Five-O Poker

Shai Aharon, Moshe Mandel

***Abstract*—The Five-O Poker game is a variant of poker. In this project we tried building an agent that will play the game well. We have built a state space, different evaluation functions and applied different methods for building various agents. The results show that the Expectiminimax agent plays better than all the other agents, including the average human player.**

I. Introduction

A. The Rules of Five-O Poker

The game is played between two players, with a standard fifty two card deck. As the game progresses, each player gradually builds five poker hands that compete against the opponent's five hands.[1] That is, each hand competes against the hand directly across it. Initially each player is dealt five cards in a row facing up, where each column represents a hand. At each round, each player draws the next card from the deck and places the card facing up in one of the columns. After adding a card to a hand, the player must add a card to all his other hands before he can place another card in the same hand (i.e. all hands must be of the same size, or missing at most one card to reach that size, at all times). At the last stage of the game, the last card of every hand is placed facing down; its identity hidden from the opponent.

B. Mathematical Background

A few mathematical equations helped us derive useful tools for our program.

The chance to get a flush in a particular hand:

The chance to get a flush is calculated by the hyper-geometric probability function.

Let:

- N : number of currently unseen cards.
- K : number of relevant cards for this hand (in this case, the number of cards of the same suit of this hand) that are currently still in the deck.
- n : number of draws that will be available for this player to place his drawn cards in this hand.
- k : number of relevant cards needed for this hand (i.e.: number of cards needed to complete hand to a 5 cards hand)

Then, the chance to get k relevant cards out of n draws is:

$$P\left(X=k\right)=\frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}} \tag{1}$$

[1]We assume the reader is familiar with the standard poker hand ranking. Those who are not may refer to: http://en.wikipedia.org/wiki/List_of_poker_hands.

The chance to get a straight in a particular hand:

The probability for a straight is calculated by the multivariate hyper-geometric distribution function, an extension of the hyper-geometric function, which involves not only a single type of relevant card, but various types of cards that are relevant for this hand.

Let:

- N : number of currently unseen cards.
- $L : 0 \leq L \leq 4$, number of cards needed to complete this hand.
- n : number of draws that will be available for this player to place his drawn cards in this hand.

For a particular straight to aim to (e.g. from the hand '$6, 7, 8$', aiming to get another '$9, 10$'), we will denote:

- $K_1, \ldots, K_L$: for each needed card of rank $k_1, \ldots, k_L$, K_i denotes the number of relevant cards of rank k_i that are currently still in the deck.

Then, the probability to get a specific straight is the multivariate hyper-geometric distribution function, where exactly one card is needed from each rank:

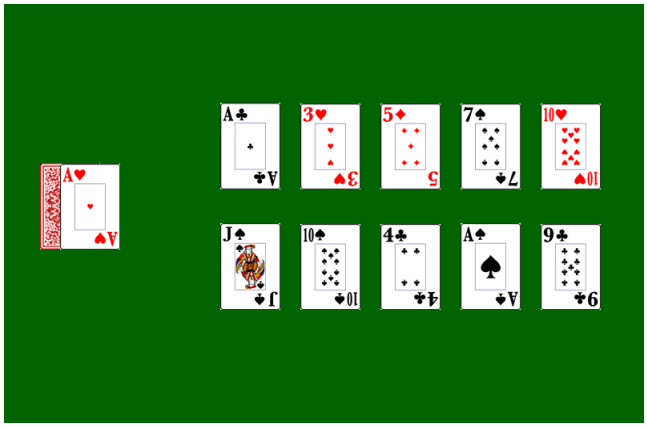
$$\begin{aligned} P\left(k_1=1,\ldots,k_L=1\right) &= \frac{\binom{K_1}{k_1}\cdots\binom{K_L}{k_L}\binom{N-(K_1+\cdots+K_L)}{n-(k_1+\cdots+k_L)}}{\binom{N}{n}} \\ &= \frac{\binom{K_1}{1}\cdots\binom{K_L}{1}\binom{N-(K_1+\cdots+K_L)}{n-L}}{\binom{N}{n}} \\ &= \frac{K_1\cdots K_L\cdot\binom{N-(K_1+\cdots+K_L)}{n-L}}{\binom{N}{n}} \end{aligned}$$

Since for every hand of one to four cards there can be a number of possible five-card straights that can be achieved for the current hand, the exact probability takes into account every set $A' = \{K'_1, \ldots, K'_L\}$ that is relevant for achieving a possible straight. Every different straight (of a different set of ranks) is a disjoint event. Denoting $S = \{A_1, A_2, \ldots\}$ as the set of possible straights that can be achieved from this hand, the exact probability of achieving a straight is:

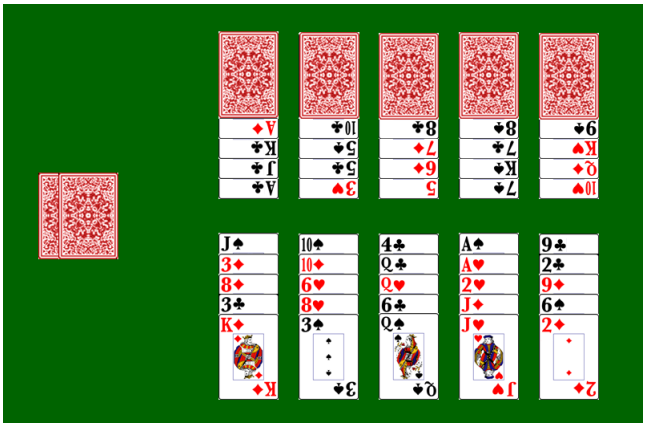
$$P\left(A\right)=\sum_{A_i\in S}P\left(A_i\right) \tag{2}$$

The chance of winning a game

Given the player's five hands, if we were to calculate each of the hands probability to beat its competing hand, how would we calculate the total probability of winning a game? First, we should observe that only in a case where random actions are taken by both players, the event that a player's hand will beat its opposite hand is independent of the events that the player's other hands will beat their opposing hands. However, since we are assuming both players are rational, there does



(a) Initial state of game



(b) Terminal state of game, before cards are revealed

exist a certain degree of relation between the event of one hand winning to the event of another winning. Assuming that the relation between the events (of each hand winning) has a a small impact on the actual probability to win the game, we will calculate the odds of winning as if the events are independent.

Assigning A_i, $i \in \{1,\ldots,5\}$ as the event that a player's hand i will win, and A as the event that the player will win the game (i.e. at least three of its hands have beaten their competitions), we will calculate $P(A)$:
Denoting A_3^5, A_4^5, A_5^5 as the events that the player will win exactly $3,4,5$ hands out of five respectively, we observe that these events are disjoint and therefore independent of each other. We will denote $X_3 = \{x_3 \in A_3^5\}$, $X_4 = \{x_4 \in A_4^5\}$, and $X_5 = \{x_5 \in A_5^5\}$ as the groups of the events describes above. Trivially, $|X_3| = \binom{5}{3}$, $|X_4| = \binom{5}{4}$, and $|X_5| = \binom{5}{5}$.
For any two events $x_3, x_3' \in A_3^5$, x_3 and x_3' are disjoint. For any $x_3 \in A_3^5$ we will denote $A_{i_1}, A_{i_2}, A_{i_3} \in \{A_1,\ldots,A_5\}$ as the three winning hands and $\overline{A_{j_1}}, \overline{A_{j_2}} \in \{A_1,\ldots,A_5\} \setminus \{A_{i_1}, A_{i_2}, A_{i_3}\}$ as the losing hands. We will calculate $P\left(A_3^5\right)$:

$$P\left(A_3^5\right) = \sum_{x_3 \in A_3^5} P(A_{i_1}) \cdot P(A_{i_2}) \cdot P(A_{i_3}) \cdot P(\overline{A_{j_1}}) \cdot P(\overline{A_{j_2}})$$
$$= \sum_{x_3 \in A_3^5} P(A_{i_1}) \cdot P(A_{i_2}) \cdot P(A_{i_3}) \cdot (1 - P(A_{j_1})) \cdot (1 - P(A_{j_2}))$$

Similarly:

$$P\left(A_4^5\right) = \sum_{x_4 \in A_4^5} P(A_{i_1}) \cdot P(A_{i_2}) \cdot P(A_{i_3}) \cdot P(A_{i_4}) (1 - P(A_{j_1}))$$

$$P\left(A_5^5\right) = P(A_1) \cdot P(A_2) \cdot P(A_3) \cdot P(A_4) \cdot P(A_5)$$

So summing the disjoint events we get:

$$P(A) = P\left(A_3^5\right) + P\left(A_4^5\right) + P\left(A_5^5\right) \tag{3}$$

II. Approaches and Methods

The game is a sequential zero-sum game, with a chance factor involved. Therefore, we have built a state space that describes the possible states of the game. From the state space, we have constructed a game tree on which we can traverse our various methods in order to achieve optimal play.

Let S be the state space, every $s \in S$ holds the current ply number ($1 \leq p \leq 40$, total number of moves done by both players: 20 moves are done by each player in every game), the current deck, and 2 agent states - one for each player. An agent state consists of the agent's current 5 hands, his own current ply number ($1 \leq p_i \leq 20$) and the "unseen cards" - cards which are hidden from this agent (until the last stage of the game, both players unseen cards consist only of the cards currently left in the deck. At the last stage, cards which are drawn by the opponent remain hidden from the agent and therefore are not removed from the agent's unseen cards).

At every state, A is the set of legal actions available for the player whose turn is currently. $a \in A$ is a hand chosen by the player to place his next card. $a \in A$ is legal if the player can currently place a card in that hand according to the game rules. Applying an action $a \in A$ on a state $s \in S$ adds the next card to the chosen hand, reveals the card to the opponent (i.e. removes it from the unseen cards list) if the game is still at its first stage, and advances the player's own ply number and the game's total ply number by one.

$s_t \in S$ is a terminal state if the total ply number equals 40.

From the initial state $s_i \in S$, our game tree consists of all the states that can be reached by s_i, where each level $1 \leq n \leq 40$ of the tree includes the states that can be reached by agent $i = n\%2$. Since the game tree is intractable to search through all the way down to the leaves,[2] the ordinary minimax algorithm is impractical to implement. We needed to find other ways to choose the optimal action at each state. We chose different methods for designing the agent, each method with its own advantages and drawbacks.

The last stage of the game is of imperfect information; the cards placed by the opponent are hidden from the player. At this stage, the agent chooses an action based on probability

[2]At the initial stage $s_i \in S$, each player is dealt 5 cards, and throughout the game another 40 cards are dealt in total. So out of 42 unique cards left, the size of the state space is the number of permutations of 40 cards out of 42. Specifically

$$\frac{42!}{2!} \approx 7.02 \cdot 10^{50}$$

Table I: $7,462$ equivalent poker ranks

Hand Value	Unique	Distinct
Straight Flush	40	10
Four of a Kind	624	156
Full Houses	3,744	156
Flush	5,108	1,277
Straight	10,200	10
Three of a Kind	54,912	858
Two Pair	123,552	858
One Pair	10,982,240	2,860
High Card	1,302,540	1,277
Total	2,598,960	7,462

values it can calculate based on the information which is accessible to him.

A. Evaluating Hands and States

At the first stage of the game, the game is of perfect information, i.e. all cards put by the players are revealed. Correctly evaluating how a current state affects the outcome in the terminal state is an essential part of optimal play. We evaluated a state by evaluating each hand separately, and combining the different hand values into one. In some methods, we chose to focus only on the player's own hands, or focus solely on one hand at a time; while in others, we evaluated the state as a whole, consisting of five pairs of hands, each competing against one another. When designing the different evaluation functions, we had to bear in mind the methods in which they will be applied. So we tried to balance between accuracy – for precise evaluations, and speed – for allowing to evaluate millions of states in reasonable time.

1) Hand Evaluation: A standard deck of cards consists of $\binom{52}{5}$ unique poker hands, but many hands actually share the same poker hand values (e.g. $AQ845$ flush in spades is the same as a $AQ845$ flush in diamonds). In fact, there are exactly $7,462$ different equivalent classes of poker ranks.

We have used a running-time efficient library[1] written in C to evaluate a complete five-card poker hand. Given a five-card poker hand, our function returns a value between 1 and $7,462$, where 1 is the lowest value and $7,462$ is the highest. This function is used when an agent has reached a state which holds a five-card hand, and at the terminal state to check which player won. But when trying to evaluate an incomplete hand (of $1,\dots 4$ cards), we needed to find ways to predict the hand's value based on its current cards.

a) Precise hand evaluation: This function is defined as '`eval`', documented in '`greedyEval2.py`'.

The function is composed of two attributes of the hand: the hand's current value and the hand's possible future value. The hand's current value consists of the minimum possible rank (out of $7,462$ complete hand ranks) out of all possible complete hands that can be reached by the current cards in hand. This value is efficiently available by a look-up table constructed beforehand. The look-up table has every possible incomplete hand as keys, where each key is a tuple of the incomplete hand's cards ranks (e.g. $A, J, 9$ would be $(12, 9, 7)$[3]). This look-up table is calculated at the beginning of each program run, and therefore achieving the current value is time efficient. The possible future value itself consists of two parts, the probability of reaching a flush and the probability of reaching a straight. The probabilities are calculated according to equation (1) and equation (2). These calculations are calculated based on the hand's current cards and the relevant cards remaining in the agent's unseen cards list.

Let $size$ be the hand's current number of cards, $currVal$ be the hand's current value, $flushVal$ the minimum possible flush value reachable by this hand (based on its current highest card), p the probability calculated for reaching a flush, $straightVal$ the minimum possible straight value reachable by this hand and q the probability calculated for reaching a straight. So the value returned by this function is:

$$(1-p-q)\cdot currVal + (p\cdot flushVal + q\cdot straightVal)\cdot\frac{size}{5}$$

This function seems to generally predict (to some extent) the final outcome of the hand.

b) Imprecise but faster hand evaluation: There are two imprecise, but fast, hand evaluation functions. The first is '`fastEval`' (documented in '`greedyEval2.py`'), and the latter is '`getPackExpect`' (documented in '`greedyEval1.py`').

'`fastEval`' is similar to the precise function described above, but instead of calculating precise probabilities for straights and flushes, it roughly calculates them by the percentage of relevant cards (cards of same suit or of rank values adjacent to hand's cards rank values for flush and straight probabilities respectively) that remain in agent's unseen cards list.

'`getPackExpect`' returns a value between 1 and 17. It sums different values to the total hand value according to various attributes. This function doesn't try to predict the actual outcome of this hand, but rather roughly adds positive values based on the cards rank (high card), common ranks (two-of-a-kind, three-of-a-kind,...), common suit (flush), and the cards rank range (straight). The attributes were weighted in such a way that it will consider instant rewards, such as pairs, over possible future rewards. But it still encourages future investments, such as straight or flush hands. Although this function is less accurate, it does effectively compare between a pair of hands and estimates which hand is better. More importantly, it evaluates a hand fast – faster than '`fastEval`' – allowing for more states to be explored in the belief state. So what this function lacks in accuracy, it compensates in speed and quantity.

2) State Evaluation: When evaluating a state in the game, we had in mind a few observations that led us to design the evaluation functions as we did:

- A state evaluation function must be consistent with the agent's preferences. For some hand evaluation function $u : S \to [-1, 1]$, let's assign for some hand in the state

[3]Card ranks are set as integers from $(0,\dots,12)$, where 0 is a deuce and 12 is an Ace.

a, competing against the opponents hand b:

$$u(a) = \begin{cases} -1 & a \text{ surely losses to } b \\ 0 & a \text{ equals } b \\ 1 & a \text{ surely beats } b \end{cases}$$

u is a continuous function, that ranges between $-1, 0$ and 1 accordingly. In which the more strongly a implies that it will eventually win/lose, $|u(a)|$ is bigger (and positive/negative respectively).
Let us assume that $s_1 \in S$ is a given state in which u values the state's hands as $(1, 1, 1, -\varepsilon, -\varepsilon)$, and another state $s_2 \in S$ in which u values that state's hands as $(1, 1, -\varepsilon, -\varepsilon, -\varepsilon)$. In other words, s_1 "barely" wins and s_2 "just" loses. So for our state evaluation function $U{:}[-1, 1] \to \mathbb{R}$ to be consistent with the agent's preferences, we would like U to assign values to s_1 and s_2 in such a way where $U(s_1) > U(s_2)$ is always true.

- A low-ranking incomplete hand can drastically improve its ranking just by a single card added to it, whereas a high ranking incomplete hand will likely maintain its ranking till the end of the game. This implies that when comparing two hands, it isn't enough just to calculate the linear difference between them, but rather find other nonlinear functions which return values that portray the actual difference between the hands in a more precise manner.
- There exists a correlation between estimated value difference between a pair of cards, and the probability of the higher valued hand actually beating the other hand. This implies that a state-evaluation function should assign a higher value to a game won with a higher total score (of all its hand ranks), in respect to a game won with a lower total score.

a) `greedyEval1`: 'greedyEval1' uses the 'getPackExpect' hand-evaluation function to value the hands in a state. Given a state $s \in S$, it iterates over each competing pair of hands a_i, b_i $i \in \{1, \ldots, 5\}$, and evaluates the difference of each pair as:

$$diff_i = \left\lfloor \frac{\texttt{getPackExpect}(PlayerHand) - \texttt{getPackExpect}(OpponentHand)}{17} \right\rfloor$$

After evaluating the difference between each pair, the function estimates the state value by setting the score as:

$$score = \sum_{i=1}^{5} e^{\tanh(9 \cdot diff_i)}$$

b) `greedyEval2`: 'greedyEval2' uses the 'eval' hand-evaluation function to value the hands in a state. For each pair of competing hands a_i, b_i $i \in \{1, \ldots, 5\}$, it evaluates the difference between them as:

$$\begin{aligned} diff_i &= \ln\left(\texttt{fastEval}(PlayerHand)\right) - \ln\left(\texttt{fastEval}(OpponentHand)\right) \\ &= \ln\left(\frac{\texttt{fastEval}(PlayerHand)}{\texttt{fastEval}(OpponentHand)}\right) \end{aligned}$$

The function estimates the state as a whole by setting the state value as:

$$score = \sum_{i=1}^{5} diff_i$$

c) `fastGreedyEval2`: 'fastGreedyEval2' is similar to 'greedyEval2', but for faster performance it uses the 'fastEval' hand-evaluation function to value the hands in a state. It calculates the ranking difference between a pair of opposing hands and the state value the same way as 'greedyEval2':

$$\begin{aligned} diff_i &= \ln\left(\texttt{fastEval}(PlayerHand)\right) - \ln\left(\texttt{fastEval}(OpponentHand)\right) \\ &= \ln\left(\frac{\texttt{fastEval}(PlayerHand)}{\texttt{fastEval}(OpponentHand)}\right) \end{aligned}$$

and:

$$score = \sum_{i=1}^{5} diff_i$$

B. Agents

We have designed different agents for the game. Some agents evaluate the state as a whole, while others solely focus on a partial part of the state (such as only the player's own hands). Most of the agents rely on the evaluation functions to evaluate a state or hand, but some have their own mechanism to calculate a state or hand value. Some agents consume more running time than others, a tradeoff for precise evaluation and optimal play.

1) Reflex Agents: The reflex agents differ from the other agents in the way they precept the environment. While other agents try to deduce some assumption on a state as a whole by evaluating the different hands and manipulating their values into one concise value, these agents focus only on a partial attribute of the state and try to maximize its value. This usually results in high scores achieved for all hands. But reflex agents lack the ability to "choose their battles": instead of focusing on the hands that are more likely to win and ignoring hands likely to lose, they try to maximize all their hands simultaneously. All reflex agents use the 'eval' function to evaluate hands.

a) `ReflexAgent1`: This agent tries to find the hand that best suits the next card by comparing each of its hands against the opponent's hands, and choosing the hand that, after the next card will be added to it, will have the maximal (positive) linear difference from its competing hand. Denoting A as the set of the agent's legal actions, $hand_a$ as the player's hand $a \in A$ after action a is applied and $oppHand_a$ as the opponent's hand competing against it. The agent will choose:

$$\arg\max_{a \in A} \{\texttt{eval}(hand_a) - \texttt{eval}(oppHand_a)\}$$

b) `ReflexAgent2`: This agent tried to find the hand that best suits the next card by choosing the hand that will have the maximal value after the next card is added to it. Unlike 'ReflexAgent1', it ignores the opponent's hand, even though its cards are visible to him. Denoting A as the set of the agent's legal actions and $hand_a$ - the player's hand $a \in A$ after a is applied. The agent will choose:

$$\arg\max_{a \in A} \{\texttt{eval}(hand_a)\}$$

c) `ReflexAgent3`: This agent tries to find the hand that best suits the next card by choosing the hand that will mostly improve after the next card is added to it. Denoting A as above, $preHand_a$ - the player's hand $a \in a$ before the card is added and $postHand_a$ - the player's hand after the card is added. The agent will choose:

$$\arg\max_{a \in A} \{\texttt{eval}\left(postHand_a\right) - \texttt{eval}\left(postHand_a\right)\}$$

2) Greedy Agent: '`GreedyAgent`' evaluates the state as a whole, and uses any of the state-evaluation functions described above.[4] Out of all the states that can be reached from the current state, the agent chooses the action that will lead to a maximal valued state. Denoting A as the set of legal actions in the current state $s \in S$, $Result\left(s, a\right) \in S$ as the state reached by a and $U : S \to \mathbb{R}$ as the agent's state-evaluation function. The agent will choose:

$$\arg\max_{a \in A} \{U\left(Result\left(s, a\right)\right)\}$$

3) ExpectiMiniMax, averaging over clairvoyance: Since it is impossible to explore the entire game search space with the standard minimax algorithm, the obvious approach would be to cut off the search at a certain depth and use evaluation functions to evaluate the current state. We also use the alpha-beta pruning method to cut off branches that cannot possibly influence the final decision. Since there is a chance factor involved in the game, we use the generalized version of the minimax algorithm: Expectiminimax. This version acts the same as Minimax, except on chance nodes, for which it computes the expected value[2].

When applying Expectiminimax on card games, one may think of all possible permutations of a fifty two card deck, and initially refer to the particular order of a game's deck as one of $52!$ permutations. Although this analogy turns out to be incorrect, it suggests an effective algorithm[3]: consider all possible deals of the invisible cards, solve each one as if it were a fully observable game, and then choose the move that has the best outcome averaged over all the deals. Since each deal of cards will be equally likely, denoting A as the set of legal actions in state s, $p = P\left(s\right)$ as the equally likely chance of deal s occurring and $R\left(s, a\right) \in S$ as the state action $a \in A$ leads to. At each state the agent chooses:

$$\arg\max_{a \in A} \{p \cdot HMINIMAX\left(R\left(s, a\right)\right)\}$$

Since the number of possible deals is too large to iterate over, we resort to a Monte Carlo approximation[3], and instead of adding up all of the deals, we take a random sample of N deals:

$$\arg\max_{a \in A} \left\{\frac{1}{N}\sum_{i=1}^{N} HMINIMAX\left(R\left(s_i, a\right)\right)\right\}$$

In our program we implemented the method slightly differently. Instead of iterating over all the decks for each hand, for each deck we iterate over all five hands and return the hand that has the best outcome for that specific deck. After iterating over all the decks, we choose the hand that has been returned the most times. This is more time-efficient and provides better results.

Using the method described above forces us to provide a state evaluation function that is as time-efficient as possible, since in order for the algorithm to correctly choose the optimal action, it must evaluate millions of states. In order to cope with these quantities, we use alpha-beta pruning to reduce the number of redundant states searched at each level. The deeper we allow the algorithm to search the game tree at each turn, the better the returned results. The major drawback is that this is very time consuming, but even allowing the algorithm to run for merely a minute per turn produces good results.

4) Monte-Carlo Simulation: An alternative way to evaluate a state is by Monte Carlo Simulation. We would like to calculate the probability of each hand beating its opposing hand, and return the approximated chance of winning by calculating the winning probability using equation (3).

Given a set of legal actions - A we calculate the probability value of each hand before any action is applied, assigning to each hand the value - $preActionProb_i (1 \leq i \leq 5)$, and after each legal action is applied we assign $postActionProb_i$ $(1 \leq i \leq |A|)$. For each action $a_i \in A$ we calculate the probability of winning the game using equation (3), where each hand $j \neq i$ is assigned $preActionProb_j$ and hand i is assigned $postActionProb_i$. The agent returns the action that results in a maximal winning probability.

Since the exact probability of a hand to win is not always time and memory efficient (especially at the early stage of the game), a way to approximately calculate the probability of a hand beating its opposing hand is to simulate many possible outcomes and return the percentage of wins (out of the number of simulations simulated). The more simulations simulated, the more accurate the approximation is. But even after only 10,000 simulations, the agent can reach good results. We have used 100,000 simulations to evaluate each of the players' hands, and evaluated the state by calculating the probability of winning using equation (3).

5) Calculating exact probabilities for each hand: At later stages of the game, due to a smaller number of possible outcomes, it is possible to calculate exact winning probabilities of each pair of competing hands. We use this method at the second stage of the game, choosing the action in the same way described in the previous section: the action which maximizes the winning probability.

The exact probability of a hand can be computed by iterating over all of the possible outcomes of the hand and its opposing hand, and calculating the exact percentage of winning outcomes (from all the possible outcomes).

Although calculating probabilities at early stages of the game are very memory-consuming, except for the first round of the game (where at least one player has at least one hand of one card), we have found a way to calculate exact probabilities.[5] This is done by building a huge look-up table for each of the players' hands at once, saving it on the

[4]The agent uses a single evaluation function throughout the game, set at initialization time.

[5]This is due to memory limits, using a slightly different method or using a more advanced computer will allow us to calculate exact probabilities at all stages.

Figure 1: Monte Carlo simulation, win probability

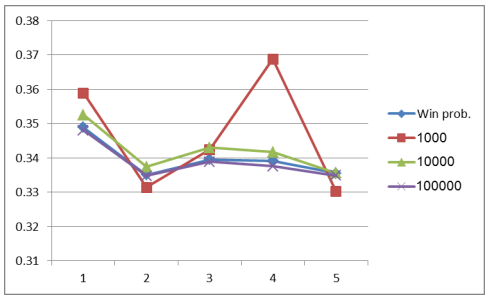
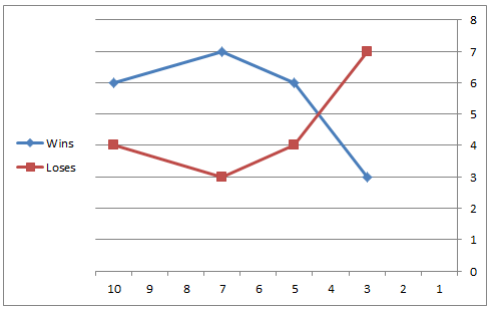


Figure 2: Expectiminimax performance at different depths



computer's static memory and using it for all future outcomes of the hand. When calculating probabilities for the last stage, this is not necessary and the values can be calculated just by using the computer's dynamic memory.

a) Probabilities look-up table: For each pair of competing hands, we build a look-up table as follows:

Let Δ_{x_l} be the sample space of all possible outcomes of some particular hand $l = 1, \ldots, 5$ of the opponent.[6] Let Δ_{y_l} be the sample space of all possible outcomes of a player's hand l. The table will be of size $|\Delta_{x_i}| \times |\Delta_{y_i}|$, where each cell will be filled as follows: $f : \{\Delta_{x_l}\} \times \{\Delta_{y_l}\} \to \{0, 1\}$, for outcomes $\delta_x \in \Delta_{x_l}, \delta_y \in \Delta_{y_l}$

$$f(\delta_x, \delta_y) = \begin{cases} 1 & \delta_y \text{ wins or ties} \\ 0 & \delta_y \text{ loses} \end{cases}$$

For accessing the table we keep a dictionary of 52 keys for each of the competing hands. For each of the 52 cards, we keep a set of the table's indices of rows (player's hand) or of columns (opponent's hand) that contain the card. This will help to choose only the rows/columns that we are interested in at each stage.

Calculating the hand's pre-action winning probability is done as follows:

1) Get all rows containing current cards in hand but not containing the next card, and all columns that contain current card in opposing hand.
2) Let us assign $S \subseteq \{\Delta_{x_l}\} \times \{\Delta_{y_l}\}$ - all cells that belong to rows containing current cards in hand, and also belong to columns containing current cards in opposing hand. Let us assign $T \subseteq S$ - all cells s.t. $\forall t \in T : f(t) = 1$.
3) Return $\frac{|T|}{|S|}$.

Calculating the hand's post-action winning probability is done in a similar way, except the rows containing the next card are also included in the process.

III. Results

As described above, the Monte Carlo simulation agent uses simulations on sample spaces to calculate approximated probabilities. In table II and figure 1, it can clearly be seen how the approximation is more accurate as the sample size grows.

We have tested the Expectiminimax agent to see how well it will perform with respect to how deep it searches the game tree. We simulated the agent against the greedy agent (which used '`greedyEval2`' as an evaluation function), simulating ten games each at different depths. The results can be seen in figure 2. The decrease in the amount of wins when the agent searches ten nodes deep may be due to the time limit set for each turn, which did not allow the agent to simulate through a large number of decks when searching at this depth.

The average scores of all players are shown in figure 3. We have compared the different agents by their scores achieved in descending order (i.e. the highest score of each agent is compared to the respective highest score of each of the other agents, etc.). The scores can be compared to the average score achieved by a human player. We can see how the agents who evaluate the state as a whole (Minimax, Monte Carlo, Greedy) focus on maximizing the values on their three highest hands, while the agents who evaluate each hand separately from the other hands (reflex agents) maximize each of their hands. This results in lower values achieved in their highest hands. The difference between the two state evaluation functions, '`greedyEval1`' and '`greedyEval2`', used by "*Greedy 1*" and "*Greedy 2*" respectively, can easily be seen. "*Greedy 2*" by far outperforms "*Greedy 1*". This can be explained by the way '`greedyEval2`' takes more time for precise calculations, but also by the way it assigns values to current information at hand.[7] '`greedyEval2`' uses a much larger range of values (7,462) to assign to each hand, opposed to the range used by '`greedyEval1`' (17). This allows it to more easily distinguish between similar states, which differ by current subtle differences that can evolve into major differences in the future.

All the minimax agents tested in these games, unless stated otherwise, were allowed to search three plies into the game tree, and the Monte Carlo simulation simulated 100,000 decks at each state evaluation.

We have tested each of the agents against each other, the results may be seen in table III. The overall win-loss ratio can be seen in figure 4. The top three agents who won the most games are *Expectiminimax* – using `fastGreedyEval2` and searching three plies in the game tree, *Monte Carlo simu-*

[6] "possible outcome": a set of cards that will complete the current hand to a complete five-card hand

[7] Pun intended.

Table II: Probabilities of different sample sizes

(a) Put Probability, the probability of a hand winning after placing next card in hand

Put prob.	1000	10000	100000
0.691712	0.716	0.6939	0.68984
0.098279	0.078	0.0974	0.09886
0.399533	0.39	0.3986	0.39877
0.298829	0.328	0.2998	0.29743
0.343097	0.348	0.3452	0.3443

(b) Wait probability, the probability of a hand winning if next card is not placed there

Wait prob.	1000	10000	100000
0.742933	0.755	0.7421	0.74103
0.175254	0.179	0.1755	0.17461
0.465409	0.463	0.4624	0.46429
0.363734	0.335	0.3651	0.36423
0.417712	0.451	0.4264	0.41836

(c) Win Probability, the probability of winning a state

Win prob.	1000	10000	100000
0.34889	0.358913	0.352483	0.348053
0.335084	0.331424	0.337362	0.334738
0.339556	0.342415	0.342972	0.338865
0.339141	0.368798	0.341662	0.337548
0.33549	0.330146	0.335578	0.334906

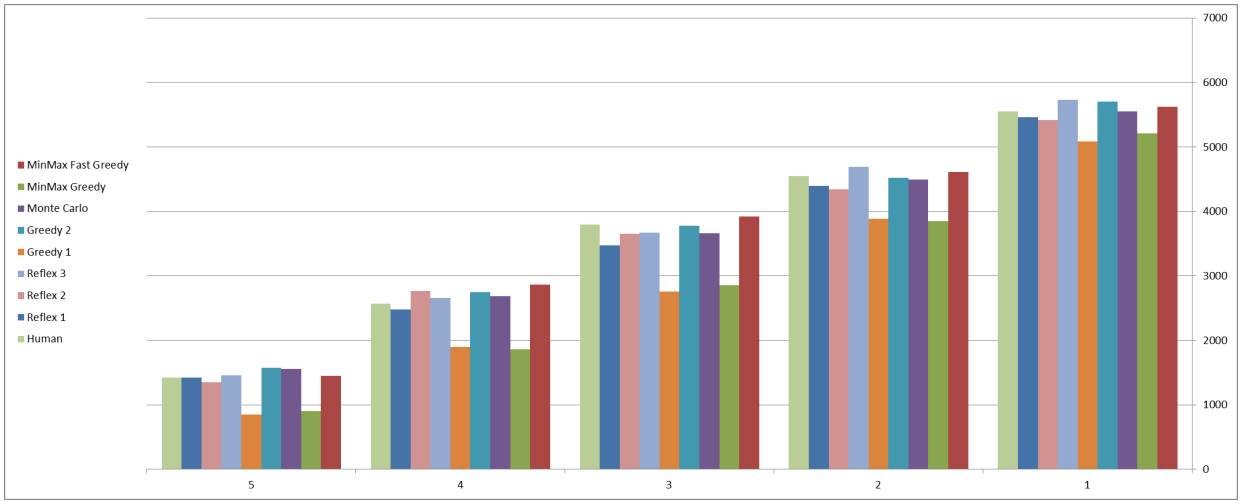


Figure 3: The average scores achieved by the agents, sorted from highest(1) to lowest(5)

lation and `ReflexAgent3`. Surprisingly `ReflexAgent3` performed better than greedy 2, even though both used the same evaluation function. Here also, the difference between the two greedy agents (because of the different state-evaluation function used) can easily be seen.

We have tested our top three agents against human players, and the results can be seen in table IV. As expected, *ExpectiMinimax* performed better than the rest and even beat the human player[8] most of the time (as did the greedy agent using '`greedyEval2`'). It did so by using the '`fastGreedyEval`' state-evaluation function. Even the greedy agent, with a good evaluation function, performed well against the human player.

A. Conclusions

From the results we conclude how in a game in which correct mathematical calculations lead to optimal play, a computer agent can outperform a human player. This is true even for agents who do not necessarily search further into the belief state space, such as the greedy agent. A good evaluation function which focuses on the right attributes and evaluates them correctly into a single concise value, is sufficient in order to excel in the game. The agent which does search the game belief state space does better than all other agents, including a human player. This corroborates with the theorem that states that the minimax algorithm returns the optimal play in a finite zero-sum game[4]. Although the final stage of the Five-O poker game is of imperfect information, the manner in which the agent reaches that final stage and the number of possibilities each of its hands have to beat its competition, greatly influence the game's terminal result. So a major part of obtaining good results in this game is based on optimal play in the first stage, where all the information is revealed. The hidden information in the last stage, although having some influence on the players actions, has a less significant impact. Applying different methods allowed us to see how each method focuses on a different aspect of the

[8]The human player was very proficient at five-O poker, after playing hundreds of games against various computer agents in a short time span.

Table III: Wins and losses between all agents

		reflex 1		reflex 2		reflex 3		greedy 1		greedy 2		MonteCarlo		MinMax Greedy		MinMax fastGreedy	
1	reflex 1	X		3	7	5	5	5	5	5	5	4	6	6	4	4	6
2	reflex 2	7	3	X		7	3	8	2	6	4	3	7	6	4	4	6
3	reflex 3	5	5	3	7	X		7	3	6	4	3	7	8	2	8	2
4	greedy 1	5	5	2	8	3	7	X		1	9	2	8	5	5	3	7
5	greedy 2	5	5	6	4	4	6	9	1	X		5	5	7	3	2	8
6	MonteCarlo	6	4	7	3	7	3	8	2	5	5	X		7	3	2	8
7	MinMax Greedy	4	6	4	6	2	8	5	5	3	7	3	7	X		4	6
8	MinMax fastGreedy	6	4	6	4	2	8	7	3	8	2	8	2	6	4	X	
	Wins	32		39		40		21		36		42		25		43	
	Losses	38		31		30		49		34		28		45		27	

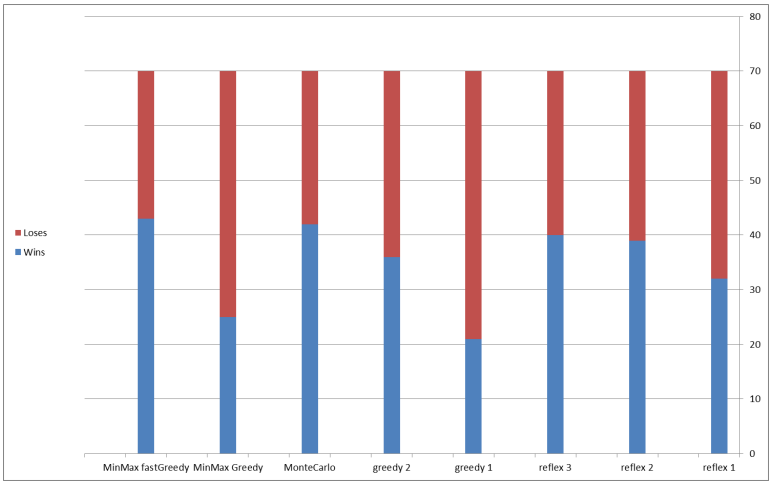


Figure 4: Wins-losses ratios

Table IV: Wins and losses against a human player

	Monte Carlo	Reflex 3	Greedy (greedyEval2)	MinMax (fastGreedy)	Total
Human losses	7	6	8	9	30
Human wins	3	4	2	1	10

game. This results in a non-transitive relation between the agents (e.g. “reflex 3” outperforms “Minmax fastGreedy”; which outperforms “greedy 2”; which in turn outperforms “reflex 3”), which can be explained by the fact that focusing on one attribute of the game may lead to good results against one agent, but to poor results against another.

IV. Summary

In this project we have implemented different methods learned in class, and from other resources (mainly chapter 5 in “Artificial Intelligence A Modern Approach” by Russel and Norvig), in building an agent to the Five-O poker game: a non-deterministic zero-sum card game which comprises perfect and imperfect information sets. We have built different agents, some with surprisingly good results. A main part of our work emphasized on correctly evaluating the game state in a non-deterministic environment. One of the major principles we have discovered was how using non-linear combinations, opposed to linear combinations, in the state evaluation function greatly improves its accuracy. Trying different methods to achieve good results led us to observe the problem from a different angle each time. Overall, the Expectiminimax agent outperformed all other agents. This can be explained by the way in which *Expectiminimax* best predicts future outcome value[9] based on the current information.

This was a great learning experience, although at times it was very challenging, but in retrospect it was as well rewarding. We look forward to implement what we have learned in this project and throughout the course into our future ideas and projects.

References

[1] Cactus Kev’s Poker Hand Evaluator: http://www.suffecool.net/poker/evaluator.html
[2] S. Russel and P. Norvig. Artificial Intelligence A Modern Approach, p. 178, Pearson: Boston *et al.*, 2010.
[3] S. Russel and P. Norvig. Artificial Intelligence A Modern Approach, pp. 183-4, Pearson: Boston *et al.*, 2010.
[4] http://en.wikipedia.org/wiki/Minimax_theorem#Minimax_theorem

[9]Not necessarily predicting future actual outcome (that - the *Monte Carlo Simulation* does), but future outcome value.