An MDP Blackjack Agent

Brendan Reilly
University of Massachusetts Lowell
One University Avenue, Lowell, Massachusetts, 01854
brendanreilly38@gmail.com

ABSTRACT
In this paper an implementation of a Blackjack agent is discussed. The agent uses a Markov decision process (MDP) to learn about the game world of Blackjack and exploits its knowledge to play successfully. Value iteration and q-learning are used, allowing the agent to propagate its knowledge back to every state from the terminal states. Feature extraction is used to speed up this process, as the agent requires fewer training games to learn about the world. A user interactive game was created with the agent to demonstrate the choices it would make at each state.

Author Keywords
Blackjack, Markov decision processes, q-learning, value iteration, feature extraction.

INTRODUCTION
Blackjack and its precursors have existed for hundreds of years. Despite the unfair odds, strategies have been created to maximize success. I wanted to implement an agent that would create its own strategies to be successful at Blackjack. An MDP based agent was used, as MDPs work well with stochastic worlds.

While there were no apparent works specifically related to solving Blackjack with an MDP agent, there were works related to Blackjack in general, and solving general stochastic games.

“Blackjack-Playing Agents in an Advanced AI Course” [3] discusses how Blackjack can be used to teach advanced AI concepts, specifically Markov chains. The students begin with simpler versions of Blackjack named “Whitejack” and “Grayjack.” They build upon what they have learned in those simpler versions in order to compete in a multiplayer Blackjack tournament, pitting their agents against each other. This paper reinforces the fact that a MDP agent is a suitable approach to solving Blackjack, as many MDP concepts can be taught from its game world.

“A Blackjack Simulation for Teaching the Use of Scientific Method” [4] discusses the use of Blackjack in the classroom to teach the scientific method. While only tangentially related to using MDP agents to solve Blackjack, the statistics of Blackjack win rates and strategies are discussed, specifically the results of a “no bust” strategy.

“Optimal Strategies for Testing Nondeterministic Systems” [5] discusses the use of test graphs to optimally solve stochastic games. The goal is to find an optimal strategy by adding and removing edges to a game path, similar to how search algorithms work. This method, however, would create a different path for each specific game state. It seems to be a cross between MDP and a search algorithm.

PROJECT DESCRIPTION
To implement an MDP solution for Blackjack, I first had to create a playable version of Blackjack. I created three separate classes, a card class, deck class, and shoe class, to act as my base of cards. A card consists of two values: rank and suit. Rank is an integer from one to thirteen (1-10, Jack, King, Queen), and suit is an integer from one to four (Clubs, Spades, Hearts, Diamonds). The deck and shoe classes were lists of cards and decks, respectively, both having the ability to draw and shuffle. Once I created this base, I then made a dealer class that contained all the game rules and kept track of both the player and dealer hand. I used standard Blackjack rules, but the only action the player could take was to hit or stand. The most important function
in the dealer class was handValue, which calculated the value of both the player and dealer hand and stored them in a data member. This function gave all cards with rank above ten a value of ten, to keep in line with Blackjack rules. It also raised flags if an ace was present in either the player or dealer hand, allowing for distinction between “hard” and “soft” hands. Once I implemented the game world, I began work on the agent.

The MDP Q-Learning Agent
I started with a basic q-learning agent with no feature extraction. The agent consisted of a map of state to qvalues, where a qvalue held the value for an action (either hit or stand) for that state. The state consisted of the value of the player hand, the value of the dealer hand, and the two ace flags. The agent picked one of the two actions either randomly, or the one that had the highest qvalue for that state. This choice of “explore” or “exploit” was modulated by a user-inputted constant epsilon. When epsilon is 1, the agent chooses between hit and stand randomly, exploring the state space. When epsilon is 0, the agent only chooses the best action at each state, exploiting its knowledge. If epsilon is 0.5, the agent explores half the time and exploits the other half.

Q-Learning
Before training, each state had a qvalue of zero for both actions; the agent did not “know” anything about the game world. The agent gradually learned about the world by iterating through each state thousands of times, steadily updating its knowledge every time a transition from state to state occurred. This function was used to update the mapping:

\[ qvalue = self.getQValue(state, action) + self.alpha * (reward + self.discount * self.getValue(nextState) - self.getQValue(state, action)) \]

GetQValue is a function that returns the qvalue for a specific state action pair. Alpha was a user-inputted constant that controlled the “learning rate,” or how much a transition would affect the knowledge that was already in that state to action mapping. Reward only came into play at terminal states: if the agent won, 1 was returned, rewarding the agent's action. Otherwise, -1 was returned, detersing the agent from making that failed action again. Discount was a constant applied to the qvalues of future states; for example, a hand two actions away from winning was not as valuable as winning hand. GetValue was a function that returned the highest possible value for that specific state. It is used here to get the highest value of the next state in order to propagate values backwards from the terminal states. With enough iteration, q-learning alone works well enough for the agent to play competently, though it requires many iterations. In order to cut down the number of required iterations, I went on to implement feature extraction.

Feature Extraction
Feature extraction works very similarly to q-learning: values are propagated through the agent's knowledge base until it knows enough about the world to make the right decisions. The difference lies in the fact that the feature extraction agent makes inferences about its current value through weighted features within a given state. Because of this, not every state has to be visited during the training phase; the weighted features will properly infer the value of the state based on similar states. Also, the qvalue array is not directly updated in this model. Instead, only the weights are modified. The agent learns the best weights to give to specific features using this function:

\[ weight = self.weights[feature] + self.alpha * ((reward + self.discount * self.getValue(nextState) - self.getQValue(state, action)) * features[feature]) \]

This function is very similar to the q-learning one. The only difference is, instead of using qvalues you use and update weights. The extra multiplication of features[feature] is the actual feature value being multiplied. The features I used were player's proximity to 21 and the average value of played cards as a pseudo card counting method. To be more specific, I averaged the values of all cards played and currently in the player and dealer hands, and if the resulting value was less than 9, the agent was more likely to hit. Otherwise, the agent was more likely to stand. Similarly, if the player's hand was close to 21, the agent was more likely to stand and less likely to hit.

Demonstration
For my project demonstration, I decided to have a user playable game of Blackjack with agent suggestions. To do this, I created and trained the agent, and then polled it for the suggested action at each new player state. It is important to note that once the user starts playing, the agent has no input on the game world, and it no longer learns from it.
Data Set
I used no external data set for this project, as it was simply iterative games of Blackjack. I built the system to generate the data set using the basic rules of Blackjack.

ANALYSIS OF RESULTS
I found both my q-learning and feature extraction agents to be successful. My q-learning agent peaks at 10000 trials with an average 44% success rate. My feature extraction agent peaks at 1000 trials with an average 43% success rate. According to [1], the average success rate for a competent Blackjack player is 44%, so I find my results to be adequate. For both the q-learning and feature extraction agents, my discount was 0.8, my training alpha and epsilon were 0.2 and 0, respectively.

Q-Learning Versus Feature Extraction
My results suggest that feature extraction is the clear winner, but an argument for q-learning could be made. First of all, while q-learning peaks at 10000 trials, it still performs competently at 1000 trials (around 40-41%). Feature extraction, in contrast, only performs well around 1000 trials. At 500 trials or less it’s success rate drops off significantly (below 40%). More importantly, due to the extra computations the feature extraction agent needs to take at each state, q-learning performs much faster. Most significantly, the q-learning agent completes 10000 trials in less time than the feature extraction agent takes to complete 1000.

This is not to say that q-learning is always better than feature extraction, however. The problem is, feature extraction aims to approximate the values q-learning would obtain from a large number of successes and failures, and so it relies on its features to be very thorough. I believe my feature extraction agent did not perform as well as my q-learning agent because my features were not robust enough. Given more time, I could have developed more robust features that better showcased the benefits of feature extraction. Given my constraints and my goal, however, I still find both agents to be successful.

DISCUSSION
This project gave me a much better idea of how to actually implement q-learning and feature extraction, and AI agents in general. Before starting this project, the only experience I had with MDP agents was from a Berkeley problem set [2]. While I was able to use much of what I learned from that problem set for this project, the original problem had everything other than the essential q-learning and feature extraction functions set up already. Starting from the ground up, I was able to see exactly how an agent would interact with a game world, and how to properly construct the game world itself to accommodate for both the agent and the user. For example, in the Berkeley problem set, state was already constructed and supplied to you. For my own project, I not only had to abstract a feasible state from the game world, I also had to figure out how to represent that state within the agent.

For my feature extraction agent, I learned how vital the features were to the agent. I had a lot of initial trouble coming up with adequate features; my first attempts resulted in success rates worse than the agent would get with random chance. It also made my knowledge of how feature extraction worked more concrete. Because of my initial failure with features, I had to reexamine exactly what was occurring during feature extraction and adjust my features accordingly.

CONCLUSION
Despite issues with the feature extraction agent, both the feature extraction agent and q-learning agent were successful. For new work, the features for the feature extraction agent could be improved and expanded upon. Taking the dealer hand into account, and having a more robust card counting system are both likely to be good directions to improve the agent.

ACKNOWLEDGMENT
The work described in this paper was conducted as part of a Fall 2012 Artificial Intelligence course, taught in the Computer Science department of the University of Massachusetts Lowell by Prof. Fred Martin.

REFERENCES