CS 3110 Final Project Design

Amber Wiens, Nikhil Dhawan, Amrit Kwatra

System Description

Core Vision: An interactive poker game with a text-based interface, where a human plays against an AI. This game will include betting, an AI that can intelligently determine the expected return of its hand and place bets accordingly, as well as some randomization (using a metric for confidence) to simulate bluffing.

Key Features:

- An AI to compete against in Texas Hold'em Poker
- Gamble against an AI
- Interactive text-based interface in which the player can provide commands

Description:

We will design a Texas Hold'em Poker game/bot in which a single human player competes against one AI. The player and AI will having a certain starting amount of money and will be able to gamble, while the house will have unlimited money. Interactions between the player and the AI will occur via a text interface, which will display information about the current state of the game and the AI's moves, and allow the player to input commands.

We also have made some executive decisions regarding our game design, namely, blinds are fixed, and do not scale. Given the one-on-one nature of the game, we found it advantageous to our gameplay if the blinds did not progressively increase over time.

Architecture

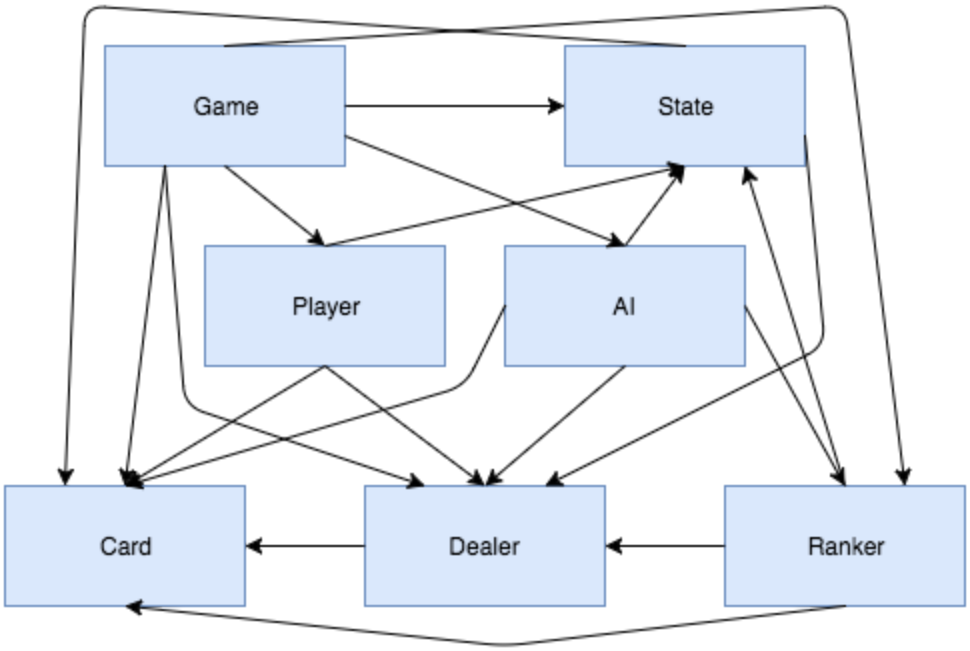
Our system represents a client-server architecture:

Components: There are three components: the player, the AI, (both clients) and the server, which represents the system that controls our game. The server takes command input from the player and the AI, and relays information about the current state of the game to both of them.

Connectors: The channel between the server and the player client is the text-based interface that allows the player to make requests by inputting a command and returns a response from the server regarding the state of the game. The channel between the server and the AI client sends information about the player's betting and the current cards in the river. The channel also accepts commands from the AI.

System Design

The **Game** module acts as a control module that initializes the state and holds the REPL which prompts user and parses user commands, it also holds functions having to do with the validity of data being held in the state.
The **Dealer** holds deck commands such as shuffling and dealing of the player (hole) cards as well as the community cards that every player can use to create a hand.
The **Ranker** determines who wins a round only when both the player and the AI complete their final betting round.
The **AI** determines the best command to issue based on the player data it receives, such as the player's total money, list of cards, and added functionality to account for bluffing etc. It also determines that in the case the player wishes to compete in a round, how much money they should bet, whether or not they should call/check or not.
Card holds the data type for how a playing card is defined as well as a to_string function.
Player holds the data type for each player and function to determine if a command made by the player is valid or not. These functions for instance determine if the player can make a bet or not based on the money they have left.
State contains definitions of player and state types, and serves to avoid circular modular dependency.



Data

The game holds each player's money and current hand. It also holds the round number, and the deal number, which determines the current deal or betting round (i.e. 0 for posting blinds, 1 for the first betting round, 2 for flop, and 3 for the second betting round, etc.). Moreover, it keeps track of which player is the AI and which is the human, along with whether this player is active (hasn't folded in the current round) This data is not persistent, as the state is updated after each turn and the older states are not longer used and are left for the garbage collector to consume.

External Dependency .

Our prototype will not be utilizing any third-party libraries other than requiring Camomile (unicode library), ANSITerminal, Str, and OUnit, which are all standard in OCaml 4.03

Testing Plan

.

Testing in game functions such as ranking, dealing, betting, and the REPL will be done using OUnit testing. These tests can be isolated from the rest of the components of the game and are thereby more conducive to be tested via unit test. Functionality of the AI and the overall user/game experience and non-state-changing commands will be tested via beta testing. Testing the AI and UX/UI through beta testing will allow us to assess and tweak the AI's behaviors, and is preferable to unit testing as the AI has a degree of behavioral randomness associated with how we assigned it a sense of "confidence" - something we use to give it the capability of bluffing. We will also be making use of the rep_ok function to debug the base data structures that the game relies on. The rep_ok is abstracted from the user and not part of mli files as no client would need access to a function that's sole purpose is debugging.

Testing Part II: Findings

.

As expected the unit testing did illuminate several issues with regards to mutability issues in dealing cards to players in the many branches that the game could navigate. We managed to fix these issues through use of rep_ok functions that allowed us to isolate when the player hands were no longer valid as per their representation invariants. Unit testing also helped us assure correctness of functions that modified state and handled ranking and betting in the game. However, the majority of progress we made as part of a testing was due to our beta testing. The results were helpful in two key ways. Firstly, it allowed us to examine instances of poor UI/UX, in which the game state was formatted poorly, difficult to understand or interpret and in some cases spiralled out of control. The feedback from our friends allowed us to systematically improve upon our initial design, and ultimately present a much cleaner UI. Secondly, it allowed us to fine tune our AI such that it behaves in a reasonable manner. Initially our calculations were incorrect and the AI was wild to say the least, acting in odd manners that were incredibly predictable, so much so that our friends found many ways to trick the AI into revealing the strength of it's hand as per it's actions. This feedback allowed us to make adjustments such that the final user will not face such distinct similarities, and the AI is now smarter, and more capable of making not only statistically informed decisions about it's hand, but also has the ability to bravely gamble, and attempt to bluff out a player. To our knowledge there are no known game breaking bugs, however, the printing of the state after commands has an irregularity as far as erasing the screen is concerned. This is slightly odd in the first stage of the game, but after exploration of other alternatives, we determined the current state printing was good enough given the time crunch we faced.

Division of Labor .

Initially each member of the team worked independently on implementing specific modules, however, towards the last stretch of the project's implementation we all worked together on each module to debug and solve small and large issues.

Amrit:
Amrit was responsible for implementing the AI, doing Unit testing for all modules, coming up with rep_ok functions for debugging, and any minor corrections to modules that had to be made as a result of failed test cases. Amrit also did the design for the mli files and helped resolve system issues such as circular dependencies, and poker logic related errors in game towards the last leg of implementation.

Nikhil:
Nikhil worked on the base modules for the project and came up with ranking system for hands. He came up with the abstraction for decks, cards, hands and in the last leg of implementation worked with Amber on fine tuning the game module in order to make our project operational. He helped rewrite buggy code that we discovered was logically inconsistent as a result of Alpha/Beta Testing.

Amber:
Amber worked mainly on implementing the Player and Game modules, which included creating and altering the type of a player, maintaining all aspects of the game state as play progressed, parsing user commands, interpreting the results of AI functions, and printing out information about the current state to the human player.

Hours worked: The hours worked by each member was variable in the first weeks of project development, as each member worked independently on their own modules. Once we all began working together in the final stages of implementation each member worked roughly 6-8 hours each night for a week and a half to ensure our project was functioning properly.
The rough estimations of our hour work load are between 50-65 hours.