Project Report – Foundations of Artificial Intelligence

Jorge Delgado
Delgado.j@husky.neu.edu

Abstract

The goal of the project was to use reinforcement learning to design an implementation that would provide a suitable policy for Blackjack. The algorithm used was Q-Learning with actions taken using an Epsilon Greedy policy for the player, and a fixed policy for the dealer. Allowing the player to improve their chances of winning the game, while still abiding by the rules set by the Blackjack implementation.

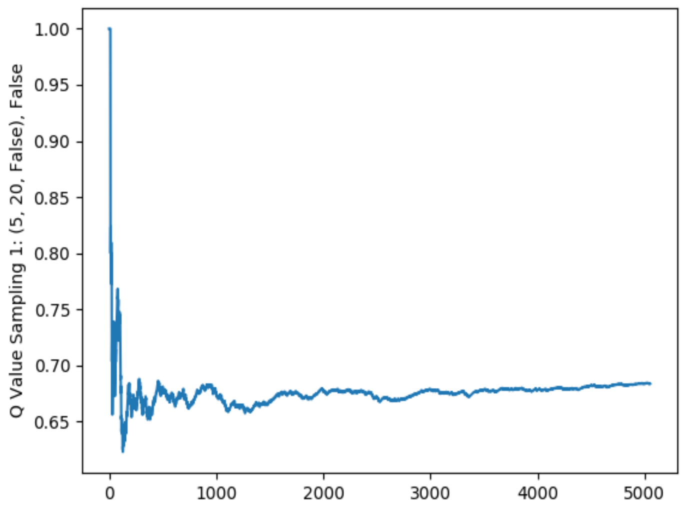
The policies, coupled with the toy Blackjack implementation, allowed me to devise an optimal policy in order to play the game of Blackjack with a higher probability of winning.

Introduction

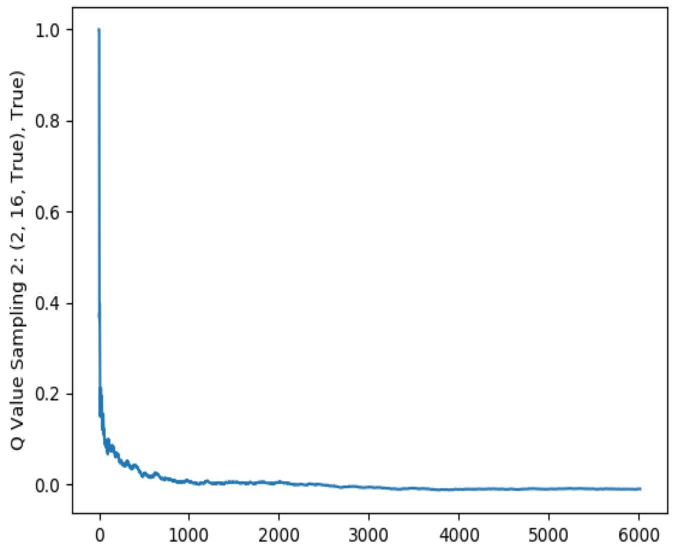
Q-learning was used because it is "an excellent method for approximating an optimal blackjack strategy because it allows learning to take place during play."[1] Each state consists of the dealer's shown card valuation, player's hand valuation, and the ace status.[2] Q-learning allows for a comprehensive, evaluation algorithm to determine the score obtained by either hitting, or staying, given the current state.

Because Blackjack has a relatively small number of states, Q-learning provides a complete methodology and algorithm for creating an optimal policy. Furthermore, because only two actions may be taken at each state—either staying or hitting—the Q-Learning values rapidly converge to their final values.

Q-Value Random Sampling 1 – (Figure 1) [X Axis is Iterations]



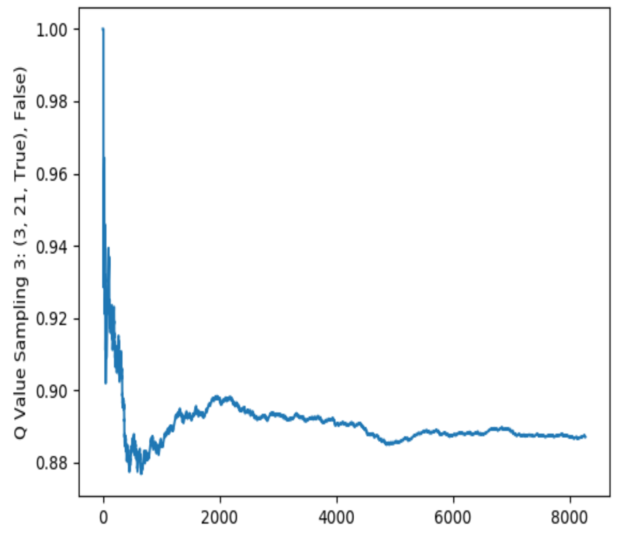
Q-Value Random Sampling 2 – (Figure 2) [X Axis is Iterations]



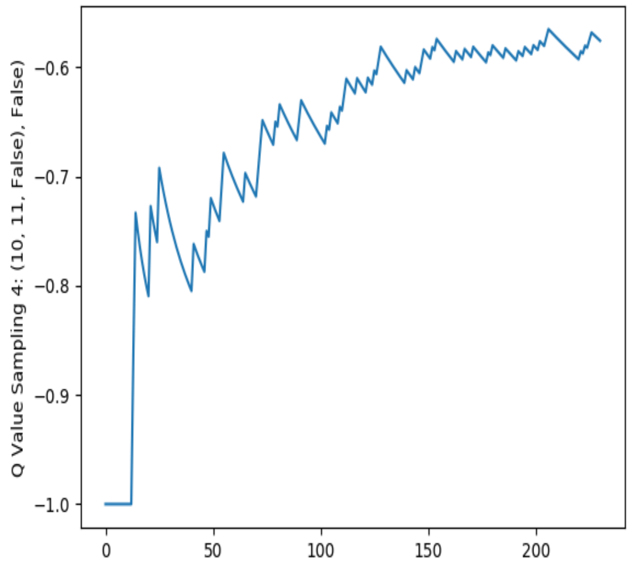
[1] de Granville Charles, Applying Reinforcement Learning to Blackjack Using Q-Learning, http://cs.ou.edu/~granville/paper.pdf, access on April 14th, 2017.

[2] These states and their meanings are discussed in a later section.

Q-Value Random Sampling 3 – (Figure 3) [X Axis is Iterations]



Q-Value Random Sampling 4 – (Figure 4) [X Axis is Iterations]



Figures 1 through 4 are random samplings. Below is the breakdown of what each sampling depicts.

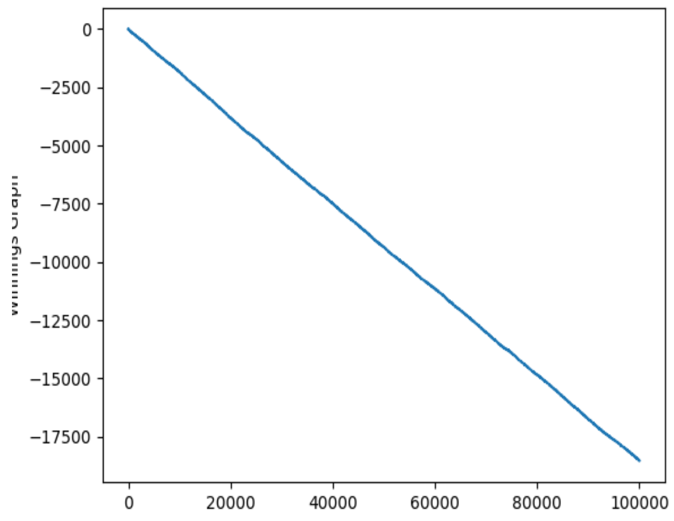
- Figure 1
  - Dealer hand valuation: 5
  - Player hand valuation: 20
  - Ace Status: False
  - Action: Stay
- Figure 2
  - Dealer hand valuation: 2
  - Player hand valuation: 16
  - Ace Status: True
  - Action: Hit
- Figure 3
  - Dealer hand valuation: 3
  - Player hand valuation: 21
  - Ace Status: True
  - Action: Stay
- Figure 4
  - Dealer hand valuation: 10
  - Player hand valuation: 11
  - Ace Status: False
  - Action: Stay

This random sampling shows a wide array of different states: some where the dealer is at an advantage, some where the player is at an advantage, and some where they are fairly equally matched. Despite the case, the graphs show a rapid deviation from the initialized Q-Value and stabilization over time with minute differences between iterations.

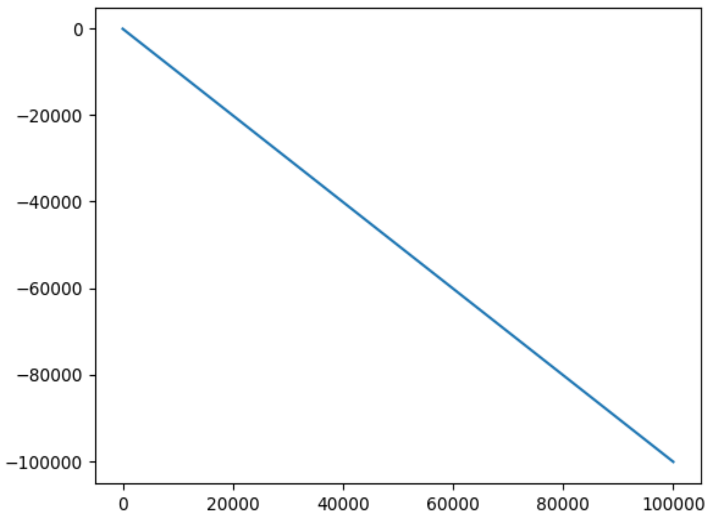
The use of a Greedy-Epsilon policy was also instrumental in quickly determining the optimal strategy. This strategy ended up being “optimal in the long-term sense”[3] because it provided a way to explore the state-action mapping space, allowing for the exploration of potentially better policies.

[3] S. Sutton R., and G. Barto A., Reinforcement Learning: An Introduction, http://people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf, page 69, accessed April 14th 2017

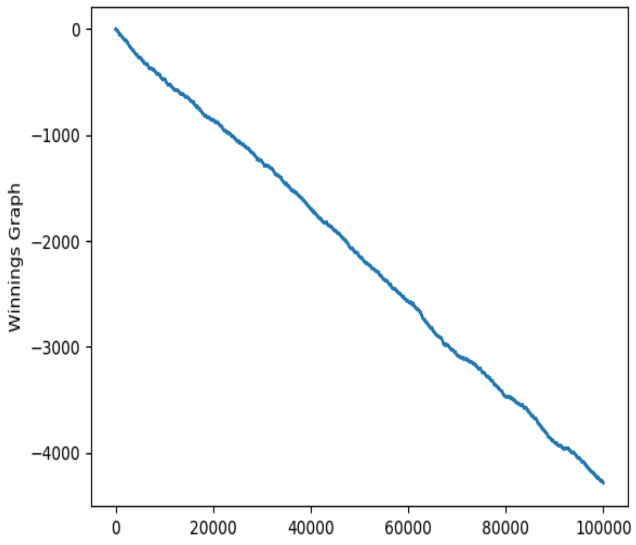
Winnings Curve – Always Maximizing Q Value (Figure 5) [X Axis is Iterations]



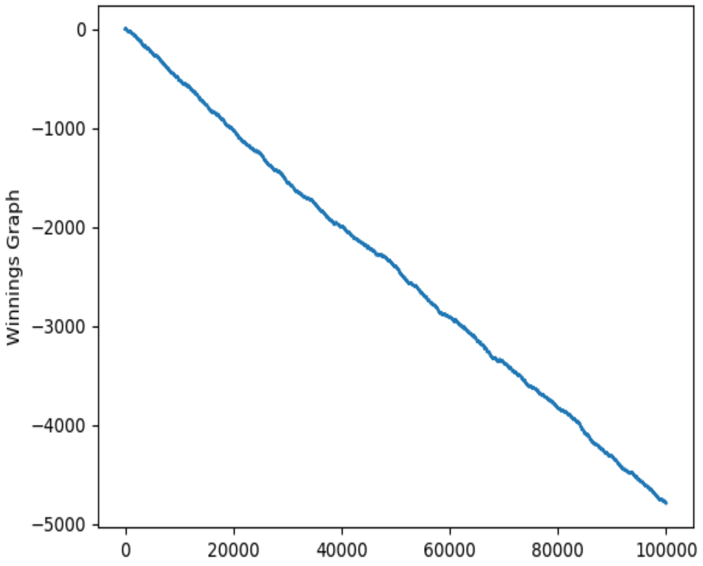
Winnings Curve – Always Picking Random Values v2 (Figure 7) [X Axis is Iterations]



Winnings Curve – Always Picking Random Values v1 (Figure 6) [X Axis is Iterations]



Winnings Curve – Learned Policy (Figure 8) [X Axis is Iterations]



The previous figures depict several graphs illustrating the results of different approaches. It is important to note that despite the learned policy producing the most favorable results, the game of Blackjack as defined, is intentionally rigged against the player.[4] As such, the expected winnings[5] per game after one million Q-Value iterations and ten thousand games was: -0.0476100580217. In other words, despite an optimal policy, after enough games you will have a net loss in the game of Blackjack as defined in this problem.

As the graphs above illustrate, if I did absolutely zero exploration (Figure 5), I would end up with very low winnings for Blackjack. However, if I only took random actions, I could either end up with a decent (Figure 6), or absolutely abysmal winnings policy (Figure 7) for Blackjack. Resulting in a very unreliable approach.

The winnings graph for the Learned Policy (Figure 8) is the ideal marriage of randomized actions, while also using actions I know to be ideal based on my learned Q-Learning valuations.

Throughout this project report, I will look at my implementation of a toy Blackjack environment, as well as the algorithmic implementations used to create an optimal Blackjack policy.

Background

In order to understand the solution provided in this project, it is essential to illustrate the rules established for this toy Blackjack representation. The rules and underlying assumptions are:

1. Player cannot double down or split.
2. Dealer policy is fixed.
   a. Dealer will continue to hit until it has a hand value of 17, or a “stand on All 17’s” policy.[6]
3. You gain + 1 for a win.
4. You gain a – 1 for a loss.
5. Ties are possible, and they net a + 0.
6. It is possible for both dealer and player to bust.
7. Player can transform an ace from a + 1 to + 11
   a. Note: Aces are considered + 1 initially, and a + 10 when transformed.
8. Aces are transformed to + 11 automatically if the hand will not go over 21.
9. If the hand valuation is below 19, there is an inherent deviation towards hitting for the player.
10. The implementation is not concerned with the dealer learning a better policy, only the player.
11. There was no discounting.

Based on these assumptions, the following parameters were used:

- Q-Value iterations: 1,000,000
- Number of games played with learned policy: 100,000
- Alpha: 1
- Epsilon: 0.1[7]

The following Q-Value formula was used to update the Q-Values for the state-action mapping:

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

A state is defined as a triple with:
1. Dealer Shown Card
2. Player Hand Valuation
3. Ace Status, which determines if the player has an Ace that can be transformed into a + 10 or not.

A hand is defined as tuple with:
1. The value of the hand
2. The Ace Status

The Q Value to action mapping consists of a state, and an action (either hitting or staying) inside a dictionary. Therefore, all possible states where a blackjack is possible are considered. At each iteration, the Q-Values are updated for both actions and the higher of the two is selected as the desired action for the policy.

My policy deals with both the case where the Ace is usable in the player’s hand and, and where there is no usable Ace.[8] Through this approach, I’m able to devise the optimal strategy to take, given a specific player hand, their ace status, and the dealer’s hand.

Related Work

Q-Learning is the ideal implementation in order to learning an optimal policy for Blackjack due to its quickness, versatility, and ability to learn as you play.

[4] More on this in the Background section.
[5] Expected Winnings is defined as the total cumulative winnings, divided by the number of games played. In this case, 10 thousand.
[6] The Big "M" Casino, "Do dealers and players have to abide by the same black jack rules?", http://www.bigmcasino.com/learn-more/do-dealers-and-players-have-to-abide-by-the-same-black-jack-rules/, Accessed April 15th, 2017.
[7] BURLAP: Tutorial: Creating a Planning and Learning Algorithm, Brown University, http://burlap.cs.brown.edu/tutorials/cpl/p3.html, accessed April 15th, 2017
[8] Full policy is listed in policies.txt in the same directory as this file.
It was removed from this document as it was too large.

However, other methods are possible. One of which is the Monte Carlo approach.

The Monte Carlo approach relies "on repeated random sampling to obtain numerical results."[9] As a result, it can be instrumental in devising a policy that could converge on the optimal policy through exploration. However, Monte Carlo solves a problem through average sample returns.[10] Therefore, Monte Carlo estimate of other actions will not improve with experience.[11]

A major problem with Monte Carlo is its expensive nature. Because Monte Carlo has no knowledge of its environment and relies almost exclusively on its samples, it can often lead to a large number of samples needed in order to achieve an optimal policy. Therefore, "convergence is rather slow and in practice a very large number of Monte Carlo trials are often used." Therefore, it is not ideal for my implementation.

Another option would be the SARSA algorithm. While Q-Learning determines values based on the maximum possible Q value of all possible actions, SARSA relies on the value obtained through a specific action based on a policy.[12] Therefore, in theory, it could potentially provide a more optimal policy than Q-Learning could.

However, in practice, SARSA tends to produce similar results to Q-Learning and an almost identical policy. This is due to the action that produces the best Q value, is often the most valuable one as well. Furthermore, "Q-learning learns the fastest path to the exit" when compared to SARSA.[13] Because SARSA would more than likely produce similar results to Q-Learning, I decided to use Q-Learning and avoid the additional complexity.

Project Description

The project uses a toy blackjack model to implement Q-Learning coupled with a Epsilon-Greedy policy for the player, and a fixed policy for the dealer in order to learn the optimal policy for a given state-action coupling.

In terms of structure, the project has the following Python 2.7.13 files:

- project_blackjack.py
  - The main function for the program.
- actionhandler.py
  - Handles all action decision making.
- cardhandler.py
  - Handles all card functionality.
- constantshandler.py
  - Contains the constants used by all other files.
- graphhandler.py
  - Handles the graphing functions.
- statehandler.py
  - Handles all state creation, select, and extraction functions.
- testsuite.py
  - A custom test suite that tests every function in the program with a multitude of tests.

The program can be run by issuing the following command from within the code directory:

$python\ project\_blackjack.py$

This will run the project which generates the Q Values using Q-Learning and produces the following internal variables:

1. The Q Values
2. Four random samplings used for generating a graph (currently commented out)
3. Calculate the best policy given that an Ace is present and usable within the player's hand.
4. Calculate the best policy given that an Ace is not present nor usable within the player's hand.
5. Output of the test suite.

The test suite is a custom program that will run over sixty tests on the project. If they succeed, it will return the results of each test, and finally a notification of "All tests completed successfully," assuming all tests completed without errors.

To stop the test suite from running, simply comment out the following line in project_blackjack.py:

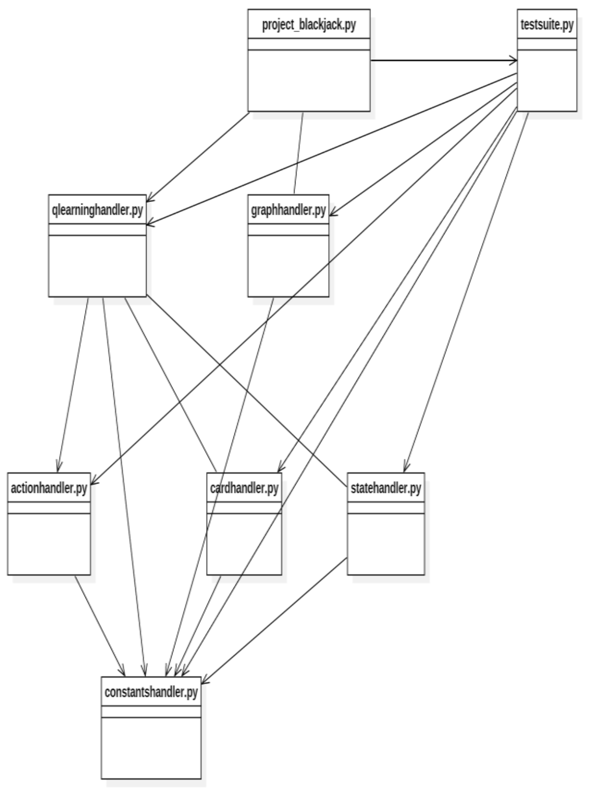
```
testsuite.Tests()
```

This will stop the Test Suite from running. When this is done, the only output will be:

1. The parameters used to run Q-Learning
2. The expected winnings per game for the predefined number of games
3. The optimal Policy where ace status is: True
4. The optimal Policy where ace status is: False

[9] Quantitative Journey, Reinforcement Learning - Monte Carlo Methods, http://outlace.com/Reinforcement-Learning-Part-2/, Accessed April 16th 2017.
[10] University Bermen, Dynamic Programming, Monte Carlo Methods and Q-Learning, Accessed April 17th 2017.
[11] University Bermen, Dynamic Programming, Monte Carlo Methods and Q-Learning, Accessed April 17th 2017.
[12]UNSW Sydney Engineering, Reinforcement Learning, http://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms, Accessed April 17th, 2017.
[13] Northeastern University, CZXTTKL, Difference between SARSA and Q-Learning, https://nb4799.neu.edu/wordpress/?p=1850, Accessed April 18th, 2017.

Project Structure (Figure 9)



Q-Learning Algorithm Workflow (Figure 10)

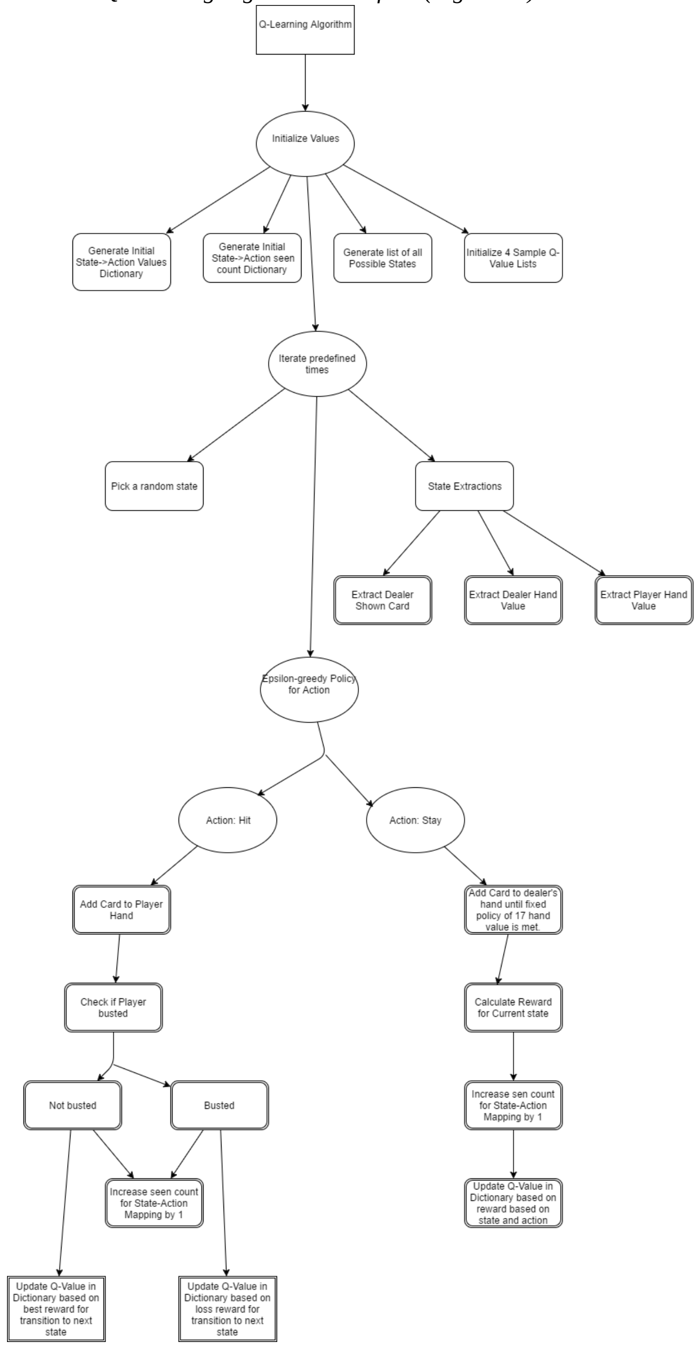

Q-Learning Algorithm
Initialize Values
Generate Initial State->Action Values Dictionary
Generate Initial State->Action seen count Dictionary
Generate list of all Possible States
Initialize 4 Sample Q-Value Lists
Iterate predefined times
Pick a random state
State Extractions
Extract Dealer Shown Card
Extract Dealer Hand Value
Extract Player Hand Value
Epsilon-greedy Policy for Action
Action: Hit
Action: Stay
Add Card to Player Hand
Add Card to dealer's hand until fixed policy of 17 hand value is met.
Check if Player busted
Calculate Reward for Current state
Not busted
Busted
Increase sen count for State-Action Mapping by 1
Increase seen count for State-Action Mapping by 1
Update Q-Value in Dictionary based on reward based on state and action
Update Q-Value in Dictionary based on best reward for transition to next state
Update Q-Value in Dictionary based on loss reward for transition to next state

Figure 9 illustrates the file structure of the project. project_blackjack.py calls upon qlearninghandler.py, graphhandler.py and testsuite.py to produce the Q Values, calculate the expected earnings, calculate the optimal policy for all possible actions, and run tests. Furthermore, Qlearninghandler.py calls upon actionhandler.py, cardhandler.py, statehandler.py and constants handler to perform specific functions used in determining the Q Values. All files call upon constantshandler.py as it contains all of the predetermined variables for the project.

Q-Learning Algorithm Pseudocode (Figure 11)

```
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Choose $a$ from $s$ using policy derived from $Q$
    Take action $a$, observe $r$, and $s'$
    $Q(s, a) \Leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \Leftarrow s'$
until $s$ is terminal
```
[14]

Figure 10 is a work flow of the logic behind the project's Q-Learning algorithm implementation, while Figure 11 is the pseudo code used for inspiration for the implementation in this project.

Experiments

The following algorithmic experiments were run once the Q-Learning algorithm was implemented:

1. Always maximizing Q Values, never randomizing actions.
2. Always randomizing actions.

My policy for the action selected is based on Epsilon-greedy, therefore, certain actions will not necessarily be the ones that produce the highest Q value, but instead, random. This allowed me to explore the state space thoroughly, to ensure I have an optimal policy. However, it was important to evaluate this approach itself.

In test #1, the algorithm was modified to never pick random actions. Instead, only pick the action that maximized Q-Values. As a result, my winnings graph (Figure 5) for one hundred thousand games suffered significantly. The reason for this decline is that the state space was not thoroughly explored, resulting in the policy picked being a safe, but not optimal path.

In test #2, the algorithm was once again modified to never pick the Q-Value maximizing action. Instead, it selected a random action. This produced 2 drastically different results. Figure 6 shows a favorable result, where the random actions picked resulted in a similar Q values to that of my learned policy. However, Figure 7 illustrates what happens when the same algorithm does not produce favorable results. This graph shows abysmal performance in regards to winning rates. Illustrating that while random actions might produce favorable results for a set of iterations, a subsequent iteration could produce a significantly sub-optimal policy.

Dozens of non-algorithmic tests were also run on the project. The test suite is a compilation of over fifty of them, each testing their specific function for weaknesses and unfavorable results. These tests permit the experimentation of the individual algorithms within the project, testing each for undesirable results.

Conclusion

The project was instrumental in furthering my understanding of Q-Learning, and its implementation within a toy Reinforcement Learning environment. Through its implementation, the algorithm's intricacies became apparent. My project produced a concise and complete policy for playing the game of Blackjack while optimizing the chances of winning.

Through experimenting with different action-selection strategies, I determined that Epsilon-Greedy was an acceptable approach, producing consist results. Furthermore, the exploration of the state space not only allows the selection of a better path to the goal for a specific state, but also expands the chances of winning. In short, through the interactions with the environment, the implementation was able to demise an optimal action, given a specified state, for the game of Blackjack.

[14] de Granville C, Applying Reinforcement Learning to Blackjack Using Q-Learning, http://cs.ou.edu/~granville/paper.pdf, Access on April 14th, 2017.