Introduction:

Blackjack is marketed as a simple game, one that casino novices should pick up with ease. The premise is simple: draw cards until your total card value is as close to 21 as possible and you beat the dealer’s hand. Past analysis on the game has been done, and it’s been found that blackjack has the best odds of any table game in the casino, with only a 0.5% house edge when played properly.

While casino novices often decide to risk their money on the blackjack table, they often don’t play with the optimum policy. Blackjack is a game with many states (over 200,000 for just a 26-card deck), and knowing how to maneuver through all of them to give yourself the best chance to win can be confusing.

Some companies try to capitalize on this by selling ‘Basic Strategy Cards’, which claim to give the player information on how to approach betting at every state. However, these cards don’t account for differences in state such as the number of cards left in the deck and the card count at certain points. I believe that I can make a program that can improve the basic strategy of the starting player, by providing information on how to bet, and what their odds are in every situation on the board.

Project Summary:

I plan on creating an MDP that properly plays out the game of blackjack. Using this, we can compute the optimum policy with each of the states associated with a game of blackjack. At the end of the learning process, the user will be able to enter a starting state (card count, player cards, dealer card shown) and be returned with the optimum policy and expected return by which to play with. If the program works correctly, the user should generally see positive returns while playing, especially if they decide to not bet large in games where the odds are not in their favor.

Additionally, I would like to do some additional work regarding the variability involved in playing a concrete number of blackjack games. While value iteration will give us an expected value of return (average), I’d like to estimate the dispersion involved in playing blackjack a finite number of times. With this information, we can give the player a better estimate of the variability in draws, at how different factors can affect this as well.

State Space Minimization:

Challenges: The biggest challenge that is faced when creating an algorithm like this is to minimize the state space. While blackjack seems like it would have a relatively small state space given the limited number of cards dealt during every hand, this is not the case. For each state, we must keep track of information regarding the total number of each card left in the deck. This is because we are tracking the probability of each hand with no replacement, meaning that once each card leaves the deck, the probability for each new card being drawn changes. At the end of each action (except those that win or lose the game), the total number of each card is inputted back into the state.

Additionally, the number of cards drawn in a single game of blackjack is quite variable. Since an MDP evaluates every state possible no matter how small the probability, states where both the dealer and user draw only a 2, 3, or 4 will come up, creating an extended state space if all of these cards are included in the state.

Approach: I approached minimizing the state space in the following way. For each state, 3 separate values were stored in a tuple:

1. Player’s Hand (string with numbers, letters used as code for what exactly is in hand)
2. Dealer’s Hand (string with numbers, letters used as code for what exactly is in hand)
3. Cards Remaining the deck (tuple with index corresponding to number of cards for certain card)

Both the Player’s Hand and the Dealer’s Hand have a specific code that tells the program exactly what is in each hand. The code is as follows:

1. First 1 or 2 characters: Value (numerical representation of value that the dealer or player holds)
2. Letter “D” in characters following Value: Double Down allowed on next action
3. Letter “S” in characters following Value: Splitting allowed on next action (duplicate cards in hand)
4. Letter “A” in characters following Value: One ace in hand who’s value is taken at 11.

By creating a code to represent the different states while not having to store exact card values or the order in which cards are drawn, I hoped to minimize the state space. For example, if the player draws the 4 and a 5 in the initial draw, this would be represented by ‘9D’ state. The ‘9D’ state could also represent a draw of 3 and 6, 2 and 7, etc. This minimizes the state space since you do not have to keep track of the individual values of cards drawn.

Actions:

For the current code, 4 actions can be taken:

1. Begin: Player and dealer each draw 2 cards
2. Stay: Player stays, dealer draws until value is over 17
3. Draw: Player draws one card
4. Double: Player draws one card, dealer draws until value is over 17

To limit the tree width, some actions were eliminated for certain states that reflect simple blackjack strategy. For example, you aren’t allowed to stay if your value is less than 11, as there is not chance of losing if you take another card. Additionally, you are only allowed to double down on your bet with certain numbers, mainly you cannot double down on hand value greater than 12 unless you have an ace in your hand. Lastly, you are not allowed to draw another card if you have an 17 or higher with no ace.

Value Iteration:

Currently, I have implemented a value iteration algorithm similar to that used in earlier class assignments. The goal of this is to find the optimum policy and expected value associated with a certain deck of cards. As currently oriented, this value iteration on the MDP has full information about the state of the problem, including the total number of the cards in the deck and the dealer and player cards. An initial deck state can be inputted into the program to see how the value changes in regards to playing with different deck states, sizes, and counts.

The runtime for a starting deck state of 13 cards is about 10 seconds for the value iteration process, which seems reasonable to me. However, when the number of cards in the deck increases, the state space of the MDP increases exponentially, causing very long runtimes for the program. My goal for the next assignment is to increase the efficiency of my code with larger decks.

Playing a game with a start state of 26 cards (each card type having an equal number in the deck) nets an expected value of 0.02 for each unit that you spend. Unfortunately, I believe that this analysis is wrong, since the odds should be against you for a normal game of blackjack. I need to go back and check my succProbReward function, to ensure that all different actions taken in this section give pay out the proper reward.

Q-Learning:

Additionally, reinforcement learning will be implemented, since value iteration isn’t totally realistic showing of the game. In a real game of blackjack, the player will have knowledge of the count (the state), but no knowledge of the exact probabilities of the reward associated with each action. While the count gives the player a general knowledge of whether a high or low card is drawn, it does not tell the player exactly what cards are left in the deck.

This portion of the algorithm has not been implemented yet.

Features:

For Q-Leaning, the following features are in the planning phase:

1. Current Card Count (true count, way to represent deck state as float)
2. Dealer’s Top Card (only card visible to the player)
3. Player Card Value (total value of players cards)
4. Ace Counter (more aces in players hand provides more flexibility)

Next Submission:

For the next submission, I plan on working on finishing my algorithm, and completing Q-learning as well.

References:

Jensen, Kamron, "The Expected Value of an Advantage Blackjack player" (2014). All Graduate Plan B and other Reports. 524

Professional Blackjack by Stanford Wong, page 31, 1994 ed.

Introduction:

In 1962, Edward Thorpe, published his book, “Beat the Dealer”, summarizing a system that could be used to gain advantage for the player in the game of blackjack. Since then, casino-goers have been scheming, trying to implement this strategy in order to make easy money in this supposed game of chance. The most well known success story associated with card counting stems from the notorious ‘MIT’ blackjack committee, which is rumored to have won more than $500,000 dollars in one night counting cards as a team. While counting cards sounds like an enticing strategy, it’s not an easy thing to pull off. The ability to keep track of the card count and manage the current state of the game is no easy task, and one that only the most cool-headed players could accomplish.

The process of counting cards is as follows:

* For any card with a value of 6 or under, add one to the count.
* For any value 7-9, keep the count the same.
* For any value of 10 or higher, increase the count by one.

The larger the count is, the more in favor the odds are in favor of the player. This is because with high card counts, there are more larger cards in the deck, making it more likely for the dealer to bust.

Utility, Purpose:

The purpose of the project will be to create an engine that learns the optimum policy for playing blackjack while using a card counting strategy. At the end of the learning process, the user will be able to enter a starting state (card count, player cards, dealer card shown) and be returned with the optimum policy and expected return by which to play with. If the program works correctly, the user should generally see positive returns while playing. If there is enough time, a graphical interface will be created to display all of these values for the user for ease of access.

Preliminary Data:

Datasets used to test this algorithm will small and will consist of deck layout inputs. One instance of data would include data of how many of each card are in the deck. From this data, we can create code that will generate a randomized order of these cards (shuffling) and create an associated ‘true deck count’ associated with the permutation of cards from the dataset. From this dataset, we can play various game of blackjack, and come up with an optimum policy for playing a game at a certain state (card count) through a reinforcement learning algorithm.

Baseline and Oracle (Bridging the Gap):

A baseline algorithm was created that play blackjack without incorporating any card counting strategies or letting the player split. To do this, a blackjack class was created in python, and a value iteration was run on the problem to compute the optimum policy of a single round. We found that the expected reward value was \_\_\_, when the player plays only one game against the dealer with a full fifty-two card deck. We believe that we can produce much greater returns than this and find a strategy that allows the player to actually gain money in the process.

To find an oracle (upper bound) associated with this project, some background research was conducted. In many studies, card counting does increase your expected value. From a non-research perspective, individual claim to have won over $100,000 dollars a year playing advantage blackjack. One team out of Seattle even claim that they won over 3.5 million in just a couple of years. While these total value numbers are amazing, it doesn’t give us a good estimate of the expected value of the game, since we don’t know how much was invested in the game. However, there is documentation of card counting providing a 1.2% advantage to the player, with a standard deviation of 3.5 (very high).

Challenges; Implementation:

Card counting dramatically increases the number of states associated with the MDP. Add this in with incorporating the full game of blackjack with doubling down, splitting cards, and variable starting states based off card counts, this problem will get a lot tougher to implement. Defining the state for the MDP will be a difficult decision and keeping only the necessary information will be important in order to minimize our state space and keep the problem efficient.

Additionally, reinforcement learning will be implemented, since value iteration isn’t totally realistic showing of the game. In a real game of blackjack, the player will have knowledge of the count (the state), but no knowledge of the exact probabilities of the reward associated with each action. While the count gives the player a general knowledge of whether a high or low card is drawn, it does not tell the player exactly what cards are left in the deck.

Goals for Next Submission:

For the next submission, I want to minimize the state space associated with this problem to keep the MDP efficient. I also want to work towards creating the MDP blackjack process, complete with doubling down, splitting, and keeping track of the count. MPD should also be modified to incorporate variable betting strategies. These strategies would allow for the expected returns to be greater, as a player could adjust his bet based on the likelihood of winning in a specific state.

References:

Jensen, Kamron, "The Expected Value of an Advantage Blackjack player" (2014). All Graduate Plan B and other Reports. 524

Professional Blackjack by Stanford Wong, page 31, 1994 ed.