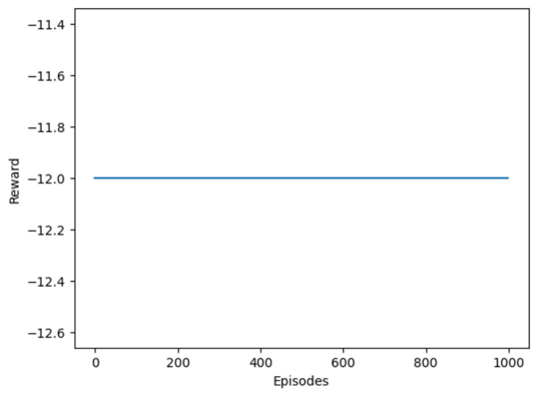

FIGURE 8. **Q-learning | Level-3 | epsilon = 0.1**

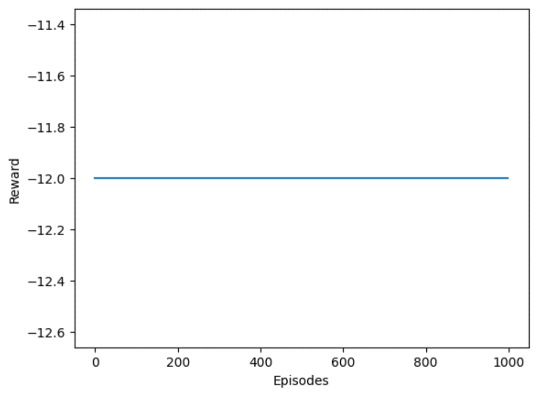

FIGURE 11. **Q-learning | Special-3 | epsilon = 0.9**

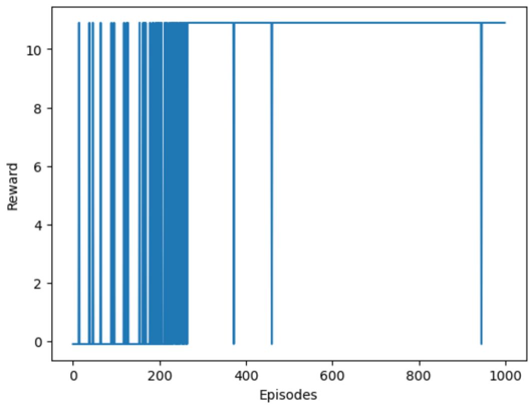

FIGURE 12. **SARSA | Level-3 | epsilon = 0.1**

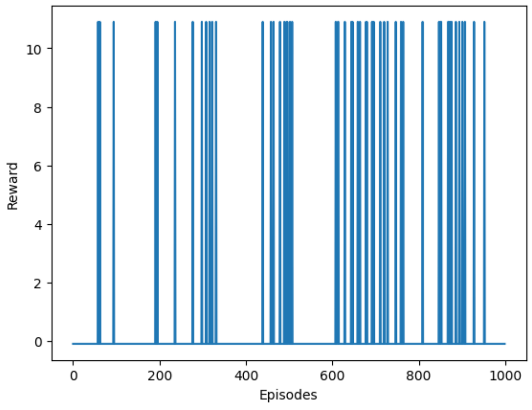

FIGURE 13. **SARSA | Level-3 | epsilon = 0.9**

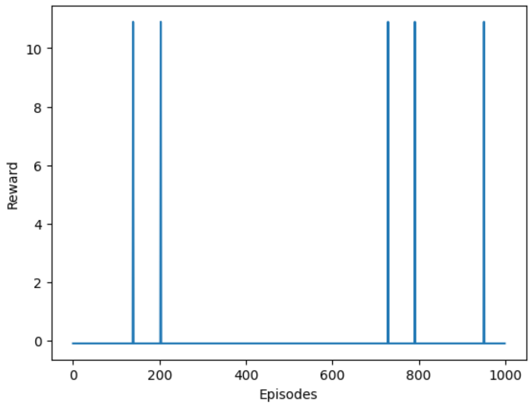

FIGURE 14. **SARSA | Special-3 | epsilon = 0.1**

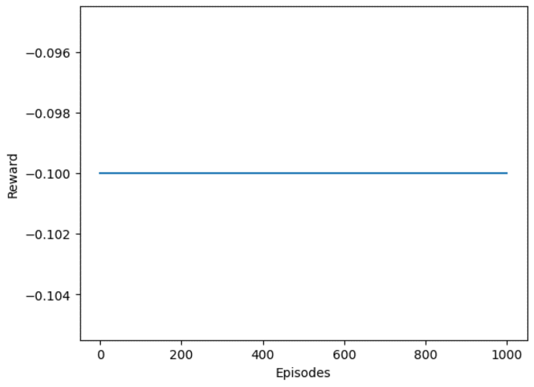

FIGURE 15. **SARSA | Special-3 | epsilon = 0.9**

Following the experimental and training processes, the outcomes of SARSA, Q-learning, and Monte Carlo have shown promising results in effectively solving both types of Sokoban maps, including the traditional and standard maps. Moreover, we employed the most successful algorithm, SARSA, to address all six maps, with consistent success. Looking ahead, we plan to explore advanced Reinforcement Learning techniques to further enhance our approach in tackling this game.

REFERENCES

[1] Richard S.Sutton and Andrew G.Barto, "Reinforcement Learning: An Introduction", 2nd edition.
[2] Sudharsan Ravichandiran, "Deep Reinforcement Learning with Python", 2nd edition.
[3] GAO Huaxuan, HUANG Xuhua, LIU Shuyue, WANG Guanzhi, ZHONG Zixuan (2017). A Sokoban Solver Using Multiple Search Algorithms and Q-learning. GitHub: https://github.com/XUHUAKing/sokoban-qlearning.git
[4] Ethan Heavey (2023). Sokoban-rl. GitHub: https://github.com/VeiledTee/sokoban-rl.git
[5] Gym-Sokoban. GitHub: https://github.com/mpSchrader/gym-sokoban.git
[6] Guez, A., Mirza, M., Gregor, K., Kabra, R., Racanière, S., Weber, T., . . . & Lillicrap, T. An investigation of model-free planning. In Proceedings of the 36th International Conference on Machine Learning pp. 2424-2433, 2019. GitHub: https://github.com/google-deepmind/boxoban-levels