**An Investigation on Monte Carlo Methods for an Omaha Poker AI**

**Introduction**

Pot-limit Omaha is an variant of poker that has grown in popularity over the past few years. It is similar to the popular Texas Hold'em variant, but with a few significant differences. The game consists of four cards in the hand of each player and up to five community cards on the table. Omaha is broken up into several stages, each with its own betting phase. Prior to any community cards being shown is the pre-flop stage, where each player chooses if they wish to play their hand by calling or raising the current big blind. Betting does not stop until all players have bet equal amounts, given they have enough money. The next stage is the flop, where three community cards are shown on the table, after which another round of betting ensues. In Pot-limit Omaha, the max bet is the current amount in the pot. If all players have bet, the next two stages, the turn and the river, will continue with the same betting patterns. If, at any betting phase, a player folds (refuses to call a bet), they forfeit their hand and lose all money they bet into the pot. The term showdown is used when there is at least two players left in the game after the last betting phase. In a showdown, players show their hands and the pot is awarded to the winner. Players must use exactly two cards in their hand and 3 cards on the table to create a final hand. In the case of a tie, the pot is split.

Our project will be based on the competition hosted by The AI Games[[1]](#endnote-1) that allows users to upload bots for a heads up (1 vs. 1) Pot-Limit Omaha game. Each bot can be programmed with its own AI and will face off against other users' bots. The rankings are measured through an Elo system where each bot can play other bots within 10 ranks above or below itself. One technical rule that has significant impact on the bot AI is that there is a time bank in which a bot must make its move. For each game, each bot begins with a time bank of 10 seconds and after each successful move, 0.5 seconds are added to the time bank. Bots can be coded in any of 5 languages - C#, C++, Java, JavaScript, and Python - supported by the game's framework, and communication between the bot and the server is done through input and output channels. For our bot, we chose to use Java as our language of choice.

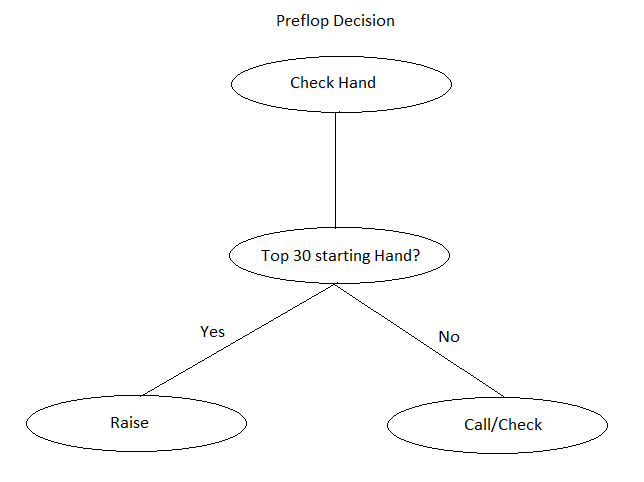
**Approach**

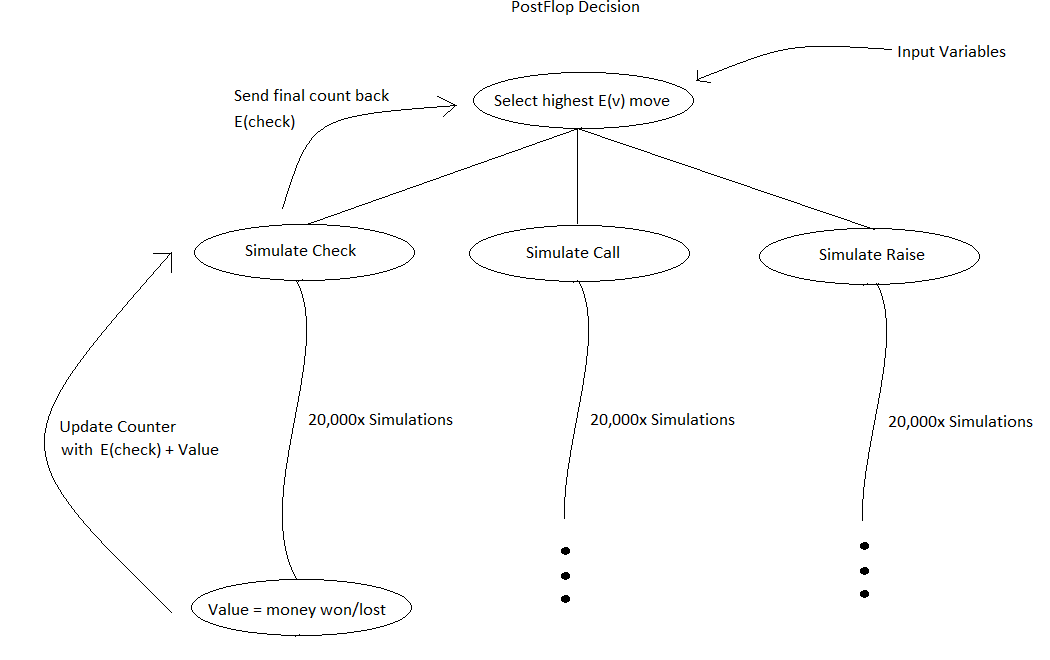
Due to the fact that poker is a stochastic game, it is not feasible to construct a standard game tree for each possible state. In addition, the sheer number of possible hands and combinations make creating a complete game tree too time consuming for this format of competition. Instead, we approach the problem using Monte Carlo simulations. Using this method, we will be able to simulate thousands of games, adjustable according to our time bank, and keep track of games won, or for a more advanced method the amount of money won depending on what move is taken. We programmed two independent algorithms based on different principles. The first method is based on the final gain of an action, assuming the opponent plays as an intelligent machine with fold, call or raise action based on their hand strength. The second method is based on the probability to win given the hand strength, and use this probability to derive an optimal strategy that maximize our gain or minimize the opponent gain. We implement both methods in this project and note the differences in effectiveness over a long period of time. Considering the fact that Omaha poker has several stages of the game, we are able to simulate each possible move allowed at that stage of the game to measure what is the most profitable move, given that the opponent is also playing optimally.

**Define Algorithms**

**Method 1: Monte Carlo method based on money won each round**

**Procedure:** We split the implementation of our poker bot into two separate algorithms based on the game state. When the game has just started and there are no community cards on the table, our bot implements a simple rule based decision making process. Once cards are dealt to the table, our strategy changes to analyze each possible move from that position and choose the best one through running multiple Monte Carlo simulations on each possible next move. A simplified model of our pre-flop and post-flop logic can be seen in the diagrams below:





Our pre-flop decision is simply based on whether or not the cards in our hand are within the top starting hands in Omaha poker.[[2]](#endnote-2) If so we will raise, otherwise we will call (or check if the opponent checked).

Our post-flop decision implements a modified Monte Carlo Tree Search. Given a number of input variables, we run simulations for each possible next move through to the end of the current round. For example, if the opponent had just raised, we have the option of calling, checking (folding), or raising. We will set our next move to one of these values, then run 20,000 simulations with the immediate next move fixed to one of the possible values. After each simulation gets to an end of a round (some player has folded, or both players compare hands), we calculate who the winner of that instance was and how much money was won (or lost). When comparing our hand to the opponent's, we randomly assign 4 cards that are available (not within our hand or on the table) for each simulation to the opponent. After each simulated round ends, we pass the expected value of the leaf back up and add it to a counter that keeps track of the sum value of taking this move. We repeat this for all remaining moves and end up with at most 3 numbers, each representing the value of choosing one of the next possible moves (E[Check], E[Call], E[Raise]). Then we take the highest of these values and select it for our next move.

A sample situation is shown below:

Initial conditions:

Cards on Table: 3

Opponent's last move: none (start of the round)

Money in pot: $100

First move we are simulating: raise

Opponent hand: randomly generated at start

Since this path is for checking the expected value of choosing raise as a first move, we simulate that we raise some amount to the opponent. Then we randomly generate a response from the opponent. They call. Since the table is not full, we flip another card to the table (randomly generated from remaining cards in the deck). Now we randomly choose a move for the opponent and ourselves. Suppose we both check. A final card is flipped on the table. This time we randomly select the opponent to raise, and we call. Since the table is full, we can now compare hands. If we won, all money in the pot is passed up and added to E[raise]. Otherwise, we subtract this number from E[raise]. We run this simulation 20,000 times with different cards being given to the opponent and flipped on the table. The reason we chose 20,000 was because we were constrained to be able to make a move within .5 seconds, and found that our chosen number lead to an acceptable runtime. Now we can run another 20,000 simulations for check (and call if applicable) and choose the next move that gives us the highest return.

**B. Define Heuristic**

The heuristic used for our Monte Carlo simulations were based on the value of many different input variables. The most important of which were the following:

Opponent's last move (Either check, call, or raise)

Cards in our hand

Cards we randomly assigned to the opponent

Depending on the variables above, we make some changes to the standard Monte Carlo simulation based on the properties of Omaha poker. Most importantly, Omaha is referred to as a game of "the nuts".[[3]](#endnote-3) In other words, the winning hand in any given round is almost always the best (or close to the best) hand possible for a given table. This fact heavily influenced many probabilities that we set for randomized nodes. In a simulation of a poker game the random events include each player's move, each card flipped onto the table, and the possible hands that an opponent can have. For a standardized Monte Carlo simulation, each of these events would typically be modeled by a uniform distribution.[[4]](#endnote-4) However, we found that this method had many flaws if naively adapted for our model. For example, if we set a uniform probability of the opponent's move (33.3% check, 33.3% call, 33.3% raise) we found that the expected values for certain game positions were wildly inaccurate. One such position may be a case where we have a very weak hand and the opponent has a strong hand. Under normal logic, we would think the best move in this position would be to check (or fold). However, a bot running a uniform distribution would find that the highest expected value in this case would be to raise. This is because every time we raise in a simulation, the opponent has a 1/3 chance to fold meaning we would win the entire pot. Because of this and related situations, we made the following modifications to our simulation:

1. If the opponent's last move was a raise, we will always simulate a hand for them that is at least as strong as a two-pair. As stated before, Omaha is a game of the nuts so bluffing is normally not an advisable move. If we apply min-max thinking to how the opponent is moving, it makes sense that they should only raise when they have at least a moderately strong hand.

2. If either player has a strong hand, the probability of raising and calling should be higher than that of checking. Again, this is an approximation on min-max since we expect that one will win more money by betting large on good hands

3. If we raise at any time during the simulation, we do not allow a fold from the opponent. The rationale behind this was due to the betting nature of Omaha poker. Unlike in no-limit Texas Hold-Em, Omaha limits the amount a player can bet at any time. Because of this, calls were much more common and folds were very rare. Thus when we simulate a raise, we want to be able to get to showdown without any premature folding to accurately gauge the true value of our hand.

**Method 2: Monte Carlo based on probability to win (Overall hand strength)**

**A. Estimate probability to win by Monte Carlo**

The second method that we tested was one where we are only concerned with how many games we will win given 4 specific cards in our hand and some number of cards on the table. The rationale behind this method was to check whether total wins or amount of money won was the more important factor in betting. To do this, we simply simulate a random set of 4 cards to the opponent (that are possible given cards already seen) and fill the table until it has exactly 5 cards. Then we check who has the better hand. If our hand was better, we increment a win counter (win\_num) by 1, if the opponent had a better hand, then we increment a loss counter (lose\_num) by 1. Both counters start at zero, and are incremented through a set of 20,000 simulations. At the end of all simulations, we calculate our win ratio as so:



We will refer to this number as our "win strength". We will utilize this ratio along with the heuristics below to make a decision on our next move.

**B. Define Heuristic**

**Strategy 1: fold when the expected value of calling is lower than that of folding**

We will define  as the amount in the pot, and  as the amount that we need to call (given that the opponent has raised). We can either call or fold in this situation. From our simulation, we have the win strength . If we call and win, we will win for a winning game and lose an extra  for a lost game. Thus the expectation of benefit is:



From which:



We fold when , and call otherwise

**Strategy 2: Raise when the expected value of raising is greater than that of calling**

In this situation, we want to see how profitable it will be to raise. We want to calculate the extra money we win for a raise compared to a call. Assuming hand strength is , raise amount is , pot is , the expectation is:



Where



**3. Min-Max algorithm to deal with raise action**

Now we will make the assumption when we fold, it is because the opponent chose to raise, and also that the opponent raises only when they have a good hand. Thus, simply modeling the opponent hand as a random distribution is not a proper assumption. To deal with this issue, we choose a min-max strategy, which will minimize the opponent benefit when assuming they will choose the optimal strategy. The evaluation function will be defined as how much benefit the opponent can receive from its raise action. To be more specific, we compute the amount of extra money the opponent can receive from its raise action compared to if it had chosen to call.

Assume the amount in the pot is , and the amount to call is . And then the amount the opponent raises is . Assume the opponent raises only when his probability to win is greater than , and that we fold when the probability to win is less than. In a particular round, assume our hand strength is and my opponent strength is . Then there are two cases:

**case 1: **

1.

In this situation, as we always fold, the opponent can’t receive any benefit from raising.

2. 

Since we still always choose fold, the opponent can receive benefit only when his actual hand is weaker than ours since a showdown would lead to all the money in the pot going to us. The total benefit is:



3. 

In this situation, as we will choose call, if we win, the opponent will lose and extra  for raising, and if we lose, the opponent will win an extra . Thus the expected return for the opponent is:



The total return for a raise action for the opponent is :



**case 2: **

In this situation, since when we fold, our win strength is always weaker than the opponent, the benefit of raising for the opponent is:



**Strategy 3:** Since we want to minimize the opponent benefit, and the opponent wants to maximize it, it should be:



We can prove that this is equivalent to:



Solving this equation will gives us a value 

**Strategy 4:** When it is our turn, the min-max algorithm defines that the opponent always minimizes our gain while we try to always maximize our gain. The min-max function is the same as eqn, we raise only when our probability to win is above:



**Describe/Analyze Properties**

**Method 1:** Previous work in poker AI has mostly focused on Nash Equilibrium, Monte Carlo, and other similar methods.[[5]](#endnote-5) The main reason these approaches are favored is due to the fact that poker is a game of imperfect information. It is stochastic, partially observable, competitive, and zero-sum. These properties of the game highly influence our design decisions. Particularly, we will explain why we used a simple rule based approach for the early game, and switch to Monte Carlo Simulations during the mid-game.

During heads up play, the pre-flop situation is very different that in multiplayer poker games. The only variables that matter are what cards we have, and what cards the opponent has. If we have a better hand than the opponent at this point in the game, then we should win strictly more than 50% of all games with this same configuration. Thus, as the number of games go to infinity, we will see a positive return on investment from always playing this hand. For this reason, we chose a sample of the 30 tops hands in Omaha poker and set our bot to always raise when we have these hands. Statistically, there are 270,725 possible distinct starting hands in Omaha poker.[[6]](#endnote-6) The top 30 hands comprise about .01% of these possibilities. Thus, if we have one of these hands, it is extremely likely that we have a better hand than the opponent and raising would be the optimum choice. The second rule we made was that if we do not raise, we will always call. This goes back to the fact that pot-limit rules disallow betting too much at the beginning of a game, so it makes sense to play more hands at a low cost to improve the probability of hitting a strong hand later in the round. Using these two simple rules, we were able to decide on a move extremely quickly, thus adding unused time to our time bank for when we might need extra computing time. Performing Monte Carlo simulation at this point would place a large strain on resources with little to no improvement in return on investment.

After the flop (when 3 cards are added to the table) the situation becomes much more complex as the possible hands each player can make is increased exponentially. At this point a rule based approach can no longer accurately estimate the relative value of each subsequent move. This is where we use the properties of a Monte Carlo simulation to our advantage. First, Monte Carlo simulates entire game states exactly. Thus, the stochastic and partially observable nature of poker can be nullified by simply randomly assigning cards and moves to the opponent. In this way, each single iteration of a simulation becomes deterministic. Next, the final value of a simulation is exactly how much money we can expect to win or lose in that simulation. With enough simulations, we can accurately gauge the best return on investment for each move. This is very different from simply simulating our cards versus a collection of possible opponent cards such as in method 2. As an example, simply winning a large number of games is not sufficient in poker to win the entire match. The combination of limits to betting and lack of bluffing causes extremely good hands to actually make very little money with each win. On the other hand, a mediocre hand can produce a large return on investment if played intelligently. The power of a drawing hand is also much higher in Omaha than other poker variants. Since players have 4 cards in their hands, they can produce many more combinations with cards on the table. A dead hand can become the strongest hand after the addition of just one new card. These types of situations are hard to see and judge for humans, but random simulations can easily determine how likely we are to draw a necessary card. With the power of a large number of simulations, we are assured within a certain confidence interval that the sum of values of each simulation can average out to an accurate relative value for that move. Since poker is a competitive no-sum game, any money we lose goes to our opponent and vice versa. Gauging the value of a move by how much money we can expect to make from it is the best path to victory. Monte Carlo simulations reduce the game to a simple deterministic match where we need only to pick the move that makes us the most money (or loses us the least).

**Method 2:** The Monte Carlo method is a direct method that can fairly estimate the change of winning any particular hand through a large number of situations. The min-max algorithm also gives us a very good estimate of the a way to maximize our own gain while minimizing the opponent gain. Compared to method 1, this method is simpler and more rule based. Since it is simpler, it will be easier to introduce a learning algorithm. For example, we can calculate the probability of the opponent raising after playing several rounds, and then use the data we collect to adjust our strategy of deciding when to fold and raise. On the other hand, the disadvantage of this method is that the estimate of the chance to win based on win probability may not be accurate, and thus we can't always make the optimal decision. For example, the win probability may not be normally distributed, and if we use this as a criterion to make our decision, the expected benefit isn’t accurately estimated, which can further lead us to making the wrong decision. However, method 1 doesn’t have this issue since it runs over all the possible moves by both opponents and estimates the expected reward at the final stage.

**Results and Discussion**

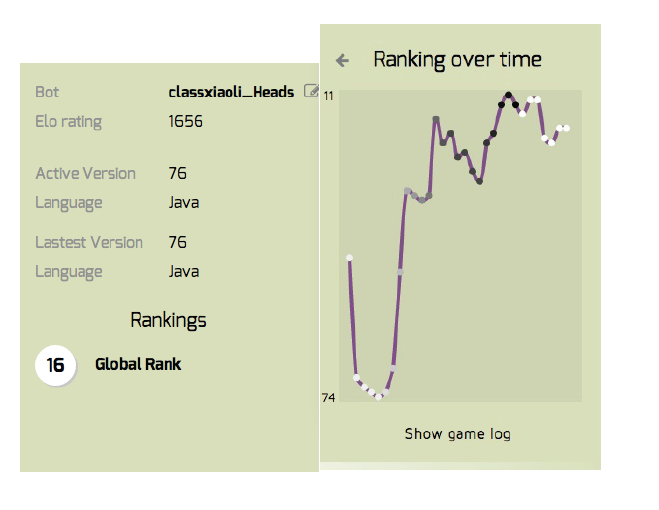
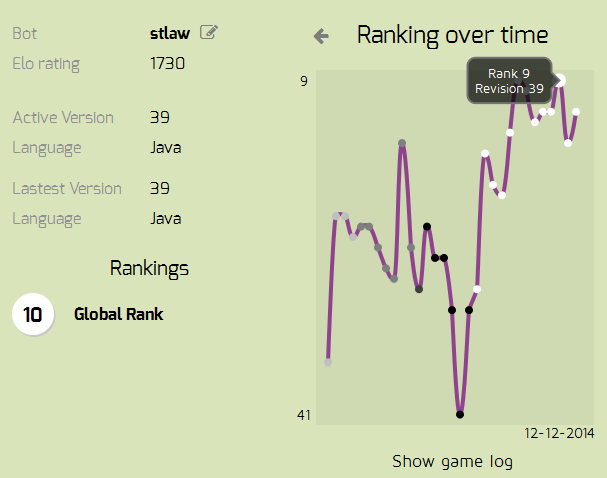
Both bots were submitted for testing for a one month period of time. They both returned with very positive results, though the bot implementing method 1 edged out slightly better in average ranking. The rankings are measured against 80 other active bots and each bot begins with an Elo rating of 1400. The initial bot sent in was essentially what we used as our "control". This bot would raise whenever it had a hand that had a value that was equal or greater than three-of-a-kind; otherwise, it would fold if the opponent raised. This bot showed fairly promising results and had an average ranking of 23. This is a fairly positive result and is based on the idea that the bot will only play good hands and bet everything on that hand. Obviously, this bot is quite easily exploited against more intelligent bots and thus, its ranking stagnated.

Our bot using method 1 was first implemented when the bot had a rank of 31. It immediately climbed the ranks very rapidly with a dominating win ratio of over 70%. We must note that the Elo rating system serves to match up players with equal ratings; thus, it is expected to have a win ratio close to 50% if matched up evenly. Therefore, this win ratio is remarkably impressive. This ascent in ranking continued until it hit its peak ranking of 9 out of 80, which was achieved over the span of 5 days. From then until its current standing, it has not dropped lower than rank 13 with its current Elo being 1730.

In analyzing the games played, we note that many of its losses were not due to bad calls. Instead, the bot made a correct call percentage wise, but was beat due to the opponent getting a winning card that had a fairly low chance of occurring. This is the nature of poker and cannot be helped. Instead, it is suitable to play for the long run and have an overall winning record, which allows one to climb in the Elo system.

If we evaluate the bot's individual moves, some of its moves seem to be unusually aggressive. However, over a collection of many similar moves, it is observed that the net winnings tend to be quite positive. This is due to the fact that this algorithm is based off of winnings and therefore, if the winnings are large enough, the bot will play accordingly. However, for the most part, this bot plays fairly "tight", meaning it does not take unnecessary risks. Its most volatile opponent are those that are extremely aggressive, to the point where they raise the bet on every turn. In these scenarios, our bot often folds repeatedly, putting themselves at a fairly sizeable deficit if a favorable hand does not show up. Consequently, this bot can take advantage of these aggressive bots if they have a winning hand because of the fact that it bets intelligently and is able to manipulate the betting phases to retrieve the most money possible. These overly aggressive bots are prominent in the ranks between 15-25, but against opponents in the top 10 rankings, their bots also utilize intelligent betting schemes and therefore, unlucky streaks are less bound to cause us to lose immediately.

The bot utilizing method 2 had very similar results, also peaking at rank 9, although dropping to a lower overall ranking of 16 when compared to method 1's bot. While they use similar methods, one main disadvantage that method 2 has is that it does not incorporate the monetary winnings for each simulated move. This is the feature that allows bot 1 to essentially "bluff" during a losing hand or slow play a winning hand, which lets them milk more money out of their opponent. Over the long run, this advanced detail will net a fairly significant difference in winnings.



Method 1 Method 2

In all, both bots utilize Monte Carlo simulation to achieve very positive results, both finishing with rankings in the top 20 and peaks in the top 10. Further modifications to the algorithm could include a more advanced pre-flop strategy that is not simply rule based and a learning algorithm. The learning algorithm will track the opponent's move and after a certain number of moves, it will use that data to alter the simulations, which will allow certain simulations to hold more weight over others and thus, providing more accurate simulations against specific opponents.

1. http://theaigames.com/competitions/heads-up-omaha [↑](#endnote-ref-1)
2. http://www.pokerlistings.com/strategy/potlimit-omaha-starting-hands [↑](#endnote-ref-2)
3. http://en.wikipedia.org/wiki/Omaha\_hold\_%27em [↑](#endnote-ref-3)
4. http://en.wikipedia.org/wiki/Monte\_Carlo\_method#Monte\_Carlo\_and\_random\_numbers [↑](#endnote-ref-4)
5. http://en.wikipedia.org/wiki/Computer\_poker\_players#Artificial\_Intelligence [↑](#endnote-ref-5)
6. http://en.wikipedia.org/wiki/Poker\_probability\_%28Omaha%29#Starting\_hands [↑](#endnote-ref-6)