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, and program limitations / 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 **Black Jack Problem** as outlined in the reference documentation.

To summarize the problem briefly, we were required to recreate the famous game Black Jack 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 the project prompt may seem short, it 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 need 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 in (which I will call state here on) is probably the most fundamental problem any programmer has 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 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 need some way for a user 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. 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 representing any non possible card (i.e. 11 of spades) not possible; Though, this 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 games 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 is 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 implementation, 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 a `std::set` structure to store all cards in the deck. As in a proper standard deck of cards, there are no duplicate cards, a set becomes a natural choice. This of course means, we are physically unable to represent duplicate cards in the program. We list this limitation 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 a `Hand` for the dealer , a `Hand` for the player, a `double` for the player's bet, a `double` for the player's money, a `Deck` for the deck of cards, 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 transitions 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. For every transition, you will see an assignment of game state. It also means that each step knows exactly what state the game is in.

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 passed as a parameter, a randomly chosen card will cause the hand's value to exceed the given `value` parameter. We calculate this by calculating the number of cards that would cause the value to exceed `value` in an untouched deck, then we subtract away the number of cards in the `Deck::_missing_cards` member variable that are less than that value.

Next, since we know the dealer's hand and behavior, figuring out the probability of the player wining after a hit become quite trivial.

For the case where the player stands, if the dealer's score is greater than 16, meaning the dealer wouldn't hit, the probability is 100% if the player's score is higher than the dealer's, otherwise 0%.

In the case where the player does hit, we calculate the probability the player will get a card that make's their score higher than the dealers, but not busting.

Then, if the dealer will hit, given it's current hand, we calculate the probability the dealer will get a card making their score higher than the players, less the probability they bust. Then we multiply this value by the previously calculated probability.

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. We then create a method that takes a card and renders it. Then, we add a couple other embellishments to make the game more visually appealing, and some buttons 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.

Discrete Structures Use

Though the use of the discrete structures taught in CIS-7 Discrete Structures have already been discussed, to make grading and reviewing this project a bit easier, I have included this section.

In this project we use several discrete structures to solve the given problem. We use a multitude of **sets** to represent the state of the `Deck`, and `Hand` classes. We use a **Finite State Machine** in the `BlackJack` class to represent and manage Game State. We use a multitude of enums, which one could consider a set, to represent the possible state of many classes in this project. We also use a multitude of **probability calculations**, on said sets, to answer the original project prompt.

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 (now) 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 probability calculations can definitely be improved. There is almost certainly a closed form solution for the total probability of winning, including all possible branches of actions the dealer can take. This program doesn't perform this calculation. Forming such a calculation on it's own would take lots of time, and by the rules of this project, we can not hard code any probabilities. So whilst there is most definitely a closed form solution, and it may even be possible to optimize the search algorithm (possibly) required to form one, doing this in real time most likely isn't feasible.

Hand Splitting is not Implemented.

While implementing this wouldn't be very difficult, it would require quite a bit of refactoring in the current state of the program. Refactoring to make this behavior possible would exceed time constraints. So it has been omitted.

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.
end
```