Exploring agents in Dice Chess

Sam Glassman, Bianca Caissotti di Chiusano, Enders Turkers, Sanchi Bhalla, Simonas Bubnys, Mischa Rauch

CONTENTS

Abstract—**Although artificial intelligence agents for Chess have been widely researched and studied; the types of algorithms these agents traditionally use have been selected because they reflect the Chess environment. Specifically, this environment is fully observable, deterministic, competitive, contains two agents, and has a discrete set of possible actions.**

Chess variants generally do not share the same environment as traditional chess. The variant that is the focus of this research, Dice Chess, differs in environment only by being stochastic instead of deterministic. There is limited research and implementation of Adversarial Search (AS) and Machine Learning (ML) algorithms in Dice Chess. Therefore, this report will investigate MiniMax, ExpectiMiniMax, Q-Learning (QL), Monte Carlo Tree Search (MCTS), and a Expecti-QL hybrid agent, and their performance against two baseline agents. Moreover, it dives into what kind of techniques, and parameters are most beneficial for the AI agent's ability to reliably perform best in the stochastic environment of Dice Chess.

The way this study was approached is by basing our agents on those of standard Chess, which then use revised algorithms to account for the stochastic element. Furthermore, we research a model-free agent from the machine learning domain to investigate how it would respond to the Dice Chess environment. All these agents are compared to two control agents, the weaker, random agent, which simply chooses its move at random and the stronger, basic agent, which always captures when it can and moves to a favourable board position otherwise.

We found that, against both baseline agents, the MCTS agent had the highest win rate, and outperformed ExpectiMiniMax against stronger agents. However, MCTS was also found to score the second highest in number of turns taken to win.

This study demonstrates that simple changes in environment otherwise identical games require sometimes complex adjustments model-based agents or entirely new approaches to best preserve its performance.

I. INTRODUCTION

Since its invention, the strategy board game, Chess, has received nearly unrivaled attention and popularity compared to other board games, especially for researchers due to its complexity and variety in terms of strategy and possible board states. As such, since the dawn of the computing era, Chess remains a popular and relevant topic for research into Artificial Intelligence agents, complex search algorithms, and efficient data structures.

Traditionally, Chess agents were implemented with some form of Search Algorithm, based on Adversarial Search; however, due to the sheer amount of possible moves resulting in an extremely large search tree, progress has been historically limited by computing power, memory capacity, and search tree depth to choose a favorable move promptly. It wasn't until 1997 when a Chess agent, Deep Blue, was able to defeat a World Chess Champion for the first time [20].

Since then, progress in computer science, computing technology, and research has improved and successful chess programs have been designed using techniques from the field of Machine Learning, such as neural networks, which have experienced remarkably promising success in comparison to the traditional approaches. For example, DeepMind's AlphaZero was able to achieve a superhuman level of play, defeating world-champion Chess programs such as Stockfish, after only 24 hours of self-training [15].

Due to the popularity of Chess, many variants of the game have been designed and created. One such variant is Dice Chess, which changes the deterministic environment of Chess into a stochastic one, with the introduction of a dice roll. While there exists a wide range of research on regular Chess, the Chess variants often don't experience the same level of academic attention, especially Dice Chess. Since not much research has been made for developing Chess agents for stochastic Chess variants, this report aims to explore approaches to allow Chess agents to perform well in stochastic environments.

The purpose of this report is to present research on how the stochastic nature of Dice Chess affects the ability of both the traditional Chess AI approaches and the more novel approaches from the Machine Learning field, to form coherent, competitive, and successful strategies against human or AI players.

The report will present an adversarial search-based agent design to handle probabilities in its search tree, for which we investigate what types of heuristics are beneficial and how to optimize the search space using pruning techniques, as well as how to handle the stochastic element.

The report will also explore an agent using techniques from the Machine Learning field, using a temporal difference learning approach that utilizes reinforcement learning. A reinforcement approach was considered because for this type of learning, no available data was needed, and a similar approach was implemented successfully for chess.

The main idea of the AI approach presented in this report is to use either a search tree or machine learning algorithm, that can predict and evaluate the value of moves and choose more favorable ones which should generally lead to the AI winning the game. Additionally, prevalent and dominant chess strategies are incorporated in combination with our AI algorithms, to further enhance the success rate of the AI agent The main research question we will investigate is ”To what extent do our AS and ML algorithms perform well against a baseline agent in dice chess?”, followed by three sub-questions:

1) ”How does changing the linear evaluation function affect the AS agent’s performance?”
2) ”Which out of all our agents performs the best in terms of win rate, computing time, consistency and least turns taken per win?”
3) ”How well can MCTS be adapted to the stochastic environment of Dice Chess in comparison to our other agents?”

The report will include the method section which will describe the approaches used to guide the research and fulfill the goal in a mathematical and theoretical way. The implementation section will provide information about the software implementation of these methods and approaches. The experiment section will include an exhaustive description of all the experiments and simulations we have been performed to test our algorithms. The results section will present the results of the conducted experiments concisely and without interpretation. The discussion section will then include the interpretations of the results compared with the corresponding research questions. Finally, the conclusion section will concisely summarize the results, while tying everything back to the main research question.

II. Methods

The different agents developed to play dice chess effectively and their primary evaluation function are described in this section. The choices made regarding the agents or modifications of the agents are also described and motivated.

A. Board State Evaluation Function

The evaluation function is used to evaluate a given state and this number is used by our AI algorithms. The board state evaluation number is used to inform to what degree one state is more or less favourable when compared to another state for a given side. MiniMax, ExpectiMiniMax, Reinforcement Learning and the Neural Network agent use this evaluation function to assess the value of a given state. It is used primarily to evaluate the value of a given state. It’s inputs are the state of the board and the color of the player whose turn it is.

It takes into account three main factors of the state. Firstly, it takes into account the sum of the total pieces on a board for a given side and the piece’s position on the board. Each piece type is assigned a specific value I. Secondly, each piece type has a corresponding matrix of board positions whose weights are influenced by position favor-ability 4. Thirdly, it takes into account the number of isolated and blocked pawns a state contains, as having isolated and blocked pawns is a disadvantageous thing to have in a state. The details of the board state evaluation function are discussed in the implementation section.

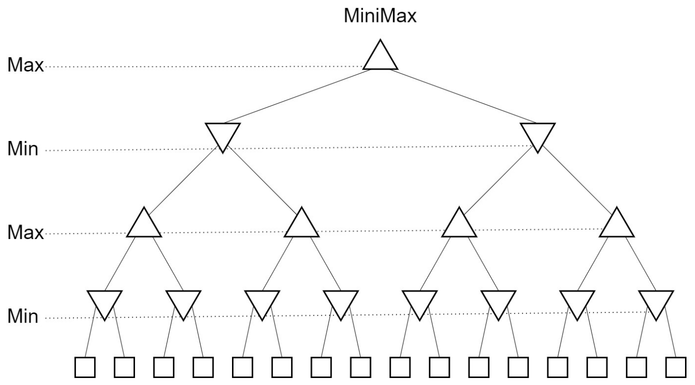


Figure 1. MiniMax Search Tree

B. Random agent

The simplest form of an agent is the agent which performs random moves. This agent receives an ordered list of legal moves for the piece type chosen by the dice roll, and chosen one at random to apply as its turn using a random number generator [12]. This agent represents the baseline control agent on which the other agents are compared too in order to verify that our research decisions have an actual affect on an agents being developed.

C. Basic agent

The basic agent is a step up from the random agent, as it is randomly aggressive. It has two primary upgrades to the random agent. The first upgrade has to do with how the agent performs captures. It is randomly aggressive in the sense that if it has the ability to capture a piece, it captures it. If a move from the list of valid moves involves a capture move, it chooses the first available capture move and executes it. This capture is random because the agent does not differentiate between how valuable a capture is and simply chooses the first possible capture in the array moves. The second upgrade has to do with how the agent moves if there is no capture available. If there is no capture available at that given board state the agent will move to a board location for a given piece that has a favourable location on the board. The favourable board location is determined by the hard coded eight by eight matrix of evaluation integer numbers that each piece type has. The agent chooses the matrix index that will give it the highest value and executes the move accordingly.

D. MiniMax agent

The MiniMax agent uses the MiniMax algorithm to find an optimal move for a given depth (Figure II-D). The MiniMax is an recursive adversarial search algorithm that always assumes that the adversary will choose the optimal move at a given turn (Figure 2). In the context of dice chess, it generates all possible valid moves for a given board state at each layer. Then it backtracks to evaluate each state at each node and chooses the best move at the root.

E. ExpectiMiniMax agent

The ExpectiMiniMax agent uses the ExpectiMiniMax algorithm to find an optimal move for a given depth (Figure 2). It differs from MiniMax as it can account for the stochastic element of dice chess. It contains chance nodes that alternate between the maximising player and the minimizing player (Figure 3). In the context of dice chess, the tree is generated where each node contains a list

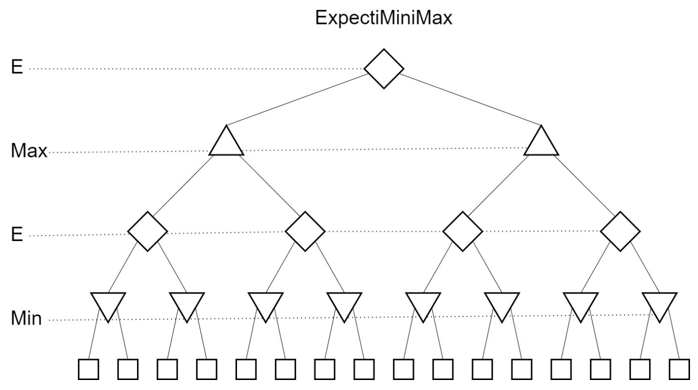


Figure 2. ExpectiMiniMax Search Tree

of possible states for a given piece that can be reached from the best state in the parent node. Then the algorithm backtracks to evaluate each node value by taking the average of the average board evaluation numbers.

F. Reinforcement Learning

A Reinforcement approach was considered to deal with the lack of data in supervised models, the disadvantage with supervised models is that to train the agent initial data is needed where the quantity and quality can make a huge difference towards the success of the agent. Reinforcement learning in short divides into two topics, model-free algorithms and model-based algorithms. Model-free algorithms are much simpler to implement, but it does not focus on building a model (as the name suggests) and instead just mainly focuses on computation of learning good estimates of the values/rewards [9]. Thus, this leads to agent not being able to predict the consequences of the actions, which makes it hard to use planning algorithms. On the other side, model-based algorithms use the model, which can simulate what would happen if we take a sequence of actions and therefore allows us to use planning algorithms, but it is much more complex and hard to implement and may lead to the cost of building a model of the environment being very expensive sometimes. By looking at the pro's and con's of these we chose to try a model-free algorithm. Now, again, model-free algorithms divide into two topics: on-policy algorithms and off-policy algorithms. To summarize it up shortly; on-policy algorithms, as the name suggests, are good to use if we have a good starting policy. Thus, we bypass the option of exploration of the other policies. However, chess requires lots of different policies and during the game the best policy the agent needs may change [**?**]. So, since we need more exploration of policies we need to choose an off-policy algorithm. Finally, this leads us to the Q-learning algorithm. Which is a good algorithm for exploring policies and thus requires no knowledge of the model. It also uses a reward system containing temporal-difference learning, where the agent is tested on its ability to predict its evaluation in the near future [**?**]. Thus, the Q-learning approach was chosen to be evaluated in dice chess. However, the Q-learning algorithm itself requires a big state space which makes it hard to apply to dice chess. This furthermore gives the opportunity to create a deep q learning agent which combines Neural Networks with q-learning and therefore handling the state space problem much better. However, since creating a Feed-forward neural network for DQL (deep Q-learning) using the Eclipse Deeplearning4j library [18] proved to be tough to create in a short time span we had. We decided to create just a normal Q-learning agent with limited depth size. By limiting depth size we do not need to require to create such a large state space.

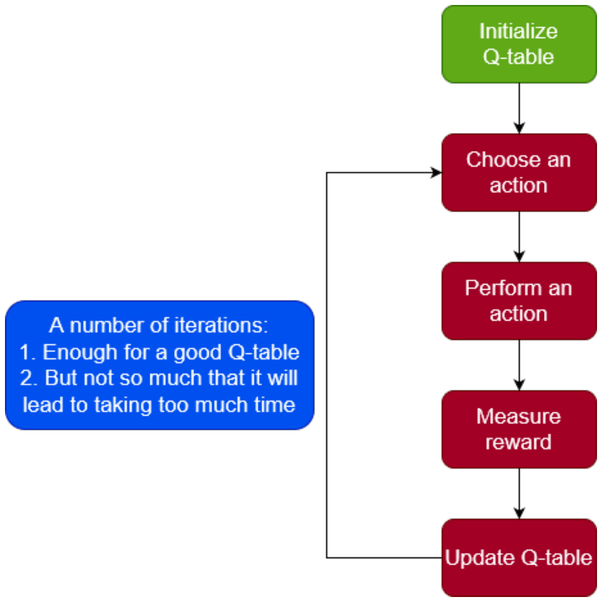


Figure 3. Steps of Q-table([9])

G. Q-Learning agent

Steps of Q-Learning;

1) Initialize Q-table Q(s,a), this table calculates the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state.
2) We will choose an action randomly or will take a learned action.
3) Apply chosen action to the state
4) To learn each value of the state-action pair of the Q-table we will use a Q-Learning algorithm.
5) This algorithm uses the Bellman equation which tells us what new value to use as the q-value for the action taken in the previous state. It takes two inputs: state (s) and action (a).
6) Update the new value, then get back to the second step again.

Bellman equation [16]:

$$Q^{new}(s_t, a_t) = Q^{old}(s_t, a_t) + \alpha \times TD(s_t, a_t) \tag{1}$$

Bellman equation decomposes the value function into two parts, the immediate reward plus the discounted future values. This equation simplifies the computation of the value function, such that rather than summing over multiple time steps, we can find the optimal solution of a complex problem by breaking it down into simpler, recursive sub problems and finding their optimal solutions. [10]

Temporal Difference (TD) method [16]:

$$TD(s_t, a_t) = r_t + \gamma \times maxQ(s_t + 1, a) - Q(s_t, a_t) \tag{2}$$

There were some pseudo-codes for implementation but we mainly got help by looking at python implementations since it was more clear. Then later we wrote a pseudo-code of the algorithm, written with the help of an online journal article [7].

H. Expecti-QL Hybrid Agent

This new hybrid agent is made with combination of ExpectiMiniMax and Q-Learning algorithm. So, an AS algorithm and a ML algorithm. However, before explaining this new agent one question

Table I
PIECE VALUES [22]

Pawn	Knight	Bishop	Rook	Queen	King
100	350	350	535	1000	2000

Figure 4. Point Values for Bishop [23]

-20,	-10,	-10,	-10,	-10,	-10,	-10,	-20,
-10,	0,	0,	0,	0,	0,	0,	-10,
-10,	0,	5,	10,	10,	5,	0,	-10,
-10,	5,	5,	10,	10,	5,	5,	-10,
-10,	0,	10,	10,	10,	10,	0,	-10,
-10,	10,	10,	10,	10,	10,	10,	-10,
-10,	5,	0,	0,	0,	0,	5,	-10,
-20,	-10,	-10,	-10,	-10,	-10,	-10,	-20

must be asked to clearly understand the changes we have made for this hybrid agent.

What is evaluation function? In our case evaluation function is a function where we give a state of the dice chess, and then it returns us a value that defines how good/desirable the given state is. Basically, this is the main focus of this new Hybrid Agent. The basis of the hybrid agent is ExpectiMiniMax and normally for each node of ExpectiMiniMax an evaluation function is ran where for each type of piece a weight is given and depending on pieces' locations it returns a value. However, now, instead of using the this elemantary eval. function we will use Q-Learning.

When we run the Q-Learning by itself, it finishes with a Q-table which has values for each state-action pair. This means we have values for each state of how good each possible action (of the state) is. Therefore, we can just get the average value of each possible actions' of the given initial state and return that value as the heuristic value.

III. IMPLEMENTATION

A. Board Evaluation Function

Our evaluation function is used to evaluate a given state and this board evaluation number is used by our AI algorithms. For the AS algorithms it is always positive for the maximizer and negative for the minimizer. As mentioned earlier it takes into account three main properties.

The first main property takes into account the sum of the total pieces on a board for a given side. Each piece is assigned an integer weight, queen having the highest and pawn having the lowest [22]. We took inspiration from Larry Kaufman's 2012 point values. Additionally, we added a value to the king that would make it easier to ensure it is the target capture for all games. If the color is the maximizing color then for each piece of the same color the evaluation number gets positively updated by the amount that the piece is worth. Then for each piece of the opposite color the evaluation number gets negatively updated by the amount that the piece is worth.

The second main property takes into account the position of a given piece on the board. This is determined by having a matrix of favourable board positions for each piece [23]. The piece color and state is checked and similarly to the way the first property is handled, the evaluation number gets positively updated by the amount that is present at the index of the matrix of a given piece type and color. For the opposite side this evaluation number gets negatively updated.

The second property was improved by tracking the turn of the game and using a different weight matrix for the king for a game turn over a certain value. We chose to use this new board weight matrix after turn 50, however more analysis needs to be done to determine the optimal turn to start introducing this new king board weight matrix. Our evaluation function is called for every generated state for our MiniMax algorithm. There were a number of optimizations that were considered to optimize the evaluation function. Firstly, instead of flipping the black piece matrix horizontally every time you try to apply the white board weights, we hard coded the white board weights to save computational time.

The third property takes into account the number of blocked or isolated pawns . Blocked or isolated pawns are a disadvantage and hence are weighted negatively for the appropriate side [23]. Each blocked or isolated pawn is weighted half the value of the pawn (weight of 50).

B. MiniMax agent

The main structure of the search tree is as follows. The tree consists of nodes that contain relevant variables for our dice chess game. The tree is constructed recursively with a stopping condition of the depth being equal to 0. Then the optimal move is chosen from children of nodes from the root node.

Each node contains a maximizing player boolean, a State object, a Move object, an integer board evaluation number and a list of Node object children.

The maximizing player boolean is true if the Node object is the maximizing one and false if the player is the minimizing player. The State object keeps track of the locations of the pieces on the board with two different representations. It contains a 0x88 board and a unique list of Tuples consisting of the Piece enum and Square enum. It contains an integer dice roll at that given state and a Color enum of which players turn it is to move the given piece. It contains parameters relevant for en passant moves and castling moves. The Move object contains a Piece enum, a Square enum origin and destination and an integer dice roll. The integer board evaluation number is determined at that given state using our agent board evaluation function. The list of Node object children contains all the children of the node that would have the opposite maximizing player boolean value.

The search tree is constructed as follows. Firstly, for a given dice roll a list of all possible board states that can be reached from the current state are generated and added to a list. Similarly, the evaluation numbers are calculated for these states and added to other lists, where both list indices are in parallel. This is done for all 6 piece types. Secondly, we make recursive calls which depend on if the child is a maximizing or minimizing player. We check the child maximizing or minimizing status, then add all the children to the parent Node (which initially is the root), then we return the child with the highest board evaluation number from all the children. Thirdly, we call the recursive MiniMax method again with the best child and with a depth of one smaller. Finally, the stopping condition is reached when the depth is 0 and we return the parent node which is the root. This Node contains a list of children with board evaluation numbers that have been maximized.

The optimal move is chosen by taking the child of the statically returned Node that has the appropriate piece type for the dice roll in the game and has the state with the highest board evaluation number.

There have been some optimizations done to decrease the run time of the MiniMax algorithm such as pruning children that are added.

Children are only added to nodes if they have a more favourable evaluation number for the given adversary (maximizing or minimizing). This speeds up the tree generation process and has no effect on the algorithm as the opponent adversary will always choose the best possible evaluation number of a child.

C. *ExpectiMiniMax agent*

The main structure of the search tree is as follows. The tree consists of nodes that contain relevant variables for our dice chess game. The tree is constructed recursively with a stopping condition of the depth being equal to 0. Contrary to the basic ExpectiMiniMax search tree, chance nodes between maximizing and minimizing nodes are neglected and for the sake of optimization the chance average divider integer value is stored in each node of the layer of the maximizing and minimizing nodes. Each layer contains the same number of nodes as the number of pieces that can move at a given parent node board state. Then the optimal move is chosen from children of nodes from the root node.

Each node contains a maximizing player boolean, a State object, a Move object, a list of Node object children, an integer chance divider, an integer node value, a Piece enum of a piece that moved, a list of board evaluation numbers for a given piece, a list of possible states for a given piece and a list of possible moves for a given state for a given piece.

Similarly to MiniMax the player boolean the maximizing player boolean is true if the Node object is the maximizing one and false if the Node is the minimizing Node. The State and Move objects have been described previously.

Each node contains a list of children nodes. The number of children nodes is equal to the number of pieces that have the possibility to move for a given turn. This is because of a rule in dice chess, where if you roll a piece that you can't move as it is obstructed by friendly pieces you have to roll again.Thus, in our dice chess program we would only provide dice rolls that are valid, meaning that if for example your rook is blocked and can't move you can never roll the dice number that corresponds to a rook.

The integer chance divider is always set equal to the number of children nodes that will be added to the parent node during the downwards tree generation. The integer node value is determined by taking the sum of all evaluation numbers of all the possible generated states for a given piece and dividing it by the chance divider (number of children the parent node will have). It works like a chance node as it potentially represents the best possible board evaluation number that a state could achieve by taking into account that there is a possibility that a different dice roll could potentially yield superior average evaluation numbers when comparing all the children of a parent node for a given state. Each node contains a Piece enum representing the best piece that has moved for the best board state. This is assigned by taking the current best board state (the state that has the highest board evaluation number) and simply taking the last index of the Piece and Square tuple linked-list which takes O(1) time.

Each node contains a list of board evaluations numbers for a given piece that has a parallel index to the list of possible board states for a given piece that is also in the node. Finally, each node contains a list of possible moves given a state for a given piece, which is of size (number of moves for given state times the number of possible board states for a given piece).

The search tree is constructed as follows. Firstly, all possible board states are generated and stored in a list, their respective board evaluation numbers are also evaluated and stored in a separate list with a parallel index. These two lists get added to a tuple whose size corresponds to the number of piece types that can move for a given state and a given color turn. Secondly, the pieces that can move for the current side are computed and stored in a separate list. It is important to note that the piece that can move lists has a parallel index with the possible board states for a given piece and evaluation numbers tuple. Thirdly, there are two main conditions for recursion, one being the child is a minimizing node, the second being the child is a maximizing node. All the children are added to the parent node and the best child for one of these two conditions is returned. Then the ExpectiMiniMax method is called recursively with the best child and a depth of one smaller. The children are added by looping through a list of all states and evaluation numbers for a given piece type (that is at most ever size 6). For each new child node all the node parameters get computed. The best child is identified by identifying a node with the highest integer value given if the parent node is a maximizer or minimizer node.

The optimal move for the ExpectiMiniMax agent is chosen by taking the statically returned best node during the tree generation and then generating valid moves for a given piece (the best piece that moved which is stored in the node) and then using the best state of that node to determine where the piece moved and returning this move.

Other than the overall optimizations we used in our entire program one main optimization for the ExpectiMiniMax agent was the way the chance node was implemented. The chance node integer value was stored in children and computed during the tree generation, which reduced the time taken to generate the tree and saved memory. Without this approach the tree would contain roughly twice and many minimizer nodes and twice as many maximizer nodes as each parent node would require a chance node (except for the root).

D. *Q-learning Agent*

The implementation of Q-learning algorithm consists of 4 simple steps; First, it creates 2 types of Q-table, one where it saves the rewards of each state-action pair, the other saves the information of the state-action pair. Such as whose turn it is, what pieces there are, what is the action and so on... For now we will call them as info Q-table and value Q-table. After creating the Q-tables, the algorithms define the parameters such as: how prone we want the agent to be for exploring, how many episodes, which means a complete game, and how many iterations for each episode, which is the same term for depth in our situation, is it going to have. Now after all this the algorithm will have everything set up, the only thing it will need to do is to run. At each loop, the algorithm will pick either a random action or either it will take the learned path, meaning it will take the action with the highest reward of the current state that had been calculated from the previous episodes. After picking it's action, it will apply that action to the state and with the new state it gets it will give a reward for that action. Finally, with the Bellman equation, which contains the TD algorithm, it will calculate the actual reward value for the current state-action pair it has now.

The algorithm constructs the info Q-table by using linked hash map with the given parameters of current state, side (black or white), and depth. The reason linked hash map has been chosen is that it's operations like get(), put(), contains(), are all done in O(1) [1] just like hash-map, and since it is linked it has insertion order [1]. This insertion order helps the algorithm to keep the indexes of each state-pair to be same in both Q-tables. Thus, when the algorithm tries to access the reward of the state-pair the only thing it needs to do is to find the index of it at info Q-table, then by using the same index it found to value Q-table it gets the pair's corresponding reward.

Now, let's get back to how the algorithm uses its parameters called: current state, side, depth. With these parameters it first constructs the state space, meaning all the possible states we can have from the given initial state, until it reaches the limit of the depth. After that it creates the action space, which is all the legal moves of the given state, of each state. The action space of each state is saved as an array-list of actions. After creating the array-list of actions of each state, we can link each key, which is the state, to a value (an array of actions) in a hash map. In short, it uses Linked-Hash Map where each state is connected to an array of actions. However, instead of saving the action as an object Move, it only saves the origin square and the destination square of the move. This helps the algorithm to run faster and get rid of redundant information the object Move had. Additionally, it has 1 more optimization too, and that is instead of using State object it uses a newly defined object type (which ExpectiMiniMax uses, so QL tries to inherit that object type and replace it with State object) where it just saves the pieces that exist and in which square they exist. Therefore, Q-Learning agent is even further optimized. There is one step-back tough, which is Q-Learning can not perform special moves (en passant, castling, ...). Because to able to do that we would need additional information other than available pieces and it's locations. And to add that extra information for special moves (e.g. is castling still possible?) to the newly defined object type would mean refactoring a big part of the program because ExpectiMiniMax uses that object type too. If instead for QL we had created a new object type that additionally contains information of the these special moves we would have to create lots of new methods compatible with this newly defined object type.

Fortunately, creating the value Q-table is not that hard, it just creates a 2D array with size of state space it had just created, number of rows; and the number 4672 [?] (which is the most possible actions a state in chess can have) , number of columns.

This algorithm has many parameters. So each parameter will be explained one by one;

1) Episode: this parameter defines the number of complete games the algorithm will play at each turn.
2) Iteration: In our case this defines our depth.
3) Exploration probability: This defines the probability that the agent will explore (taking a random action)
4) Exploration Decay: Since at first we don't have any policy, meaning we don't have any q-value. We are forced to pick a random action, but after each episode the agent gains more knowledge and we want to start to slowly use this knowledge. Thus, what we do at the end of each episode is; exploration probability = exploration probability- exploration decay. This helps the algorithm to slowly start to use the knowledge it gains.
5) Minimum exploration probability: However, we don't want the agent to fully use only it's knowledge after a while. So, this parameter defines a lower bound to exploration probability. This way there will be always, the chance of exploration for the agent.
6) Discount factor: This helps to tell the algorithm how much to strive for a long-term high reward. It is set between 0 and 1.
7) Learning rate: This parameter defines the learning rate of the agent. This is also set between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly. [17]

There are only 2 options for choosing an action. And to choose which one to do it, it only generates a random number between 0 and 1, then checks whether that number is below the exploration probability or above. If it's below it picks a totally random action, else picks the action with the highest reward of the current state it is at. Here you can observe how we increase the probability of picking a learned path. At each loop, the probability of the chosen number to be above the exploration probability will be higher.

Here it mostly uses the same reward system that is used in the MiniMax algorithm. But there is a slight variation. The reward values of the pieces and the state itself is calculated separately. Thus, this gives us the possibility of defining the style the agent is going to apply. For example, if we want it to play more aggressively we can increase the reward of the pieces by just multiplying it with 10 and decreasing the reward value of the state by dividing it with 10 or vice versa. There is also the case of not being able to get the immediate reward after applying a move to the state, because the value of the total reward will be completely dependent on how the opponent player will play, but we need to play several games inside the loop at each turn. What algorithm does here is that it uses another agent that has been created previously, such as random agent, basic AI agent, and so on. Then after choosing the agent it is going to use, it applies the move that agent created and with the new state it gets it will get an immediate reward. The better algorithm it uses, the better results. Because the immediate reward will be more accurate, e.g. a random agent may not capture the king but a better agent like ExpectiMiniMax will capture the king, so the reward that action gets will be much lower, thus more accurate. Beware that this situation also leads to QL agent only being able to get depth of even numbers, because at each iteration it will apply QL agent's move, then chosen agent's move and it will continue with that state (so it gets' back to QL agents turn/side), that the moves has been applied, to the next iteration.

However, this reward value it got for the current state-action pair is not final yet. With the reward value it gets it puts into the Bellman equation, it also uses the parameters it had defined in the beginning. After getting the final reward value it finally updates the value of the current state-action pair it was at. The only thing left to do is to check whether the current state has reached an end, meaning checking whether the king of either side has been captured, if not, it keeps going through the loop until it reaches the depth limit.

E. Expecti-QL Hybrid Agent

To able to use the methods of ExpectiMiniMax without creating a new class which contains the same methods, we decided to only pass an additional boolean parameter to ExpectiMiniMax agent. Where "true" would mean using QL agent as the evaluation function, and "false" would mean use the elementary evaluation function we have.

After that situation is out of the way, we added new methods for the hybrid for a new evaluation function in a way where it is compatible with using Expecti and QL agents at the same time. So, this part is mainly about trying to transform from one object type to the other between the methods without getting any errors.

In the end, we also added one additional method to QL agent. Where it returns us the average value of all the possible actions the given state has. After all this implementations we were able to get a new hybrid agent with the combination of Expecti and Ql agent.

F. Program Optimization

We have made multiple optimizations to help decrease the total run time of our program and the time taken to compute optimal moves for the AI agents. We used enums as they are less computationally expensive than classes, we used lists of Piece and Square enum tuples and we had separate threads running during AI game simulations. Lists of Piece enum and Square enum tuples are used to track where certain pieces are on the board, which decreases the run-time of our program. This is because at most we have to loop through 32 pieces when generating possible board states, instead of all 64 pieces if we used a traditional eight by eight matrix approach. Additionally, when tracking the state of the game the piece that moved is always added in the last index, this way we can always access it in O(1) time using a Linked-List. This was helpful when generating the tree of Nodes for both MiniMax and ExpectiMiniMax as we could easily update the Move object in the Nodes.

A separate thread for the adversarial search tree algorithms (MiniMax and ExpectiMiniMax) was also created to trim off actual run time for simulations and data collection.

When running two AS AI's against each other every time the Game object called the makeMove method a thread would start computing for an AI with a certain color, at the same time another AI would start computing its move on a separate thread. The main game thread would wait for the method in the first AI thread to complete before continuing execution. If this first thread took a long time by the time it's finished the second AI thread could have already been done computing (if the second thread was computing the move with an AI of smaller depth) which could execute the code straight away.

It is important to note that the MiniMax and ExpectiMiniMax AI's don't need to know where the opponent moved before starting computations because they always assume the adversary will make the best move possible (as these algorithms are maximising the minimum gain or minimizing the maximum gain).

When running one AS AI against another AI (such as QL) the AS would compute on a separate thread while the QL AI would compute on the main thread. Before the other AI would start its computation the AS AI thread would join with main and pass the updated move, the state would update and the other AI can begin its computation again.

G. Data logging

The results of our experiments were recorded on CSV files which have, then, been further processed. These store information of numerous AI vs Human, and AI vs AI Games. For both game types, the winner and the number of turns needed to win are recorded, furthermore for the AI vs AI games the file also keeps track of the AI algorithms used. Additionally to gather further insights into the behavior of the implemented agents the average time needed to calculate a move is stored, for both used algorithms, as well as the total game time. Lastly the board evaluation score is tracked and the final piece array of available pieces on the board for each final state of the game is being tracked. The above described data is stored at the end of each game, to get insights into the full game a second CSV file gets stored which keeps track of each move in a AI vs AI game. This stores the time needed for that move, the captured piece (if there was one), the remaining pieces on the board, and the board evaluation score.

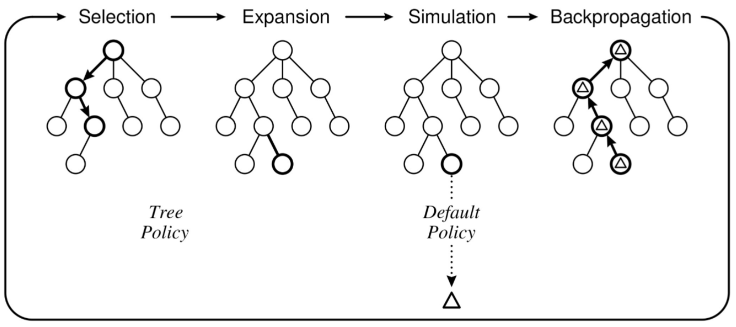


Figure 5. One iteration through the phases of the general MCTS process

IV. Monte Carlo Tree Search

A. The General Approach

Monte Carlo Tree Search (MCTS) is distinguished from other game tree approaches from the Adversarial Search domain; in that instead of a brute-force search up to a defined depth limit, it uses a process of repeated samplings of the search space in the form of random or pseudo-random simulations in order to collect statistics on the utility value of any given action or state. These statistics are propagated through the game tree such that MCTS can make more informed choices in the next iteration of its search. Each iteration traverses the current game tree using a best first search strategy tailored to balance exploitation of statistically high-valued actions, with exploration of less promising actions for which there aren't enough samples to make an educated assessment of their value. The exploitation vs exploration trade-off aims to build an asymmetrical search tree, such that each subsequent iteration can make more statistically informed choices on which actions are the most promising.

Whereas brute-force game trees rely on strongly crafted heuristic evaluation functions, which are often expensive computations requiring detailed domain knowledge of the game in order to determine the value of its leaf nodes; the MCTS technique does not have this limitation because the value of its nodes are based on the utility value of the terminal state for each of its simulations, thereby replacing heuristics with statistics such that MCTS at its base only requires the rules of the game and the utility function of terminal states.

The general outline of the Monte Carlo Tree Search approach is as follows:

Algorithm 1 Monte Carlo Tree Search

```
1: function MCTS($state_0$)
2:     $root \leftarrow Node(s_0)$
3:     while within computational budget do          ▷ e.g.: time limit
4:         $leaf \leftarrow SelectionPolicy(root)$
5:         $Node(q) \leftarrow Expand(leaf)$
6:         $\Delta \leftarrow SimulationPolicy(q)$
7:         $Backup(q, \Delta)$
       return $action \leftarrow BestChild(root)$
```

MCTS has seen recent success in games with high branching factor, such as Go, where a heuristic evaluation function is hard to design; whereas the traditional AS approaches with pruning and strong evaluation functions have remained dominant for well researched games like Chess. A factor for this is MCTS using long term statistical analysis to inform its move selection, which makes it vulnerable trap states, where a specific sequence of actions can lead to defeat that brute force algorithms are easily able to detect, whereas MCTS may fail to recognize and avoid them.

B. *Expecti-MCTS*

Dice chess is distinct from Chess due to its stochastic nature, and as such is less prone to trap states since specific sequences of moves simply aren't probable. Additionally, Dice Chess has a larger branching factor due to the chance events, so we considered the stochastic statistical sampling property of MCTS to be an appropriate approach in creating a consistently winning agent.

In order to adapt standard MCTS to Dice Chess, chance nodes are introduced into the game tree in a similar fashion to Expectimax algorithms, where the chance nodes model the possible dice rolls, given its state.

In the Expect-MCTS algorithm, decision nodes represent $(state, roll)$ tuples, where the roll determines the legal actions from that decision node. The children of decision nodes are chance nodes where the action chosen is applied to the decision node state. Therefore, chance nodes represent $(state', action)$ tuples where $state'$ is the state resulted by applying the parent node's $action$. The children of chance nodes are in turn decision nodes with $state'$ and the additional information of $roll$, given randomly by the chance node.

The Selection, Expansion, and Back-propagation methods were also adapted from the standard implementation to take the chance nodes into account.

During selection, when a fully expanded chance node is encountered, the selection algorithm randomly chooses one of its children to continue the selection process. Alternately, if a decision node is selected for expansion, then the expanded child is a chance node and is immediately further expanded such that the selection policy only ever returns decision nodes for the simulation phase. This is because simulation can only occur from decision nodes which have an assigned dice roll that determine the available actions. Chance nodes don't have available actions to choose from since they only represent dice rolls.

The simulation phase of Expect-MCTS is largely the same as in standard MCTS, except the alternating players must first "roll the dice" such that they can apply an action.

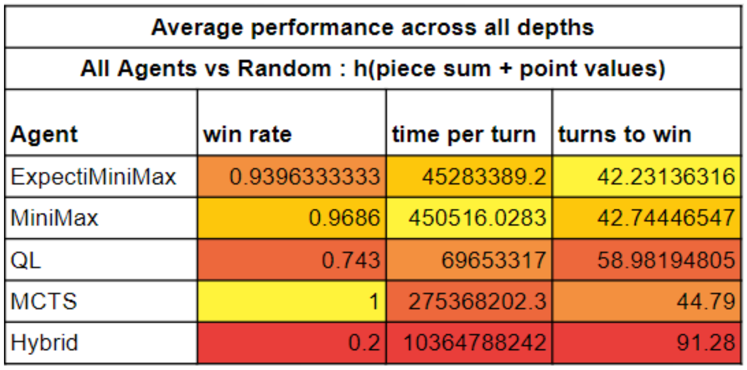
In the back-propagation phase, in addition to the visit count of each node along the path being incremented; only chance nodes have their win values adjusted according to Δ. This is due to the fact that decision nodes select their actions based on the statistics of their chance node children, whereas during selection, a child of chance nodes is randomly selected regardless of the child's value.

V. Experiments

Firstly, we compared how different agents performed against the baseline agents, the random agent and the basic agent. We ran 100 games with our agent being white and the baseline being black and 100 games vice versa. Then we adjusted for the 54 percent win rate for white, and compared all our agents. We compared how the win rate, average time to compute a move and average turns taken to win where affected by this change in our linear evaluation function.

Secondly, we compared how including blocked and doubled pawns in the evaluation function in addition to the sum of pieces and their point values, would affect the performance of our AS algorithms. The first variant of our evaluation function which only used piece sums and point values as our evaluation function. The second variant of our evaluation function also accounted for blocked or isolated pawns in addition to using piece sums and point values. Similarly like the initial experiment, we compared how the win rate, average time to compute a move and average turns taken to win where affected by this change in our linear evaluation function.

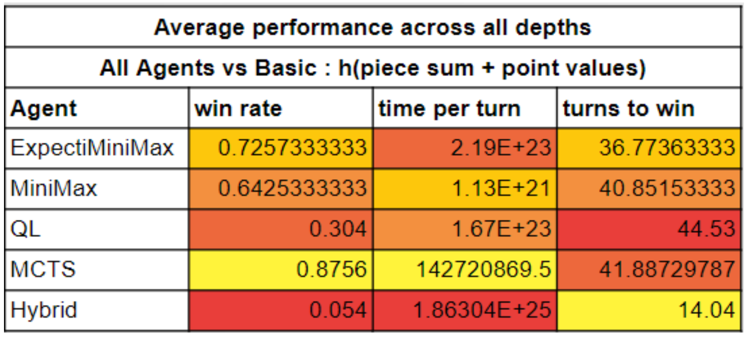
Additionally, for the Expect-MCTS agent, the exploration constant C, which determines how the children of decision nodes are selected by the UCT equation, had to be tuned to best fit Dice Chess. Starting with the game theoretic value of $sqrt_2$, we ran several hundred games against the other agents for different values of C in order to determine which value led to the least amount of average losses.


Average performance across all depths

All Agents vs Random : h(piece sum + point values)

Agent	win rate	time per turn	turns to win
ExpectiMiniMax	0.9396333333	45283389.2	42.23136316
MiniMax	0.9686	450516.0283	42.74446547
QL	0.743	69653317	58.98194805
MCTS	1	275368202.3	44.79
Hybrid	0.2	10364788242	91.28

Figure 6. All agents vs random agent


Average performance across all depths

All Agents vs Basic : h(piece sum + point values)

Agent	win rate	time per turn	turns to win
ExpectiMiniMax	0.7257333333	2.19E+23	36.77363333
MiniMax	0.6425333333	1.13E+21	40.85153333
QL	0.304	1.67E+23	44.53
MCTS	0.8756	142720869.5	41.88729787
Hybrid	0.054	1.86304E+25	14.04

Figure 7. All agents vs basic agent

VI. Results

For our first experiment while running our agents against the random (weaker) baseline agent the MCTS agent had the highest win rate against the random agent with a win rate of 100 percent (Figure 15). The MiniMax agent had the lowest time to compute moves with a average time to compute a move of 0.00045 seconds (Figure 17). The ExpectiMiniMax agent had the lowest average turns to win with an average turns to win of 42.2 turns (Figure 19). The results condensed to a table can be seen in Figure 6. It shows the performance of our agents after running 200 games, 100 for the agents being white, 100 for the agents being black and then adjusting for the 54 percent win rate bias of the white starting side in chess. Additionally it shows the three main things we investigated such as the average win rate, the average time taken to compute a move per turn and the number of average turns taken to win. It is important to note that this data was collected and averaged across all 6 depths (1,3,5,7,9,11) for our AS algorithms, at a depth of 2 for QL, 2000 iterations for MCTS and a depth of 2 for the Hybrid agent.

While running our agents against the basic (stronger) baseline agent the MCTS agent again had the highest win rate against the random agent with a win rate of 87.6 percent (Figure 16). The MCTS agent had the lowest time to compute moves with a average time to compute a move of 0.14272 seconds (Figure 18). The Hybrid agent surprisingly had the lowest average turns to win with an average turns to win of 14.0 turns (Figure 20). The results condensed to a table can be seen in Figure 7. The same exact procedure was applied to it as for Figure 6.

When comparing the win rate of our agents against the random agent, both MiniMax, ExpectiMiniMax and MCTS performed similarly well being all within a range of 1.6 percent (Figure 15). This shows that all these algorithms perform quite well against an easy baseline agent. It is important to note that the MiniMax agent which is meant to excel in deterministic environments performed almost as well as ExpectiMiniMax. When comparing the win rate of our agents against the basic agent, ExpectiMiniMax performed better than MiniMax by 3.2 percent and MCTS performed better than ExpectiMiniMax by 5.8 percent 16). This shows clearly that MCTS outperformed ExpectiMiniMax against stronger agents.

For our second experiment we saw that adding blocked and doubled pawns to our linear evaluation function in addition to the piece sums and point values had a rather significant affect on the win rate, average time to compute moves and the average turns taken to win.

Firstly, to evaluate the performance of the second evaluation function variant we compared how it affects the average win rate of our AS agents. When comparing both of these evaluation function variants we saw that the first variant caused ExpectiMiniMax to have a higher average win rate for depths of 1, 5, 7, 9 and 11 (Figure 9). The second variant only improved ExpectiMiniMax's win rate for a depth of 3. The first variant caused MiniMax to have a higher average win rate for depths of 3, 5 and 9. The second variant improved MiniMax's average win rate for depths 1, 7 and 11 (Figure 10).

The second variant evaluation function had a slightly higher win rate overall than the first variant.

Secondly, to further evaluate the performance of the second evaluation function variant we compared how it affects the average time to compute a move for our AS agents. For ExpectiMiniMax, we saw that for depths 1,3,5 the time taken to compute a move was relatively the same being less than 0.1 seconds (Figure 11). After a depth of 5 the time taken to compute moves grew exponentially for the second variant taking around 7.83 seconds to compute a move for a depth of 11 (Figure 11). For MiniMax, we saw that for all depths it took on average 0.0005 seconds to compute a move (Figure 12). For the second variant it took 0.00416 seconds to compute a move for a depth of 1 and 0.0297 seconds for a depth of 11 and the increase was quite linear (Figure 12).

The first variant evaluation function could compute moves much quicker overall than the second variant for both ExpectiMiniMax and MiniMax.

Thirdly, to further evaluate the performance of the second evaluation function variant we compared how it affects the average turns taken to win a game, or how fast an agent can win in terms of turns to our AS agents. For ExpectiMiniMax, for the first variant, for depths 5, 7 and 11 the turns taken to win a game were lower than for the second variant (Figure 13). For the second variant, for depths 1, 3 and 9 the turns to win were lower than that of the first variant (Figure 13). For MiniMax, for the first variant, for depths 1, 5, 7 the turns to win were lower than that of the second variant (Figure 14). For the second variant, for depths 3,9,11 the turns to win were lower than that of the first variant (Figure 14). There was no clear difference in the turns taken to win for both the first and second variant of the evaluation function.

Additionally, we can see that for the first variant of the evaluation function ExectiMiniMax had the highest win rate at a depth of 9 (Figure 9) and won in the fewest amount of turns at a depth of 7 (Figure 13). MiniMax followed a similar trend where it had the highest win rate at a depth of 9 (Figure 10) and won in the fewest turns at a depth of 1 (Figure 14).

From the experiments conducted to tune the exploration parameter, we found 0.8 to be the global optimum against all the implemented agents. However, against specific agents smaller or larger values of C resulted in a higher average win-rate.

VII. Discussion

From our first experiment, overall, against both baseline agents, the MCTS agent had the highest win rate. Against the basic agent it would take the least time per turn. However, the MCTS agent had for both baseline agents, the second highest turns taken to win, meaning whenever it won, it would win in more turns than three of our other agents. This is due to MCTS informing its move choice by long term aggregated statistics of simulated games, rather than the locally optimal move in contrast to traditional AS agents. This resulted in it playing more conservatively due to low certainty in the value of its available moves. However, towards the end-phase of the game when there are less pieces, less possible moves, and less possible dice rolls; MCTS would have a much higher degree of confidence in its move selection due to the reduced search space allowing the MCTS process to collect more accurate statistics in its search tree. Therefore MCTS would often let the opening and mid-game phase drag on, relying on stochastic of dice chess to prune the search space until it can gain more accurate and confident estimates in its move selection.

The ExpectiMiniMax agent had a higher win rate against the MiniMax agent against the stronger baseline agent (basic agent) as expected (Figure 16). This makes sense because it always assumes the adversary will play optimally in a stochastic environment and the basic agent played more optimally than the random agent.

The ExpectiMiniMax had a win rate of 0.8 percent smaller than MiniMax which was unexpected (Figure 15). This can be explained by the number of games that we ran, as we only ran 200 games which wasn't enough for the win rate to converge. Additionally, the random agent and basic agent are fundamentally different as the random agents chooses purely random moves while the basic agent chooses more intelligent (a little more optimal) moves. As both AS algorithms assume the opponent will make the optimal move, and ExpectiMiniMax will further make a more optimal move in our stochastic environment (while MiniMax assumes this in a deterministic environment), this explains why ExpectiMiniMax in particular was under-performing when compared to the MiniMax agent.

From our second experiment, adding blocked or isolated pawns to our evaluation function in addition to the piece sums and point values (variant 2), gives the AS agents a slightly higher win rate for certain depths, at the cost of a nearly exponential (for ExpectiMiniMax) or a nearly linear (for MiniMax) increase in the time taken to compute moves at a higher depths with no clear effect on the speed with respect to the number of turns taken to win a game. As mentioned previously, the increase in win rate was small for the second variant and only increased the win rate for ExpectiMiniMax for a depth of 3 and for MiniMax for depths 1,7 and 11. It was unexpected however that the second variant would perform worse. This could be due to the fact that isolated/blocked pawns had a very small impact on the overall evaluation number for both algorithms as they where weighted half that of a pawn. Additionally, we would see this heuristic have a higher impact if we ran thousands more games at every depth, however due to our implementation and computational resources this was not feasible. The Java implementation of our second variant evaluation function could have been improved as different data structures could have been used to achieve at least linear time for both algorithms, hence allowing us to run more tests at higher depths.

Overall, the ExpectiMiniMax and MiniMax agents had the highest win rate at a depth of 9 using the first variant of the board evaluation function. It is important to note that the depth of 9 was the second highest depth we tested. This was surprising as we expected a higher depth to yield a higher average win rate. This could have been solved by running more experiments at much higher depths. Additionally, more investigation into using different programming languages like C that are more efficient at running algorithms especially with our 0x88 board setup would be needed to be able to run algorithms quickly at very high depths.

QL agent had an average performance across the three main criteria we measured against. This makes sense because QL is a model-free and off-policy algorithm. This means that QL agent is designed to able to perform well enough at any given kind of environment, not to perform really well. That is why QL agent can sometimes be a good option for unknown environments. But of course as can be seen from the experiments it performs worse than half of the agents we have such as MCTS, ExpectiMiniMax and MiniMax. Perhaps the only way for the QL agent to outperform the other agents would be to further optimize so that we can explore in higher depths (after all the current optimizations that we have done it could only go as far as depth 2). However, there also could have been an option of trying to combine the QL agent with an agent much more successful than itself such as ExpectiMiniMax.

This brings us to Hybrid Ex ecti-QL agent. But interestingly this new hybrid agent performed much worse, even worse then QL agent. This could be due to not being able to increase the depth as much as we wanted. Efforts were made to create other agents such as a MCTS-NN (neural network) agent but it proved to be too difficult to accomplish in this short time-span we had and perhaps this would under-perform, or outperform all the agents we had.

To further improve the state of the art, we tried to implement a Neural Network which is based on supervised learning. For this we made use of the Keras library in python [13] and further did the inference in java using tensorflow java library [**?**] The Data that we need the Neural Network to train on is collected from every game of the MCTS agent playing as white. The data consists of the probabilities of the MCTS winning against the opponent agent based on the pieces on every square on the board which are assigned values ranging from -6 to 6.These piece values are the features of the dataset. The goal of the Neural Network is to predict the probability of MCTS to win a new game based on the pieces on the board. The dataset is then converted into a data frame using the pandas import and split it into train and test data for labels Y and features X using the train test split command imported from sklearn. Next, we scale the train and test data for features X using the Minmax Scaler imported from sklearn so that values are within the range from 0 to 1. We create a simple feed forward network using the sequential method which is imported from Keras and add the layers sequentially to it. There is a total of three layers in the network. The input layer has 70 neurons which is equal to the number of features in our dataset. The activation function it uses is relu. There is only one hidden layer which has 35 neurons and the activation function it uses is also relu. There is one output layer which has only 1 neuron which outputs the probability of MCTS winning the game based on the input features making use of the sigmoid activation. The network is trained on the training data for the features X and labels Y. We take 400 epochs and a batch size of 1382 which is equal to the training examples in our dataset .The model is compiled in the end and makes use of the adam optimizer and mean squared error to calculate the loss. The network is then evaluated on the training and test data. The loss on the training data is then compared to the validation loss on the test data. The network predicts the values on the training data X. The network is saved in python and then is exported to java using the SavedModelBundle of the Tensorflow java library .For the inference , we create a NdArray of type float and size 1,70 and pass the scaled input [2] into the NdArray. A tensor object is created of the same type and size. The session().runner() creates a runtime of the session. The tensor object is then fed to the network using the feed method and the output is returned using the fetch method in the runtime.

Due to the lack of existing research on agents for Dice Chess, we can conclude from our experiments that the MCTS agent is the current best state of the art agent for Dice Chess.

VIII. Conclusion

In conclusion, against both baseline agents, the MCTS agent had the highest win rate. Against the basic agent it would take the least time per turn. However, the MCTS agent had for both baseline agents, the second highest turns taken to win, meaning whenever it won it would win in more turns than three of our other agents.

Adding blocked or isolated pawns to our evaluation function in addition to the piece sums and point values (variant 2), gives the AS agents a slightly higher overall win rate for certain depths, at the cost of a nearly exponential (for ExpectiMiniMax) or a nearly linear (for MiniMax) increase in the time taken to compute moves at a higher depths with no clear effect on the speed with respect to the number of turns taken to win a game.

ExpectiMiniMax and MiniMax agents had the highest win rate at a depth of 9 using the first variant of the board evaluation function.

MCTS can clearly be adapted to the stochastic environment of dice chess very effectively.

References

[1] Time complexity of java. 12/05/2021.

[2] Jason Brownlee. How to use standardscaler and minmaxscaler transforms in python, June 10,2020.

[3] DBepdia. Expectiminimax, 2021.

[4] Tim Eden, Anthony Knittel, and Raphael van Uffelen. Reinforcement learning - algorithms.

[5] Herrmann and Michael. On-policy and off-policy algorithms, 27/01/2015.

[6] JavaPoint. Mini-max algorithm in artificial intelligence, 2020.

[7] Huang (Steeve) Kung-Hsiang. Introduction to various reinforcement learning algorithms. part i (q-learning, sarsa, dqn, ddpg), 2018.

[8] Lai and Matthew. Giraffe: Using deep reinforcement learning to play chess, 2015.

[9] Mitchell, Tom, Hill, and McGraw. Machine learning, 1997.

[10] Nguyen and Hai. How should i model all available actions of a chess game in deep q-learning? 28/06/2018.

[11] Ortega and Pedro A. Expectiminimax tree, 2014.

[12] Paquier and Micaël. Implementing a chess engine from scratch, 2020.

[13] Jose Portilla Pierian Data Inc. python for data science and machine learning, 2020.

[14] Shyalika and Chathurangi. A beginners guide to q-learning. 15/11/2019.

[15] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis1. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[16] Singh and Chaitanya. Linkedhashmap in java. 07-09-2014.

[17] Soper and Dr. Daniel. Foundations of q-learning. 23/4/2020.
[18] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the jvm, apache software foundation license 2.0.
[19] Torres and Jordi. The bellman equation. 11/07/2020.
[20] Standford University. Deepblue, Fall, 2020.
[21] TensorFlow Core v2.7.0. org.tensorflow.
[22] Chess Programming Wiki. Point value, 2021.
[23] Chess Programming Wiki. Simplified evaluation function, 2021.
[24] Wikipedia. Minimax, 5 October 2021.
[25] Wikipedia. Expectiminimax, April 21, 2020.

IX. Appendix

Algorithm 2 MiniMax algorithm

```
function MINIMAX($node, depth, maximizingPlayer$)
    if $depth = 0$ or $node\ is\ a\ terminal\ node$ then
        return $the\ heuristic\ val\ of\ node$
    if $maximizingPlayer$ then                    ▷ (maximizing player)
        $val \leftarrow -\infty$
        for each $child\ of\ node$ do
            $val \leftarrow$ MAX($val, minimax(child, depth - 1, FALSE)$)
        return $val$
    else                                          ▷ (minimizing player)
        $val \leftarrow +\infty$
        for each $child\ of\ node$ do
            $val \leftarrow$ MIN($val, minimax(child, depth - 1, TRUE)$)
        return $val$
```

Algorithm 3 ExpectiMiniMax algorithm

```
function EXPECTIMINIMAX($node, depth, maximizingPlayer$)
    if $depth = 0$ or $node\ is\ a\ terminal\ node$ then
        return $the\ heuristic\ val\ of\ node$
    if $we\ are\ to\ play\ at\ node$ then
        $val \leftarrow -\infty$
        for each $child\ of\ node$ do
            $val \leftarrow$ MAX($val, expectiminimax(child, depth-1)$)
        return $val$
    else if $the\ adversary\ is\ to\ play\ at\ node$ then
        $val \leftarrow +\infty$
        for each $child\ of\ node$ do
            $val \leftarrow$ MIN($val, expectiminimax(child, depth-1)$)
        return $val$
    else if $random\ event\ at\ node$ then
        $val \leftarrow 0$
        for each $child\ of\ node$ do
            $val \leftarrow val +$ P( $[child] \times expectiminimax(child, depth - 1)$)
    return $val$
```

Algorithm 4 Q-Learning algorithm (altered for Dice chess)

```
function Q-LEARNING($Depth, RewardFunction, TransitionFunction, LearningRate, DiscountingFactor, explorationDecay$)
    Initialize $new\ empty\ States(S) \times Action(A)\ Qtable$
    for defined $number\ of\ iterations$ do
        $s \in S$                                  ▷ (starting at given initial state)
        for given $depth\ number\ /\ becomes\ terminal$ do
            $calculate\ \pi\ according\ to\ exploration\ strategy\ and\ Qtable$   ▷ (either by random or using argmax function)
            $a \leftarrow \pi(s)$
            $s' \leftarrow T(s, a)$
            $s'' \leftarrow T(s', a)$       ▷ (additional action is performed here (random) to get back to AI's turn)
            $r \leftarrow R(s'', a)$                ▷ (getting new reward)
            $Q(s'', a) \leftarrow (1 - \alpha)\times\ Q(s', a) + \alpha \times (r + \gamma \times max'_a(Q(s'', a')))$
            $s \leftarrow s''$
        if $current\ exploration\ prob.(Cprob) >= min.\ exploration\ prob.$ then
            $Cprob \leftarrow Cprob - explorationDecay$

            return $final\ Qtable$
```

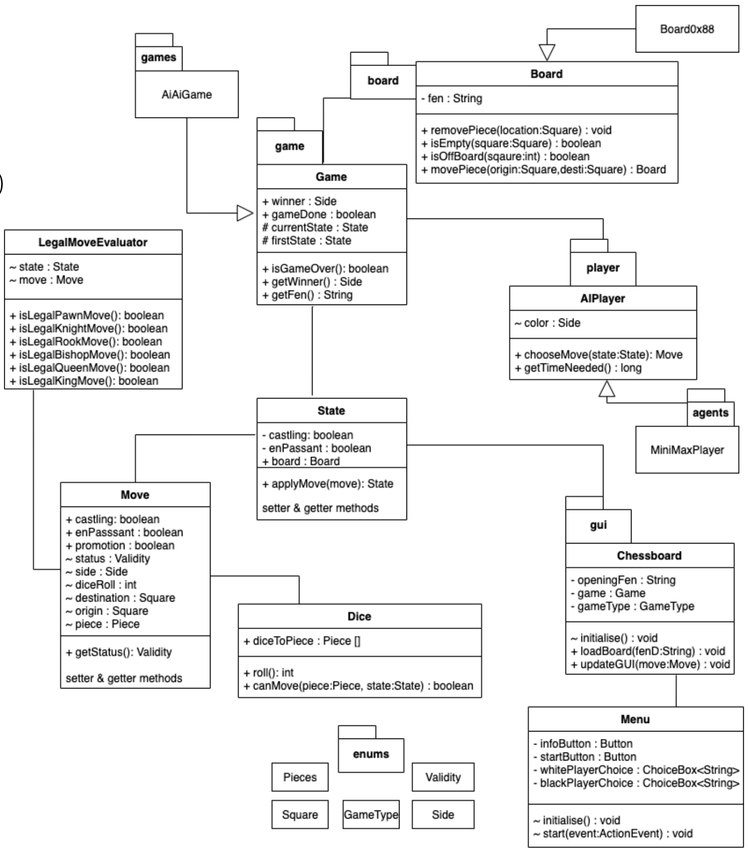


Figure 8. Software UML Diagram

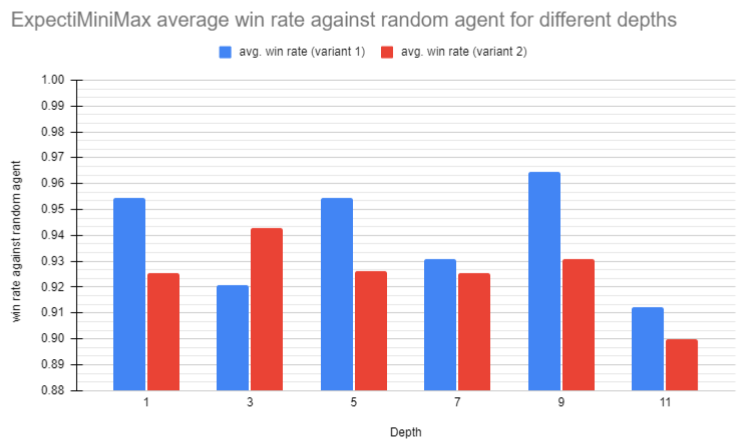


Figure 9. ExpectiMiniMax average win rate against random agent for different depths

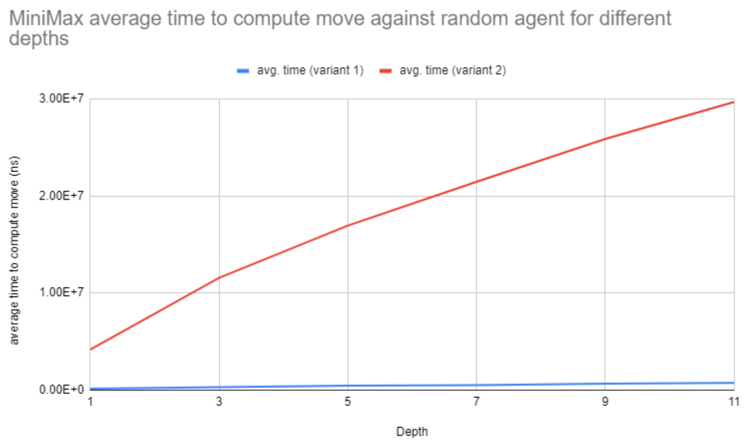


Figure 12. MiniMax average time to compute move against random agent for different depths

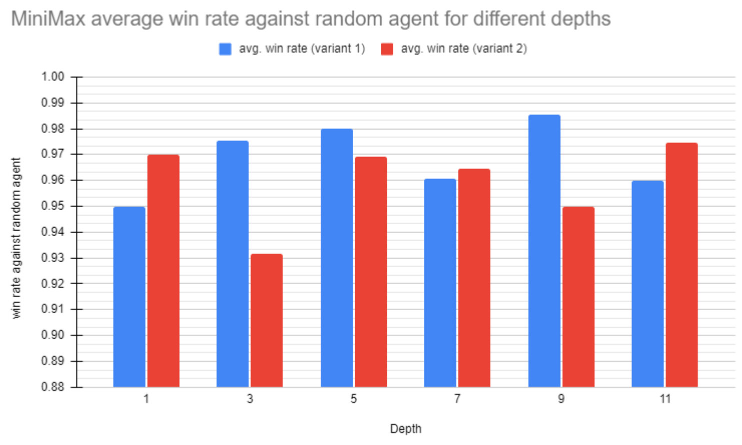


Figure 10. MiniMax average win rate against random agent for different depths

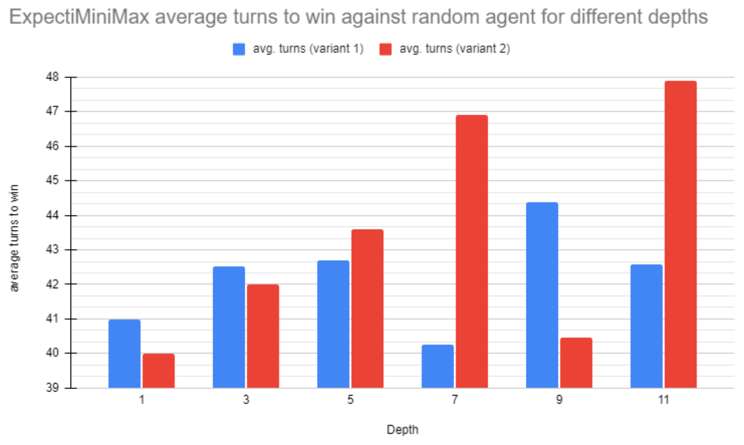


Figure 13. ExpectiMiniMax average turns to win against random agent for different depths

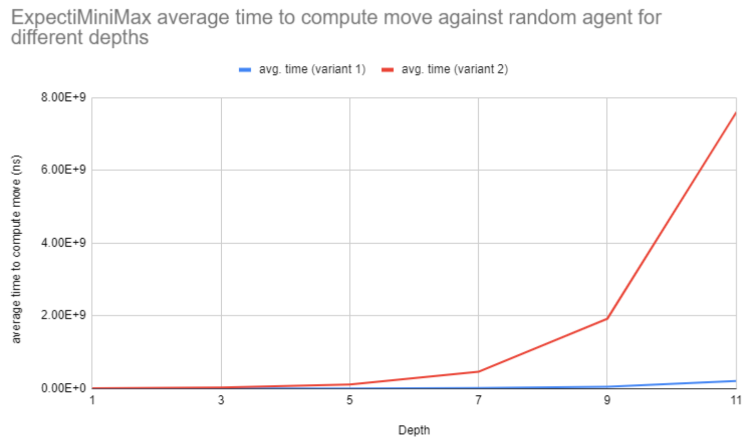


Figure 11. ExpectiMiniMax average time to compute move against random agent for different depths

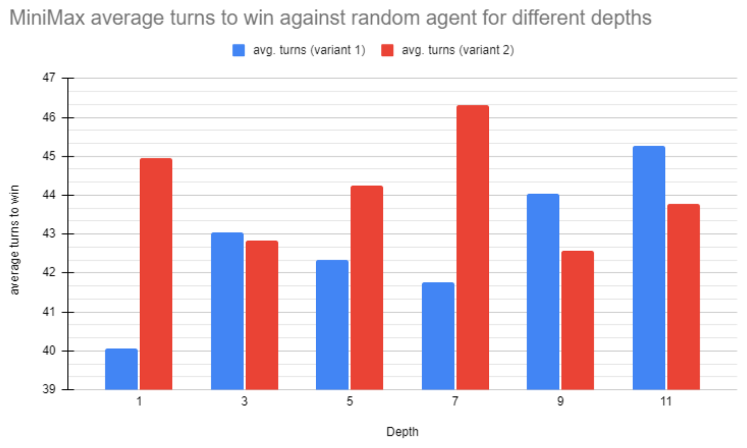


Figure 14. MiniMax average turns to win against random agent for different depths

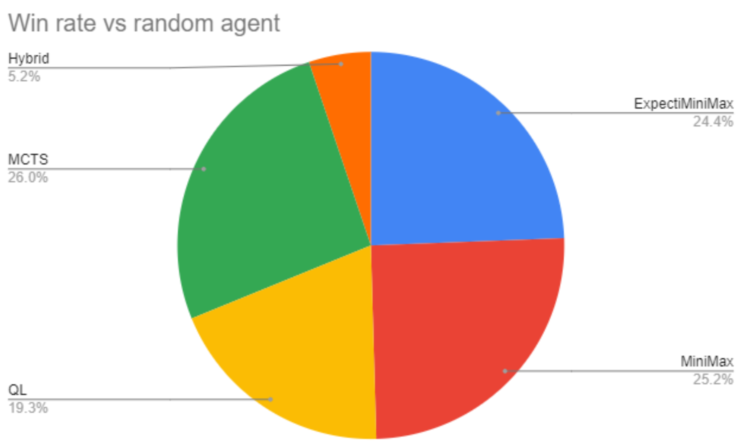


Figure 15. Win rate vs random agent

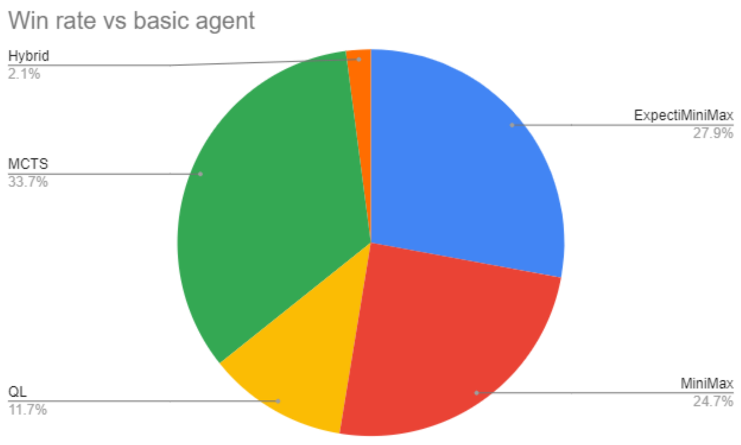


Figure 16. Win rate vs basic agent

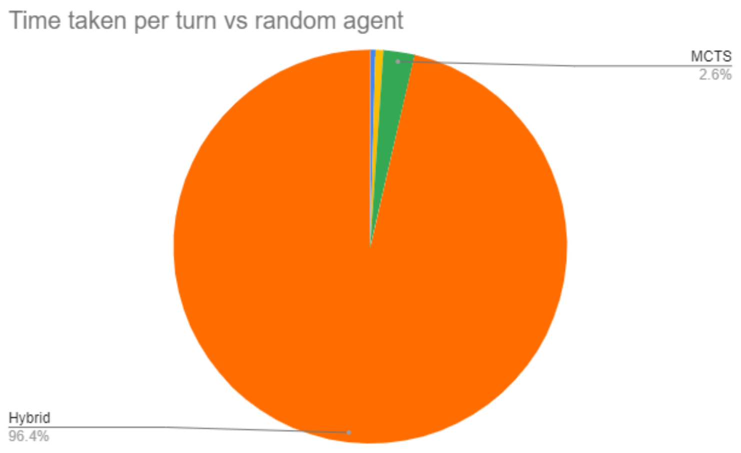


Figure 17. Time taken per turn vs random agent

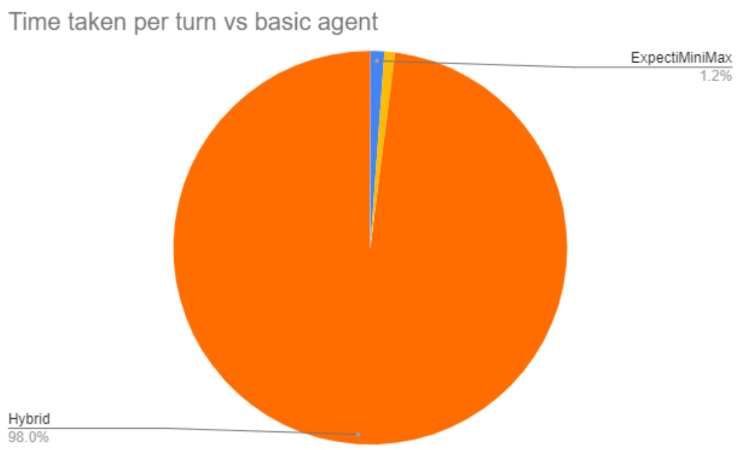


Figure 18. Time taken per turn vs basic agent

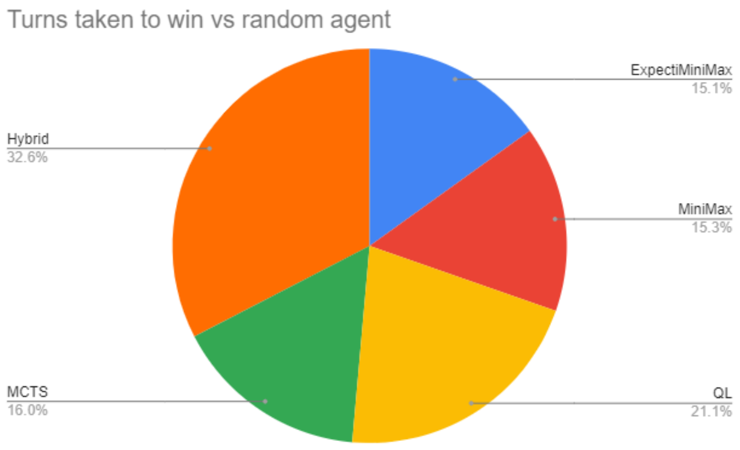


Figure 19. Turns taken to win vs random agent

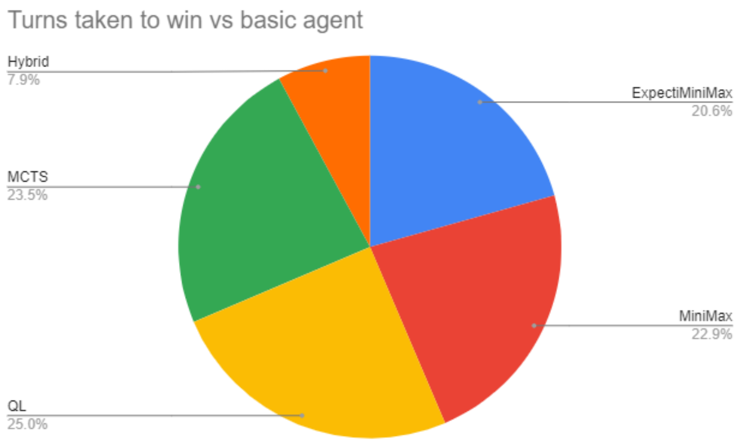


Figure 20. Turns taken to win vs basic agent