Poker: Programming Final Project

Flavie Qin

3 May 2024

Contents

1 Introduction

I found Poker to be an especially interesting topic for this project, because it is a zero-sum game of imperfect information (every amount a player wins must be lost by another player, and some information is hidden). Indeed, this game is not guided purely by probability theory, since players are able to "bluff" opponents. Poker was even the inspiration for the invention of game theory for John Von Neumann and Emile Borel in the 1920s. Game theory is very useful in many fields, such as economics and war interactions, which model zero-sum interactions. (Source: `https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/neumann.html`.)

More significantly, I think that Poker is a very fun game, and I really enjoy playing it. I also thought it would be a great challenge to try and design a bot, since the decisions are just completely trivial computations (as it is a game of imperfect information). Furthermore, this project would also allow me to practice making graphical user interfaces and doing some event-driven programming, which would reward me with a tangible application that I can actually interact with.

My application is an implementation of the popular Texas Hold'em no limit Poker variant. However, right now, the user can only play against a computer bot (i.e. there is no multiplayer function). Thus, it is exclusively Heads up Poker, which means that there are only two players in the game.

2 User Mini-manual

2.1 Texas Hold'em no limit rules

Basically, in Texas Hold'em Poker, each player receives two personal pocket cards, which are only visible to themselves. Cards are placed onto the community table face up in phases, followed by betting rounds. There can be a total of 5 community cards. The goal of the game is to have the best ranked hand 5-card hand out of the 7 cards (pocket cards and community cards) and win the pot. The game ends when one player runs out of money.

I will summarize the rules below, but if they are still unclear, you can read more about it on this Wikipedia page: `https://en.wikipedia.org/wiki/Texas_hold_%27em`.

2.1.1 Game structure

I defined a game to be separated into many substructures:

- One game is from the very start, until one of the players has no more money left in their balance. A game is separated into rounds.

- A round starts when blinds are posted and pocket cards are distributed, and ends once the pot has been given to the winner. A round is separated into many phases.
- A phase is every time a new card is placed on the community table. There are 4 phases: preflop (0 cards on community table), flop (3 cards), turn (4 cards) and river (5 cards). Each phase is followed by a betting round: hence, there are 4 betting round in total.

The game is played with a single standard 52-card deck. Each round starts with shuffling the full deck.

If both players are still in the round after the river betting phase, or one of the players has gone all in and the other agreed, then they go into showdown: the players' pocket cards are turned face up, cards are revealed to complete the 5-card community pile and the winner is determined.

2.1.2 Blinds

At the beginning of every round, players must post their blinds, which is just a small, mandatory bet at the very beginning of a round. This ensures that the game does not go on forever and kickstarts the preflop betting phase. If a player does not have enough money to post their blind, then they lose the game.

In a game with only 2 players, one player will be the small blind, and the other player is the big blind (which generally has a value of double the small blind). At the start of every new round, the blinds are switched. Therefore, in Heads up Poker, each player will be alternating between being the small blind and the big blind. In my application, I used 5$ for the small blind and 10$ for the big blind. I also decided to make the players start off with a balance of 500$ each.

2.1.3 Betting structure

No limit Poker just means that there is no limit to how much or how often a player raises.

A betting phase ends when all players have done at least one action and all players agreed on the amount to bet on. There is an exception to this rule: if the bet proposed by another player is higher than the maximum possible bet (i.e. going all in) for a player, then that player has the option to call, by going all in, and that counts as agreeing on the amount bet.

During betting rounds, there is an order of which player gets to go first. In the preflop phase, since it is right after the blinds, the small blind must go first as it will have a smaller current bet. However, even if the small blind agrees to call the same bet, the big blind player must also have to chance to choose an action since they would have not done an action yet. In the postflop betting phases (those following the flop, the turn and the river), the big blind player bets first. However, in any given betting

round, there is no guarantee on which player closes the betting, since it depends on if players choose to raise or not.

When betting, here are the possible actions a player may choose, if available:

- Check: this action is only available when the other player also checked, or if the bet of the current phase is 0$. This is equivalent to betting 0$. If both players agree on checking, then the next phase of cards can be revealed without betting any money that phase.
- Call: this action is available when the opponent has already posted a bet in the current phase. This action means the player agrees on that bet.
- Raise: this action is generally always available, unless the opponent's bet is already above your maximum bet, in which case you can only call to go all in. This action means the player wants to bet even more than the opponent's bet. The value a player raises to is the value they want the bet to be. The minimum raise amount is to raise by the last raise amount in the current phase. If there has been no raises yet, the minimum raise amount is the value of the small blind.
- Fold: this action is generally always available, unless the player has already gone all in (in which case folding would be useless, as it is just forfeiting their entire balance). This action entails discarding your pocket cards and forfeiting the current round, thus leaving any amount previously bet into the pot for the opponent. After a player folds, a new round is started.

For example, this may be a possible procedure in a postflop betting phase: Player A starts off the betting phase and checks. Player B raises to 20$. Player B could call, which means that they agree to bet 20$ and the next phase would start. Player B could fold, which means that they forfeit whatever amount they had already put into the pot previously and a new round would be started. Lastly, Player B could raise: the minimum amount would be raising to 40$, since the previous raise was to increase by 20$, and the maximum raise would be to go all in. If Player B raises, then Player A must do another action. In this situation, Player B cannot check anymore, since Player A has already put a nonzero bet into the pot this phase.

2.1.4 Hand rankings

The rank of a card corresponds to its number, and its suit corresponds to its symbol (Diamond, Heart, Spade, Club). Generally, in Poker, Ace is the highest ranked card. Moreover, all suits are of equal value.

When determining the ranking of a hand, since there are 7 cards between the community pile and a player's two pocket cards, the 5 cards forming to best possible hand is chosen.

Below is a list, from best to worst, of 5-card hand rankings. If there is a tie in categories between the players, the high card or kickers will be used to determine the winner. If all of that is the same, then a true tie will be drawn.

- Royal flush: 10-Jack-Queen-King-Ace, all of the same suit. This is a combination of a straight and a flush, with the high card being an Ace.
- Straight flush: 5 cards in a sequence, all of the same suit. This is a combination of a straight and a flush. The rank of the highest card in the straight flush determines the winner.
- Four of a kind: 4 cards of the same rank and 1 other card. The rank of the four of a kind cards is used to determine the winner.
- Full house: 3 cards of the same rank and 2 cards of (another) same rank. The rank of the three of a kind is used to determine the winner, then the rank of the two of a kind cards is used as a kicker.
- Flush: 5 cards of any rank, but all in the same suit. The highest ranked card of the suit is used to determine the winner, then the second highest, then the third highest, then the fourth highest, then the fifth highest.
- Straight: 5 cards in a sequence, offsuit. In a straight, ace can act as the top of bottom of a straight. For example: Ace-2-3-4-5 or 10-Jack-Queen-King-Ace.
- Three of a kind: 3 cards of the same rank and 2 other unrelated cards. To determine a winner, the rank of the three of a kind card is used.
- Two pair: 2 cards of the same rank, 2 other cards of (another) same rank and 1 unrelated card. To determine a winner, the highest ranking pair wins, then the rank of the other pair, the the rank of the unrelated card.
- Pair: 2 cards of the same rank and 3 unrelated cards. To determine a winner, the rank of the pair is used, then the 3 unrelated cards are used as kickers, such that the highest rank is used, then the second highest, then the third highest.
- High card: any hand that does not qualify for the other categories. TO determine the winner, the rank of the highest card is used, then the second highest, then the third highest, then the fourth highest, then the fifth highest.

(Source: `https://www.pokerstars.com/poker/games/rules/hand-rankings/`)

The player with the better ranked hand will win the pot. However, if that player bet less than the other player (by going all in), then they can only get from the opponent an equal amount of what they put into the pot. In other words, the winning player can win up to double the amount of their total bet, and any remaining money in the pot (if any) will go back to the other player.

2.2 Using the application

2.2.1 Getting started

This application was built purely with standard Python libraries, so nothing extra needs to be installed. Thus, setting up is very simple: to start playing, the user must have the `poker` folder as their directory and run the `ui.py` file.

2.2.2 Table layout

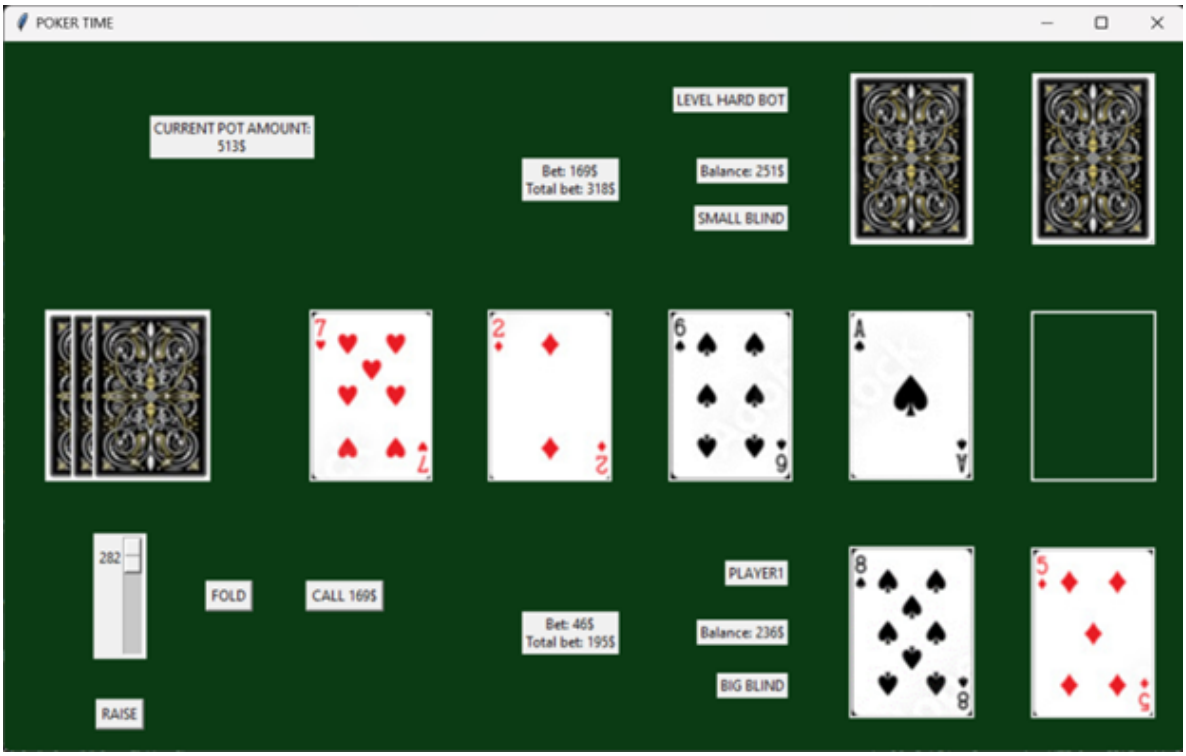

Figure 1: Table layout of the Poker application during a betting phase

When betting, the user may pick an action by clicking on one of the buttons shown in the bottom left side of figure 1. To raise, the user must adjust the slider to the value they want to raise by, and then click the `RAISE` button to submit that value. To adjust the value, the user may drag the slider on the scale, which will give large changes in value, or press on the vertical bar, which will slowly increment or decrement the raise value. If the user cannot do a certain action, that button will not be

displayed. If it is not the user's turn to bet, then no buttons will be shown. The values displayed in the `CALL` button or `RAISE` slider correspond to the value to bet to in that betting phase.

The label in the top right side of figure 1 displays the current pot amount. The amount of the bet of a certain phase corresponds to the value shown after `Bet`, and the total amount bet over that round corresponds to the shown value after `Total bet`.

The two cards at the bottom right of figure 1 correspond to the user's pocket cards, while the two cards at the top right correspond to the opponent's pocket cards, which are not visible to the user. The cards in the middle row correspond to the community cards.

2.2.3 The Start and End menus

The first thing that will pop up is the user start menu. There, the user must choose a difficulty level for the bot (the default is set to `EASY` mode) and enter a username. There is a limit of 25 characters for the username. By pressing `Enter` on the keyboard after writing the username, the main game will start. If nothing is written in the username entry box when the game is started, it will default to the username `PLAYER1`. Then, the main Poker application will run.

After a game has ended (or if the user decides to quit the current game by closing the window of the Poker application), an end menu will be displayed. There, a user may choose to play a new game, by clicking the `PLAY AGAIN` button, where the start menu will be redisplayed, or to quit, either by clicking the `QUIT PLAYING` button or by closing this window, in which case the program is terminated.

3 Design Guide

3.1 Object Oriented Programming

I had to make use of a lot of object oriented programming to structure my Poker application.

I also referred to slides 12 and onwards of this lecture on class diagrams to help me think through the structure of my program: `https://courses.cs.washington.edu/courses/cse403/13au/lectures/08-classdiagrams_updated.ppt.pdf`.

I implemented the following classes: `Card` and `Deck` in `table.py`, `Player` in `players.py`, `Bot` in `bot.py`, `PokerGame` in `game.py`, `StartMenu` and `EndMenu` in `ui.py`.

The `Card` class allows me to have card objects, of which the deck can be composed of. These objects have attributes `faceUp`, `rank` and `suit` and a few methods such as `flip`. The main thing was that I could implement total ordering, which was extremely useful for finding the rank of a hand (which relies on sorting through all the cards in a hand). Also, I used the `__repr__` method to return a string, which corresponded to the rank and suit of the card if it was face up, or the string `"card"` if

it was face down. Thanks to this trick, all I had to do for the display of card images in the GUI was to save images of all the cards following the naming convention of this method. Therefore, all I had to do was add the file extension to the string representation of a card and it will display the correct image, without the need for any mapping or dictionaries, or checking if it was face up or down! Moreover, I also named my other images based on this principle, such as `"None"` for the image with no card. All of these images are saved in the `images` folder. I found the image for a full deck cards online and cut out each card and saved them individually. Unfortunately, it appears that the page that I got the images from is no longer available, because my link no longer works. Some testing code for this class can be found in `table.py`.

The simple `Deck` class allowed me to have a deck for the Poker game itself, with the implementation of a few methods, thus simplifying the usage of the deck. It consists of a list of 52 `Card` objects. Instead of having to create a list and manipulate the deck that way, using a `Deck` object just made the code a bit clearer and a bit more direct. Some testing code for this class can be found in in `table.py`.

The `Player` class consists of a few attributes for a player in a Poker game, such as places to store pocket cards, balance, bet in a phase, total bet in a round, etc., as well as Boolean attributes to keep track of whether or not they are alive in a game, a round and if they did an action yet, which are useful to know when to terminate a game/round/betting phase. Moreover, the only methods in the `Player` class are the methods that handle the modifications to attributes when an action is taking while betting (namely `check`, `raisebet`, `callbet` and `fold`). These methods do not do any checking themselves to make sure it is a legal move, because they assume that if a certain action is taken, it must be legal (this assumption is taken care of by only displaying the legal buttons for the human user and by providing a list of legal moves to choose from for the bot).

The `Bot` class inherits from the `Player` class. Its only extra attribute is `difficulty`, which is used to determine what move to make, as the different difficulties correspond to different betting strategies (more about that in section 3.5). The main method that is called by the game engine is `doAction`, which makes the computer pick an action depending on its betting strategy. The other methods define the decision process for the different betting strategies.

The `PokerGame` class is the actual game engine, as well as the graphical user interface for the game (see section 3.4 for more). It inherits from the `tkinter.Tk` class. I inspired myself a lot from the `simon.py` code, from Lab 6, especially for the game state idea and the event bindings. The basic idea is that this class handles the game engine (i.e. takes care of when the computer needs to make a move, etc.), as well as displaying all the stuff needed for the user to play. It also takes care of actions from the dealer, such as giving pocket cards or revealing community cards.

Lastly, the `StartMenu` and `EndMenu` classes inherit from the `tkinter.Tk` class as well. They take

care of displaying some options for the user before and after a game, and save the options chosen by the player to feed somewhere else.

3.2 Finding the rank of a hand

One large task of this project was coding a function that could determine the category of a hand, as well as being able to determine the winning hand between two players (which is what I need for the Poker game itself). The code for all of this can found in `winninghand.py`.

First, I needed to build smaller functions, that could find if certain combinations were present (`flush`, `straight`, `four_kind`, `three_kind`, `two_pairs`, `pair` and `highcard`) and returned a list of cards that could possibly be part of that category. This is very important, because I do not want it to just return 5 cards that form the best hand of that smaller category, as it would not work for the categories that are built on multiple others, such as full house (three of a kind + pair) or straight flush (straight + flush). Then, to find the category of a hand, I just took the results from those functions and found ways to determine if a category was present, from best to worst. I had to do some manipulations for the Ace case in straights, where it could act as the highest or the lowest card, but I solved that using sorting and extra checking. This is the `rankHand` function. It returns a list, with the first element being an integer corresponding to its best possible category of the hand (with 9 being the highest, royal flush, and 0 being the lowest, high card). The rest of the elements are the 5 cards of the hand formed for this best category. It is also important to note that they are ordered based on their importance for determining winners in the case of a tie, where the more important ones come before the less important ones.

Then, I exploited the lexicographic nature of list comparisons, so that I could just compare the two lists returned from `rankHand` to find the better hand, even if they were of the same category, because all the kickers would be placed in the right spot relative to one another. This is the function `getwinner`.

There is also a function `category`, which just returns the string corresponding to the integer category. This is useful for displaying messages in the Poker game, to know what category hand the winner had. This uses simple mapping in a list, where the index corresponds to that integer category and the item at that index the string representation.

Testing code for `rankHand` and `getwinner` can be found in `testing_winninghand.py`, which is in the `testing` folder. I separated it into a different file simply because it was quite long.

3.3 Getting legal moves

Finding legal moves was another important task. The method `legal_moves` is in the `PokerGame` class, and it returns a list of lists, one list for each of the players. Since there are only four possible actions when betting, these lists are of set length of 4 each. Each index of the list corresponds to an action. If an action is illegal, the item at that index will be set to `False`, otherwise it will be set to `True` or it will be some values that are representative for that action (such as the amounts for calling or raising). To find the legal moves, I just used a lot of conditions, since the legality and amounts just depends on the situation of the game and other players' moves. There is no testing code for this method, because it depends massively on the situation, and because it is included in the game engine itself. Thus, I tested it by playing the game, to check for any bugs, and it seems to be all correct.

3.4 Graphical user interface and Event-driven programming

The graphical user interface and the event handling were all done using `tkinter`. I referred a lot to the `tkinter` reference manual by John W. Shipman, provided in Lab 6, since this is essentially my first time doing and understanding any GUI or event driven programming.

For the window of the game itself, I used the `grid` method, because it was easier for me to place the widgets relative to each other, rather than placing them using pixel values. Also, this allows for the window to be resizeable! (To a certain extent; the widgets themselves are not, so if the window changed to be too small, then all the widgets get smushed together. However, the window is still resizeable, unlike `Canvas`, by altering the spacing between the widgets.) I separated the window into 6 rows and 14 columns, just to be able to place all widgets where I wanted.

Moreover, I used many widgets for all the `tkinter` applications, including `Label`, `Button`, `Scale`, `PhotoImage`, `messagebox`, as well as `OptionMenu` and `Entry` for the starting menu.

I was able to make the application update dynamically, but I also implemented some time delays using the `after` method, to make the experience more enjoyable and less confusing for the user. I also added some message boxes, notably at the end of each round, so that the user could take the time they wanted to process what happened or look at the cards in the case of a showdown.

For the game engine itself, I used two game states: `INPLAY` (which is when the user is allowed to do an action) and `WAITING` (which is when the user is not allowed to do an action). Every time a new round is started, it will do all the initial stuff needed (blinds, distributing pocket cards), then it will make the computer play if needed. However, after any computer move, it checks whether the two players agree in their bets, and if not, then the game state changes to being `INPLAY`, thus waiting for the user to do something. Once the user does something (since the buttons are bound to methods which process these actions), the game state changes to `WAITING` and it will check for what needs to

be done (start a new phase or make the computer play again). Thus, this loops on, until a player folds or the betting ends!

3.5 Betting strategies for the bots

Of course, one major goal for this project was to implement a computer bot to play Poker. I decided to make heuristics, rule-based bots. The bot is fed its possible legal moves and picks an action that is legal.

3.5.1 Difficulty levels

I implemented three difficulty levels, `EASY`, `MEDIUM` and `HARD`, which all correspond to a different betting strategy. The ideas for different Poker strategies come from this paper: `https://nlp.fi.muni.cz/aiproject/ui/kollarova_martina2014/Kollarova_Martina-poker.pdf`.

Here are the betting strategies for each level:

- `EASY`: The easy level bot just picks a random move. However, if it picks fold, when check is available, it will check instead. If it picks raise, then it will raise to a random amount within the legal range.
- `MEDIUM`: The medium level bot employs a "passive loose" strategy: it raises only if cards very good, and in preflop, it plays pocket cards even if they are not very good.
- `HARD`: The hard level bot uses a "aggressive tight" strategy: it raises even if cards are not very good, and in preflop, it plays pocket cards only if they are very good.

The easy bot is quite easy to implement, using the standard `random` module from Python. However, the medium and hard bot decide on an action based on calculations and comparisons of pot odds, hand strength and some thresholds. For the raising, the medium and hard bots also use random value. However there are more thresholds as to when to raise, and how much to raise by.

3.5.2 Hand strength

Hand strength informs us about how good a hand is. It is essentially the probability that a set of pocket cards will win, also depending on the cards which have already been revealed on the table. I got the idea to use hand strength from `https://nlp.fi.muni.cz/aiproject/ui/kollarova_martina2014/Kollarova_Martina-poker.pdf`.

Since I need my bot to evaluate hand strength every time there is a new card placed, which can be simplified to every time it makes a bet, it needs to be a fast process. Also, there are so many combinations of cards in Poker, that I did not think saving these values and mapping them somehow

would be a good idea. Moreover, I did not want to have my bot calculate the true hand strength every time, since it gets pretty slow (for example, in the flop stage where there are 3 known community cards, that would be 1070190 combinations minimum to go through). Therefore, I had the idea for it to do some random sampling, by generating some cards and completing the community pile, and counting how many times it would win. I wanted to be sure that doing this sampling would be accurate enough and would not be somehow biased. Thus, I tested this idea, by comparing its performance to the performance when finding the true hand strength, and it seemed to be quite accurate! The testing code can be found in `testing_handstrength.py`, in the `testing` folder, and a much fuller analysis of the results can be found in `testing_handstrength_analysis.txt`, in the `testing` folder as well.

Hence, I made the `handstrength` function, which returns the estimated hand strength of a set of pocket cards based on what is already there on the community table. It can be found in `bot.py`.

3.5.3 Pot odds

I got the idea of pot odds from `https://nlp.fi.muni.cz/aiproject/ui/kollarova_martina2014/Kollarova_Martina-poker.pdf`. Basically, the pot odd is a number from 0 to 1, which tells us about how worth it it is to call a bet. The more it costs for you to continue, the higher the number is. It is defined as:

$$PO = \frac{C}{C+P}$$

where PO is the pot odds, C is the amount needed to call a bet and P is the amount of money currently in the pot.

However, I found that this measure tended to be quite small, especially compared to the hand strength, meaning it would not be that useful when comparing to the hand strength. Thus, I modified it to be:

$$PO = \left(\frac{C}{C+P}\right)^{3/4}$$

Since PO is a number from 0 to 1, taking it to the exponent of a value smaller than 1 would make PO a little bigger, just to make it a more significant measure.

4 Conclusion

Ultimately, I think that this was quite a successful project: I made a working GUI for Poker, everything seems to work correctly with no bugs, and my bots are not terrible. In this process, I

also learned a lot about the foundations of object oriented programming, and also how event-driven programming actually works.

In the future, I would like to add more options the user can choose from the starting menu, such as the small blind/big blind values and the starting balance. That should not be too hard, since those values are just attributes for the `PokerGame` class or the `Player` classes.

I would also like to work on the bots, and maybe make them a little more sensitive to the human user's actions. Also, I want to try and find a way to implement more intentional bluffing for the bot, as that is a big part of Poker. I might even try to make a bot using reinforcement learning, which would be a little smarter.

Another thing I want like to eventually implement is some sort of multiplayer function, at least to have the ability to play against multiple bots. If I do, then I would probably need to rework some things, especially in the game engine of the `PokerGame` class.