UNIVERSITY OF REGINA

ENSE 480: KNOWLEDGE BASE AND INFORMATIONS SYSTEMS

SOFTWARE SYSTEMS ENGINEERING

AI Project: Blackjack Bot

AUTHOR

Konstantin Kharitonov

200354502

SUPERVISOR

Christine Chan

Ph.D., P.Eng.

April 12, 2019

Contents

List of Figures

1 Introduction

Blackjack is a popular gambling card game that has been inside casinos since around the late 1500s. It involves up to 7 players at a table facing off a dealer. At the top of a round, each player and the dealer recieves two cards in their hand, and the dealer having to place their first card faced down on the table and their second card faced up. The objective of the game is to win by either getting a score of 21 from the first two cards, achieve a combined high score higher score at the end of the game without going over 21, or have the dealer's score be higher than 21. At the top of the game each player will submit their bets and recieves their cards. A player will continue recieve cards (hit) until they are satisfied with the score of their hand (stand) or they go over 22 and bust, which results in a loss. Once all players have completed their turns, the dealer reveals the face down card and will automatically hit if the hand has a score until it reaches 17 (based on most casino rules) or the dealer busts.

As with most casino games, the very nature of the game is set up so that the dealer, which represents the casino, wins the majority of the time. Different strategies have been developed for the game, such as using probability analysis, can help a player improve, but ultimately it not lead to the player winning over the dealer. Other strategies such as counting cards have proven to be successful, but once these methods were figured out, casinos quickly outlawed them and were aggressive in enforcing the ban.

As such, the purpose of this AI bot was to find if a player can still be successful enough at the game of blackjack but still be seen as a legal player. Can a computer artificial intelligence on a consistent, use legitimate and legal blackjack strategies, start from a pot of $200 and win its way to a pot of $1000?

2 Representation and Data Structures

As currently designed, the application first creates an array of 52 cards, representing each card in the deck. Every card is created as its own object, with its own suit and value. Once created, a deck creates an array of cards, populating them with a suit and a value. When requested, a card will be displayed in the form of XofY, with a X being the card value and Y being the suit. For example, a jack of hearts will be shown as JofH. The following image showcases how the cards are stored into a deck.
This method was chosen since every single card has two distinct attributes associated with, and there is a finite amount of cards in a deck. Each time a card is drawn from the deck, it can be removed from the deck array and then used in a hand. As well, using a multidimensional allows for tracking cards that have the same value, but with a different suit. For example, there are 4 of king cards in a standard deck, but there are only one king per suit. It also allowed for a card to be stored in the XofY format as a string, which then gets added to a vector array for the player's hand and the dealers hand. Using vector arrays were an easier

```
Deck::Deck(){
  string values[] = {"2","3","4","5","6","7","8","9","10","J","Q","K","A"};
  char suits[] = {'C', 'H', 'S', 'D'};
  deck = new Card[cardsInDeck];
  current = 0;

  for(int i = 0; i < cardsInDeck; i++){
    deck[i] = Card(values[i % 13], suits[i/13]);
  }
}
```

Figure 1: Creation of a deck

choice than static arrays, as the built in functionality allowed for easier parsing through each element in the array, specifically push_ back() and size() functions. Each hand array will only be as large as there are cards inside the hand. For example, if the AI player only hits once and has three cards, the array will only need to be 3 elements big for that particular round. Once all cards drawn are inside the hand array, each element can easily be evaluated to see how much a card is worth.

3 Techniques and Algorithms

There are two functions which control the Blackjack Bot and its decision making, hitOrStand and runSimulation, with playHand being the function that runs the rules for a round of Blackjack.

Function Name: playHand
Input Parameters: deck (Deck object), player1Win(& bool), player1Lose (& bool), player1Push (& bool), successCount (& int), failCount (& int)
Output Parameters: None
Function Requirements: Function for calculating each hand, check for a bust
Description:

1. Deals two cards for the player and two cards for the dealer.
2. Calculates the score of each hand, displays the player's score, the player's hand and one card of the dealer's hand.
3. call hitOrStand and wait for the Bot to respond with either a hit or a stand.
4. If there is a hit, then a card will be added to the player's hand, calculates to the hand's score and sees if the player busts
5. If there is a stand, then a the dealer will reveal second card, hit until score is over 16.
6. checks to see if player won, lost, or pushed (tie).

```
void playHand(Deck deck, bool &player1Win, bool &player1Lose, bool &player1Push, int &successCount, int &failCount){
  Card card1, card2, newCard, dealerCard1, dealerCard2, dealerNewCard;
  vector <string> yourHand, dealerHand;
  char ans;
  int score = 0;
  int dealerScore = 0;
  int exit = 0;

  //Player Cards
  card1 = deck.dealCard();
  card2 = deck.dealCard();
  yourHand.push_back(card1.print());
  yourHand.push_back(card2.print());
  //Dealer Cards
  dealerCard1 = deck.dealCard();
  dealerCard2 = deck.dealCard();
  dealerHand.push_back(dealerCard1.print());
  dealerHand.push_back(dealerCard2.print());
  //This section calculates your score at the top of the game
  score = calculateHand(yourHand);
  cout << "Jim's Hand is " << yourHand[0] << " and " << yourHand[1] << endl << "Jim's score is " << score << endl;
  //This section shows the dealers hand
  dealerScore = calculateHand(dealerHand);
  cout << "The dealer has " << dealerHand[1] << " showing." << endl;
  //Loop for until hand is done
  while(exit == 0){
    ans = hitOrStand(yourHand, dealerHand, score, successCount, failCount);
    cout << endl;
    //if a hit is made
    if(ans == 'h'){
      cout << "Hitting" << endl;
      newCard = deck.dealCard();
      yourHand.push_back(newCard.print());
      cout << "Jim's Hand is ";
      for(int i = 0; i < yourHand.size(); i++){
        cout << yourHand[i] << " ";
      }
      cout << endl;
      score = calculateHand(yourHand);
      cout << "Jim's Score is " << score << endl;
      //Check for Bust
      if(bust(score) == true){
        cout << "Jim Busts!" << endl << endl;
        player1Lose = true;
        exit = 1;
      }
    }
```

Figure 2: Part 1 of playHand

```
    //if standing
    else if(ans == 's'){
      cout << "Standing" << endl;
      cout << "Dealer's original hand" << endl << dealerHand[0] << " " << dealerHand[1] << endl << endl << "Dealer's score is " << dealerScore << endl;
      //Dealer check
      while(dealerScore < 16){
        cout << "dealer hits!" << endl;
        dealerNewCard = deck.dealCard();
        dealerHand.push_back(dealerNewCard.print());
        for (int i = 0; i < dealerHand.size(); i++){
          cout << dealerHand[i] << " ";
        }
        cout << endl;
        dealerScore = calculateHand(dealerHand);
        cout << "Dealer's Score " << dealerScore << endl;
      }
      cout << endl;
      if(bust(dealerScore) == true){
        cout << "Dealer busts!" << endl;
        cout << "Jim Wins!" << endl << endl;
        player1Win = true;
        exit = 1;
      }
      else if(dealerScore > score){
        cout << "Jim Lost!" << endl << endl;
        player1Lose = true;
        exit = 1;
      }

      else if(score > dealerScore){
        cout << "Jim Wins!" << endl << endl;
        player1Win = true;
        exit = 1;
      }
      else if(score = dealerScore){
        cout << "Jim Pushes!" << endl << endl;
        player1Push = true;
        exit = 1;
      }
      else{exit = 1;}
    }
    else{ cout << "Oops!" << endl; exit = 1;}
  }
}
```

Figure 3: Part 2 of playHand

Function Name: hitOrStand

Input Parameters: currentHand (vector<string>), dealerHand(vector<string>), score (int), successCount (int), failCount(int)

Output Parameters: move (char)

Function Requirements: Need to know the score of the dealer's face up card

Description:

1. checks to see value of players score
2. will determine whether or not to send an h (hit) or s (stands), depending on the score of the player's hand, score of the dealer's hand, amount of cards inside the Bot's hand, and amount of wins in a particular game.
3. returns the Bot's decision to playHand

```
char hitOrStand(vector<string> currentHand, vector<string> dealerHand, int score, int successCount, int failCount){
  char move;
  float chances;
  int dealerCardScore = getCardValue(dealerHand[1]);

  //Jim's hand is 21 or 20
  if(score == 21 || score == 20){
    move = 's';
  }

  //Jim's is 19 or 18 or 17
  else if((score == 19 || score == 18 || score == 17)){
    //Dealer's show card is less than 5
    if(dealerCardScore < 5){
      //Currently on a streak where there more than double wins over losses
      if(successCount > (2 * failCount)){
        move = 'h';
      }
      else{
        move = 's';
      }
    }
    //dear has a 10 or 9, only two cards in player's hand
    else if((dealerCardScore == 10 || dealerCardScore == 9) && currentHand.size() == 2){
      move = 'h';
    }
    else{
      move = 's';
    }
  }

  //Jim's hand is between 10 and 16
  else if((score >= 10 && score <= 16)){
    //Dealers best chance of victory, most likely to lose, best to go for it
    if(dealerCardScore == 10 || dealerCardScore == 9){
      move = 'h';
    }
    //Dealer's best hand without hitting is 19,
    //worse hand without hitting is 16
    else if(dealerCardScore == 8){
      //no chance of busting with a hand score of 10 or 11
      if(score == 10 || score == 11){
        move = 'h';
      }
```

Figure 4: Part 1 of hitOrStand

```
    //Chance of busting on a 10, J, Q, K, 30% chance to bust
    else if(score == 12 || score == 13){
      //Bigger chance to get a card that will bust
      if(currentHand.size() == 5){
        move = 's';
      }
      else{
        move = 'h';
      }
    }
    else if(score == 14 || score == 15){
      if(currentHand.size() == 4){
        move = 's';
      }
      else{
        move = 'h';
      }
    }

    else if(score == 16){
      if(currentHand.size() == 4){
        move = 's';
      }
      else{
        move = 'h';
      }
    }
  }
  //Dealer's best hand without hitting is 18,
  //worse hand without hitting is 16
  else if(dealerCardScore == 7){
    //no chance of busting with a hand score of 10 or 11
    if(score == 10 || score == 11){
      move = 'h';
    }
    //Chance of busting on a 10, J, Q, K, 30% chance to bust
    else if(score == 12 || score == 13){
      //Bigger chance to get a card that will bust
      if(currentHand.size() == 5){
        move = 's';
      }
      else{
        move = 'h';
      }
    }
```

Figure 5: Part 2 of hitOrStand

```
      else if(score == 14 || score == 15){
        if(currentHand.size() == 4){
          move = 's';
        }
        else{
          move = 'h';
        }
      }
      else if(score == 16){
        move = 's';
      }
    }
    //Dealer's best hand without hitting is 17,
    //worse hand without hitting is 16
    else if(dealerCardScore == 6){
      //no chance of busting with a hand score of 10 or 11
      if(score == 10 || score == 11){
        move = 'h';
      }
      //Chance of busting on a 10, J, Q, K, 30% chance to bust
      else if(score == 12 || score == 13 || score == 14){
        //Bigger chance to get a card that will bust
        if(currentHand.size() == 5){
          move = 's';
        }
        else{
          move = 'h';
        }
      }
      else if(score == 15 || score == 16){
        move = 's';
      }
    }
    //Dealer's hand cannot stand on two cards alone
    else if(dealerCardScore <= 5){
      if(score == 16){
        if(currentHand.size() == 5){
          move = 's';
        }
        else{
          move = 'h';
        }
      }
      else{
        move = 'h';
      }
    }
```

Figure 6: Part 3 of hitOrStand

```
  //Jim's hand is less than 10, no chance of busting
  else if(score < 10){
    move = 'h';
  }

  return move;
}
```

Figure 7: Part 4 of hitOrStand

Function Name: runSimulation

Input Parameters: numOfTurns (& int), successCount (& int), failCount (& int), gameCount (& int) successArray(& int)

Output Parameters: None

Function Requirements: starting player pot of $ 200

Description:

1. Starts off by choosing the type of bet based on the previous rounds.
2. Calls the playHand function to find the result of the hand.
3. If the Bot wins, increase the pot by the bet and increment the streak counter.
4. If the Bot loses, decrease the pot by the bet and set the streak counter to 0.
5. Increment the number of rounds counter.
6. If the Bot has reached a pot that is greater than or equal to $1000, then the particular simulation is successful and is stored into the success array.
7. If the Bot has a pot of $0, then the simulation ends with an increment of the fail count.
8. Simulation continues on until one of these two end states occur.

```
void runSimulation(int &numOfTurns, int &successCount, int &failCount, int &gameCount, vector <int> &successArray){
  Deck deck;
  bool player1Win = false;
  bool player1Lose = false;
  bool player1Push = false;
  double player1Bet = 0;
  double player1Pot = 200;
  int turnCount = 1;
  int streak = 0;
  bool play = true;

  while (play == true){
    deck.shuffle();
    if(turnCount == 1){
      cout << "This is game " << gameCount << endl;
    }
    cout << "Jim pot is $" << player1Pot << endl;
    cout << "This is hand " << turnCount << endl;

    if(turnCount == 1){
      player1Bet = 100;
    }

    else if(player1Win == true){
      if(streak > 2){
        if(streak > 4){
          if(streak > 6){
            player1Bet = 500;
          }
          else{
            player1Bet = 200;
          }
        }
        else{
          player1Bet = 150;
        }
      }
      else{
        player1Bet = 100;
      }
    }
    else if(player1Push == true){
      player1Bet = 100;
    }
    else{
      player1Bet = 50;
    }
```

Figure 8: Part 1 of runSimulation

```
        cout << "Current Bet is $" << player1Bet << endl;

        player1Win  = false;
        player1Lose = false;
        player1Push = false;

        playHand(deck, player1Win, player1Lose, player1Push, successCount, failCount);
        if(player1Win == true){
          player1Pot += player1Bet;
          streak += 1;
        }
        else if(player1Lose == true){
          player1Pot -= player1Bet;
          streak = 0;
        }
        else {
          cout << "No change in pot." << endl;
        }

        turnCount += 1;
        if(player1Pot >= 1000){
          cout << "Jim got to $1000! Jim did it!" << endl;
          cout << "Jim's total pot is $" << player1Pot << endl;
          successCount += 1;
          numOfTurns = turnCount;
          successArray.push_back(gameCount);
          gameCount += 1;
          play = false;
        }
        if(player1Pot <= 0){
          cout << "Jim lost all of his money!" << endl;
          numOfTurns = turnCount;
          failCount += 1;
          gameCount += 1;
          play = false;
        }
      }
      cout << endl;
      cout << "Thanks for playing!" << endl << endl;
    }
```

Figure 9: Part 2 of runSimulation

4 Structural Diagram for Major Modules

For this program, the three functions act as their own modules of the entire project. In the first module, the user selects the amount of times the simulation will run. As such, the system will first chose the amount of money it will bet on the first hand in 'runSimulation', then call the 'playHand' module to start a round by drawing cards for both the player and the dealer. The player will then either hit or stand depending on what their hand is. Once the player stands, then the dealer will show their face down card, continue to hit until the score is above 16 and will then end it turn. Depending on who has the final score, the system will mark either a player win or a player loss. A player bust will automatically end in a player loss and a dealer bust will automatically end in a player win. Then the system is run again for the top of the round depending on how long it is set to run for.

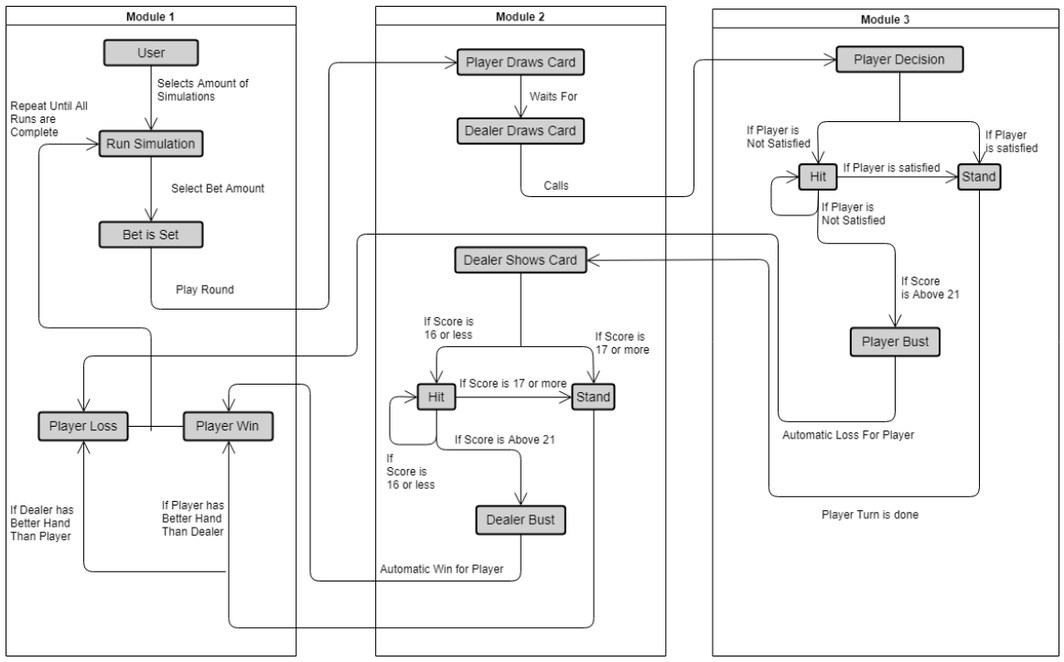

Figure 10: The Blackjack Bot Module Diagram

There are different paths that the Bot may take during runtime, but there are only two different end states before the system is looped to the beginning.

5 User Manual

The Blackjack bot is compiled using C++, so an executable is created. To run the program, one must simply double click on the file and the Bot will start up. If a user wants to execute the code from scratch, the user must have a C++ compiler on their system or use an IDE with a C++ compiler installed. It is recommended to use a g++, as that is what I used for creating this project. If compiling from a command line, please use the following commands.

For a Windows System using MinGW or similar compiler:

1. g++ -c main.cpp Hand.cpp Card.cpp Deck.cpp
2. g++ main.o Hand.o Card.o Deck.o -o Bot.exe
3. Bot.exe

For a Linux System:

1. g++ -c main.cpp Hand.cpp Card.cpp Deck.cpp
2. g++ main.o Hand.o Card.o Deck.o -o Bot
3. ./Bot

6 Sample Session

Once the program is loaded up, a prompt will appear asking how many simulations the user wants to run, between 1 and 1000.

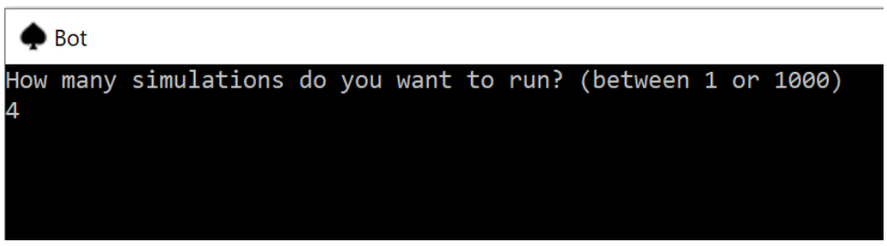

Figure 11: Prompt at the start

At this point, after inputs a number, the simulation will run for as many amount as times specified, and see if the Bot was successful in its goal of reaching the $1000 threshold.

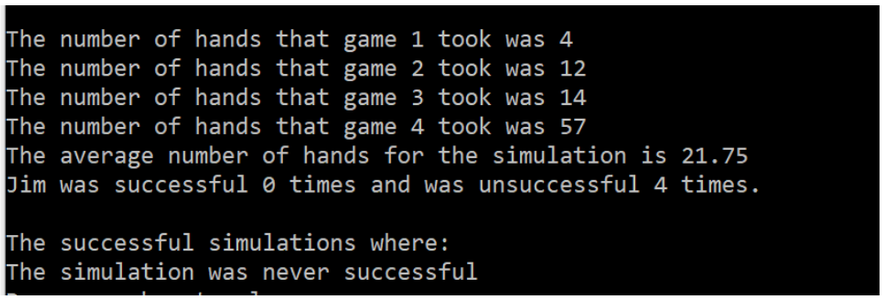

Figure 12: Stats of the simulation

Figure 12 showcases the end result of a run. The program will showcase the number of hands there were in a particular simulation, the average amount of hands in the simulation and how many times the Bot was successful in achieving the $1000 and how many times the Bot lost all of its starting money.

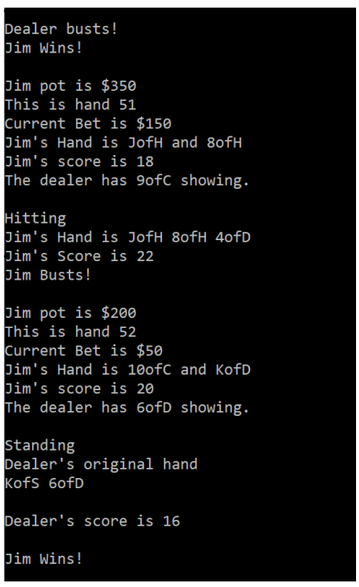

Figure 13: A particular Hand

Figure 13 showcases how a particular hand will go. It is shown how the previous hand effects the type of bet that the Bot is going to make. It also showcases the type of moves

that the Bot will make depending on the current status of their success and the status of the game.

7 Discussion

After the implementation of the Blackjack Bot, it was not able to consistently win, even with the help of evaluation of dealer's hands, probability analysis of what the next card in a deck might be, the system was still only winning well below what was expected. On a run of 1000 simulations, the Blackjack bot was only able to achieve its goal of getting to $1000 starting from an inital pot of $200 only 95 times. That leaves the Blackjack Bot with roughly a 10% chance for the bot to have a success simulation. It begs the question, why is it so unsuccessful?

The first potential for the futility of the limitations of the kind of cases that the system does not currently have implemented. It is always possible to expand on the currently algorithm and add more specific cases to the system that are currently not there.

Due to time contraints, not all blackjack moves are currently implemented. The two biggest rules would be the ability to split a hand as well as the double down rule. To split a hand, the player must have a hand where they have two of the same card values in one hand. When this happens, the player essentially has two hands where the player can bet on, allowing the player to double their money on a bet. However, this also means that if the player loses both hands, they lose twice the money. Another such feature is doubling down, where the player can double their bet and request only one card, forced to stand after the card is given. Another way that could increase the amount of money that the bot can make on a particular good hand, but can also be detrimental if not done properly.

However, the ultimate reason as to the unsuccessfulness of the system is the very nature of the game of blackjack itself. The way it is currently designed, the odds of winning are always heavily swayed in the favour of the dealer. A player has to make a guess as to what the dealer might have in a certain hand as well as make a guess on what hand might come during a hit. As well, since it is more likely that the dealer will in a particular round of blackjack, the odds of winning enough rounds that a player is able to quintuple their money playing the game are extremely low compared to the chances of bottoming out. Without cheating, it is just not very likely a person is able to accomplish such a large goal. When the score was halved to only be at $500, the Blackjack Bot was much more successful, being able to reach that goal 55% of the time, more than half. There are some methods, such as counting cards that can result in the player winning more than the dealer, but those methods are deemed illegal in casinos, defeating the purpose of the Blackjack bot at being one that simulates a real life scenario. As such, it is difficult for the bot to be lucky consistently so many times, and it may be limited because of the very nature of the game it was designed to play.

8 Conclusion and Future Features

The Blackjack Bot is a humble AI, resulting in a small scope. Since it intended to aid a person with their blackjack playing skills, it was never intended on being a completely revolutionary AI system. The methods of developing an AI that searches for the best possible outcome in a probability situation allows for the system to be as complex or as simple as need be. This AI was the first attempt at implementing a depth-first searching AI system, as it attempts to find a possible solution through many different potential nodes that can be processed through.

In terms of future work, the main objective would be to implement more refined rules to better enhance the playing ability of the Bot. While the current iteration is able to play, the system can still use some definite improvement such that it is able to win more games in the future. As well, the two gameplay features of doubling down and splitting are ones that I want to have in the game for the next release as they are rudimentary rules of the game and by having then in my system does make the Bot much more realistic. I also do want to implement a counting cards system. While yes it is an illegal play, it is still a legitimate strategy that has proven to be successful and showcasing it may help develop strategies that are effective but also will not get a player kicked out of a casino.

9 References

Cardoso, Karla R., et al. "Extracting Rules for Black Jack Using Machine Learning and Fuzzy Systems." *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2018, doi:10.1109/fuzz-ieee.2018.8491688.

Moon, H., Yang, T., Kim, S., & Park, J. (2017). Comparison of Gambling Industry through Study Analysis in the World. *2017 IEEE International Congress on Big Data (BigData Congress)*. doi:10.1109/bigdatacongress.2017.81

Fogel, D.b. "Evolving Strategies in Blackjack." *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753)*, doi:10.1109/cec.2004.1331064.

Sir Joseph the Pladin. (2013, September 6). *C++ Programming 49 - Deck of Cards*[Video File].Retrieved from https://www.youtube.com/watch?v=NAAEMILMt-Q