Report: Blackjack Game Program

Abstract:

My Blackjack Program is a Java implementation of the classic casino card game, Blackjack. This program allows users to play Blackjack with various customizable settings, including number of players, number of decks used, the presence of high-low aces, level of user control, and different AI strategies. The primary goal of this project was to develop our understanding and control of the ArrayList data structure; however, I used several other structures in addition to the ArrayList structure. Two different AI strategies were fully implemented and in simulation neither were able to get the house advantage below 2%, nor were they able to achieve a win rate greater than about 42.6%.

Core CS Concepts Used

Data Structures:

- **HashMap:** The program uses HashMaps to manage decks, player hands, game outputs, data storage, and various other small uses. HashMaps allow for efficient storage and retrieval of the data stored in them.

- **TreeMap:** The program uses the TreeMap data structure to store ArrayLists containing players who achieved the same score. It then uses the built in sorting in the TreeMap structure to determine winners.

- **ArrayList:** ArrayLists are used to store and manage collections of cards, hands, and other data structures throughout the game. They also serve as a wrapper for many other data types when they are being packaged for data extraction and management. Also used as vectors in the unimplemented MachineLearningAI Class.

Algorithms:

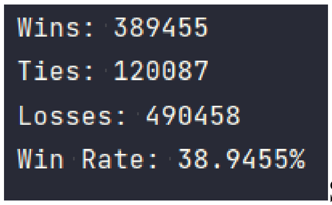
- **Decision-Making Algorithms:** The program employs decision-making algorithms for player and dealer actions. These algorithms determine when to hit or stand based on the current state of the game. Three different strategies were conceived; however, only two were fully implemented: simpleAI, and advancedAI.

- **Card Counting (Advanced AI):** The advanced AI strategy incorporates the Omega-II card counting algorithm to make decisions. This algorithm assigns values to cards and adjusts the player's strategy based on the count, an attempt at a more sophisticated AI approach which unfortunately fell short of expectations.

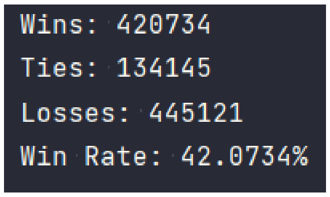
Results:

The program successfully simulated Blackjack games based on user input settings. Similarly, the Simulation Class was successful in simulating games for up to as many as one million iterations. It provides the following results:

- **Win Percentage:** The program calculates and displays the percentages of wins for the dealer, players, and ties over multiple game sessions. The win percentage across one million games quite consistently fell within 0.7% of 38.9% using the advancedAI. Sadly, this was a worse performance than simpleAI with a win percentage across one million games within 0.5% of 42.0%.

  Advanced: 

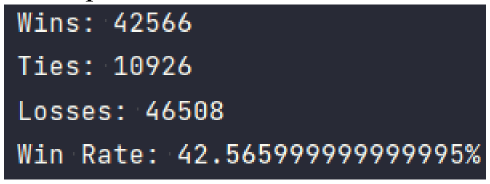
```
Wins: 389455
Ties: 120087
Losses: 490458
Win Rate: 38.9455%
```

  Simple: 

```
Wins: 420734
Ties: 134145
Losses: 445121
Win Rate: 42.0734%
```

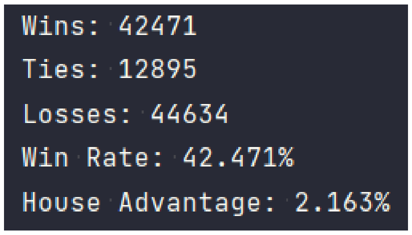
  The simpleAI's comparatively better performance compared to the more advancedAI could likely be adjusted by increasing therisk tolerance on the advancedAI when dealing with a "soft" hand, a hand where an ace can be read as either a 1 or 11. Similarly tweaking the weights on the linear equation might help remedy this disparity.

  Update: Having tweaked the weights in the advancedAI method and having implemented a "soft" hand handler, I was able to bump the advancedAI method's performance up to consistently within 0.25% of 42.2%. While not a tremendous increase this at the very least indicates an increase over the simpleAI method. However, these tweaks had an unintended consequence covered next/



```
Wins: 42566
Ties: 10926
Losses: 46508
Win Rate: 42.565999999999995%
```

- **House Advantage:** The tweaks made to the advancedAI while bringing the win rate slightly above that of the simpleAI were not able to bring its results in line with the house advantage exhibited by the simpleAI. However, after further tweaking, minimizing house advantage while making efforts not to cut into the gains provided by the previous optimization, we were able to reach a house advantage consistently within 0.5% of 2.5%, all while maintaining a win rate within 0.3% of 42.2%.

Tweaked advancedAI: 

```
Wins: 42471
Ties: 12895
Losses: 44634
Win Rate: 42.471%
House Advantage: 2.163%
```

<u>Discussion</u>:

<u>Fairness of the Game</u>

As expected, blackjack, like all other casino games, is an inherently unfair game. While neither of my AI strategies were particularly stellar, they both performed far above how the average person plays, yet neither of these AIs were able to surmount the house advantage. The only way I could imagine beating the house would be the card counting MachineLearningAI which I ran out of time to implement.

<u>Extensions</u>:

<u>Multiplayer Play</u>

I created a multiplayer feature wherein the user can input any integer quantity of players. Then depending on the users input for interactive state and npcAI state these players can either be controlled by one of the AIs or by the player themselves. The number of players also determines the default reshuffle cutoff value for the game.

<u>Multi-Deck Play</u>

I created a multi-deck feature wherein the user can input any integer quantity of decks. The deck creator will then create a multideck containing however many decks the player requested. Larger decks allow for more players and make card counting more difficult.

<u>High-Low Mode</u>

I implemented a rule set and various other features that allow the player to select whether they'd like to play with high or high-low aces. This is incorporated into all functions and readouts such that regardless of whether you play with high or high-low aces everything should function properly.

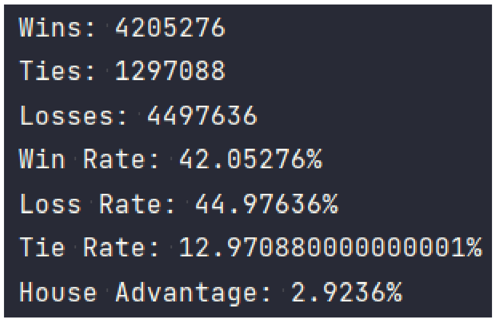
<u>Interactive Gameplay</u>

I implemented a feature for user interaction. If you run the Blackjack Class directly the main method takes a bunch of user input to create a custom game. Enabling the interactive game feature allows the player to control either just player 1, or if npcAI is disable, all players except the dealer. This is handled via the function getCardDecision().

Advanced NPC Strategy

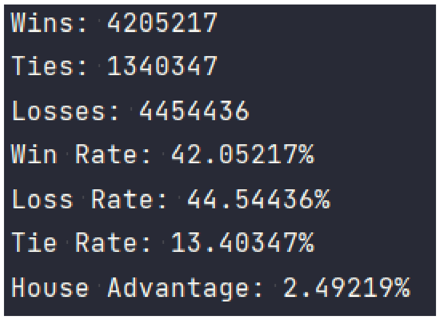
I created two different AI strategies for the NPCs. SimpleAI is an implementation of basic blackjack strategy, whereas the advancedAI implementation implements an attempt at incorporating the Omega-II card counting algorithm. Omega-II is a balanced algorithm as such, after counting every card in the deck the true count will be zero. The algorithm gets all the cards visible and seen by the player and assigns each of these cards a value of either -2, -1, 0, 1, or 2 which is added to the running count. The running count is then divided by the number of decks remaining to determine the true count. Decks remaining are determined by dividing the remaining cards by 52. This value and the current hand value are both sent to a linear function which determines the betModifier, which, if less than 0.5, will make the decision false. The advancedAI performed very slightly better than the simpleAI. I was in the process of implementing a third AI implementing some form of a reinforcement learning algorithm I came up with; however, I ran out of time.

Result of 10,000,000 rounds of Blackjack, advancedAI:



```
Wins: 4205276
Ties: 1297088
Losses: 4497636
Win Rate: 42.05276%
Loss Rate: 44.97636%
Tie Rate: 12.970880000000001%
House Advantage: 2.9236%
```

Result of 10,000,000 rounds of Blackjack, simpleAI:



```
Wins: 4205217
Ties: 1340347
Losses: 4454436
Win Rate: 42.05217%
Loss Rate: 44.54436%
Tie Rate: 13.40347%
House Advantage: 2.49219%
```