Applications of Adversarial Search and Machine Learning in Briscola

Hemachandra Konduru, Oscar von Moeller, Parand Mohri, Joaquin Monedero Simon, Alexander Safi, Alisa Todorova,
Department of Data Science and Knowledge Engineering
Maastricht University
Maastricht, The Netherlands

***Abstract*—The game discussed in this paper is the Italian card game Briscola, which is a turn-based, imperfect information game with partially observable, stochastic, multi-agent, sequential, static, and discrete environment. The goal is to combine an Adversarial Search technique with Machine learning approach to create a strategy the AI agent can follow, such that it can compete against a human player (or another AI agent) in a full game of Briscola. The Adversarial Search technique here is MiniMax with Alpha-Beta pruning with Monte Carlo Approximation and Determinization, while the Machine learning approach is Artificial Neural Network. Input-output pairs are required to train the neural network. They are obtained through the MiniMax with Alpha-Beta pruning and then fed into the neural network. After running experiments on time consumption and percentage of games won, a balanced value for the determinization was found. The major findings and implications of this study include the impractical training times of a Q-Learning based agent and a value for the number of Determinizations that efficiently balances the percentage of games won and time consumption.**

***Index Terms*—Briscola, Card Game, MiniMax, Alpha-Beta Pruning, Monte Carlo Approximation, Adversarial Search, Machine Learning, Artificial Intelligence, AI, Reinforcement Learning, Q-Learning, Artificial Neural Network, Greedy**

I. Introduction

Games have a fixed set of rules, as well as a constrained environment, which makes it easier for Artificial Intelligence (AI) researchers to check how the AI agent is performing and learning in problem-solving challenges than straight in complex real-world problems [9]. The more complex the game is, the harder it is to create an AI that can compete and win against a human. Complexity of a game is defined in terms of the number of actions available at each round, as well as the size of the set of all possible states of the system [7]. For example, real-time strategy (RTS) game AI has large complexity because such games have partially observable, non-deterministic, multi-player, dynamic, continuous environments with simultaneous durative moves (more than one player can take an action at the same time and these actions can take some time to complete) and very small time for decision-making (which makes them "real-time") [7]. On the other hand, more traditional turn-based games (such as chess) have fully observable, deterministic, sequential, multi-player, static, discrete environments [10]. Therefore, the implementation of AI in such games would be more straightforward that in RTS games. Here, the term *straightforward* is used in the sense that a suitable search algorithm could be directly applied without having to add additional functionality.

Some state-of-the-art algorithms for implementing strategy game AI are Counterfactual Regret Minimization (CFR) algorithm, DeepStack and Monte Carlo Tree Search [2]. They are widely used in poker game AI because poker is a great example for imperfect information, competitive, real-time game. The research of such state-of-the-art algorithms could have potential applications in many other fields, such as game theory, marketing, medicine, and cybersecurity [2].

The goal of this article is to combine an Adversarial Search technique with Artificial Neural Network, to create a strategy the AI agent can follow, such that it can compete against a human player (or another AI agent) in a full game of Briscola. Note that the word *compete* here is used in the sense that the AI agent aims to play cards in a manner such that it improves its chances of winning in the long run. The game discussed in this paper is the Italian card game Briscola, which is a turn-based, imperfect information game. It has a partially observable, stochastic, multi-agent, sequential, static, and discrete environment. This will be further explained in section II. Thus, to implement a game strategy AI, we will need a goal-based AI agent. To achieve this, the following two techniques were examined: Q-Learning and Artificial Neural Network (a Machine Learning technique) with MiniMax Alpha-Beta pruning (an Adversarial Search approach). These techniques were chosen because of the given Briscola environment states. The main research question discussed in this article is: How can Machine Learning and an Adversarial Search approach be combined to create a hybrid goal-based AI for the card game Briscola?

This paper is organized as follows: First, section II explains the game Briscola, its rules and its environments. Next, section III describes in detail the implemented algorithms developed for this project, as well as the data sets used for the experiments. In section IV our implementation of the said methods is explained, as well as the complex analysis of these implementations, the confidence bounds and the creation of the Graphical User Interface. Detailed description of the experiments is given in section V. In VI all results from the experiments are presented, which are then interpreted and

explained in the section VII. In the end, in section VIII a summary of this paper is provided.

II. Briscola

Briscola is an Italian card game that uses a special Italian deck of cards, which consists of 40 cards and 4 suits - swords, coins, cups and clubs. The card order with values in the brackets is as follows: Ace (11 points), Three (10 points), King (4 points), Knight (3 points), Jack (2 points), while 7, 6, 5, 4, 2 all have 0 points. The rules for 2-player game are as follows: Shuffle the deck and deal 3 cards to each player. Then, take the next card as the high suit - this will be the suit which will always beat all other suits, card ranking notwithstanding. Player A starts, then Player B plays a card based on these three simple rules:

1) If B and A play the same suit, then the round is won by whoever played the higher card.
2) If B and A play different suits, but neither card is a high suit, A wins the round.
3) If B and A play different suits, and one of the cards is a high suit, then the player with the high suit wins.

The game continues until all the cards have been played. The winner is the player with the highest score.

As mentioned in the introduction, Briscola is:

- A **turned-based** game because after every action a player performs, no moves are allowed until the other player plays.
- An **imperfect information** game due to the fact that it is partially observable - only the hand of 3 cards and the high suit are visible to each player.

Thus, the environments of Briscola are the following:

- **Stochastic** as the outcome involves some randomness and has some uncertainty, given that at the deck of cards is shuffled at the start of every game. Moreover, at each round the cards assigned to the player are random.
- **Multi-agent** since Briscola is a 2-player game.
- **Sequential** given that the player can save a good card (eg: highsuit) for the upcoming rounds.
- **Static** as the environment cannot change while a player is thinking.
- **Discrete** as after 20 rounds, the end of the game is ensured.

Given these environments of Briscola, the algorithms in section III were chosen for implementation.

III. Method

This section discusses the algorithms MiniMax and Mini-Max with Alpha-Beta pruning. It also explains the concepts of Reinforcement Learning (Q-Learning) and Artificial Neural Network. Further, a description of the data sets used for the experiments is provided.

A. Algorithms

1) MiniMax: MiniMax algorithm is a type of Adversarial Search decision rule [10]. It assumes that both players play the best possible card for them each round. In other words, Max player will always choose the card that will maximize its win score, while Min player will always choose the card that will minimize its loss. In fact, MiniMax is a way of tackling the limits the player is set to face, due to having a sequential environment. This algorithm has the following six elements [10]:

1) S_0: The initial state, which specifies how the game is set up at the start.
2) Player(s): Defines which player has the move in a state.
3) Actions(s): Returns the set of legal moves in a state.
4) Result(s,a): The transition model, which defines the result of a move.
5) Terminal-Test(s): A terminal test, which is true when the game is over and false otherwise. States where the game has ended are called terminal states.
6) Utility(s, p): A utility function, which defines the final numeric value for a game that ends in terminal state *s* for a player *p*.

"The initial state, ACTIONS function, and RESULT function define the game tree for the game - a tree where the nodes are game states and the edges are moves" [10].

2) MiniMax with Alpha-Beta pruning: Alpha-Beta pruning aims to decrease the number of nodes that the MiniMax algorithm would usually evaluate by ignoring card choices of the tree that would be worse compared to the previous card choice. We used the following pseudocode, which is from the book "AI a modern approach" [10], for implementing the alpha-beta pruning:

Algorithm 1 The alpha–beta search algorithm for MiniMax with Alpha-Beta pruning

```
function ALPHA-BETA-SEARCH(state) return an action
    v ← Max-Value(state,-∞ ,∞) return the action in
Actions(state) with value v
end function
```

```
function MAX-VALUE(state, α, β) return a utility value
    if Terminal-Test(state) then return Utility(state)
    end if
    v ← -∞
    for a : ACTIONS(STATE) do
        v ← Max(v, Min-Value(Result(s,a), α,β))
        if v ≥ β then return v
        end if
        α ← Max(α, v)
    end for
     return v
end function
```

```
function MIN-VALUE(state, α, β) return a utility value
    if Terminal-Test(state) then return Utility(state)
    end if
    v ← ∞
    for a : ACTIONS(STATE) do
        v ← Min( v, Max-Value(Result(s,a),α,β))
        if v ≤ α then return v
        end if
        β ← Max(β, v,)
    end for
  return v
end function
```

As mentioned above, because Briscola has a partially observable environment, the MiniMax with Alpha-Beta pruning algorithm needs Monte Carlo Approximation with Determinization to be able to make good decisions.

Since card games introduce the element of chance due to randomness (for example, dealing of the cards to each player), we can add the key word *stochastic* to Briscola's environments. A way to choose the best move is to consider all possible card deals as if it is a fully observable game, then average all the deals. However, number of card deals could be very large. That's why a Monte Carlo approximation (MCA) is used: Instead of adding up all the card deals, a random sample of *N* deals is considered, where the probability of deal *s* appearing in the sample is proportional to *P(s)* [10]:

$$\arg\max_{a} \frac{1}{N} \sum_{i=1}^{N} \text{MiniMax(Result}(s_i,\ a)) \tag{1}$$

Determinization, also known as perfect information Monte Carlo (PIMC), turns an imperfect game to a perfect information instance of a game in order to obtain more accurate results each round [3]. In terms of card games, that would mean that the agent can see its opponent's cards, the high suit, and the shuffled deck. Because we don't want the agent to "cheat", i.e to know this information, we generate random hands as human hands. Then, several determinizations are created from the current game state, and later combined in order to return a decision for the original imperfect game [3].

B. Q-Learning

Reinforcement Learning is a Machine Learning technique aiming to create an agent able to make a sequence of decisions [14]. The agent learns through trial and error by perceiving and interpreting its environment in order to come up with a solution to the problem [8]. For Briscola, the goal is to pick the best card in a given state. Furthermore, by following the procedure of perceiving and interpreting its current state, the agent gets either rewards or penalties for the actions it performs [1]. Thus, the agent learns and improves its decisions over the course of its training [8].

Q-Learning was the considered approach for this paper because it is able to compare the utility of the available actions without requiring a model of the environment [6].Q-Learning is an off-policy Reinforcement Learning algorithm. The aim is to find the action with the highest Q-Value at a given state. The algorithm tries different actions at different states, so the agent can learn each state-action's expected reward, and then updates the Q-table with the new Q-value. The Q-table simply stores all the calculated Q-values in the form of [states, actions]. In Briscola the states and actions used for Q-Learning are defined as follow:

1) *State(s)*: The high suit, the opponent's card and the 3 cards of the agent.
2) *Actions(s)*: The card picked by the agent.

The process of performing an apparently sub-optimal action, i.e, an action different from a one with the highest Q-value at a given state is known as *Exploration*. Conversely, choosing the best known action, thus the action with the highest Q-value, at a given state is called *Exploitation* [12].

Hence, throughout the learning phase, the agent balances between exploration and exploitation in order to fill the Q-table. Note that *filling the Q-Table* refers to covering all possible states. The above mentioned is done using the ϵ - Greedy method.

Algorithm 2 Q-Learning

```
Initialize Q(s, a) arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (i.e. algorithm 3)
    Take action a, observe r, s'
    Q(s,a) ← Q(s,a)+α[r+γ max_α, Q(s',a')−Q(s,a)]
    s ← s'
until s is terminal
```

Algorithm 3 The ϵ Greedy for Q-Learning

```
p = random()
if p < ε return random action then
else return current best action
end if
```

Greedy methods of performing an action are *greedy* because they always exploit current knowledge to maximize the immediate reward, and are not concerned with checking if the other available actions might be better. Thus, the method can be sub-optimal. In comparison, the optimal plan would be to sufficiently test all actions. An alternative is to behave *greedily* most of the time and with a small probability, such as ϵ, randomly choose an action with all available actions at a given state having equal probability [13].

ϵ at first is high so that the agent can take big leaps and learn. As the agent learns about future rewards, epsilon decays. This way it can exploit the higher Q-values it had found.

Furthermore, if the agent chooses to perform the best current action, it will use the Bellman equation, described below, in order to calculate it:

$$Q(S_t, A_t) = (1-\alpha)*q(S_t, A_t)+\alpha*(R_t+\gamma*max_a Q(S_{t+1}, a)) \quad (2)$$

where,

- S_t = the state at time step t
- A_t = the action performed by the agent at state S_t
- α = the learning rate
- $q(S_t, A_t)$ = the current q-value of action A_t at state S_t
- R_t = the immediate reward obtained by performing action A_t at state S_t
- γ = the discount factor

Alpha (α) is the learning rate. Setting the alpha value to 0 means that the Q-values are never updated, thereby nothing is learned. If we set the alpha to a high value, such as 0.9, the learning can occur quickly.

Gamma (γ) is the present value of the future reward. It can affect learning quite a bit. If γ is set to 0, the agent is concerned with maximizing immediate rewards. On the contrary, if γ is equal to 1, the agent values future reward as much as the current reward. This means that in a given number of actions, if an agent performs a good move, this is just as valuable as doing this action directly [13].

After calculating the Q-value, the value is stored in the Q-table. It repeats the same process until all possible states of the games is covered.

C. Artificial Neural Network

Artificial Neural Network (ANN) is a Machine Learning technique, inspired by the human nervous system, used for Supervised Learning [5]. Supervised learning is the Machine Learning task of learning a function that maps an input to an output based on sample input-output pairs [11]. In other words, the ANN learns from examples from labeled data. With longer training, and thus, with more labeled examples, the network can make better decisions. Therefore, a large number of sets of input with correct output is needed in order to train a neural network.

ANNs take some features to create a layer of input nodes to then return the best output as a layer of output nodes. In between those two layers there is at least one layer of hidden nodes [5]. An activation function decides whether the neuron is activated, and thus, the information from the input layer is passed into the hidden layer(s). Using the sigmoid function (see equation 3), also known as Logistic activation function, as the activation function allows the output to be between 0 and 1, increasing monotonically with its input:

$$\sigma(z) = \frac{1}{1+e^{-z}} \quad (3)$$

where z=w*x+b, where
w = weights, b = bias, x = inputs.

The hidden layers apply a weighting function, which checks if the sum of all the weighted inputs surpasses a certain threshold value [11]. If this is the case, this value is passed to the output layer. Therefore, different number of hidden layers can also influence the performance of the neural network.

To train the neural network, the initial weights are picked from a range [-1,1] at random. Then, with each simulation the error is calculated. Backpropagation uses this error to adjust the weights [5]. This continues until the error is less than a certain threshold.

D. Data sets

Throughout the extent of this research, the data set for the implementation and simulations is all the possible combinations in a shuffled deck. We decided to use all possible decks rather than creating a set of decks that could match the number of simulations performed, due to time and memory complexity. However, an ordered deck was created to validate the decisions made by the agents (see section XI).

IV. Implementation

This section explains the implementation of the four AI agents with different levels of algorithm complexity and playing difficulty - Baseline agent, Greedy agent, MiniMax with Alpha-Beta pruning agent, and Hybrid agent. Additionally, the Cheating agent is described. Further, here is discussed the complexity analysis of these implementations, the confidence bounds, and the creation of the Graphical User Interface (GUI).

A. Baseline agent

At its core, the baseline agent is a random number generator from 0 to 2 that generates three random cards. To have a better agent than just a random card picker, there are several specialty cases:

1: **if** *highsuit* **then return** *highsuit*
2: **else if** second player & have same suit as the human card **then return** *card with same suit*
3: **else if** first player & card without points **then return** *card*

B. Greedy agent

The greedy agent is our implementation of a 1-ply greedy algorithm. It is important to divide the greedy agent's actions into two parts. First, the case in which the agent plays the first move; second, the case in which the agent plays second.

Starting with the first case, the agent checks whether it contains a high suit within its hand. If this is the case, the high suit is played. If the agent possesses only other (non-high suit) suits, the card worth the highest amount of points is played. When the agent plays its card second, this strategy

does not work optimally in terms of ensuring the winning of a round.

When the player starts the round, the agent checks the points, in hierarchical order, of every card in its current hand to see how many points each card can bring. If all the cards return a negative result, this would imply that the agent cannot win this round. Therefore, the agent plays the card that will minimize the opponent's point gain:

```
1: Choose best card from only the current state
2: if Agent starts then return card with higest probability to
   win
3: else
4:     if possible to win then return card to win
5:     else return card with smallest losing points
```

C. MiniMax with Alpha-Beta pruning agent

The MiniMax with Alpha-Beta pruning algorithm was modified to work in a partially observable environment. This was done by randomly generating cards for the opponent's hand. This is our implementation of the Monte Carlo Approximation from section III-A2. For obtaining more accurate results each round, the algorithm is not dependent on just one tree, but instead it's creating multiple trees. Then, the most frequent action is taken as the best move (i.e. the best card) for the agent to play. This is our implementation of the Determinization from section III-A2.

D. Hybrid agent

In this paper the term *Hybrid agent* refers to an AI agent using the combination of both Machine learning and Adversarial search techniques for making a decision of which action is the best. The Adversarial search technique used here is MiniMax with Alpha-Beta pruning (see section III-A2). Upon testing the Q-Learning agent, we found that the training times are far too long and impractical. This result motivated us to consider Artificial Neural Networks as an alternative Machine Learning approach. That is why the Machine learning approach is Artificial Neural Network (see section III-C).

In the case of Briscola, the neural network's input values are the AI hand's cards, the card the opponent played, the high suit, and the cards already played. However, unlike other approaches, the input and output of the neural network can only be numerical values. Thus, the input state should be represented as an array of numbers.

For input, a list of size 40 (total number of cards) is created. For each card:

- If it is in the AI hand, the input value is 1.
- If it is a high suit card, the input value is -2.
- If it has been already played, the input value is -1.
- If it is a card left in deck, the input value is 0.
- If it is a card that is on the floor, the input value is 2.

The output of this network should be the best card to play from the AI hand. This needs to be represent as numerical values as well. Therefore, the output is also a list of size 40 with all the values equal to 0, except the correct card to play, whose value equals 1.

As discussed in section III-C, ANN needs a training set. This is a set, which has a subset of all the states possible in the game and the correct decision in the said states, to train on. In case of Briscola, each state represents a round. Thus, by simulating a game, input data is set for training. However, to be able to train the ANN, the best action in those input states is needed as well. Here comes in the usage of the MiniMax with Alpha-beta pruning algorithm. As it can be seen from the experiments discusses in section V, MiniMax with Alpha-beta pruning agent is the strongest agent compared to the others. That is why the states are given to the MiniMax with Alpha-beta pruning algorithm and the best action suggested from this algorithm is saved as correct output. These decisions might not be considered the best in real situation because the opponent's card is unknown. Nevertheless, it can still output the best action from what it knows.

E. Cheating Agent

Two different ways of cheating have been implemented to give more information to the agents in order to reduce the effect of chance in the game.

Firstly, the MiniMax with Alpha-Beta pruning agent with Determinization was adapted in order to have more information when generating the random hands for the MiniMax tree. Whilst generating the random hand for the opponent, the agent has the ability to check 0, 1, 2 or 3 cards from the opponent's current hand. With this knowledge, the agent then fills the unknown information with random hand so it is able to construct a tree and use the MiniMax approach, as the agent would when not cheating. The previously mentioned tree, as the non-cheating, has a depth of 6 and a branching factor of 3, as there are no new cards dealt during the tree deepening. The amount of cards that the agent will be able to see every round, can be declared in the code settings.

For the second cheating agent, the goal was to transform the environment into a fully observable one. This was possible due to one of the limitations of the final product, as the order in which players receive new cards is maintained throughout the whole game. This means that one of the players will always receive the odd cards or the even cards, depending if they are the first or the second player. This being said, the cheating agent has the ability to check the cards that both players will receive each round and the starting hand of the opponent, to then assess the game as a fully observable environment. The cheating creates a similar tree to the MiniMax with Alpha-Beta pruning agent but without Determinization, where a branching factor of 3 is maintained. However, the depth of the tree has increased and now has the same size as the amount of cards left to play.

F. Complexity analysis of the implementation

Notation used:

- b = maximum branching factor of the search tree
- m = maximum depth of the state space
- n = number of rounds played
- c = number of cards in the deck
- Time complexity explores the number of nodes generated or expanded.
- Space complexity explores maximum number of nodes in memory.

Complexity analysis for	Greedy agent:	MiniMax with Alpha-Beta Pruning agent:	Hybrid agent:
Time complexity	$O(n)$	$O(b^{m/2})$	$O(n^3)$*
Space complexity	$O(1)$	$O(b \times m)$	$O(c^2)$

where b=3, m=6.
*Note that the time complexity of the Hybrid agent is $O(n^3)$ due to matrix multiplication.

G. Confidence intervals

A confidence interval is the range of values we expect our estimate results of the game to fall between if the test is repeated, within a certain level of confidence. The level of confidence interval refers to the range of values we expect to estimate between a certain percentage of time. As for the experiments of Briscola games, a 95% was taken into consideration. Note that by *results of the game* we are referring to the win/lose ratio.

1) Statistical analysis of play experiments: A scoring rate is defined as the following:

$$w = \frac{x}{n} \tag{4}$$

where,

- x = Number of games won
- n = Number of games played
- w = Scoring rate

In fact, we will assume the scoring rate to be the sample mean of a binary-valued random variable that counts the total amount of draws, divided by 2, added to the wins and losses. In case of an odd number of draws, the sample mean will decide the distribution of the draws. If the sample mean exceeds 0.5 included, the additional draw game would be added to the winning random variables. Otherwise, the losing random variables cover it. Furthermore, with a scoring rate the calculation of the standard error and the confidence bounds is possible.

2) Standard Errors of Scoring Rates: The standard error *s(w)* of a scoring rate $w = x / n$ is known as :

$$s(w) = \frac{\sqrt{w * (1 - w)}}{\sqrt{n}} \tag{5}$$

where,

- *s(w)* = Number of games won
- w = Scoring rate
- n = Number of games played

3) Confidence Bounds on Differences of Winning Probabilities: z denotes the upper critical value of the standard N(0,1) normal distribution for a %-level of statistical confidence of (z%95 = 1.96). With the confidence level set, we can specify the lower and upper bounds on the difference of winning probability between two agents [4]:

Lower Bounds:

$$l\% = \max((w1 - z * s(w1)) - (w2 + z * s(w2)), -1) \tag{6}$$

Upper Bounds:

$$u\% = \max((w1 + z * s(w1)) - (w2 - z * s(w2)), -1) \tag{7}$$

We will denote the range [l%, u%] by 95% confidence level. Given that there are a lot of possible different combinations of agents with and without cheating, it was decided to include one example of how the confidence interval was calculated. For this example we decided to calculate the confidence interval for MiniMax (determinization = 30) vs. simple brains, after completing:
Minimax = x1 = 78235 games won
Simple = x2 = 19342 games won
The confidence interval for a confidence level of 95% x1 lies between $0.1985352 <= 0.78235 <= 1.376395239$.

H. Graphical User Interface (GUI)

Using JavaFX, a Briscola game with GUI was created by implementing the rules from section II. Our graphics mimic the simplicity of old computer card games so the unique design of the Briscola card deck can be the main focus (see figure 2). Each card is a JavaFX Rectangle object, filled with the corresponding scanned image of the original Italian deck. When the player clicks on the card he chooses to play, a Mouse Event is triggered. This event sets the play card for the first player, flips his cards upside down, and then switches to the second player so he can choose a card as well. To maintain the real-world shuffling of cards, we first put the whole deck of ordered cards into a linked-list. Then we generate a random number between 0 and the remaining number of cards in order to find a random card and put it into a stack. This process is repeated until there are no cards left in the linked list, and instead they are all in the stack in random order. This way we make sure that 40! different possible decks are generated.

The player can choose between three game modes: Human vs. Human, Human vs. AI, and AI vs. AI. When choosing

the Human vs. Human mode, both players first have to write their names before starting the game. In the Human vs. AI mode, as well as in the AI vs. AI mode, the user can choose between five levels of difficulty: Random(Easy), BaseLine(Medium), Greedy(Hard), MiniMax(Difficult), Hybrid(Extreme). Easy level uses a random agent, which is just a random number generator. Medium level uses the baseline agent from section IV-A. Hard level uses the Greedy agent from section IV-B. Difficult level uses the MiniMax with Alpha-Beta pruning agent from section IV-C. Extreme level uses the Hybrid agent from section IV-D

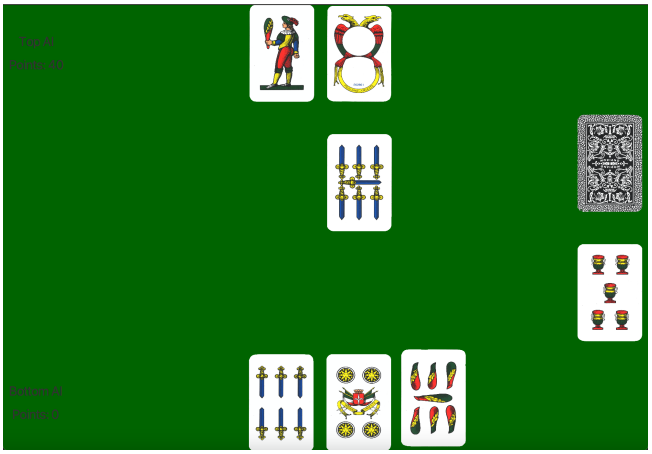

Fig. 1. Game view when 2 AI agents are competing against each other.

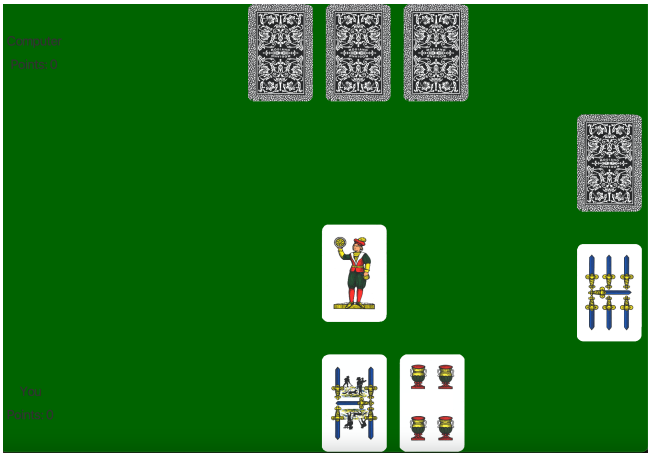

Fig. 2. Game view when human player competes against another human player. Note here the opponent's cards are always flipped upside down, so they can't be seen by the other player.

I. Basic structure of the implementation

A detailed UML diagram of our software can be found in section XIV.

V. Experiments

To facilitate experimentation, a game simulator was created where the AI agent's can play versus each other under a controlled environment. This game simulator is running a similar setup to the one used in the back-end of the GUI, where one AI agent is at the top and the other is at the bottom of the screen, thus creating a two player setup where each agent can have a different difficulty.

Firstly, an experiment was designed in which an agent played 10,000 games versus each of the other agents. The winner and the points obtained by each player was record in a text file, after each game. Once all agents played versus the other, we assessed which AI agent was the most efficient, by taking in consideration the victory percentage and their computational complexity. Once the results were recorded in a table, a set of graphs (see section VI) was created in order to compare the performance of each agent and draw conclusions.

The second experiment that was carried out was designed in order to conclude which was the best number of determinizations for the MiniMax with Alpha-Beta pruning agent with Determinization in order to find a balance between time complexity and won games. During this experiment, 10,000 games were simulated with different determinization values. This was repeated 3 times, as the MiniMax with Alpha-Beta pruning agent with Determinization played against the Random, Baseline and Greedy agents individually.

In order to reduce the effect of being the starting player at the first simulation, the 10,000 games were played twice so both players can have that advantage. The reasoning behind this decision is the following: if an agent starts a game, the knowledge of the cards is very limited and can only play on luck; whilst the second player can act on the card that is given by the opponent. With this, the second player can gain more points if interest arises, and start taking control of the game. After running the 10,000 games twice, the results of both simulations were averaged and recorded in the table from section XIII and shown in 2 graphs: one showing the percentage of games won and the other one - the time complexity in seconds.

The possibility of different decks ($40! = 8.159 \cdot 10^{47}$) is so large that it poses one main difficulty, as running experiments on the different agents against each other can not replicate the same deck most of the times, thus, not having the same points scored. To counter this, we ran tests over a fixed deck with same card dealings, as well as high counts of tests over random deck deals.

As mentioned in the start of the section, using the game simulator helped to create a controlled environment for the experimentation. For both experiments, the game was ran using the fixed amount of simulations and a fixed difficulty. Additionally, each game was played completely (i.e. 20 rounds), so 120 points can be won in each game. During the experimentation, there was no output in the terminal and only the end game results were printed in the text file. Both experiments were carried out using an Apple MacBook Pro computer with Intel i5 with 8 GB of memory in a single processor core.

VI. Results

The Greedy agent averaged over the 10,000 game runs both as first and second player, except in Greedy vs. Greedy - this is used to show the first vs. second player difference:

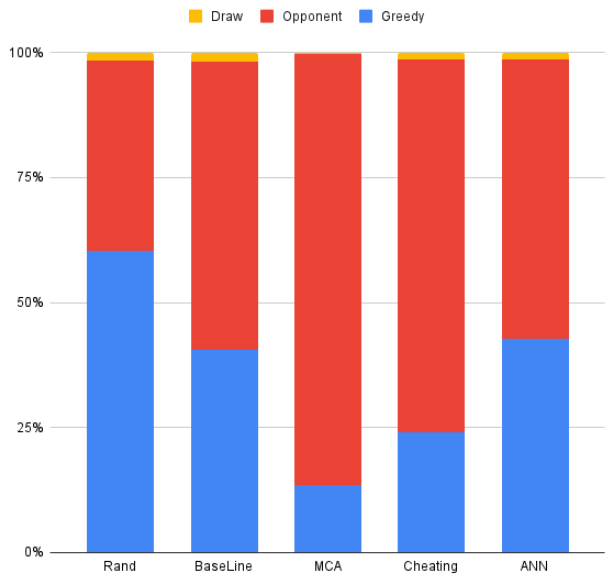

Fig. 3. Greedy agent against the other agents

The experimentation of first vs. second player in each game is representing the simulations the blue player is starting. The agents are playing against another version of the same level agent - this is done with every game the first player being the same:

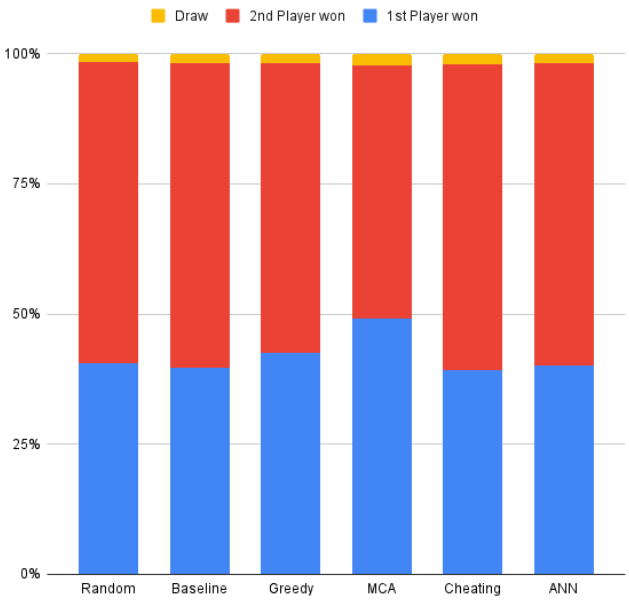

Fig. 4. Each agent against a second version of the same agent

The Random Hand, generated by the MCA, can take in a certain number of real cards of the opposing hand. Here we can see the impact the number of cards of the opposing hand can have:

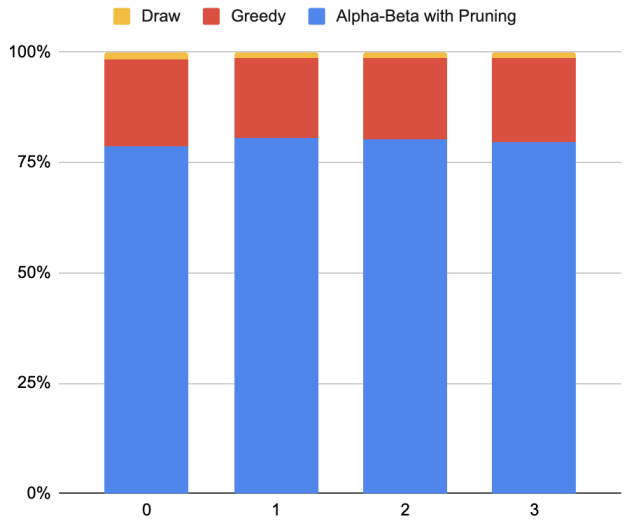

Fig. 5. Number of cards taken from the opponent's hand to cheat in the alpha-beta pruned MCA

VII. Discussion

The three single state agents: Random, Baseline, and Greedy all were very quick, within seconds, for the 10,000 game runs. All three together in all possible combinations, starting with Random against each agent including itself, take 6 seconds. On the contrary, the multi-state agents, MCA with and without pruning and the Hybrid agent all require longer due to their time complexities (as stated in section IV-D). The time required for running all the multi-stage agents against each other with Determinization of 30 took 40 minutes.

Testing all the agents against each other through a game simulation of 10,000 rounds, showed the experiment number 2 on checking the first vs second player win rates as written out, the single state greedy agent are 42.6% for the first player and 55.6% for the second player while 1.84% go into a draw. This is coming down that every game a fixed starting player is being used and further this player is on the back foot as the other player can already act on and gain more wins.

As shown in figure 5, the winning possibility against the Greedy agent do not massively increase while raising the amount of cards taken from the opponents hand. This comes mainly down to the Determinization used in this run. The 30 determinizations result in a wide array of hands and check the possibility of them all. The usage of a differentiating random cards for each randomized hand and the rest from the remaining cards unknown to the agent.

For the determinization experiment it is clear that a higher value will bring a higher percentage of games won, as the difference in the percentage of games won between 3 and 500

determinizations is approximately 12%. However, running 500 determinizations is almost 400 times slower (see section XII). That being said, it was crucial to find an intermediate value. Looking at the table in section XII, on average, the better performance comes after 30 determinizations compared to 3, as it is at least 8% better after this. Even though this positive difference keeps increasing as determinizations are increased, the MiniMax winning percentage starts to flatten out and the difference between 30 and 500 determinizations is below 2%. Furthermore, looking at the change of time complexity, it is evident that the effort done to win more games becomes higher. For example, the time complexity of 30 determinizations (69 seconds) compared to 500 (1143 seconds), is around 15 times faster but the winning percentage only increases by 1.5% (see section XII). Additionally, even the difference between 30 and 40 determinizations makes it more convenient to run 30 rather than 40, as the winning percentage of MiniMax increases by 0.05% but it is 1.3 times slower. That being said, it is evident that 30 determinizations bring the highest winning percentage at a cheap computational time.

VIII. Conclusion

This paper examined the MiniMax algorithm with Alpha-Beta pruning as an Adversarial Search technique and was enhanced by using Determinization with a value of 30 to balance the percentage of games won and time consumption. In regards to the Machine Learning approach, Q-learning was examined. However, due to impractical training times, an Artificial Neural Network, utilizing input-output pairs calculated using the MiniMax algorithm. was used to train the agent and act as a hybrid system. That being said, it was crucial to find the best Adversarial Search technique, whilst maintaining an appropriate time consumption. It is clear from the experimentation that these methods in combination resulted in the best performance hybrid agent against other agents. In conclusion, this paper provides the answers to the proposed research question that a MiniMax algorithm with Alpha-Beta pruning enhanced by Determinization in combination with an Artificial Neural Network can work as hybrid goal-based system to play Briscola.

IX. References

References

[1] B. Bakker, V. Zhumatiy, G. Gruener, and J. Schmidhuber. Quasi-online reinforcement learning for robots. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2997–3002, 2006.

[2] Burch N. Johanson M. Tammelin O. Bowling, M. *Heads-up limit hold'em poker is solved*, volume 347. 2015.

[3] Peter I. Cowling, Edward J. Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

[4] E. A. Heinz. Self-play, deep search and diminishing returns. *J. Int. Comput. Games Assoc.*, 24:75–79, 2001.

[5] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997.

[6] Ms.S.Manju and Dr.Ms.M.Punithavalli. An analysis of q-learning algorithms with strategies of reward function. *International Journal on Computer Science and Engineering*, 3, 02 2011.

[7] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4):293–311, 2013.

[8] Deepshikha Pandey and Punit Pandey. Approximate q-learning: An introduction. In *2010 Second International Conference on Machine Learning and Computing*, pages 317–320, 2010.

[9] Glen Robertson and Ian Watson. A review of real-time strategy game AI. *AI Magazine*, 35(4):75–104, 2014.

[10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition, 2010.

[11] Rehan Sadiq, Manuel J. Rodriguez, and Haroon R. Mian. Empirical models to predict disinfection by-products (dbps) in drinking water: An updated review. In Jerome Nriagu, editor, *Encyclopedia of Environmental Health (Second Edition)*, pages 324–338. Elsevier, second edition edition, 2019.

[12] Yuanxia Shen and Chuanhua Zeng. An adaptive approach for the exploration-exploitation dilemma in non-stationary environment. In *2008 International Conference on Computer Science and Software Engineering*, volume 1, pages 497–500, 2008.

[13] Barto Sutton, R.S. *Reinforcement Learning: An Introduction*. MIT Press, second edition, 2018.

[14] Csaba Szepesvári. Algorithms for Reinforcement Learning. *Synthesis lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

X. Appendix A: Q-Learning

The Reinforcement Learning agent is our implementation of the Q-Learning algorithm. At the beginning of the training phase, a Q-table is initialized, which in essence is a two-dimensional matrix of size 13525720 x 3. When a new game of Briscola is started, a high suit card is drawn randomly from a shuffled deck of 40 cards and kept aside. Then, 3 cards are drawn for the agent from the remaining deck of 39 cards. The total number of 3-card hands that can be drawn from 39 cards is ${}_{39}C_3$ which is equal to 9139. Lastly, one card is drawn for the opponent's hand in each round. The total number of opponent's cards possible is ${}_{37}C_1$ equal to 37. Note here that the high suit card can be dealt to the opponent in the final round of the game. Hence, the total state space for our model of the game can be calculated as $40 * 9139 * 37 = 13,525,720$. Therefore, the number of states represent the number of rows of the q-Table while the total number of actions per state or per row is equal to 3.

All the q-values in the Q-table are initialized to be zero because the agent has no knowledge about the environment.

It is important to describe the two do-while loops that dictate how the agent is trained. The outer do-while loop continuously simulates a large number of games and terminates when all the q-values of the Q-table are non-zero. Another option is to simulate some number of games to reach a certain value and terminate the outer while loop when the `gameCounter` reaches this value. In the body of the outer while loop, a method to start a game of Briscola is called. In regards to the inner do-while loop, a card is drawn from the deck and is assigned to be the opponent card. An action to be played by the agent is picked using the ϵ - Greedy policy (see section III-B), and a state constructor is called which gives as parameters the agent's hand, the high suit of the game, the action played, and the opponent's card. After the action is picked, the card played by the agent and the opponent's card is checked, and the points won or lost by the agent are assigned to a variable called `immediateReward`. The value of `immediateReward` is positive for the number of points it has won or negative for the number of points it has lost.

After the two cards have been played and checked, a new card is dealt to the agent to replace the previously played card in the hand, and a second new card is drawn from the deck and is assigned as the opponent's card, effectively transitioning from state S_t to state S_{t+1}. Upon transition to the new state, the q-value for the action played in the previous round is updated using the Bellman equation (see equation 2 in section III-B), which updates the q-value of the performed action in the row of the Q-table corresponding to the state. After updating, the state S_{t+1} is set to be the `currrenState` and the whole process is repeated until the termination condition is reached, which is, the deck runs out of cards. To keep track of the correspondence of states and rows in the Q-table, we implicitly assign each state an integer value `stateID` every time a state constructor method is called. This `stateID` also represents the row number of that particular state in the Q-table.

Upon completion of training, the agent would be able to play against an opponent. Every time the agent has to act, it recognises the state and looks up the Q-table to play the action with the highest Q-value.

XI. Appendix B: The Ordered Deck

Each card of a deck contains an index or "ID" which is described as follows:

- Index 0-9: Cards 2 to ACE of suit 'Club'
- Index 10-19: Cards 2 to ACE of suit 'Coins'
- Index 20-29: Cards 2 to ACE of suit 'Sword'
- Index 30-39: Cards 2 to ACE of suit 'Trophy'

"Cards 2 to ACE" is described in the following manner: $2, 4, 5, 6, 7, Jack, Knight, King, 3, Ace$

To generate a fixed deck, an empty stack is initialized and cards are added to the stack in a random manner. After all 40 cards are added, the stack is stored and can be used for experimenting and simulating games.

XII. Appendix C: Average performance of Minimax vs the Simple agents

Determinization	% games won by MiniMax	Time (sec)
1	72.62333333	3.030166667
3	75.50833333	6.977
5	77.32	13.07133333
10	79.76	24.3685
20	81.77333333	47.152
30	83.01333333	69.59216667
40	83.05	93.06683333
50	83.02166667	116.8173333
100	83.925	233.8076667
150	84.05	343.514
200	84.29166667	446.543
500	84.525	1143.4745

XIII. Appendix E: Table with Determinization results

	Random agent		
Determinizations	Games won	% Games won	Time (sec)
1	7667.5	76.675	3.75
3	8001	80.01	8.607
5	8111.5	81.115	13.201
10	8329.5	83.295	24.497
20	8452	84.52	47.1585
30	8589.5	85.895	69.731
40	8604	86.04	95.989
50	8654	86.54	121.797
100	8701	87.01	237.155
150	8676.5	86.765	336.2825
200	8712	87.12	449.048
500	8773.5	87.735	1210.446
	Base line		
Determinizations	Games won	% Games won	Time (sec)
1	6610	66.1	1.746
3	6914	69.14	4.06
5	7158	71.58	12.8135
10	7426.5	74.265	24.166
20	7726.5	77.265	47.386
30	7816	78.16	69.29
40	7807	78.07	90.5085
50	7781.5	77.815	113.8005
100	7925.5	79.255	224.823
150	7933	79.33	353.6145
200	7918.5	79.185	440.4975
500	7922.5	79.225	1107.3575
	Greedy		
Determinizations	Games won	% Games won	Time (sec)
1	7509.5	75.095	3.5945
3	7737.5	77.375	8.264
5	7926.5	79.265	13.1995
10	8172	81.72	24.4425
20	8353.5	83.535	46.9115
30	8498.5	84.985	69.7555
40	8504	85.04	92.703
50	8471	84.71	114.8545
100	8551	85.51	239.445
150	8605.5	86.055	340.645
200	8657	86.57	450.0835
500	8661.5	86.615	1112.62

XIV. Appendix D: Basic structure of the implementation