Project Proposal (15-112) – Fa Phanachet

Project Description

BridgeBuddy: An application to play the card game ‘bridge’ complete with an AI to play as an automated opponent.

Competitive Analysis

BridgeBase

BridgeBase Online (BBO) is the most popular bridge site, particularly for serious players. The program allows for gameplay from solo bridge with bot opponents/partners to global tournaments for ACBL (American Contract Bridge League) points and prize money.

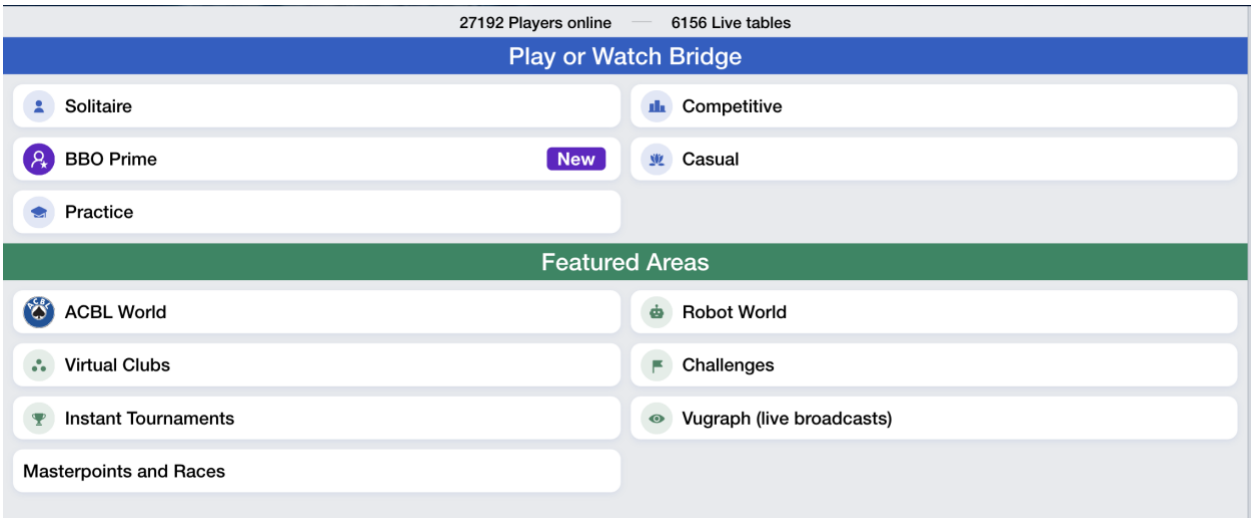
AI system:

BBO uses GIB (Ginsberg's Intelligent Bridgeplayer), which was the 1998 and 1999 World Computer Bridge Champion. However, the quality of the bots vary on BBO players can rent higher quality bots for a small fee.

More details on how GIB plays here

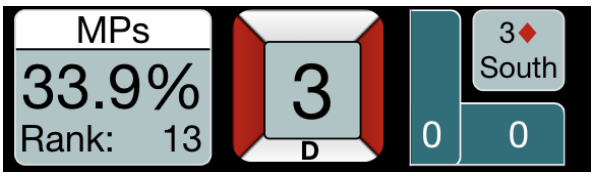
UI:

⇒ BridgeBase menu



⇒ large cards (only the top half is shown to make it more visible)

⇒ display details about the current board on top of the playing area

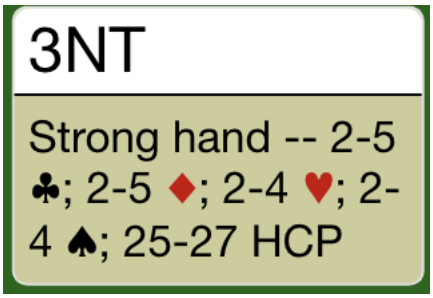


⇒ sounds for card playing

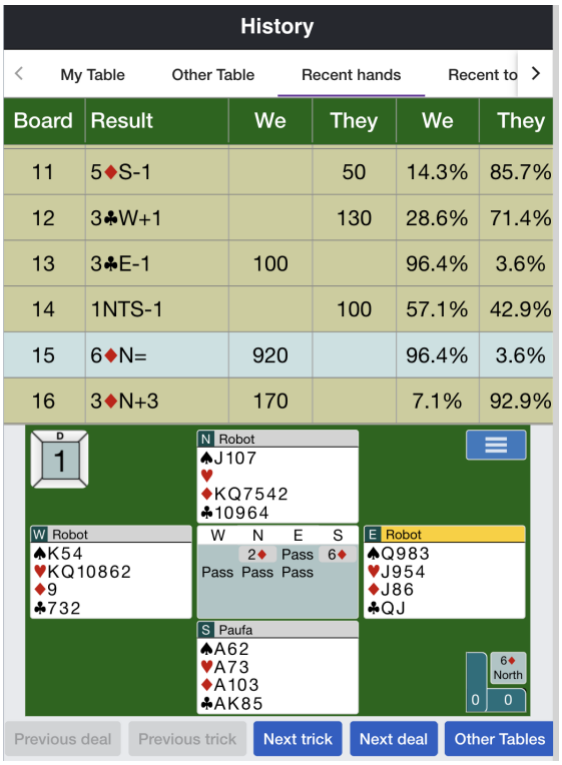
⇒ buttons for bidding (expands when you click on the number)

⇒ Also has a hand diagram feature

⇒ gives cue for AI bidding system to help AI-human communication



⇒ Hand history (with MP results and replays)



Funbridge

AI System:

Funbridge uses an AI system called Argine. It can play the following bidding systems: French 5-card major system, English ACOL system, Polish Club, SAYC, Nordic system and NBB standard

Argine assumes opponents are perfect players and does not anticipate player mistakes/lack of information.

More details on Argine here.

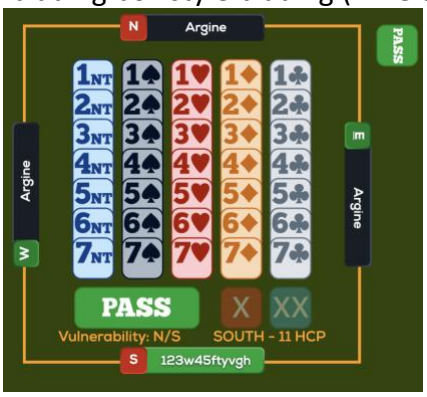
UI:

I like their UI

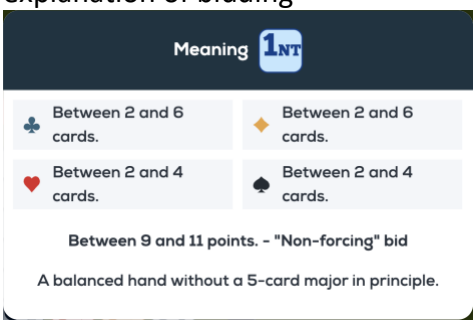
⇒ really nice menu design

⇒ cards represented in hand image (I don't like this → unclear)

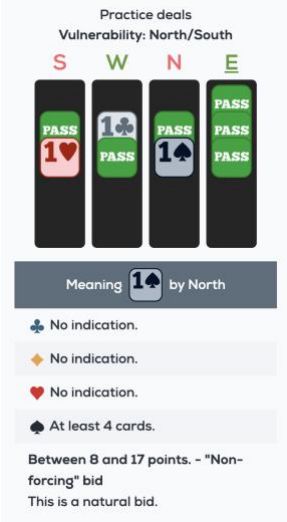
⇒ bidding box style bidding (I like this, but preferably, bigger)



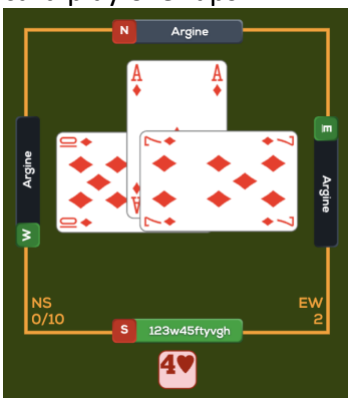
⇒ explanation of bidding



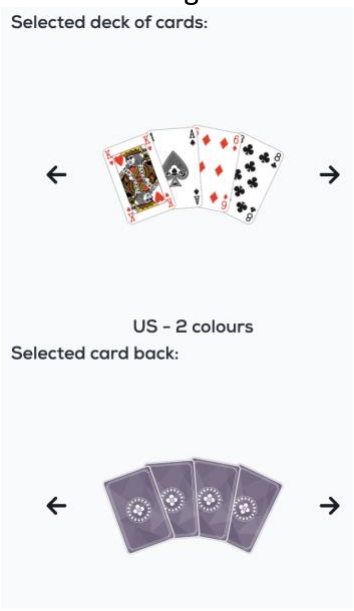
⇒ shows bidding on the side



⇒ card play overlaps



⇒ skins are changeable



⇒

247 Bridge

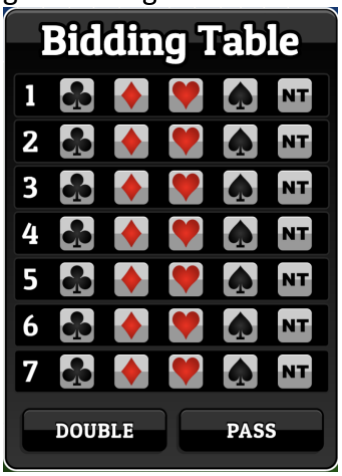
This is not a program for serious players. the bidding system does not understand standard conventions in bridge.

AI System

247 Bridge uses an unidentified system that is only described as “amazing.”

UI

⇒ grid bidding table



⇒ card dealing animation

⇒ I don’t like their aesthetics

Structural Plan

Files

Class Files:

→ button

→ card

→ bid

→ board

→ game

→ player

Mode Files:

→ login

→ register
→ welcome
→ menu
→ gameplay

Algorithmic Complexity
→ some number of minimax files
→ some number of socket files

Organization (descriptions of methods/attributes/helper functions is non-exhaustive)

Classes:

1. Button (class of everything I need to click on)
    a. attributes
        i. location (tuple of x,y or None)
        ii. width
        iii. height
        iv. action (function once button is activated)
            1. app
    b. method
        i. __init__
            1. location
            2. (width, height)
            3. action
        ii. isPressed (returns True if mouse press is inside button): assumes all buttons are rectangles (even rounded ones)
            1. event.x (mousePressed)
            2. event.y (mousePressed)
        iii. isHover (returns True if mouse moved inside the button)
            1. event.x (mouseMoved)
            2. event.y (mouseMoved)
2. Card (button)
    a. attributes
        i. suit (nswe)
        ii. number (int)
        iii. color (red or black)
    b. methods
        i. __init__
            1. suit
            2. number
        ii. __repr__ (prints as '[number] [suit]')
        iii. getCardColor (changes self.color to suit color)
        iv. __lt__ (allows cards to be sorted)

v. moveCard

3. Board
    a. attributes
        i. hands
            1. (dict, key=position+'played', value=list of cards)
            2. hands[played] = list of list of lead + cards in round
        ii. bids (list of tuples (position, Bids) in order)
        iii. bidOptions (available
        iv. ewTricks (int)
        v. nsTricks (int)
        vi. points (tuple of (pair, points))
        vii. vulnerabilities (ns, ew)
        viii. dealer (nesw)
        ix. status (bidding or playing)
        x. boardIndex (boardNumber – 1 for easier calculations)
        xi. currentRound = list of tuples
        xii. activePlayerIndex = 'nsew' index
    b. methods
        i. __init__
            1. boardNumber
        ii. makeDeck (creates a deck of 13 cards)
        iii. dealDeck(randomly assigns 13 cards to each player)
        iv. locateHand(assigns location to each card given hand position)
            1. dict(key=position, values = tuple(x,y) of hand center)
        v. playCard
        vi. getWinner(returns position who won the round based on currentRound)
        vii. nextTurn
            1. currentRound.append(card)
            2. moveCurrentPlayer to next player
            3. if currentRound >= 5, add current round to played (as tuple?) and set the first character of current round to the winner (simultaneously → because getWinner relies on currentRound)
        viii. isBoardEnd(True if board has ended)
4. Game
    a. attributes
        i. board (current Board)
        ii. pastBoards (list of Boards, index should equal boardNumber – 1)
        iii. ~ewTotalPoints (int – will change if we can get tournaments working)
        iv. ~nsTotalPoints (int – will change if we can get tournaments working)
        v. players (dict to position)
    b. methods
        i. __init__
            1. playerDict
        ii. isGameEnd(True if game has ended)

5. Player
    a. attributes
        i. username
        ii. password
        iii. profile pic
    b. methods
        i. makeGuestPlayer
6. Bid(button)
    a. attributes
        i. trump
        ii. number
    b. methods
        i. __init__
            1. trump
            2. number
        ii. __repr__
            1. number + trump

Animation

7. Controller (App)
    a. attributes
        i. player
        ii. screen (what type of screen is being displayed)
        iii. buttonDict (dict of list of buttons on the screen, key = screenName) → remember to remove when screen changes
            1. welcome
                a. register (screen=register)
                b. login (screen=login)
                c. play as guest (loginComplete, player=makeGuestPlayer)
            2. log in = []
            3. menu
                a. play solo (screen = gameplay)
                b. settings
                c. create private table
                d. play with my partner
                e. history
        iv. passwordsDict
        v. history
    b. functions
        i. appStarted
        ii. loginComplete
            1. mode=menu

2. player=player

iii. passwordUnverified (returns False password syncs with , else: returns error message for password or username)
    1. username

iv. mousePressed
    1. for button in buttonDict[app.mode]: if button.isPressed(event.x, event.y): button.function(app)
    2. mode gameplay: if playing: for card in hands[activePlayer]: if card.isPressed:
        a. nextTurn
        b. if isBoardEnd set game board to new board, add game to history
        c. if isGameEnd app.mode==menu, add game to history
    3. mode gameplay: if bidding: for bid in board.bidOptions: if bidPressed
        a. nextBid
        b. if bidEnded: status = p

v. mouseMove
    1. for button in buttonDict[app.mode]: if buttom.isHover (event.x, event.y): buttom.effect(button)

8. View
    a. function
        i. redrawAll
            1. drawBackground(app.mode)
            2. mode welcome: drawWelcome
                a. drawTitle
                b. drawGraphics
            3. mode login: drawLogin
                a. drawLoginUsername (text instruction + textbox)
                b. drawLoginPassword (text instruction + textbox)
                c. drawEnter
                d. if error == True: drawErrorMessage
            4. mode register: drawRegister
                a. drawInputUsername (text instruction + textbox)
                b. drawInputPassword (text instruction + textbox)
                c. drawInputProfilePicture
                d. drawRegister
            5. mode menu: drawMenu
                a. all buttons drawn with part 7
            6. mode gameplay: drawGame
                a. drawHands
                    i. drawCard for hands[positions] in ‘nswe’
                b. if app.game.status == ‘b’ and activePlayer==yourPlayer: drawBidOptions

    c. drawSidePanel
        i. drawBidHistory
        ii. drawLastTrick
        iii. drawNavigation
7. for button in buttonDict[app.screen]: drawButton

For the bot, see algorithmic plan

Algorithmic Plan

The main algorithmic complexity for this project will come from a minimax alpha beta pruning algorithm which is used for the card playing section of the game. As bridge is a stochastic game of incomplete information, the minimax algorithm will be altered to suit the less than ideal nature of the game.

Minimax algorithm

Based on pseudocode from (https://en.wikipedia.org/wiki/Minimax) and some interpretation by me, I will implement the minimax algorithm this way:

```
def minimax (node, depth, alpha, beta, isMaxPlayer):
# node will be an OOP class containing a list of nodes in node.children
if depth == 0 or node.children == []:
      return heuristic(node)
if isMaxPlayer:
      value = float('-inf')
      for child in node.children:
            value = max(value, minimax(child, depth - 1, alpha, beta,
False))
            if value >= beta:
                  break
            alpha = max(alpha, value)
      return value
else:
      value = float('inf')
      for child in node.children:
```

```
            value = max(value, minimax(child, depth – 1, alpha, beta,
True))
            if value <= alpha:
                  break
            α = max(alpha, value)
      return value
```

Modifications for Bridge

As minimax is typically used in two-player perfect-information win/lose/draw games, some modifications must be made. Accomodations based on information from (https://en.wikipedia.org/wiki/Computer_bridge#Comparison_to_other_strategy_games) are discussed below

- → Bridge is played with four players, but as each pair plays as a team, from a game theory perspective we can consider this a two-player constant-sum game since each pair competes for 13 tricks. It is trivial to change a constant-sum game into zero-sum. While the goal in bridge may concern risk management more than getting maximum tricks (depending on contract and tournament), for our purposes, we will ignore those externalities.

- → Bridge is a stochastic game of imperfect information, but we can account for that using Monte Carlo experiments. Using the information provided by the play and bidding, we can generate representative samples and run a double dummy evaluation minimax algorithm on it. One significant way to improve the algorithm would be to improve the samples provided by inferring more information from the bidding/play.

- → There are a couple other modifications that can be made to improve the algorithm due to certain unique characteristics of the game. Bridge is scored incrementally which provides natural lower and upper bounds for alpha-beta pruning and sure winners can be easily calculated in certain situations to further improve pruning. Furthermore, as some cards are essentially equal in value, equivalence classes can be exploited to improve hit rate.

Bidding

The algorithm for bidding is far simpler than the cardplay, but there are different nuances. One key aspect of the program is an evaluator which judges whether the hand should be forced to game or slam. There are a couple possibilities for the evaluator function.

- → The simplest algorithm mimicks the human evaluation method which assigns points to honour cards and unusual distributions (AKQJ = 4-3-2-1, void-singleton-doubleton = 3-2-1)

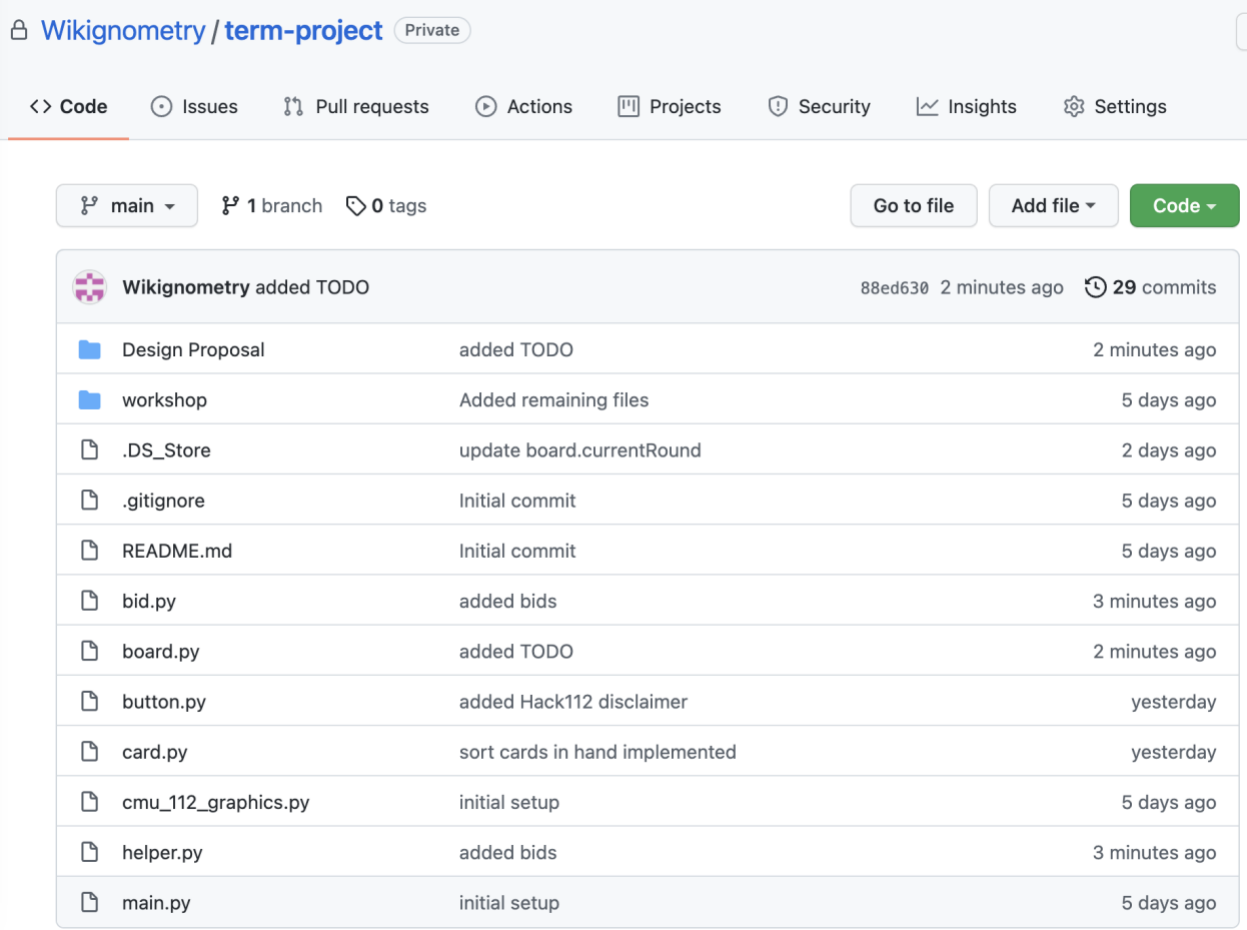
→ A better algorithm involves Binky points which looks up an exhaustive table for every pattern and distribution.

Timeline Plan

SUN	MON	TUE	WED	THU	FRI	SAT
	01	02	03	04	05	*06*
07 classes and methods written + tested	08	09	10 gameplay animation complete	11	12 **TP0 submitted (internal deadline)**	*13* (TP0 Due) standard heuristics complete
14	15 standard minimax complete	16	17 **TP1 submitted (internal deadline)**	18 (TP1 due) minimax with a/b pruning complete	19 —busy— (minimal work on TP)	*20* —busy— (minimal work on TP)
21 —busy— (minimal work on TP)	22 **TP2 submitted (internal deadline)**	23 (TP2 due)	24 —travel—	25 add appropriate UI additions	26	*27* —travel—
28	29 Additional features complete	30 **TP3 submitted (internal deadline)**	1 (TP3 due)			

Version Control Plan

Code is backedup on github on a term-project repository. I use the github desktop app linked to vs code to commit my files to the remote repository.



Module List

None required

TP3 Update

Since TP2 I've added:

- sockets (to play with a single partner)
- more UI (splashscreen etc.)
- error handling (especially for sockets timeout)