Department of Mathematics & Computer Science
University of Southern Denmark | IMADA
Friday 31st May, 2024

A Study on Poker Agents
DM879: Artificial Intelligence

Group Name

Kappa, κ

Authors
Kian Banke Larsen
kilar20@tudent.sdu.dk

Magnus Victor Boock
maboo20@tudent.sdu.dk

Thomas Wulff Heissel
thhei20@student.sdu.dk

Contents

1 Introduction **1**

2 The Game **1**

3 A Tale of Heuristics **2**
3.1 Studying a Relaxed Version of the Problem 2
3.2 The Main Principles of Poker 3
3.3 A Genetic Approach to a Heuristic 3
3.3.1 Individuals 4
3.3.2 Genetic Algorithm 6
3.3.3 Monte Carlo Simulations 7

4 Results **7**
4.1 Playing against other strategies 8

5 Discussion **10**
5.1 Pattern databases 10
5.2 The Definitive Winner 10

6 Conclusion **11**

1 Introduction

This paper explores a venture into the creation and development of a strategy, to be implemented by some artificial agent, which has the goal of playing the game of *Heads Up Texas Hold'em Poker*. Given that the game of Poker is a game of chance and deception, to build such an artificial intelligence, we would need to rely on a heuristic estimation approach to the problem. Opponents may *bluff* or they may actually have a decent pair of cards in their hand in combination with the cards on the table, this we do not know and similarly the opponent does not know which of the cases we rely on. We begin by introducing the game at hand. Then, we should analyse the game being played by making use of different approaches solving adversarial game search learned from the course.

2 The Game

We will be focusing on a Poker variation known as *Heads Up Texas Hold'em*. Each player is dealt two cards and some currency units. The game then consist of four betting rounds, where five playing cards are gradually revealed on the board. The outcome of these cards might change the players confidence in their own chances of winning, but their goal remains the same: win the opponents units and maximise personal bankroll. The players units are reset after each game round, but the bankroll is accumulated throughout a number of games being played in a tournament. We rely on a custom version of the MIT Poker Engine to facilitate the game and provide legal actions, as our goal is to develop a strategy for an agent and not the engine itself. This framework defines a bot based on four methods: `init`, `handle_new_round`, `handle_round_over`, and `get_action`. Thus, we only need to consider what should happen on these events. Our implementation focuses mainly on strategysing the retrieval of the next action for the currently moving player (the `get_action` method). So, every time it is *our* turn, the agent observes what actions are legal and selects one based on the characteristics of the round and its traits.

```
Your stack: 399
Opponent stack: 398

Your cards: [Card("Qc"), Card("2d")]
Cards on board: []

Your actions are: [
    (0, <class 'skeleton.actions.FoldAction'>),
    (1, <class 'skeleton.actions.RaiseAction'>),
    (2, <class 'skeleton.actions.CallAction'>)
]
Select an option: <user_input>
```

Figure 1: Query interactive player for an action.

A game is initialised in `Game`. The game can be played with any number of players, as the `run` method only require a list of players and tournament rounds to

play. The game can be played interactively using `players/interactivePlayer`. This player will query the user for what actions to perform, indicated in Figure 1. It is also possible to initialise a game using `players/evolutionpPlayer`, which has logic implemented to make its own decisions. The engine has been modified to be quite flexible, as it can include both human and artificial players.

3 A Tale of Heuristics

The game of Poker is essentially a game of luck, deception and heuristics. This yields a search space which is partially observable and nondeterministic, two ingredients to complicate adversarial game search problems even more than usual. In this section we discuss our approach to solve this problem and the different considerations we have made.

3.1 Studying a Relaxed Version of the Problem

There are multiple sub-problems throughout the game of Poker, initially, an agent should consider whether or not it should play or stay from an initial state, where it has been given a pair of cards. After considering the initial step, an agent should consider the betting aspect of the game, namely, when it wishes to continue playing given the current state, it should consider how much should be bet to maximise its expected winnings.

Our initial thoughts on how to approach the problem was to develop an agent which made use of the *expectiminimax* algorithm, which is a variation of the *minimax* algorithm for two-player, zero-sum games where the outcome is determined by a combination of skill and elements of chance. Initially, we ignored the concept of betting and only focused on the choice of staying in the game or not. Since our problem is not only nondeterministic but also partially observable, our state space would still be exponentially large and complicated to comprehend since each state consists of multiple belief states, even though the search tree to explore is of fixed height, because of the five rounds in Poker.

Thus, for the expectiminimax algorithm to make an intelligent decision at the initial state of the game, it would need an estimation of the strength of its own two cards. There are various ways this could be estimated, we have chosen to make use of *Monte Carlo simulations*, where we estimate the probability of winning by simulating games with the given hand. The number of simulations made determines the preciseness of the estimation since Monte Carlo simulations converge towards optimal solutions. Therefore a higher number of simulations would be desirable but the limitation is that making more simulations takes more time. One problem arises when following this approach, it will solely base its strategy upon the results of the simulations. Therefore, the developed heuristic for expectiminimax will essentially play whenever it is at a benefit by following this approach. Similarly, this will repeat when moving further in the game and more of the universe becomes observable to the agent. This will just make the simulation of hand strength more precise with less time, since there are less variables to consider.

This observation tells us that we would essentially make an agent which runs a number of simulations on how its current position in the game is against some opponent in every state. But the agent would still have to bet some amount of

currency. Of course, if the agent believes that it is going to win, it should enter its entire pot into the game, but when doing so in the root of the game you eliminate the possibility of betting further into the game. This simply yields a *showdown agent*, which goes all in at the beginning of the game or folds. Thus, we were in need of other strategies for our agent.

3.2 The Main Principles of Poker

To improve upon the agent discussed in the previous subsection, we began studying and learning different approaches and methods used in the professional Poker scene. Generally speaking, a Poker strategy is build upon four principles:

- *Strength*: One should bet their strong hands, check their middling hands and fold or bluff their weakest hands.
- *Aggression*: Aggressive actions, such as raising, are better than passive actions like checking and calling.
- *Betting*: A bet should be able to accomplish at least one of the following: it should force a better hand to fold, a weaker hand to call and cause drawing hands to draw at unfavorable odds.
- *Deception*: One should never follow the same principle all the time.

The strength principle is stating that one should place trust in the estimation of hand strength. The aggression principle has its benefits in the way that it introduces an additional way of winning, namely by making your opponent fold to your bet, in combination with the probability of winning at the showdown. The betting principle should limit the way bets are made and in what amounts one makes a bet. It should not be too aggressive, since you want weaker hands to follow, but also not too passive, since you want to make your stronger opponents fold, which is quite contrary to the agents strategy in the previous section where we do not maximise the outcome from our weaker opponents. Deception, being a central part of Poker, is important in the way that an opponent should not be able to figure out what ones strategy is. When playing deceptively, we essentially claim to the other agent, that have a better and higher estimation of our probability of winning over them, than we actually have.

3.3 A Genetic Approach to a Heuristic

The game of Poker is not simple to implement a beneficial strategy for, but given the four principles in the previous section, we might be able to generate one, without explicitly determining how an agent should play. This can be accomplished by making use of an evolutionary approach to evolve the strategy of an agent based on the principles of Poker, through many iterations and tournaments.

The evolutionary algorithm, which we have chosen, operates by first generating a random population of a certain size. Each individual in this population then competes in x tournaments, with each tournament comprising of y rounds. This entire process constitutes what we refer to as a generation. Following each generation, a certain amount of the *fittest* individuals undergo mutation

and recombination, giving rise to new offspring that populate the subsequent generation. This cycle continues, driving the evolution of the population.

We can model the four principles as a parameterised model that uses each of the principles in different ways but still in a coherent decision making strategy. This kind of modeling allows us to use a genetic algorithm to let a population of randomly assigned individuals in the population progress towards better strategies over generations, in the hopes of finding a *good* strategy for our agent.

3.3.1 Individuals

In order to use a genetic algorithm, we should first define the individuals that will make up the population.

As previously stated, we want to model the four principles as a parameterised model, and this model needs to work in the context of genetic algorithms. Hence, we want to be able to inherit and mutate parts of individuals, allowing for individuals in our population to learn from the best performing individuals, while also differentiate from them as to not get stuck in local optima for the population. All of the parameters defining the behaviour in terms of the four principles are described below, note that all parameters take on values within the domain $[0; 1]$. Let p_{win} be the estimated probability of winning at each state in the game, then

- **strength s:** a threshold for p_{win} used to decide whether to raise, i.e. how much influence does p_{win} have on our decision to raise.
- **aggression a:** a probability that we add to p_{win} to enhance our perceived probability of winning, which in turn will favor raising actions.
- **deceptiveness d:** a probability of bluffing in case $p_{win} + a < s$.
- **betting mean μ, standard deviation σ:** two parameters defining our betting distribution, which is a Gaussian distribution in the interval $[0; 1]$, that when sampled, based on our current remaining stack, determines the amount we wish to bet in terms of a percentage of our remaining funds.

Additionally, individuals need a fitness score that determines how well the individual is performing compared to its peers. We model this as the accumulated bankroll of an individual after every tournament it has participated in. Restricting the domain of all of the parameters that determine the behavior of an individual to the range $[0; 1]$ makes possible to combine parameters from two individuals to generate a child, it also makes mutations simple, as any parameter can be easily be altered in any direction (within the range) by a simple computation.

When playing a tournament, an individual is queried for an action at specific points in the game, each of which are determined by the same function `GetAction`. Algorithm 1 displays the pseudo-code for this method, and an auxiliary method, called `Bet`, can be seen in Algorithm 2, which together determine which action is chosen, and if a betting amount is needed (RaiseAction) it will determine this as well.

The intuition for `GetAction` is that if the player can raise, and believes that their hand is strong enough, then the player will choose that before considering

calling, checking or folding. It is essentially our interpretation of how the principles play together and in which precedence we deem them most valuable. It is important to note that many of the operations performed here could be different, as it is only our current best algorithm for providing an interface that encompasses many possible strategies, it is not by any means *the* optimal way of deciding on an action.

Algorithm 1 GetAction

Input: round state S, tournament stack size T, player model P
if Round state is pre-flop or flop (2-5 visible cards) **then**
  $p_{win} \leftarrow lookup(S_{cards_{player}} + S_{cards_{board}})$
else
  Calculate probability of winning, using Monte Carlo simulation over 1000 iterations. $p_{win} \leftarrow MonteCarlo(S, 1000)$
end if
Determine player and opponent current contributions to pot.

$$C_p \leftarrow T - S_{stacks_{player}}$$
$$C_o \leftarrow T - S_{stacks_{opponent}}$$

Determine total pot. $T_p \leftarrow C_p + C_o$
Determine cost of continuing. $C_c \leftarrow S_{raiseBounds}$
Calculate expected winnings. $E_w \leftarrow (p_{win} * T_p) - ((1 - p_{win}) \cdot C_c)$
determine legal actions. $A \leftarrow S_{legalActions}$
if RaiseAction $\in A$ **then**
  if $p_{win} + P_a > P_s$ **then**
    return Bet(p_{win}, S, T, P)
  else if $P_d \geq n \sim \mathcal{N}$ where $\mathcal{N}$ is the uniform distribution $[0; 1]$ **then**
    Determine probability of winning parameter for bet function, to act like player is raising without bluffing. $p_{bluff} \leftarrow (p_{win} + P_d)/2$
    return Bet(p_{bluff}, S, T, P)
  end if
end if
if CallAction $\in A$ **then**
  return CallAction()
end if
if CheckAction $\in A$ **then**
  return CheckAction()
end if
return FoldAction()

Algorithm 2 Bet

1: Input: probability of winning p_{win}, round state S, tournament stack size T, player model P.
2: Determine raise lower and upper bounds.

$$R_u \leftarrow S_{raiseBounds_U}$$
$$R_l \leftarrow S_{raiseBounds_L}$$

3: Determine an initial betting amount based on remaining stack and betting distribution.

$$Amount \leftarrow S_{stacks_{player}} \cdot s \quad \text{where} \quad s \sim \mathcal{N}(P_\mu, P_\sigma)$$

4: Include influence of p_{win} to $Amount$.

$$Amount \leftarrow \lfloor Amount \cdot p_{win} \rfloor$$

5: Bet at least R_l and at most R_u.

$$Amount \leftarrow \max\{R_l, \min\{R_u, Amount\}\}$$

6: **return** RaiseAction($Amount$)

3.3.2 Genetic Algorithm

A genetic algorithm is responsible for generating an initial population, with randomly assigned parameters, after which the main loop of the algorithm starts. Each individual plays in a number of tournaments against randomly assigned opponents, then the n worst individuals are removed and replaced by children and mutations of the best individual. The ratio between children and mutations of the best individual is determined by a parameter. With the new population, a new round of tournaments is performed. This continues for a number of generations, again based on parameters. The idea is that the fittest individuals will influence the new generation of the population, which should result in the population where individuals evolve better playing strategies, against its peers, over time.

As the fitness is based on bankroll, and the bankroll is reset after each generation to ensure each individual gets a fair shot at becoming the best, we have no way of determining if the population actually improves over time. To decide if the population learns, one could try to play against it, though this is not a viable solution in the long run. Ideally, we want an oracle that can tell which individual is better given any pair of individuals, and use that to verify that the population is improving. Another option is using a third party Poker agent as a special opponent to evaluate the fitness of individuals after each iteration, although this approach will be biased, since agents will solely be evaluated on its eminence over a specific strategy and therefore only prove to be

useful against any chosen strategy and should the opponent agent be a human player, it is doubtful that they would participate in millions of games to evolve a strategy which beats theirs. Therefore, we have chosen to make the accumulated bankrolls from repeated tournaments with other individuals in the population of the individuals, the defining fitness criterion of the population.

3.3.3 Monte Carlo Simulations

To estimate the probability of winning used by individuals to make decisions on which action to make in a given state, p_{win}, we adopt our initial strategy of simulating a large number of game states using Monte Carlo simulations. This simplifies our consideration of belief states substantially, since we simulate all of our unperceived details in the universe and therefore estimate what our probability of winning will be. In other words, this approach lets us estimate the probability without explicitly calculating it by considering all belief states and chance nodes of the adversarial search tree.

4 Results

There are many ways the results, i.e. populations generated by our genetic algorithm can be interpreted. As each generation is only tested against itself, the best one in a later generation might not beat the best one from an earlier one. This is a general issue with our algorithm that should be considered. When pitting the best model from each generation against the best model from each other generation, we find that the latest generation beats all others, though the second best is from the first (0th) generation. The results from this showdown of elites can be found in figure 2.

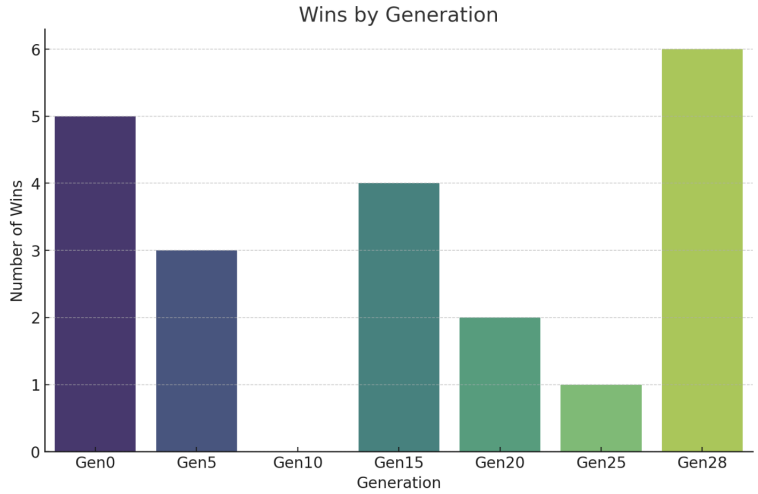


Figure 2: The number of wins by each generations best individual in the showdown.

We expect that running the algorithms using more tournaments in each generation would greatly increase the probability of finding *the* correct ordering

of individuals from each generation, and therefore improve on future generations, as children and mutations are based on this ordering.

With our current algorithm, we have managed to complete 28 generations, using a population size of 50, with 150 tournaments at each generation (split among the individuals). Running the 28 generations took approximately 2 days, which is the reason we have not managed to run the algorithms with a 'better' configuration.

Possible options for improving the run time include concurrently running the tournaments, as no modification except for bankroll is made to the individuals at this step in the algorithm. The bankroll could be collected from each tournament and applied to the affected individuals after finishing the tournaments.

4.1 Playing against other strategies

As a way of estimating the strength of our evolved strategies, we have pinned the 10 best individuals from the final generation against two types of opponents: a completely random opponent and an opponent whom seeks pairs and goes all in when one is present. The results can be seen in figures 3 and 4. As can be seen by these figures, the top 10 strategies beat these two kinds of opponents, even the ones whom did not perform the best in the population. Meanwhile, the top of the population beat both of these simple agents by quite a significant margin.

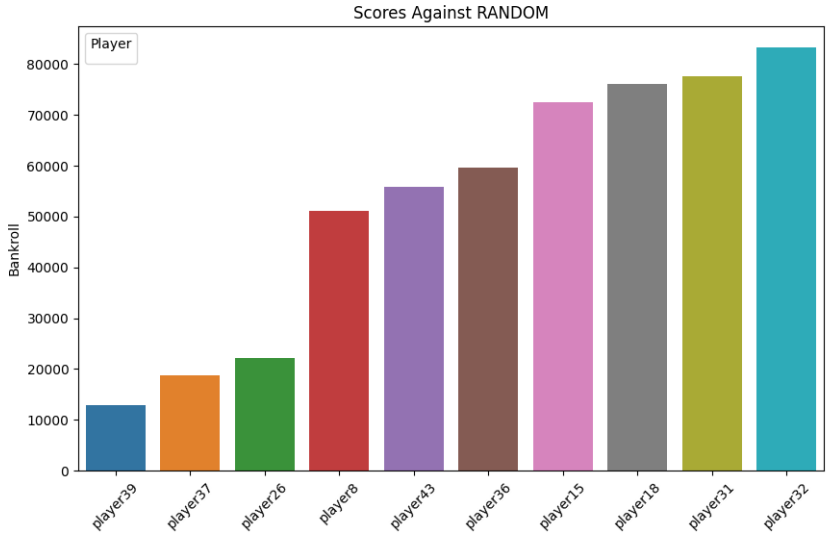


Figure 3: Results of the 10 best individuals against a random opponent in 1000 tournaments.

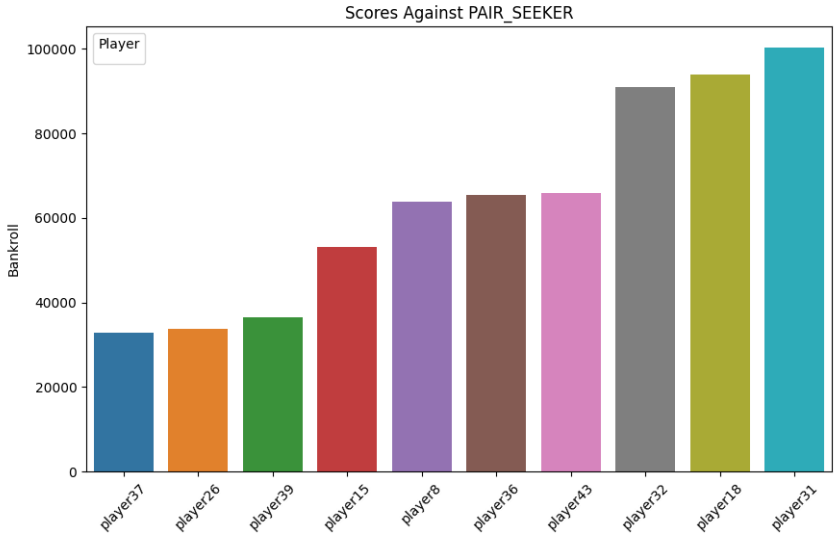
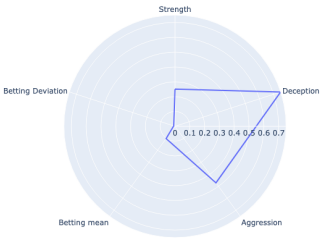
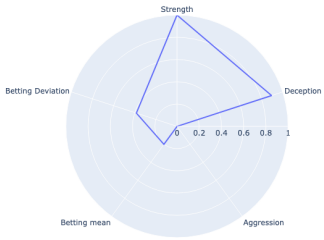


Figure 4: Results of the 10 best individuals against a pair seeking opponent in 1000 tournaments.

Consider the attributes of the individuals scoring highest against these opponents in figure 5. As can be seen, they are very deceptive. It can also be seen that one has chosen to evolve higher levels of strength and the other has chosen to become more aggressive. These attributes are anti-symmetric in a sense that higher aggression implies playing weaker hands whilst higher strength implies placing most of its strategy in playing stronger hands, when not playing deceptively. Although this is the case, it can also be seen that both winners are highly cautious in the amounts which they bet, supposedly aiming at 'safer' incremental improvements to the bankroll.



(a) Individual32.



(b) Individual31.

Figure 5: The distribution of values for the attributes among the winning individuals.

5 Discussion

This section is reserved for discussing some difficulties we have tried to mitigate. One approach is to pre-compute hand strengths such that we can speed up the evolutionary algorithm. Another issue is how to ensure that our evolutionary algorithm actually improves, as there is no definitive winner. All agents play against opponents fairly similar to itself, which introduces a bias we cannot foresee.

5.1 Pattern databases

In the development of our Poker agent, we have considered various optimisation methods. One promising approach is pre-computation of Monte Carlo simulations, since these simulations will be carried out at every step where the evolutionary algorithm asks an agent for an action. This involves running simulations beforehand and storing the results in some lookup database. During execution of the game, the agent can then quickly retrieve the estimated hand strength from this database, effectively solving our runtime computation problem. However, it is not enough to save details of the game state as if some cards are on or off suit and test any pair cards due to the complexity of poker. Pre-computation and caching the strength of on or off suit and any pair cards can provide a good starting point, but it will not capture all the complexities of the game. For a more accurate estimation of hand strength, one would need to consider the specific game state, including the number of players and their playing styles. So, not only do we need to store the winning probability for any pair of cards, but we also need to encode how the strength of a hand can change dramatically as cards will be dealt in the future or what cards other players are holding. The naive version of the computation, will have to consider every set of 2,5,6 and 7 visible cards to make sure we do not discriminate any combinations which drastically alter the probability of winning. Thus, the resulting database, containing these patterns, will be of considerable size with

$$\binom{52}{2} + \binom{52}{5} + \binom{52}{6} + \binom{52}{7} \approx 1.5 \cdot 10^8$$

entries in it.

This was found to be impractical, hence only precomputations of 2 and 5 cards have been done, and are used. This lookup results in an over 50% reduction in run time, as a lot of rounds do not go past the flop, due to some party folding. Only using $\binom{52}{2} + \binom{52}{5} \approx 2.6 \cdot 10^6$ entries, brings the memory needed to store the entries down to a much more manageable size, ensuring fast lookup times.

5.2 The Definitive Winner

Every poker player will eventually be deceived by another player. This includes the agent which plays the strategy, that we have evolved through evolutionary work. It is not a trivial task to consider the usefulness of any one strategy in Poker, since there are so many unknown variables at play. The evolution of our agents strategy is somewhat biased from the fact that it will be very good at playing against other strategies in the population, which may be slightly similar to itself. Thus, when taking any one strategy out of this simulation, we may

find that it proves to perform terribly worse in any other environments that we can place it in. This is somewhat an issue in which larger population sizes, different amounts of entropy and more generations might improve upon. These potential benefits comes with a rather large cost in terms of computation time with, *possibly*, diminishing returns. The strategy of choosing an action based on the four principles of Poker, which we proposed in Algorithm 1 might also be flawed in its design and thereby be discriminating of certain strategies, which could prove to be superior.

To obtain better and more definitive results, one might consider letting every individual in a population battle every other individual in the same population, to ensure the best one is found. For our tests we had to restrict the number of tournaments significantly, as the processing time rises exponentially with the number of tournaments.

6 Conclusion

In this project we aimed to figure out a way of creating an intelligent Poker agent. Our strategy for doing so greatly depended on our existing and acquired domain knowledge, specifically the four principles of Poker. With the principles in mind, we decided on creating a genetic algorithm based on a scheme we believe is capable of encapsulating a wide range of strategies, and then let the population improve generation after generation in hopes of ending up with the ultimate Poker agent. Clearly we aimed a bit high, and did not reach our goals in terms of finding a truly great Poker agent, though we do believe our strategy, with certain improvements and tweaks, is sound given enough resources.

In short, generating poker agents using genetic algorithms has many aspects, requires careful attention to detail and requires enormous amounts of computing resources.