

Blackjack

Due: 6:00 PM 11/24/09



“You cannot beat a roulette table unless you steal money from it.”

—Albert Einstein

Introduction

Expert game playing is one of the claims to fame of modern AI. But the majority of AI’s popular success stories involve deterministic games, like chess and checkers. In this assignment, you will build an intelligent agent that is capable of playing the classic card game of Blackjack, which, unlike chess and checkers, is rife with randomness. You will achieve this goal by formulating the game as a Markov Decision Process (MDP), and solving for an optimal policy. Your `BlackJackPlayer` should even be able to tip the odds in its favor by counting cards.

CS141 Blackjack

Blackjack is card game commonly played in casinos, in which one or more players compete against a select player called the dealer. The object of the game is achieve a point value as close to 21 as possible without going over. The CS141 version of Blackjack is a slightly simplified version of the typical casino game. Each hand proceeds as follows:

1. Each player places his or her bet before the hand begins.
2. The initial cards are dealt to the players:
 - Each player is dealt two cards face up.

- The dealer is dealt one card face up and one face down, the latter of which is revealed to the players later in the hand.
3. The cards have the following point values:
 - Cards of rank 2–10 have point values equal to their rank.
 - Picture cards (Kings, Queens, Jacks) have point values equal to 10.
 - Aces have point values equal to 11, unless the player has a score above 21, in which case Aces take on a value of 1.
 4. If any of the players is dealt a *natural*—two cards that sum to 21—that player need not make any decisions. A natural pays 3:2, unless the dealer is also dealt 21, in which case no money changes hands.
 5. Otherwise, after the initial cards are dealt, the players choose between two different actions:
 - Stay: If a player chooses to Stay, the player’s turn is over.
 - Hit: If a player chooses to hit, the player is dealt another card face up. If the player’s point value exceeds 21 as a result of the newly dealt card, the player *busts* and loses his or her bet. If the player’s point value is less than or equal to 21, the player again has the option of hitting or staying.
 6. After all the players have either gone bust or opted to stay, the dealer reveals his hidden card, and takes the following actions, deterministically:
 - If his score is ≤ 16 , the dealer hits.
 - If his score is ≥ 17 , the dealer stays.
 7. If the dealer busts by drawing cards that lead to a point value greater than 21, all the players who have not themselves busted win money equal to their bets, except players with a natural who win money equal to 150% of their bets.

If the dealer accumulates a point value between 17 and 21, all the players with point values greater than the dealer win money equal to their bets, except players with a natural who win money equal to 150% of their bets.

All the players with scores less than the dealer lose their bets. All the players with scores equal to the dealer *push*, or tie, and don’t win or lose any money.

Card Counting

In theory, we can assume an infinite deck, which implies that the probability of dealing each possible card is always $\frac{1}{52}$. In practice, however, we play blackjack with a finite set of decks (say 10). Under these circumstances, the probability of dealing a certain card depends on the number

of times that card has already been dealt. One can count cards to determine these probabilities exactly, as Raymond Babbitt did in the film *Rain Man*. Or, more simply, one can keep track of (something like) the ratio of high cards (e.g., those with values greater than or equal to 10) to low cards (e.g., those with values less than or equal to 7). If a player keeps track of such a ratio (or a score that reflects that ratio), he can then adjust his strategy to tip the odds in his favor.

What To Do

Your task in this assignment is to formulate blackjack as an MDP and solve it. One effective way of computing an optimal strategy in an MDP is by implementing a Monte-Carlo control algorithm.

1. **Part I:** The first thing that you have to do is to formulate blackjack as an MDP assuming an infinite deck and a fixed bet. This involves deciding what the state space will be, what the available actions are at each state, what rewards are associated with each state (or state-action pair), and what transitions are possible between states. You need not explicitly specify all transition probabilities. *Do this part on paper first please!*

Now that you understand how to represent Blackjack as an MDP, implement the MDP interface, which contains methods for representing the information contained in an MDP. We recommend that your implementation should map `States` to `ActionValueMaps`.

2. **Part II:** Solve the control problem (i.e., compute an optimal policy) for your formulation of Blackjack as an MDP by filling in `BlackjackPlayer`. Your implementation should use either SARSA or Q -Learning to solve this problem. For this part, continue to assume an infinite deck and a fixed bet.
3. **Part III:** Drop the assumption that the deck is infinite. In this setting, your `BlackjackPlayer` should count cards. Do not add exact card counts to the state space of your MDP; that would make your MDP far too large. Instead, you should augment your state space with a score that somehow reflects which cards have been dealt so far. Finally, add an initial betting decision to this enhanced MDP and to your card-counting `BlackjackPlayer`.

Support Code

You can get the source code you need for this project by running `cs141-install blackjack`, which will copy all of the files in `/course/cs141/src/blackjack` to your home directory in `course/cs141/blackjack`.

The support code is configured as an Eclipse workspace. Simply import the `blackjack` directory as an existing Eclipse project to get started.

Javadocs for all of the support code are available at <http://cs.brown.edu/courses/cs141/blackjack/index.html>. Here's a summary of the important classes available to you.

BlackjackPlayer: Represents a player of blackjack. You must implement methods for betting and getting the next action. The methods for reshuffling and processing cards notify your player about events that happen during the game so that it may count cards.

BlackjackMDP: Represents a Markov Decision Process. You should implement either SARSA or Q-Learning to generate a map from states to values. We recommend implementing the **ActionValueMap** class and using it to encapsulate all possible values for the actions at a given state. Your player should then call `getBestAction` to select the best action in the given state.

BlackjackSimulator: Simulates hands of blackjack. This is where the game is played, and this class also contains the main method.

What to Hand In

We need:

1. A README containing a high level description of your solution, and any unresolved bugs.
2. All code necessary to run your `BlackjackPlayer` class.
3. Your written formulation of Blackjack as an MDP. This formulation can include a figure and/or a mathematical description.

Our handin script turns in everything in the current directory and all subdirectories. To hand in this project, navigate into the directory containing the files and directories you wish to hand in and run `cs141-handin blackjack`.

Extra Credit

Easy Extend your Blackjack player and the simulator to allow for *doubling-down*. This means that after looking at only your first two cards, you may double your initial bet and only take one more card, at which point the players turn is over.

Hard Extend your Blackjack player and the simulator to allow for *splitting*. Splitting is an action that can only be taken after the initial cards are dealt, and only when the two cards have the same rank. The cards are split into two hands and the player must place an additional bet equal in value to their original bet. A new card is dealt to accompany each of the split cards and the two hands are played separately.