Exploring the Potential of Non-Learning AI in Poker

Daniel Persson
daniel.p-00@hotmail.com

Elias Vahlberg
elias.vahlberg.2@gmail.com

Hamed Haghjo
hamedhaghjo@hotmail.com

Mohammad Omar Abdullah
omarabd9904@hotmail.com

Halmstad University
Master of Science in Computer Science and Engineering

***Abstract*—The objective of this project was to exhibit the feasibility of utilizing artificial intelligence techniques to enhance performance in the popular card game Poker, specifically its simplified version, 5-Card Draw. To accomplish this objective, a mix of a model- and a utility-based agent was implemented with several features, such as the capability to evaluate its own hand, employ Iterative Deepening Depth-First Search (IDDFS) to investigate the future utility of actions and adapt its strategy based on a behavior model of the opponents. The agent was tested against a reflex agent created for benchmarking. The testing results indicate that the agent performed very well, demonstrating its effectiveness in making strategic decisions. However, the agent did not place first in its group during the tournament due to a few oversights. One oversight with the agent was the inability to ignore opponent actions when irrational behavior was displayed.**

***Index Terms*—Reflex Agent, 5-card poker, hand strength evaluation, poker strategy.**

I. Introduction

A. Poker Game

Poker is a widely renowned card game that has experienced a significant surge in popularity [2]. The game is played with a standard deck of 52 cards, and the objective is to attain the highest-ranked hand of cards by the game's conclusion. In descending order of value, the hand ranks are as follows: Straight flush, Four-of-a-kind, Full House, Flush, Straight, Three-of-a-kind, Two-pairs, Pair, and High card. This project will be centered on the *5-Card Draw* poker variant, running on the same principle hand ranking and considered one of the most straightforward poker variants. The ultimate objective of this project is to create a Poker Agent that can compete against other agents in an Artificial Intelligence course, with the ultimate goal of being selected as the best agent in a pre-tournament to compete in the final tournament, where the winner will be determined based on their performance. Thus, this project will provide a unique opportunity to apply artificial intelligence techniques and strategies to the game of poker and to challenge and improve the skills of the agents created.

The following are fundamental actions in the game:

- **Opening** - The initiation of the initial voluntary wager in a betting round is referred to as opening the round. It is also referred to as opening the pot in the first betting round.
- **Calling** - Matching an opponent's wager or raise is called calling.
- **Checking** - If no player has initiated the betting round, a player may pass or check. Checking involves declining to make a wager, indicating a desire to retain one's cards and the right to call or raise later in the same round if an opponent initiates. If all players check, the betting round concludes with no additional funds placed in the pot apart from the ante from each player.
- **Raising** - Increasing the wager size required to remain in the pot, thereby forcing all subsequent players to at least match the new amount to stay in the round, is referred to as raising. The opening is a particular case of raising, executed when no other player has raised. A player making the second or subsequent raise in a betting round is referred to as re-raising. The raise in AI Poker must be at least as high as the previous raise (if any). If a bet has been placed that the player, in turn, cannot match, the player must fold unless he chooses to go all-in. A player must match the bet and cannot check or call with a lesser amount.
- **Folding** - Discarding one's hand and forfeiting interest in the current pot is called folding. The folding player requires no further wagers but cannot win.
- **Ante** - An ante is a mandatory wager in which each player places an equal amount of money or chips into the pot before the deal begins. It is a common feature in many poker games and serves as a means of building the pot before the start of the game.

Client and server files were made available to simulate a game of 5-Card Draw within a virtual environment. Before initiating any connections with the server, the game settings must be configured through the provided Poker Server GUI. Once the configuration process has been completed, clients may be connected to the server, representing the Poker AI players participating in the poker game.

Additionally, the structure of each playing round in the game is as follows:

1) The round starts with the server informing all players of the amount of chips each has.
2) Each player is then dealt five cards from the server.
3) A forced bet, known as the *ante*, is taken from each

player and added to the pot.
4) The first betting round begins.
5) The first action a player takes is bound to the player, meaning changing the action is not permitted. The first action after the *ante* must be to *check* or *open*.
6) After the first bet, players have the possibility to draw.
7) Players can discard some of their cards in the draw phase and receive new ones as replacements. The player must decide which cards they want to discard.
8) The second betting round commences.
9) The showdown determines the winner of the round, who receives the contents of the pot. In the event of multiple winners, the pot is split among them.

B. *Possible Approaches*

From an AI perspective, 5-Card Draw and other poker games are classified as incomplete information games (IIG) [4]. It is imperative to thoroughly comprehend the game's mechanics to implement suitable artificial intelligence methods and create a highly effective poker agent. One potential strategy for achieving this objective is using a Memory-based agent. This agent would assess the strength of its hand and the actions of other playing agents by utilizing data collected from previous rounds, enabling it to make informed decisions based on opponents' tendencies and betting patterns (this was the strategy chosen for the agent WildCard Willy).

Another possible approach would be to use reinforcement learning to analyze to create different strategies for different opponents. Updating the strategies using TD-Learning or Q-Learning. The performance of such a method is described in the paper "*Adapting Strategies to Opponent Models in Incomplete Information Games: A Reinforcement Learning Approach for Poker* " [4]. The issue with this approach is that the strategies take many games to converge. The agent in the mentioned paper learned from 100 000 simulated games [4]. When only facing an opponent for a few games, the strategy can be quite ineffective. Another complication is scheduling the model, updating, predicting, and collecting data while still meeting response deadlines not to get disqualified. The paper "*A Reinforcement Learning Algorithm Applied to Simplified Two-Player Texas Hold 'em Poker*" concurs with the choice not to use TD and Q-learning for IIGs [1].

II. Method

A. *Environment*

The environment of the 5-card draw poker game described in the provided information and code can be divided into two parts: *actions* and *percepts*. The actions include the choices that players can make during the game, such as opening, raising, calling, and folding. Percepts encompass information accessible to players, including player chip count, dealt cards, and current pot state. The agent interacts with the environment by making decisions based on the game's percepts and rules, such as the probability of winning the hand and the hand strength. The agent uses the normal distribution of hand strengths to take action.

Agent actions:

OpenAction a_1: Action during opening round of a new dealing. $a_1 = (a, b) \in \{\{"Check" : 0, "Open" : 1\}, \{intbet\}\}$.
Action specific percepts:
{int[] $playersCurrentBet$, int[] $playersRemainingChips$}.

CallRaiseAction a_2: Action during round once pot has been opened. $a_2 = (a, b) \in \{\{"fold" : 0, "call" : 1, "raise" : 2\}, \{\text{int } bet\}\}$.
Action specific percepts:
{int[]$playersCurrentBet$, int[]$playersRemainingChips$}

CardsToThrow a_3: Action after eah round. $a_3 = (c) \subset \{card1, card2, card3, card4, card5\}$.
Action specific percepts:
$\emptyset$.

Table I displays the list of percepts.

$\mathbf{p_n}$	**Info type**	**Content**	**When it's updated**
p_1	**Chips**	int chips	After each round, OpenAction (a_1), CallRaiseAction (a_2), and after dealings.
p_2	**CurrentHand**	String[] hand	At the beginning of dealings and after CardsToThrow (a_3).
p_3	**Ante**	int ante	Before round starts.
p_4	**CurrentBet**	int currentBet	Before player's OpenAction (a_1) and CallRaiseAction (a_2).
p_5	**Round**	int round	After new round.
p_6	**PlayerChips**	String player, int chips	Before dealings, after oponents OpenAction (a_1), CallRaiseAction (a_2), and after dealings.
p_7	**AnteChanged**	int ante	After ante changes.
p_8	**ForcedBet**	String player, int forcedBet	At the beginning of each dealings.
p_9	**PlayerOpen**	String player, int openBet	After opponents OpenAction (a_1)
p_{10}	**PlayerCheck**	String player	After opponents OpenAction (a_1)
p_{11}	**PlayerRise**	String player, int amountRaisedTo	After opponents CallRaiseAction (a_2)
p_{12}	**PlayerCall**	String player	After opponents CallRaiseAction (a_2)
p_{13}	**PlayerFold**	String player	After opponents CallRaiseAction (a_2)
p_{14}	**PlayerAllIn**	String player, int allInChipCount	After opponents CallRaiseAction (a_2)
p_{15}	**PlayerDraw**	String player, int cardCount	After opponents CardsToThrow (a_3)
p_{16}	**PlayerHand**	String player, String[] hand	After dealings is over.
p_{17}	**RoundUWin**	String player, int winAmount	After all players have folded.
p_{18}	**RoundResult**	String player, int winAmount	After after dealings is over.

TABLE I
TABLE CONTAINING ALL PERCEPTS, THEIR CONTENTS, AND WHEN IT IS UPDATED.

B. *Evaluation*

To determine the strength of the agent's hand, it needs to produce a single numerical metric from any possible hand.

This task is done by the *handEvaluation* function. Hand evaluation is done during *queryOpenAction*, *queryCallRaiseAction*, and *queryCardsToThrow*.

An appropriate measurement for assessing the strength of a poker hand is the likelihood that no other player will hold a superior hand. This probability can be mathematically expressed as $(1 - P(X >= H))^N$, where $P(X >= H)$ represents the probability of drawing a hand that is equal to or superior to the current hand in question, and N denotes the number of other players involved in the game. The frequency of each unique hand type, such as "Pair" or "Flush," can be depicted in Figure 1.

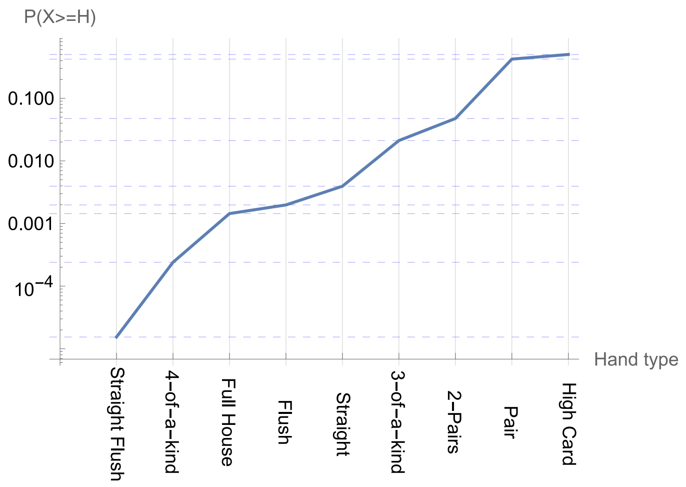

Fig. 1. Probability to draw a hand equal to or superior to each distinct hand type.

Since the probability of receiving a hand strength is approximately linear between each distinct type, the hand strength could be viewed as a type: integer number $H_t \in \{\mathbb{Z}; 1 \leq H_t \leq 10\}$ and a rank/suit: a fraction $H_{rs} \in \{\mathbb{R}; 0 \leq H_{rs} \leq 1\}$ for the final value $H = H_t + H_{rs}$. However, this approach presents challenges in implementation and execution time, as finding the values of H_t and h_{rs} for a 5-card hand can be cumbersome. Another issue is that python is a comparatively slow language, and the agent would need more time to find each move. To address this, a library called *PokerHandEvaluator*[1] was utilized, which utilizes the python C++ interface. This library, which is based on the article "Cactus Kev's Poker Hand Evaluator" [3], converts a poker hand to an *E-value* between 1 and 7462 (for each distinct combination). The *E-value* 1 is assigned to the best possible hand, and 7462 is the worst. However, this value alone is not sufficient to determine the probability of a hand, as the probability is not proportional to the *E-value*. To calculate the probability for any given E-value, we can use the probabilities from figure 1 and find the lowest and highest *E-value* for each hand type and linearly interpolate between them. The resulting probability graph can be shown in figure 2.

[1]https://github.com/HenryRLee/PokerHandEvaluator

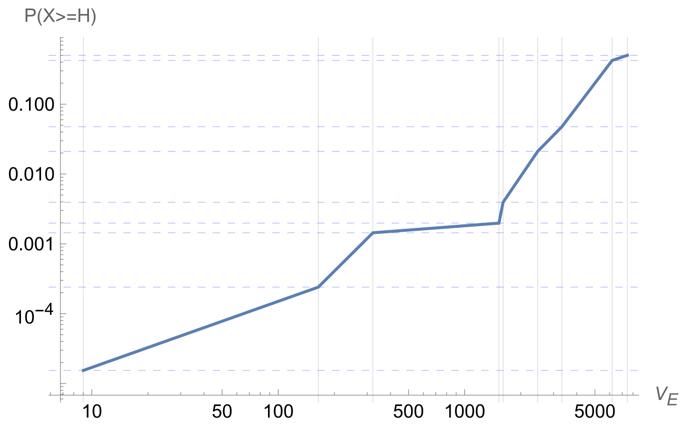

Fig. 2. Probability to draw a hand equal or better for each *E-Value* (V_E).

Structure of the *handEvaluation* function is shown below.

```
def handEvaluation(self, hand, num_players):
    global K_xvals, P_yvals
    hand_p = np.interp([evaluate_cards(*hand)],
     K_xvals, P_yvals)[0]
    return (1 - hand_p)**(num_players-1)
```

The function *evaluate_cards* is the one imported from *PokerHandEvaluator*. The number of times *handEvaluation* could be called during a 1-second turn was 128363. This number may vary based on the computer.

C. Search

During a *queryCardsToThrow* call, the agent needs to find the optimal cards to throw to improve its hand. The agent uses IDDFS[2] to explore each possible action. Each action is simulated with a random sample of replacement cards. The sample hands are evaluated using the *evaluate_cards* function and averaged to get an expected value for the specific action. It then repeats the process for each sample. It stops once a certain time (depending on the turn length has passed) and takes the best action based on the expected value from the last completed depth in the search tree. Figure 3 shows an example of one such search tree.

[2]Iterative Deepening Depth First Search

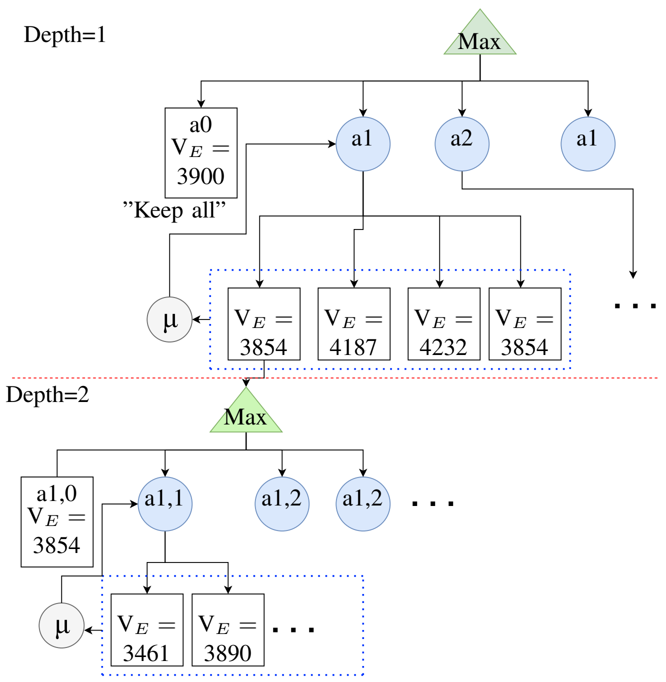

Fig. 3. Cards to throw, example search tree (*Circle = chance nodes, Square = value node, Triangle = max node, μ = averaging node*).

D. Adaptation

An adaptive mechanism addressing dangerous behaviors was implemented to improve the agent's abilities. This mechanism incorporated a hand evaluation threshold into the agent's decision-making process, allowing it to fold in situations where its hand was determined to be below the threshold and the opponent exhibited a strong hand. The primary objective of this adaptation strategy is to diminish the occurrence of costly calls made by the agent in unfavorable situations.

Additionally, the agent was furnished with the capability to evaluate the opponent's betting patterns and adapt its strategy accordingly. This was achieved by analyzing the opponent's past behaviors and identifying any patterns or tendencies that could be exploited. For instance, if the opponent tended to bluff, the agent could adjust its strategy to call more frequently, taking advantage of the opponent's tendency to over-bet. This was accomplished by assuming that the opponents acted rationally. By utilizing this assumption, the agent could infer the opponent's likely range of hands based on their betting patterns and adjust its strategy accordingly.

Implementing this adaptation mechanism is expected to improve the agent's performance by enabling it to exhibit more cautious and calculated behavior. Consequently, the frequency of costly mistakes made by the agent will be anticipated to decrease, resulting in more efficient and effective performance. The pseudo-code below gives an insight into the agent's evaluation of the opponent based on its betting history.

```
**Procedure:** evaluate_and_adapt_strategy(o_b_h)
**Input:** o_b_h (opponent betting history)
**Output:** None
$patterns \leftarrow$ analyze_betting_patterns(o_b_h)
$tendencies \leftarrow$ identify_tendencies($patterns$)
**if** "bluffs frequently" in $tendencies$ **then**
  adjust_strategy("call more frequently")
**else if** "rarely bluffs" in $tendencies$ **then**
  adjust_strategy("call less frequently")
**else**
  No exploitable tendencies identified. Maintain current strategy
**end if**
**end procedure**
```

```
**Procedure:** analyze_betting_patterns(o_b_h)
**Input:** o_b_h (opponent betting history)
**Output:** bluff_ratio
$bluff\_count \leftarrow 0$
$bet\_count \leftarrow 0$
**for** $i$ in o_b_h **do**
  **if** $i$.action = "bluff" **then**
    $bluff\_count \leftarrow bluff\_count + 1$
  **end if**
  $bet\_count \leftarrow bet\_count + 1$
**end for**
$bluff\_ratio \leftarrow bluff\_count / bet\_count$
**return** $bluff\_ratio$
**end procedure**
```

```
**Procedure:** identify_tendencies(patterns)
**Input:** patterns
**Output:** tendencies
**if** $patterns > 0.3$ **then**
  **return** "bluffs frequently"
**else if** $patterns < 0.1$ **then**
  **return** "rarely bluffs"
**else**
  **return** "normal betting pattern"
**end if**
**end procedure**
```

```
**Procedure:** adjust_strategy(new_strategy)
**Input:** new_strategy
**Output:** None
**if** $new\_strategy =$ "call more frequently" **then**
  Increase the calling rate
**else if** $new\_strategy =$ "call less frequently" **then**
  Decrease the calling rate
**end if**
**end procedure**
```

E. Approach

The agent's decision-making is a combination of the three different parts mentioned in sections II-B, II-C, II-D. The logic for each separate action is listed in the figures below.

OpenAction $\mathbf{a_1}$:
Methods: {handEvaluation,analyzeBettingPatterns, adaptStrategy}
Percepts: $\{p_1, p_2, p_1 1, p_1 2, p13, p_1 5\}$
Formally represented as :

$$f_{a_1}(p_1, p_2, p_{11}, p_{12}, p_{13}, p_{15}) = (h_{eval}(p_1, p_2), h_{mem}(p_{11}, p_{12}, p_{13}, p_{15}))$$

Action type: $\text{h}_{eval(t)}(p_1, p_2) = f_{eval}(p_2)$
$$\begin{cases} "Open" & x * f_{eval}(p_2) > 0.25 \land p_1 > l_o \\ "Check" & 0.25 \leq x * f_{eval}(p_2) \end{cases}$$
Bid: $\text{h}_{eval(b)}(p_1, p_2) = min(p_1,$
$$\begin{cases} l_o + \frac{(f_{eval}(p_2) l_o x)}{2}) & h_{eval(t)}(p_1, p_2) = "raise" \\ 0 & true \end{cases})$$
Where l_o is the minimum pot after open. Let x be a random variable following the uniform distribution in the range $[0, 1)$. For $h_mem(...)$ see section II-D.

CallRaiseAction $\mathbf{a_2}$:
Methods: {handEvaluation, analyzeBettingPattern, adaptStrategy}
Percepts: $\{p_1,p_2,...\}$
Formally represented as :

$$f_{a_2}(p_1, p_2, p_{11}, p_{12}, p13, p_{15}) = (h_{eval}(p_1, p_2), h_{mem}(p_{11}, p_{12}, p_{13}, p_{15}))$$

Action type: $\text{h}_{eval(t)}(p_1, p_2) = f_{eval}(p_2)$
$$\begin{cases} "raise" & x f_{eval}(p_2) > 0.50 \land p_1 > l_r \\ "call" & 0.5 > x f_{eval}(p_2) > 0.25 \\ "fold" & 0.25 \leq x f_{eval}(p_2) \end{cases}$$
Bid: $\text{h}_{eval(b)}(p_1, p_2) = min(p_1,$
$$\begin{cases} l_r + \frac{(f_{eval}(p_2) l_r x)}{2}) & h_{eval(t)}(p_1, p_2) = "raise" \\ 0 & true \end{cases})$$
Where l_r is the minimum amount to raise to. Let x be a random variable following the uniform distribution in the range $[0, 1)$. For $h_{mem}(...)$ see section II-D.

CardsToThrow $\mathbf{a_3}$:
Methods: {searchCardTrow, handEvaluation}
Percepts: $\{p_2\}$
Formally represented as:

$$f_{a_3}(p_2) = h_{search}(p_2)$$

$h_{search}(p_2)$ uses the search algorithm described in section II-C. The result is a subset of the cards in hand, i.e., $h_{search}(p_2) \subseteq p_2$.

III. Results

A. Performance

Results from 90 games with 400 starting chips (1vs1) for reflex, utility, and model + utility agents is shown in table II.

Agent vs.	Reflex(1)	Utility(1)
Utility(2)	{39,51}	-
Model+Utility(2)	{13,77}	{33,57}

TABLE II
RESULTS FROM 90 GAMES OF EACH COMBINATION OF OPONENTS, WITH THE FORMAT {AGENT(1) WON TIMES,AGENT(2) WON TIMES}.

B. Tournament Results

The preliminary evaluation results of the agent's performance in the tournament are presented in Table III. The agent WildCard Willy placed second, having won 14 out of 50 games, representing a win rate of 28%. The agent RareQueens placed first, with a win rate of 42%, by winning 21 out of 50 games. However, it is essential to acknowledge that several agents were disconnected from the tournament due to incorrect responses to the server, which may have influenced the tournament's outcome. It is also worth noting that the sample size of the tournament, being limited to 50 games, may not fully reflect the agents' overall performance. Therefore, it is crucial to interpret these results cautiously and consider them as a preliminary evaluation. Further testing and analysis are necessary to evaluate the agents' performance comprehensively.

Placement	**Agent**	**Wins**
1	RareQueens	21
2	WildCard Willy	14
3	PotDigger	8
4	Bames Jond	7

TABLE III
PRE-TOURNAMENT RESULTS FROM 50 GAMES

IV. Conclusion

Creating a poker agent that performs well with unknown opponents is a challenging task. The approach chosen was to have one heuristic that focused on analyzing its hand and evaluating its strength. The other heuristic was to analyze opponents' behavior and adapt their strategy. These heuristics were weighted and combined to produce one action for each request. Something that became apparent was that the opponent analysis heuristic was more of a challenge than an evaluation. The primary behavior that was analyzed was the tendency to bluff since the opponent's strategy in the tournament was utterly unknown. A point of improvement would be to ignore the behavior of agents with a high degree of randomness or with a bluffing pattern that is too complex to analyze. Another improvement would be to *"fake"* a predictable pattern and switch for high stakes rounds. These improvements or any improvements proposed also have their counter. Thus, the best improvement will always depend on the opponent it is facing.

References

[1] Fredrik A Dahl. A reinforcement learning algorithm applied to simplified two-player texas hold'em poker. In *European Conference on Machine Learning*, pages 85–96. Springer, 2003.

[2] N. WILL SHEAD, DAVID C. HODGINS, and DAVE SCHARF. Differences between poker players and non-poker-playing gamblers. *International Gambling Studies*, 8(2):167–178, 2008.

[3] Kevin Suffecool. Cactus kev's poker hand evaluator, 2005.

[4] Luís Filipe Teófilo, Nuno Passos, Luís Paulo Reis, and Henrique Lopes Cardoso. Adapting strategies to opponent models in incomplete information games: a reinforcement learning approach for poker. In *International Conference on Autonomous and Intelligent Systems*, pages 220–227. Springer, 2012.