Introduction
Poker is a game played with a standard 52 card deck, usually with 4 to 8 players per game. During each hand of poker, players are dealt two cards and must choose whether to bet on their hand, or fold and exit the round at a small loss. Due to the nature of the game and the interaction between players, poker strategy very much revolves around outwitting the opponent. Unlike other betting games like blackjack, there is no single strategy that is optimal at all times and against all opponents. The information available to each poker player is incomplete, and sometimes may be misleading. For example, an opponent may place a large bet; this would indicate that the opponent has a good hand, but in reality that player could be bluffing with a poor hand. For these reasons, an application of artificial intelligence to the game of poker could prove to be useful. The goal of this project is to create an AI system that is capable of beating a poker bot that implements a simple formula over an extended number of rounds.

Approach
The first step in the process to find a poker framework in java to modify. Several modifications were made to the game in order to simplify the task:
- The game was reduced to 2 players: an AI system, and a basic bot
- The number of betting phases was reduced from 4 (regular poker) to 1
- The betting options for each player were reduced from an analog system (players can bet however much they want) to a binary system (players can only bet a low value or a high value)
- The AI was always given the responding turn. This means that it always had the knowledge of the bot’s bet before making its decision.

Given the nature of the problem, it was decided that Q-learning would be the most effective method for the AI system. Q-learning will be able to adapt to the different strategies of different opponents and will overcome the noise in the data produced by the randomness of the card draws for each player.

Action Algorithm
In each round, the AI system must take an action based on the information available to it. Every round, it knows the cards in it’s own hand, and it knows whether the opponent bot placed a high bet or a low bet. For the purpose of the algorithm, this was defined in the program as a state:
The information contained in a state is then passed into the action algorithm. The action algorithm creates 2 alternate possibilities based on the state: one where it folds on the state, and one where it bets on the state. These two alternate possibilities are called state-actions. The AI will check its database to see if either of the state-actions have already previously occurred, and find the associated value of each state action. Generally, it will return the action associated with a higher value:

```java
public boolean action(int botBet, Hand h) {
    HashSet<Card> set = new HashSet<>();
    set.addAll(h.cards());
    State s = new State(set, botBet);
    StateAction sa1 = new StateAction(s, true);
    StateAction sa2 = new StateAction(s, false);
    double value1 = 0;
    double value2 = -1;
    Object o1 = data.get(sa1);
    boolean b1 = (o1 != null);
    Random rand = new Random();
    int n = rand.nextInt(10) + 1;
    if (b1) {
        value1 = data.get(sa1);
    }
    if (value1 > value2) {
        return true;
    } else if (value1 <= value2) {
        if (n > 1) {
            return false;
        } else {
            return true;
        }
    }
    return true;
}
```

If the learned value of betting is greater than the value of folding, then the AI will bet. However, even if the learned value of betting is lower than the value of folding, 10% of the time the AI will still bet. The reason for this is that it takes several data points to accurately determine the value of a state-action. Each state-action has a probability of winning, and must be tried several times before eliminating an action as a viable possibility from that state.

If the AI doesn't retry betting on states where it has previously lost, then it will be folding excessively.

One thing that is important to note is that only the rank of each card in the hand was considered, not the suit. This was done to reduce the total number of possible states. There are 2652 combinations of 2 cards when suit is considered, but only 169 combinations if only card rank is considered.
Learning Algorithm
The AI must learn from the results of each round. In order to do this, it stores a key-value pair in a hashmap after every round, where the key is a state-action, and the value is the result of the round.

```java
public void learn(StateAction sa, double result){
    Object o = data.get(sa);
    if (o != null) {
        double old = data.get(sa);
        double newval = (old * 0.95) + (result * 0.05); // alpha is 0.05
        data.put(sa, newval);
    } else {
        data.put(sa, result*0.05); // if current SA has never occurred
    }
}
```

A hashmap is used so that the lookup time to find the learned value of a state-action remains constant. The alpha value of 0.05 was chosen after experimenting with a range of values. Larger alpha values result in the AI eliminating the option of betting from certain states too quickly.

Opponent Bot
In order to test the effectiveness of the AI, a very basic bot was created as an opponent:

```java
double score = bot.chenFormula();
int table;
Random rand = new Random();
int n = rand.nextInt(10) + 1;
int botRandomness = 0;
if ((score >= 7) || (n <= botRandomness)) {
    table = 1 + 10;
} else {
    table = 1;
}
```

The chen formula is used by the bot to evaluate the strength of its hand. The chen formula is regularly used by professional poker players as a basic indicator of hand strength. The bot has 2 variables that can be modified: the chen formula score required to bet, and the randomness. These parameters were designed for easy modification so that the AI could be tested against a wider variety of types of opponent.
Results:
In order to test the AI’s learning speed as well as adaptability against various opponents, it played 100 games of poker, each consisting of a number of rounds between 100 and 10000. For each combination, the average score per round was recorded. A range for the average score is indicated. The results are as follows:

<table>
<thead>
<tr>
<th>Opponent</th>
<th>100 games of 100 rounds</th>
<th>100 games of 1000 rounds</th>
<th>100 games of 10000 rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>100% random bot</td>
<td>-0.10 to 0.10</td>
<td>0.04 to 0.16</td>
<td>0.08 to 0.10</td>
</tr>
</tbody>
</table>
| 40% random bot  
Minimum chen score = 3 | 0.08 to 0.30 | 0.16 to 0.27 | 0.2 to 0.22 |
| 40% random bot  
Minimum chen score = 8 | 0.03 to 0.20 | 0.15 to 0.20 | 0.16 to 0.19 |
| 0% random bot  
Minimum chen score = 3 | 0.05 to 0.42 | 0.23 to 0.33 | 0.3 to 0.32 |
| 0% random bot  
Minimum chen score = 8 | 0.06 to 0.31 | 0.17 to 0.28 | 0.24 to 0.26 |

Analysis:
The first thing to note is that the lower limit of performance for the AI always increases as the number of rounds per game increases. This is as expected, because the values of each state-action can be more accurately derived if there are more data points for each state-action. Because there are

Another correlation that can be observed is the effect of the bot’s minimum chen score on the the AI’s performance. When the bot has a minimum chen score of 3, it bets more frequently and on weaker hands. Once the AI has gathered enough information, it can realize that the bot bets high even on weaker hands. A human player might be intimidated by a high bet, but in this scenario the AI learns the trends of the bot and will respond by betting. This leads to a greater average score per round. On the other hand, when the bot has a minimum chen score of 8, the AI’s performance actually decreases. Although the AI is likely winning a similar number of rounds as the previous scenario, the bot is betting high far less frequently, so the potential winnings for the AI are also smaller.
The last and most interesting trend in the results is the effect of the bot’s randomness on the AI’s performance. Although one would describe the chen formula bot with 0% randomness to be “smarter” than the 100% randomness bot, the AI actually has the worst performance against the most random bot, and the best performance against the least random bots. This is because the AI cannot learn anything meaningful from an opponent that acts randomly. However, when an opponent acts in a predictable manner, the AI can use this to it’s advantage to predict the strength of the opponent's hand, and decide whether to bet or not.

**Conclusion**

The AI system designed for the simplified poker game was able to achieve positive scores against all the different bots that it played against. However, it did require a large number of rounds to be played for optimal performance. One thing that could be done would be to increase the alpha value for the learn algorithm, while also forcing the AI to retry state-actions with a learned negative value. Another area of improvement for the AI is playing against seemingly random opponents. Professional poker players change their strategies frequently to make sure that their opponents cannot decipher their actions; the AI must be able to overcome this. However, the AI must also become random itself, in order to be an effective opponent against human players. The current state of this project’s AI is very predictable. A further step would be to make its actions seem more random without compromising the value of each move.