Beating Chutes and Ladders

Amna Khalid (khalid_a1@denison.edu),
Khoi Le (le_k2@denison.edu),
Mark Raney (raney_m1@denison.edu),
and Philipp M. Srivastava(mouras_p1@denison.edu)

Spring 2021

1 Introduction

Since the rise of widespread computing a set of algorithms known as Machine Learning has begun to solve more and more problems in our daily lives. This has allowed for things like image recognition, speech recognition, traffic prediction, self driving cars, and much more. Though subsets of Machine Learning like Deep Learning have received much attention in recent years; one older subset of Machine Learning algorithms known as Reinforcement Learning has been receiving increased attention. While the majority of Machine Learning algorithms learn by feeding vasts amount of data to the algorithm, Reinforcement Learning (RL) techniques act like a teacher to an agent in an environment by updating the decision making mechanisms of the agent based on feedback given after every action taken by the agent. In this paper we will be applying four different Reinforcement Learning techniques to solve the game known as Chutes and Ladders. At the end of the paper we will show that one of these techniques outperforms all of the other not only in beating the game faster, but also learning to beat the game faster.

2 Background

Before discussing the experiment and its results it's important that we gain some background knowledge in regards to the game Chutes and Ladders. The game involves a board, which is shown in figure 1, where the aim is to reach the hundredth square on the board by rolling one of four dice and moving forward by the number produced by each roll. The four dice are as follows, [4, 4, 4, 4, 0, 0], [6, 6, 2, 2, 2, 2], [5, 5, 5, 1, 1, 1], [3, 3, 3, 3, 3, 3] where each entry is a face of the die. We will refer to each die in the respective order as the black, red, green, and blue dice.

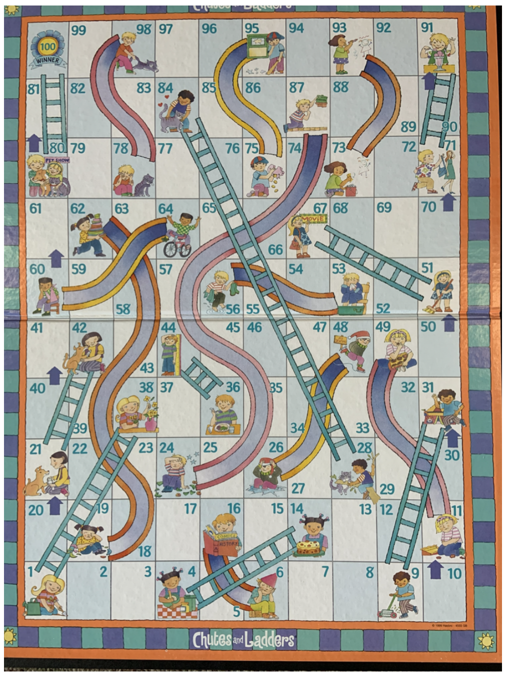


Figure 1: Chutes and Ladders Board

However, specific squares on the board will have either ladders of chutes connecting that square to another square. If we land on such a square we take the chute or ladder to the square that it is connected to. Chutes are connected to squares below the square landed on while the ladders are connected to squares above the square landed on. Reinforcement Learning techniques often utilize a policy which determines the action taken in each state of the game, or square on the board. Since landing on ladders is most desired the optimal policy will choose the die that increases the chance of landing on the squares with ladders. Now that we understand how the game and basic Reinforcement Learning work let's take a look at the experiment.

3 Methodology

3.1 Mathematical Formulation of Dynamic Programming

We begin with the algorithmic technique known as Dynamic Programming. Given a discrete time dynamic system f, we have

$$x_{k+1} = f_k(x_k, u_k, \omega_k),$$

where x_k is the ***state*** of the system at time k, u_k is the ***action/central signal*** at time k and ω_k is the ***stochastic element*** (random variable). The new state $k+1$ of the system is calculated by the function f_k defined above.

Next, we define μ_k to be the function that map a state x_k to the corresponding action u_k (denoted as $\mu_k(x_k)$). Then, we define the ***policy***

$$\pi = \{\mu_0, \mu_1, \ldots, \mu_{n-1}\}$$

as the set of functions that determine the action to take for each state.

Then, the ***cost***, or ***reward***, is the signal received from the environment to indicate the goodness of our action. We refer it as cost if we want to minimize, and we refer it as reward if we seek maximization. For a state k, the incremental cost is

$$g_k(x_k, u_k, \omega_k)$$

However, not every state k has the same cost function as for the last state, state n, the cost is defined to be terminal cost and it is denoted as $g_n(x_n)$.

In addition, we define the cost-to-go from a state k is $J_\pi(x_k)$ and it is equal to the sum of future cost starting in state x_k. In other words,

$$J_\pi(x_k) = \sum_{i=k}^{n} g_k$$

Based on the above definition, our task is to find the optimal policy that gives us the optimal cost $J^*_\pi(x_k) = min_\pi J_\pi(x_0)$.

To solve the DP, Bellman Principle of Optimality asserts that given the optimal policy $\pi^* = \{\mu^*_0, \mu^*_1, \ldots, \mu^*_n\}$, then the truncated policy of $\{\mu^*_k, \mu^*_{k+1}, \ldots, \mu^*_n\}$ is the optimal policy for the subproblem of x_k at k.

Given the above principle, we define the DP Algorithm to be:

1. Compute the final cost as $J_n(x_n) = g_n(x_n)$.

2. For $k = n - 1$ down to 0 as we descend from the final step to the beginning, we calculate

$$J_k(x_k) = min_{u_k} E_{w_k}[g_k(x_k, u_k, \omega_k) + J_{k+1}(f_k(x_k, u_k, \omega_k))]$$

and apply the optimal policy in the equation for the next calculation.

3.2 Dynamic Programming for Chutes and Ladders

Given the above definition of the mathematical formulation of dynamic programming, for our specific game, we define the state s_i to be the value of being in Square i. In other words, s_i is the expected number of rolls needed to get to Square 100 when we start at Square i. Therefore, s_0 is the value we need to estimate.

Given a dice that we use in Square i, the state s_i is measured by the sum of the probability of landing on a number on the dice, or the stochastic element, times the corresponding state that it ends up on plus 1. For example, if we are at Square 7, and we use the Green dice, then we will have the equation

$$s_7 = \frac{1}{2}s_8 + \frac{1}{2}s_{12} + 1$$

because we have a probability of $\frac{1}{2}$ to get one, which gets us to Square 8, and we also have a probability of $\frac{1}{2}$ to get 5, which gets us to Square 12. The constant 1 indicates that we use one roll in that Square, or the cost. However, it is not the same formula for every state:

- The state $s_{100} = 0$ by default because we are at the last Square, so the expected number of roll must be 0.

- If we land on a Square that is at the bottom of a ladder or at the top a chute, then we represent that state by the other side of that ladder or that chute. For example, if we start at Square 0 and we use the green dice, then instead of getting $s_0 = \frac{1}{2}s_0 + \frac{1}{2}s_5$, we get $s_0 = \frac{1}{2}s_{38} + \frac{1}{2}s_5$ because there is a ladder connect Square 1 with Square 38.

- However, we do not apply the above change to a state if we are calculating the expected number of roll from that state. For example, if we are in Square 1, and we use the Green dice, we still get $s_1 = \frac{1}{2}s_2 + \frac{1}{2}s_6$.

- If we are close to 100, but the number we get from a roll makes us go exceed 100, we still end up at 100. For example, if we are in Square 99 and we use the blue dice, we have $s_{99} = s_{100} + 1$.

Based on this approach, given a set of dices for each state (policy of the DP), we can isolate all the states from the constant. Then, we put the states in a 101×101 matrix A. The Square i that we are settled in will be represented in the i^{th} row of matrix A, and all the constant corresponding to each state j included in the function will be represented in the j^{th} column at the i^{th} row of the matrix. To illustrate, if we are in Square 2 and we use the blue dice, we get $s_2 = s_5 + 1$, or $s_2 - s_5 = 1$. In the matrix A, the second row is represented as $[0\ 0\ 1\ 0\ 0\ -1 \ldots]$. For every incremental cost, we put them in a 101×1 matrix b. Matrix b will always have the first 100 elements to be 1, and the last element is 0 for the state s_{100}.

After we have generated matrix A and b, if we choose the set of element to be a 101×1 matrix x containing $s_0, s_1, s_2, \ldots, s_{100}$, we can get the result of s_0 by solve the equation $Ax = b$ by using Numpy.

The way we find the optimal solution is to implement the Hill Climbing method. We initiate our set of policies for each state at the beginning. In our DP, we choose the set of policies to contain 101 green dices. To solve for the optimal solution, we repeatedly change the policy corresponding to each Square as for a specific Square, we choose among 4 dices while keeping other policy the same and solve the equation to see which dice provides the smallest value for s_0. The update is executed from Square 99 to Square 0, and it repeats updating all over again until no policy is changed compared to the previous set of policies.

3.3 Monte Carlo Simulation

The purpose of this section is to apply Monte Carlo simulation methods towards finding the optimal dice choice for each tile as well as the expected score when making the optimal choices.

Similar to both methods of Reinforcement Learning, Monte Carlo simulations follow the same basic structure of examining the relationships between states, actions, and rewards.

In a Monte Carlo simulation, a sequence of state, action $\rightarrow$ reward $\rightarrow$ ... makes up a trajectory, or unit of experience. The ultimate goal of this method is to use random number generation to produce several trajectories from which we generate the expected sum of future rewards when starting in state s and following policy π, denoted as $V^{\pi}(s)$. From there we generate the expected sum of future rewards when using each possible action from state s when using policy π, denoted as $Q^{\pi}(s, a)$, and take either the minimum or maximum depending on the goal of our experiment to identify the best possible action when in each state.

To build the framework for this specific problem, we first identify each possible starting location of a turn (i.e. not on any chute or ladder starting points or the winning tile) as a state.

Naturally, with each state we are presented with four options, one for every dice.

$$U_k \in \{\text{Dice 1, Dice 2, Dice 3, Dice 4}\}$$

The choice of which dice to roll become the actions we are to compare in $Q^{\pi}(s, a)$.

As for the reward value, we identify the sum of future "rewards" as the number of turns that followed the given state, action pair before reaching tile one hundred. In this case, because we wish to find the dice selection that minimizes the expected turn count, we take the minimum of the four actions in $Q^{\pi}(s, a)$ and select that dice as the optimal choice.

Ultimately, within this experiment we wish to examine the effects that the various parameters to the Monte Carlo simulation have on the optimal dice selection and the resulting expected turn count when following that policy. These variables include iteration count: the number of times we perform both policy evaluation and policy improvement, evaluation count: the number of times we generate and process a new trajectory, gamma: the factor by which we modify the expected sum of rewards for the given state-action pair following the results of a given trajectory generation, epsilon: The probability of choosing the currently identified best dice for the given state instead of selecting a dice at random, and finally the default dice: the dice we initialize every state's best dice selection as.

3.4 SARSA

State-Action-Reward-State-Action (SARSA) is an on-policy reinforcement learning algorithm that predicts the value of the policy being pursued. An event in SARSA is described in the form of:

$$s, a, r, s_0, a_0$$

This event can be perceived as follows: the agent was initially in state s, performed action a, received reward r, and concluded at state s_0, from which it further chooses to do some action a_0. This provides a new event to update $Q(s, a)$, and the new value returned after undergoing this event can be represented as:

$$r + \gamma Q(s_0, a_0)$$

Thus, the SARSA algorithm could be described as below:

$$Q(s_t, a_t) \leftarrow Q(s, a) + \alpha(r_0 + \gamma Q(s_0, a_0) - Q(s, a))$$

Consequently, the SARSA agent will interact with the environment and update the policy based on the actions exercised which is why SARSA qualifies as an on-policy machine learning algorithm.

Q-values are the possible reward received in the subsequent step for taking an action a in state s, plus the reduced expected reward received from the following state-action event. A Q-value for the action of the state is updated when the action chosen is not maximizing the reward, and thus adjusted via the learning rate α.

```
procedure PSEUDOCODE FOR SARSA IMPLEMENTATION
    Begin Chutes and Ladders
    while  do
        Initialize Q(s, a)
        Choose $a$ from $s$
    while  do
        Perform action $a$
        Choose $a$ from $s$
        $Q(s_t, a_t) \leftarrow Q(s, a) + \alpha(r_0 + \gamma Q(s_0, a_0) - Q(s, a))$
        $s \leftarrow s$
        $a \leftarrow a$
    Update game parameter
    End
```

For the purposes of this research, the game board of Chutes and Ladders is implemented as a SARSA algorithm where states are defined as:

- the initial state is the starting position of the player
- the action is defined as what moves the player makes on the board (i.e. up, down, left, right, etc.)
- the final state is defined as the next state after an action has been taken

When defining the policy for the game, the probability of a player winning based on the color of the dice they chose is also accounted for. Further, the policy is iterated over until the optimal path is determined. Combinations of rewards for various actions are performed to get a combination of reward values as one optimal combination among all the test cases.

3.5 Q-Learning

Next we will look at a variation of SARSA algorithm known as the Q-Learning algorithm. The Q-Learning algorithm is known as an off-policy learning algorithm while the SARSA algorithm is known as an on-policy learning algorithm. This is because in Q-learning the action taken is and ϵ greedy action instead of the action specified by the policy. We begin the algorithm by defining a Q table q and a policy p. The Q table is a two dimensional table with the states or positions on the board as the rows and the actions or which dice to roll as the columns. To begin, we initialize each entry of the table with a zero. Similarly the states are the rows of the policy table while the one column in the table is the action taken in that state according to the policy. Next we pick take an ϵ greedy action, which takes a random off policy action with probability ϵ, and observe the reward and the subsequent state. We then update q with the following equation:

$$Q(s, a) = Q(s, a) + \alpha \cdot [(Q(s', a') + r) - Q(s, a)]$$

where s', a', and r are the next state, next action specified by policy p, and the reward. Finally we update the policy by replacing the policy action with the action with the smallest Q value of state s.
There are two parameters that are integral to the operation of this algorithm. They are α and ϵ. The former is known as the learning rate and can be seen in the equation that updates Q values. The higher the value the faster it will learn since the changes to the Q values, which the policy updates are based off of, will be greater. The latter parameter, ϵ, is what is known as the exploration rate. The default policy is obviously not one that beats the game so we need the algorithm to take random actions and effectively "explore". With the ϵ greedy action which is used to guarantee that the algorithm explores ϵ amount of the time.
Now that we understand how the algorithm works we can take a look at some pseudo code and discuss how the algorithm will be used in our experiment. The pseudo code for the algorithm can be seen below.

```
procedure Pseudocode for Q Learning implementation
    Initialize Board
    Initialize policy $P$
    Initialize table $Q$
    while Goal not reached do
        Perform $\epsilon$ greedy action $a$
        $r \leftarrow 1$
        $Q(s,a) \leftarrow Q(s,a) + \alpha \cdot [(Q(s',a') + r) - Q(s,a)]$
        $P(s) \leftarrow min(Q(s,a)$
        $s \leftarrow s'$
```

As for the experiment we will see how many iterations of this algorithm are required to yield an optimal policy. We will then compare the effectiveness of this algorithm to the three other algorithms to determine which one is best.

4 Results and Analysis

4.1 Dynamic Programming

The results show that the DP method provides the optimal solution of 10.2, which is the best among the other three approaches. This is due to the fact that Dynamic Programming method solves this problem offline, which means that it has the information of every Square on the board as well as the dice. Therefore, it gives us the best optimal solution among the four methods. The running time is very fast as the result is returned after 2 iterations of updating all dices and the total number of dices gets updated is 81, taking 3 to 4 seconds.

4.2 Monte Carlo Simulation

Results

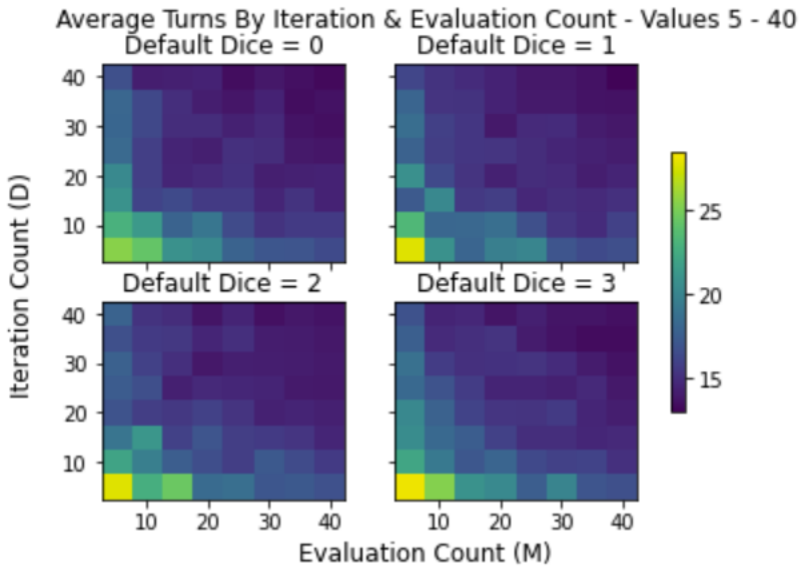
The first parameters we chose to examine was that of policy iteration count, iteration count, as well as default dice. Ultimately, what we want to find through this portion of our experiment is in what manner different combinations of iteration and evaluation counts affect expected turn count as well as how often the result yields an infinite loop. Additionally, we want to gauge how these relationships are affected by altering the default dice chosen.

To examine this, we iterated over each of the four die, then over both iteration and evaluation counts for values five through forty in increments of five. Then, within each iteration we ran our Monte Carlo simulation one hundred times, keeping track of the occurrences of infinite loops as well as the expected turn count for successful runs after five hundred test trials.

Note: For the following sub-experiment, we use the following values as constants.

- Epsilon = 0.1
- Gamma = 0.001

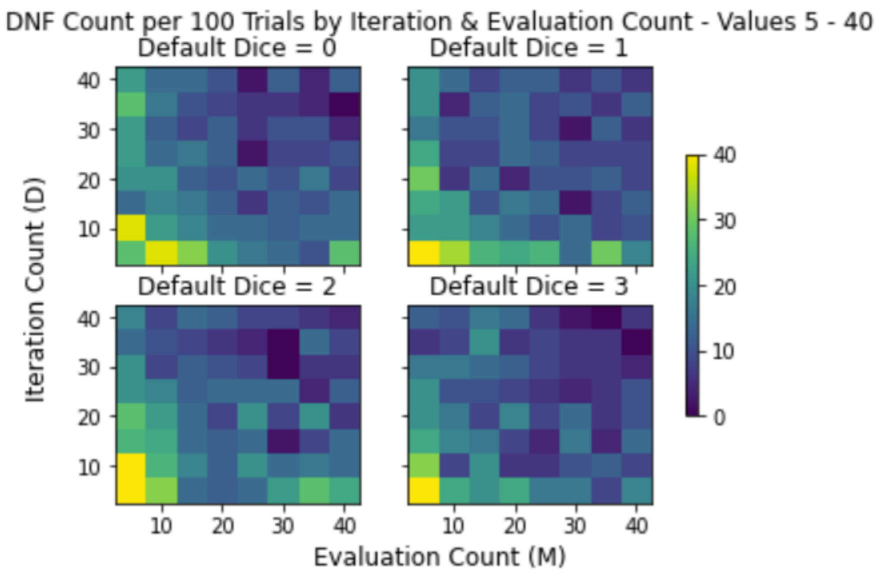
Our observed expected values are as follows:



Through examining the affects on expected turn count, it becomes apparent that all plots appear to be largely symmetric along the $y = x$ line. This suggests that of the two variables iteration and evaluation count, neither has a substantially more significant affect on expected turn count than the other. Knowing this, to save time in our following experiments we provide the same value as parameters for both variables.

As for whether there is a disparity caused by the default dice selection, while there does not appear to be a sizeable impact the graphic does suggest a marginal benefit from using dice number two as the default.

From the same experiment, our observed DNF (Did not finish) count are as follows:



For DNF count, it is difficult to argue that there is as clear a relationship based on the above graphic when compared to the previous. With that said however, we still notice the general trend continues as expected where as iteration and evaluation counts increase, DNF counts decrease. Because of this, we continue with our decision to use shared evaluation and iteration counts in future experiments.

As for whether there is a disparity caused by the default dice selection, there appears to be even less of a difference between the four mesh plots than was observed in the previous graphic. Ultimately, we do not allow this to affect our decision on default dice selection.

With the relationship between default dice, evaluation count and iteration count observed, we next examine the affect that gamma values have on expected turn count as well as the rate of DNFs.

It is important to mention prior to the experiment that while we have already examined the relationship between evaluation and iteration counts, we continue to use them as independent variables in our further experiments, the difference now being that we use shared values for both given that we determined from the graphics that neither has a significantly greater impact on our results than the other.

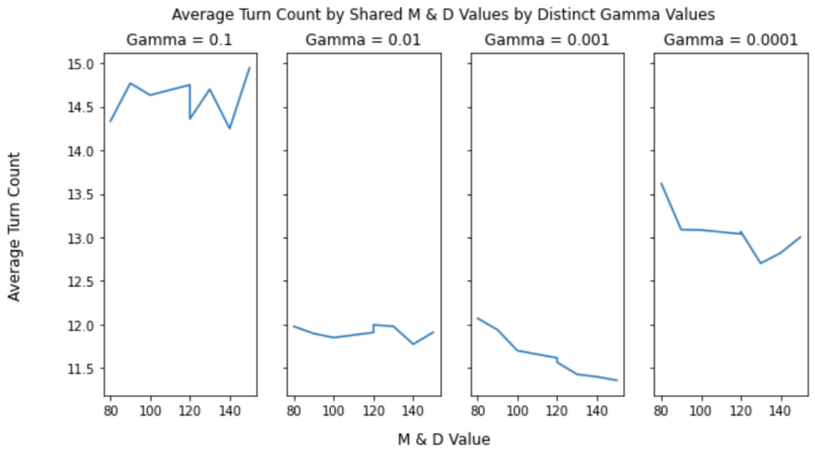
Additionally, from this point we refer to iteration count as variable D, and evaluation count as variable M as that is how they are identified in the algorithm.

Note: For the following sub-experiment, we use the following values as constants.

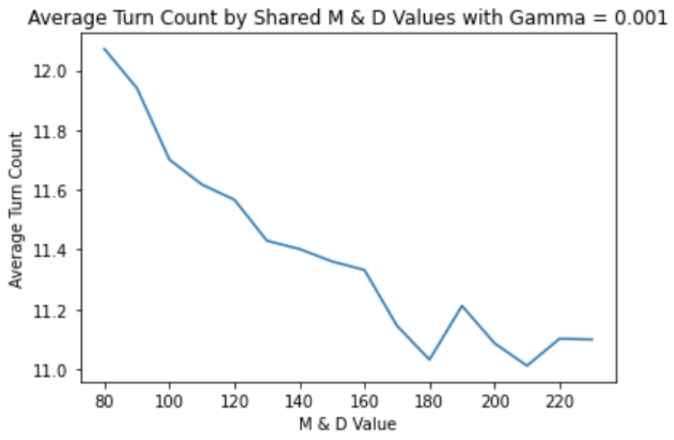
- Epsilon = 0.1
- Default Dice = Dice 2

Much like in the previous experiment, we wish to examine how our expected turn count and DNF rates are affected by our independent variables. In this case, we examine how they are affected by shared M and D values, and compare the results against those produced by differing gamma values. Additionally, we continue to execute each combination of independent variables one hundred times, and within each run test trials five hundred times.

The results for our observed expected values are as follows:

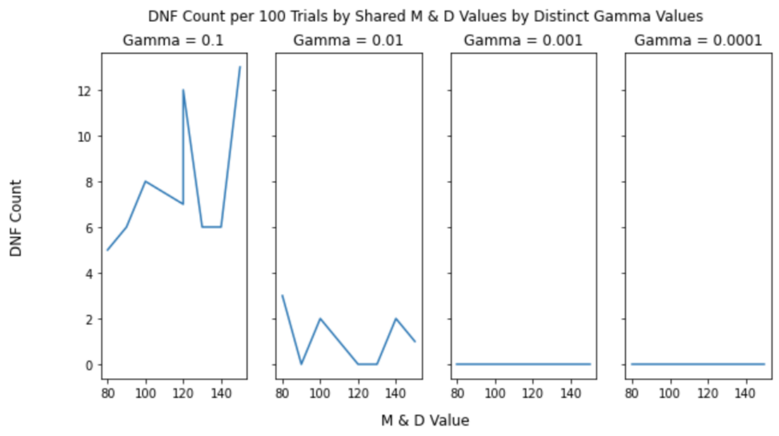


Examining the above graphic, it appears that the best results with regards to expected turn count appear with a gamma value of 0.001. Additionally, we note that while the other three plots appear to have largely plateaued, the plot demonstrating a gamma value of 0.001 appears to continue a downwards slope. For this reason, we expand the domain to better understand how low the expected turn count can reach with the current parameters in place.



As suggested by the slope of the original gamma = 0.001 graph, the average turn count does appear to continue to decrease as the shared M and D value increases. Ultimately, the graph suggests the expected turn value plateaus following an M and D value of 180.

As for how the value of gamma affects the DNF rate of our Monte Carlo simulation, our results are as follows:

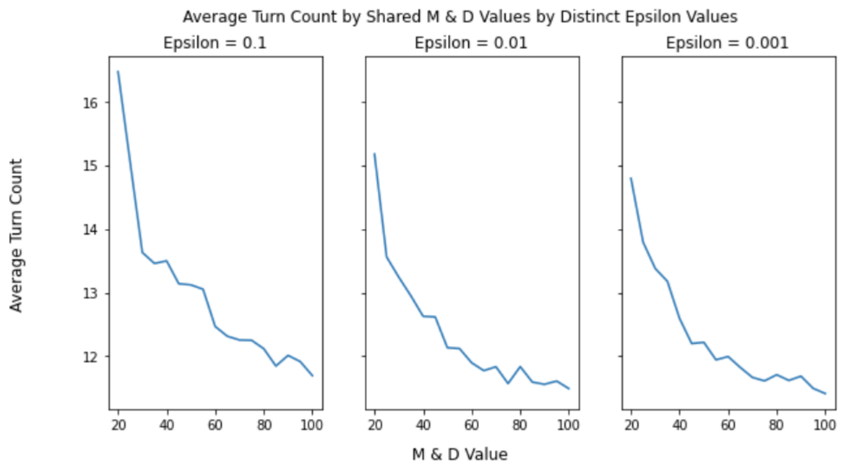


Again, a gamma value of 0.001 appears to have produced the best results, demonstrating no infinite loops through all one hundred trails. This, along with the results of the previous graphics highly suggest that we may expect this value to yield the best performance from our Monte Carlo simulation.

Finally, with an optimal gamma value of 0.001 identified, we next examine the relationship between expected turn count and DNF rate by our final independent variable of epsilon.

This final sub-experiment follows much of the same structure as the previous two; we measure expected turn count against a shared M and D value, comparing the results with those coming from alternate epsilon values.

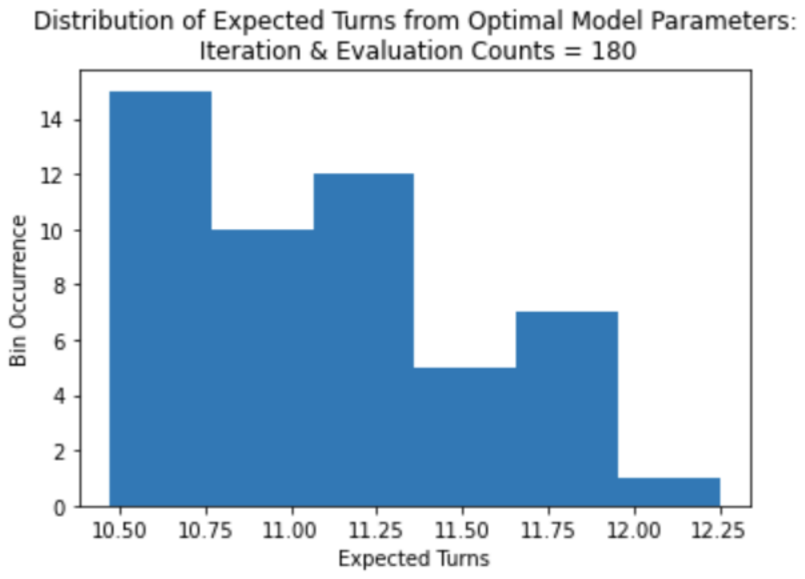
The results for our observed expected values are as follows:

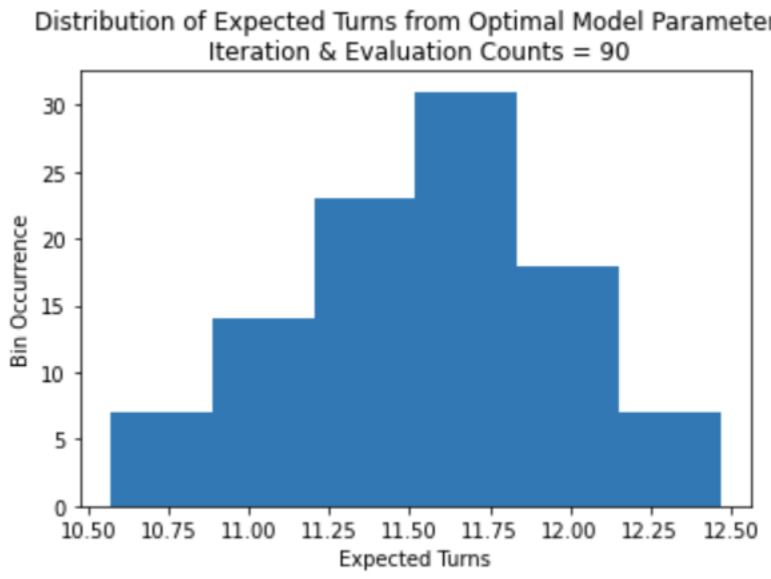


Examining the above graphic, while it does appear that smaller epsilon values do produce better results for the same shared M and D value, at the same time is appears that with large enough M and D values, the results will eventually reach the same expected turn count with enough iterations and evaluations.

This is not to say that there is no benefit towards using the lower epsilon values of course, only that the value for epsilon perhaps is less conducive to improved performance in our Monte Carlo experiment as the value of gamma is.

At this point, we have accounted for all independent variables in our Monte Carlo simulation, and found the values which optimize its performance; a gamma value of 0.001, a shared M and D value of 180, a default dice of dice 2, and an epsilon value of 0.001. Thus, to examine the performance of our simulation with these parameters in place, we run fifty trials and examine the distribution of the results. Additionally, we examine the distribution of the same model with a shared iteration and evaluation count of 90.



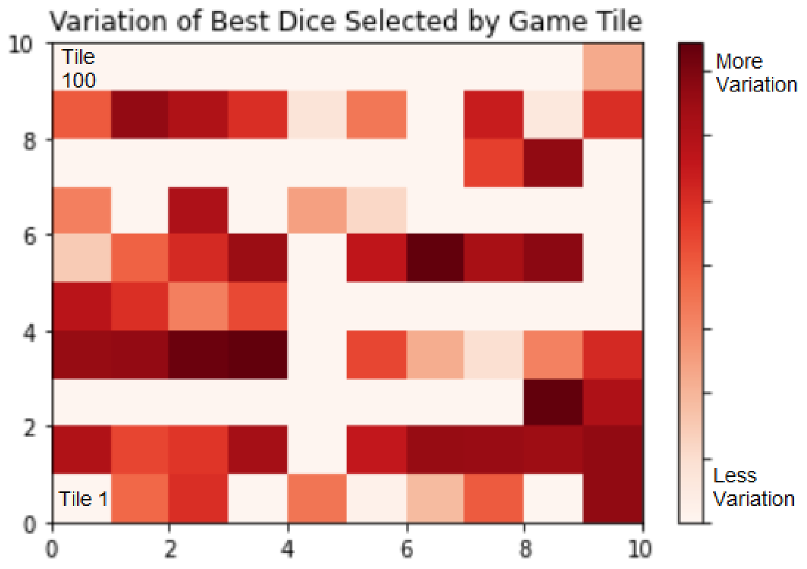


Interestingly, we notice that the distribution of the two histograms are different from one another; The first showing a negative linear trend, and the second exhibiting a normal distribution. This suggests that with our optimal parameters in place as well as the makeup of the game itself there is an absolute floor to how low our expected turns can be which in our case is approximately 10.5 turns. This of course makes sense intuitively, as 7 turns is the fewest possible needed to complete the game, and that is only in the case where every roll comes out to be exactly what is needed, which if you know anything about statistics, after 7 rolls becomes quite unlikely.

Furthermore, this suggests that we could theoretically reduce the variation in expected turns to nearly zero from the same model parameters if we were to use a large enough iteration and evaluation count. However, as found earlier, the improvement gained from increasing those values depreciate fairly significantly following a value of 180.

Ultimately, this highlights the unique issue that plagues Monte Carlo simulations more than the other methods examined in this paper; consistency is far more difficult to establish as is the nature of being founded upon random number generation.

To further examine the variation in the optimal dice selection from the same model, we compare the values of all one hundred trials with one another and generate a mesh plot where tiles that are more red showed more variation.



Examining the above graphic, it is evident that to a certain extent, areas with more and less variation are clustered together as demonstrated by the repeated occurrence of multiple tile-spanning horizontal strips of fairly homogeneous coloring. This makes sense, as adjacent tiles have very similar immediate ”tile prospects” in terms of how many chute and ladder entry points exist in the next few tiles.

With that said however, It is difficult to attribute more variation with more "special tiles" (i.e. either a chute or ladder entry point) and less variation with fewer as we initially attempted to do. This is the case because we notice in the third and eighth rows where there are very few special tiles yield long sequences of low variation in optimal dice selection which is then contrasted by the fifth and tenth row similarly exhibiting low variation while having far more special tiles than average. At this point we acknowledge that the root behind variance of optimal dice selection with regards to tile location and future tile types is more complex than we can explain at this point of our experiment.

Monte Carlo Conclusion

In our Chutes Ladders Monte Carlo simulation experiment, we first examined the effects that the parameters to our simulation had on both the expected turn count and DNF (did not finish) rates per one hundred trials. These parameters included iteration count, evaluation count, default dice choice, gamma value, and finally epsilon value.

Through our experiments we identified that, as expected, increased values of both iteration and evaluation counts improved both the expected turn count as well as the DNF rate. Additionally, we noticed that neither of the two independent variables had a significantly greater impact than the other, and for that reason in the following experiments we made use of a shared value for both to increase efficiency. Finally, we noticed that ultimately the choice of our default dice did not appear to significantly affect either of our dependent variables, though it was suggested that there may be a marginal benefit to using dice two as the default.

From this point, we examined the affect of the independent variable gamma, the factor by which we modify the expected sum of rewards for the given state-action pair following the results of a given trajectory generation. This was accomplished through plotting both the expected turn count and DNF rate for shared iteration and evaluation counts. Then, we compared the results with those of the same simulation with only distinct gamma values. This resulted in our finding that a gamma value of 0.001 was the best option. Using this value, we increased the domain to conclude that the optimal shared iteration and evaluation count is approximately 180.

The final independent variable we examined was epsilon, the probability that in each trajectory generation that we use the current identified best dice available as opposed to choosing at random. While we found that a lower epsilon value performed better for the same shared iteration and evaluation count, we suspect that with large enough iteration and evaluation counts, simulations will ultimately reach the same performance. This of course does not tell us the lower epsilon values are not the better choice, but that they may be less conducive to optimizing the performance of the simulation than gamma appears to be.

Then, with the optimal simulation parameters established, we examined the distribution between the results coming from the optimized parameters and the results of a less optimized simulation. In this, we found that while approaching optimal parameters, the distributions are fairly normal until they get close enough that they reach a floor in expected turn count of approximately 10.5. Furthermore, we suspect that increasing our iteration and evaluation counts beyond 180 will reduce the variation in the results, though at that point the most common expected turn count is within the 10.5 to 10.75 range, and the computation time increases significantly with higher values, so we again assume the optimal iteration and evaluation count to be 180.

Finally, we examined the variation in our observed results using the optimized parameters. In doing so, we noticed that there were indeed distinct portions of the game board that experienced high variation and others that showed very little variation. While we did not isolate the exact reasoning behind this variation beyond simply the fact that the basis of a Monte Carlo simulation is randomness, because we found that the variation in expected turn count were fairly predictable using the parameters we examined, we are able to assume the simulation reaches an optimal expected turn count of approximately 10.5 following 180 iterations when using our identified optimized parameter values.

4.3 SARSA

Results show that the SARSA method provides the optimal solution of 10.4 over 1000 iterations, concluding that SARSA does not provide the optimal solution. In our implementation for chutes and ladders with SARSA, the policy adapts quicker based on expected value calculations for the next state.

As there is a negative reward associated with making an action that is not the optimal path, Sarsa learns the safe path by taking the action selection method into account when learning. Since Sarsa learns the safe route, it receives a higher average reward per trial than Q-Learning even though it does not take the optimal path.

4.4 Q-Learning

After implementing the Q-learning algorithm we met our first challenge which was to tune the parameters α and ϵ. We found that both parameters should decrease as the algorithm is repeatedly executed, updating the Q and P tables. Say we execute the algroithm n times, then we run the algorithm $n/3$ times with $\alpha = 1$ and $\epsilon = 0.1$, then we run the algorithm another $n/3$ times with $\alpha = 0.5$ and $\epsilon = 0.05$, finally we run it another $n/3$ times with $\alpha = 0.1$ and $\epsilon = 0.01$. This makes sense because at the beginning we want to learn a lot so Q values should be updated aggressively to quickly determine the best action in each state and we also want to try out a lot of other actions to determine which one is best. But as we determine which actions which are best in each state we want to explore less and less as doing so will not improve our policy.

Next we used our findings to determine the number of iterations required to train an optimal policy. We quickly found that iterations numbering 1-75 tended to yield policies that simply did not ever reach the hundredth square as the policy caused the agent to enter into an infinite loop. However we did find that policies yielded through 75 plus iterations of the algorithm yielded effective policies. These policies

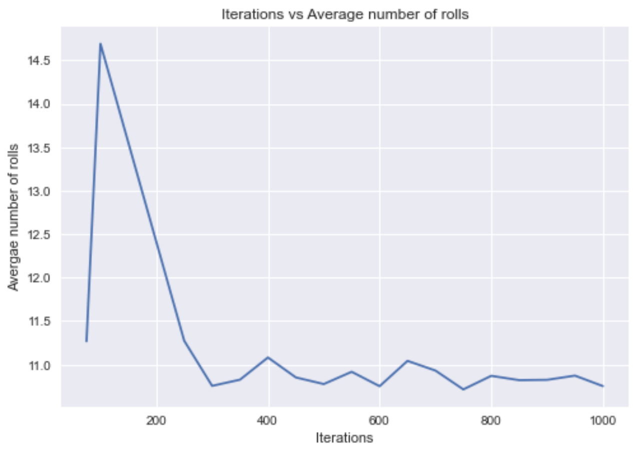


Figure 2: Iterations vs Number of rolls

beat the game in anywhere from 10.5 rolls to 14.6 rolls as can be seen in figure 2.

Interestingly we found that more than 250 iterations of training did not increase the effectiveness of the policy and that after that point increasing iterations would cause the effectiveness of the policies to fluctuate. This can most likely be attributed to the element of randomness present in the game. With these results the Q-learning algorithm was effective in finding the optimal policy for the game but the algorithm does require a considerable number of iterations to arrive at this policy.

5 Conclusions

Method	Iterations	Peak Performance	Variation of Results
Dynamic Programming	2 (3-4 Seconds)	10.2	Minimal
Monte Carlo	22,000 (3-4 Seconds)	11	High
Q-Learning	250 (<1 second)	10.6	Medium
SARSA	10^4	10.4	Medium

Through our experiments we found that while all algorithms were largely successful in reaching an expected turn count close to 10, we ultimately realized each different method found success through fairly different paths.

Unsurprisingly, dynamic programming was able to achieve the lowest expected turn count of all the methods. This stems from the fact that it is regarded as an offline method, and in effect perfectly calculates the expected sum for each tile through starting with the final tile, and working downwards. As a result, variance is largely a non factor with regards to dynamic programming. While this method poses impressive accuracy, the drawback of time performance does not scale up as well with more difficult problems. Though in the case of our project and its limited scope, dynamic programming is irrefutably the best choice.

Monte Carlo simulations on the other hand performed most adversely to that of dynamic programming. To illustrate, being founded upon random simulations there was far more variation in the observed results, and certainly no guarantee of finding the most efficient set of optimal dice selections. This, compounded with a similarly high computation time and higher than average peak performance suggests that for the given experiment, Monte Carlo simulations are the worst suited of the four to efficiently and accurately solve the problem.

Finally, the two reinforcement learning algorithms Q-Learning and SARSA demonstrated effectively a middle-ground performance compared to that of dynamic programming and Monte Carlo simulations. That is, both algorithms showed more efficient computation time than that of dynamic programming, and more accurate results than that of Monte Carlo. This is not completely unexpected as these reinforcement learning algorithms largely stemmed as an improved version of Monte Carlo simulations, and for that reason are expected to outperform Monte Carlo simulations in nearly all facets. As for their merit compared to that of dynamic programming, the choice of which method to use largely stems from the complexity of the problem they are being asked to solve. In our case, we have a fairly simple problem to solve in identifying the optimal dice to use for each tile in the game of Chutes and Ladders, and for that reason dynamic programming is the clear choice. However, as reinforcement learning shows to perform better in terms of computation time, for problems where that is a legitimate concern the best choice often is indeed one of Q-Learning or SARSA.

6 Appendix

Khoi Le: Dynamic Programming, Mathematical Formulation of DP, DP for Chutes and Ladders
Mark Raney: Monte Carlo Simulation, Conclusion
Philipp M. Srivastava: Introduction, Background, Q-Learning
Amna Khalid: SARSA, Sarsa background,