Using Adversarial Search and Machine Learning to create a Dice Chess Playing Agent

Gevorg Chakrian (i6244940), Sebas van Sluijsdam (i6252182),
Laurence Manoukian (i6249348), Imanol Mugarza (i6241042),
Dan Parii (i6254321), Alessandro Corvi (i6218941), Marie Picquet (i6259291)
Department of Data Science and Knowledge Engineering
Maastricht University
Maastricht, The Netherlands

***Abstract*—Dice chess is a variant of chess. In dice chess, chess is transformed into a probabilistic game because there is a dice involved. It is studied so that an artificial agent can make better decisions when faced with probability. However, no research has been conducted on Dice Chess specifically. This paper is primarily concerned with finding the best possible combination of algorithmic methods. An adversarial heuristic search algorithm has been developed based on Monte-Carlo tree search. A logistic regression component has been added to this algorithm, making it a hybrid agent that uses both adversarial search and machine learning methods. In addition, expecti-minimax, another adversarial search algorithm, has been developed to compare with the hybrid agent. The MCTS' performance is influenced by a variety of factors. The win percentage lowers from 99 percent to 93 percent when the win score is increased from 0 to 2.4. Because MCTS only has one second per move, varying the Exploration coefficient produced no useful results. The assessment functions are critical for MCTS performance, according to the findings. It has a 50 percent victory rate with a standard time of one second. While the time grows to ten seconds, the win rate improves to 70 percent. At two seconds MCTS against expectiminimax has a win rate of 38 percent, while the hybrid has only 33 percent. increasing the time to five seconds, MCTS has a 51 percent win rate while the hybrid has 56 percent. a combination of three parameters can significantly increase the performance of a hybrid MCTS agent. Furthermore, the Hybrid MCTS Agent performs poorly at low execution times. It will outperform a normal MCTS agent at a particular point of higher execution time.**

I. Introduction

Chess has fascinated people for many years and has taken on a symbolic meaning since the first pre-computer devices more than a century ago [1] [2]. In 1997, the AI Deep Blue was the first chess machine to defeat the then-reigning World Chess Champion Garry Kasparov in a six-game match [3]. Since Deep Blue beat the grand-master, other AIs have been created, such as Alpha Zero, which uses a Neural Network, trained via reinforcement learning, to determine the best possible move [4], or the Stockfish agent, which uses a database of records from matches of professional chess players to compare the done turns and return the best possible move. [5]

Machine Learning techniques have enabled a new form of chess-AI's, very different to the initial Adversarial Search agents (such as Deep Blue), adding new possibilities for hybrid agents into the world of chess [6].

The game's popularity resulted in the creation of many variants [7]. Chess is a strategic game that does not include luck or probability. Dice chess is a variant that includes a random component. In particular, each turn the piece that can be moved is determined by a die roll.

Adding this type of stochastic component to a strategic game makes it different to the well-known classical chess as it is more difficult

for a player to build a strategy and predict the opponents next moves. Not much research has been done on this variant and specific AI agents for Dice chess were not discovered when doing research.

The main purpose of this report is to investigate what type of Artificial Intelligent agent, deals the best with probability in the game Dice chess. It will be discussed what specific parameters can improve an agents performance and how Adversarial Search and Machine Learning techniques can be combined to form a good hybrid agent.

Dice chess itself has many variations [7]. In this project the variation with the following rules is used [8]:

Dice Chess follows the same rules as classic Chess [9]. The moves available to each player are determined by the outcome of an ordinary six-sided die, where each number represents a chess piece to be moved. The pawn is represented by 1, the knight by 2, the bishop by 3, the rook by 4,the queen by 5 and the king by 6. If a piece does not have any available moves, the according die number cannot be rolled. There are no check or checkmates; the game only ends when a king is captured. If a pawn is to be promoted (would advance to the last row), the player can move it even if the die does not show 1. However, he can only promote it to the piece chosen by the die roll - for example, if 3 is rolled, the pawn can be promoted to a bishop only. If 1 is rolled, the pawn can be promoted to any piece.

The paper is organized as follows. In section 2, the different methods used in this project are explained. Next, in section 2, a short explanation to the implementation of the different agents is provided, namely the Goal-based Agent, the Monte-Carlo Tree Search, the Hybrid Agent composed of the Monte-Carlo Tree Search and a Logistic Regression bias and the Expectiminimax. Section 3 is dedicated to the experiments performed with the different agents and section 4 will present the results of these experiments. Finally,the conclusion will be presented in section 5.

II. Methods

This section introduces the methods used in this paper and is organized as follows. First, the Evaluation Function used to interpret the quality of a certain state in the adversarial search algorithms is introduced in part A. Then follows the Monte-Carlo Tree Search in part B, the Logistic Regression method in part C and finally the Expectiminimax method in part D.

A. Evaluation function

An evaluation function is a type of function that uses the states of the chess pieces as parameters, compares them to the states of neighbouring pieces, and predicts the best possible goal state [10]. For instance, a square on the chessboard for that specified piece to move to. The good evaluation function adjusts a balance between the time consumption and accurately estimating the positional advantage. The evaluation function has to be skillful so that the AI makes moves that lead to better positions. On the other hand, the algorithm also needs to be quick and flexible because the faster it is, the deeper into the future moves the engine can look, and consequently better the AI will play. Evaluation functions are comprised of material cost evaluations that iterate through every piece given the positional state of each piece. Generally, the sum of the positions does not exceed that of a pawn evaluation. Since in this variant, where an additional parameter of discrete probability is added and there exists no checkmate, the evaluation function take this from:

$$\begin{aligned} Eval_x &= c_1 b_1 + c_2 b_2 + c_3 b_3 + c_4 b_4 + \cdots + c_8 b_8 \\ &= \sum_{i=1}^{m} c_i b_i, \forall i = 1, \ldots, m = 8, \end{aligned}$$

Where b_1, b_2, …, b_8, denote respectively the material, mobility, king safety, center control, king tropism, trapped pieces, piece eatable, and suicide rate, respectively and x represents the given state.

Queen	9
Rook	5
Knight	3
Bishop	3
Pawn	1
King	M

TABLE I: The values for pieces in Material.

The material score is obtained by assigning a value in pawn-unit to each piece Table I. Where M is a sufficiently massive value to differentiate this piece from other pieces, as the king is the most important for each player on the chessboard. An assumption on the material score is if Player 1 has a greater material score than than Player 2, then Player 1 has a material advantage over Player 2.

The mobility is the amount of legal, allowable squares a specific piece can move towards. The higher the number of moves, the better the square the piece is in. [1]

Trapped pieces are examples of poor mobility. Their position is awkward and often endangered. It is great to know the enemy's piece that cannot escape from your attack.

The king's safety score is a set of positive scores considered as advantages, and negative scores considered as disadvantages that are evaluated based on:

- state position of the king piece.
- position of ally pieces with respect to the king position.
- position of enemy pieces with respect to the king position.

The center control evaluates from how many pawns and pieces occupy or carry the four center spaces, with the maximum of 12 spaces to expand the center.

King tropism is a reward for the proximity of certain chess pieces, especially queens and knights, to the opposing king. It was also calculated whether a piece is eatable on the board, which helps targeting the direction of our attacks. Suicide rate evaluate the rate that the piece will be captured by an opponent piece in that square position.

B. Monte-Carlo Tree Search

The Monte-Carlo Tree Search (MCTS) is an adversarial heuristic search algorithm. "Monte Carlo" algorithms are randomized algorithms named after the Casino de Monte-Carlo in Monaco. [11, Chapter 5] The MCTS is simulation-based and is used to get the best move at a given state in a Game play. [12] As it is shown in Figure 1, the algorithm is divided in different phases: the Selection, the Expansion, the Play-out(or simulation phase) and the Backpropagation. These phases are repeated until the algorithm reaches the set number of iterations and the move with the most playouts is chosen.

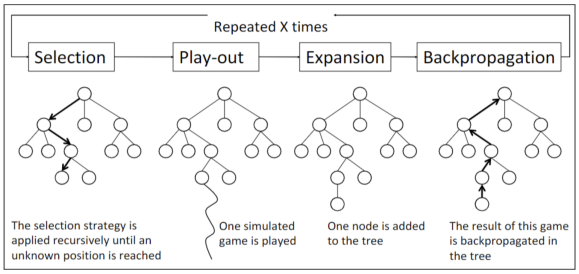

Fig. 1: Outline of Monte-Carlo Tree Search [13]

1) UCT strategy: A selection policy is used to balance the exploration of states that have had few playouts and the exploitation of states that have been successful in the previous playouts. The "Upper Confidence Bounds Applied to Trees" (UCT) was used as a selection policy as it has proven to be very effective. [11] The formula is:

$$b \in argmax_{x_i \in I}(v_i + C\sqrt{\frac{ln(n_p)}{n_i}})$$

where v_i is the value of the node, n_i is the visit count of node I, n_p is the visit count of node P and C the coefficient to balance exploration/exploitation.

It was chosen to implement MCTS using an evaluation function together with the UCT to determine the most promising move.

2) Selection: Starting at the root node, a move is chosen using the selection policy. This is executed until leaf node is reached.

3) Expansion: During the Expansion phase, the most promising node according to the UCT formula is expanded. All the possible moves at that node, represent the new layer of the tree.

4) Play-out: Using the selection policy to choose the action, a playout is performed from the resulting child node.

5) Backpropagation: The result of the playout is used to update all the nodes of the tree and the steps are repeated.

6) Progressive Bias: Progressive Bias is used to direct the search according to heuristic knowledge.A method of doing so is modifying the selection strategy according to a heuristic function. The formula is setup in such a way that this knowledge is important when few games have been played, but this importance decays with the increase of played games. The UCT formula was modified:

$$b \in argmax_{xi \in I}(v_i + C \times \sqrt{\frac{ln\ n_p}{n_i}} + f(n_i) \quad (1)$$

$f(n_i) = \frac{H_i}{n_i+1}$ where H_i was chosen to represent the heuristic knowledge, which in this case is determined by Logistic Regression, H_i being the probability of a move being good.

C. Logistic Regression

[14]

Logistic Regression is a classification algorithm, that learns to approximate $P(Y \mid X)$, so for a single training data-point(x,y),LR assumes:

$$P(Y = 1 \mid X = x) = \sigma(z)\ where\ z\ = \theta_0 + \sum_{m}^{i=1} \theta_0 x_i$$

where θ is the weight of the respective parameter x_i.

Generally it builds upon the idea of Linear Regression, by predicting the probability that a set of parameters yields some result, however whereas Linear Regression is used to predict the continuous, Logistic is generally used to predict the categorical.

It is a machine learning algorithm, as it utilizes labelled data, to determine the error between a predicted output, and the actual one. Thus, the algorithm uses the error to determine how to change the weights of the parameters such that error is minimized, learning how to make a more accurate prediction.

1) Sigmoid function: In order to map predicted values to probabilities, the Sigmoid function is used.The function maps any real value into another value between 0 and 1:

$$h_\theta(x) \qquad = \qquad \frac{1}{1 + e^{-\theta^T x}} \quad (2)$$

The Decision Boundary determines if a sample is classified into one of two categories. For example,considering classes 1 and 0, for sample X: $P(Y = 1|X) \geq c$ then X is categorized as 1, else it is categorized as 0.

2) Cost function and Gradient Ascent: As mentioned previously, Logistic Regression uses the difference between the actual and predicted output to change the weights. However, how is this difference used to optimize the predictions? This is achieved by using a cost function, that reflects the error, and continuously minimizing it:

$$Cost(h_\theta(x), y) = \begin{cases} -log(h_\theta(x)) & \text{if } y\text{=1} \\ -log(1 - h_\theta(x) & \text{if } y\text{=0} \end{cases} \quad (3)$$

where h_θ is the predicted class, and y is the actual class.

The gradient of a continuous function f is defined as the vector that contains the partial derivatives $\frac{\delta f}{\delta x_i}(p)$ computed at that point p. The gradient is finite and defined if and only if all partial derivatives are also defined and finite. With formal notation, the gradient is indicated as:

$$\nabla f(x)\ =\ [\frac{\delta f}{\delta x_1}(p), \frac{\delta f}{\delta x_2}(p), ..., \frac{\delta f}{\delta x_{|x|}}(p)]^T \quad (4)$$

When used for optimization, either gradient descent or ascent can be used, in this research the latter was used.

3) Data: Dice-chess is a complex game, so to make relatively valid predictions a large data-set is necessary. For our training, a set comprising of 1.6 million moves was used. [15] However, the data comes from regular chess, whereas dice chess moves are required, the approach used to deal with this inconsistency will be mentioned later. The data-frame's columns are the following:

- The first 64 columns are a representation of the 64 squares of the current board.
- Columns from 65 to 128 represent the starting position of a move: 1.0 if a piece was moved from that square, 0.0 otherwise.
- Columns from 130 to 192 represent the ending position of a move: 1.0 if a piece was moved to that square, 0.0 otherwise.
- The last column is the label: True if it was a good move, False otherwise.

A 'good move' is a move that the winner played at some point during the game. A 'bad move' is a legal move that the winner chose not to play.

4) One-hot encoding: The representation of the board was with Strings, however Logistic Regression uses only doubles.In order to circumvent this issue one-hot encoding was used. There are 12 distinct pieces in chess, so for representing a square a vector of length 12 was used, with each index corresponding to a specific piece, if the element at that index is 1 then piece is on the square,else it is 0.

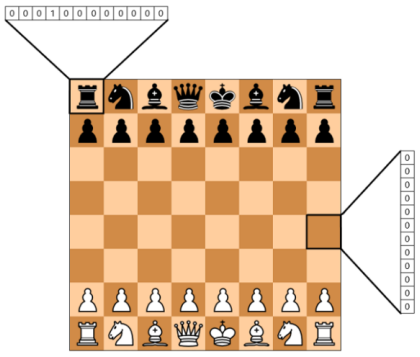

Fig. 2: Board Representation

As a result, a sample is represented by an 896 length vector: $12 \times 64 + 64$(starting square)+64(ending square)

5) Data converted for Dice-chess: For chess, in a certain state, some moves may be considered bad, whereas in dice chess they are not, since there is only one piece that can be moved, the collection of legal moves decreases.This is why 12 separate weight were trained for vectors for each piece in particular, and used w.r.t to the dice roll.The accuracy and precision of each vector may be seen in the results section.

6) Data Imbalance: The Data-set is structured in such a way that, the bad moves provided are the legal moves a player could have played in a state, but did not, and the good move is the one that was played. As you can imagine, the samples of each class are quite imbalanced, so oversampling was used to make the proportion of good/bad move more suitable.

D. Expectiminimax

Furthermore, an Expectiminimax algorithm was implemented in order to compare and test the other agents. The Expectiminimax is an adversarial search algorithm which given an initial state outputs the best possible move selon.... It follows the logic of the Minimax algorithm used in deterministic environments. However, due to the probability in the game of Dice Chess, it is not possible to use the latter. An altered version of the algorithm is used instead, Expectiminimax, which includes Chance nodes as well as the /or classic Max and Min nodes and is suitable for a stochastic environment such as Dice Chess.@ [11, Chapter5] The Expectiminimax tree starts with a Max node at a given state and roll. Using depth first search, the tree is traversed in the following order: Max, Chance, Min, Chance, Max, Chance, Min, etc. This is executed until a terminal state or the set depth limit has been reached. The score of that leaf node is calculated using the evaluation function and the values of the previous nodes are calculated by backpropagating up the tree. As shown in appendix A, the branches leading from

the Chance nodes to their child nodes represent the possible dice rolls and their probability. The value of a Chance node x is calculated using the formula:

$$value(x) = \sum_{\forall child(x)} (P(r) \times value(child(x))$$

where r is the dice roll, $P(r)$ the probability of getting this specific roll and $child(x)$ a child of node x

The values of a Max node is calculated by choosing the the child with the highest score and for a Min node the child with the lowest score:

if x is a max node:
$\forall child(x), value(x) = max(value(child(x))$

if x is a min node:
$\forall child(x), value(x) = min(value(child(x))$

III. Implementation

In this section, the different agents implemented in the purpose of the experiments will be presented. In part A, the implementation of the Goal-based agent will be discussed. Then, in part B, the implementation of the Monte-Carlo Tree Search will be explained. Follows the logic behind the implementation of the Hybrid Agent using Logistic Regression in part C and finally the implementation of Expectiminimax in part D.

A. Goal-Based agent

As a baseline agent a simple reflex agent was chosen which later evolved into a goal-based agent.

There have been mainly two types of the agent goal models, task oriented model and state oriented model [16].

For our project a state oriented model was the best fit. The process of the agent can be observed in Fig. 1. The goal based agent uses a simplified version of the evaluation function which only takes into account what the pieces are able to be eaten and a state risk evaluation function. A state risk evaluation function analyzes the probability of getting eaten in that particular state and evaluates how much to subtract from the base score based on which pieces are at risk.

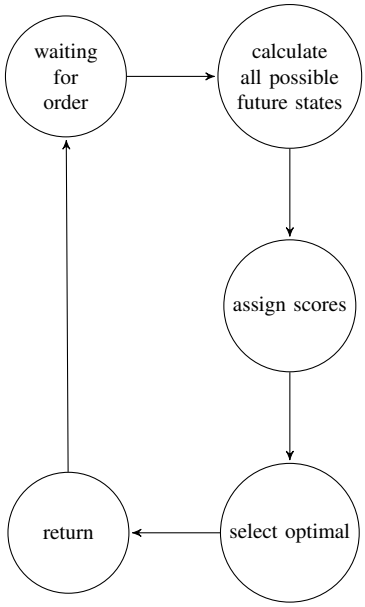

Fig. 3: Goal Based agent process.

Piece	Value
Pawn	1
Knight	2
Bishop	3
Rook	4
Queen	20
King	1000

B. Monte-Carlo Tree Search

The Monte-Carlo Tree Search algorithm was performed by mainly three classes:

- MonteCarloTreeSearch
- UCT
- Node

1) MonteCarloTreeSearch: This is the main MCTS class where the four different steps (Selection, Play-out, Expansion and Back propagation) happen, for more information Fig. 2.

We begin by creating a tree starting the root node which is of course the current state. We then continue by entering a while loop which revolves about the variable **MCTSTIME** which defines the maximum execution time of a single MCTS turn. We begin with finding a promising node with the

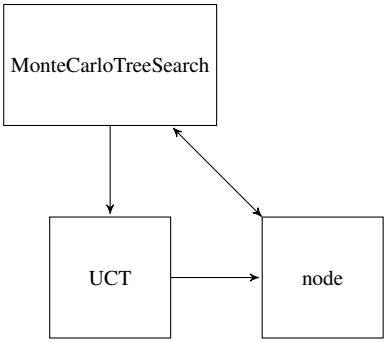

Fig. 4: MCTS UML.

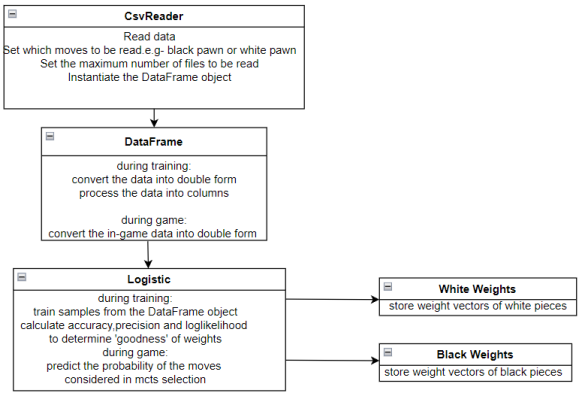

Fig. 5: Logistic Regression Implementation

selection algorithm; if by now we don't have a good enough node we explore the original node by adding all possible children. If the node we found is pleasing we can then get a random child and play out the scenario. In the end of the loop we arrive at the backpropagation algorithm which goes back to increment the visit count.

2) UCT: The UCT class is used for the Selection phase and is a *static* class directly called from the MonteCarloTreeSearch class. The class begins by calculating all possible children from the root node and once a node satisfies the **uctValue** it get returned as the chosen node.

3) Node: The node class is a very important class because it allows us to create the tree for the MCTS to work on. Every node class contains the information about its children and about its parent. Thanks to this feature from each node we are able to reach every node in the tree.

C. Logistic Regression

The logistic regression aspect of the project was performed by five classes: CsvReader,Dataframe,Logistic,Black Weights,White Weights. The CsvReader class is used to read the data from the kaggle data-sets, and convert them to samples.The Dataframe class processes each sample in different columns,converts each move to double a double vector, to be used by the Logistic class in training the weights.

1) Hybrid Agent/Progressive Bias: Once all of the weights have been calculated, at the selection stage of the MCTS, the in-game data was converted into a double vector,suitable for Logistic Regression, and then use the probability of the good move in the UCT strategy.1 This is the extent of the Machine Learning aspect in our Monte-Carlo Agent.

D. Expectiminimax

The tree for the Expectiminimax algorithm was implemented recursively using the following pseudo-code:

$$ExMinMax(s) = \begin{cases} \text{UTILITY}(s, MAX) & \text{if Is-TERMINAL}(s) \\ max_a \text{if ExMinMax(RESULT}(s,a) & \text{if To-Move}(s) \\ min_a \text{if ExMinMax(RESULT}(s,a) & \text{if To-Move}(s) \\ \sum_r P(r) \quad \text{if ExMinMax(RESULT}(s,r) & \text{if To-Mov} \end{cases} \tag{5}$$

After getting the best value for the Expectiminimax, a choose move method was implemented to get the best resulting state.

IV. Experiments

For the experiments, several simulations of different algorithmic implementations were carried out over a span of a total n large times, where n=100. The algorithmic implementations for which the experiments were performed on were:

- Different variants of Monte Carlo Tree Search (MCTS), namely a standard and a hybrid variant.

- Adversarial Search, which is expectiminimax.

A. MCTS Experiment

In order to test the performance of the MCTS, four different subexperiments were performed, each varying a different variable. Before conducting these experiments, a base case was defined to be compared, and standard values remained the same for each experiment. The following values made up this base case:

Exploration Coeff.	3.8775
Win score	0.4
Bias	True
MCTS time	1000

To conduct the experiments, the MCTS played a certain number of games against the goal-based agent, with half of the games being played as white and the other half as black. The results of each game were recorded so that the winning percentage of MCTS could be calculated after the experiment.
The first test involved varying the exploration coefficient, c, ranged within an interval (0,10), without a standard increment. A total of n=100 games were played for each value. Secondly, the win score ranged between (0-2.4), with a standard increment of 0.2. In this experiment, 100 games were played for each value. The third experiment compared the difference made by the bias. Two experiments were conducted, one with bias and the other unbiased. For each experiment, n=400 games were played. Finally, an experiment was conducted to determine the effect that the amount of time MCTS was given for each move could have. However, the experiment was conducted without the evaluation functions, without loss of generality, such that the importance of the functions could be determined.
These experiments were chosen to identify the factors that have the most impact on the performance of the MCTS, and to discover the values for which it performs best.

B. Final experiments

Following the completion of MCTS and expectiminimax, final experimental trials were conducted in which both agents played against each other. For MCTS, both hybrid and non-hybrid agents were used. This experiment consisted of 100 matches, in which half of the games were played as white and half as black, for both agents. The only variable that was altered was the maximum time MCTS could use for each move. A total of three different time allowances were used: 2000, and 5000. The experiments help answer the research questions. The results will show what type of agent performs best and whether hybrid agents outperform their non-hybrid counterparts.

V. Results

A. MCTS vs Goal-based Agent

After reviewing the fundamental characteristic of the MCTS its functionality on Random Agent was tested. The results of MCTS playing 1000 games versus Goal-based Agent (500 per board side) are visualized on Fig. 6

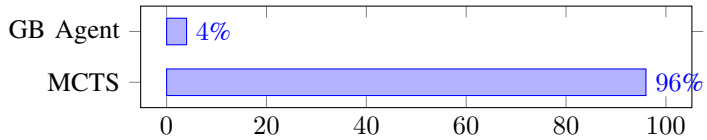

Fig. 6: MCTS VS Goal-based Agent

B. Overview of the Results

The following results were calculated, using three distinct statistical analysis [17] equations to make observations and deductions from the outcomes:

- The score rate, w, is given by:

$$w = \frac{x}{n} \tag{6}$$

where in Eq.6,x: score rate, n: number of tests, x: number of wins

- The standard error in Eq.7, $s(w)$ of a scoring rate w is given by:

$$s(w) = \frac{\sqrt{w * (1 - w)}}{\sqrt{n}} \tag{7}$$

- The confidence upper and lower bounds that determine the potential of one agent performing better than the other is given by:

$$l_{\%,(w_1,w_2)}=max((w_1-z_\%\times s(w_1))-(w_2+z_\%\times s(w_2)),-1) \quad (8)$$

$$u_{\%,(w_1,w_2)}=max((w_1+z_\%\times s(w_1))-(w_2-z_\%\times s(w_2)),1) \quad (9)$$

where in Eq.8 and Eq.9, the value $z_\% = z_{95\%}$=1.96 is known as the critical constant of a standard Normal Distribution, $N(0,1)$, such that$\forall -1 \leq l_\% \leq u_\% \leq 1$. Thus, the results of the upper and lower confidence bounds between the distinct algorithms are provided below in form of tables II and III provided below.

List of performed experiments against ExMinMax			
MCTS	$t(1000l)$	w	$s(w)$
Normal	5000	0.51	0.04999
Hybrid	5000	0.56	0.04964
Normal	2000	0.38	0.04854
Hybrid	2000	0.33	0.04702
Normal*	5000	0.12	0.03250

TABLE II: This is a table comparing the win rate of MCTS variants against the ExpectiMinMax Algorithm, with execution time in $1000l$, where l is the long format. Normal* is a special case of normal, where the goal side is 1.

List of confidence bounds between agent and expectimax		
(w_1,w_2) comparison	$l_\%$	$u_\%$
Normal_5,Hybrid_5	-.02452748	0.1452748
Normal_5,Normal_2	-0.0631188	0.3231188
Normal_2,Hybrid_2	-0.1372976	0.2372976
Hybrid_5,Hybrid_2	0.00405464	0.4194536

TABLE III: This table compares the performance between the different MCTS algorithms, using different long times, where $t \in \text{Agent}_t$ represents the time of computation for the selected MCTS agent, in 1000s.

VI. Discussion

In this section, the results of our experiments will be discussed. In the first part of the experiments, a simulation of 1000 games between the MCTS and Goal-based agent was performed. The results of these experiments are in the graph Figure 6. The MCTS has won 96% of the games and on the other hand, the Goal-Based agent has won the remaining 4%. We can come to the conclusion that the MCTS is much more effective at play than the Goal-Based agent.

In the experiments performed in part B of section 4, various simulations of our MCTS agents against the Expectiminimax agent were performed. The results of these experiments in table II show that increasing the time using the normal MCTS as well as the Hybrid MCTS showed improvement in the value of the standard error. The experiments also show that when using the same time setting the standard error values are lower for the hybrid and therefore it seems that the latter performs better then the normal MCTS algorithm. An experiment with the normal MCTS where the goal side was changed, was also performed. This experiment yields to a lower standard error value this could be due to the advantage one has when starting the game.

In III further analysis of these experiments were made. The upper and lower confidence bound on the real probability were calculated. If $l\% > 0$ we are %-level confident that the player with the higher scoring rate is indeed stronger than other. [17] Which is the case when comparing Hybrid with set time being 5000L against Expectiminimax and Hybrid with set time being 2000L against Expectiminimax and proves that increasing the time makes the agent more efficient. For the other experiments however, nothing can be deducted with the desired confidence. Additionally, it's evident that if as the number of games played and the execution time approach to infinity, then the closer the performance and win rate values of each agent will converge to the real values, by definition of the Central Limit Theorem, depicted in Eq.10.

VII. Conclusion

Finally, it was shown that using a hybrid agent and changing the combination of specific parameters can significantly increase the performance of an agent. Additionally, the Hybrid MCTS Agent doesn't perform as well at low execution times. Therefore, for the Hybrid Agent

to optimally compute the Probability of Dice Chess, there exists a threshold, beyond which it will always slightly outperform a standard MCTS Agent against Adversarial Search. Finally, the combination of machine learning techniques and the implementation of a hybrid agent will yield a more efficient agent that will solve games quicker and obtain a greater win rate than a standard MCTS. A further improvement for this project would be to create a different hybrid with a different Machine Learning algorithm and see what algorithm performs best with MCTS.

REFERENCES

[1] David Levy and Monroe Newborn. How computers play chess. In *All About Chess and Computers*, pages 24–39. Springer, 1982.

[2] Andrew E Soltis. *chess*. June 2021.

[3] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[4] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[5] Jens Golz and Rolf Biesenbach. Implementation of an autonomous chess playing industrial robot. In *2015 16th International Conference on Research and Education in Mechatronics (REM)*, pages 53–56. IEEE, 2015.

[6] Johannes Fürnkranz. Machine learning in computer chess: The next generation. *ICGA Journal*, 19(3):147–161, 1996.

[7] John Gollon. *Chess Variations: Ancient, Regional, and Modern*. Tuttle Publishing, 1974.

[8] Game rules (dice chess).

[9] Game rules (chess).

[10] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[11] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2021.

[12] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search.

[13] Chiara Federica Sironi. *Monte-Carlo Tree Search for Artificial General Intelligence in Games*. PhD thesis, Maastricht University, November 2019.

[14] S. P. Morgan and J. D. Teachman. Logistic regression: Description, examples, and comparisons. *Journal of Marriage and the Family*, 50(4), 1988.

[15] EthanMai. Chess moves. Accessed: 2022-01-08.

[16] Jeffrey S Rosenschein and Gilad Zlotkin. *Rules of encounter: designing conventions for automated negotiation among computers*. MIT press, 1994.

[17] EA Heinz. Self-play experiments in computer chess revisited. *Advances in Computer Games 9, Proceedings, HJ van den Herik and B. Monien (eds.), to be published*, 2000.

APPENDIX

Evaluation Function

For the purpose of this test the relation between the winning probability of the MCTS algorithm (versus goal-base agent) and each component of the evaluation function was observed. The main goal of the experiment is to define which evaluation parts of the function are fundamental. Therefore, one hundred games without each component were simulated. As can be seen on Figure 7, without ”mobility” component the MCTS has lost 9% of games and without the ”king safety” - had a lost rate of 12%. Therefore, it is possible to group the parts of the evaluation function and define the most important elements.

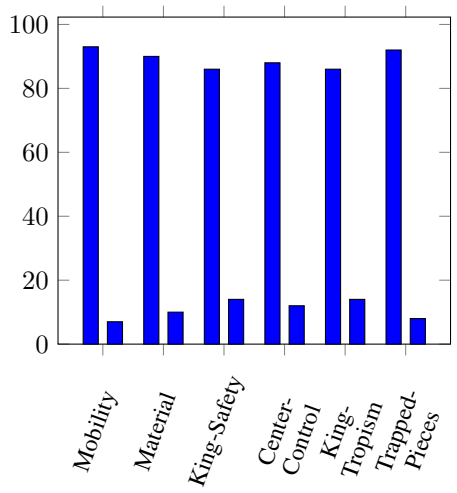

Fig. 7: Separate test of evaluation function I

As Figure 8 represents, the MCTS’s win probability with ”king safety”, ”king tropism” and ”center control” is 81%. Alternatively, without those components the wining probability is slightly more then 71% . Thus, ”king safety”, ”king tropism” and ”center control” have a higher contribution in the move evaluation process and

can be considered fundamental. When testing the relation between the winning probability of the MCTS algorithm (versus goal-base agent) and each component of the evaluation function, it was proven that ”king safety”, ”king tropism” and ”center control” together are the main ones in the evaluation function.

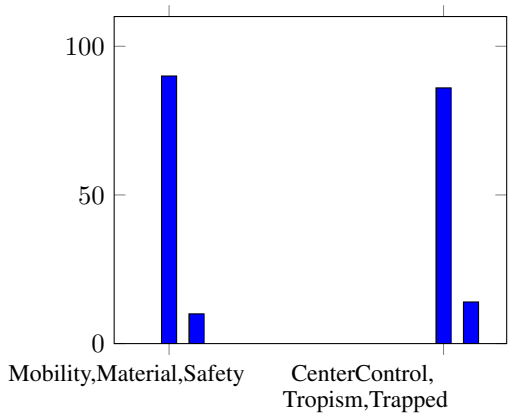

Fig. 8: Separate test of evaluation function II

JUnit

To make sure that our software works well, some JUnit tests were made. These tests were applied to all the pieces and to all the evaluation functions The JUnit tests have helped us to correct those failures that are not very clear and that require paying a little more attention to them.
For the testing of the pieces legal and illegal movements have been proven. Within the legal movements different types of movements have been tested, such as moving a piece to an empty square to which it can move and trying to eat a piece of the other color. On the other hand, for the testing of illegal movements the following one can be highlighted: trying to move a piece to a square that is not inside the board, make an unreachable movement and last but not least try to eat a piece of the same color.
For the evaluation functions, different plays were simulated to check if the score of that specific evaluation function was the correct one. In order to do this, the tests have been divided into three parts. On the one hand, there is the test of the highest possible score. On the other hand, there is the test of the lowest possible score. And finally, there are different plays made with each piece to check that they give the correct score depending on where they are placed on the board.

FORMULAS AND EQUATIONS

$$n \to \infty, \sqrt{n}(\frac{X_n - \mu}{\sigma}) \to N(0,1) \qquad (10)$$