Creating a Blackjack Artificial Intelligence Using Monte Carlo Search Methods

Author: Connor Graves
Advisor: Dr. Franz Kurfess
CSC 492 Senior Project II

Cal Poly San Luis Obispo
Computer Science and Software Engineering Department

June 10, 2020

Abstract

The aim of this project is to create an intelligent agent that will play Blackjack optimally using a Monte Carlo Tree Search (MCTS) algorithm. This algorithm will be able to simulate thousands of possible outcomes from any given game state in order to recommend to a human player the action most likely to result in a win. Due to the randomness inherent in the game of Blackjack, it is impossible to always make the "correct" play. However, by choosing the move that will statistically lead to the most wins, the AI should be able to perform better than the average human player.

In order to implement the AI, I will have to develop a way to model the game state of Blackjack. I will also have to implement the Monte Carlo Tree Search algorithm itself to simulate games using the game states. A simple adversarial AI will also have to be created to serve as the dealer. Because the rules of how dealers act are very well defined, this will be a simple, but necessary component. Lastly, there will need to be a way of evaluating the deck in order to determine the risk incurred by even starting a hand. This will be based on card counting methods used by human players.

Introduction

Blackjack is the second most popular card game in the world, second only to poker, and can be found in most casinos today [1]. Sometimes called twenty-one, the game is relatively easy to learn but offers higher strategies that can give a player the edge they need to win big. These strategies usually involve counting cards and doing statistical analysis to inform the decisions of the player. Coincidentally, computers are excellent at doing both.

The object of the game is to beat the dealer by getting a card count of as close to 21 as possible without going over, also known as busting. Cards are scored as their number values. Aces can be worth either 1 or 11 points, and face cards are worth 10. The game begins with two cards dealt to each player and the dealer. The dealer begins with his or her second card face-down. Players then may choose to "hit" as many times as they wish, being dealt a card each time until they bust or decide to stay, or end their turn. Then the dealer takes their turn, revealing their card and then hitting until they have a score that beats the player, busts, or is greater than or equal to a threshold (usually 17) set by the house. Once the winners are determined, winnings are paid and a new hand begins [1].

There are multiple examples of Monte Carlo Tree Search being used to simulate games of Blackjack on the internet. However, what I have found generally attempts to play a simplified version of the game. Missing features often include: deciding whether to play a hand, choosing a bet amount, and taking into account doubling down (doubling your bet and taking only one more card) or hand splitting (turning a pair of same-value cards into two separate hands, making an

identical bet on the new hand). In addition, many of these implementations are used only to make generalized strategies. They fail to take into account the state of the deck, which dictates the likelihood of what outcome hitting, staying, or doubling down will bring.

The intended use of this AI will be to augment a Blackjack player's ability to make informed decisions in the game. By tracking the state of the game perfectly and performing on-the-fly simulations, it will be able to recommend what the player should do to increase their odds of winning. Because of the randomness involved, it is not possible to play Blackjack perfectly, but it is possible to play in a way that maximizes your chances.

Background

A Monte Carlo Algorithm is a non-deterministic algorithm that uses randomness to approximate a correct answer to a problem. Monte Carlo Tree Search explores a tree of all possible actions semi-randomly by simulating taking those actions and using feedback from the outcomes to determine the most promising branches. As more simulations are run, the algorithm begins to bias its branch selection towards better-performing branches. Once a sufficient number of simulations are run, the branch with the best performance represents the action to take that has the strongest likelihood of resulting in a win.

For playing a game of Blackjack, the Monte Carlo Search algorithm can be used as it can account for the randomness of drawing cards from a deck. As an adversarial search model, it can also easily simulate both the player and the dealer. Blackjack is also a zero-sum game, as any money lost by the player is gained by the house and vice versa, making it possible to empirically determine how good a game state is for a player. Other players can be ignored, since only one player is taking actions at a time. When creating the tree of game states, all possible card draws will have to be simulated for either a hit or double down, but this can be simplified by making a child action based on card value instead of the card itself (the face cards all have a value of 10, for example).

In terms of performance, MCTS will perform better than many other search algorithms since it is able to combat the high branching factor of Blackjack by sampling possible moves randomly and exploring promising paths more closely [3]. The game state tree of Blackjack, if each node is represented as a unique game state, can have up to 21 children per node. This is the result of being able to hit or double down, both options that result in drawing a card that has one of ten values, and staying. After the first action is taken, this reduces to 11 since doubling down can only happen in the beginning of the hand. In order to more easily map the win percentage to an action so that one can be recommended, the nodes reachable by the same chain of actions are combined. However, this does not reduce the branching factor because nodes must be visited multiple times in order to obtain an accurate picture of its performance. If the game state tree had to be completely explored in order to produce an answer, that algorithm would have a Big-O of

$O(21n^{11})$, where n+1 is the max depth of the tree, or alternatively the length of the longest possible sequence of actions for that hand and deck combination.

Because MCTS can be terminated at any point, we can avoid having to explore the entire tree by giving the best solution known at a predetermined point of termination. This allows for a "good enough" solution to be reached in a reasonable amount of time if exploring the entire tree would take too long. Because of the randomness involved in real Blackjack, a good enough solution is the best that can be done without cheating so accuracy is not strongly impacted, provided enough simulations are run.

Related Work

Below are some overviews of other implementations of Monte Carlo Tree Search used for simulating Blackjack found online:

Towards Data Science: Jeremy Zhang [4]

- Uses Monte Carlo Simulation
- Assumes infinite card deck
- Uses very simple policy (Dealer hit ≤ 17, Player hit ≤ 20)
- Only looks at one isolated hand, not overall strategy

Towards Data Science: Donal Byrne [2]

- Also uses MC simulation
- Only looks at the hand and face-up card. Does not take deck into account
- Determines an optimal policy for a generic hand
- Article notes that the real strategy is to know when and how much to bet - a feature not implemented in this implementation but which I would like to attempt

WikiHow: How to Win at Blackjack [6]

- Designed for use by humans--not an implementation but a strategy
- Has a strategy chart that resembles a policy vector
- Again, acts only as a general guide and does not take into account anything other than your hand and the dealer's face-up card
- This link also describes the rules, payoffs, and other intricacies of the game

Other implementations not listed also seem to be based around simply creating a policy table to be used for every hand and deck that can be encountered. I would like to generate a new policy for each game state separately. Specifically, I believe tracking the state of the deck will improve results for individual hands. This is similar to humans counting cards, except computers can memorize much better than people can. While the difference in the generated policy for each

game state versus a single policy for all game states may be small, it may still mean the difference between a net win or net loss for a player over hundreds or even thousands of hands.

Requirements:

Features

- Able to simulate a game of Blackjack
    - Track hands, face-up cards
    - Track cards in deck and allow for random draw
    - Allow cards to be added to deck (for multi-deck games)
    - Track player money & payouts
    - Calculate win, loss, or draw of each hand
- Able to make a recommendation on whether a hand should be played based on cards left in the deck, and how much to bet.
- Able to make a recommendation on whether to hit, double-down, or stay given a game state (deck, your hand, dealer face-up card) by running Monte Carlo simulations

Requirements

- The software will have some form of knowledge representation data structure in order to describe the state of a game (hands, face-up cards, cards in deck, bet amount, aces in hands)
- The software will be able to generate a new state of a game given a game state, player and a requested action (hit, stay, double-down)
- The software will be able to give a recommendation on whether a game should be played based on deck state and betting budget
- The software will be able to construct a Monte Carlo search tree by simulating games from any given point
- The software will be able to generate a “score” of a given game state given these simulations. Specifically looking at their likelihood of occurring and payoffs.
- After a given number of simulations from a game state, the software will recommend one of three actions to take by the player in order to give the most likely and largest payoff: hit, stay, or double-down

Evaluation Criteria

- The software is able to simulate a game of Blackjack
- The software is able to give a recommendation of whether a hand should be played or not

- The software is able to take a game state consisting of a player hand, dealer face-up card, and deck state as inputs and simulate a game to completion, calculating a score based on likelihood and payout.
- The software is able to run multiple simulations in order to generate a Monte Carlo Search Tree
- The software is able to recommend an action for the player to take based on the generated Monte Carlo Tree
- The recommended actions, if taken, result in earning more winnings over time than following other strategies

System Design & Implementation:

All parts of this project are written in Python for ease of use when dealing with large sets of data. A GitHub repository containing the source code can be found in Appendix IV.

Blackjack Game Simulation

Games of Blackjack have been implemented as a Python object that both track the game state and offer methods to interact with the game. The game state is represented by the values in the below table:

Game Object

Item	Type	Notes
Deck	10 element list	Instead of storing individual cards, store the number of each value remaining in the deck. These are A, 2-10. A single deck would look like [4,4,4,4,4,4,4,4,4,16], where the index = the value - 1, and index 0 is the ace. The value is the number of that value in the deck. Note that there are 16 cards in a standard deck with a value of 10 (10, J, Q, K) This list is updated whenever a card is drawn, or a new deck is shuffled in.
Player Hand	list	A list of values 'A' or 2-10. A value of A will be interpreted as 11 if the sum of all other cards is <= 10 - (num currently uncounted A's). Else A = 1 Initial game state should have **two** cards in hand
Dealer Hand	list	A list of values 'A' or 2-10. A value of A will be interpreted as 11 if the sum of all other cards is <= 10 - (num currently uncounted A's). Else A = 1 Initial game state should have **two** card in hand The second card will be hidden from the player/agent until the dealer's turn begins
Budget	Float	The starting amount of money the player has

Total Won / Lost	Float	The amount of money the player has won or lost from the beginning of play. Will be used as the score that the agent uses to evaluate the “goodness” of a game outcome.
Dealer Stay Value	Int	The value of a hand that will force a dealer to stop hitting, regardless of the state of your hand. The most common value for this to be set at is 17, but rules can vary and so this value must be changeable. A value of 0 will indicate that this rule is not in effect
Turn	String	Defines which entity is acting, or if the game is at an end state. “Player” - The human player’s turn “Dealer” - The automated dealer’s turn “End” - The game has ended

Game Object (cont.)

The Game object contains methods that act as an API, letting other programs perform actions such as drawing cards, scoring hands, and modifying the deck.

A method has also been created that allows a player to play the game via a command-line interface, in order to both help test that the game is represented correctly and to be used in experiments to compare human performance to the AI’s. This method contains simple decision-making capabilities to simulate the dealer. Since the dealer in a game of Blackjack has no discretion in the moves they make (all actions by the dealer are defined by the rules of the game), no simulation or probability analysis is needed to inform a decision like it is for a player.

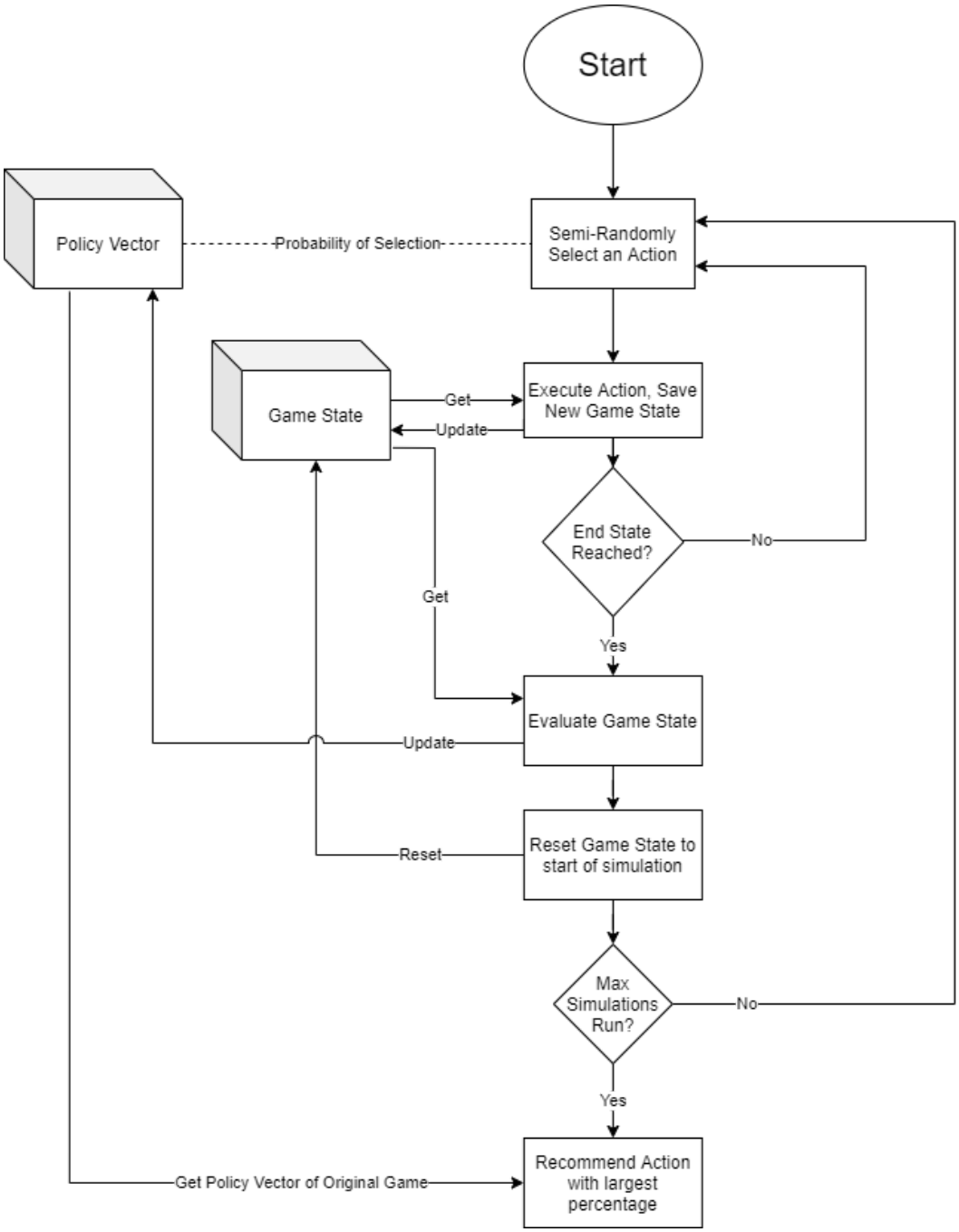
Move Recommender

The move recommender uses the Monte Carlo Tree Search algorithm to simulate games before the player decides on an action. Using the score defined as the amount of money won, the algorithm will explore many possibilities to find the set of actions most likely to beat the dealer. Possible actions will be the available player choices (hit, stay, or double down) paired with the possible cards to draw if applicable.

The algorithm explores all the possible moves from the initial game state and its successor states. This creates a tree data structure. Each node in the tree will have between zero and three child nodes depending on the number of actions that are legal at the time. This seems like a small branching factor, but because of the randomness of drawing cards from decks, each child node must be explored many times in order to create a percentage of wins and losses for each game state. Because of this, for purposes of building the tree games states reached by the same history of actions are treated as identical, even if hands or decks differ. However, evaluating whether a win or loss occurred is done with the individual game states.

Once a game is determined to be completed, the result is determined and all nodes traversed on the path to the final node are updated with the result. These results of each node create a policy vector in the parent node that reflects the viability of each action. Once enough simulations are run, the policy vector of the root node is examined. The action with the highest success rate is recommended to the user.

Move Recommender Flow Diagram



Below are some of the main functions that make up the move recommender, as well as descriptions of them.

reccomend_action():

Given a game object and a string representing previous actions taken, creates a copy of the given game and puts the dealer's card back into the deck, since it is supposed to be unknown. The copy is made so that the "real" game is not affected by the simulation. An action history is created, which maps the record of previous actions taken (stored as a string that can be used to traverse the tree) to metrics objects in order to track the performance of a specific path through the tree.

Simulations are then run, and the win percentages of each child node are retrieved. The action with the highest win percentage is recommended.

```
def reccomend_action(game, actions):
  # create a game that doesn't know the real facedown card
  # the dealer will automatically draw an extra card at the beginning of
  # its turn to compensate
  temp_game = copy.deepcopy(game)
  dealer_facedown = temp_game.d_hand[-1]   # check facedown card
  temp_game.deck[dealer_facedown] += 1     # add facedown back to deck
  temp_game.d_hand = temp_game.d_hand[:-1] # remove facedown card from hand

  action_history = {}

  if actions not in action_history:
    action_history[actions] = Metrics()

  run_simulations(temp_game, actions, action_history)
  possible_actions = get_possible_actions(temp_game, actions)
  values = [action_history[a].get_win_percentage() for a in possible_actions]

  print(action_to_text(str(possible_actions[np.argmax(values)])) +
                       " ({:.1f}% confidence)".format(np.max(values)*100))

  return str(possible_actions[np.argmax(values)])[-2:]
```

Metrics:

The metrics object is used to track the performance of a given node in the game state tree. It contains methods to calculate the upper confidence bound (UCB) and update itself. The UCB is the sum of the win percentage of a node and the exploration term, which is based on the proportion of times the node has been visited by its parent in order to encourage exploring less traveled nodes. The exploration constant, c, used to generate the exploration term determines how much weight the exploration term carries. The square root of two, 1.41, is generally considered to be the most effective value for the constant. The win percentage counts a draw as half a win, in order to incentivize it over a loss. It is better to have your bet returned to you than to lose. In addition, wins and losses are incremented by the score instead of the number of times they occur. This is so that doubling down can be counted double, since it brings double the reward and double the punishment.

```python
class Metrics:
  def __init__(self):
    self.wins = 0
    self.draws = 0
    self.played = 0

  # update with the result of a simulation
  def update(self, result):
    self.played = self.played + max(1, abs(result))

    # Player Loss Condition
    if result < 0:
      pass

    # Draw Condition
    elif result == 0:
      self.draws = self.draws + 1

    # Player Win Condition
    elif result > 0:
      self.wins = self.wins + result

  # get the win percentage of a metric
  # a draw is worth something, but less than a win,
  # so we skew to more draws than losses if needed
  def get_win_percentage(self):
    try:
      return (self.wins + (self.draws / 2)) / self.played
    except ZeroDivisionError:
      return -1

  # calculate a component of the UCB to determine viability of option
  def get_explore_term(self, parent, c = 1.41):
    if self.played > 0:
      #c times the square root of natural log of sims run by parent
      #divided by sims run by this state
      return c * (math.log(parent.played) / self.played) ** .5
    else:
      return 0
```

```
# calculate UCB, or return a very high UCB if never visited before to
# incentivise trying options at least once
def get_upper_confidence_bound(self, parent, default = 6):
  if self.played > 0:
    return self.get_win_percentage() + self.get_explore_term(parent)
  else:
    return default
```

(Metrics continued)

run_simulations():

Given a game object, previous actions, and action history, randomly traverses the game state tree until a leaf (end game state) is reached.. The child node that is selected to be explored is determined by select_action(). A non-leaf node returns null when get_score() is called, while a leaf returns a score. Once a score is returned, the function updates the metrics object of all nodes traversed. This sequence is considered a single simulation.

This is repeated *count* times (default 1000). The resulting metrics object of the original game state will have been updated *count* times, and will be used by reccomend_action() to select an action (hit, stay, or double down) to recommend to the player.

```
def run_simulations(game, actions, action_history, count = 1000):
  if actions not in action_history:
    action_history[actions] = Metrics()

  for i in tqdm(range(count), mininterval = 0.2):
    path = [actions]
    game_path = [game]
    result = get_score(game, actions)

    while result is None:
      child = select_action(game_path[-1], path[-1], action_history)
      if child not in action_history:
        action_history[child] = Metrics()
      path.append(child)
      #last two letters of action string are the next action to take
      actionCode = child[-2:]

      child_game = copy.deepcopy(game_path[-1])
      child_game = make_move(child_game, actionCode)

      game_path.append(child_game)
      result = get_score(game_path[-1], child)

    for a in path:
      action_history[a].update(result)
```

select_action():

Given a game object, previous actions, and action history, generates a policy vector for the given game state by getting the upper confidence bounds (UCB) of each possible child and randomly chooses an action, with probabilities of selecting an action determined by the policy vector. The UCB is a value that represents how promising the child branch is and is based on the win percentage of the branch and an exploration term. The generation of the UCB is done by the metrics object method get_upper_confidence_bound(). A previously unvisited child is given a very high UCB in order to promote exploring new nodes. The higher the UCB, the higher the likelihood that an action is selected by the function. In addition, a global variable named aggression is used to modify the policy vector. This allows the player to nudge the algorithm to select aggressive actions (hitting and doubling down) more or less in comparison to staying. A value of 1 leaves the algorithm unchanged. Greater than 1 yields a slightly more aggressive selection, while less than 1 yields a slightly less aggressive selection. This doesn't affect selection of the final move, but rather encourages exploration of these parts of the tree which may indirectly affect the final selection. The selected action is returned.

```
def select_action(game, actions, action_history):
  possible_actions = get_possible_actions(game, actions)

  # initialize policy vector beginning with equal chance for all possible actions
  policy_vector = np.ones(len(possible_actions)) / len(possible_actions)

  # fill policy vector with all UCBs
  i = 0
  for a in possible_actions:
    # if we haven't done this combination of moves yet, init metrics
    if a not in action_history:
      action_history[a] = Metrics()
    policy_vector[i] = action_history[a].get_upper_confidence_bound(action_history[actions])

    # adjust policy vector by aggression factor
    if a[-2:] == 'Ph' or a[-2:] == 'Pd':
      policy_vector[i] *= aggression
    elif a[-2:] == 'Ps':
      policy_vector[i] /= aggression

    i += 1

  # convert UCBs to selection probabilities
  policy_sum = np.sum(policy_vector)
  if policy_sum != 0:
    policy_vector = policy_vector / policy_sum
  else:
    # choose randomly if you can't win
    policy_vector = np.ones(len(possible_actions)) / len(possible_actions)

  # randomly select an action based on probabilities stored in policy vector
  selected_action = possible_actions[np.random.choice(np.arange(len(possible_actions), dtype=int),
                                                      1,
                                                      p=policy_vector)[0]]
  return selected_action
```

make_move():

Takes an action code and a game state, and returns a new game state where that action has been taken. Action codes are as follows:

- Ph Player Hit
- Pd Player Double Down
- Ps Player Stay
- Dh Dealer Hit
- Ds Dealer Stay

A string of concatenated action codes represents the actions taken to get to a certain game state. For example, a string “PhPhPsDhDs” represents a game state that was reached by a player hitting twice, staying, then a dealer hitting once and staying. The last action taken can be retrieved from the string of actions by slicing it like so: actions[:-2].

```
def make_move(game, actionCode):
  # player hit
  if actionCode == "Ph":
    game.player_draw()
    if game.score_p_hand() > 21:
      game.turn = 'End'

  # player stay
  elif actionCode == "Ps":
    game.turn = "Dealer"

  # player double down
  elif actionCode == "Pd":
    game.player_draw()
    if game.score_p_hand() > 21:
      game.turn = 'End'
    else:
      game.turn = 'Dealer'

  # dealer hit
  elif actionCode == "Dh":
    game.dealer_draw()

  # dealer stay
  elif actionCode == "Ds":
    game.turn = "End"

  else:
    raise ValueError(actionCode + ' is not a valid action.')

  return game
```

Pre-hand Recommendation

By mimicking card counting techniques used by skilled Blackjack players to evaluate the state of the deck, we are able to determine whether the deck favors the player or the dealer. By keeping track of the ratio of 10 value cards and aces to low-value cards played, a player knows that the deck favors them if there are more aces and 10 value cards, and favors the dealer if there are more low-value cards. This is because when there are more high-value cards in the deck, the dealer is more likely to bust and the player will win by default. This ratio can be used both to help determine bet size and to decide whether or not to play a hand before the cards are even dealt.

The recommendation used in this project is a modified Hi-Lo card counting system. Normally in this system, you keep a count for every card played. Cards with values 2-6 add one to the count, and cards with values of 10 and aces subtract one from the count. Other cards do not affect the count. This count is divided by the number of decks in use to create a "true count" that is representative of the makeup of the decks. If the count is positive, the deck favors the player as it has more 10's and aces in it. If it's negative, it favors the dealer.

For this project, it is easier to keep track of the cards remaining in the deck, since we already track it to maintain the game state. Since we are tracking the cards *not* drawn, we can get the same count by inverting the Hi-Lo algorithm. Cards with values 2-6 in the deck subtract 1 from the count, and cards with values of 10 and aces add one to the count. The true count is calculated in the same way. This produces the same deck count without requiring extra tracking.

```
def evaluate_deck(deck, numdecks):
  # higher values favor player, lower values favor house
  value = 0.0
  # subtract 1 for each card with value 2-6
  value -= sum(deck[1:6])
  # add 1 for each card A 10 J Q K
  value += sum(deck[-1:]) + deck[0]
  # divide by number of decks remaining to get a true count that reflects the concentration of high cards
  value = value / float(max(1, numdecks))
  return value
```

With a true count of zero, Blackjack normally has a house edge of 0.5%. However, for each count added above zero, the player gains about half a percent of advantage. At a true count of 1, the odds are even, and at a true count of 2, the player actually has a slight advantage! On the other hand, a negative true count gives the dealer an even stronger advantage, also at about 0.5% per full point.

By evaluating the true count of the deck, the program can make recommendations on whether to play the next hand, and even make betting suggestions.

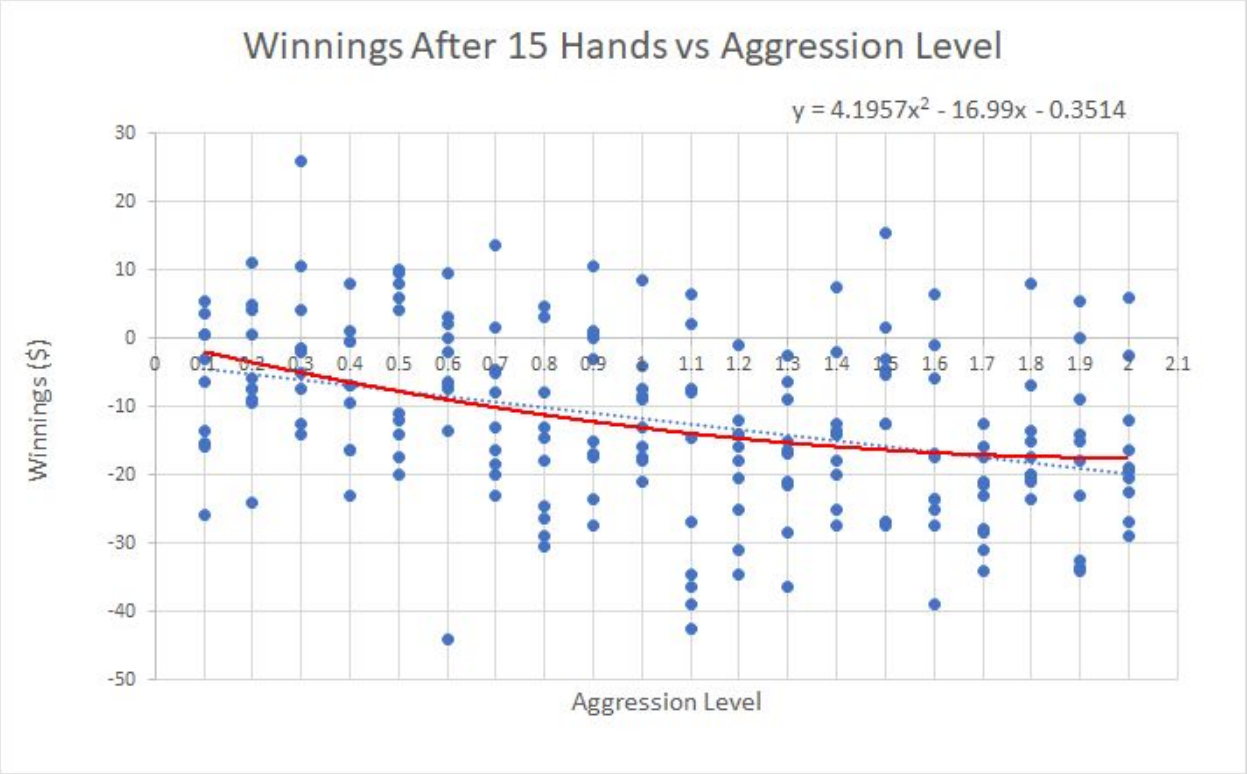
Testing and Validation

Aggression Variable

While initially testing the move recommender, I felt that it was playing slightly too aggressively. The program would recommend so-called "aggressive" actions in situations where it maybe shouldn't have. Because of this, the recommendations would cause the player to bust more often. In order to adjust this, I created a global variable called aggression. Aggression is a value that is used when select_action() is called and adjusts the values in the generated policy vector of a game state. When aggression is greater than one, select_action() is biased towards either hitting or doubling down. When aggression is less than one, select_action() is biased towards selecting the stay action. Leaving aggression set to 1 leaves the algorithm unchanged. While this does not affect the final move recommendation process, what it does do is cause the Monte Carlo Algorithm to explore the aggressive action paths more or less depending on the value of the variable.

To find the relationship between this variable and performance, 10 batches of 15 hands were simulated for each 0.1 interval from 0.1 to 2.0. A relationship was found between lower aggression levels and higher winnings, with a polynomial trendline of

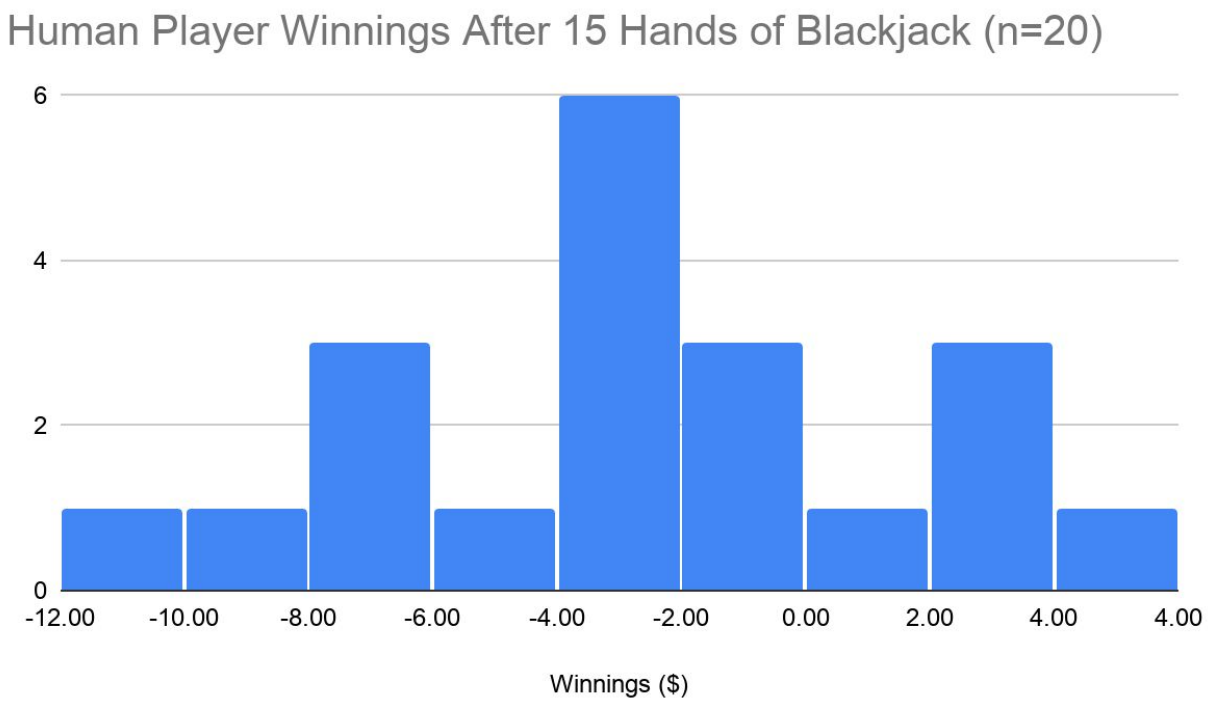
$$Winnings = 4.1957(aggression)^2 - 16.99(aggression) - 0.3514$$



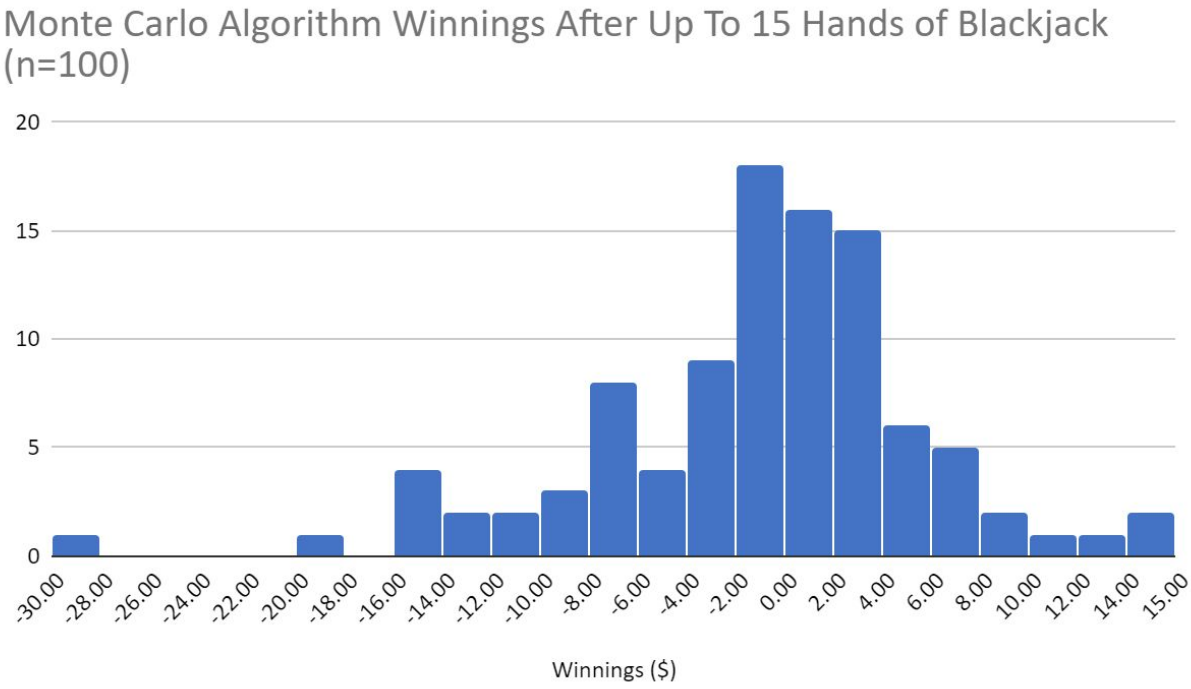
This negative trend shows that the algorithm should be encouraged to explore the “stay” branches more by using small aggression values.

Effectiveness of AI

In order to evaluate the performance of the Monte Carlo Algorithm, fellow students (n = 20) were asked to play 15 hands of Blackjack and report their performance so that it may be compared to the AI. The games used two decks, and betting was locked to $1 per hand, with the exception that doubling down doubles that bet and a “natural”, where the first two cards you are dealt add up to 21, returns 1.5 times the bet. The students, who self-rated their knowledge of the game at an average of 5.4/10, lost an average of $2.90 after all 15 hands. Only five of the students had a net gain of money.



To compare to this, the AI was run for 100 sets of 15 hand games. However, it was also allowed to use the deck evaluation method to slightly modify its bet amounts between $1 and $3, or to quit playing a set of hands early if the deck became too unfavorable. The AI lost an average of $1.24 per set of hands played. An aggression level of 0.3 was used.



While the sample size for the college student trial is smaller than I would've liked it to be, I believe that it is just large enough to show that the AI performs better than the average casual Blackjack player.

Future Work

Apart from further tuning of the aggression value, there are a few more areas of improvement that could result in better recommendations. I think exploring more deck evaluation methods could be beneficial. The method I chose is commonly used by human card counters, but computers are able to track more information reliably than human players, and so more accurate deck evaluation techniques may be able to be developed. Simply running a MCTS simulation before the cards are dealt would not work, as it would only have the contents of the deck to look at. Most of the value from the MCTS algorithm comes from knowing the two cards dealt to the player and the face-up dealer card. That value is lost when you try to evaluate your chances before cards are dealt, which is the point where you would be deciding whether or not to play and how much to bet.

It may also be worth experimenting with setting required thresholds of confidence to recommend actions. For example, would the AI perform better if it required doubling down to have a 1.5 times higher UCB than just hitting? Would it perform better if it would never hit unless the UCB

was above a set value? How would adding these thresholds interact with the aggression variable? Investigating these questions will require large amounts of testing to be done.

Usability could also be improved. Creating a graphical user interface or web plugins for online Blackjack sites could make the program much more seamless and user-friendly. A mobile app could be used in casinos (while this is frowned upon and you may be kicked out, it is not illegal). This exceeds the scope of this project, which was to develop the algorithm, but it is the logical next step once everything is tuned in order to create a user base.

A Blackjack training tool could also be developed. Instead of presenting the user with a recommendation before a move is taken, the algorithm could tell the user how they could've played better after each loss. For example, it could say that you should have stayed once you drew that second card, or that you should have hit one more time. Essentially, it would run it's simulations for each state and tell the user of any deviations from the recommended moves after the conclusion of the hand so that the player can improve their game without becoming reliant on the AI.

An improvement that could be made to the current algorithm concerns the modeling of the dealer. Because Blackjacks (when the original 2 dealt cards add up to 21) end the game immediately, we know that if the dealer has a face-up Ace, their face-down card cannot be a ten. If the dealer has a face-up ten, their face-down card cannot be an Ace. If the face-up card is a ten or Ace, the game would've already ended before simulations were run [5]. If this is taken into account in the simulations, it can increase the accuracy of the predictions made by the algorithm.

Conclusions

The Monte Carlo Tree Search algorithm, when applied to the game of Blackjack, is able to take any given state of the game and recommend the move that maximises the player's chances of winning. While it is not possible to guarantee a win or even winning the majority of the games played due to the random nature of card games, the results of following the recommendations made by the algorithm are generally better than not following them as demonstrated by the experiments done comparing the results of students to the results of the move recommender.

I have previously implemented this algorithm for Tic-Tac-Toe in Dr. Kurfess' Artificial Intelligence course (CSC 480), but implementing it for blackjack was a different experience. In Tic-Tac-Toe, every action taken in that game is deterministic. That is, when a player decides to put an 'X' in the center square, that 'X' will always end up in the center square. The successor game state is always known given a state and an action. In Blackjack, a player may decide to hit, but they can draw any card from the deck. There is not one singular successor game state generated from taking an action (unless the deck only has one card in it). This non-determinism introduces a new problem: visiting a node once is not sufficient to gauge its performance. Nodes

must be visited many times in order to paint a picture of what *might* happen if the player's actions go down that path. This non-determinism is also the main reason behind why it is not possible to be sure whether a recommended action will result in a win or loss.

Being able to explore the intricacies of MCTS and apply it to a real-world problem has been an interesting experience. Knowing this algorithm well opens up many possibilities for its use in the future. So long as it is possible to accurately represent a state of whatever problem I am trying to solve and evaluate it to some form of score once an ending state is reached, it is possible that Monte Carlo Tree Search could be able to solve that problem, given enough time to simulate it.

References

[1] Bicycle Playing Cards. Blackjack – Card Game Rules. Retrieved December 2019 from https://bicyclecards.com/how-to-play/blackjack/

[2] Donal Byrne. 2018. Learning To Win Blackjack With Monte Carlo Methods. (November 2018). Retrieved December 2019 from https://towardsdatascience.com/learning-to-win-blackjack-with-monte-carlo-methods-61c90a52d53e

[3] Franz J. Kurfess. CSC 480: 5-Games.pdf. Retrieved November 2019 from https://drive.google.com/file/d/1zrehgxM-YZYWO_KFjnb-JC3NJc-NG0on/view?usp=sharing

[4] Jeremy Zhang. 2019. Monte Carlo Methods - Estimate Blackjack Policy. (December 2019). Retrieved December 2019 from https://towardsdatascience.com/monte-carlo-methods-estimate-blackjack-policy-fcc89df7f029

[5] Kevin Coltin. 2012. *Optimal strategy for casino blackjack: A Markov chain approach*, Retrieved June 2020 from https://pdfs.semanticscholar.org/d9fc/b037f9ea68862f96124c78c702cb6afe6ea8.pdf

[6] wikiHow. 2020. How to Win at Blackjack. (March 2020). Retrieved May 2020 from https://m.wikihow.com/Win-at-Blackjack

Appendix

I. Schedule
II. Weekly Progress Report
III. Project Proposal
IV. Code Repository: https://github.com/ConnorGraves/BlackjackAI