Reinforcement Learning and Monte Carlo Tree Search poker strategies using Decima: Poker Simulation Software

By Chris Kottmyer and Shahin Shirazi

Abstracts:

We use Decima: A Poker Simulation Framework to develop two types of poker strategies: one using Reinforcement Learning and the other Monte Carlo tree search algorithm (MCTS). We compare both Monte Carlo Tree search and Reinforcement Learning strategies to our previously developed strategies: smart and conservative players. We discover that the Monte Carlo Tree search player plays poker conservatively in a similar fashion to our previous conservative player, but note that he plays a wider variety of hands. Monte Carlo Tree search player only performs better in 6-player poker games compared to conservative player. Monte Carlo tree search player performs better in 5 and 6-player games compared to the smart player. We divide our Reinforcement learning algorithm into two modules, first module named Simple Learner Player which learns how to play poker by observing its own hand and learning the winning probability of his hand by looking at the first two cards. This module outperformed both Always Call and Always Raise players, however it didn't beat more advance players that used Monte Carlo simulation (more advance players that we test against include Smart Player, Gamble By Probability Player and Calculated Player). Second module is more advanced and called Aware Learner Player, this player not only learn how to play poker by observing his hand and calculating the probability of winning of his hand by the first two card. He also takes a note of each game and how each opponent plays. Our Aware Player learn which player is more conservative and only play the good hands and which player is more aggressive that plays different type of hands. After playing our Aware Learner player against Always Call, Always Raise and Smart player we noticed that our Aware Learner picks very conservative approach and fold most of the hands to minimizes his lost for first few thousands of hands. After other two players (Always Call and Always Rise players) lost the game he starts to change his strategy and slowly switches to more aggressive betting. We saw that our Aware Player was able to take about $50K from Smart Player after playing 30,000 hands, then lost all those earning and switched to more conservative betting before gaining another 200K. Based on this trend we expect that our Aware Learner Player eventually beat the smart player, but we didn't have time and resource to proof this as running 100K of simulation already is expensive enough and took more than a day.

Note: This project is a continuation of mini-project 1. We borrowed some charts and sections from mini-project 1 when the content does not vary. Examples being installation of our poker simulation software and explaining how the simulation software functions. We borrow charts from mini-project 1 analysis section so that we can directly compare them to the newer players. We believe that providing this chart enriches our overall analysis.

Background – Description

"The development of probability theory in late 1400s was attributed to gambling; when playing a game with high stakes, players wanted to know what the chance of winning would be" [1]. Like most gambling games, probability plays an important role in 7 card Poker. There are 133,784,560 ways to draw a given hand out of 52 cards in standard deck. Using simple probability formula, we can calculate the probability of getting specific hand. For example, there are 4,327 different ways to draw a royal flush, therefore one would expect to draw this hand once every 30,940 draws, or 0.0032% of the time.[1] Although calculating probability of each hand is a simple and important factor in poker games, it is not enough. Probability of getting a specific hand for any specific player constantly changes depending on their current hand, number of players involved, how many cards have been dealt, etc. Using math to calculate the probability of every possible hand at any given moment during the game can be cumbersome. Simulation can help us to evaluate different scenarios without going through detailed probability theory to get the answer. The caveat to this solution is that if we don't run enough trials, we may not get the correct answer. To avoid this risk, we checked the probability of different hands from our simulation against actual probability for the given hand using probability theory.

Definitions:

Win probability: Calculated probability of winning a hand based on current hand and community cards. This probability is calculated by running 100 iterations of Monte Carlo simulation of the players current hand, community cards and number of opponents playing. It can be summarized as the percent of wins a player received after playing the current hand and community cards 100 times (ties are counted as wins).

Equal chance probability: Probability of choosing a player at random to win the game. This is equivalent to rolling a N-sided dice to assign winning position. Where N is the number of players.

Community cards: cards that placed on table and everyone can incorporate in his/her hand. Community cards are not used during pre-flop and during post flop the dealer begins by showing 3 community cards. He/She then flips one additional card for the next two phases for a total of 4-community and 5-community cards respectively.

Poker phase: We define a poker phase as either: pre-flop, 3-community card, 4-community card or 5-community card portion of a poker game. A phase is composed of 3 betting rounds where each player can make a bet assuming they did not fold in previous betting rounds.

Betting round: We define a betting round as a round where all non-folded players have an opportunity to fold, check, call or raise the bet. The 3rd betting round in our version of poker does not allow for raises.

Active Player: A poker player that has not folded during a poker game. Active players continue to make bets until they fold and become inactive. If an active player reaches the end of the game, his hand is scored and a winner is determined. A poker game winner is always an active player.

Player Strategy: A set of rules that defines how a player makes a bet. Player strategies can be shared by multiple players.

Player: An agent at a poker table that uses a player strategy to make decisions. Multiple players can exist at a table using the same or varying strategies. Each player is independent of other players retaining his own balance and receiving his own set of cards.

Scenario: A set of simulations that have a common theme. A good example would be testing a single strategy by running a set of 5 simulations: 2,3,4,5 and 6 player tables to see how the agent performs when faced with different number of opponents. In our analysis, we run two scenarios to test two different player strategies: Conservative and Smart player.

Pot: The sum total of all player bets that a player can win at the end of a poker game. **Background – Reinforcement Learning**

Reinforcement Learning (RL) is an unsupervised machine learning that is used in many disciplines, such as game theory, operation research, multi agent systems, swarm intelligence, etc. In this method intelligent agent takes actions in the environment to maximize its reward. The environment is usually stated in the form of a Markov decision Process (MDP) and is different than classical dynamic programming. Reinforcement learning doesn't assume knowledge of exact mathematical formula and find the solution by finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge).

The basic reinforcement learning interacts with environment by taking different actions, it then checks the reward or penalties (defined as negative reward) and adjust its future actions accordingly. Reinforcement learning is well-suited for the problems that includes a long-term vs short-term reward trade-off such as robot control, backgammon, checkers and Go (AlphaGo). [3]

Background – Monte Carlo Tree Search

Monte Carlo tree search algorithm (MCTS) is a heuristic search algorithm commonly used in board games, video games and card games [3]. It has been used in games such as go, chess, bridge, poker as well as the video game: Total War: Rome 2 [3]. In 2015, Google's Deepmind group developed a combined MCTS and reinforcement learning algorithm that defeated Lee Sedol, the reigning champion of the board game Go [3].

MCTS algorithms has gained prominence in making decisions in perfect information games. MCTS involve simulating a set of player moves and using the results of the simulation to update all proceeding moves win probability. An implicit assumption of MCTS is that specific moves have a large impact on win probability. A good example of this is chess where a chess master makes better overall moves then a novice, which almost always leads to his victory. When hidden information significantly impacts the odds of winning a game, this implicit assumption breaks down. In the extreme case, hidden information completely determines the outcome of a game independent of what moves each player makes. A naïve MCTS implementation would at best have a random chance of winning in this circumstance.

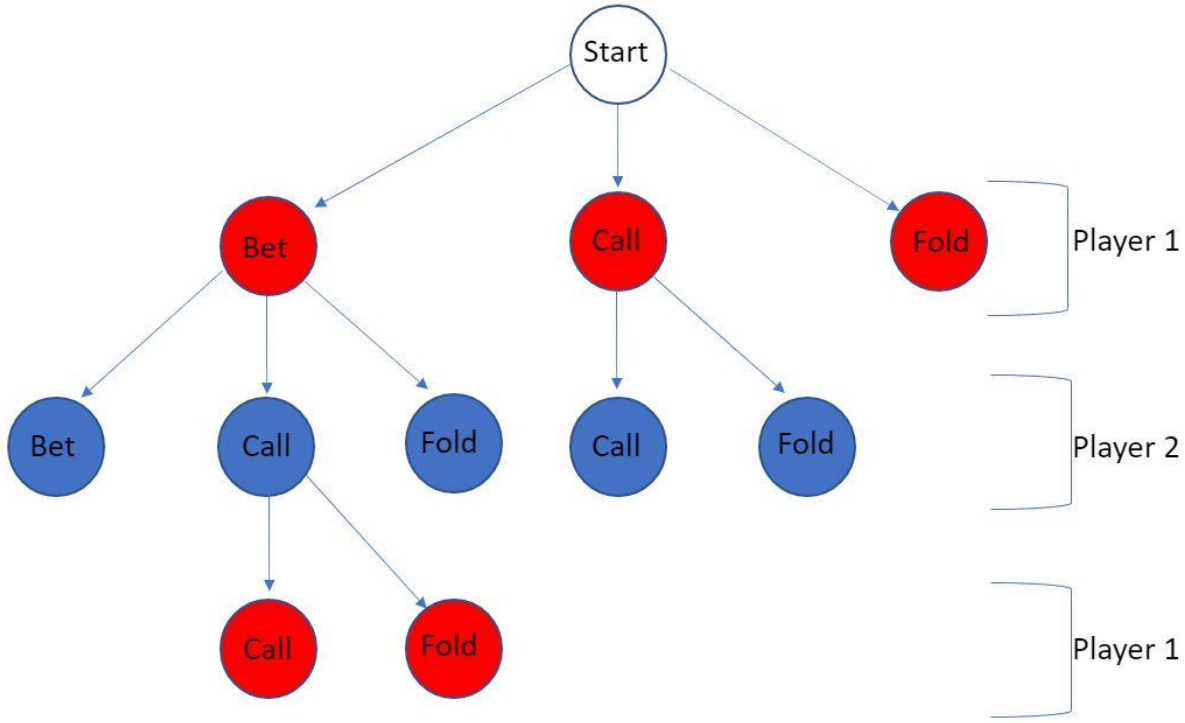
Poker is an imperfect information game. Each player is dealt a 2-card pre-flop hand, which significantly determines his chance of winning. These 2-cards are hidden to opponents. Poker also has 3 phases

involving a set of community cards. When these community cards are dealt, they can have a signficant impact on the players behavior. For poker, this becomes even more difficult when you look at the number of 3,4 and 5 community card combinations. To get around this problem, researchers have come up with a few methodologies. One possible method is to simulate a distribution of outcomes for opponents [5]. One common technique for acquiring a distribution is to look at real world poker data [5]. Alternative method is to use the reinforcement learning concept of self-play [6]. In this situation, each player has his own MCTS and both MCTS algorithms interact with each other to determine opponent behavior [6].

Background – Monte Carlo Tree Search Structure

MCTS is an algorithm represents player moves as a tree. The root of the tree is the start of the game. No player moves have been made at this point. Each subsequent level of the tree represents a player. The order of levels is based on the games turn order with the 2nd level representing the first players first turn and the 3rd level representing the next players response to the first player. Nodes within a level represent moves a player can make that turn. Child nodes are direct responses to a parent move (node). Paths in the tree represent a sequence of legal moves made by the players at each level. MCTS creates one new move/node per iteration.

Below, we provide an example image of an MCTS tree. This example has 2-players and shows the result of running the MCTS algorithm for 10 iterations. Player 1s actions are on even levels and his nodes/moves are colored in red. Blue nodes represent player 2. The root node is labeled start and is colored white.



Figu.1.Example image of MCTS Tree

Background – Monte Carlo Tree Search Algorithm

MCTS algorithm builds the above tree using four steps: selection, expansion, simulation and back-propagation [3,4]. It iterates over these 4 steps for a fixed amount of time for example half a second. The steps are outlined below:

Selection: Each iteration begins at the root node. It searches the roots children for a node that has a missing child (this can include root itself). When MCTS encounters a node with no missing children, it picks a child node to search using the upper confidence bound algorithm shown below:

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Variables are:

wi – child node's wins.

ni - child node's total games played.

Ni – parent's total games played.

c - a constant that when increased causes the algorithm to explore more nodes instead of exploiting nodes with good win probability: wi/ni.

It's worth noting that wi/ni represents a node's win rate and the term with a constant encourages MCTS to explore less played nodes.

Expansion: One of the selected nodes children is added to the tree. This represents a missing response to that move.

Simulation: A Monte Carlo simulation is run. The Monte Carlo simulation uses all moves up to and including the missing move as the games initial position. It then selects moves at random and records the number of wins and total games played. Traditionally only 1 game is played. No nodes are added as a result of this simulation.

Back-propagation: The wins and totals for the Monte Carlo simulation are added to the child node and all ancestors of the child. This continues until the root node is reached.

A pictorial representation of the process can be seen below [3]:

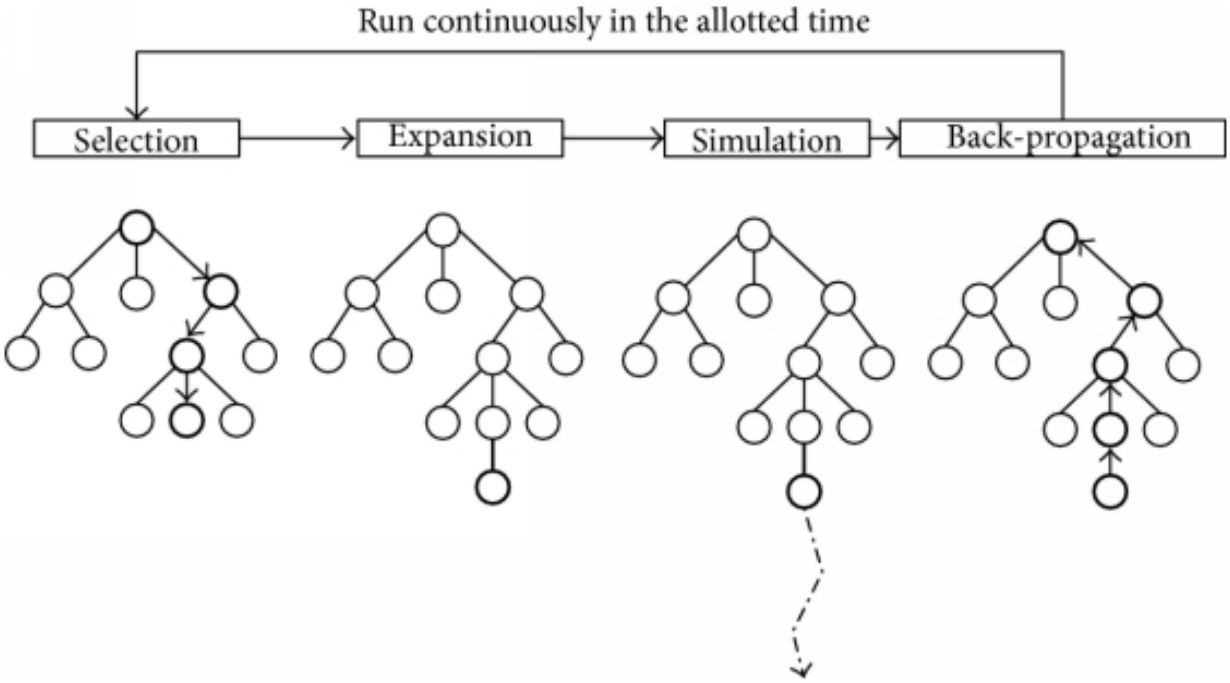


Fig.2. Representation of MCTS algorithm

Below, we have a representation of the back-propagation algorithm. Each node runs its own Monte-Carlo simulation. The wins and total games played are recorded in each ancestor node. This means that the total games played for a node include all its children plus itself. Wins are recorded based on the player representing that level. So by player 2 is recorded as a win for player 1. This guarantees that both the player and his opponents best moves are represented in the tree. Each node's win to total games played ratio represents that player's probability of winning the game.

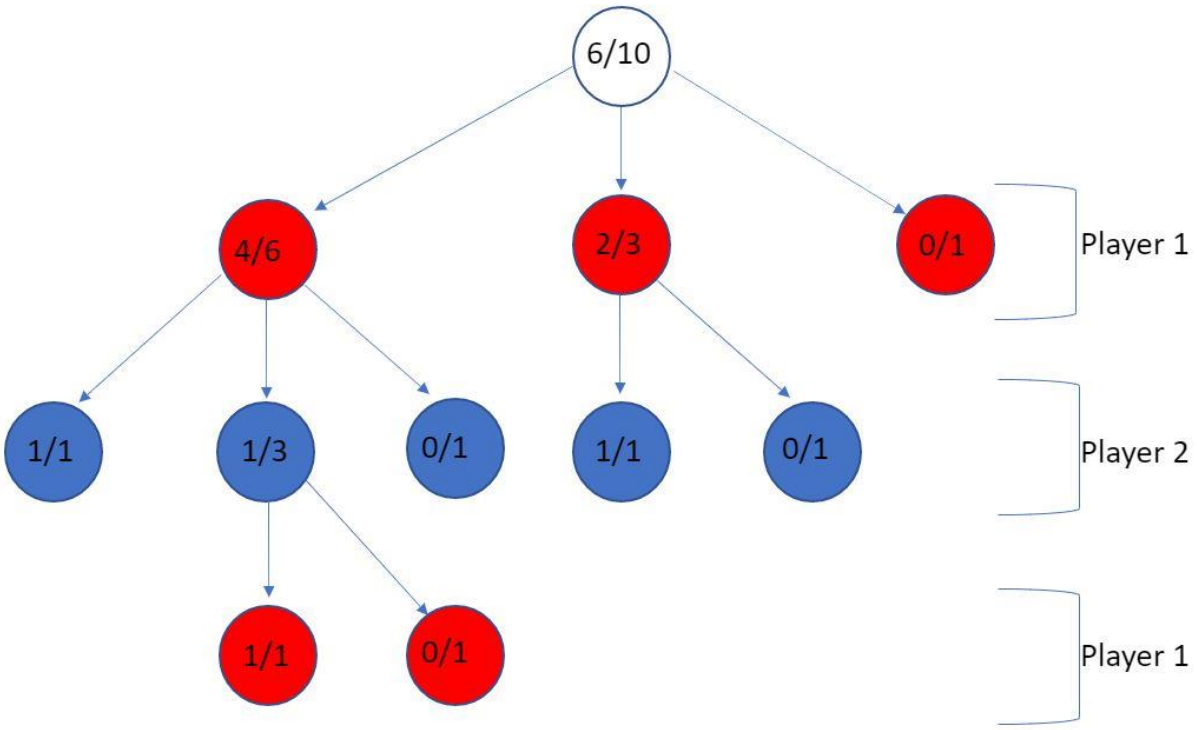


Figure 3. Back-propagation schema with win probability

To use the MCTS tree, start at the root. Follow the moves currently made in the game until you get to a node representing your opponent's last move. Call this the decision node. Children of this decision node represent responses to your opponent's last move. Ideally, you should choose the child node with the highest probability of winning. In the above example, player 1 has a 4 in 6 chance of winning if he bets. Player 2s best move in response to player 1s bet is to raise since that node has won 1 out of 1 total games and represents the best probability of winning the game.

In the below image, we replace player 2s probabilities with question marks. Poker has hidden information in the form of 2-cards dealt to each player. The probability of player 2 winning is highly dependent on his hand. A common approach to handling this uncertainty is to use a probability distribution to represent these nodes instead of relying on a single simulation.

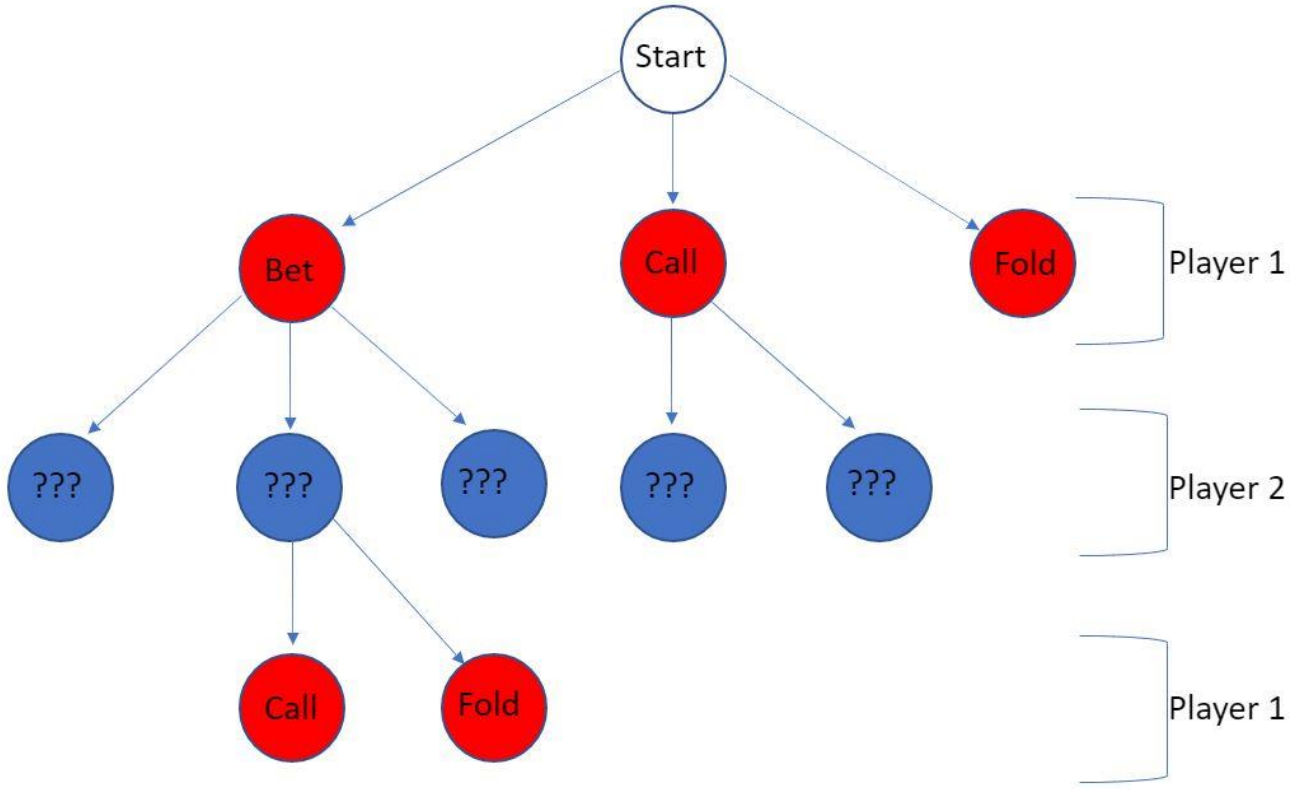


Fig.4. Back-propagation schema without winning probability for player 2

Background – Decima

Decima: A Poker Simulation Framework was developed by Chris Kottmyer and Shahin Shirazi to rapidly prototype poker strategies and test them in 2-6 player games. Decima represents poker strategies using python classes that inherit from a GenericPlayer class. GenericPlayer class imposes few restraints on designing poker strategies. The only requirements are that it implements a bet_strategy method and only uses self.fold_bet, self.call_bet or self.raise_bet once within the bet_strategy method. Setting up a poker scenario involves modifying a single python variable. A benefit of Decima is that it has parallel processing built into it. Allowing us to use all CPUs simultaneously for simulating poker scenarios.

Installing and Running Decima:

Running Decima: All you need to run Decima is Python 3. Python 3.7 was used for this simulation; however other version of Python 3 should be compatible with our code. Python 3 is typically installed on Linux and MacOSX systems. For Windows users, Python 3 can be download from https://www.python.org/downloads/. All the libraries that were utilized in Decima come pre-packaged with Python 3 to simplify installation.

The source code for Decima can be downloaded from our GitHub account: https://github.com/Silber8806/PokerPro. Once downloaded, go to the directory: source_code and type python poker.py to run our simulation. This can take upwards of 2 hours. poker.py runs a total of 20

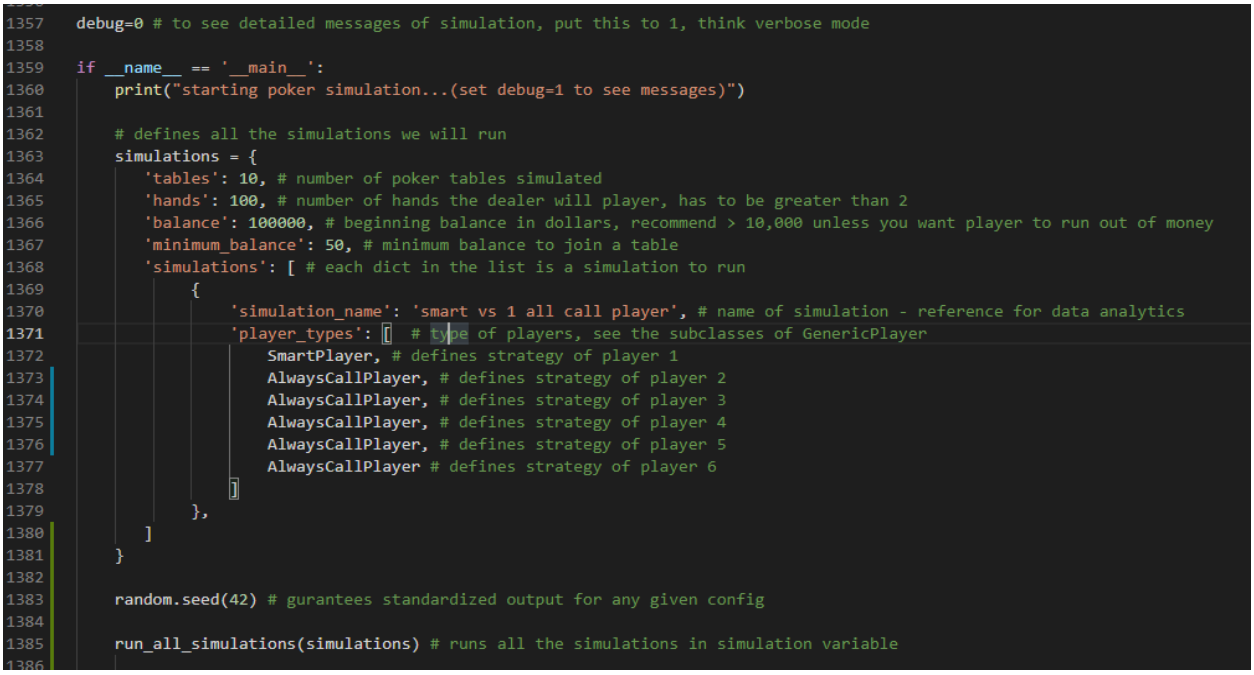
simulations with 30 replications (poker tables each) and 100 poker games per table. After Decima is finished running, a folder: source_code/data is created with 3 types of files. These files are: poker_balances_[table_id].csv, poker_hands_[table_id].csv and poker_info_[table_id].csv where [table_id] represents a unique integer identifying a poker table. These files store information about player balances, player bets and high-level table information.

Note: re-running the simulation does not delete information in source_code/data folder. We suggest deleting the source_code/data folder after each run.

Warning: Monte Carlo Tree search player is extremely memory intense. We tested out a simulation with 6-player poker game, 30 tables and 100 hands on a high-end gaming laptop with 16 gb of ram and 6 CPU cores. It ran out of memory, caused an operating system level crash and forced us to reinstall python. We did eventually get it to run by limiting the number of scenarios analyzed at one time. We suggest for computers with less than 16 gb of ram to lower the table and hands settings when simulating monte carlo tree search player.

Configuring Decima and changing simulation parameters:

Poker.py is the only file required to run Decima. It contains two sections that user can manipulate to change the simulators behavior. First is the simulation variable, which tells Decima what simulations to run (Fig.5), how many poker tables to run for each simulation (tables), how many poker games to play for each table (hands), the starting balance of each player (balance) and the minimum balance to join a game (minimum balance). Top four parameters (tables, hands, balance, minimum_balance) apply to all simulation scenarios. For example, if you set tables to 100 and hands to 50, it will run 100 poker tables and 50 games per poker table for a grand total of 5,000 poker games per simulation. The simulations key is a list of simulations to run. A simulation in this list, has two keys: simulation_name which is a human friendly name for the simulation and player_types which includes list of player strategies to use in the simulation.



```
debug=0 # to see detailed messages of simulation, put this to 1, think verbose mode

if __name__ == '__main__':
    print("starting poker simulation...(set debug=1 to see messages)")

    # defines all the simulations we will run
    simulations = {
        'tables': 10, # number of poker tables simulated
        'hands': 100, # number of hands the dealer will player, has to be greater than 2
        'balance': 100000, # beginning balance in dollars, recommend > 10,000 unless you want player to run out of money
        'minimum_balance': 50, # minimum balance to join a table
        'simulations': [ # each dict in the list is a simulation to run
            {
                'simulation_name': 'smart vs 1 all call player', # name of simulation - reference for data analytics
                'player_types': [  # type of players, see the subclasses of GenericPlayer
                    SmartPlayer, # defines strategy of player 1
                    AlwaysCallPlayer, # defines strategy of player 2
                    AlwaysCallPlayer, # defines strategy of player 3
                    AlwaysCallPlayer, # defines strategy of player 4
                    AlwaysCallPlayer, # defines strategy of player 5
                    AlwaysCallPlayer # defines strategy of player 6
                ]
            },
        ]
    }

    random.seed(42) # gurantees standardized output for any given config

    run_all_simulations(simulations) # runs all the simulations in simulation variable
```

Fig.5. Selecting simulation variable

Defining Player Strategies to use in Decima:

The second way user can manipulate Decima is by defining poker strategies. To create a new player strategy, first define a Python class that inherits from GenericPlayer and implements the bet_strategy method and then add the class (not instance) to a simulation's player_types list. See Fig 6.

```
# sophisticated player with more complicated strategy.
class SmartPlayer(GenericPlayer):
    def bet_strategy(self,hand,river,opponents,call_bid,current_bid,pot,raise_allowed=False):
        win_probabilty = simulate_win_odds(cards=hand,river=river,opponents=opponents,runtimes=100)
        expected_profit = round(win_probabilty * pot - (1 - win_probabilty) * current_bid,2)
        equal_chance_probability = 1 / float(opponents + 1)
        high_probability_of_win = 1.4 * equal_chance_probability

        if win_probabilty > high_probability_of_win:
            self.raise_bet(100)
            return None

        # if you statistically will make money, call the bid else just fold
        if expected_profit > 0:
            self.call_bet()
        else:
            self.fold_bet()
        return None

# Player that always calls
class AlwaysCallPlayer(GenericPlayer):
    def bet_strategy(self,hand,river,opponents,call_bid,current_bid,pot,raise_allowed=False):
        self.call_bet()
        return None
```

Fig.6. Defining a player strategy.

The GenericPlayer class provides common functionality for all player strategies. This includes interacting with poker games (make_bet method), default action to call/check when a player is the only person not to fold representing an automatic win for the player and 3 methods used to signal a betting decision: fold_bet(), call_bet() and raise_bet(). It also has mechanisms for recording a player's decisions for simulation analysis. A Decima user should avoid modifying this class, but should be aware of what functionality it provides.

Once a user has created a class that inherits from GenericPlayer, it needs to create a bet_strategy method. Bet strategy represents a players decision making process when deciding to fold, check, call or increase the bet (raise). It is typically written in Python in the body of the bet_strategy method. The bet_strategy method needs to implement all the following parameters, which the user is encouraged to use within the bet_strategy method:

- **hand:** represents the two-card hand a player receives during pre-flop round. The hands are Python named tuples: Card(rank, suite) in the form of a list. You can access suite and rank using dot notation: card.rank and card.suite.
- **river:** represents 0,3,4 or 5 community cards. It uses the same Python named tuple: Card(rank,suite) in list format. Cards that have yet to be dealt are represented by None.
- **opponents:** an integer representing the number of opponents still playing the game.
- **call_bid:** an integer representing the amount of money needed to match the current bid.
- **current_bid:** an integer representing the amount of money the player has already bet.
- **pot:** an integer representing the money contributed by all players to the pot or reward.
- **raise_allowed:** a Boolean value indicating if raises or increasing a bet is allowed. When this is set to False raise_bet method will use call_bet under the cover.

To make a bet decision, a user has to call one of these functions once and only once: fold_bet(), call_bet() or raise_bet(raise_amount). The functionality is explained below:

- **fold_bet():** the player folds his hand and doesn't play for the remainder of the game.
- **call_bet(allow_all_in=True):** a player matches the current bid (increases players bet by call_bid) and continues playing the game. allow_all_in deals with the situation where the call_bid amount is greater than his balance. If set to True, the player will use all chips possible in the bet, if set to False the player will fold instead. The default is True.
- **raise_bet(raise_amount, allow_all_in=True):** a player matches the current bid and then increases it by [raise amount] chips. The allow_all_in parameter determines if a player will use every remaining chip to fund this bid or fold instead. This mirrors the call_bet methods allow_all_in parameter. The default is True.

Note, we didn't get around to writing a guarantee that one of the above is called once and only once. This is a feature that will be implemented soon.

Strategies.

Below is a list of poker strategies defined in Decima used in our analysis including a short description of what each strategy accomplishes.

Always Call Strategy- A simple strategy that always calls the bet and plays every single hand.

Always Raise Strategy- This strategy always raises the bet by $10 and plays every single hand.

Smart Player Strategy: In this strategy player raise the bet by $100 if Win Probability is higher than Equal Chance probability by 40%, otherwise Smart Player calculates expected profit (expected profit=Win Probability *Pot -[1-Win Probability]*current bet). If expected profit is positive, it calls the bet otherwise it folds it's hand.

Conservative Player Strategy- Unlike previous strategies player doesn't compare its Win Probability against Equal Chance Probability, instead it uses following rules based on its Win Probability. Fold if its less than 50%, call the bet if it's between 50% and 70%, raise the bet by 1% of his current balance if Win Probability is between 70% and 90%, raise the bet by 2% if it's between 90% and 95%, raise the bet by 5% if it's between 95% and 99% and finally raise the bet by 10% if its Win Probability is above 99%.

Simple Reinforcement Learner - A strategy that learns the probabilities of winning each hands by playing them. In this strategy, player started out by randomly selecting between fold, raise and calling different hands. After player makes the decision for each hand it memorizes the reward (in this case amount gain or lost for selecting specific action.) and next time that he/she receives the same hand, it searches for the action that maximize his reward (or minimize the loss).

Aware Reinforcement Learner – This player uses similar strategies as Simple Reinforcement Learner by memorizing the rewards for each playing hand and optimizing its action to maximize its reward. There are some minor differences between Aware Reinforcement Player (ARL) and Simple Reinforcement Learner (SRL) when it comes to selecting the hand. ARL plays each hand for the first time instead of randomly selecting between fold, raise and call. This allows him to expedite his learning process and minimize the loss by folding a good hand or raising the bad hand. After it learned about the specific hand it is going to select between two actions for fold and Call. If rewards shows that Call action is the optimize action it then look at the ratio of the reward for call to total bets and if it is above specific thresholds, it then raise the bet. ARL also learns from other players actions, it memorizes the probability of winning from other players and adjusts its action accordingly.

Monte Carlo Tree Search Player- This strategy builds an MCTS tree for each possible turn order. It expands the tree if a new pre-flop hand is encountered. Expansion process involves the 4 MCTS steps: selection, expansion, simulation and back-propagation. Once built, a tree can be queried for the best move to respond to an opponent. During the querying process, any missing nodes are added to the tree and if there are less than 100 children it expands the subtree around the decision node. Once the best move is selected, it is vetted. The vetting process depends on the pokers card phase. For pre-flop cards, the player uses the MCTS best move if it's win probability is greater than the average win probability of it's last 20 pre-flop hands. For 3 and 4 community card phase, it uses the MCTS best move if the win probability improved compared to the last card phase. So for 3 community card phase, it checks if it's win probability improved since the pre-flop phase. It also uses the MCTS best move if he has better chance than average of winning the game. For the 5 community card phase, the player uses the MCTS best move if his odds of winning are better than random (equal chance probability). If Monte Carlo Tree search player doesn't use the MCTS best move it automatically folds. If Monte Carlo

Tree search player has greater than 90% chance of winning, he always bets $1,000. If he decides to bet in any other circumstance, he bets $100.00.

Reinforcement Learning Player Strategy Overview:

For the scope of this project, we utilized very simple value iteration to build two different learners. (Simple Learner Player and Aware Learner Player). As any Reinforcement Learning we needed to define environment, actions and reward. Environments for these players was defined as the first two cards in the hand. Actions are simply any of the tree options between call, fold and raise. Finally, we defined each game gain/loss as measurement of reward. Assigning the gain/loss as a reward is consistent with exploitation and exploration idea of reinforcement learning. Player exploits the best action based on historical reward, however having negative value for loss naturally provides exploration options. For example, if Reinforcement learner takes a call or raise action on a good hand and then lose the game by bad luck it's going to fold the hand next time, as it folds the hand it gets negative points till the negative points of losing hand became lower than negative points of calling/raising hand in which the player will switch to calling the hand instead of folding it. The caveat to this method is that if our player calls a good hand and wins the game it keeps calling the hand based on this simple strategy of picking the maximum value. We improved the strategy for our Aware player by implementing the following function, if call action has the maximum value, player going to calculate the proportion of reward for call action to the rewards for all actions. If the ratio is above 70% it will switch to raise otherwise it keep calling the hands. We also improve the first-time action for Aware learner compare to simple learner by making the Aware Learner to play each hand for the first time. This helps to minimize the loss by folding the good hands or raising the bad hands, plus player don't get any gain (learn the hand) by folding the hand for the first time. Last improvement on picking the actions for Aware Player compare to simple player was to only raise the hand if player already won the hand by calling it. This will minimize the loss of raising on the hand with lower winning probability.

Aware Player also learns by observing other players actions and memorizing the ratio of wining for each player. For example, if it encounters a conservative player that only plays the best hand with high chance of winning and notice the ratio of winning the hand to total hand played is above certain thresholds, it learns to fold its hand regardless of the hand, on the other hand if it checks the ratio and notice his opponent plays both good and bad hands then it makes decision based on the quality of his/her own hand.

Both players are using the first two cards to make a decision. We are aware that this is very simplified version of what actually happens in the game of Poker and the proper way would be to use all 7 cards by using policy iteration, however for learning propose and to keep the scope of this project within reasonable timeline we decided to only use the first two cards for decision making.

Reinforcement Learning Player Implementation:

Following you will find the method that we used for our Reinforcement Learner:

bet_strategy(self,hand,river,opponents,call_bid,current_bid,pot,raise_allowed=False): This is a common method for all the players where each player implements different strategies based on the hand, river, opponents action and bids.

Simple Learner Player: This player was defined as a class with the following functions to take an action about each hand, update the reward after each game, and repeat the action for the remaining of the game based on the first two cards. Following you will find the explanation for each method.

simpleLearnerCall(self,hand): This function takes the hand (first two cards), create a

Dictionary_Key, where it stores the ranking of the two cards and add simbole 'Y' if cards are the same suite and 'N' if cards are different suits. For example player receives 10 club and 2 hards

It makes a dictionary_key of '102N' and if it receives 10 heart and 2 heart it creates dictionary key of '102Y'. Next it checks if the hand was played before by checking the value inside the dictionary. If hand shows up for the first-time player picks a uniform random variable, if the value is less than 0.33 it folds the hand, if value is between 0.33 and 0.66 it calls the hand and if value is greater than 0.66 it raises the bet by 20. If the hand has been already played, it looks at the dictionary of the hand to check the win/loss amount for historical actions for this specific hand. Values are stored in a list with 3 components for 'fold', 'call', 'raise' actions. The value of each component is the cumulative summation of the specific actions for those specific hands. In our example, if player gets '102N' for the fifth time and lost $10 by folding it, won $40 by calling it first time and lost $30 by calling it the second time and won $60 by raising it the 4^{th} time the list will be [-10,10,60]. Then it looks at maximum value (in our example maximum value is 60 which is corresponding to raising) and use the action that has the maximum value to raise the hand by $20 . Every time that player gets repeated hand (hand that already been played by Simple Learner player) it runs through the same process and decide whether to fold, call or raise by picking the action corresponding to maximum value in the matrix.

repeat_action(): This function simply looks at the decision that was made after receiving the first two card and repeat that decision for each betting round. For example, if player decide to raise the bet by $20 after receiving the first two cards, this function keeps raising the bet by $20 after dealer deals the river cards for each round. The same thing is true if player decides to call the bet.

update_SimpleLearnerReward(): This function updates the dictionary for each hand after each game is finished. In our previous example after all player raise the bet by $20 for 4 rounds for total of $80, if player lose the game, it will update the previous list of [-10,10,60] to [-10,10,-20] by subtracting $80 from previous balance of $60 for raise action. This means next time that our player receives the same card '102N' it finds the maximum value of 10 which is corresponding to calling the hand.

AwareLearner Player: This player was defined as a class with the following functions to take an action about each hand, update the reward after each game, record opponents winning probabilities and repeat the action for the remaining of the game based on the first two cards. Following you will find the explanation for each method

AwareLearnerCall(self,hand): Very similar to SimpleLearnerCall methods with few minor upgrades. First player going to call each hand for the first time instead of randomly selecting between call, raise and fold for SimpleLearnerCall. It also going to raise the hand which has at least 70% chance winning.

temp_balance_dictionary(self,active_players,temp_dictionary): This function takes number of active players with their balance and return a dictionary of each player with their corresponding net change for the game.

opponent_winning_probability(self,active_players,temp_dictionary): This function takes number of active players and temp_dictionary from previous function and updates winning probabilities for each of the opponents.

max_opponent_probability(self,action_list): simply takes the action list from the previous game for each player and their winning probability and return the maximum winning probability amongst all the opponents.

action_based_on_opponent(self,probability,hand,thereshold=0.7): This function takes the probability which is basically the same as maximum probability from previous function and compare it to the threshold, if maximum probability is above threshold it force the player to fold the hand, otherwise it calls **AwareLearnerCall(hand)** to make the action based on the hand. This function can be improved by letting player pick the optimum threshold through learning the process, however for the scope of this project we decided to define the threshold.

post_game_hook(self): made this function to capture the actions and balance from previous game.

Reinforcement Learning Player Analysis

To bench mark our Reinforcement Learner players we run the following scenarios and used the graph of each player balance against number of hands played for our analysis.

Simple Learner vs Always call players: In this scenario, our simple learner player played against 5 other players with Always Call strategy. Interestingly, our player performed below mediocre for the first 3000 games as it was learning the probabilities of different hands. This player was ranked 5th amongst the 6 total players before it starts learning the strategy. After playing about 10,000 games, it optimized its strategy and outperformed all the other Always Call players.

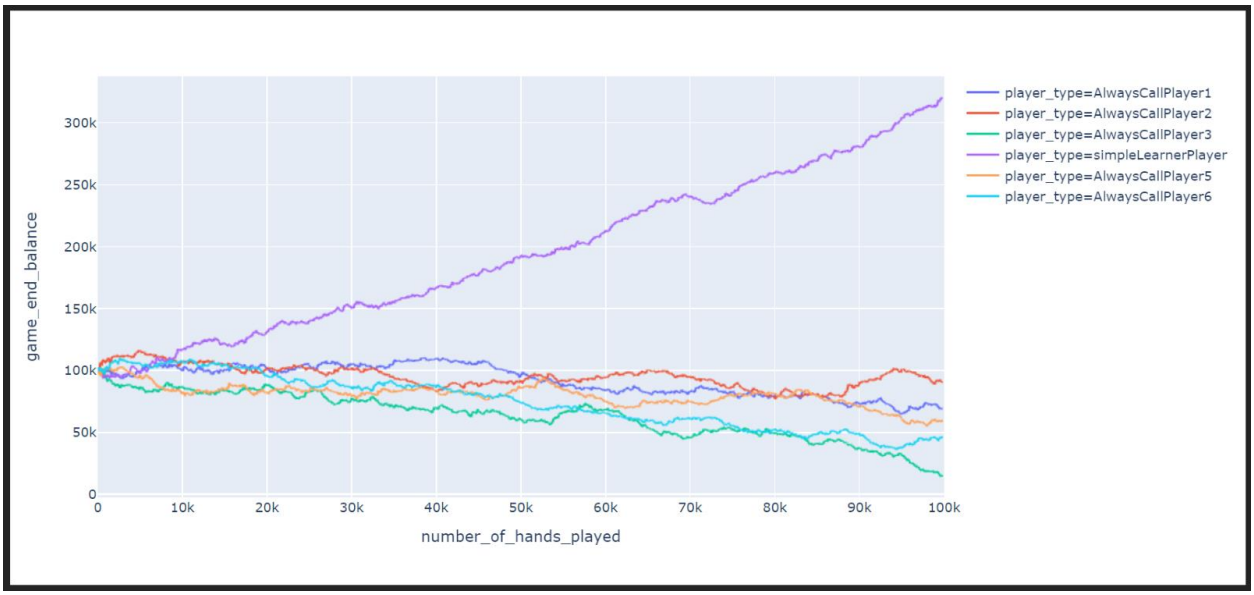


Fig.7. Always Call Players vs Simple Learner Player balance.

Simple Learner vs Always Raise players: The same as previous scenario with the exception that our Simple Learner player was playing against 4 other players with Always Raise strategy. Even though Simple learner player was able to outperform other players after 10,000 games the transition wasn't as smooth as previous scenario. As you can see from the following graph the balance (which is the indication of reward) moves in the logarithmic fashion instead of linear gain. This could be explained by the fact that every time that one of the players lose the game and get out it reduces the reward for the rest of the game by having less players.

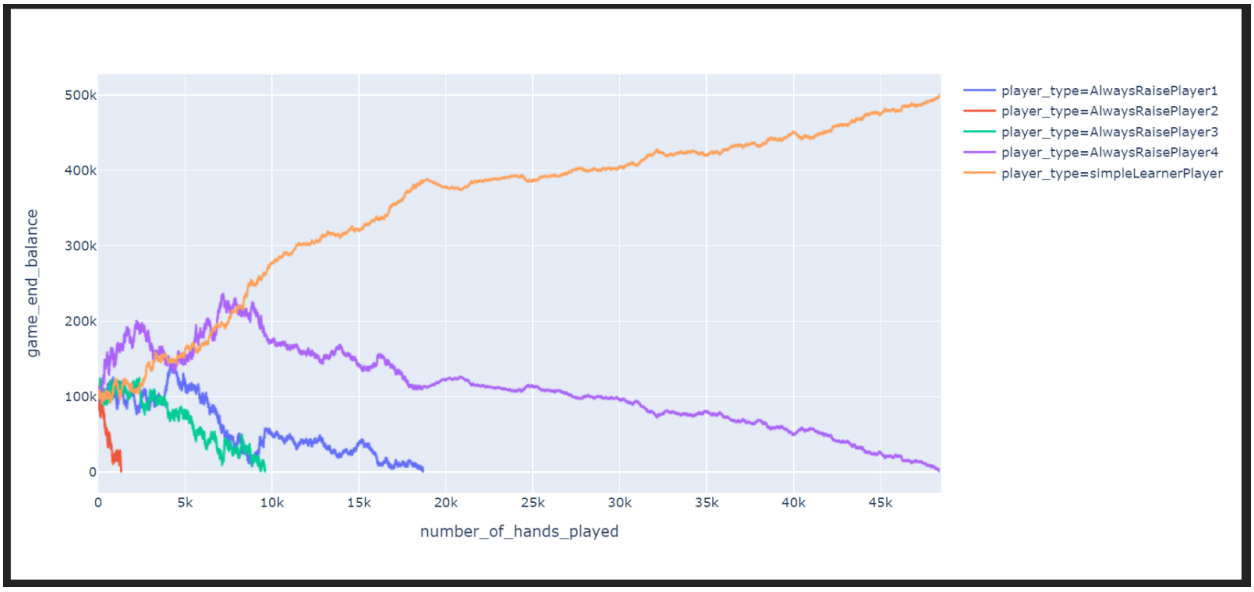


Fig.8. Always Raise Players vs Simple Learner Player balance.

Simple Learner vs other players: In this scenario, we tested our Simple Learner player against other players (Always Call, Always Raise, Smart Player, Gamble By Probability player). Surprisingly, our simple learner player didn't perform well and only outperformed Always Call and Always Raise player. This can be explained by the fact that simple player only calculates the probability of winning for the first two cards vs Smart Player, Calculated Player and Gamble By Probability players that use Monte Carlo simulation to calculate the winning probability after each round.

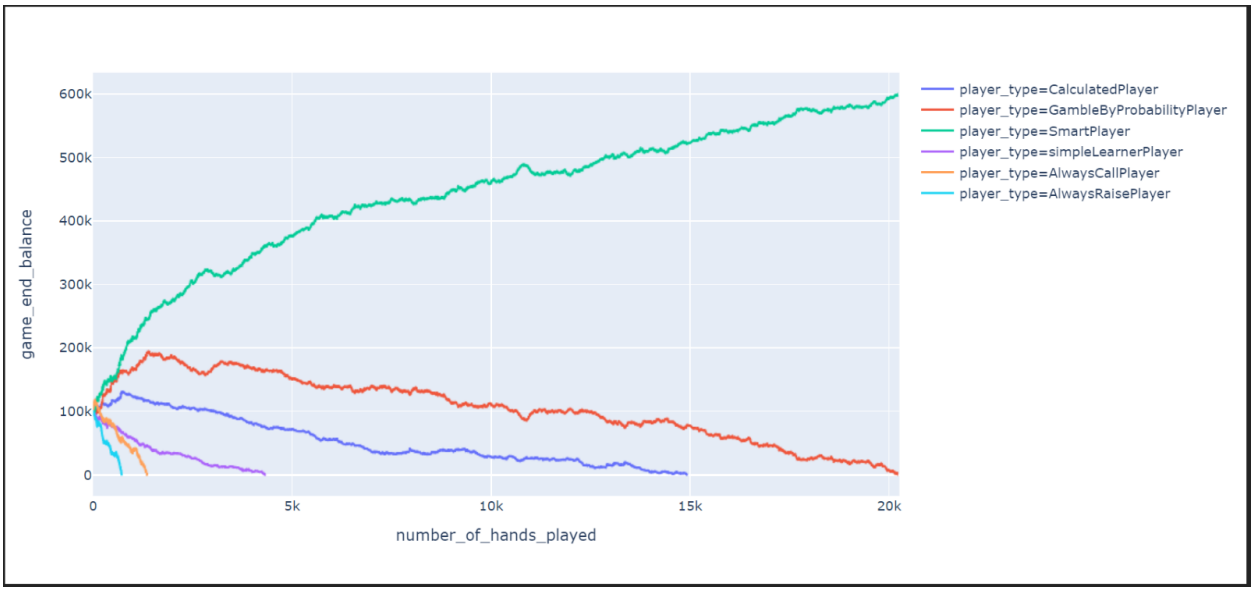


Fig.9. Simple Learner Player vs other players balance.

Aware Learner vs other Players: Finally, we ran our Aware Learner against Always Call, Always Raise and Smart player. It was interesting to observe that our Aware Learner player selected the conservative strategy to minimize it's loses against Smart Player and after 5,000 hand it came to steady states where it kept its balance. This can be explained by the fact that our Aware Learner learned Smart Player has advantage over him and has higher chance to win a game whenever he calls or raises the hand and it simply folded those hands to minimize the losses. After he played about 30,000 hands, he slowly figured out a way on how to play against Smart Player with slowly increasing its balance and then made some aggressive move when he had a better hand, you can see this from the following graph. He started to drop smart players balance by 50k first around 31,000 hands before he had a big lose and got back to conservative mode again. Then he tried this method after 82,000 hand and was able to get the smart player's balance down by almost 200k. We expect that our Aware Learner player continue to use this strategy and reduces smart player balance by bigger and bigger portion before have a big lose and get back to more conservative mode. We expect that in next few jumps it will take all the money from smart player and win the game, however we didn't run this scenario as it takes a long time to run this scenario more than 100K. It took about 24 hours to run 100 simulations for this scenario.

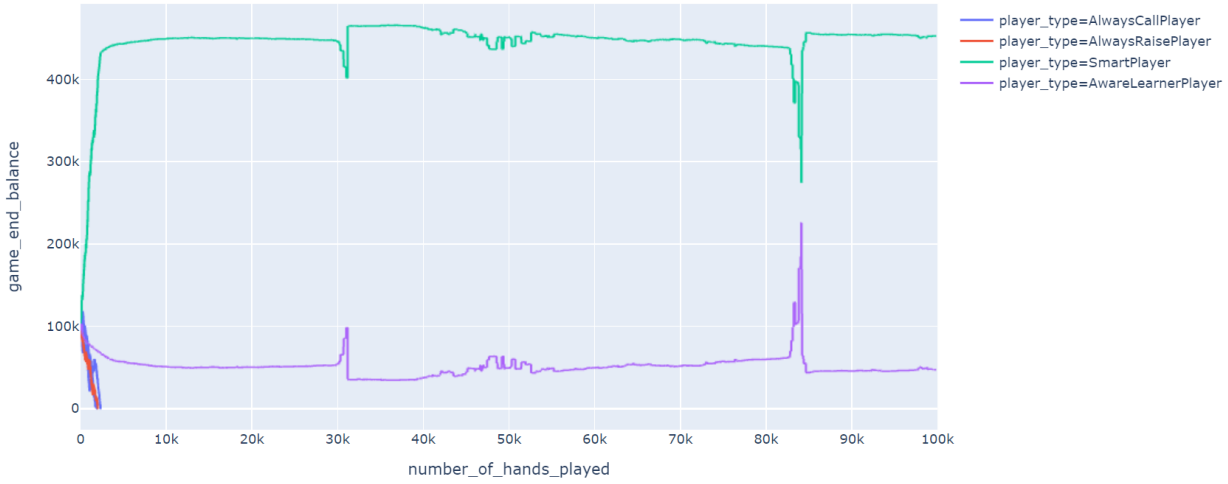


Fig.10. Aware Learner Player vs other player balance.

Monte Carlo Tree Search Python Implementation:

Our MCTS implementation is composed of 3 python objects. MCTS Set, MCTS and PlayerNodes. MCTS Sets act as a hash map that takes a player turn order configuration: ('current', 'opponent 1') maps it to a MCTS object representing an MCTS algorithm and its tree. Current in the above key means Monte Carlo Tree search player and opponent represents his opponent.

MCTS object represent the MCTS algorithm and its associated tree. The player interacts with the MCTS algorithm in two primary ways. It either: builds out a portion of the tree using the 4 MCTS steps or it queries the tree to find a specific node. Every interaction with a MCTS tree involves providing a card context. A card context represents either a player's current hand (query method's hand parameter) or the current hand and all available community cards (build method's card parameter). This is vital since the MCTS object records all information on a hand and community card level. Specifically, card_context is used to determine what cards the current player has during the Monte Carlo simulation (simulation step), it determines what keys (representing cards) are updated with wins and total games played and during the selection stage when it checks PlayerNode dictionaries for specific keys to determine if the cards have been used in a Monte Carlo simulation. This design separates information about cards from a player's moves allowing us to use a single tree to represent both types of information. We use the upper confidence bound to select nodes and it is represented by the UCB function.

PlayerNodes are nodes in the MCTS tree and represent a player's moves. The initial node is called root and is accessible by calling the MCTS objects get_root method. Children of a PlayerNode can be accessed conveniently with node.fold, node.call and node.bet properties. A parent can be retrieved using the node.parent property. It's worth noting that if a child or parent does not exist, it is represented by None. Each node keeps track of all cards played by the card_totals attribute. This is a dictionary mapping a tuple of cards to the number of games played with those cards. card_wins

attribute represents the analogous mapping, but for a player's wins. Note, back-propagation step populates both card_totals and card_wins dictionaries. One thing worth noting, select_node method checks if a node is missing in two ways. It first checks if a series of moves: fold, call, bet has never been simulated before by checking if a child node exists (checks if it is None). If the node exists, it checks the card_totals dictionary to see if the current players hand and community cards have been used in a monte carlo simulation for that node (checks for missing key).

MCTS object is explained in more detail below:

MCTS – Represents a Monte Carlo Tree Search Algorithm. It stores the root of the tree in the self.get_root() method.

build(self,card,node='root',compute_time=1,max_nodes=100000) – expands the MCTS tree by starting at the node provided by the node parameter and using the cards provided by the card parameter to execute the 4 MCTS steps: select_node, simulate_node and back_propagate_node. Optional parameters compute_time determines the amount of time in seconds allowed for expanding the tree. Max nodes parameter determines the maximum number of nodes it can create during this process. Note, node defaults to root node, but can be any PlayerNode in the tree. If the node is not root, it builds only the subtree under the node.

query(self,hand,query_set) - this method is used to query the tree for a node. The query_set parameter is a list of tuples representing player moves in sequential turn order. Query iterates over query set and uses it to trace a path from the root node to the node that the player wants to retrieve. If at any time, an intermediate node in the path or the final path doesn't exist, query will create the node, simulate it and then do the back-propagation step. So long as query_set represents a legal poker move it will return a node.

select_node(self,root,node) - this method combines MCTS selection and expansion steps. It finds a node with a missing child node and adds the missing child node to the tree. It returns the missing child. select node finds missing nodes by checking if a child is None or by using the card_context to detect cards that have not been simulated yet. select_node uses the UCB function to select nodes when all children are present. select_node implements the entire Decima poker rule set as a recursive function. This makes it rather complex. select_node also checks if a node represents the end of a game and will prevent itself from recursing down branches that have all end game states. select_node has non-recursive analogs used in the query function. select_node treats the root parameter as the top of the tree and node parameter as the active node being worked on. root parameter does not have to be the root of the MCTS tree. This is why build can expand any MCTS subtree.

simulate_node(self,node) – this method takes a missing child node and runs a Monte Carlo simulation. The cards used in the simulation are determined by the card_context attribute. For post-flop simulations, simulate_node has an option to try out more than one set of community cards. This is represented by the MCTS attribute: card_branching. The number of Conte Carlo simulations to run is represented by MCTS attribute: monte_carlo_sims. Both of these default to 5 if not specified in the MCTS constructor (__init__ method). Simulation totals and wins are recorded in the MCTS attribute: back_propagation_list, which is used during the back-

propagation step. simulate_node skips PlayerNodes that fold and the PlayerNodes representing monte carlo tree search player's opponents.

Back_propagate_node(self,node) - this method takes a missing child, retrieves the updates from MCTS attribute: back_propagation_list and uses it to update all ancestors with relevant win and game total information. It's worth noting that during the back-propagation process, it updates the ancestor's based on what card phase they are in: pre-flop, 3-community, 4-community or 5-community phases. So, if the child has the following cards: [A,A,Q,K,J] and the direct ancestor occurred during the pre-flop phase, the ancestor would only update its [A,A] key since it is not aware of the 3-community cards at that point in time.

Monte Carlo Tree Extensions:

These are extensions we made to the traditional MCTS model:

MCTS per initial player turn order - each turn order has it's own MCTS and it's stored in MCTS_Set. We discovered that turn order could influence win probabilities.

Model multiple hands in a single MCTS – each MCTS has a card_context attribute representing the current cards being processed. Each node keeps separate information on a per card basis. This prevents us from having a MCTS for each hand or community card phase. MTCS trees requires a lot of memory and so memory conservation is important.

Opponent Modeling – we didn't simulate opponent nodes or model their behavior. Instead, we skip these nodes and simulate their behavior in the Monte Carlo tree search players nodes. Opponent nodes are only used for navigational purposes by the MCTS query method.

Folding – folds are not simulated since folding always leads to a loss for the current player. We wanted to expand our tree to include information about the pot and current bids, which makes folds more relevant. That ended up being out of scope for this project.

Decima poker rules – Decima only allows 3 rounds of bidding. This is implemented in the MCTS object.

Post-flop hand – MCTS default implementation runs a single Monte Carlo Tree search simulation per node. Our implementation runs 5 as a default. For post-flop hands, we take the current players hand and community cards, for the remaining missing community cards we simulate 5 different scenarios. We found that post-flop Monte Carlo simulation was sampling from a large event space. We wanted to provide more information about win probability to the player for those post-flop nodes.

Below is a visualization of some of the changes we made compared to traditional MCTS:

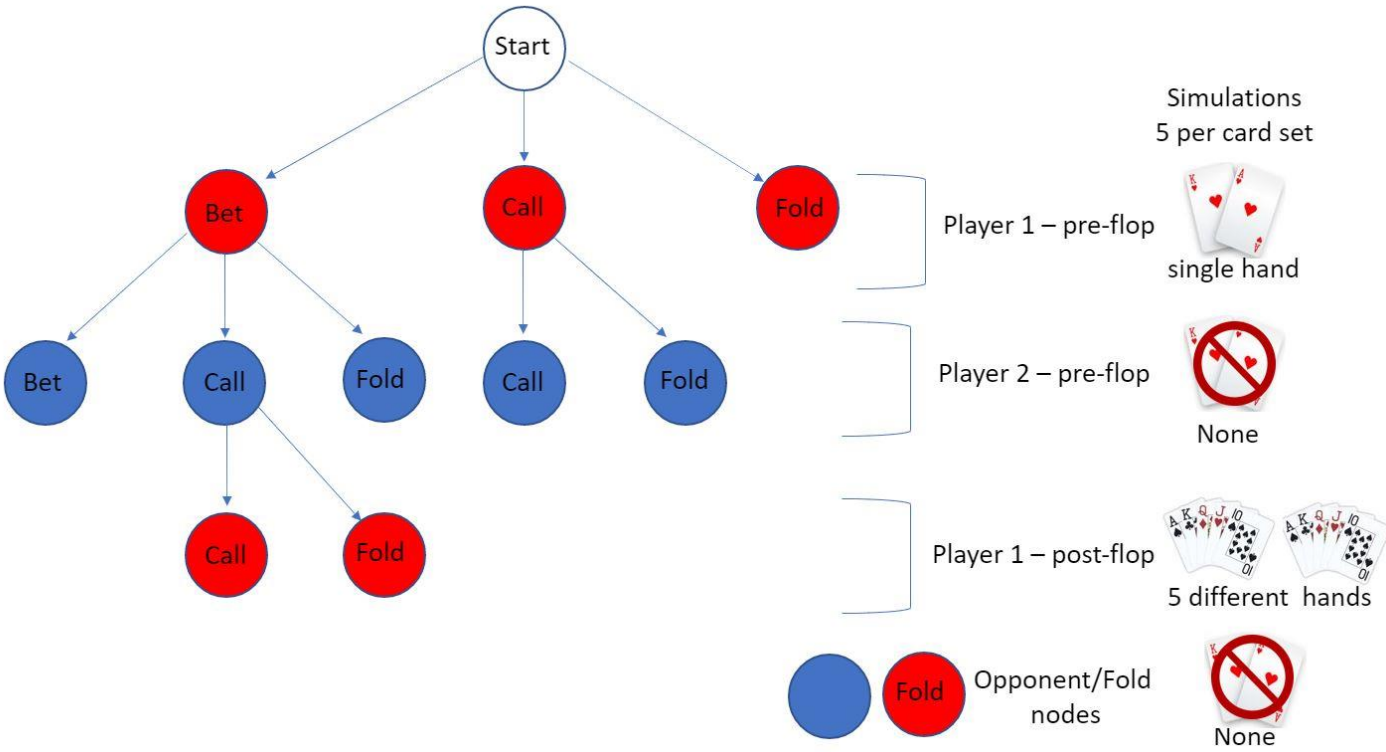


Fig.11. Modification to traditional MCTS

Monte Carlo Tree Search Betting Strategy:

When Monte Carlo Tree search player is created, he creates a MCTS set object. When he starts making betting decisions, he treats the MCTS set as a hash map and checks if the player turn order is present. If it is not, he adds a key for the turn order and associates it with a new MCTS object. Whenever he encounters this specific player turn order he will retrieve and only use that MCTS object.

Monte Carlo Tree search player then takes this same MCTS object and checks if his current hand has been simulated within the MCTS tree. If it has not been simulated, he expands (build method) the MCTS tree using his current hand as the card context. He builds for .5 seconds. This is the initial set-up of the MCTS tree.

Once his hand has been simulated at least once, he retrieves a list of all player's past moves. This is represented as a list of tuples. He then uses the query method with his hand as card_context to retrieve the node he will use for decision making. During this process any missing intermediate or final nodes are simulated and go through the back-propagation step (see query method). As a last check If the decision node has less than 100 children, the Monte Carlo tree search player will run a build process on the node for .1 seconds to supplement the decision node with more children.

Monte Carlo tree search player will then look at the children of the decision node and pick the child with the highest probability of winning. This is the MCTS best move for that specific decision.

During pre-flop, MCTS best move is used only if it is better than the average of the last 20 pre-flop win probabilities. We use this as a crude estimate of the average pre-flop win probability. For post-flop,

Monte Carlo tree search player uses the MCTS best move if his win probability has been improving over different card phases. Card phases would be pre-flop, 3-community card and 4-community card phases. If during post-flop phase, he has a better chance than random of winning (equal chance probability) he will use the MCTS best move automatically. For 5-community card phase, he only uses the equal chance probability for decision making and discontinues the increasing winning probability rule. If Monte Carlo Tree search player does not use the MCTS best move, he folds.

Monte Carlo Tree search player defaults to $100.00 when making a raise. If his win probability is greater then 90%, he bets $1,000. He will do this even if his MCTS best move is to call. This was inspired by Conservative Player from the previous paper.

Monte Carlo Tree Search Poker D3.js Visualization:

Below is a D3.js visualization of the MCTS tree (we used https://bl.ocks.org/ as a template and just added our own data) [7]. The MCTS used in this visualization is much simpler than the one that Monte Carlo Tree search player uses. Only 1 Monte Carlo simulation is run per node. Only 1 post-flop hand is simulated per post-flop node. We don't show fold nodes since they will always be 0% and add no value to the back-propagation step. The below image shows the complexity of this tree.

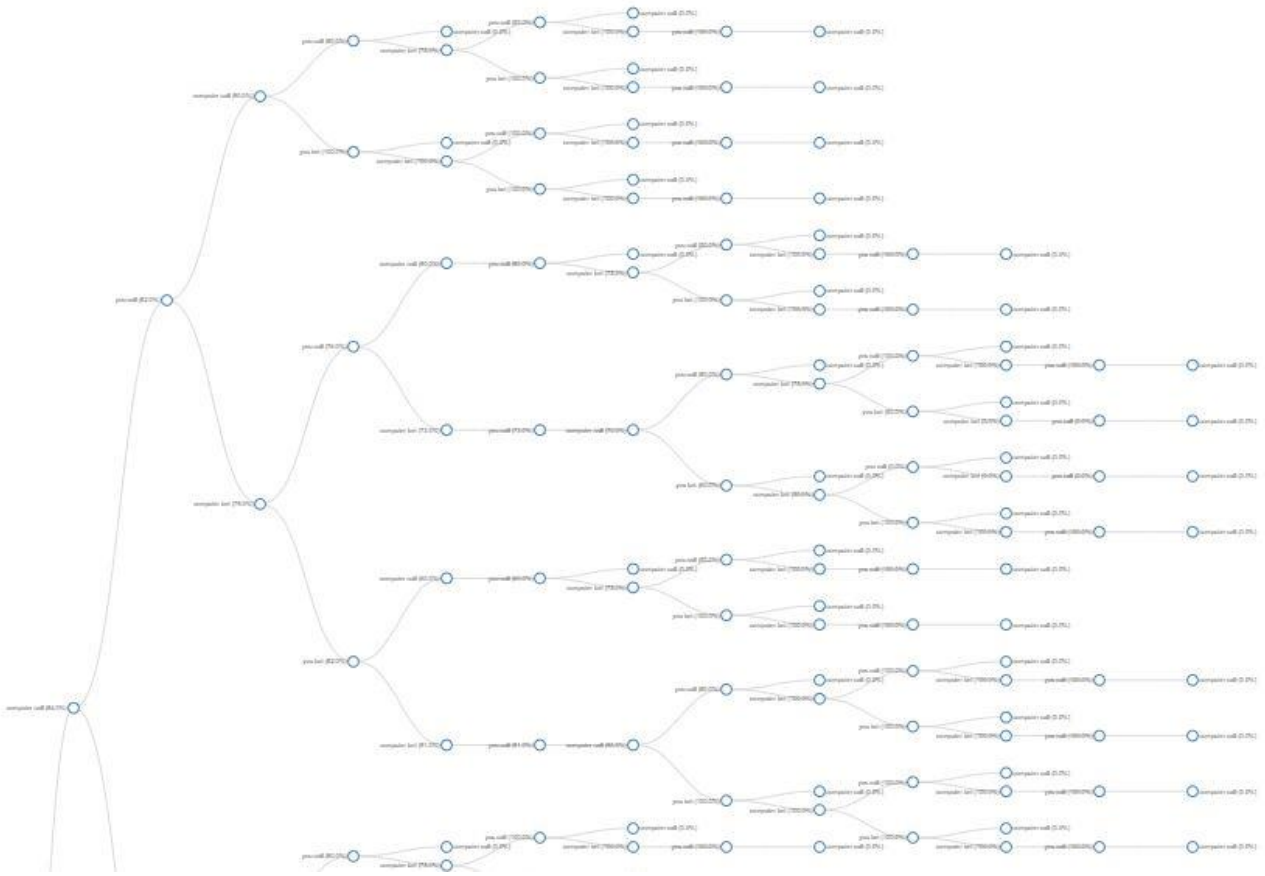


Fig.12. D3.js visualization of MCTS tree

Below a zoomed in subsection of the tree. You can see back-propagation in action. Child nodes have a high probability of winning all or none of their games. As you aggregate more nodes right to left, the odds of winning or losing all games decreases.

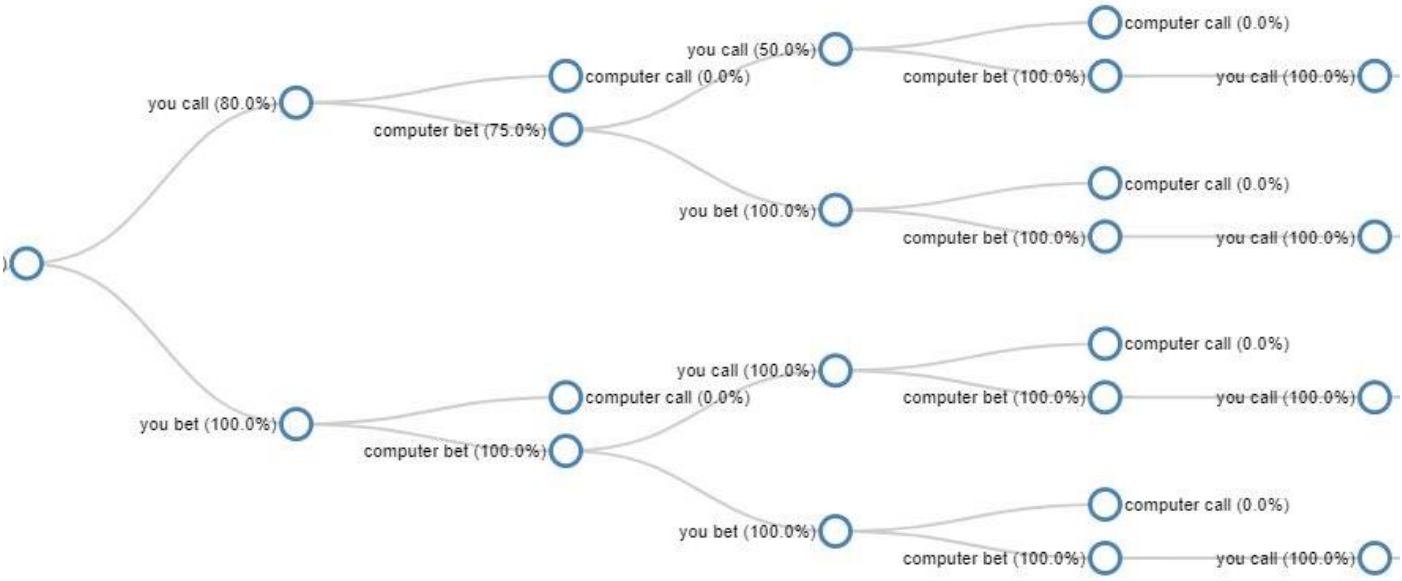


Fig.13. D3.js visualization of subsection of MCTS tree

Another image, here we see a node with a single child. Decima enforces a 3-turn maximum and prevents raises on the 3rd bid round. This forces a bidding round to end after the 3rd round of bids have been made. The below visualization shows this rule.

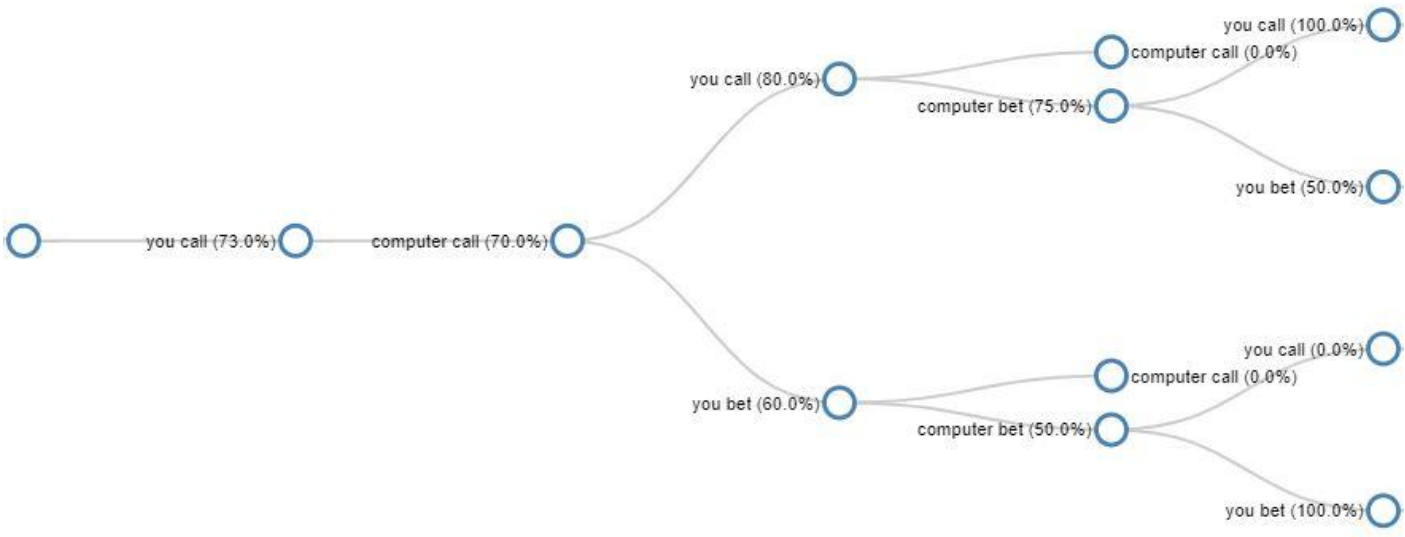


Fig.13. D3.js visualization MCTS tree a node with single child.

Monte Carlo Tree Search Analysis - Running the Analysis outputs:

Our analysis of Monte Carlo Tree search player can be found in "Poker Simulation Analysis – Monte Carlo vs Smart Player.ipynb", which can be found in source_code/analysis folder. Dependencies include: Python 3.3, Pandas, Numpy and Seaborn python libraries. We reference charts and data from "Poker Simulation Analysis.ipynb" in the same folder. This is analysis from our previous paper.

Scenarios:

Always Call Strategy vs Smart Player Strategy (benchmark): In this scenario we stacked Always Call player against Smart Player and ran different simulations by changing the number of Always Call Player from 1 to 5 (5 scenarios) to check if number of Always Call Player makes any difference on the outcome of Smart Player balance.

Always Call Strategy vs Conservative Player Strategy: In this scenario we stacked Always Call player against Conservative Player and ran different simulations by changing the number of Always Call Player from 1 to 5 (5 scenarios) to check if number of Always Call Player makes any difference on the outcome of Smart Player balance.

Always Call Strategy vs MonteCarloTreeSearchPlayer Strategy: In this scenario we stacked Always Call player against Monte Carlo Tree Search Player and ran different simulations by changing the number of Always Call Player from 1 to 5 (5 scenarios) to check if number of Always Call Player makes any difference on the outcome of Conservative Player balance.

Mean Game Balance Net Change:

To determine whether Monte Carlo Tree Search player or Smart player did better, we decided to look at the mean number of chips gained or lost per game. A high mean would indicate a better strategy. Since we ran 10 simulations for the first two scenarios, we can plot each of the 30 table means in their own histogram(Fig.14). Each row in Fig.4. represents a simulation with the Always Call players histogram being in the left column and the Smart or Monte Carlo Tree Search player being presented on the right column. The top 5 rows represent the Monte Carlo Tree Search player scenarios, while the bottom 5 represent the Smart player scenarios. Within each scenario, the histograms are ordered in ascending number of players with total number of players indicated after type of player (i.e: SmartPlayer-6 shows the balance for Smart Player when he played against 5 Always Call player). This allows us to see how players did in their specific simulation, spot trends associated with increased number of players and compare the two scenarios with each other. Fig. 15 is provided as a reference for the conservative player scenario and is based on our previous paper.

First we quote some basic statistics from our analysis. The game mean for smart player is:

2-player smart game mean: $61.00/game

3-player smart game mean: $123.00/game

4-player smart game mean: $153.00/game

5-player smart game mean: $169.00/game

6-player smart game mean: $186.00/game

Compared to Monte Carlo Tree Search game mean of:

2-player monte carlo game mean: $50.00/game

3-player monte carlo game mean: $118.00/game

4-player monte carlo game mean: $159.00/game

5-player monte carlo game mean: $243.00/game

6-player monte carlo game mean: $329.00/game

Conservative player mentioned in a previous paper had game means of:

2-player conservative game mean: $104.00/game

3-player conservative game mean: $241.00/game

4-player conservative game mean: $330.00/game

5-player conservative game mean: $369.00/game

6-player conservative game mean: $246.00/game

The variance of smart player ranged from around $43,000 to $803,000 for 2-player and 6-player games respectively. Monte Carlo Tree Search player's variance was between $309,000 and $5,797,000 respectively. Conservative player varied from: $1,916,000 to $21,609,159. Monte Carlo Tree Search player exhibits some properties of both smart and conservative players. Bet gains increase with number of players like smart player. The variance of returns for Monte Carlo Tree Search player seems to be intermediate between Smart and Conservative players. Monte Carlo Tree Search player does worse than smart player in 2, 3 and 4 player games. He approaches the same performance as smart player around 4-player games and then surpasses him for 5 and 6-player games. Conservative player beats Monte Carlo Tree Search player in everything except 6-player games.

Below we compare Monte Carlo Tree Search vs Smart scenario with that of the Conservative player vs Smart player. We notice that both Conservative and Monte Carlo Tree Search player exhibit large variance in mean gains. Conservative player has a significantly longer right-tail than Monte Carlo Tree Search player. Monte Carlo Tree Search player has more tables with greater than $400.00 gains. Smart player has exceptionally low variance compared to both players. Both Monte Carlo Tree Search and Smart player's means shift right as more players are added. Both Monte Carlo Tree Search player and Conservative player occasionally lose money independent of the number of players involved. Smart player never loses money and the overall distribution shifts outwards as more players are added. It seems that Monte Carlo Tree Search has similarities to both Smart and Conservative player, but probably leans closer to Conservative player in behavior.

Always call players in the Smart and Monte Carlo Tree Search scenarios exhibit what looks like a normal distribution centered slightly left of 0. Meaning a small loss. This contrasts with Conservative player, whose Aalways Call player opponents exhibit wild fluctuations with a distinct left tail indicating large losses.

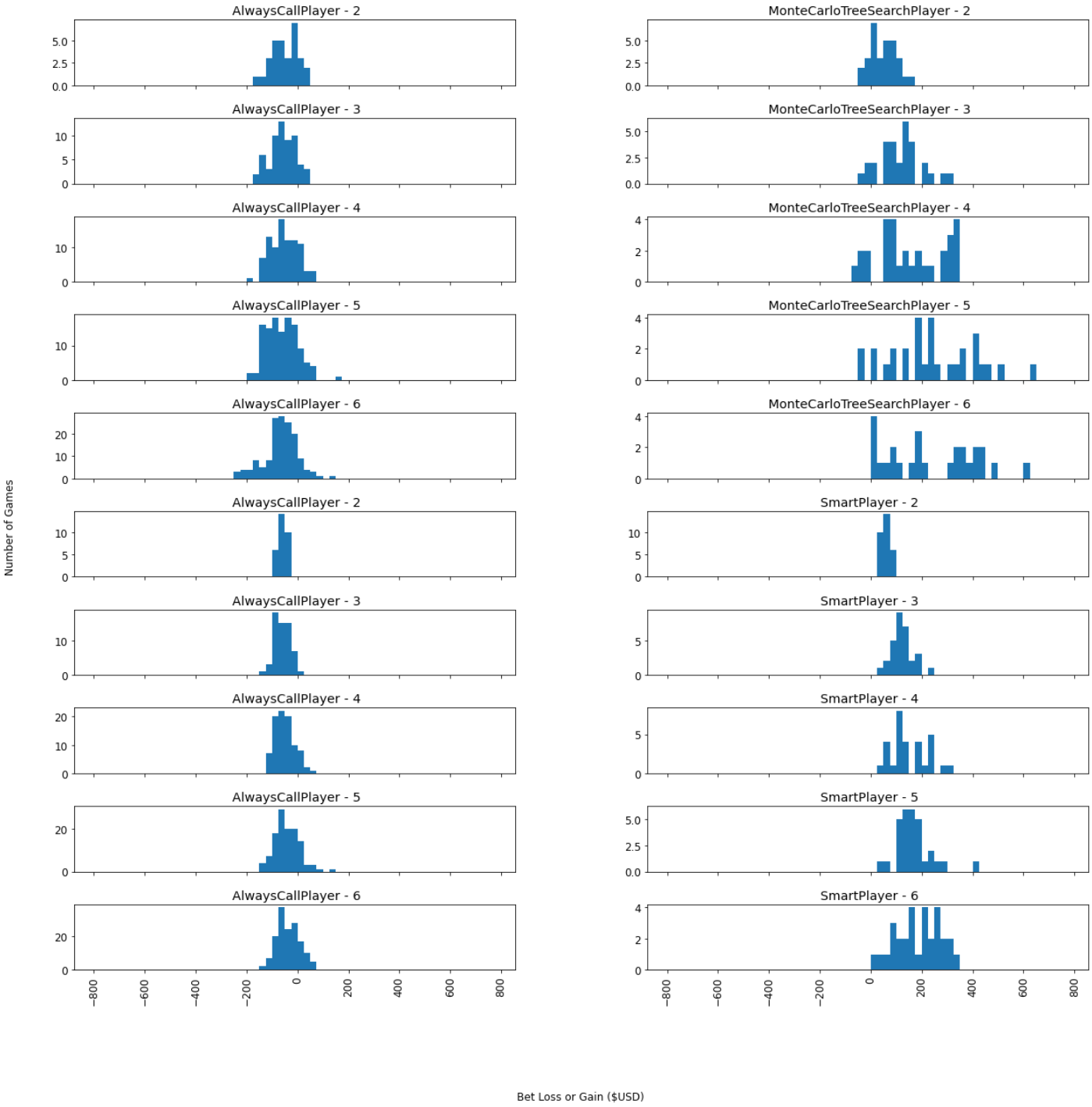


Fig.14. Bet Loss or Gain(USD) Histogram – Smart vs Monte Carlo Tree Search player

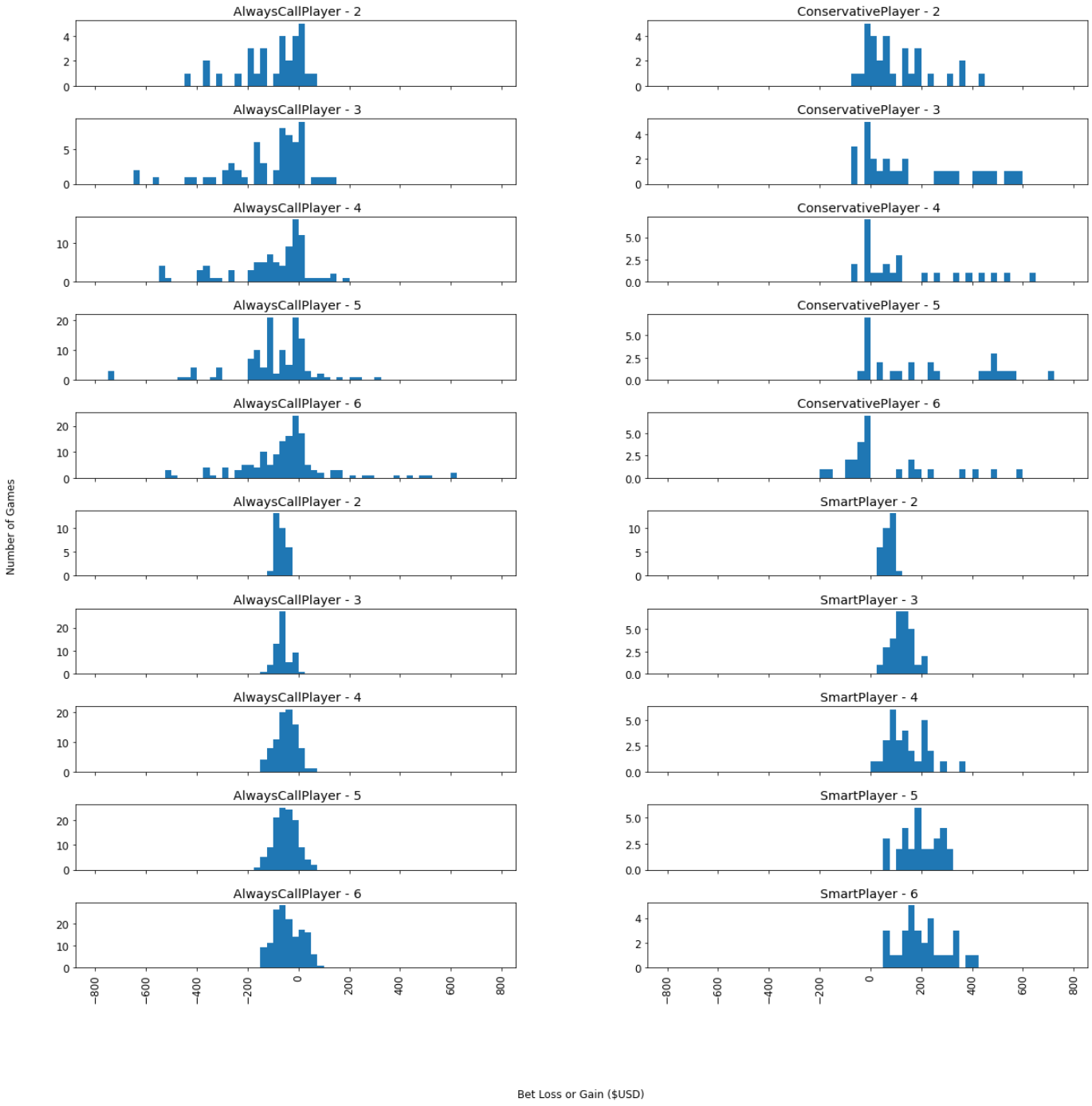


Fig.15: Bet Loss or Gain(USD) Histogram – Smart vs Conservative player

Poker end game balance over many games:

After each game completes, we record the player's balance. We call this the end game balance. Studying the end game balance shows how player's gains or losses accumulate as he plays more poker hands. A good strategy will generally lead to players with an upward trending balance, while bad

strategies will have a downward trending balance. Before we plot the end game balance as a time series, it's worth looking at the overall distribution of end game balances for each of the four player strategies used in the first 2 scenarios. It's worth noting $100,000 is a player's beginning balance. Fig 16 and 17 are provided for this analysis representing the Monte Carlo vs Smart and Conservative vs Smart player scenarios respectively.

Looking at Fig 16 and 17, we can see that both Monte Carlo Tree Search player and Conservative player show extreme gains in their end balances with an occasional loss. Monte Carlo Tree Search player's balance ranges from around $87,000 to $200,800 dollars compared to $81,400.00 to $334,500.00 for Conservative player. Smart players balance rarely reaches $141,000. Always call players facing Conservative player range from $25,600 to $193,000, while Monte Carlo Tree Search opponents have balances between $75,100 and $114,950. It seems that Conservative opponents can occasionally increase their balance by 90% or lose up to 75% of their money within 100 games, while the losses against Monte Carlo tree search player is limited to 25% of their opponent's balance and gains are capped at 15%. The results mirror the variances in the previous chart.

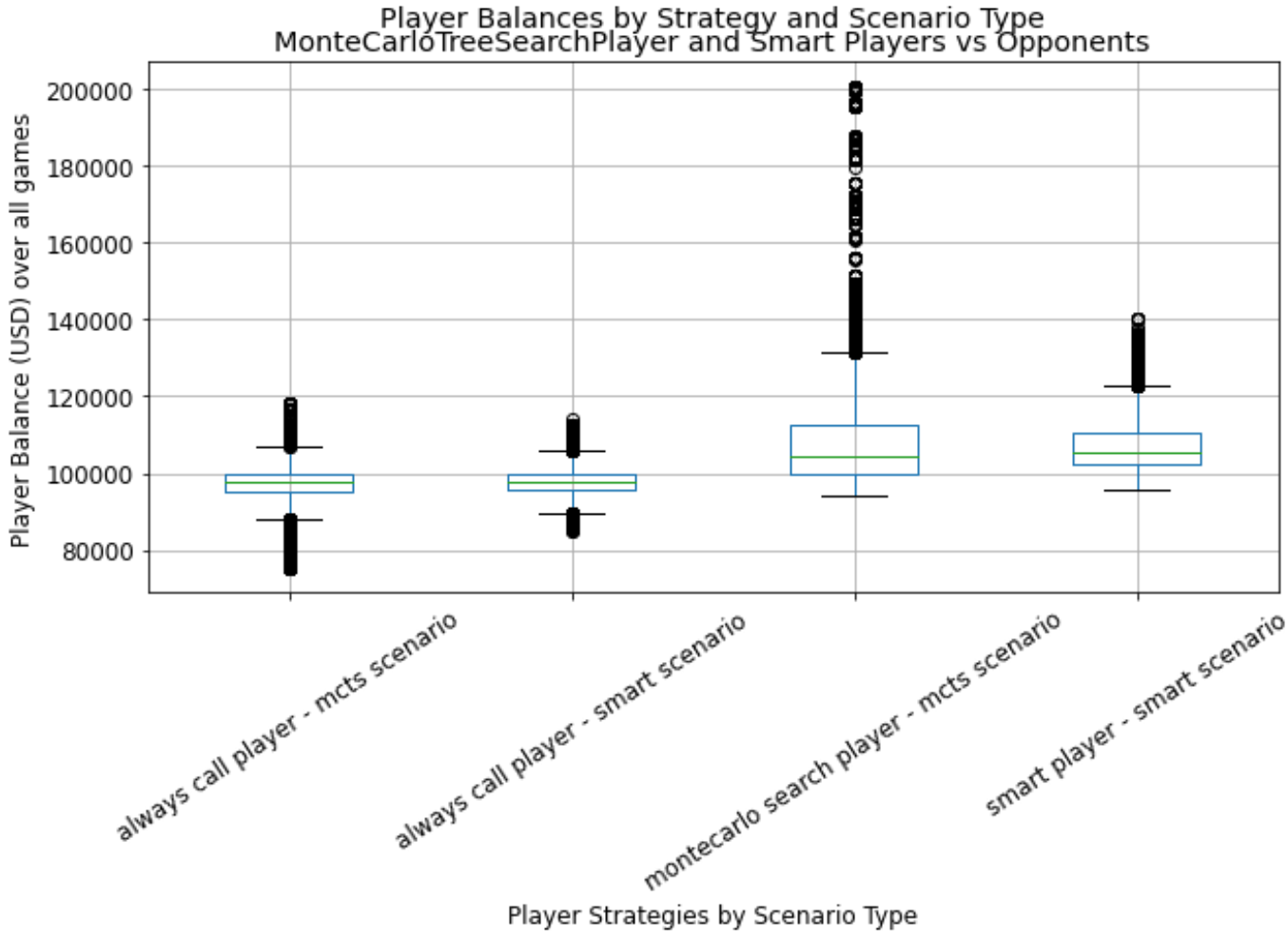


Fig.16. End game balance for each player by scenario type. Monte carlo tree search vs smart player.

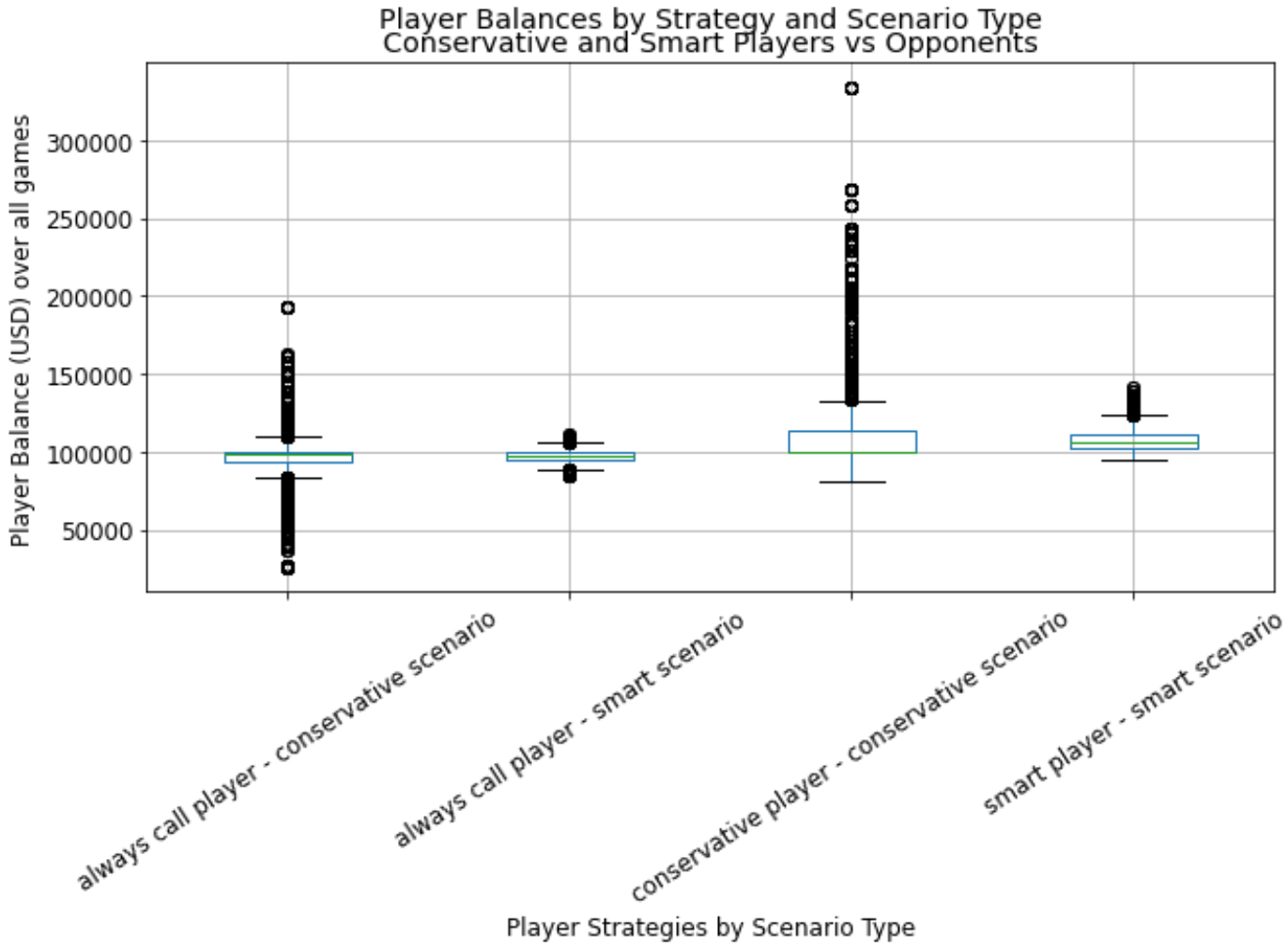


Fig.17. End game balance for each player by scenario type. Conservative vs smart player.

Adding a time-based component provides even more insight. In the following two graphs (Fig. 18 and 19), the top row represents the Smart Player scenario, while the bottom represents the Monte Carlo Tree Search or Conservative Player Scenario. The graphs depict end game balance over a sequential series of poker games and the 95th confidence interval for that mean.

Below, we can see that Monte Carlo tree search player has a wider 95th confidence interval then Smart player. Its boundary is also less erratic than Conservative player. We note in the below two graphs that Conservative player has the same balance around the 50th game as Monte Carlo Tree Search has at its 100th game. Smart player is overtaken at game 60 by Monte Carlo Tree Search player and game 40 by the Conservative player. It is significantly slower at earning chips. smart player's only redeeming quality is that its chip gains are predictable since it has a narrow 95th confidence interval.

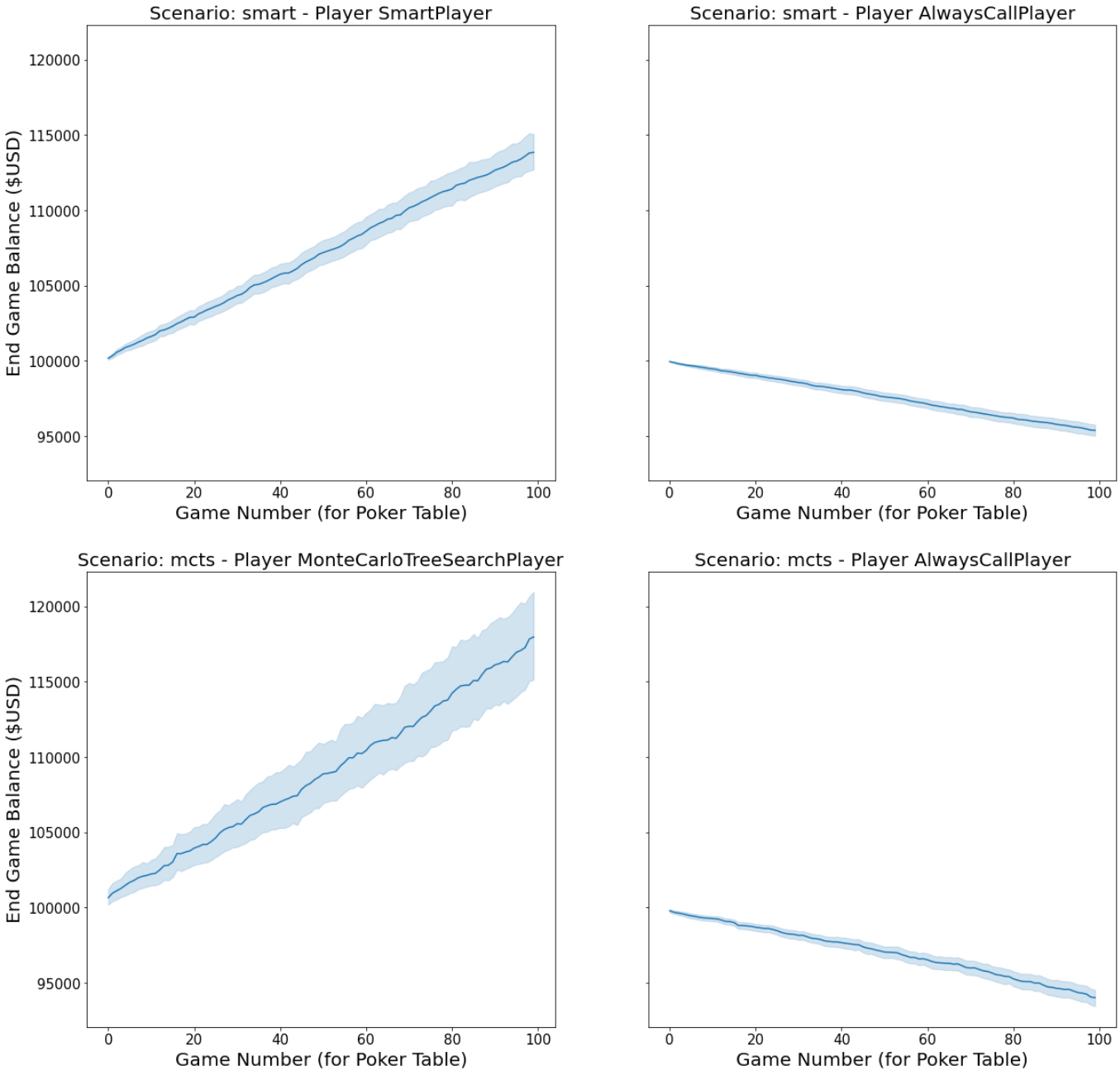


Fig.18. Strategy balance over number of games – monte carlo tree search vs smart player

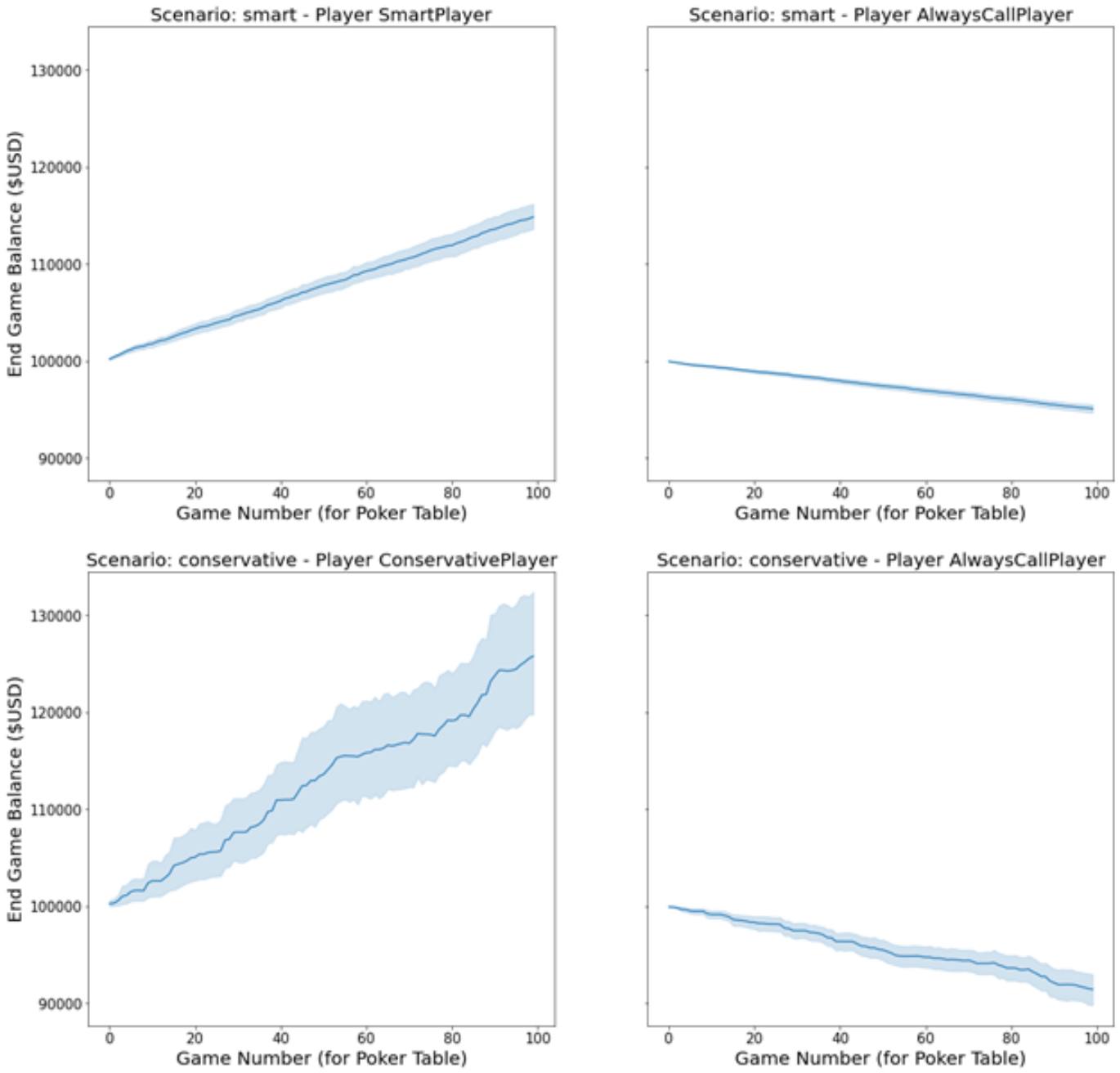


Fig.19. Strategy balance over number of games – conservative vs smart player

To explore the impact of number of players on the end game balance over time for the two scenarios, we tracked each player's balance over the number of games(Fig. 20 and 21). In the above graph, the first two rows correspond to the Monte Carlo Tree Search Player scenarios, while the last two rows correspond to the Smart Player Scenarios. The Always Call Player graphs are shown on odd rows, while the Monte Carlo Tree Search and Smart players are shown on even rows. Number of players increases in left-to-right order from 2 players to 6 players. We provide the corresponding conservative player scenario graph below it.

Below we notice that Conservative player's 95th confidence interval is erratic and less smooth than the Monte Carlo Tree Search player. The Monte Carlo Tree Search Player's scope increases at an increasing rate as more players are added. This suggests that it benefits a lot from the number of opponents it is facing. Conservative player has a lot of uncertainty involved but tends to perform better overall than Monte Carlo Tree Search player. The benefit of adding more players seems to help Monte Carlo Tree Search player the most in 6-player games where it begins to gain parity against the conservative player. Smart player makes consistent small gains that increase with more players. It's trend upwards is too slow to catch either the Conservative or Monte Carlo Tree Search players.

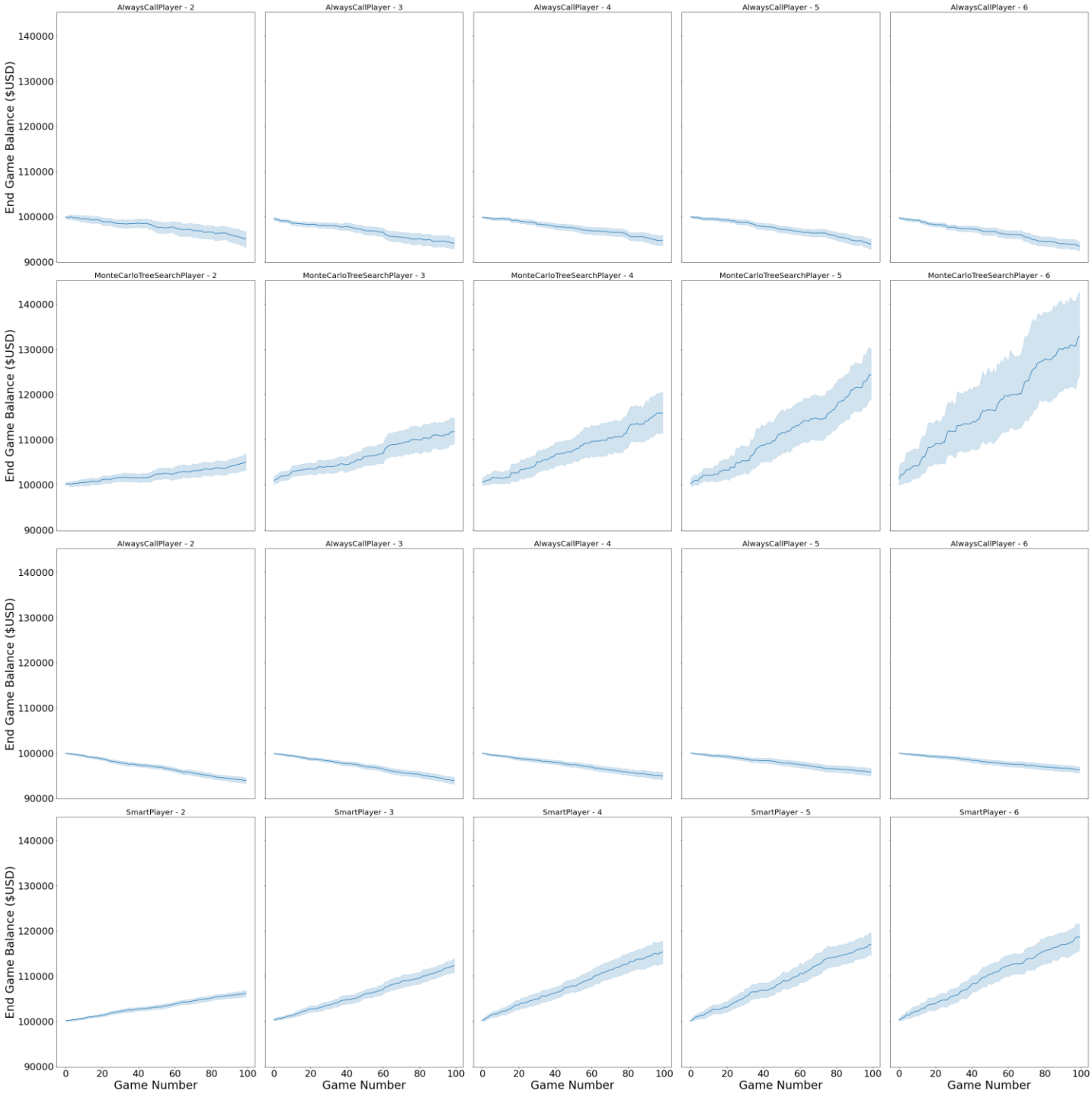


Fig.20. Player balance - Monte Carlo vs Smart player

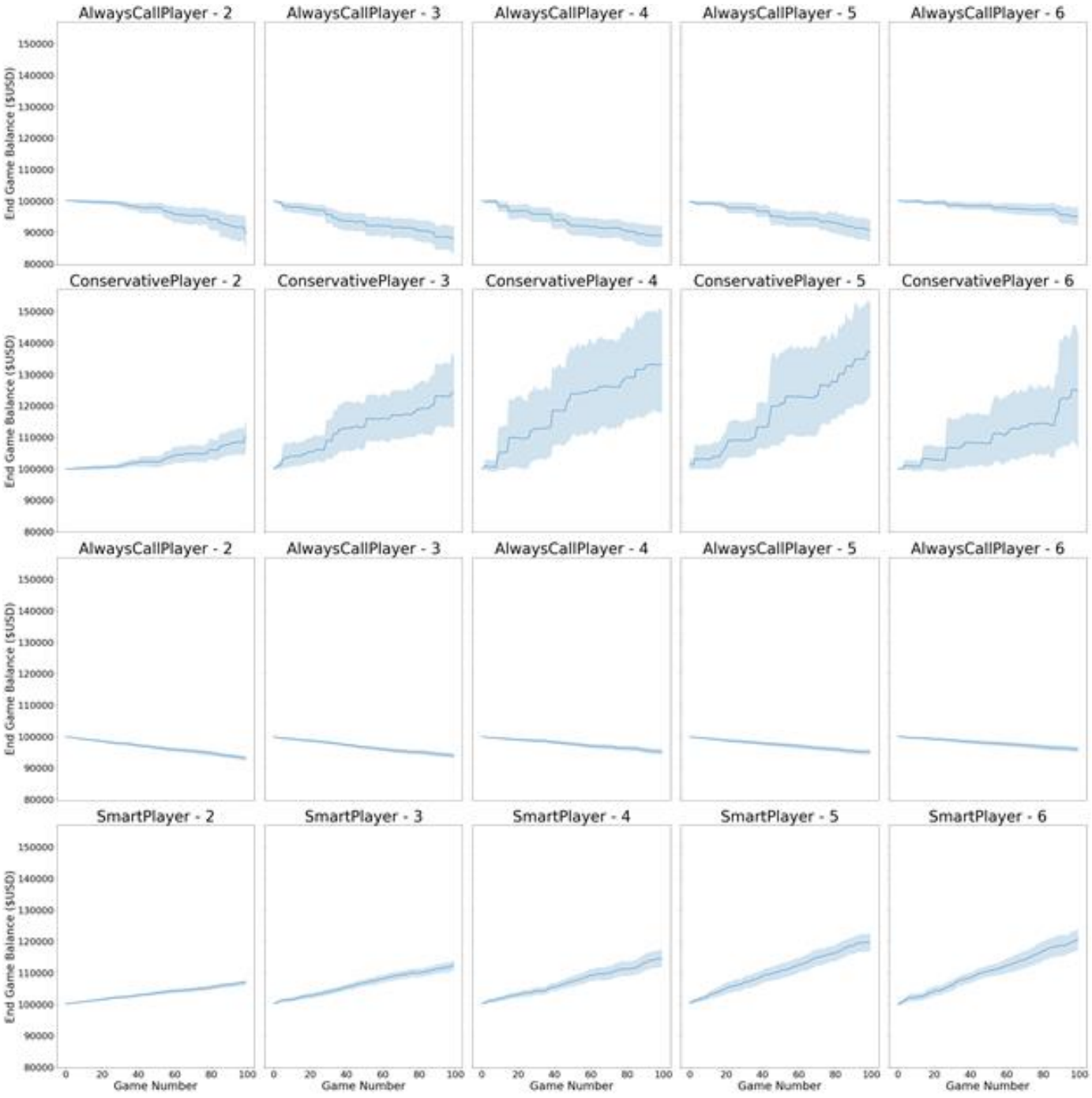


Fig.21 Player balance - Conservative player vs Smart Player

Pre-flop odds compared to end game win and fold rates:

Looking at end game balance and per game gain/loss means is a good way of determining which player strategy is better. However, It does not tell us why those player strategies are superior and what one can learn from these strategies to improve ones own playing style. This section delves less into "what" happened and more into "how" the players came to their specific conclusions.

In this section, we will look at the 2-cards dealt to each player and then look at what percent of those hands went into one of four end game states: fold, last man standing, lost game and won game states. The end game states are as follows:

- **Fold:** the player folded his hand and loses the game.
- **Last man standing:** the player was the only individual that did not fold. He automatically won the round and didn't have to reveal his hand.
- **Lost Game:** the player did not fold during the game, had his hand scored and ended up losing. Hand scoring follows typical poker ranks: high card, pair, two pair etc.
- **Won Game:** the player did not fold during the game, had his hand scored and ended up winning/tying the round.

In the following series of heatmaps (fig. 22 & 23), high card means the higher ranked card of the two-cards a player was dealt. Low card represents the lower of two cards dealt. We also use the term "same suite" meaning that both high and low card have the same suite: hearts, spades, clubs and diamonds as well as the "opposing suite" meaning the suites did not match. A pair in this analysis means both high and low card have the same rank an example being King-King, while a non-pair indicates both ranks were different. An example is King-Queen.

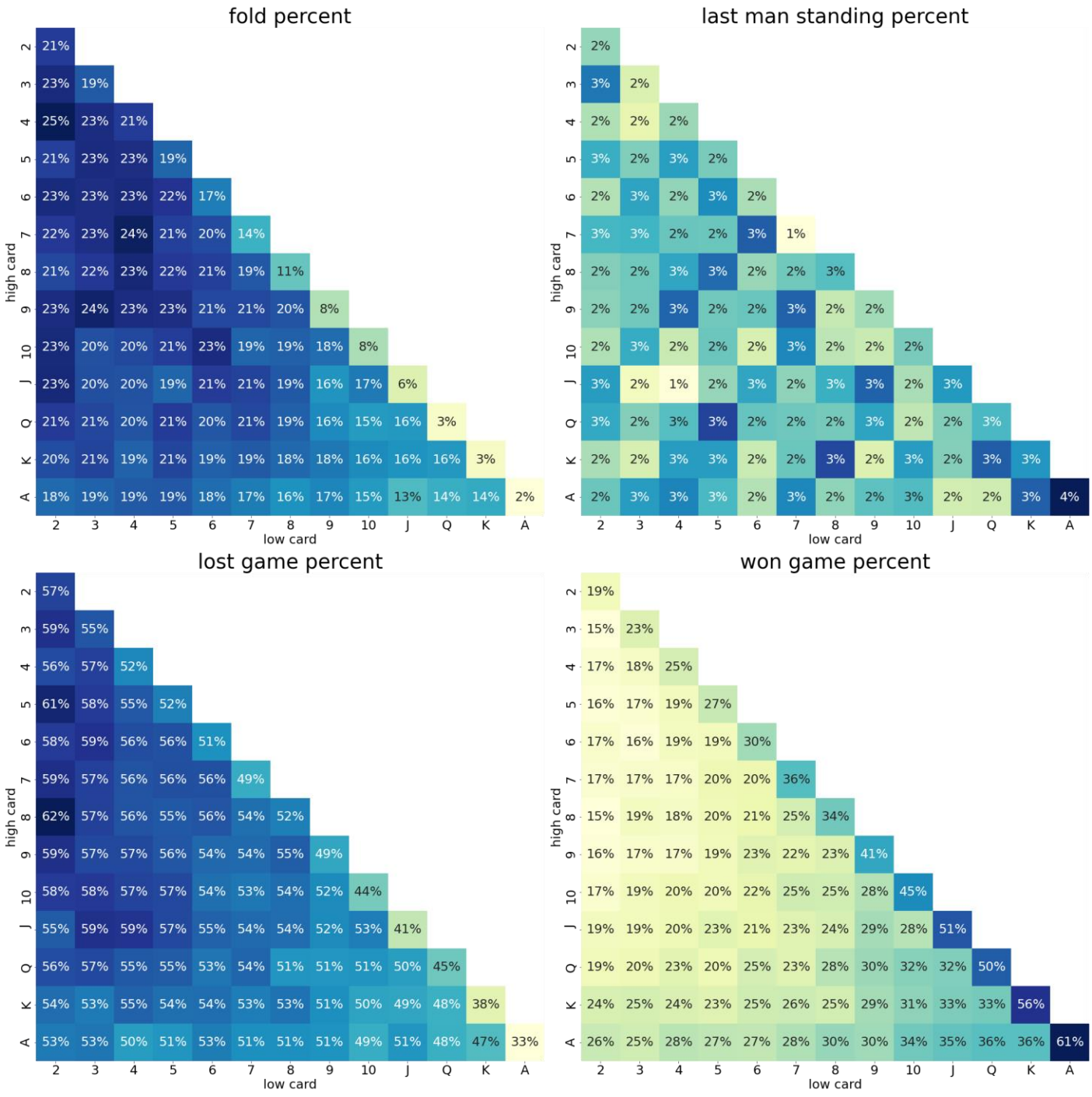


Fig.22. Percent game won based on first two cards.

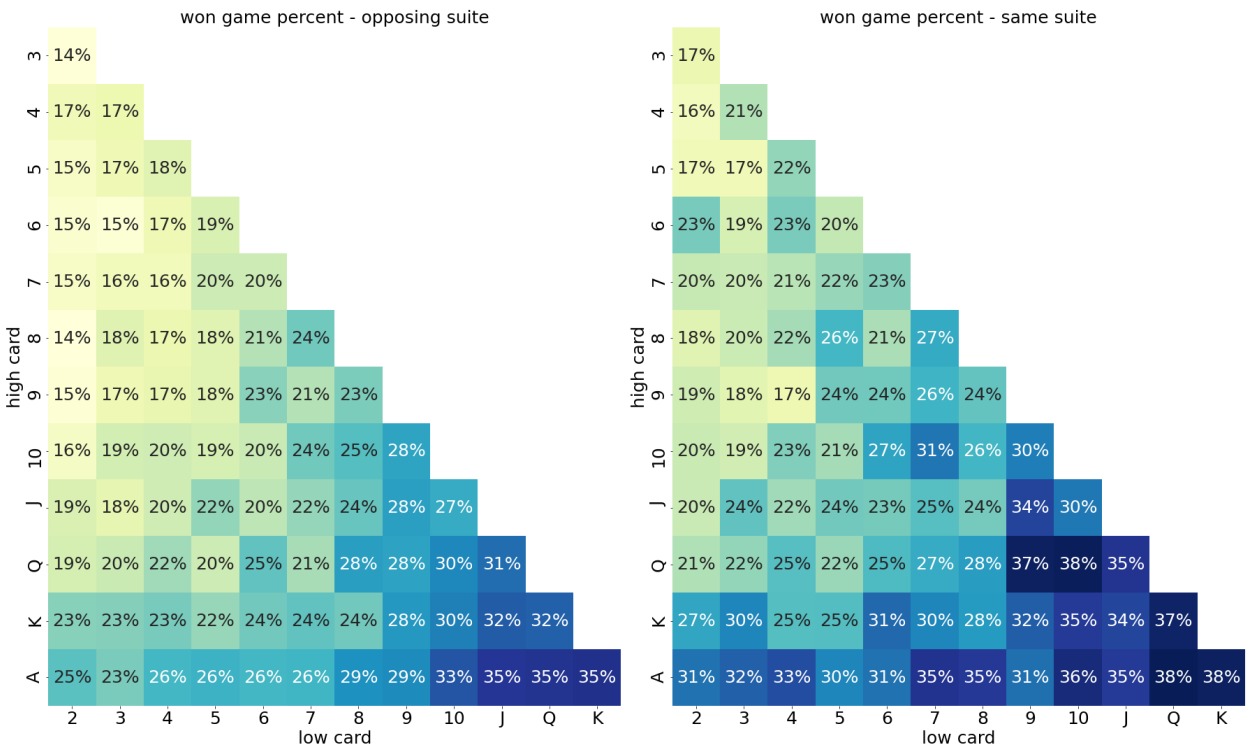
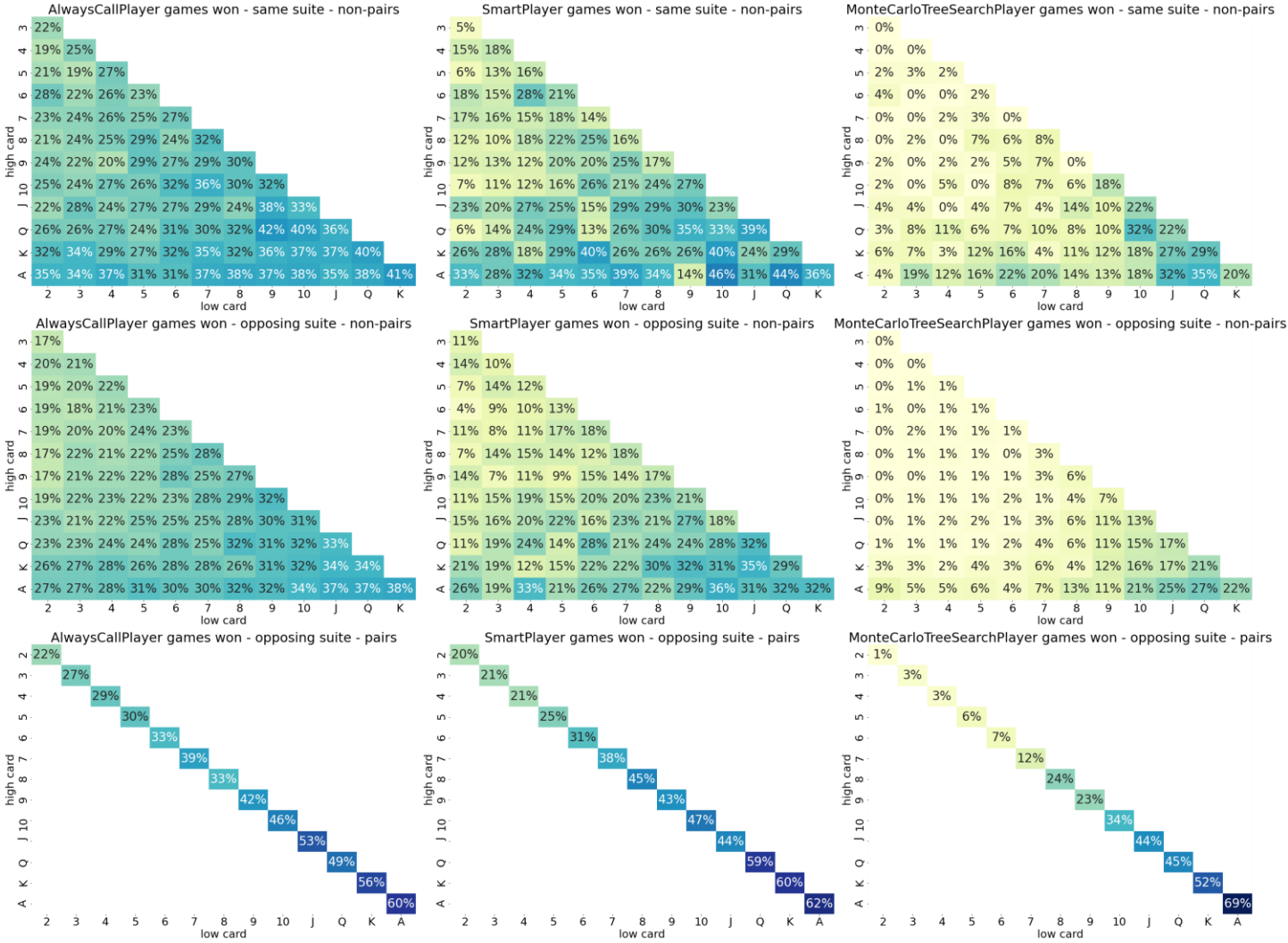


Fig.23. Percent game won based on first two cards.

The above set of 4 heat maps (Fig.22 &23) shows the end game states across all player strategies for both the Monte Carlo Tree Search and Smart player scenarios. This gives us a sense of the "typical" result for any given 2-card hand. The results of the above graphs are similar to the Conservative vs Smart player scenario and so it has been omitted.

For the next three pair of charts (Fig.24, 25 & 26), we broke down the 2-card heatmaps by player strategy with always call player being in the first column, Smart player 2nd column and Monte Carlo Tree Search player 3rd column. Rows represent three types of 2-card hands: same suite non-pairs 1st row, opposing suite non-pairs 2nd row and pairs 3rd row. We provide three series of 9 charts, the first for the game won state followed by the lost game state and the final set of 9 heatmaps for folded game state. We provide the corresponding conservative player equivalents below the Monte Carlo vs Smart player heat maps.


AlwaysCallPlayer games won - same suite - non-pairs
high card
3 22%
4 19% 25%
5 21% 19% 27%
6 28% 22% 26% 23%
7 23% 24% 26% 25% 27%
8 21% 24% 25% 29% 24% 32%
9 24% 22% 20% 29% 27% 29% 30%
10 25% 24% 27% 26% 32% 36% 30% 32%
J 22% 28% 24% 27% 27% 29% 24% 38% 33%
Q 26% 26% 27% 24% 31% 30% 32% 42% 40% 36%
K 32% 34% 29% 27% 32% 35% 32% 36% 37% 37% 40%
A 35% 34% 37% 31% 31% 37% 38% 37% 38% 35% 38% 41%
2 3 4 5 6 7 8 9 10 J Q K
low card
SmartPlayer games won - same suite - non-pairs
high card
3 5%
4 15% 18%
5 6% 13% 16%
6 18% 15% 28% 21%
7 17% 16% 15% 18% 14%
8 12% 10% 18% 22% 25% 16%
9 12% 13% 12% 20% 20% 25% 17%
10 7% 11% 12% 16% 26% 21% 24% 27%
J 23% 20% 27% 25% 15% 29% 29% 30% 23%
Q 6% 14% 24% 29% 13% 26% 30% 35% 33% 39%
K 26% 28% 18% 29% 40% 26% 26% 26% 40% 24% 29%
A 33% 28% 32% 34% 35% 39% 34% 14% 46% 31% 44% 36%
2 3 4 5 6 7 8 9 10 J Q K
low card
MonteCarloTreeSearchPlayer games won - same suite - non-pairs
high card
3 0%
4 0% 0%
5 2% 3% 2%
6 4% 0% 0% 2%
7 0% 0% 2% 3% 0%
8 0% 2% 0% 7% 6% 8%
9 2% 0% 2% 2% 5% 7% 0%
10 2% 0% 5% 0% 8% 7% 6% 18%
J 4% 4% 0% 4% 7% 4% 14% 10% 22%
Q 3% 8% 11% 6% 7% 10% 8% 10% 32% 22%
K 6% 7% 3% 12% 16% 4% 11% 12% 18% 27% 29%
A 4% 19% 12% 16% 22% 20% 14% 13% 18% 32% 35% 20%
2 3 4 5 6 7 8 9 10 J Q K
low card
AlwaysCallPlayer games won - opposing suite - non-pairs
high card
3 17%
4 20% 21%
5 19% 20% 22%
6 19% 18% 21% 23%
7 19% 20% 20% 24% 23%
8 17% 22% 21% 22% 25% 28%
9 17% 21% 22% 22% 28% 25% 27%
10 19% 22% 23% 22% 23% 28% 29% 32%
J 23% 21% 22% 25% 25% 25% 28% 30% 31%
Q 23% 23% 24% 24% 28% 25% 32% 31% 32% 33%
K 26% 27% 28% 26% 28% 28% 26% 31% 32% 34% 34%
A 27% 27% 28% 31% 30% 30% 32% 32% 34% 37% 37% 38%
2 3 4 5 6 7 8 9 10 J Q K
low card
SmartPlayer games won - opposing suite - non-pairs
high card
3 11%
4 14% 10%
5 7% 14% 12%
6 4% 9% 10% 13%
7 11% 8% 11% 17% 18%
8 7% 14% 15% 14% 12% 18%
9 14% 7% 11% 9% 15% 14% 17%
10 11% 15% 19% 15% 20% 20% 23% 21%
J 15% 16% 20% 22% 16% 23% 21% 27% 18%
Q 11% 19% 24% 14% 28% 21% 24% 24% 28% 32%
K 21% 19% 12% 15% 22% 22% 30% 32% 31% 35% 29%
A 26% 19% 33% 21% 26% 27% 22% 29% 36% 31% 32% 32%
2 3 4 5 6 7 8 9 10 J Q K
low card
MonteCarloTreeSearchPlayer games won - opposing suite - non-pairs
high card
3 0%
4 0% 0%
5 0% 1% 1%
6 1% 0% 1% 1%
7 0% 2% 1% 1% 1%
8 0% 1% 1% 1% 0% 3%
9 0% 0% 1% 1% 1% 3% 6%
10 0% 1% 1% 1% 2% 1% 4% 7%
J 0% 1% 2% 2% 1% 3% 6% 11% 13%
Q 1% 1% 1% 1% 2% 4% 6% 11% 15% 17%
K 3% 3% 2% 4% 3% 6% 4% 12% 16% 17% 21%
A 9% 5% 5% 6% 4% 7% 13% 11% 21% 25% 27% 22%
2 3 4 5 6 7 8 9 10 J Q K
low card
AlwaysCallPlayer games won - opposing suite - pairs
high card
2 22%
3 27%
4 29%
5 30%
6 33%
7 39%
8 33%
9 42%
10 46%
J 53%
Q 49%
K 56%
A 60%
2 3 4 5 6 7 8 9 10 J Q K A
low card
SmartPlayer games won - opposing suite - pairs
high card
2 20%
3 21%
4 21%
5 25%
6 31%
7 38%
8 45%
9 43%
10 47%
J 44%
Q 59%
K 60%
A 62%
2 3 4 5 6 7 8 9 10 J Q K A
low card
MonteCarloTreeSearchPlayer games won - opposing suite - pairs
high card
2 1%
3 3%
4 3%
5 6%
6 7%
7 12%
8 24%
9 23%
10 34%
J 44%
Q 45%
K 52%
A 69%
2 3 4 5 6 7 8 9 10 J Q K A
low card

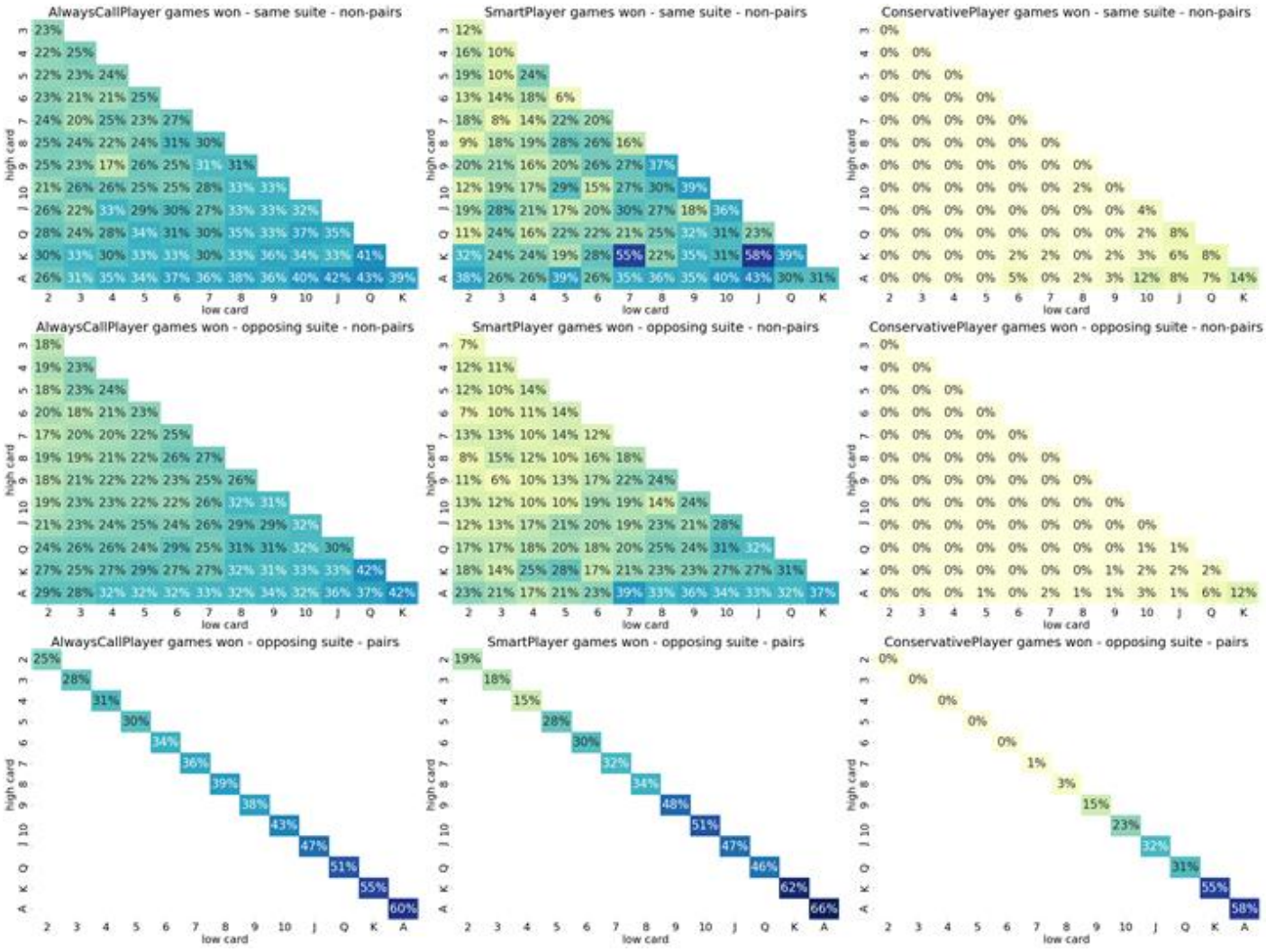
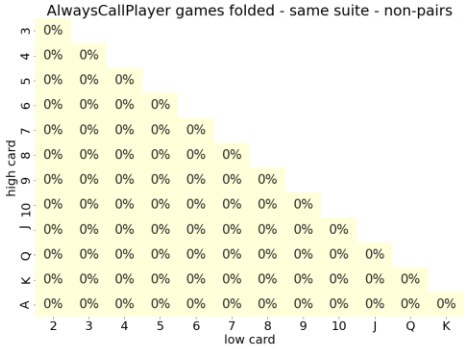
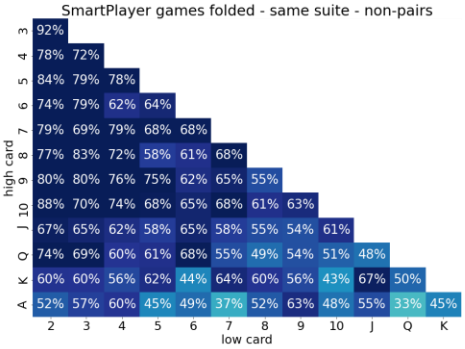
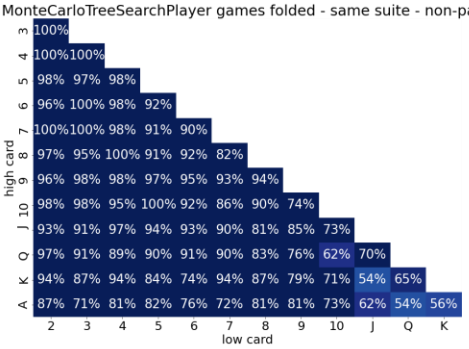
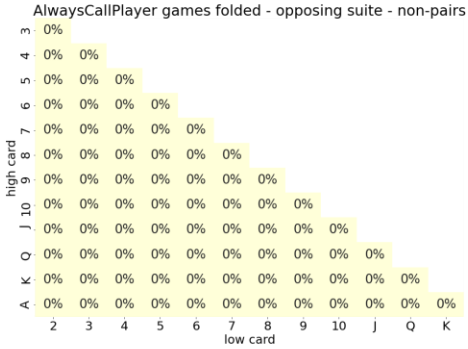


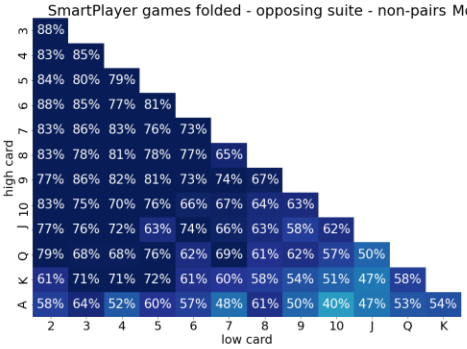
Fig.24. Probabilities of winning hands based on first two cards.

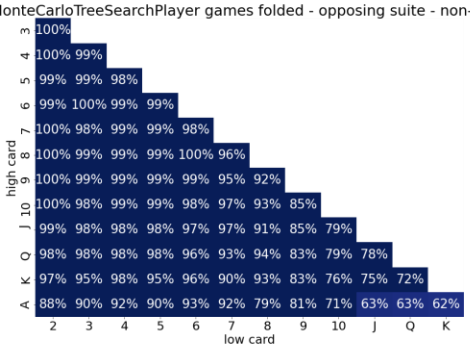

AlwaysCallPlayer games folded - same suite - non-pairs
high card
low card
3 0%
4 0% 0%
5 0% 0% 0%
6 0% 0% 0% 0%
7 0% 0% 0% 0% 0%
8 0% 0% 0% 0% 0% 0%
9 0% 0% 0% 0% 0% 0% 0%
10 0% 0% 0% 0% 0% 0% 0% 0%
J 0% 0% 0% 0% 0% 0% 0% 0% 0%
Q 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
K 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
A 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
2 3 4 5 6 7 8 9 10 J Q K

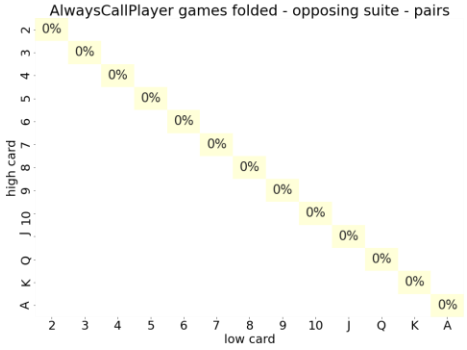

SmartPlayer games folded - same suite - non-pairs
high card
low card
3 92%
4 78% 72%
5 84% 79% 78%
6 74% 79% 62% 64%
7 79% 69% 79% 68% 68%
8 77% 83% 72% 58% 61% 68%
9 80% 80% 76% 75% 62% 65% 55%
10 88% 70% 74% 68% 65% 68% 61% 63%
J 67% 65% 62% 58% 65% 58% 55% 54% 61%
Q 74% 69% 60% 61% 68% 55% 49% 54% 51% 48%
K 60% 60% 56% 62% 44% 64% 60% 56% 43% 67% 50%
A 52% 57% 60% 45% 49% 37% 52% 63% 48% 55% 33% 45%
2 3 4 5 6 7 8 9 10 J Q K

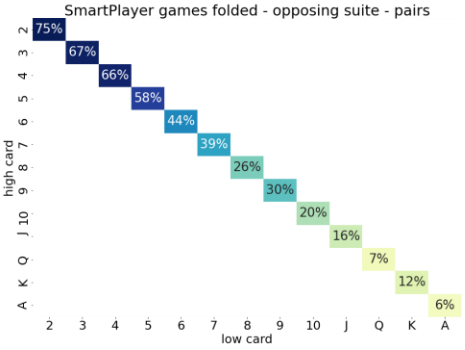

MonteCarloTreeSearchPlayer games folded - same suite - non-pairs
high card
low card
3 100%
4 100% 100%
5 98% 97% 98%
6 96% 100% 98% 92%
7 100% 100% 98% 91% 90%
8 97% 95% 100% 91% 92% 82%
9 96% 98% 98% 97% 95% 93% 94%
10 98% 98% 95% 100% 92% 86% 90% 74%
J 93% 91% 97% 94% 93% 90% 81% 85% 73%
Q 97% 91% 89% 90% 91% 90% 83% 76% 62% 70%
K 94% 87% 94% 84% 74% 94% 87% 79% 71% 54% 65%
A 87% 71% 81% 82% 76% 72% 81% 81% 73% 62% 54% 56%
2 3 4 5 6 7 8 9 10 J Q K

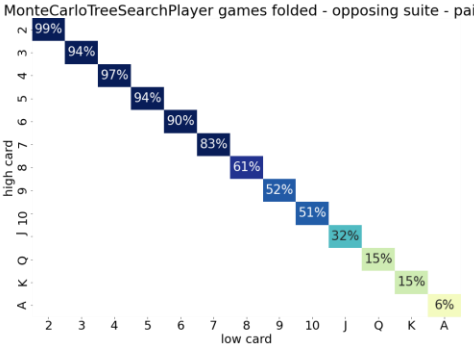

AlwaysCallPlayer games folded - opposing suite - non-pairs
high card
low card
3 0%
4 0% 0%
5 0% 0% 0%
6 0% 0% 0% 0%
7 0% 0% 0% 0% 0%
8 0% 0% 0% 0% 0% 0%
9 0% 0% 0% 0% 0% 0% 0%
10 0% 0% 0% 0% 0% 0% 0% 0%
J 0% 0% 0% 0% 0% 0% 0% 0% 0%
Q 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
K 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
A 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0% 0%
2 3 4 5 6 7 8 9 10 J Q K


SmartPlayer games folded - opposing suite - non-pairs
high card
low card
3 88%
4 83% 85%
5 84% 80% 79%
6 88% 85% 77% 81%
7 83% 86% 83% 76% 73%
8 83% 78% 81% 78% 77% 65%
9 77% 86% 82% 81% 73% 74% 67%
10 83% 75% 70% 76% 66% 67% 64% 63%
J 77% 76% 72% 63% 74% 66% 63% 58% 62%
Q 79% 68% 68% 76% 62% 69% 61% 62% 57% 50%
K 61% 71% 71% 72% 61% 60% 58% 54% 51% 47% 58%
A 58% 64% 52% 60% 57% 48% 61% 50% 40% 47% 53% 54%
2 3 4 5 6 7 8 9 10 J Q K


MonteCarloTreeSearchPlayer games folded - opposing suite - non-pairs
high card
low card
3 100%
4 100% 99%
5 99% 99% 98%
6 99% 100% 99% 99%
7 100% 98% 99% 99% 98%
8 100% 99% 99% 99% 100% 96%
9 100% 99% 99% 99% 99% 95% 92%
10 100% 98% 99% 99% 98% 97% 93% 85%
J 99% 98% 98% 98% 97% 97% 91% 85% 79%
Q 98% 98% 98% 98% 96% 93% 94% 83% 79% 78%
K 97% 95% 98% 95% 96% 90% 93% 83% 76% 75% 72%
A 88% 90% 92% 90% 93% 92% 79% 81% 71% 63% 63% 62%
2 3 4 5 6 7 8 9 10 J Q K


AlwaysCallPlayer games folded - opposing suite - pairs
high card
low card
2 0%
3 0%
4 0%
5 0%
6 0%
7 0%
8 0%
9 0%
10 0%
J 0%
Q 0%
K 0%
A 0%
2 3 4 5 6 7 8 9 10 J Q K A


SmartPlayer games folded - opposing suite - pairs
high card
low card
2 75%
3 67%
4 66%
5 58%
6 44%
7 39%
8 26%
9 30%
10 20%
J 16%
Q 7%
K 12%
A 6%
2 3 4 5 6 7 8 9 10 J Q K A


MonteCarloTreeSearchPlayer games folded - opposing suite - pairs
high card
low card
2 99%
3 94%
4 97%
5 94%
6 90%
7 83%
8 61%
9 52%
10 51%
J 32%
Q 15%
K 15%
A 6%
2 3 4 5 6 7 8 9 10 J Q K A

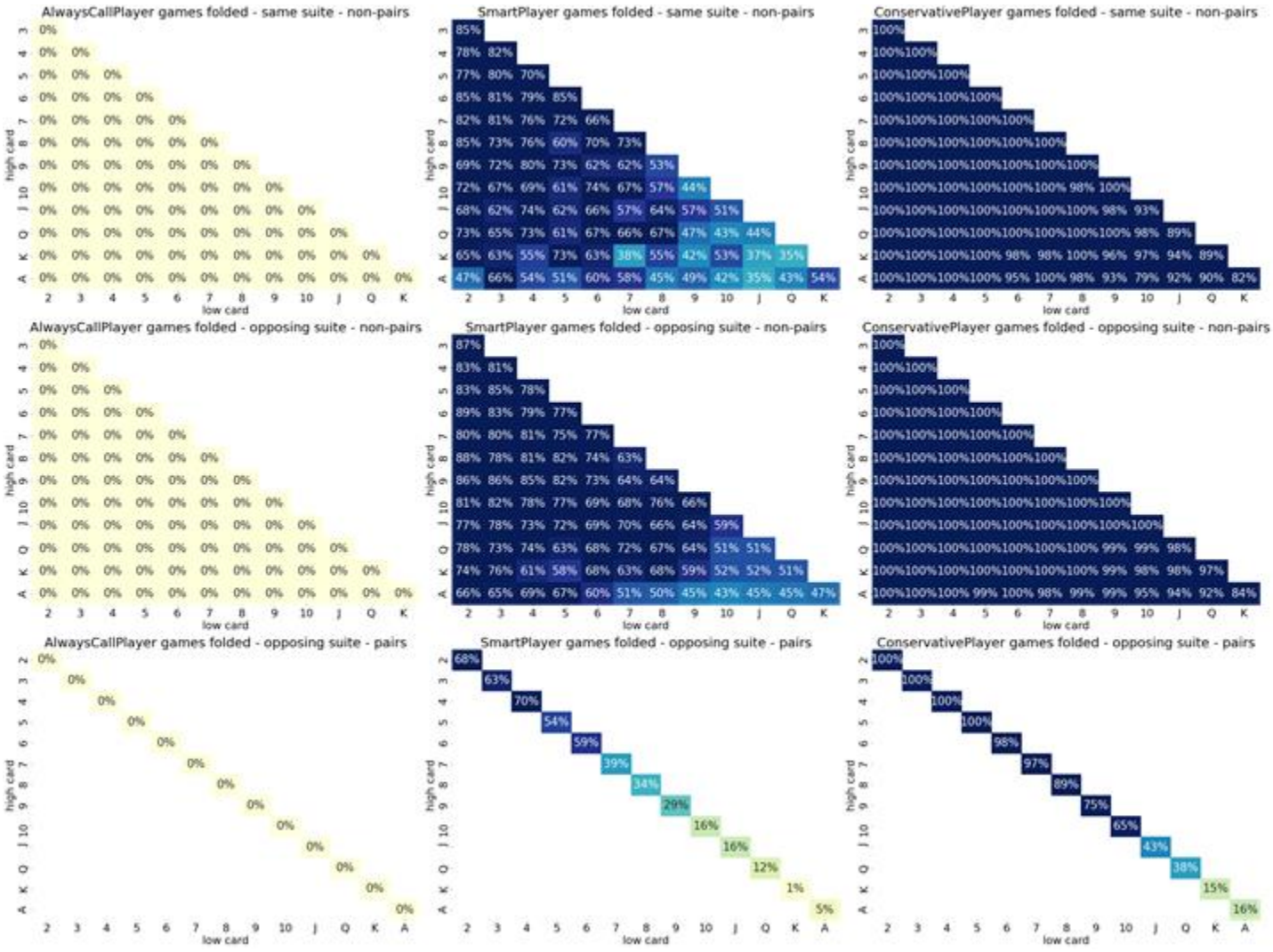
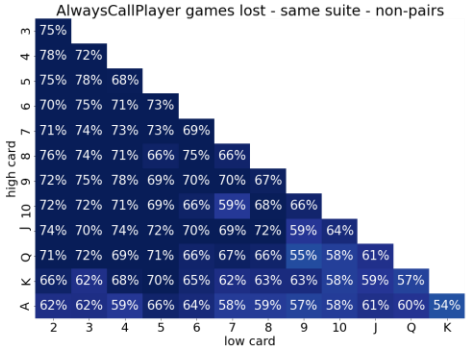
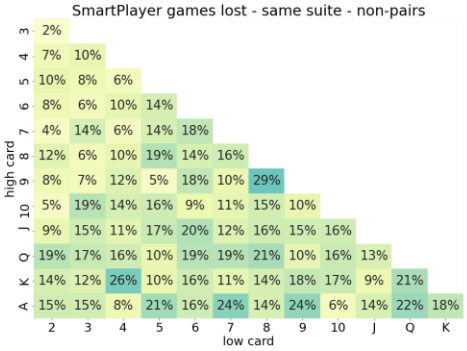
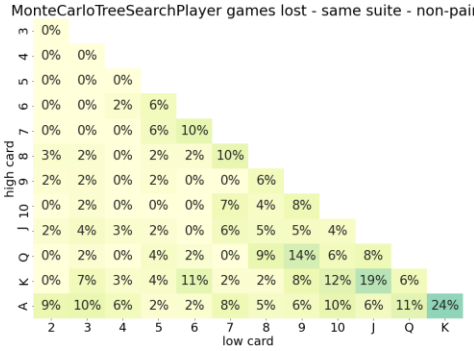
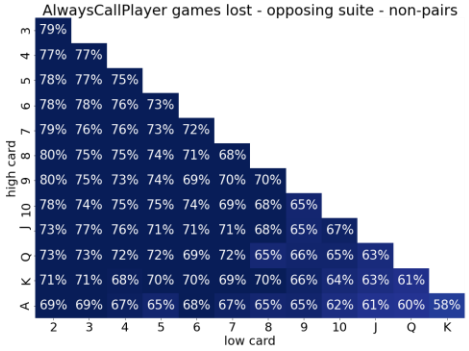


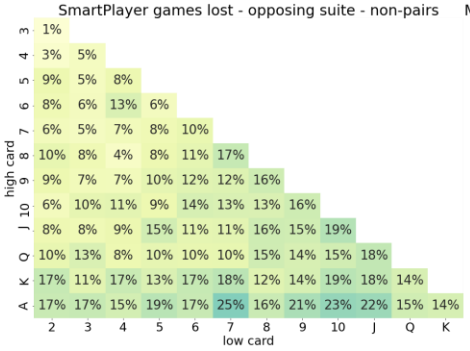
Fig.25. Probabilities of losing hand based on the first two cards.

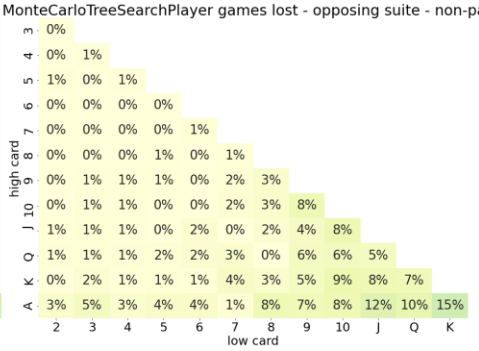

AlwaysCallPlayer games lost - same suite - non-pairs
high card
3 75%
4 78% 72%
5 75% 78% 68%
6 70% 75% 71% 73%
7 71% 74% 73% 73% 69%
8 76% 74% 71% 66% 75% 66%
9 72% 75% 78% 69% 70% 70% 67%
10 72% 72% 71% 69% 66% 59% 68% 66%
J 74% 70% 74% 72% 70% 69% 72% 59% 64%
Q 71% 72% 69% 71% 66% 67% 66% 55% 58% 61%
K 66% 62% 68% 70% 65% 62% 63% 63% 58% 59% 57%
A 62% 62% 59% 66% 64% 58% 59% 57% 58% 61% 60% 54%
2 3 4 5 6 7 8 9 10 J Q K
low card

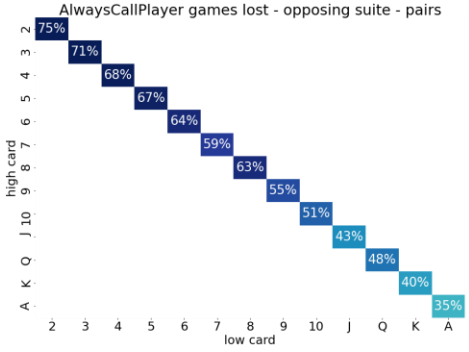

SmartPlayer games lost - same suite - non-pairs
high card
3 2%
4 7% 10%
5 10% 8% 6%
6 8% 6% 10% 14%
7 4% 14% 6% 14% 18%
8 12% 6% 10% 19% 14% 16%
9 8% 7% 12% 5% 18% 10% 29%
10 5% 19% 14% 16% 9% 11% 15% 10%
J 9% 15% 11% 17% 20% 12% 16% 15% 16%
Q 19% 17% 16% 10% 19% 19% 21% 10% 16% 13%
K 14% 12% 26% 10% 16% 11% 14% 18% 17% 9% 21%
A 15% 15% 8% 21% 16% 24% 14% 24% 6% 14% 22% 18%
2 3 4 5 6 7 8 9 10 J Q K
low card

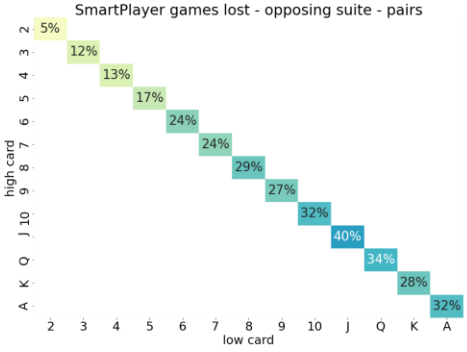

MonteCarloTreeSearchPlayer games lost - same suite - non-pairs
high card
3 0%
4 0% 0%
5 0% 0% 0%
6 0% 0% 2% 6%
7 0% 0% 0% 6% 10%
8 3% 2% 0% 2% 2% 10%
9 2% 2% 0% 2% 0% 0% 6%
10 0% 2% 0% 0% 0% 7% 4% 8%
J 2% 4% 3% 2% 0% 6% 5% 5% 4%
Q 0% 2% 0% 4% 2% 0% 9% 14% 6% 8%
K 0% 7% 3% 4% 11% 2% 2% 8% 12% 19% 6%
A 9% 10% 6% 2% 2% 8% 5% 6% 10% 6% 11% 24%
2 3 4 5 6 7 8 9 10 J Q K
low card

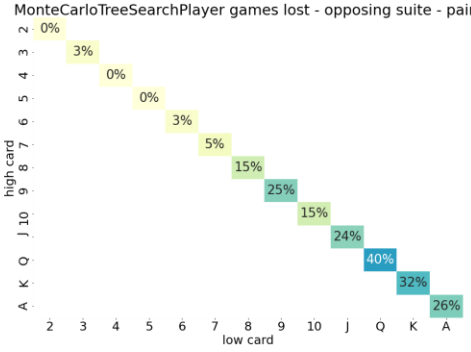

AlwaysCallPlayer games lost - opposing suite - non-pairs
high card
3 79%
4 77% 77%
5 78% 77% 75%
6 78% 78% 76% 73%
7 79% 76% 76% 73% 72%
8 80% 75% 75% 74% 71% 68%
9 80% 75% 73% 74% 69% 70% 70%
10 78% 74% 75% 75% 74% 69% 68% 65%
J 73% 77% 76% 71% 71% 71% 68% 65% 67%
Q 73% 73% 72% 72% 69% 72% 65% 66% 65% 63%
K 71% 71% 68% 70% 70% 69% 70% 66% 64% 63% 61%
A 69% 69% 67% 65% 68% 67% 65% 65% 62% 61% 60% 58%
2 3 4 5 6 7 8 9 10 J Q K
low card


SmartPlayer games lost - opposing suite - non-pairs
high card
3 1%
4 3% 5%
5 9% 5% 8%
6 8% 6% 13% 6%
7 6% 5% 7% 8% 10%
8 10% 8% 4% 8% 11% 17%
9 9% 7% 7% 10% 12% 12% 16%
10 6% 10% 11% 9% 14% 13% 13% 16%
J 8% 8% 9% 15% 11% 11% 16% 15% 19%
Q 10% 13% 8% 10% 10% 10% 15% 14% 15% 18%
K 17% 11% 17% 13% 17% 18% 12% 14% 19% 18% 14%
A 17% 17% 15% 19% 17% 25% 16% 21% 23% 22% 15% 14%
2 3 4 5 6 7 8 9 10 J Q K
low card


MonteCarloTreeSearchPlayer games lost - opposing suite - non-pairs
high card
3 0%
4 0% 1%
5 1% 0% 1%
6 0% 0% 0% 0%
7 0% 0% 0% 0% 1%
8 0% 0% 0% 1% 0% 1%
9 0% 1% 1% 1% 0% 2% 3%
10 0% 1% 1% 0% 0% 2% 3% 8%
J 1% 1% 1% 0% 2% 0% 2% 4% 8%
Q 1% 1% 1% 2% 2% 3% 0% 6% 6% 5%
K 0% 2% 1% 1% 1% 4% 3% 5% 9% 8% 7%
A 3% 5% 3% 4% 4% 1% 8% 7% 8% 12% 10% 15%
2 3 4 5 6 7 8 9 10 J Q K
low card


AlwaysCallPlayer games lost - opposing suite - pairs
high card
2 75%
3 71%
4 68%
5 67%
6 64%
7 59%
8 63%
9 55%
10 51%
J 43%
Q 48%
K 40%
A 35%
2 3 4 5 6 7 8 9 10 J Q K A
low card


SmartPlayer games lost - opposing suite - pairs
high card
2 5%
3 12%
4 13%
5 17%
6 24%
7 24%
8 29%
9 27%
10 32%
J 40%
Q 34%
K 28%
A 32%
2 3 4 5 6 7 8 9 10 J Q K A
low card


MonteCarloTreeSearchPlayer games lost - opposing suite - pairs
high card
2 0%
3 3%
4 0%
5 0%
6 3%
7 5%
8 15%
9 25%
10 15%
J 24%
Q 40%
K 32%
A 26%
2 3 4 5 6 7 8 9 10 J Q K A
low card

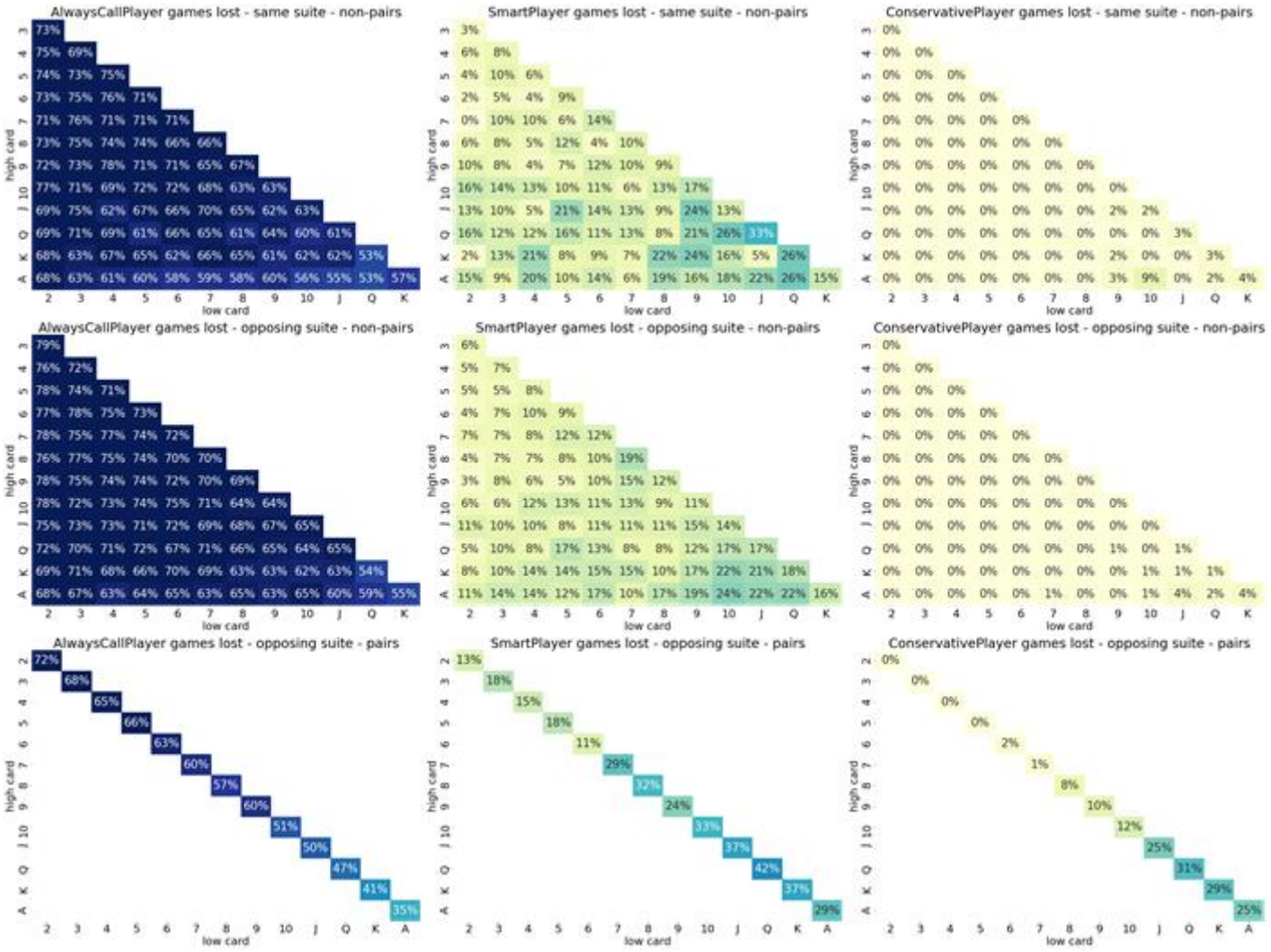


Fig.26. Probabilities of folding hand based on the first two cards

Fig 24, 25 and 26 support that Monte Carlo Tree Search player behaves in a similar fashion to the previous paper's Conservative player. The difference is that Monte Carlo Tree Search Player is less conservative than Conservative player and is willing to play significantly more hands. conservative player folds almost all non-pair hands that do not include at least a Jack. Even then, he folds more than 90% of these hands. Even with pairs conservative player folds 90% of pairs less then 9-pair. Monte Carlo Tree Search Player fold between 90-99% of all non-pair off suites that don't have a king or higher. He does in rare occasions around 1-3% of the time attempt a 2-5 off suite. So Monte Carlo Tree Search player does play a wider variety of hands than Conservative player. That said, both players are closer in folding behavior than Smart player, who often attempts the same hands 20-30% of the time. It's worth noting that for almost every card combination, Monte Carlo Tree Search is slightly more likely to play the card than conservative player. Smart player is much more likely to play a hand compared to both Monte Carlo Tree Search and Conservative player. These tendencies extend to non-pairs with the same or opposing suites and pairs. Pairs are preferred by all 3 player types compared to all other pre-flop hands.

Monte Carlo Tree Search player plays A, K more frequently than other cards when non-paired. He/she also prefers cards that are off by 1 in rank. Examples being A-K, K-Q and 9-8. Outside of the Ace row in the non-pair heat map charts, Monte Carlo Tree Search player prefers same suit non-pairs over those with opposing suites. Monte Carlo Tree Search player's preference for same suite non-pairs is stronger than Conservative players preference for them. Though it's worth noting that Conservative player has a strong dislike of non-pairs in general and almost plays pairs exclusively.

My conclusion is that monte Carlo Tree Search player behaves in a similar fashion to conservative player. The difference between these two players is that Monte Carlo Tree Search player on very rare occasions will play almost every single card combination (typically 1-3% of the time for low ranked cards). One other difference is his preference for same suit cards, off by 1 rank cards and a stronger preference for A and K cards when non-paired compared to conservative player. Compared to Smart player, Monte Carlo Tree Search player is very conservative and almost always prefers the best hands. Smart player would observe that compared to Conservative player Monte Carlo Tree Search player plays significantly more non-pair cards that contain a face card paired with a 10 or higher card. Compared to Smart player, Monte Carlo Tree search player is significantly closer in behavior to Conservative player than to Smart player.

Conclusion

Monte Carlo Tree Search player makes bets in a similar fashion to conservative player, but bets on more hands in general. He also bets more when he thinks he has a high probability of winning. Unlike Conservative player, Monte Carlo player still uses a fixed increment betting strategy compared to a percent of balance strategy. This means that Monte Carlo Tree Search player has higher variability compared to Smart player, but significantly less variability than the extremely picky Conservative player. One conjecture is that the additional rules such as the 20-pre-flop moving average and requirement to bet on only post-flop hands with increasing win probabilities acts as a significant filter on the MCTS tree and causes the Monte Carlo Tree Search player to behave in a similar manner as Conservative player. Alternatively, maybe the back-propagation algorithm combined with expanding subtrees causes the MCTS algorithm to become more conservative by oversampling pessimistic scenarios. A good follow up experiment would be to remove any decision rules used by the Monte Carlo Tree Search player and replace them with rules similar to Smart Player. This would allow us to see how conservative the MCTS algorithm is compared to a 100-card Monte Carlo Simulation.

It's worth mentioning that our MCTS implementation has many potential expansion opportunities. We could for example model the opponent nodes, tune the amount bet using maybe grid search, we could use information about betting gains and losses to modify the MCTS decision criterion replacing win rates and we could attempt a reinforcement learning hybrid where two MCTS trees are used to train each other. These extensions would take significant time but could yield interesting results. Another avenue of exploration is to modify how the Monte Carlo Tree Search player uses the MCTS object. One could fine tune: compute time, max nodes and how the win probability is used to come up with a different sets of decisions and see if that impacts his performance. We could also integrate Aware Player code into the Monte Carlo Tree search to see if that would improve its behavior. This would result in something closer to AlphaGo implementation of MCTS with policy and reward networks.

On the other hand, our Learner players was a lot simpler to implement and less computational expensive. They performed well against Always Call and Always Raise player by taking advantage of

learning the winning probability of each hand. Our Simple Learner Player didn't perform well against more sophisticated player simply because other players have an advantage of recalculating their winning probability in each run and minimize their losses. Aware Player on the other hand took advantage of learning about its opponent to compensate for its inability to calculate the probability of its hand at each round. Our Aware player was a lot simpler and took less computational power than other sophisticated players. Even with this simplicity it was able to withstand and not lose the game against smart player. The best way to tackle this type of problem with its entire complexity is to use Policy Iteration, however, to keep this simple and be able to deliver result within the timeline of this project we decided to pursue with two simple Value Iterations.

Group members: Christopher Kottmyer, Shahin Shirazi

References:

[1] https://en.wikipedia.org/wiki/Poker_probability

[2]https://en.wikipedia.org/wiki/Reinforcement_learning#:~:text=Reinforcement%20learning%20(RL)%20is%20an,supervised%20learning%20and%20unsupervised%20learning.

[3] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[4] https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238

[5] https://www.aaai.org/ocs/index.php/WS/AAAIW14/paper/download/8811/8351

[6] http://starai.cs.ucla.edu/papers/VdBACML09.pdf

[7] https://bl.ocks.org/d3noob/8375092