CIS-7 Course Project Write-Up

December 18th, 2025

Delano Leslie

Introduction

This is document contains the requested documentation as part of this course project. It contains information on what problems we solve in this project, the solutions to said problems, explanations of calculations / algorithms, program objectives, user usage / purpose, use of discrete structures, program limitations, and program improvement recommendations / further improvements.

It also contains pseudo code for the entire program's control flow.

Objective

For this project, several problems were proposed to students. This project in particular choose the BlackJack Problem as outlined in the reference documentation.

To summarize the problem briefly, we were required to recreate the famous game BlackJack in a C++ program. This program needed to show the hand of the dealer and calculate, given the current state of the game, the probability of the player wining.

Problems

Of course, whilst this objective may be short in entails a number of challenges to over come. To start, you need to actually recreate the game in a way that allows performing wining probability calculations. This means as a team, you needed to implement the game logic, implement user interaction, and do this all in a way that makes the code legible for others.

Program Structure(s)

Before writing any program, understanding how your program is going to store and represent the current state it is (state), is probably the most fundamental problem any programmer had to tackle. In this project in particular, you need to store the game state. At first glance this may seem like the only thing you would need to represent, but, in the integration of the game state, you will find your self needing to represent much more.

For starters, we need to actually represent the cards in our card game. We then need to represent our deck of cards. We also need a way to store what cards the player and dealer have in their "hand". After which, then and only then, can we actually start storing game state (things like current player money, current turn, etc.).

Control Flow

Of course being able to represent a singular game state is great, but it serves no purpose if we don't give our program some form of behavior. Choosing how we change and modify our programs state is what we call control flow. Every program requires a form of control flow, and different program objectives naturally lead to different control flow structures. For example, in game engines, there are many different control flow structures used in industry. A famous example is ECS (Entity, Component, Systems). Choosing a proper way to manage control flow can radically change the complexity and legibility of a program. As a result, making a correct choice can be a difficult problem to over come. In regards to this project, making a proper choice is one of the main hurdles to overcome.

User Interaction

After all is said and done, we of course need some way for a use to interact with our program. Ideally this interaction is straight forward and easy to understand. We need to inform the user of the current program state, and show them which ways they can interact with the program. This is formally known as **UX**, or **User Experience**. For this program, teams will need to make a **UI** (User Interface), and interaction system.

Calculations

Finally, the main discrete structures oriented problem we need to solve is performing wining probability calculations. For this project, you need to manage all of the aforementioned game state, game logic, and UX in such a way that allows these calculations to be made and presented. Assuming you made proper choices on the previous problems, this actually shouldn't be too difficult and relies more on your knowledge of the math behind the calculations.

Project Solutions

To solve the problems outlined in the **Introduction** for our project, we employ many common **C++** and **Discrete Structures** ideas.

Game State

As mentioned, we need to represent cards, the deck, player / dealer hands, and the current game state. We will demonstrate how we solved each problem in the following sections.

Card

This was the first structure we needed to represent in our game, seeing as it is a card game. To do this, internally we use a **C++ enumeration**. This enumeration is akin to that of a set which was discussed in the **CIS-7 Discrete Structures** class. By using an enum to represent the card's rank and suit. It makes it so representing any non possible card (i.e. 11 of spades) is no possible. This, of course though, means we needed to implement methods to represent the card in a way that the user could understand (to "translate" the card state to a human readable format) In our code, we create a struct called `Card` which has two enums `Rank` and `Suit`. The `Rank` and `Suit` enums are just long, human readable forms of essentially integers. Because not every card is a number, we had to come up with some ways to represent non number cards as numbers. To do this we simply assigned the "index" of the card to it's internal rank representation. When we say index, we mean the typically ordering of cards seen in game like poker (King, Queen, Jack, 10-2, Ace). Of course, the Ace and take two orderings, but as it can usually represent 1 in most games, and it commonly shown with a singular suit element, we gave it an index of 1. The suit enum is more trivial. We arbitrarily assign indices to the suits. As long as we are consistent with the choice of indices, the actual choice doesn't matter. To make interacting with the structures less error prone and easier to understand we implement a number of member methods, `get_rank`, `as_string`, etc… For a more concise description on implantation, or a complete list of member methods, see the code documentation.

Deck

With the card structure out of the way, representing a deck actually become quite trivial. We use an actually `std::set` structure to store all cards in the deck. As in a proper standard deck of cards, there are no duplicate cards. This of course means the we are physically unable to represent multiple duplicate cards in a program, which we list in a limitations section. Much like the card, to make using this structure less data prone, we implement a number of member methods. We also add some more abstract methods, like `Deck::take_random_card` that make drawing a random card from the deck (a common operation) easier.

Hand

Much like a deck, there should be no duplicate cards in a hand assuming the deck has no duplicate cards. For this reason we also use a `std::set` for representing hands. We could use the `std::set` by itself, but to make working with it a bit more simple and for adding abstract operations to hands, we create a C++ structure, `Hand` to represent hands in our project.

Game State

To represent actual game state we create the `BlackJack` class. This class contains `Hands` for the dealer and player, a `double` for the player's bet and money, and a `GameState` enum for representing the current state of the game.

Control Flow

To manage control flow in our game, we use a **Finite State Machine**. Finite state machines are powerful control flow structures covered in the Discrete Structures class. It allows us to discretely represent the state of the game without having to force the entire state of the program to follow the state of the game. It also makes managing game rules, edge cases, and state traditions much easier. In our `GameState` enum we create a number of possible states. For more information on implementation and exact state values, use the code documentation. In particular here, I would like the take a look at the implementation of the Finite State Machine behavior. To make following the control flow simple, we use a match statement in our main `BlackJack::step` method. This also makes state transitions quite clean and easy to follow as you will see a literally assignment of game state. It also means that each step knows exactly what state the game is in, as it is the state.

Probability Calculations

Calculating wining hand probability in our program at this point is quite trivial. To make doing this a bit easier, we attempt to break the problem down into simpler components.

First we need to know, given the current deck state, the probability of this hand receiving a card that causes it to "bust". To do this we add a method to the `Hand` class called, `Hand::get_score_probability`. This method returns a `double` which represents the probability, given the deck passes as a parameter, that a randomly chosen card with cause the hand's value to exceed the given `value` parameter. We calculate this by iterating through every card in the deck and creating a "possible hand". We then calculate what the value of this "possible hand" and see if it's value exceeds the given value. If it does, we add one to a `count` variable. Once all cards are counted, we take the possible number of "exceeding hands" and divide it by the possible number of hands (the number of remaining cards) to get the probability that we draw a card that causes this hand to bust.

Next, since we know the dealer's hand and behavior, figuring out the probability of the player wining after a hit become quite trivial. We can essentially perform a "consider all possible states" calculation. Since there are only three possible game states we care about (loosing, tying, and wining), we simply need to calculate the probability of two then the remaining is %100 minus the probability of the other two. Since we only need to calculate the wining probability before and after a hit, we can lump tying into loosing, then take the remaining for the wining probability.

To calculate the probability of loosing, we already have most of it done. We need to calculate the probability of the player busting, then we need to multiply that by the probability that the dealer, given their strategy, gets a score equal to our higher than ours. The ladder is the more difficult part. Since each possible card given to the dealer (assuming they draw) can lead to the dealer taking another turn, we have to manually search the tree of possible dealer states. This may sound like a very complicated problem, and for all intents and purposes, search problems are still a very active area of research, we can make some very valuable optimizations to our search. For one, though it may seem obvious, not searching any cards that cause our dealer to stop taking cards, causes or dealer to bust, or causes our dealers to get a higher score reduces the search space quite a bit. For the remaining cards, we use a recursive approach to search possible game states. Since we made our `BlackJack` class not rely on any hidden state, it can be easily cloned and copied. This means for our search operation, we can simply create a clone of our game state, then have it search it's possible next game states. This is precisely what we do in our `BlackJack::calculate_wining_probability` method. It performs the aforementioned search, and returns the probability given this current game state that the dealer wins if the player stands, or if the player hits.

UX

To make our program usable, we create a GUI (Graphical User Interface) using **raylib**. Raylib is a simple C game programming library intended to make making C/C++ games very simple and straight forward. Here we will use it for creating and managing the window and user interaction. We create a couple sprites (small game images) for our suits and create a method that takes a card and draws it. We then add a couple other embellishments to make the game more visually appealing and some button for interaction. We can then perform our calculations and using the raylib `TextFormat` function and `DrawText` function, we can show any other relevant game state to our user.

Limitations

Of course, with any program, unless we represented the entire universe, there are limitations to what our programs can do.

For this program there are some very obvious limitations I would like to share.

Code Legibility

While we try to maintain a high code legibility, there are undoubtedly some parts of this program, where legibility isn't the best. As a general rule of thumb, if a method or line of code isn't verbose enough to understand what it is doing by just the method / variable names, it needs comments. There are a number of functions and lines in this program that are not commented as thoroughly as they should be. One big contributor is the `raylib_renderer` main file. This originally wasn't meant to be the main executable of the program, but I got a bit carried away writing it and didn't give the console version of the renderer the time it needed. As such the console renderer was cut due to time constraints.

Deck representation

As mentioned, we make a few assumptions about the game. One of the big ones being that the game uses a standard deck with no duplicate cards. This does mean that if in the future a contributor wanted to change the game to allow duplicate cards, it would require a fair amount of refactoring.

Probability Calculations

The method used to calculate probabilities is quite "brute-forcish". There is certainly a way to perform these calculations "offline" without having to search the space of possible game states. We of course know this is possible because there are common "card counting" tactics used by people in real casinos. So much so that many casinos have methods in place to prevent these people from playing. It is safe to assume that these people aren't manually considering every possible game state in their head to inform their decisions. Instead they are using some kind of well mathematically formed heuristic. We assume these heuristics were formed with much more intelligent calculations, as they were made many many years before the modern era of computers. As we are in the modern era of computation though, most if not all computers as more than fast enough to perform the brute force probability search algorithm used in this program.

Pseudo Code

```
Initialize all UX and UI components.

bet := 0.0 // Create a variable for
storing the player's bet that allows the
renderer to update it as it pleases.

// Create the BlackJack game, and create a
standard Deck.
game := BlackJack()

Set Black Jack game state to the inital
state.

while the game shouldn't close:
  wining_probability := 0
  if GameState not equal to PlaceBets,
Reset. (If the game is running):
    wining_probability := calculate wining
probability.

  Render the game.
  Render the UI elements including the
wining probability.

  if GameState == PlaceBets:
    Allow the user to update their bet.
    Show a button to start the game.

    if Start game button is pressed:
      game.SetBet(bet) // Set the game's
bet amount to the current renderer's bet
amount.
      Remove the bet amount from the
players money.
      Set game State to
PlayerTurnDoubleDown

  if GameState == PlayerTurnDoubleDown or
GameState == PlayerTurn:

    if GameState == PlayerTurnDoubleDown:
      Show the Double down button

      if The User presses the double down
button:
        Deal a card to the player hand.
        Double the game bet.
        Set the game state to the dealers
turn.

        Continue the loop skip remaining
code.
    else:
      Show grayed out double down button.

    if Player pressed hit me button:
      Deal a card to the player.

      If the player busted:
        Set the Game State to PlayerBust
        Skip remaining loop code.

      if GameState ==
PlayerTurnDoubleDown:
        // The Player decided not to
double down.
        GameState = PlayerTurn

    if Player pressed the Stand button:
      Set the game state to the dealers
turn.

  if GameState == DealersTurn:
    if the dealer's hand value < 16:
      Deal a card to the dealer.

      if The dealer busted:
        GameState = DealerBust

      skip the reamaining loop code.

    if Dealer's hand value == Players hand
value:
      GameState = tie
      skip remaining code.

    if Dealer's hand value > player's hand
value:
      GameState = loss.
      skip remaining code.

    GameState = Win

  if GameState == loss:
    Show a you loose message
    GameState = Reset

  if GameState == tie:
    show a tie message.
    GameState = Reset

  if GameState == win:
    multiplier := 1.5
    If the player doubled down:
      multiplier = 2.5

    Players Money += bet amount *
multiplier

    GameState = Reset
    skip remaining code;

  if GameState = Reset
    Reset the Game deck.
    Clear the player and dealer hands.
    GameState = PlaceBets.
    skip reamining code.
```