# 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](https://www.bridgebase.com/doc/gib_system_notes.php)

#### UI:

* BridgeBase menu

Graphical user interface, text, application

Description automatically generated

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

Diagram

Description automatically generated

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

A picture containing text, sign, clipart

Description automatically generated

* sounds for card playing
* buttons for bidding (expands when you click on the number)

A picture containing calendar

Description automatically generated

* Also has a hand diagram feature

Qr code

Description automatically generated

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

Text

Description automatically generated

* Hand history (with MP results and replays)

A screenshot of a computer

Description automatically generated with low confidence

### 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](https://www.funbridge.com/bridge-artificial-intelligence).

#### UI:

I like their UI

* really nice menu design

A picture containing graphical user interface

Description automatically generated

* cards represented in hand image (I don’t like this 🡪 unclear)

A picture containing text, queen

Description automatically generated

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

Graphical user interface, application

Description automatically generated

* explanation of bidding

Graphical user interface, application

Description automatically generated

* shows bidding on the side

Graphical user interface, application

Description automatically generated

* card play overlaps

A picture containing application

Description automatically generated

* skins are changeable

A picture containing diagram

Description automatically generated

### 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

A screenshot of a phone

Description automatically generated with medium confidence

* 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)
   1. attributes
      1. location (tuple of x,y or None)
      2. width
      3. height
      4. action (function once button is activated)
         1. app
   2. method
      1. \_\_init\_\_
         1. location
         2. (width, height)
         3. action
      2. 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)
      3. isHover (returns True if mouse moved inside the button)
         1. event.x (mouseMoved)
         2. event.y (mouseMoved)
2. Card (button)
   1. attributes
      1. suit (nswe)
      2. number (int)
      3. color (red or black)
   2. methods
      1. \_\_init\_\_
         1. suit
         2. number
      2. \_\_repr\_\_ (prints as ‘[number] [suit]’)
      3. getCardColor (changes self.color to suit color)
      4. \_\_lt\_\_ (allows cards to be sorted)
      5. moveCard
3. Board
   1. attributes
      1. hands
         1. (dict, key=position+‘played’, value=list of cards)
         2. hands[played] = list of list of lead + cards in round
      2. bids (list of tuples (position, Bids) in order)
      3. bidOptions (available
      4. ewTricks (int)
      5. nsTricks (int)
      6. points (tuple of (pair, points))
      7. vulnerabilities (ns, ew)
      8. dealer (nesw)
      9. status (bidding or playing)
      10. boardIndex (boardNumber – 1 for easier calculations)
      11. currentRound = list of tuples
      12. activePlayerIndex = ‘nsew’ index
   2. methods
      1. \_\_init\_\_
         1. boardNumber
      2. makeDeck (creates a deck of 13 cards)
      3. dealDeck(randomly assigns 13 cards to each player)
      4. locateHand(assigns location to each card given hand position)
         1. dict(key=position, values = tuple(x,y) of hand center)
      5. playCard
      6. getWinner(returns position who won the round based on currentRound)
      7. 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)
      8. isBoardEnd(True if board has ended)
4. Game
   1. attributes
      1. board (current Board)
      2. pastBoards (list of Boards, index should equal boardNumber – 1)
      3. ~ewTotalPoints (int – will change if we can get tournaments working)
      4. ~nsTotalPoints (int – will change if we can get tournaments working)
      5. players (dict to position)
   2. methods
      1. \_\_init\_\_
         1. playerDict
      2. isGameEnd(True if game has ended)
5. Player
   1. attributes
      1. username
      2. password
      3. profile pic
   2. methods
      1. makeGuestPlayer
6. Bid(button)
   1. attributes
      1. trump
      2. number
   2. methods
      1. \_\_init\_\_
         1. trump
         2. number
      2. \_\_repr\_\_
         1. number + trump

#### Animation

1. Controller (App)
   1. attributes
      1. player
      2. screen (what type of screen is being displayed)
      3. buttonDict (dict of list of buttons on the screen, key = screenName) 🡪 remember to remove when screen changes
         1. welcome
            1. register (screen=register)
            2. login (screen=login)
            3. play as guest (loginComplete, player=makeGuestPlayer)
         2. log in = []
         3. menu
            1. play solo (screen = gameplay)
            2. settings
            3. create private table
            4. play with my partner
            5. history
      4. passwordsDict
      5. history
   2. functions
      1. appStarted
      2. loginComplete
         1. mode=menu
         2. player=player
      3. passwordUnverified (returns False password syncs with , else: returns error message for password or username)
         1. username
      4. 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:
            1. nextTurn
            2. if isBoardEnd set game board to new board, add game to history
            3. if isGameEnd app.mode==menu, add game to history
         3. mode gameplay: if bidding: for bid in board.bidOptions: if bidPressed
            1. nextBid
            2. if bidEnded: status = p
      5. mouseMove
         1. for button in buttonDict[app.mode]: if buttom.isHover (event.x, event.y): buttom.effect(button)
2. View
   1. function
      1. redrawAll
         1. drawBackground(app.mode)
         2. mode welcome: drawWelcome
            1. drawTitle
            2. drawGraphics
         3. mode login: drawLogin
            1. drawLoginUsername (text instruction + textbox)
            2. drawLoginPassword (text instruction + textbox)
            3. drawEnter
            4. if error == True: drawErrorMessage
         4. mode register: drawRegister
            1. drawInputUsername (text instruction + textbox)
            2. drawInputPassword (text instruction + textbox)
            3. drawInputProfilePicture
            4. drawRegister
         5. mode menu: drawMenu
            1. all buttons drawn with part 7
         6. mode gameplay: drawGame
            1. drawHands

drawCard for hands[positions] in ‘nswe’

* + - * 1. if app.game.status == ‘b’ and activePlayer==yourPlayer: drawBidOptions
        2. drawSidePanel

drawBidHistory

drawLastTrick

drawNavigation

* + - 1. 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 | 08 | 09 | 10 | 11 | 12 | 13 |
| classes and methods written + tested |  |  | gameplay animation complete |  | **TP0 submitted (internal deadline)** | (TP0 Due)  standard heuristics complete |
| 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|  | standard minimax complete |  | **TP1 submitted (internal deadline)** | (TP1 due)  minimax with a/b pruning complete | ––busy––  (minimal work on TP) | ––busy––  (minimal work on TP) |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| ––busy––  (minimal work on TP) | **TP2 submitted (internal deadline)** | (TP2 due) | ––travel–– | add appropriate UI additions |  | ––travel–– |
| 28 | 29 | 30 | 1 |  |  |  |
|  | Additional features complete | **TP3 submitted (internal deadline)** | (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.

Graphical user interface

Description automatically generated

## 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)