Analysis of the Different Methods of Reinforcement Learning on Chutes and Ladders

William Duquette, Malcolm McDonough, Jorden Anfinson, and Mitch Aschmeyer

June 22, 2023

Introduction

In the following paper, we evaluate the performance of several reinforcement learning algorithms in an attempt to find the best policy for the children's game "Chutes and Ladders". The algorithms consist of Dynamic Programming, Monte Carlo, SARSA, and Q-Learning, each using a different method to find their solution to the game. Dynamic programming makes use of an offline model and uses equations to find the optimal answer. Monte Carlo is instead online and tries random moves over multiple iterations to find the correct answer. Finally, SARSA and Q-Learning use reinforcement learning algorithms with different update functions to find the answer. Our goal is to use each of these algorithms to produce solutions and see where they are similar and where they differ.

Chutes and Ladders is a simple game where your only goal is to reach the final space by rolling the dice. Along the way, you will find spaces that have chutes and ladders. Ladders will take you to a space higher up on the board, providing a great benefit, while chutes will move you down the board, hurting your progress. In order to add strategy to the game, we will be making use of four different dice. Each of these dice can be used on any space and have different values. The four dice can be seen below:

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

The use of these dice can be used to achieve several different strategies, such as hitting every ladder, avoiding all chutes, or making the most progress in one move. In the end, the optimal game can be completed in 8 moves. One such example can be seen here, (Route: $0 \rightarrow 1$ (ladder) $38 \rightarrow 44 \rightarrow 47 \rightarrow 51$ (ladder) $67 \rightarrow 71$ (ladder) $91 \rightarrow 94 \rightarrow 97 \rightarrow 100$). However, there are multiple ways to reach this optimal move set, and it may depend on the method used.

In the end, we find that Q-Learning and Dynamic Programming yield policies that best minimize the number of moves needed to win a game of Chutes and ladders.

Dynamic Programming

Dynamic Programming is an offline method that avoids the use of simulations and instead uses mathematical formulas to calculate the expected reward and best action at each state. This method provides several benefits as we are able to achieve an answer without having to create a simulation of the problem, but also it provides us with an optimal move from each possible state.

Methods

Dynamic programming involves the creation of an equation for each state which helps us determine the expected value at each state. These values are used to determine our optimal strategy. For "Chutes and Ladders," this expected value is the average number of dice rolls we expect to reach the end of the game from our current location. We calculate this value by multiplying the expected value of each potential state by the probability of reaching it. Below are example equations for our first state using the black (1) and red (2) dice.

$$S_0 = 1 + S_0 * \frac{1}{3} + S_4 * \frac{2}{3} \tag{1}$$

$$S_0 = 1 + S_6 * \frac{1}{3} + S_2 * \frac{2}{3} \tag{2}$$

Our goal is to use equations like these for each of the four dice to minimize the expected number of dice rolls to win the game and record which dice gives us the lowest value at each step. To transfer this to code, we start by first creating a matrix of size 100 by 100. Each row in this matrix represents one state. At each index of the row, the number represents the likely hood of ending up at that state based on the dice used. In the code, each dice is represented by a function, an example of which can be seen below.

Algorithm 1 `blackDice(currentState)`, *currentState* = current state in the matrix

```
0: coef = [0.0] * 101
0: coef[currentState] = 1
1: if currentState + 4 > 100 : then
2:     ns = 100
3: else
4:     ns = currentState + 4
5: end if
6: coef[ns] += -2/3
7: coef[currentState] += -1/3
8: return coef =0
```

This function works by first creating an array of size 101 where each index represents a possible state. The probability of ending up at each possible state is calculated, including the value of 1 to represent the current state, which is then added to the array at the proper index. This array is then appended to the matrix to represent the possible outcomes from the current state. Before we can solve this using a linear matrix equation, we must first replace all of the chutes and ladder spaces by going to each of the starting states and setting their only possible end space to the index where the chute or ladder ends.

Once we have created our matrix, we begin the process of finding the best dice to use at each state. To do this, we iterate through each index of the array and try replacing it with each dice. Whichever dice results in the lowest expected moves from that state will be chosen as the optimal move. We run this process across the entire array multiple times as changing the dice used in one state can affect others, meaning we need to account for the changes through multiple iterations.

We solve this problem using a linear matrix equation from the NumPy module, which will use the provided matrix along with a second matrix of size 101. This second matrix represents the coefficients of our equations, or more specifically, the number of moves added at each space. This is one for each space except the starts of chutes or ladders and the final state.

Results

After setting up our matrix and array, we get the following policy where the color of each square represents which dice to use.

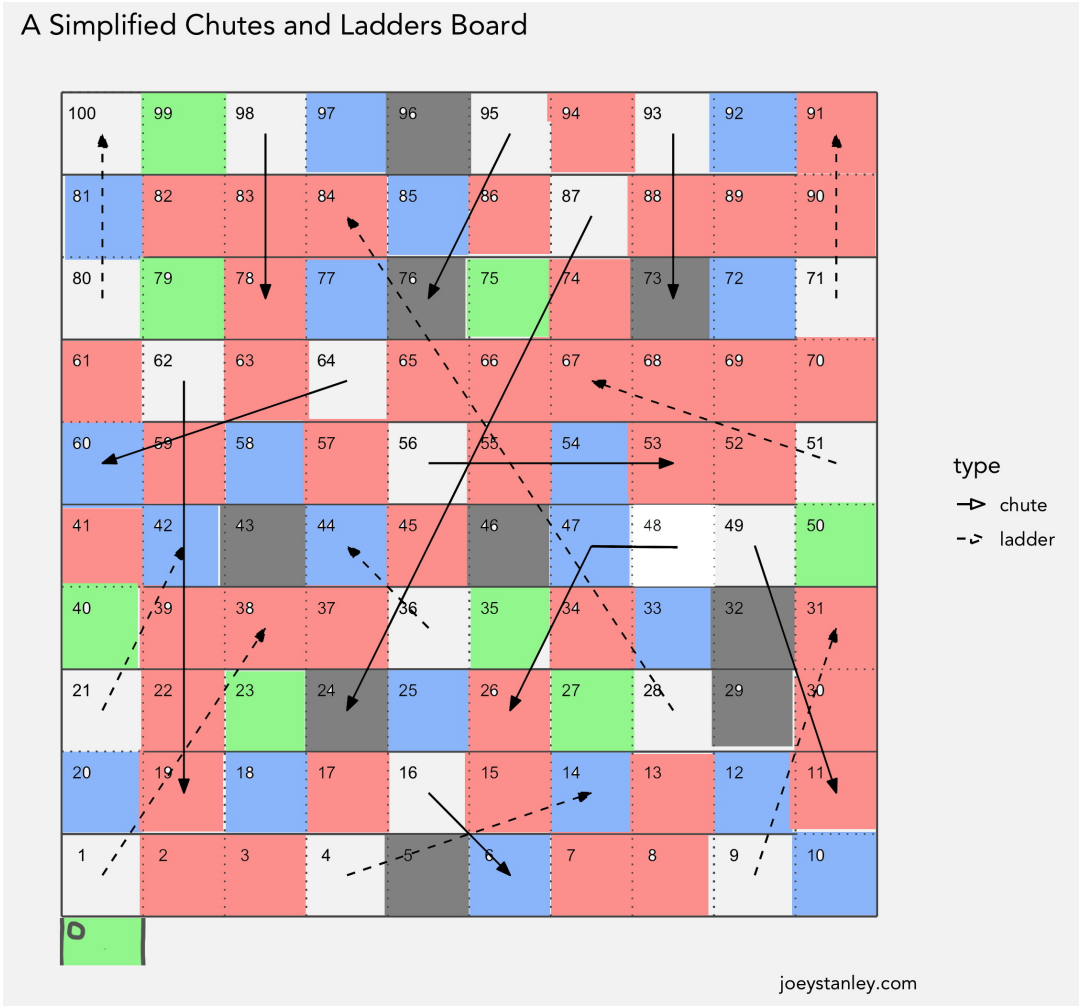

Figure 1: DP policy mapped onto a Chutes and Ladders board, colored based on die color chosen at that square.

The above board results in an expected value of 10.58, meaning we expect the game to take 10.58 turns on average using this policy. This is very close to the optimal value of 8 moves. This slightly higher value is due to the chance created in our game by the dice not having one possible value. However, if we get optimal rolls from each of our dice, the game will be complete in 8 moves.

There are a few interesting points and techniques we can see used in this policy. The policy uses a large amount of red in the middle of the board. This is likely due to it having the potential highest move amount at 6. In addition, in almost all cases, the policy prioritizes avoiding chutes, except for state 92. At this state, the policy chooses to hit the chute at state 95. This is because the chute can be taken to state 76, which has a higher chance of finishing in one move due to the ladder at state 80,

rather than all other reachable states, which are likely to need two rolls to finish.

Overall, this model performs very well and found many unique strategies which are not readily apparent. Additionally, this model is able to avoid obvious negative states like the chutes.

Monte Carlo

Monte Carlo learning is a relatively basic approach to Reinforcement Learning. Essentially, Monte Carlo learning is done by taking many different samples, or in this case trajectories, and averaging the rewards seen for each state. To be sure, hundreds, if not thousands, of samples, must be taken in order to get a realistic estimate. In this case, thousands of trajectories are taken through the Chutes and Ladders board.

Methods

To train our Monte Carlo, we use an assortment of algorithms. There are two "sections" of the learning process that take place:

1. Policy Evaluation: Includes playing the game and updating our Q matrix. Importantly, we freeze the policy, meaning we play the game and do not update our policy to allow the learning process to continue.

Algorithm 2 `policyEvaluation(totals, counts, policy, n, epsilon)`, totals = sum of total rewards from that state, counts = number of times at each state, policy = move policy that we have developed so far, n = number of trajectories we would like to do, epsilon = value used in $\epsilon - greedy$ policy

for $i \dots n$ **do**
  $t = trajectory(policy, epsilon)$ {new trajectory is made using $\epsilon - greedy$. Also note that t is a three by length of trajectory list that contains state/action/reward}
  $m = len(t)$
  $sumRewards = np.zeros(m)$
  $sumRewards[m-1] = t[-1][2]$
  for $j \dots m-2, -1, -1$ **do**
    $sumRewards[j] = sumRewards[j+1] + t[j][2]$
  end for
  for $j \dots m$ **do**
    $s, a, r = t[j]$
    $counts[s, a] += 1$
    $totals[s, a] += sumRewards[j]$
  end for
end for=0

2. Policy Improvement: Includes updating our policy using our Q matrix, which is updated in the policy evaluation step.

Algorithm 3 `policyImprovement(Q)`, Q = Q-matrix of the model

$V = np.min(Q, axis = 1)$
$P = np.argmin(Q, axis = 1)$
return V,P =0

The Policy Iteration algorithm ties this all together by running many iterations of policy evaluation and then updating the Q matrix and policy. The process described above is done n times. Note in this case, we chose 15000 policy iterations, which means that we run through the policy evaluation and update the policy 15000 times.

Algorithm 4 `policyIteration(totals, counts, policy, Q, n, m, epsilon)`, totals = sum of total rewards from that state, counts = number of times at each state, policy = move policy that we have developed so far, Q = Q-matrix of the model, n = number of trajectories we would like to do, m = the number of learning episodes, epsilon = value used in $\epsilon - greedy$ policy

{Note: This algorithm performs n iterations through the process described above. We use m trails per policy update}
for $i \dots n$ **do**
    $Q = computeQ(totals, counts)$ {Using the total at each state and the number of times each state has been visited we are able to compute the Q matrix quickly}
    $V, P = policyImprovement(Q)$ {See above}
    $policyEvaluation(totals, counts, P, m, epsilon$ {See above}
end for
$Q = computeQ(totals, counts)$ {This is done again once we have updated totals and count from our iterations}
$V, P = policyImprovement(Q)$ {With newly computed Q values we have to update our policy and V} =0

Results

After conducting Monte Carlo learning, we have a policy that should theoretically give us optimal instructions for which dice to roll at each state, also known as a policy. This policy is illustrated in the plot below:

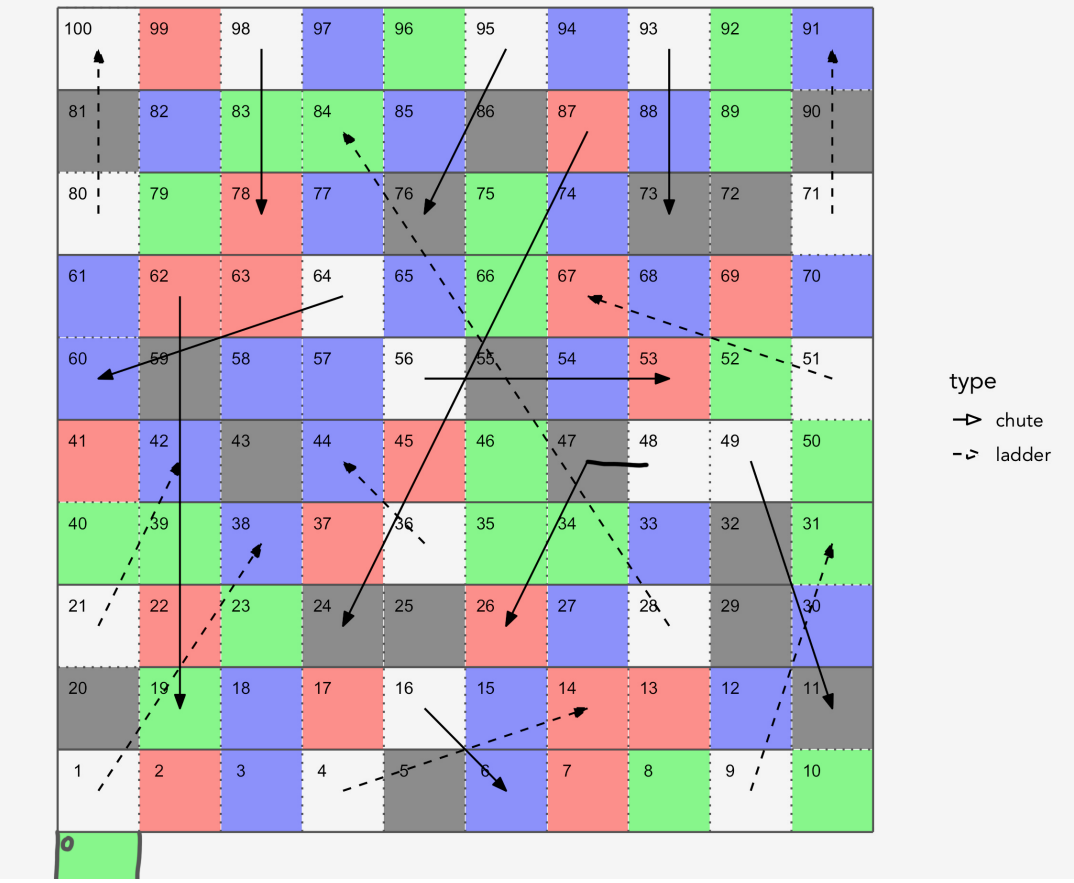

Figure 2: Monte Carlo policy mapped onto a Chutes and Ladders board, colored based on die color chosen at that square.

The above policy takes an average of 12.7 moves to complete a game of Chutes and Ladders, which is relatively close to the optimal value of 8 moves. The Monte Carlo approach discovered some interesting moves. For example, on the first move, our policy tells us to roll the green die. This makes sense because if we land on the 1st state, we hit a ladder and are taken all the way to 38. There are some peculiar decisions as well. For example, on square 92, it tells us that we should roll the green die. This is interesting because the green die has $\frac{1}{2}$ chance of rolling a one, which would hit a chute and take us all the way to 72. But there is a $\frac{1}{2}$ chance of rolling a five, which would lead us to finish the game in 2 moves because after a five is rolled, we would land on state 97. From there, our policy would tell us to roll the blue die, which would take us to the end. That is an interesting choice to make, given the risk.

Although the number of moves is higher than some of the other models, the average number of moves needed to reach the end is by no means absurdly high. Monte Carlo remains an important topic in Reinforcement Learning because it is a useful approach

to take on its own and is crucial in the implementation of the Q-Learning and SARSA algorithms discussed below.

SARSA

SARSA, or "state action reward state action", and Q-Learning are both known as temporal difference learning algorithms, where the value function is updated based on the difference in the predicted and actual rewards received at each step. SARSA in particular is an on-policy reinforcement learning algorithm, meaning that it tries to optimize its current policy as it iterates. Additionally, SARSA works by using an ϵ-greedy method to decide the next action: a', meaning that the algorithm will almost always choose the action that has the highest future expected reward, but sometimes choose a random action. In doing so, the SARSA algorithm balances exploration and exploitation.

Methods

Algorithm 5 `SARSA(`Q`,` α`)`, Q = Q-matrix of the model, α = learning rate

Initialize $s = 0$, a using ϵ-greedy
Get action a by using ϵ-greedy
while $s < 100$ **do** {Square 100 is our terminal state}
    perform action a at state s, observe s' (resulting state) and r (reward) {Note that the reward is always 1 as the model has made one move.}
    Get the next action by using ϵ-greedy to obtain the action a' at s'
    $TDerror = [r + Q(s', a')] - Q(s, a)$
    $Q(s, a) = Q(s, a) + \alpha * TDerror$
    $s = s'$
    $a = a'$
end while=0

We train the model over 10 sets of 1000 trials.

Results

After training the model, we arrive at the policy illustrated on the board below. Each color indicates the rolling of a specific die.

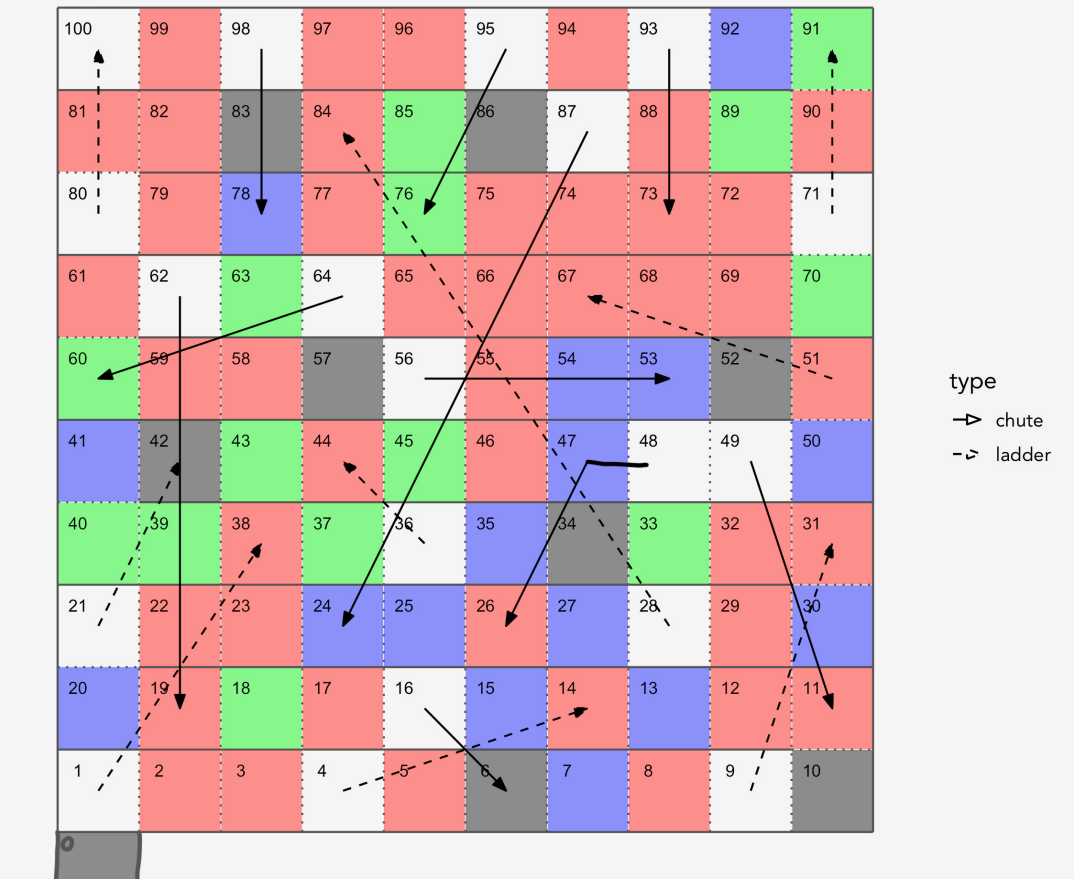

Figure 3: SARSA policy mapped onto a Chutes and Ladders board, colored based on die color chosen at that square.

The resulting policy takes an average of 11.95 rolls to reach the end state as derived from the sum of future expected reward at state 0, or V[0]. Thus, if one were to use the policy derived from the SARSA algorithm, it would on average take around 12 rolls to complete a game of Chutes and Ladders. Interestingly, the policy results in a red die being rolled for the vast majority of spaces on the board.

Q-Learning

Q-Learning is very similar to SARSA: the main differences between the two lie in their update functions and the fact that SARSA is on policy while Q-Learning is off policy. To calculate the Temporal Difference Error, Q-Learning always uses the optimal action from the resulting state for a', while SARSA uses ϵ-greedy to find a'.

Methods

To train our Q-Learning model, we use the following algorithm:

Algorithm 6 `Q-Learning(`Q`,` α`)`, Q = Q-matrix of the model, α = learning rate

Initialize $s = 0$, a using ϵ-greedy
while $s! = 100$ **do** {Square 100 is our terminal state}
    assign a using ϵ-greedy
    perform action a at state s, observe s' (resulting state) and r (reward) {Note that the reward is always 1 as the model has made one move.}
    $a' =$ optimal action at s'
    $TDerror = [r + Q(s', a')] - Q(s, a)$
    $Q(s, a) = Q(s, a) + \alpha * TDerror$
    $s = s'$
end while=0

We train the model over ten sets of 1000 trials, updating our policy at the end of each. After we have finished training the model, we will then have it play 2000 non-training games in order to evaluate the quality of the model, taking the average number of games to complete each game and averaging them to get a measure of the model's quality.

Results

After training the model, we are given a policy that gives the action to take at each square of the Chutes and Ladders board. It is illustrated in the figure below.

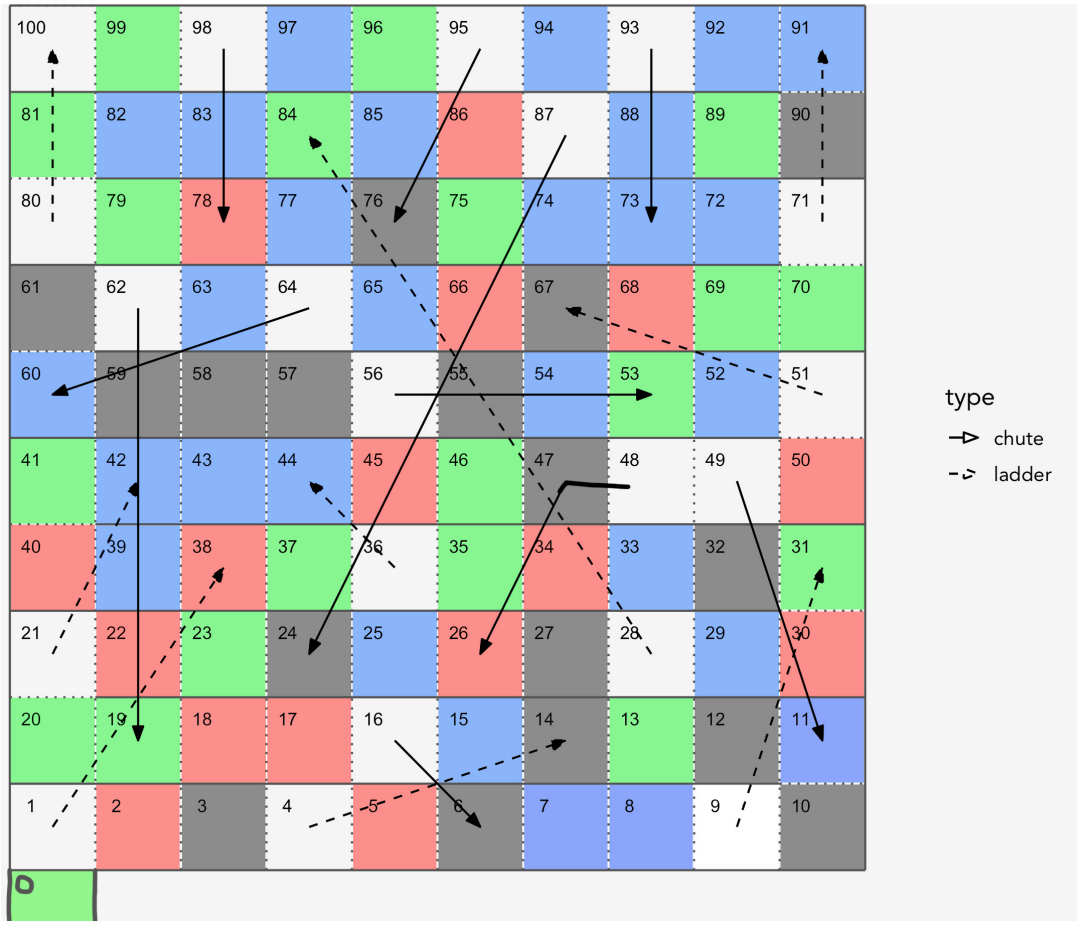

Figure 4: Q-Learning policy mapped onto a Chutes and Ladders board, colored based on die color chosen at that square.

This policy takes an average of 10.45 moves to complete a game of Chutes and Ladders. This path is very similar to the best possible game of Chutes and Ladders, with the only difference being that the best possible game will choose to use the risky red die on square 94 instead of the blue die, which has a chance of getting straight to state 100 rather than taking two turns with the blue die.

There are some strange choices on certain squares, such as choosing to use the blue die on square 92, which guarantees taking a chute on square 95, but these choices may be balanced out by moves that come after. In this case, the chute on square 95 takes you to square 75, which is five squares from a ladder on square 80 that takes you straight to square 100. Additionally, we can observe that no path through our policy will naturally result in the model landing on square 92, which may explain why the has a seemingly sub-optimal action choice at that state.

Generally, though, this model will perform quite well, with games on average taking just under 11 moves to win and always finding a path that is within one move

of the best possible route through the board.

Discussion

Looking at the table below, we can see that Monte Carlo Learning is the worst. This is unsurprising as Monte Carlo learning is very susceptible to making unrealistic changes to the Q matrix because Monte Carlo learning is worse at handling noisy samples because it gives equal weight to every trajectory. The other three approaches all yield similar averages, with the lowest being $Q-$Learning, with an average of 10.45 moves to the end. Although $Q-$Learning has the smallest number of moves, it is not a significant difference from the results seen by Dynamic Programming and SARSA. This might suggest that Dynamic Programming, $Q-$Learning, and SARSA yield the best results while Monte Carlo yields less than optimal results.

Algorithm	Average Number of Rolls
Dynamic Programming	10.58
Monte Carlo	12.7
SARSA	11.95
Q-Learning	10.45

Figure 5: Performance of the Different Algorithms

Interestingly, all of the algorithms followed the same overall trend. This can be seen in the plot below. We can see that the V values might be different, but the overall trend is almost the same for each of the algorithms. (Note: $Q-$ Learning, SARSA, and Monte Carlo all have V values of 0 at certain points, which is because these algorithms have randomness and often do not visit every state.) This suggests that although the values may be different, the algorithms are all learning in similar ways.

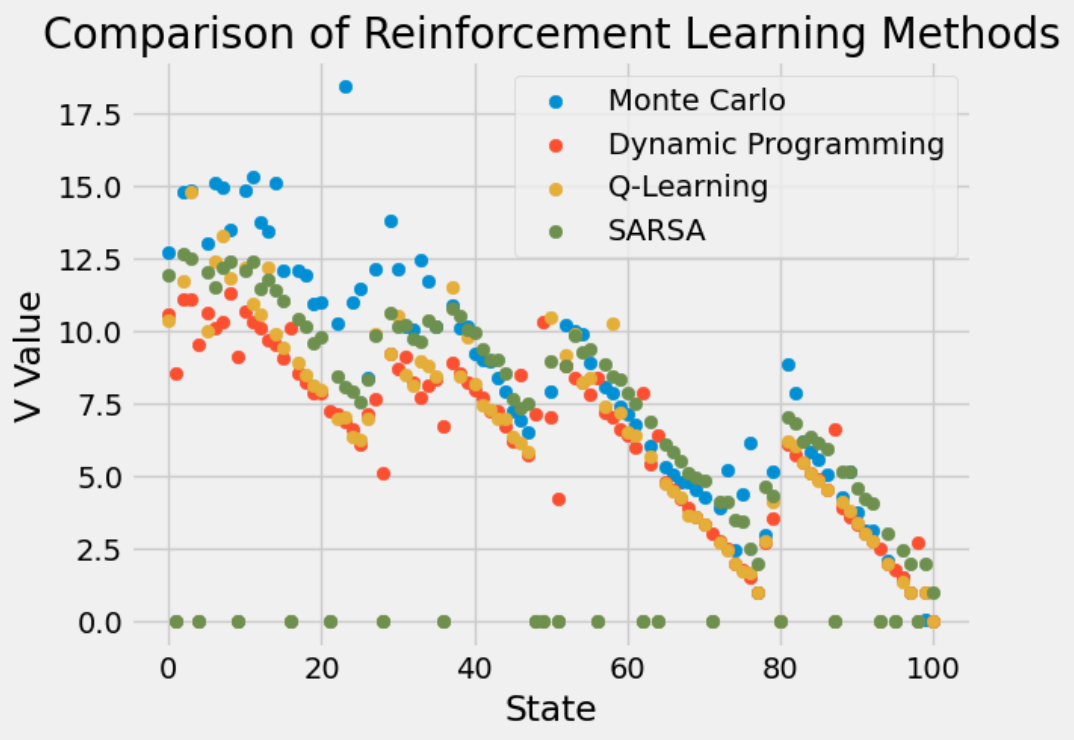

Figure 6: Comparison plot: shows the different approaches to RL

Conclusion

We analyzed Dynamic Programming, Monte Carlo Learning, SARSA, and Q-Learning to find the best strategy for the modified version of Chutes and Ladders. We found Q-Learning produced the best policy with an average of 10.45 rolls required. The next best policy was derived through Dynamic Programming, which discovered a policy that would, on average, take 10.58 rolls to win a game of Chutes and Ladders if followed. Thus, if you were to ever play a game of Chutes and Ladders again, generating a policy through the use of Q-Learning or Dynamic Programming would lead to an optimal solution.