Search and Learning Agents for Dice Chess

Mohammed Bashabeeb
Pie de Boer
Valentina Cadena Fajardo
Pierre Paul Charbonnier
Eden Rochman Sharabi
Abdullah Sahin
Saman Sarandib

22 January 2023

Contents

1 Abstract

Dice chess is a variant of classic chess that uses a six-sided die to introduce a stochastic element to the game [3]. Hence, this paper investigates different techniques for building artificial intelligent agents capable of playing dice chess.

As there are different fields that provide heuristic techniques, the techniques that this paper researched are the adversarial search techniques Monte-Carlo Tree Search and ExpectiMiniMax and the machine learning techniques Neural Network and Q-Learning.

Experiments have indicated that the ExpectiMiniMax agent is the best performing adversarial search agent despite its increased time complexity to perform a single heuristic, whereas the Monte-Carlo Tree Search agent turned out to be a worst performing agent due to the tree consisting of Forsyth-Edwards Notations instead of movements. Moreover, machine learning agents such as neural network and q-learning turned out to be non-optimal as the training data set was insufficient to yield high performance and their structures was not properly tuned for dice chess.

Hence, for games with a complex stochastic environment with a large state-action space and the lack of data from expert level game play, it is suggested to settle for adversarial search agents.

2 Introduction

This research paper explores different techniques for building artificial intelligent agents to play a game of dice chess. At every turn, dice chess introduces a constant stochastic element that is restricting the number of available moves. This constant stochastic element is achieved by causing the available moves to remain unknown until a six-sided die has been rolled by the active color of the game[1].

There are a variety of heuristics that provide the answer to the question 'How to select a promising movement from the restricted set of movements?' each dice chess player has. One such field is the field of adversarial search. A technique known as Monte-Carlo Tree Search provides a simple tree modelling of the question which is then used to probe one or more solutions. The Monte-Carlo Tree Search algorithm consists of 4 phases; selection, expansion, simulation and back propagation. Using those 4 phases the algorithm is capable of solving the question on how to select promising movements from the restricted set. However, due to the presence of the stochastic element the algorithm loses some of its efficiency as the actual next state of the game may vastly differ than what has been explored. This is mostly the case whenever a player has thrown the die on a number representing a chess piece the player either does not have or can not move.

There exists a technique that is capable of tackling those types of problems. This adversarial search technique is known as the expected min-max algorithm which is a variation of the min-max algorithm. This technique takes chance elements into account by including chance nodes and expected values into its tree search model.

As for exploring machine learning techniques to build agents capable of playing dice chess, it is a prerequisite to find out how learning could be done and how it could be used. One of the first answers that were found that seemed feasible was the concept of a value network, as this technique demonstrated a high level of success in Alpha Zero.[9]. This success was accomplished by letting the network train on a database consisting of board states and their associated outcomes. Due to its training phase, the network was able to predict the probability of winning at any moment for any team given any board state.

Inspired by other major success within deep (reinforcement) learning [7], we looked into using q-learning with neural networks. We trained a neural network to serve as a q-function approximation. The input nodes corresponded to a board state and output nodes to a q-value for taking a a specific move, also known as action. The network could be used to predict the best move based on highest q-value. The previously mentioned value network was used to play the strongest move of the adversarial agent. As both the adversarial search and machine learning techniques have their advantages and disadvantages, it seemed interesting to figure out which of two fields would provide the better agent that is capable of playing a game of dice chess.

Hence, this paper aims to answer the question 'Which technique is more suited to be implemented in an environment with a stochastic element that is capable of generating a large number actions and states?' by answering the sub-questions:

- How does a stochastic element influence an adversarial search agent?

- How does a stochastic element influence a machine learning agent?

- How does a large number of states influence an adversarial search agent?

- How does a large number of states influence a machine learning agent?

- Does a machine learning agent outperform an adversarial search agent?

To start answering the questions through implementation experiments, the definitions of the techniques are required. Some techniques require some form of evaluation for the game state. One such function that is providing a chessboard evaluation is the fitness function, a function that determines the 'fitness' of the chessboard. This function requires a set of weights to perform its evaluation of the chessboard, hence different weights may lead to different evaluations. Therefore, the Genetic Algorithm was introduced to tackle the issue of weight optimization.

3 Methods

3.1 Monte-Carlo Tree Search

The very first explored adversarial search technique was the Monte-Carlo Tree Search. Since the algorithm is defined by its four phases, each phase requires a definition before implementation.

3.1.1 Selection

In the selection phase, a search for the most promising leaf is always started from the root in a depth-first search like fashion. Once a leaf has been found, this leaf is then returned as the output of the method. Selecting children is done according to a selection policy. The default selection policy is to select the child with the highest probability of winning. The winning probability is calculated by dividing the number of wins the child has recorded by the number of times the child is explored.

3.1.2 Expansion

During this phase, if the leaf obtained from the selection phase does not represent a concluded game, the leaf will be exploited through giving it children. Each assigned child represents a possible next future state of the game. The exact expansion of the leaf is dependent on the expansion policy. The default expansion policy is to expand an arbitrary number of nodes randomly.

3.1.3 Simulation

Once the expansion of the leaf completed and leaf does not represent a concluded game, one of its children will be used to produce a simulation on how the game could conclude. Both selecting a child for simulation and how a simulation is run are dependent on policies. The default policy of selecting a child for simulation is to take one randomly, whereas the default policy of simulating the game outcome is to perform random movements until a termination condition such as the capture of a king has been reached.

3.1.4 Back Propagation

In the back propagation phase, the conclusion of the simulation is used to update the information contained in all nodes from the simulated child up to the root node. Just like all other phases, which and how information is updated is dependent on policies. The default policies are to increment the visit count of each explored node by one and the win count by one if the simulation has ended in a color winning and the color data of the node matches the winning color of the simulation.

3.2 ExpectiMiniMax

The second explored adversarial search technique is the ExpectiMiniMax algorithm, which is a variation of the min-max algorithm to manage uncertainty factors. Both algorithms have similar structures, with the only major difference between the two being the assumption on which the problem modelling is based.

3.2.1 Chance Nodes

The ExpectiMiniMax algorithm is capable of taking uncertainty into account through the use of chance nodes, besides the usual min and max nodes. A chance node, is a node that provides a probability whether a piece could be moved. Moreover, a chance node also provides an expected evaluation value of all its children through the formula:

$$E(X) = \mu = \sum_{j=1}^{+\infty} x_j P(X = x_j)$$

where, $E(X)$ represents the expected value, x_j the evaluation value of j^{th}-child and $P(X - x_j)$ represent the probability of the state contained in the child being manifested. As the movement probabilities are distributed uniformly, all possible next states from one state are linked to a single chance node. Ultimately, allowing the root node to conveniently select the chance node with the highest expected value.

3.3 Fitness Function

For the ExpectiMiniMax algorithm to function properly, it requires some form of evaluation that should be minimized and maximized. This is where the Fitness Function comes into place. The Fitness Function is more than suitable for the algorithm as the function aims to provide a more complete evaluation of the game state through evaluating different aspects of the piece placement data.

3.3.1 Function

The function performs 37 different evaluations of the piece placement data, followed by a linear combination between a set of weights and the evaluation results to obtain a final evaluation.[2]. Hence, the Fitness Function has the following formula:

$$Val = \sum_{i=1}^{37} w_i \times f(i)$$

where, Val represents the evaluation value, w_i represents the i^{th}-weight and $f(i)$ represents the value of the i^{th}- evaluation feature.

3.3.2 Fitness

Using the formulate above, the fitness of an active color could be determined. This is accomplished by evaluating the piece placement data for both the active color and its opponent color, followed by subtracting them from each other, leading to the equation;

$$Val_{fitness} = Val_{active} - Val_{opponent}$$

3.3.3 Weight Optimization

As different weights will lead to different final evaluations, two approaches have been used to optimize the set of weights, namely the Genetic Algorithm and the Hill Climbing technique with each having their own pros and cons.

3.3.4 Evaluation Features

To provide a slight idea of what the Fitness Function exactly evaluates, some of its 37 features are described below:

- Weak squares: This method find the list of squares that a player cannot protect with its pawns. This is relevant because a pawn is the lowest value piece, so it makes the perfect protector of a given square. In general, in chess one want to protect a square with a less valuable piece possible or at least with a lower value piece that one opponent. Thus, finding the unprotected squares by pawns fulfill the goal.
- Pawns position: This method is related with the first method explained, the power of pawns fall to its low value per unit.
- Pieces value: These sets of methods find the total number of pieces for a given type: number of pawns, knights, bishops, rooks, queens and king.
- King safety: These sets of methods find the potential dangers of the king and how protected the king is by other pieces of its same team.
- Queen mobilization: This method find the number of squares. The queen is the most dynamic piece; it mobility it makes it a dangerous piece because it can attack and protect at the same time many squares and pieces.
- Bishop control: These sets of methods analyse the control of the bishop over the boards. Traditional chess theory value bishops and knights with the same value but modern chess theory value a pair of bishops stronger than a pair of knights. This is due the long reach domination of the pair of bishop.
- Rook control: These sets of methods analyse the control of the rook on the board. A set of rooks is immensely strong for very similar conditions as the bishop but with the addition that a rook's movement is not limited to an unique color, unlike the bishop's. Also, a bishop and a king cannot checkmate but a rook and a king can, so bishop and king against king is a draw.

3.4 Genetic Algorithm

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that are based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce children for the next generation. Over successive generations, the population "evolves" toward an optimal solution[4]. The genetic algorithm uses three main types of rules at each step to create the next generation from the current population: Selection rules select the individuals, called parents, that contribute to the population of the next generation. The selection is generally stochastic and can depend on the individuals' scores. Crossover rules combine two parents to form children for the next generation. Mutation rules apply random changes to individual parents to form children.

The main goal of the implementation of Genetic Algorithm in the project, is to find the optimal weights for the fitness function parameters, in order to best evaluate the state of the game. In the below subsection, we will be talking about how the algorithm was used to solve a stochastic die chess problem and to get a final solution which is represented as a chromosome with the best values selected[5].

3.4.1 Chromosome

In the chromosome class, we represent the body of the chromosome that will be used in the algorithm. The chromosome body contains 37 parameter weights represented as indexes that will get their values from the fitness function. Each chromosome object has a crossover method which switches the first half body of the current chromosome with the other second half of the other chromosome. The chromosome has also a mutate method which checks the newly created chromosome body and checks if the randomly generated number is below the mutation factor then it multiplies this value with this mutation factor.

3.4.2 Bot

The bot class object is one that is in charge of picking the chromosome-generated body values and representing them as moves on the chess board. It first generates all possible moves for this specific chromosome and sees the possibilities of playing a move, based on this simulation is stored the values of each move and stored in a list which will be used at the end to define the best final move of this particular chromosome.

3.4.3 Trainer

In the trainer class, we do the whole game simulation where we first define the parameters of the mutation factor, population and generation numbers. First of all, in the constructor, we first make the generation which will later have to run the cycle of the genetic algorithm. Each generation will start first with the selection process where it first creates the chromosomes if it's the first iteration, otherwise, it directly moves to the matching part where two chromosomes play against each other and return the winner which will be returning from the match function which simulates the game between the two chromosomes. Then it moves to the cross-over method where the algorithm chooses between the best ten bots and crosses their values with each other to generate new ten bots. After that, the algorithm goes to the final part of it, where it mutates the values of the bot if it meets the condition. When the algorithm reaches the last generation, it made the last game selection between the last 20 generated bots, and based on that it gets the final best chromosome which will be used to play against other bots such as random bot.

3.4.4 Hill Climbing

As an alternative to the Genetic Algorithm, Hill Climbing was also implemented to tune the weights of the fitness function parameters. Hill climbing is an optimization technique which assumes there is only one local minima. This algorithm begins with a random solution to a problem, and checks every step if there is a better solution by increasingly modifying the current solution, until the current solution can not get any better; meaning it reached the local minima.

3.5 Learning Techniques

In order to explore machine learning techniques, two main approaches were investigated, namely value networks and q-learning. The main goal of the value network was to predict the win probability for a specific team based on just a board state. Therefore, this would give us information 'value' which moves were the best, since moves obviously result in a new board state.

Q-learning formed the basic principle for our reinforcement agent. The Q-learning algorithm is an off-policy reinforcement learning algorithm that learns the optimal action-value function through updating the Q-values (estimations of the action-value) based on the observed rewards [11].

Since these learning techniques are very dependent on their actual chosen parameters, and therefore implementation, further details can be found in the implementation section.

4 Implementation

4.1 Forsyth-Edwards Notation

The Forsyth-Edwards Notation (FEN) for classical chess has used to represent the games of dice chess as the major difference between the properties of the two is only the presence of the die. Missing the die roll information in a FEN-string turned out to be not an issue as an assumption was made that every FEN-string implies that die has not yet been rolled.

4.2 Simulation

To perform investigation on the performance for each agent, a simulator was implemented. A simulator is a tool that automatically switches active colors after each turn has ended, registers each movement an agent has chosen to play and yields the conclusion of the game.

4.3 Learning Techniques

4.3.1 Database

A database of game outcomes was generated through self-play of the genetic agent and random agent. Most games were played using the genetic agent, since it's stronger capabilities would represent more realistic matches. Every game, the encountered board states were saved in a stack. Once the game terminated, the full stack was popped and board state was send to the database alongside the outcome of the game. If a board was already present in the database, only the outcome was sent, which then was summed up on top off the existing value.

The database contained the following information:

- FEN: *Board position*
- White Wins: *Number of times white won*
- Black Wins: *Number of times black won*
- Draws: *Number of draws*

Data was fetched and received between Java and MySQL using JBDC drivers. Training of the network was done by fetching data without loading it into RAM. The most important information about the database is captured in the table below:

Attribute	
Number of games	165221
Table rows	4629455
Table size (estimate)	703.0 MB
Percentage games GA	80%
Percentage games RANDOM	20%

Table 1: Information about database generated through self play of agents

Attribute	Value
Weight Initialization	*XAVIER*
Gradient Updater	*ADAM*
Learning Rate	-
Input Nodes	*384*
Hidden Nodes	*512*
Output Nodes	*1*
Activation Functions (all layers)	*SIGMOID*
Loss Function	*MSE*

Table 2: Value Network Hyper-parameters

Attribute	Value
Weight Initialization	*XAVIER*
Gradient Updater	*ADAM*
Learning Rate	-
Input Nodes	*384*
Hidden Nodes	*128*
Output Nodes	*80*
Activation Function (input, hidden)	*ReLU*
Activation Function (output, hidden)	*IDENTITY*
Loss Function	*MSE*

Table 3: Q-Network Hyper-parameters

4.3.2 Neural Networks

Board states that were represented as FEN strings were transformed using a one-hot encoding principle to be used as input to the neural network. This encoding generated an array containing 0s and 1s of size 384. As a value network, the neural network was used in the selection procedure of Monte Carlo Tree Search (MCTS). In the reinforcement learning agent, it was used to predict the maximum q-value of the adversarial agent.

In order to have easy access to more advanced architectural features for our network the Deep4j library was used. The library can be easily imported using Gradle/Maven. Using deep4j a value network with the attributes described in table 2.

The neural network to be used for our q-function approximation had the following characteristics described in table 3.

4.3.3 Q-learning Agent

Due to the size of the state space, the usage of an Q-table was unfeasible. Therefore, we trained a neural network to the predict the q-values for each action-state pair for team white.

In dice chess, the reward had to be determined for either intermediate moves or terminating moves. The genetic algorithm was used to evaluate the difference in board strength between two consecutive states to determine the intermediate rewards.

The neural network took the encoded board state as input and the output corresponded to the q-values for each of the legal moves.

During training, moves of the agent were guided by an epsilon-greedy policy to balance exploitation and exploration, where each predicted q-value of a state was obtained from the network. After calculating the intermediate or final reward and determining the max Q of the next state-action pair, all actual Q values were gathered and added to an array.

Running many games would theoretically learn the q-value for every state-action pair and allow the agent to pick the best 'long term' move in any state. Therefore, the neural network used in q-learning served as a q-function approximation.

A high level overview of how the q-Learning loop works is as following:

```
While game is ONGOING
    For legal moves WHITE:
        predict Q(S,A) for all
        state-move with NN
        if(intermediate):
            reward :=
                eval NEXTSTATE -
                eval CURRENTSTATE
        else if(WHITE won):
            reward := + finalReward
        else if(BLACK won):
            reward := - finalReward
        else if(DRAW):
            reward := 0
        For legal moves BLACK:
            calculate Q(S',A')
            using value network
        Q(s,a) : =
            Q(s,a) + lr *
            (reward + discount *
            (max Q(s',a') - Q(s,a))
        actualQ[i] = updatedQ
    fit(input, actualQ)
    play WHITE move based epsilon-greedy
    play BLACK move based value network
```

Parameter	**Value**
Learning Rate	0.01
Discount Factor	0.99
Epsilon	0.01

Table 4: Q-Learning Hyperparameters

Since the q-function network was only trained for white. The value network trained on our self-play database served to inform which move has the highest probability of winning for the adversarial agent. Hence, the q-agent could be used as following by predicting the q-value of all moves pick move with highest q-value. Therefore, it should ideally make the best long-term move.

The table above provides parameters used in q-learning process.

4.4 Monte Carlo Tree Search

The Monte-Carlo Tree Search has been implemented through defining each of its phases as an independent method. This in combination with providing nodes the capability to contain additional information such as FEN of the game state, was more than sufficient to implement the entirety of the algorithm with merely two classes.

4.5 ExpectiMiniMax

Evaluation of the ExpectiMiniMax state:
For evaluating the current state of the game, many functions are used. The opponent's current state is also evaluated. The end evaluation value is the difference between the evaluation value of the player and the evaluation value of the opponent. If the king gets eaten, the player, who's king got eaten, will receive a large penalty. This penalty is not $-\infty$, because that will result in a large information loss: no other value in that branch will be checked because the expected value of that branch is very low (near $-\infty$). Thus exploration will be neglected, if the penalty is set to $-\infty$.

For the ExpectiMiniMax algorithm we have chosen to use a tree structure to store the nodes used by the algorithm [10]. Because of the high branching factor for Chess games, branching the entire match would be impossible. Thus we have decided to implement two approaches.

4.5.1 Breadth First Search

The first one is Breadth First Search, which it cuts the tree at a specified depth, and evaluates the match at the leaf nodes of the cut-tree. The evaluation values of the leaf nodes is back-propagated to the child nodes of the root, from there, the root chooses the child with the highest expected value (best average outcome). Breadth First Search produces a more careful agent, since it considers every possible scenario up to a specified depth. However, this is also its weakness since it does not prioritize board positions that are more likely to be chosen by both players.

4.5.2 Beam Search

The second approach is Beam Search. Beam Search is an optimization of Best-First Search; it is a heuristic search algorithm that expands the most desirable unexpanded node in a limited set of leaf nodes. The desirability of a node (state) depends on the depth of the node and the evaluation score of the state: $h(n) = (evaluation * evaluationWeight) + layer$, where $evaluationWeight$ is a hyper parameter which controls the importance of a leaf's fitness. When it is set to zero, this implementation of Beam Search is equivalent to Breadth First Search. Beam Search has the advantage that it can reach deeper layers than Breadth First Search by considering the best moves for either player, before the worst ones. However, in principle, it may be slower than Breadth First Search since it must compute an evaluation for each node as opposed to only evaluating the leafs.

4.6 Genetic Algorithm

Dice Chess uses the Genetic Algorithm by applying the biological approach of the algorithm. The algorithm first initializes the chromosomes and then it defines its parameters according to a lower and upper bound. The first generation fight against each other and the best 10 bots will be crossed over (by a single crossover point manner) to generate 10 new chromosomes. After that, it mutates the body of the chromosomes with a certain probability (mutation rate). The algorithm repeats this process for every generation until it terminates when it reaches the final generation. After the last run of the last generation, the turn is now to get the best chromosome by running the selection, without crossing over or mutating this time. As final work, when the algorithm runs the best bot part, it writes the newly created best chromosomes to a text file for later testing purposes[6].

Parameters:

- Mutation rate: 0.007
- Generations: 200
- Population: 20

4.7 Hill Climbing

The Hill Climbing algorithm is used to optimize the weight parameters of a bot. The initial solution is set as the midpoint between the lower and upper weight bounds for each parameter. The algorithm iteratively modifies the weight of one parameter at a time, while keeping the other weights fixed. The bot with the current weight plays against another bot with the new weight and the best new weight is chosen based on the number of wins. The weight of a parameter is modified by a percentage increase or decrease, starting with 10% and gradually decreasing to 1.25%. Initially, if the new weight improves performance, it is kept and the process is repeated until it stops improving. If the new weight does not improve performance, the weight is decreased until performance stops improving.

5 Algorithm Analysis

5.1 ExpectiMiniMax

The result of the time-complexity analysis for the ExpectiMiniMax algorithm is O(b^m). This is because b^m is the biggest weight in the calculation of the time-complexity of the ExpectiMiniMax algorithm.
b is the branching factor, this is each possible move from the current state.
m is the cut-off depth, this is not the actual tree depth, $(((depth-1)*4)+2)/2$ is used as m, this formula converts the tree depth into layers, excluding the chance nodes and root node. The chance node does not branch out and the root node is a layer with a single node.

The time-complexity is explained as follows: for each increase in m, the tree branches out with the branching factor b, thus for each increase in m, the time-complexity of the algorithm is multiplied by b. This is the biggest factor in the time-complexity of the entire algorithm, thus big-O notation of the ExpectiMiniMax algorithm is O(b^m). Although the time-complexity of the ExpectiMiniMax is O(b^m), there are also many other operations that further increase the time-complexity, for example the evaluation of the leaf-nodes has a time-complexity of O($b * m$), and the back-propagation of the evaluation is O(b^{m-1}). These are the operations, that have the highest contribution in the time-complexity of the ExpectiMiniMax algorithm.

5.1.1 Improvements

With the previous version of the ExpectiMiniMax algorithm, 2 issues were encountered: The first is the high time-complexity and the second is the approximation to the target function

5.1.2 Time-Complexity

Regarding the high time-complexity, only 1 practical solution is available for ExpectiMiniMax algorithms and that is the α-β pruning. But this is not applicable to the current ExpectiMiniMax algortihm, this is due to the lack of restrictive boundaries set to the evaluation values [8]. Thus we cannot determine an upper-bound for any subtree to prune it. Therefore only small optimizations have been made to the code to improve the time-complexity and space-complexity, but no significant improvements in run-time have been observed.

5.1.3 Evaluation

The performance of the ExpectiMiniMax algorithm is determined by the evaluation function it uses. Thus a good evaluation function is important. The previous version of the ExpectiMiniMax algorithm used a collection of methods to evaluate the current state of the board, with a certain constant attached to them, this constant acted as a level of significance of each evaluation method (weight of each method). Although the new version uses the same collection of methods, with a few new ones added, the weights attached to those methods are not constants anymore. By constantly generating new winning chromosomes (weights), using the genetic algorithm, the average of all generated chromosomes can be used to determine the weights used by the ExpectiMiniMax algorithm for the evaluation methods. This will result in a more accurate approximation of the target function, due to the fact that a single chromosome generated by the genetic algorithm still contains inaccurate weights assigned to the evaluation methods. Thus by taking the average of all generated chromosomes, this error is reduced, and converges to 0 given a sufficient amount of chromosomes (near ∞). This improves the approximation of the target function and consequently the performance of the ExpectiMiniMax algorithm improves significantly.

See paragraph 5.3 for further information on the Genetic Algorithm.

5.2 Monte-Carlo Tree Search

The MCTS algorithm repeatedly executes four main steps: selection, expansion, simulation, and back-propagation. The number of times the algorithm is executed is limited by a time constraint, which controls the number of iterations we denote as n. The overall time complexity of the algorithm can be represented as $O(n * (selection + expansion + simulation + backpropagation))$.

In the selection phase, the Upper Confidence Bound for Trees (UCT) selection policy is used. The worst-case scenario occurs when the Monte Carlo tree is large, resulting in a deep tree with a high branching factor. In this case, the time complexity is $O(b * d)$, where b is the branching factor and d is the depth of the tree. It is important to note that the tree might not be fully generated during this phase.

Expansion has a strong influence computational demand and time rationality. To avoid excessive memory usage but allow for sufficient exploration, an expansion parameter with value $500M$ was picked, which will never be fully searched; firstly there are not sufficient legal moves, secondly, implementations constrains for time rationality.

The time complexity of the simulation step depends on the maximum number of layers iterated through until a terminal state is reached. The process of generating available moves for a given team has a time complexity of $O(fpr)$, where f is the number of files, p is the maximum number of possible moves for a piece, and r is the number of ranks. This process is repeated k times. Therefore, the overall time complexity is $O(n*(500M(k+1)*s))$.

5.3 Genetic Algorithm

Starting with the complexity analysis of the fitness function, the methods inside the class 'FitnessFunction' have a complexity of O(1) because the values of the parameters are calculated by iterating over pieces and board squares, which are never greater than 64. Given that the selection function makes all bots play against each other, its time complexity is $O(P^2 * M)$, derived from $O(((P^2/2) - P) * M)$, where P is the population size and M is the number of movements before a player wins a match. The computation time for each game, O(M), is multiplied by the total number of combinations for all bots to play against each other, $O(P^2)$, because all bots must play against each other to determine the next generation. In addition, since worst case scenario, the bots must be created for the first generation, then there is an additional complexity of O(P). Hence, the time complexity of the selection function is $O(P) + O(P^2 * M)$, which resolves to $O(P^2 * M)$.

Furthermore, the crossover process has a complexity of O(P), since it repeats the crossover process until it generates P new bots, were P is the population size.

Overall, the process of training the bots, from their initialization to selecting the best bot, has then a complexity of:

$$O(G)*(O(P^2)+O(P)+O(P))+O(P^2) = O(G*P^2)$$

This stems from the fact that, as previously calculated, the process in each generation has a complexity of $O(P^2) + O(P) + O(P)$; therefore, this complexity is multiplied by the number of generation G. In addition, once the bots of the last generation have been generated, the time to compute the best bot (with the best chromosome) amongst them is $O(P^2)$, since it must execute the selection function one last time.

Based on the prior analysis, it is concluded that the time complexity for the Genetic Algorithm to find the best chromosome is $O(G * P^2)$. Since G was chosen to be 200 and P to be 20, then the algorithm performs around 80,000 operations.

5.4 Q-Learning

In this section, we will analyze the algorithm complexity of the Q-learning training phase iterating over the following steps. It is recommended to look at the pseudo-code for this algorithm in the implementation section.

1. Number of games, *size N*
2. White legal move, *size W*
3. For each counter move of black, therefore for every legal move of white, *total B per W*
4. Update Q-value inside for moves white, *contant time*
5. Fit Q-values in neural network after all move loops, *implementation dependant*

The overall algorithm complexity can be represented as $O(N \times B \times W^2)$ as the number of games, size N, is multiplied by the number of counter moves of black, B, for every legal move of white, W. The Q-value update step is assumed to have a constant time complexity. Fitting the updated Q-values after in the neural network is dependant on the implementation of the network.

6 Experiments

6.1 ExpectiMiniMax

To test the performance of the ExpectiMiniMax bot, it is put up against the Random bot and the Genetic algorithm. For each opponent 500 simulations have been run, to thoroughly test the performance of the ExpectiMiniMax bot. For these experiments, the depth of the ExpectiMiniMax bot had been limited to 2, this was done for performance purposes.

6.1.1 Improvements

To test the performance increase of the ExpectiMiniMax improvements, many experiments have been conducted against the previous version of the ExpectiMiniMax algorithm and against other agents. To gather information on the learning curve of this approach, different versions of the improved ExpectiMiniMax algorithm have been

created, each with weights obtained of different numbers of generated chromosomes, i.e.
1. One version uses the mean value of 50 different chromosomes as weights, called Expecti-50.
2. Another uses the mean value of 150 different chromosomes as weights, called Expecti-150.
3. And the strongest version uses the mean value of 300 different chromosomes as weights, called Expecti-300.

6.1.2 Beam Search or Breadth First Search

The first experiment regarding Beam Search, was to find the most optimal evaluation weight. For this, bots with different evaluation weights played against each other, to examine their performances. Every bot played 40 times against every other bot.

Afterwards, an experiment regarding what type of search is best for ExpectiMiniMax was pursued. For this test, it was considered Beam Search with its optimal evaluation weight and Breadth First Search.

6.2 Genetic Algorithm

As the first experiment for the Genetic Algorithm, we wanted to test different selection and crossover methods. On one hand, the algorithm finds the optimal weights by selecting randomly two different chromosomes that are within the 10 chromosomes that performed best out of the 20 chromosomes, and making the crossover of the chromosomes with a rate of 0.5, to create 20 individuals. On the other hand, the algorithm finds the optimal weights by applying the same selection method as the previous, but it also applies an elitist selection, where 5 of the 20 individuals that are created for the next generation are an exact copy of the 5 chromosomes that performed best out of the current 20 chromosomes, and to create the other 15 individuals, the crossover of the chromosomes is applied with a rate of 0.25 instead of 0.5.

Furthermore, to test how the performance of the Genetic Algorithm improves over the generations, the best chromosomes of different generations were put to play against each other. Each best chromosome bot of every generation played 30 games against every other chromosome bots.

Finally, to test the performance of the Genetic Algorithm bot, it competes against the Random bot. A total of 500 simulations have been run, to thoroughly test the performance of the Genetic Algorithm bot.

6.2.1 Genetic Algorithm or Hill Climbing

In addition, to test which of the two algorithms - between Genetic Algorithm and Hill Climbing - obtains more optimal weights, we made two chromosomes with the optimal weights that were found by each algorithm fight against each other 2000 times.

6.3 Monte Carlo Tree Search

Different agents were setup to play against each other. A small amount of matches was chosen, since this already quickly showcased which bot gave significantly better performance. The agents that were examined were the following:

- MCTS D: *distance heuristic in expansion*
- MCTS N: *selection procedure optimized with value network*
- RAND: agent that performs random moves

It is worth noting that in traditional chess, the white player is often considered to have a slight advantage due to the ability to move first. However, in a game like dice chess, where the roll of the dice plays a significant role in determining the outcome, this advantage may be less pronounced.

6.4 Q Agent

In order to investigate the performance of the reinforcement agent, a different number of training iterations was used to see whether the performance of the agent improved. Since weak performance against an agent that made random move was observed, it seemed of little value to experiment against other agents.

7 Results

7.1 Optimization of the Parameters

7.1.1 Genetic Algorithm

Regarding the first experiment which compares two different selection and crossover methods:

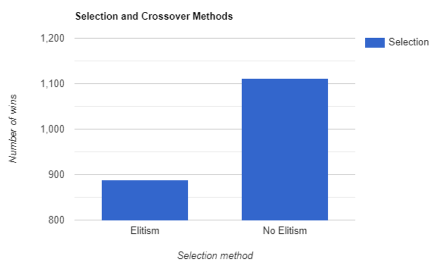


Figure 1: a plot, with the number of wins by using Elitism and without Elitism

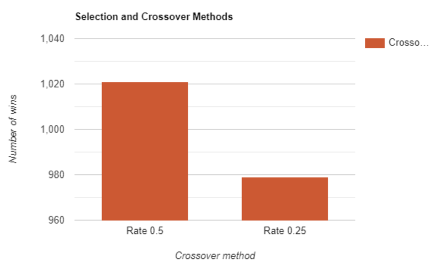


Figure 2: a plot, with the number of wins by using Crossover with rates 0.25 and 0.5

Next, the results for the number of games won by the best bot of different generations, can be shown in the following graph:

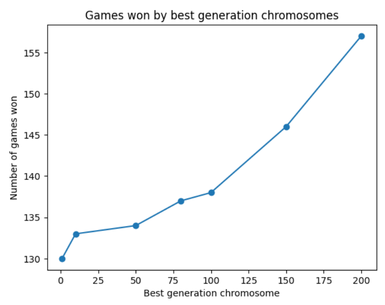


Figure 3: a plot, visualizing the experiment result of the number of wins made by the best chromosome of different generations

Lastly, it was found that over 1000 games, Genetic Algorithm wins 975 games meanwhile the Random Bot wins 15. Meaning Genetic Algorithm wins 98.5% of the time, leaving Random to win 1.5% of the times.

7.1.2 Genetic Algorithm or Hill Climbing

Lastly, after testing the performance of Genetic Algorithm with respect to Hill Climbing, it resulted that the Genetic Algorithm wins 1126 times when Hill Climbing wins 874 out of 2000 games.

7.2 ExpectiMiniMax

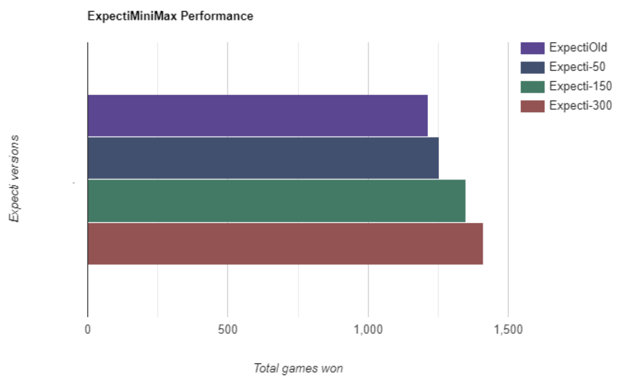


Figure 4: A bar graph showcasing the performance of each ExpectiMiniMax version, the performance is measured using the total sum of games won. (see 5.0.1 for the definitions of the different ExpectiMiniMax versions)

Player 1	Player 2	Result	Time(ms)
MCTS D	MCTS D	9-11	9966
MCTS D	GA	1-19	3602
GA	MCTS D	16-4	3961
MCTS D	MCTS N	19-0	8075
MCTS N	MCTS D	6-12	11357
MCTS D	MCTS D	3-17	9966
MCTS D	RAND	19-1	5977
RAND	MCTS D	6-14	5935

Table 5: Game outcomes of MCTS based agents

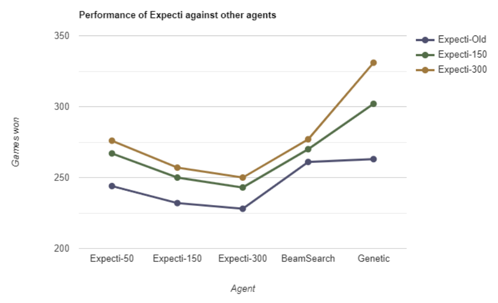
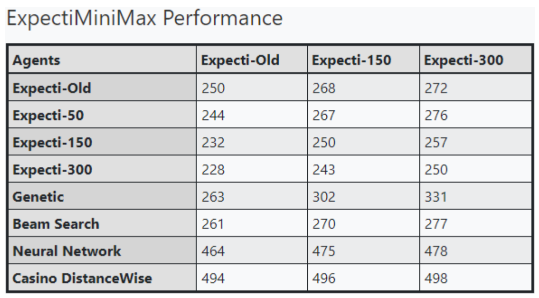


Figure 5: A line graph showcasing the performance of each ExpectiMiniMax version against other agents, the performance is measured using the total sum of games won out of 500 simulations against each agent. (see 5.0.1 for the definitions of the different ExpectiMiniMax versions)



ExpectiMiniMax Performance

Agents	Expecti-Old	Expecti-150	Expecti-300
Expecti-Old	250	268	272
Expecti-50	244	267	276
Expecti-150	232	250	257
Expecti-300	228	243	250
Genetic	263	302	331
Beam Search	261	270	277
Neural Network	464	475	478
Casino DistanceWise	494	496	498

Figure 6: A table showcasing the performance of each ExpectiMiniMax version, the performance is measured using the total sum of games won out of 500 simulations, by the agent at the top of each column, against each agent. (see 5.0.1 for the definitions of the different ExpectiMiniMax versions)

Training Rounds	qAgent	Random
100	14	10
500	12	13
1000	15	10
2500	17	8
5000	18	7
10000	12	13

Table 6: Performance of qAgent vs Random Agent playing 25 games tracking only wins

7.2.1 Beam Search

Regarding the most optimal weight for Beam Search:.

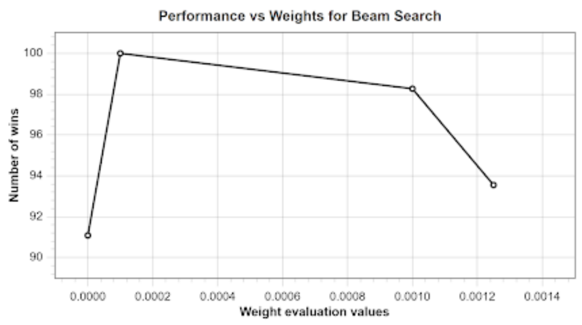


Figure 7: A bar graph showcasing the performance of different weight evaluations assigned to Beam Search

8 Discussion

8.1 Monte-Carlo Tree Search

From the experiment we can see Monte Carlo Tree Search is the least agent. Which implies that our current implementation is some how unsuccessful. Some ideas too improve it; recalling the delicate balance of constrains, a possible idea is to not simulate full games but a certain number of movements and then use some heuristic too evaluate the position.

This will allow to an immensely increase in the number of simulation MCTS can generates in the same amount of time. So instead of looking for the winning matches, the tree will find the best board positions. Eventually, sufficient strong moves will create a path to an inevitable victory.

From the results of our experiment, it appears that our implementation of Monte Carlo tree search did not perform as well as we had hoped. One potential reason for this is that we may not have found the right balance of constraints in our implementation. One idea to improve the performance of the MCTS algorithm would be to simulate a certain number of moves instead of full games. By doing this, we could increase the number of simulations that can be run in a given amount of time. Additionally, instead of solely focusing on finding winning matches, the algorithm could focus on finding the best board positions. By doing so, strong moves that lead to an inevitable victory would be more likely to be discovered.

In addition to that other functions such as UBC1, RAVE or MC-RAVE could be experiment with.

8.2 ExpectiMiniMax

The conclusion that can be drawn from these experiments is that the ExpectiMiniMax, although a lot slower than the random bot (time complexity of O(1)) and Genetic algorithm (time complexity of $O(G * P^2)$), dominates the Genetic Algorithm and the random bot, this is because the ExpectiMiniMax bot with the depth of 2, takes into account the state of the game after each player has played 2 moves, these are the expected moves, not the optimal ones. Whereas the Genetic bot only takes into account the state after the move has been played, and the random bot picks a random legal move, no form of evaluation. There are 2 explanations for the times the ExpectiMiniMax did lose:
1. the stochastic element of Dice Chess, this is linked to the die values thrown by the player.
2. the information loss as a result of calculating the expected value of a branch. This way, if a winning move can be looked over, as a result of taking the expected value of all the moves in the same branch. A solution to this problem is to give a winning move a high reward, but that impacts the exploration aspect of the bot.

8.2.1 Beam Search or Breadth First Search

Given the results for what kind of search should be used for ExpectiMiniMax, it can be inferred that using Beam Search it performs better than using Breadth First Search. As can be seen in Figure 7, where it compares the performance when having different evaluation weights, there are other weights that perform better than the weight of 0. As mentioned before, when setting a weight of 0, this is equivalent to Breadth First Search. Therefore, using Beam Search with a weight between 0.0001 and 0.00125, makes better moves than using Breadth First Search.

On the other hand, it can be derived also from Figure 7 that the optimal weight evaluation (which controls the importance of a leaf's fitness) for Beam Search is 0.0001, since it was the one that obtained the most number of wins.

8.2.2 Improvements

The ExpectiMiniMax with the improved evaluation function shows a significant performance increase over the older version of the ExpectiMiniMax algorithm, from the observed data the estimated increase is 17 percent see figure 2. Although the actual increase may vary due to the stochastic element present in the game, a total of 10.000 experiments have been conducted with the old and improved versions of the ExpectiMiniMax algorithm to minimize this variance and get a correct insight on the performance of the ExpectiMiniMax algorithm against other agents.

The ExpectiMiniMax still shows the best performance out of all the other approaches. As seen in the result section (see figures 3 and 4), the ExpectiMiniMax versions beat all the other agents by a significantly large margin, except the Beam-Search approach, which shows near similar performance as the ExpectiMiniMax breadth first search approach. Also The improved version of the ExpectiMiniMax algorithm shows an even larger win margin against other agents then the old ExpectiMiniMax algorithm.

8.3 Genetic Algorithm

Based on the experiments analysing the Genetic Algorithm's performance, it can be derived that the Genetic Algorithm has a much higher performance at playing Dice Chess compared to a Random bot.

Regarding the influence in the number of generations to find the bot with the best chromosome, it can be concluded that the higher the number of generations, the better its performance. However, it was found that after the 200th generation, the performance plateaus, which leads to think that the bots have already reached the optimal values for their chromosomes. Therefore, it is concluded that the recommended number of generations to find the best chromosome is around 200 generations, since it optimizes the values of the chromosomes while minimizing the computational time; which is why this parameter was chosen to be 200 for the report.

Following with the study of what algorithm should be used to tune the weights of the parameter functions, it is clear that the Genetic Algorithm is the algorithm that should be used, instead of using the Hill Climbing algorithm. This is because, using the weights provided by the Genetic Algorithm, it gives 28.83% better performance than Hill Climbing, given the information that Genetic Algorithm wins 1126 games out of 2000 and Hill Climbing just wins 874 games.

8.4 Learning Techniques

The machine learning techniques did not achieve good performance. Used as a value network inside the MCTS implementation it was unable to outperform a simple distance heuristic in node expansion. A weak performing value network likely had a major impact on building an effective Q-learning agent.

Networks with different architectures and hyper-parameters were used with little improvement. The main weakness, very likely had to do with the quality of the original dataset. Besides, using a neural network not for a task like image recognition, very likely provides to be way more difficult since chess is very complex and the movement of one piece, which might only cause little change in activations of neurons, might have a drastic influence on the win probabilty of a team.

9 Conclusion

Even though experiments depicted the Monte-Carlo Tree Search agent to be the worst performing agent and the ExpectiMiniMax agent to the best performing, the questions "How does a stochastic element influence an adversarial search agent?" can not be properly concluded due to not having an exact explanation on why the Monte-Carlo Tree Search has performed the worst.

However, the question "How does a large number of states infuence an adversarial search agent?" can be concluded due to the having sufficient information from the experiments, namely an adversarial search agent is capable of digesting a large number of states without any issues.

Due to lack of a good training data set, the question "How does a large number of states influence a machine learning agent?" remains unanswered, whereas the question "How does a stochastic element influence a machine learning agent" can be partially answered. Depending on how a neural network or the q-learning procedure is designed and developed, the influence of stochastic element on the performance may as well be considered as minimal.

This leaves the main research question "Does a machine learning agent outperform an adversarial search agent?" as the only unexplored question. In our case, it turned out both the machine learnig agents have been outperformed by the adversarial search agent counterpart, namely the ExpectiMiniMax agent.

Hence, for stochastic environments that lack proper data, but have a large number of possible actions and states, it is recommended to implement adversarial search agents over machine learning agents as the lack of proper training data is one of the biggest bottlenecks to obtain a highly performing machine learning agent.

10 References

[1] AftaabZia. “Expectimax Algorithm in Game Theory. Geeks-forGeeks”. In: (2021, October 25).

[2] A. Akdemir. “Tuning of Chess Evaluation Function by Using Genetic Algorithms”. In: (2017).

[3] *Game Rules (Dice Chess)*. URL: `https://brainking.com/en/GameRules?tp=95`.

[4] Jian. “Genetic Algorithms: Concepts and Designs”. In: (2005).

[5] John. “Genetic programming: An introduction”. In: *Santa Fe Institute Studies in the Sciences of Complexity* (1992).

[6] Kalyanmoy. “Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization”. In: *Evolutionary Computation* (2002).

[7] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), p. 529.

[8] S. J. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson Education, 2021, pp. 212–214. ISBN: 9781292401171.

[9] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (2017), p. 354.

[10] M. Simic. “Expectimax Search Algorithm”. In: (2022). URL: `https://www.baeldung.com/cs/expectimax-search`.

[11] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT press, 2018.

A Appendix

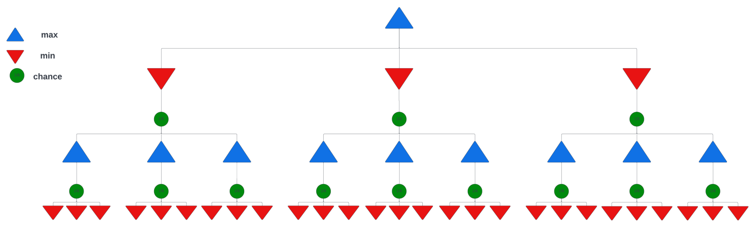
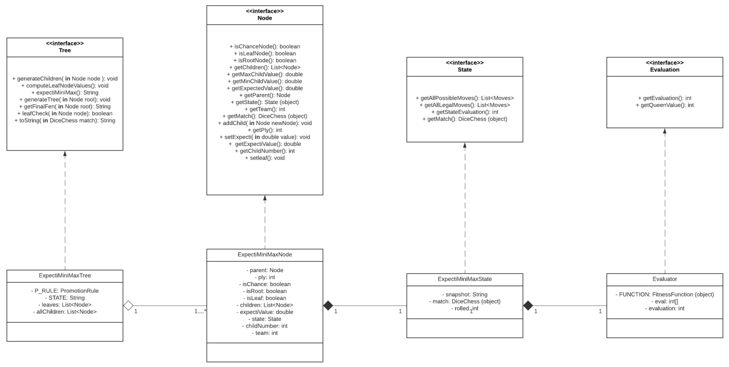


Figure 8: An example of an ExceptiMiniMax tree (depth of 2)



<<interface>>
Tree
+ generateChildren(in Node node): void
+ computeLeafNodeValues(): void
+ expectiMiniMax(): String
+ generateTree(in Node root): void
+ getFinalFen(in Node root): String
+ leafCheck(in Node node): boolean
+ toString(in DiceChess match): String
<<interface>>
Node
+ isChanceNode(): boolean
+ isLeafNode(): boolean
+ isRootNode(): boolean
+ getChildren(): List<Node>
+ getMaxChildValue(): double
+ getMinChildValue(): double
+ getExpectedValue(): double
+ getParent(): Node
+ getState(): State (object)
+ getTeam(): int
+ getMatch(): DiceChess (object)
+ addChild(in Node newNode): void
+ getPly(): int
+ setExpecti(in double value): void
+ getExpectiValue(): double
+ getChildNumber(): int
+ setleaf(): void
<<interface>>
State
+ getAllPossibleMoves(): List<Moves>
+ getAllLegalMoves(): List<Moves>
+ getStateEvaluation(): int
+ getMatch(): DiceChess (object)
<<interface>>
Evaluation
+ getEvaluation(): int
+ getQueenValue(): int
ExpectiMiniMaxTree
- P_RULE: PromotionRule
- STATE: String
- leaves: List<Node>
- allChildren: List<Node>
1
1....*
ExpectiMiniMaxNode
- parent: Node
- ply: int
- isChance: boolean
- isRoot: boolean
- isLeaf: boolean
- children: List<Node>
- expectiValue: double
- state: State
- childNumber: int
- team: int
1
1
ExpectiMiniMaxState
- snapshot: String
- match: DiceChess (object)
- rolled: int
1
1
1
Evaluator
- FUNCTION: FitnessFunction (object)
- eval: int[]
- evaluation: int

Figure 9: The UML diagram of the ExpectiMiniMax algorithm
https://www.overleaf.com/project/63904b0491a2582580d8b86b