Dynamic Programming and Reinforcement Learning Algorithms on Chutes and Ladders

Tung Luu, Duc Doan, Uyen Le, Phineas Pham

April 2023

Abstract

Chutes and Ladders is a classic childhood game in which players aim to move their token from Square 0 to Square 100 on the board by rolling dice. In the original version, strategic decision-making is not necessary, so we add a strategic element to the game by having players choose among four Efron dice with different possible moves. The objective is to determine which dice to play at each square to minimize the future number of steps. Using four different frameworks, including Dynamic Programming Solution, Monte Carlo, SARSA, and Q-Learning, we find that the optimal policy allows us to win in an average of 10 steps. However, Q-Learning outperforms the other three algorithms with the smallest expected number of steps, the smallest result variation, and shortest time to reach an optimal policy.

1 Introduction

Chutes and Ladders is a popular board game that is played on a board with 100 squares, each numbered 1 to 100. Players start at the bottom left corner of the board and take turns rolling a die with numbers 1 through 6, or spinning a spinner. Along the board, ladder squares allow players to climb to the top of the ladder when landed on, as well as chute squares that send players down to the bottom of the chute. The first player to reach Square 100 wins the game. The simple yet exciting gameplay and elements of chance make the game a favorite among children and families alike.

In this paper, we explore an adapted version of Chutes and Ladders that incorporates strategic decision-making to add an additional layer of complexity and challenge. Specifically, this version offers players four Efron (specified in Table 1 below) for each move. The objective is to strategically choose a dice with the highest likelihood of avoiding a chute or reaching a ladder to reach Square 100 in the fewest number of moves. Now that the new strategic element is introduced, we plan to employ four different algorithms to "solve" the game,

which includes finding the optimal policy for selecting which dice for each possible square of the board and determining the expected minimum number of moves required to complete the game.

Dice	**Face**					
Black	4	4	4	4	0	0
Red	6	6	2	2	2	2
Green	5	5	5	1	1	1
Blue	3	3	3	3	3	3

Table 1: Four Efron dice and their faces.

First of all, we use a Dynamic Programming approach that relies on computing the state transition probability matrix to solve Chutes and Ladders offline. Secondly, we utilize the Monte Carlo method to generate random simulations of the game and obtain estimates of the expected reward associated with each state-action trajectory. Finally, we apply two reinforcement learning techniques, SARSA and Q-Learning, to find the optimal policy and maximum reward. These algorithms leverage both the Dynamic Programming frameworks and Monte Carlo simulations to efficiently explore the state and action spaces without having to compute the reward for every possible combination of states and actions.

According to our findings, all four algorithms can reach the final state by following the optimal policy in an average of 10 to 11 steps. However, for complex problems such as Chutes and Ladders, the Dynamic Programming and Monte Carlo algorithms are not well-suited because they require heavy computation to produce the full trajectory and transition matrix for each policy. SARSA and Q-Learning are more efficient for this type of problem. Nevertheless, our results indicate that Q-Learning reaches the optimal solution in fewer training epochs than SARSA and has a smaller result variation, which suggests that Q-Learning may be a better choice for the modified version of Chutes and Ladders.

2 Background

2.1 Dynamic Programming

Dynamic programming is an offline algorithm for solving decision-making problems, which means that it assumes that all the necessary information about the problem is available before the computation starts. The goal of dynamic programming is to find the optimal value function that represents the expected long-term reward that an agent can obtain starting from a given state and following an optimal policy.

To find the optimal policy, the dynamic programming algorithm uses a bottom-up approach and iteratively computes the optimal value function for

each state by recursively combining the optimal values of the next states. In the context of decision-making problems, the algorithm must determine the state transition matrix that corresponds to the maximum possible reward, where the state transition matrix describes the probability of moving from one state to another following a particular action. Then by making small changes to one or more of the elements in the state transition matrix, we can find a local maximum of the reward value function. Once the optimal value function is obtained, the optimal policy can be computed by selecting the action that maximizes each state's expected sum of rewards.

Dynamic programming is a powerful method for solving problems like Chutes and Ladders due to its ability to find the optimal policy that maximizes the expected cumulative reward over time. However, the algorithm can be computationally expensive, especially for problems with a large number of states and actions. Additionally, it assumes that the agent has full knowledge of the environment, which may not be feasible or practical in real-world applications.

2.2 Monte Carlo

The Monte Carlo algorithm is an online reinforcement learning technique that uses random sampling to estimate the expected reward of a policy for an agent in a given environment. This estimation is based on simulating multiple trajectories of the agent interacting with the environment, where each trajectory consists of a sequence of states, actions, and rewards. By sampling these trajectories and averaging the rewards obtained, the algorithm can gradually improve the estimate of the expected reward for each state-action pair and thus learn the optimal policy for the agent to follow.

In particular, the algorithm calculates the Q value for each trajectory generated, which is the expected sum of future rewards associated with a state and an optimal policy. The Q value corresponding to a state and action is simply the total reward at that state/action divided by the number of times the model visits this state/action:

$$Q(s, a) = \frac{total}{count}$$

The Q value then gives us the V value, which represents the expected future reward at the current state of the model, follows a specific policy thereafter, and the optimal policy P to play the game:

$$V(s) = max\{Q(s, a)\}$$
$$P(s) = argmax\{Q(s, a)\}$$

The full set of Q-values (state/action values), V values (state values), and Policy (strategy) give us the solution to decision-making problems.

An advantage of Monte Carlo methods is that they are model-free, meaning they do not require knowledge of the transition probabilities and rewards of the environment. This makes Monte Carlo methods more flexible and suitable for real-world problems where the environment may be complex and difficult to model accurately. However, Monte Carlo methods can also be computationally expensive, especially when dealing with long trajectories, and their performance may be sensitive to the choice of exploration strategy. Furthermore, Monte Carlo methods require a large number of samples to provide accurate estimates, which can make them impractical in real-world settings.

2.3 SARSA

SARSA is an on-policy reinforcement learning algorithm that leverages the concept of Temporal Difference (TD) into Dynamic Programming frameworks. The acronym SARSA stands for State-Action-Reward-State-Action, which describes the process of learning an optimal policy by observing the rewards associated with taking some action in a given state. Like Monte Carlo, we define the Q-value of a state-action pair as the expected cumulative reward that the agent can obtain by taking that action in that state and following a particular policy from that point onward. The ultimate goal is to determine which action maximizes the Q value at a given state through exploration and observation.

During the learning process, SARSA follows an epsilon-greedy policy that balances exploration and exploitation. With a probability of (1 - epsilon), the agent takes the action that maximizes the cumulative reward $Q(s, a)$ corresponding to state s and action a, while with a probability of epsilon, the agent chooses a random action to explore new possibilities. For each action a at state s, the corresponding Q-value is updated using the following formula

$$Q(s, a) = Q(s, a) + \alpha \ . \ TD_{error}$$

where the temporal difference error is defined as the difference between the observed reward $Q(s, a)$ and the expected reward $Q(s', a')$:

$$TD_{error} = [r + Q(s', a')] - Q(s, a)$$

The learning rate α determines the degree to which the new information affects the Q-value. The SARSA algorithm iteratively updates the Q-value table until it converges to an optimal policy that maximizes the cumulative reward over the long term.

One specific advantage of SARSA is its ability to learn offline without requiring rewards to be computed at every action and state. This means that the agent can learn from past experiences without interacting with the environment in real-time. This is particularly useful in scenarios where obtaining rewards is costly, time-consuming, or even impossible. Additionally, offline learning enables SARSA to scale to larger state and action spaces, as the agent can learn

from a pre-existing dataset rather than relying solely on online interactions with the environment.

2.4 Q-Learning

Q-Learning is a model-free reinforcement learning algorithm that aims to find the optimal action in a given state without requiring a model of the environment (hence the name "model-free"). This makes it particularly useful for solving problems with stochastic transitions and rewards, as it can learn directly from experience without requiring further adaptations.

Q-Learning works by finding an optimal policy by trying to maximize the expected value of the total reward over any and all successive steps, starting from the current state. It can identify an optimal action-selection policy for any given trajectory, assuming infinite exploration time and a partly-random policy. The "Q" in Q-Learning refers to the function that the algorithm computes, which represents the expected rewards for an action taken in a given state.

Like the SARSA algorithm, Q-Learning also uses Temporal Difference (TD) learning to estimate the optimal action-value function in a given environment. The main difference between SARSA and Q-learning is that Q-Learning is an off-policy learner. Thus, it learns the value of the optimal policy independently of the agent's actions, while an on-policy algorithm, like SARSA, learns the value of the policy being carried out by the agent, including the exploration steps. Therefore, in computing the Temporal Difference, Q-Learning compares the observed reward $Q(s, a)$ with the reward obtained by following the optimal action $Q(s', a^*)$:

$$TD_{error} = [r + Q(s', a^*)] - Q(s, a)$$

As a result, Q-Learning may converge to the optimal policy faster and can be more sample-efficient, but it may not take an exploration into account as effectively as SARSA.

3 Methodology

3.1 Dynamic Programming

Let s_i represent the expected number of moves needed to reach square 100 starting from square i. Thus, our goal is to find s_0, and we know that $s_{100} = 0$.

We notice that the values for s_i are circular in reference, as they relate to each other through the probabilistic chances of transitioning to different states. For example, suppose we are at state three and choose to use dice red; we have a 2/3 chance of getting to square 5 (a roll of 2) and a 1/3 chance of getting to

square 30 (a roll of 6, then going up the ladder). This results in the following equation:

$$s_3 = \frac{2}{3}s_5 + \frac{1}{3}s_{30} + 1$$

where adding one account for the additional roll that we are doing now. Next, we can move all the coefficients to the LHS and the non-coefficient terms to the RHS to obtain

$$s_3 - \frac{2}{3}s_5 - \frac{1}{3}s_{30} = 1.$$

Continue in the same manner for all $s_0, s_1, ..., s_{100}$, then we can formulate them as a matrix equation $Ax = b$, in which x is the column vector of the expected number of steps from each state that we need to find

$$x = \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_{100} \end{bmatrix},$$

A is the matrix of coefficients corresponding each s_i, and b is the column vector of non-coefficient. In the above case, when we choose dice red at state 3, then the 4^{th} row of A (since we count from s_0) is

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 & -\frac{2}{3} & \cdots & -\frac{1}{3} & \cdots & 0 \end{bmatrix},$$

and b is

$$b = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \\ 0 \end{bmatrix}$$

where $b_{100} = 0$ because $s_{100} = 0$.

Each policy π indicating which dice should we use at each state gives us a different coefficient matrix A. Thus, solving for x in the above equation $Ax = b$ gets us the result of applying that policy, which is the expected number of steps to complete the game starting from each state s_i and following policy π. We can then extract the value of s_0 from x to obtain the expected number of steps needed to win the game following policy π.

Our goal is to find the policy that results in the minimum expected number of steps to win the game, which requires us to use the Hill Climbing technique. We start with an initial policy, like always playing red dice, for example, and then iteratively make small changes in our policy to improve the performance based on a heuristic function. In this case, the heuristic function solves the equation $Ax = b$ and uses the calculated value of s_0 as the performance score

of our policy. Each iteration to update policy involves iterating through every state, and for each state, trying all four dices to choose the one that results in the best reward. We continue updating our policy until the reward values converge to a local minimum.

3.2 Monte Carlo

The process of going from computing Q to V and finally P consists of two main parts: policy evaluation and policy improvement. Evaluating policy means storing the total rewards obtained and counting the number of times specific states/actions are visited, which can be computationally complex. The real learning process happens at the policy improvement step, where the model updates V and the policy based on the rewards and counts it has stored. The entire algorithm involves running policy evaluation and improvement repeatedly until it converges to the most optimal policy.

The pseudocode for evaluating and improving the policy is as follows:

Algorithm 1 Policy evaluation

procedure POLICY_EVALUATION($total, count$, policy P)
    Use policy P to make n trajectories
    Each trajectory finish we update *total* and *count*
    Compute Q follow that $Q[s][a] = \frac{total[s][a]}{count[s][a]}$
    return Q
end procedure

Algorithm 2 Policy improvement

procedure POLICY_IMPROVEMENT(policy P)
    $V = \min(Q)$
    $P = \text{argmin}(Q)$
    return V, P
end procedure

In our experiment, we run the Monte Carlo algorithm 1000 times, each time returning the best policy that the model can achieve with a specific policy after going through 100 episodes. After a total of 1000 plays, it should return the best policy that minimizes the number of steps.

3.3 SARSA and Q-Learning

In SARSA, the agent at each step selects an action a according to the current policy and then updates the Q-value for the current state-action pair based on the observed reward and the next state-action pair. The hyper-parameters

include the learning rate α, which determines how much weight is given to new information when updating the Q-value, and the exploration rate ϵ used in the e-greedy algorithm to determine how often the agent chooses a random action versus following the policy. Below is the pseudocode for the SARSA algorithm:

Algorithm 3 SARSA Algorithm

procedure SARSA(Q, P, α, ϵ)
    initialize a state s and an action a using the current policy
    while not at terminal state **do**
        take action a and receive a reward r
        sample new state s'
        choose action a' following an e-greedy algorithm
        compute $TD_{error} = [r + Q(s', a') - Q(s, a)]$
        update $Q(s, a) \leftarrow Q(s, a) + \alpha TD_{error}$
        $s \leftarrow s', a \leftarrow a'$
    end while
end procedure

A key modification of SARSA leads to the algorithm known as Q-Learning. Specifically, we do not use the policy to find the value of a' but instead, choose the one that gives the highest value. As a result, the temporal difference is computed as

$$TD_{error} = [r + \max_{a'} Q(s', a') - Q(s, a)]$$

This TD error is used to update the Q-value for the current state-action pair, which improves the accuracy of the Q-values and hence the algorithm's performance over time.

4 Results and Analysis

When applying the four approaches (Dynamic Programming, Monte Carlo, SARSA, and Q-Learning) to solve the Chutes and Ladder problem, we observe slight variations in the optimal step counts required to win the game. Additionally, the algorithms exhibit significant variation in terms of their convergence rates toward the optimal solution.

4.1 Dynamic Programming

The Dynamic Programming approach converges to different optimal solutions with varying numbers of expected steps, depending on the initial policy (initialization point). In addition, different initialization of the policy leads to a different number of iterations until convergence. Particularly, if the initial policy is always to play red dice, then the DP approach converges after three iterations, with an expected number of steps an 11.62. Meanwhile, if our initial

policy is always to choose black dice, DP converges after two iterations with the number of expected steps being 12.5. The DP approach yields the best result when the policy is always to use the green dice. In this case, DP converges after two iterations, and the expected number of steps is 10.208.

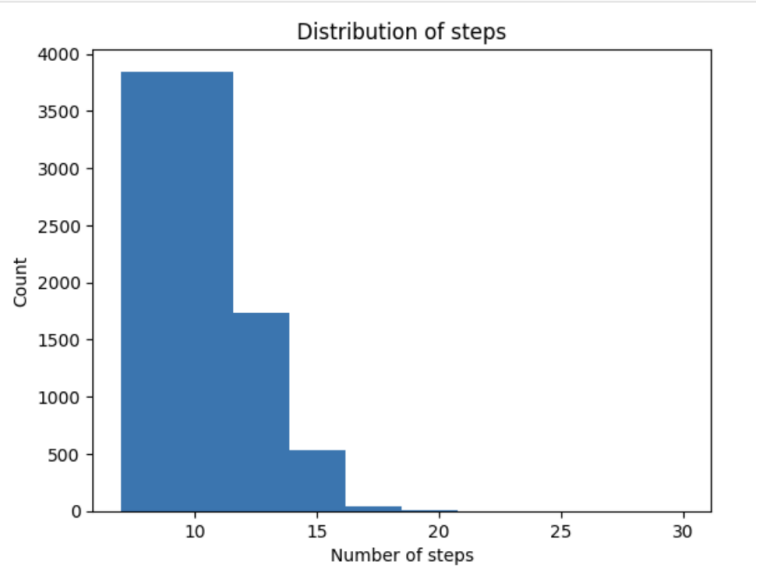

Figure 1: Dynamic Programming Performance

After running and assessing the best policy returned by DP 10,000 times, we generate a histogram for the expected number of steps as shown in Figure 1. The histogram indicates that the number of steps following this policy varies minimally, which is an advantage of approaching decision-making problems from a dynamic programming perspective. However, the algorithm's dependence on the choice of initial policy also means that it is computationally expensive, particularly when having to search large state and action spaces.

4.2 Monte Carlo

In contrast to the dynamic programming approach, Monte Carlo relies on random simulations, which can result in greater variation in the observed results and provide no guarantee of finding the most efficient set of optimal dice selections. In Figure 2, we show the recorded learning performance of the algorithm using 100 epochs and the accuracy of the policy that the model has learned so far. The graph shows that the Monte Carlo method requires a higher computation time and can have lower peak performance than other approaches. Given these limitations, the Monte Carlo model may not be the best-suited method

for efficiently and accurately solving the Chutes and Ladder game compared to other methods, such as dynamic programming.

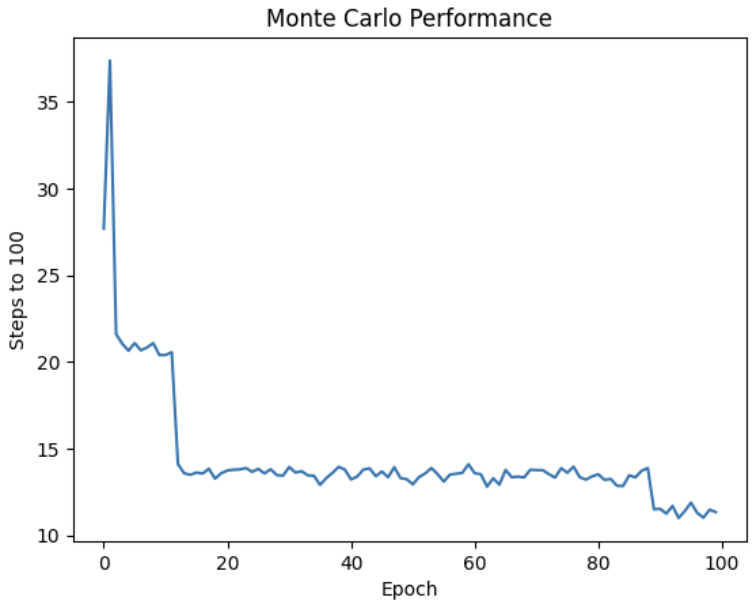

Figure 2: Monte Carlo Performance

4.3 SARSA

Figure 3 demonstrates the performance of SARSA with $\alpha = 0.1$ and varying ϵ values. The plot shows that the algorithm quickly reaches an optimal solution (around 12 steps to Square 100) after 12 epochs. However, its average numbers of steps fluctuates considerably as the number of epochs increases.

Additionally, we observe that the algorithm actually performs better with an epsilon of 0.4 compared to SARSA of $\epsilon = 0.1$. With $\epsilon = 0.4$, the algorithm reaches a better-expected number of steps while maintaining lower variation when trained for more epochs. A higher epsilon value means that the algorithm is more likely to explore new actions rather than just exploiting the best actions it obtains from previous experience. Overall, this suggests that the Chutes and Ladders game may necessitate SARSA to engage in more extensive exploration, trying out various actions to develop an optimal policy that achieves the lowest possible expected number of steps.

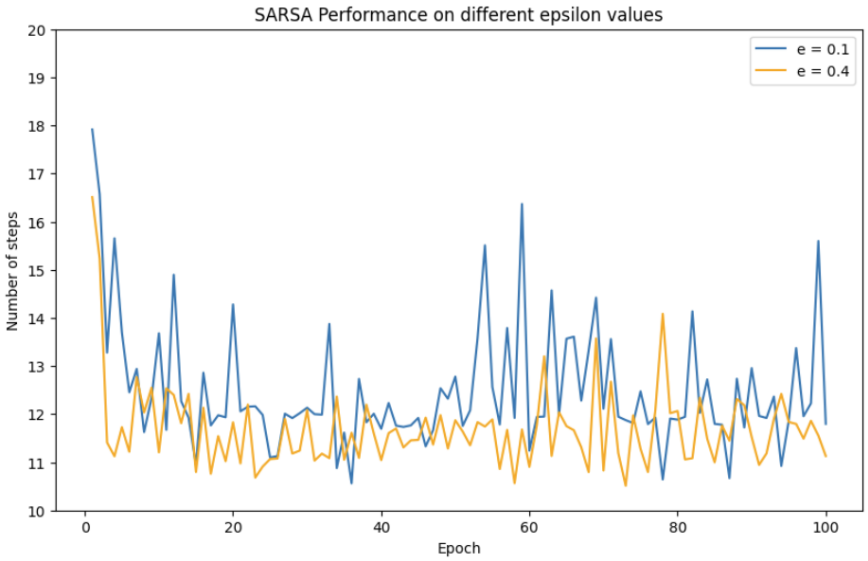

Figure 3: SARSA Performance with Different Epsilon Values

4.4 Q-Learning

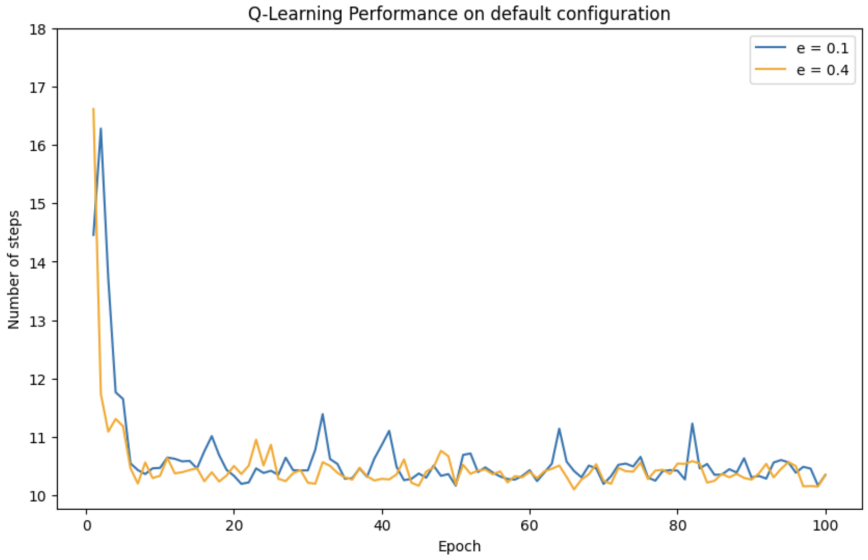

Figure 4: Q-Learning with Different Epsilon Values

We conduct an evaluation of the Q-Learning algorithm's performance on the Chutes and Ladders problem using the same configuration as SARSA learning. As shown in Figure 4, Q-Learning outperforms SARSA learning with the same configuration, achieving a lower number of steps and greater consistency with the optimal result. Notably, Q-Learning can discover a good policy (with an average of 12 steps) within the first 15 epochs, and the number of optimal steps even decreases to around 10.5 after the addition of 20 epochs. The algorithm converges to its optimal policies after about 40 epochs, resulting in an expected number of steps between 10 and 11 on average.

To further explore the performance of Q-Learning, we experiment with different epsilon values. We find that Q-Learning performs better with an epsilon of 0.4 compared to an epsilon of 0.1. Both settings reach the optimal result within the first ten epochs, but Q-Learning trained with an epsilon of 0.4 has a smaller variation value (0.81) than that of an epsilon of 0.1 (0.66). This leads us to conclude that the smallest expected number of steps recorded for Q-Learning is 10.1, with an epsilon of 0.4.

Method	**Peak Performance**	**Standard Deviation**
Dynamic Programming	10.2	–
Monte Carlo	11.4	2.04
SARSA	10.5	0.83
Q-Learning	10.1	0.66

Table 2: Peak performances and result variations for each method.

Table 2 above summarizes the peak performances and result variations for each of the four methods used to solve the Chutes and Ladders problem. Accordingly, Q-Learning achieves the best peak performance, with the lowest expected number of steps of 10.1, as well as the smallest result variation in performance. It follows that that Q-Learning is a superior method for solving decision-making problems like the Chutes and Ladders problem, producing consistently optimal results with a relatively small number of training epochs.

5 Future Work

This study suggests several avenues for future research on Dynamic Programming and Reinforcement Learning algorithms for solving the Chutes and Ladders game. Specifically, for the Dynamic Programming approach, future work could focus on exploring more efficient ways to initialize the policy that could result in a better and faster algorithm, as there are potentially 4^{100} ways to do so. Another potential future direction is to experiment with different settings of the learning rate α and the exploration rate ϵ. We can also consider adding

a new discounted factor γ to improve the Temporal Difference error function to handle infinite reward cases, thus improving the learning process. Finally, we plan to increase the complexity of the Chutes and Ladders game by introducing new elements, in order to evaluate the performance of each algorithm under more challenging scenarios.

6 Conclusion

Based on our experiments conducted on four different algorithms, including Dynamic Programming, Monte Carlo, SARSA, and Q-Learning, to solve the Chutes and Ladders game, we find that Q-Learning achieves the best performance with the smallest expected number of steps, the smallest result variation, and shortest time to reach an optimal policy. This suggests that Q-Learning is most suitable for solving complex decision-making problems, particularly those that involve stochastic processes and require an understanding of the model and environment. Our research also highlights the importance of the epsilon parameter in the performance of reinforcement learning systems and suggests the need for future studies to determine an optimal value for epsilon and other relevant parameters.

7 References

1. Wikimedia Foundation. (2023, April 21). Q-learning. Wikipedia. Retrieved May 1, 2023, from https://en.wikipedia.org/wiki/Q-learning

2. Shyalika, C. (2021, July 13). A beginners guide to Q-learning. Medium. Retrieved May 1, 2023, from https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

3. Geron Aurelien. (2023). Hands-on machine learning with scikit-learn, Keras, and tensorflow concepts, tools, and techniques to build Intelligent Systems. O'Reilly.

4. Marsland, S. (2014). "Machine learning: An algorithmic perspective, second edition." Chapman & Hall/CRC.

8 Appendix

Group Management

Person	Algorithm	Paper
Uyen Le	SARSA	Collectively
Duc Doan	Monte Carlo	Collectively
Phineas Pham	Q-Learning	Collectively
Tung Luu	Dynamic Programming	Collectively

Table 3: Group 4 Task Division

Boardgame

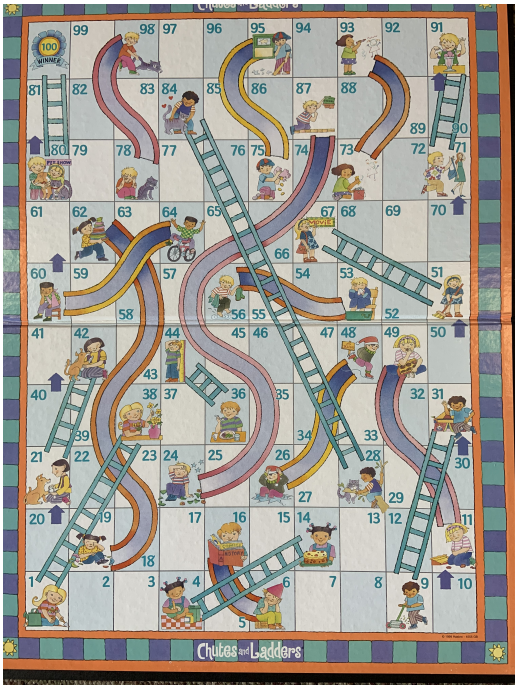

Figure 5: Chutes and Ladders Boardgame