Universiteit Leiden
The Netherlands

Opleiding Informatica

Comparing Strategic Agents for Dominance

Aäron Kannangara

Supervisors:
Walter Kosters & Jan van Rijn

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/07/2019

Abstract

DOMINANCE is an area-control board game played on a checkered seven by seven board. Players have to spawn their pieces onto the board while trying to exert dominance over the game board. As the game processes pieces will be spawned, moved, and attacked. Dominance can be exerted if the opposing player can no longer exert dominance or is no longer able to perform a move.

In this paper we compare and improve different stategic agents for DOMINANCE. The agents we will discuss are (Parameterized) Random, (Parameterized) Monte Carlo, Nega-max and Monte Carlo Tree Search. Our Nega-max agent uses a heuristic function and our Monte Carlo and Monte Carlo Tree Search agents use random game sampling.

By comparing these agents and using information about the game, we hope to find ways to easily improve the win rates of these agents while trying to find which agent performs the best for DOMINANCE.

Contents

1 Introduction

DOMINANCE is an area-control board game developed by C.M. Perry and published by The Game Crafter. The goal of this game is to use your pieces to exert dominance (or control) over the board by spawning pieces onto the board, blocking your opponent from spawning or hitting your opponent's pieces.
In this thesis we will compare several different agents, using stategic choices when playing. We will discuss how these agents are made and present their performance results.
By dividing all available moves up into move types (described in Section 2) we want to try and find which types of moves play an important role in winning, how to add value to these move types and apply this knowledge to improve agents using Monte Carlo and reduce work done by Monte Carlo Tree Search (MCTS).
By analysing what types of moves are important we want to show how computationally expensive sampling algorithms like Monte Carlo and MCTS can be tuned to improve win rates and reduce work. We also highlight how we can use data from random games to develop a heuristic function for a Nega-max agent.
For this thesis a program was developed in C++ that can play DOMINANCE on top of which several agents were developed including Monte Carlo, MCTS and Nega-max. By having stategic agents play against each other we will investigate the types of strategies which improve win rates and investigate eventual advantages for the starting player. The C++ code for our implementation is available on GitHub[1].
We will find that parameterizing the types of moves greatly improves win rates and that our parameterized Monte Carlo and MCTS agents perform the best of all our agents. When parameterizing the types of moves we divide all possible moves into three lists, choosing this list to select a move from based on a ratio $\gamma : \delta : \omega$.
This thesis is organized as follows. Section 2 describes the rules. Section 3 is related work. Section 4 discusses the different agents we will implement and improve. Section 5 prents the results of playing our different agents against eachother. Section 6 will conclude this paper and discuss further work.
This bachelor thesis was written for the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University, and was supervised by Walter Kosters and Jan van Rijn.

[1]Full details: https://github.com/akannangara/dominance

2 Game Rules

In this Section the rules DOMINANCE are explained. DOMINANCE has two opposing players on a seven by seven checkered board with an area control game mechanism and turn-based play. The goal of this board game is to exert dominance over the play field and dominate as many of the other player's pieces.
In this description of the game rules we put the rules provided with the game into our own words. This is to also help understand our implementation of the game. The official game rules can be found at [3]. This section describes the rules (which are based on the official game rules) used for our implementation of DOMINANCE.

2.1 Setup

DOMINANCE starts with an empty seven by seven black and white game board with four spawn points (marked green) in the corners of the board, see Figure 1. There are two players, Red and Black, both with ten identical pieces of their respective colors that start off the board. These pieces have to be dynamically spawned onto the playing field by using the spawn points in the corners. The players only use the (free) white and green spaces. The spawn points are at positions (a,1), (a,7), (g,1) and (g,7).

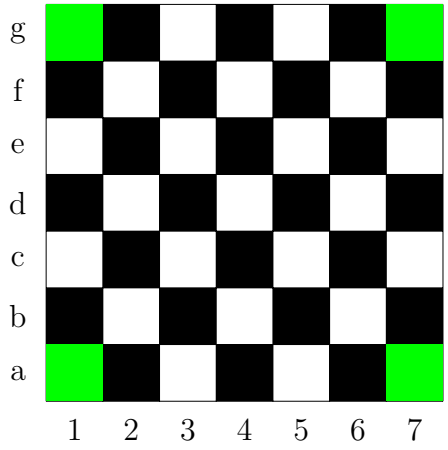

Figure 1: An empty game board. The green spaces represent spawn points.

2.2 Gameplay

The first player to start the game is the Red player and both players take turns performing an action. An action consists of either moving one of your own pieces already on the board or spawning a piece you still have off the board onto one of the available spawn points. By moving or spawning a piece a player can exert dominance over a space on the board by abridging that space from two sides. The rules concerning exerting dominance will be explained in Section 2.5.

2.3 Action: Moving

When performing a *non-attacking* action a player can move one of their respective in-play pieces on the board into one of the at most eight adjacent *white* spaces. Pieces cannot be moved onto spaces dominated by the opposing player, unless attacking (discussed in Section 2.6), or if another piece is already on that space. An example of possible *non-attacking* directions are shown in Figure 2. These types of moves will be referred to in the rest of this paper as *non-attacking* moves.

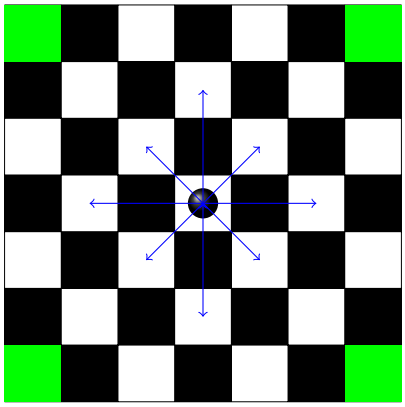

Figure 2: Example of possible movement directions (shown by the blue arrows) for a Black piece in the center of the board.

Despite not being an explicit rule of DOMINANCE, for the purposes of this paper we will not allow repetitions of game states. This is a key difference between DOMINANCE and our implementation of the game. The official game rules do not prevent a player from repeat game states if he/she chooses to do so. This is further described in Section 2.8.

2.4 Action: Spawning

When a player spawns a piece onto the playing field, one of the pieces from the players off-board pool of pieces is placed onto one of the spawn points in the corners of the board. A spawn point on which a piece is already placed cannot be used. A spawn point that is dominated by the opposing player can also not be used, see Figure 3. There is an exception to this rule: if the spawning piece is being used to attack a dominating enemy piece that spawn point can be used. This is discussed further in Section 2.6 and is shown in Figure 8.

2.5 Dominance

By moving or spawning a player can exert dominance over a space or several spaces on the game board. A space is dominated by a player when two of the surrounding spaces have pieces of that player on them. For pieces to work together to exert dominance they have to be to be diagonally next to each other or on opposite sides of the space over which they are exerting dominance.

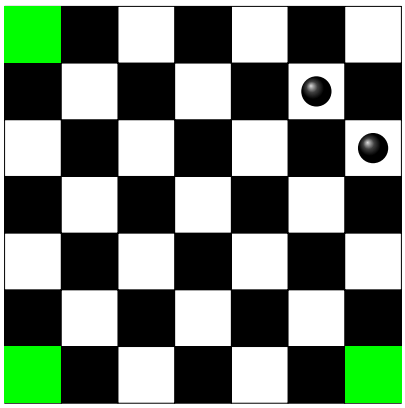

Figure 3: Example of available spawn points (marked green) when it is Red's turn and a spawn point (top right) has been dominated by the Black player. The Red player can spawn onto the green spaces.

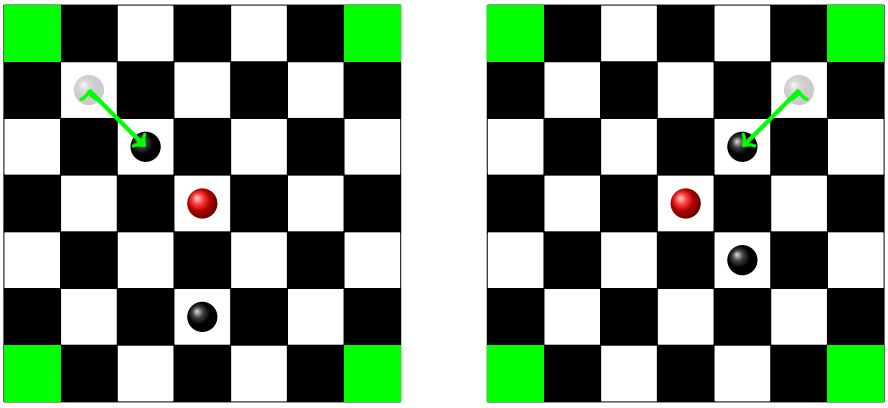

Figure 4: Examples of a space with Red player on it being surrounded from two sides by opposing player (Black) but not being dominated.

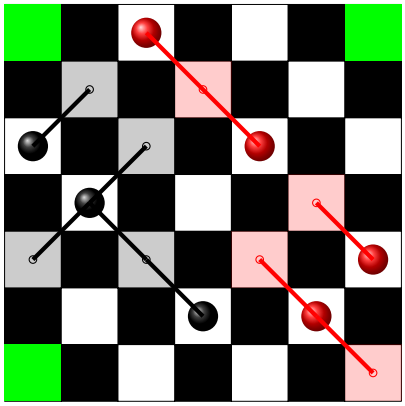

Figure 5: Example of spaces being dominated. The pieces connect by a line with a dot in the middle show pieces working together to exert dominance. The dot in the middle of a line represents the space being dominated which is also colored to mirror the player dominating that space.

2.6 Attacking

Exerting dominance over the opponents piece is also known as *attacking*. If a player's piece has been dominated by the opposing player that piece will be removed from the board and from the game. That piece will not be returned to the off-board pool of pieces and so cannot be spawned again. At the end of a turn there should no longer be any pieces in a space dominated by the opposing player. A piece can be moved into a space dominated by the opposing player if that piece is also attacking an opposing player's piece. When an attacking piece moves into a space dominated by the opponent, the attacked piece is removed but the attacking piece remains safe.

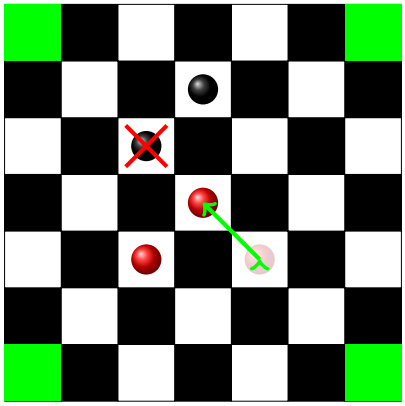

Figure 6: Example of attacking from a dominated space.

If after removing the attacked piece, the enemy is still able to exert dominance over the space the attacker moved into, the attacker is also removed from the gameplay.
Spawning can also also be a method of attacking. When spawning, dominance can be exerted over a space and then the same rules for attacking apply as described above. See Figure 8.

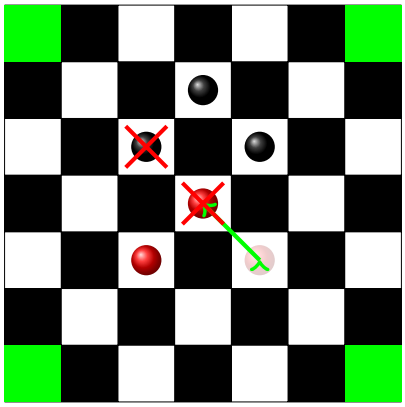

Figure 7: Example of attacking from a dominated space that is still dominated after the attack. Despite a Red piece having attacked a Black piece, the position the Red piece has moved onto is still dominated by the Black player. The attacking Red piece is dominated and removed from gameplay along with the dominated Black piece.

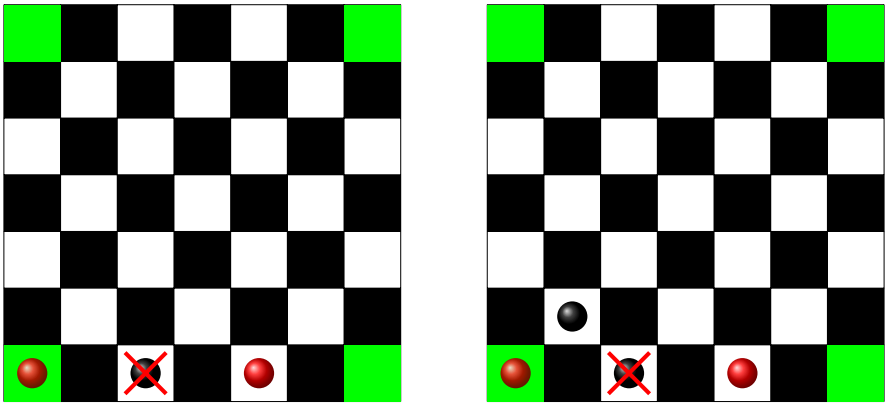

Figure 8: Examples of attacking from a spawning move (bottom left) for the Red player. A new piece is spawned into attacking at the bottom left space.

2.7 Winning the Game

There are several ways to win a game of DOMINANCE. A player has won the game when the opposing player can no longer exert dominance. One way to achieve this is by dominating nine of the ten opposing players pieces. The second way to achieve this is by dominating all but one of the opposing players on-board pieces and preventing the opponent from being able to spawn. A player cannot exert dominance over a space with less than two in-play pieces on the board.
For the purposes of this paper, if you can no longer perform any kind of move, you have also lost. This prevents draw situations. This is not explicitly an official game rule of DOMINANCE, but since we are going to block game states (see Section 2.8) it is important note how our implementation of the game is going to react if all possible moves are blocked.
The final way to win the game is to have your opponent concede. For the purpose of this paper we will assume that only humans would do this and will therefore not have any of our strategic agents used in this paper concede. The agents will keep playing until they have definitely lost.

2.8 Blocking Game State Repetitions

For the purposes of this paper we will maintain a list of all game board states reached during a game. When generating a list of possible moves, we will remove the *non-attacking* moves that lead to a game board state already present in the list of reached game board states.
If a player performs an *attack* or *spawn* move, the list of reached game board states is cleared. Since *spawing* or *attacking* significantly alters the game state, because pieces are added or removed, they are considered to be move types from which the game state cannot reverse.
With the implementation described in this Section we prevent the players from ending up in a loop of moves. An example of a blocking a game state is presented in Figure 9.
Note that this changes the gameplay in such a way that draws are now unlikely. This might not be anything like the real game of DOMINANCE since the official game rules do not prevent a player from repeating game states. In fact repeating game states might some times be considered part of a winning strategy. All of our strategic agents (described in Section 4) will not be allowed to repeat game states.

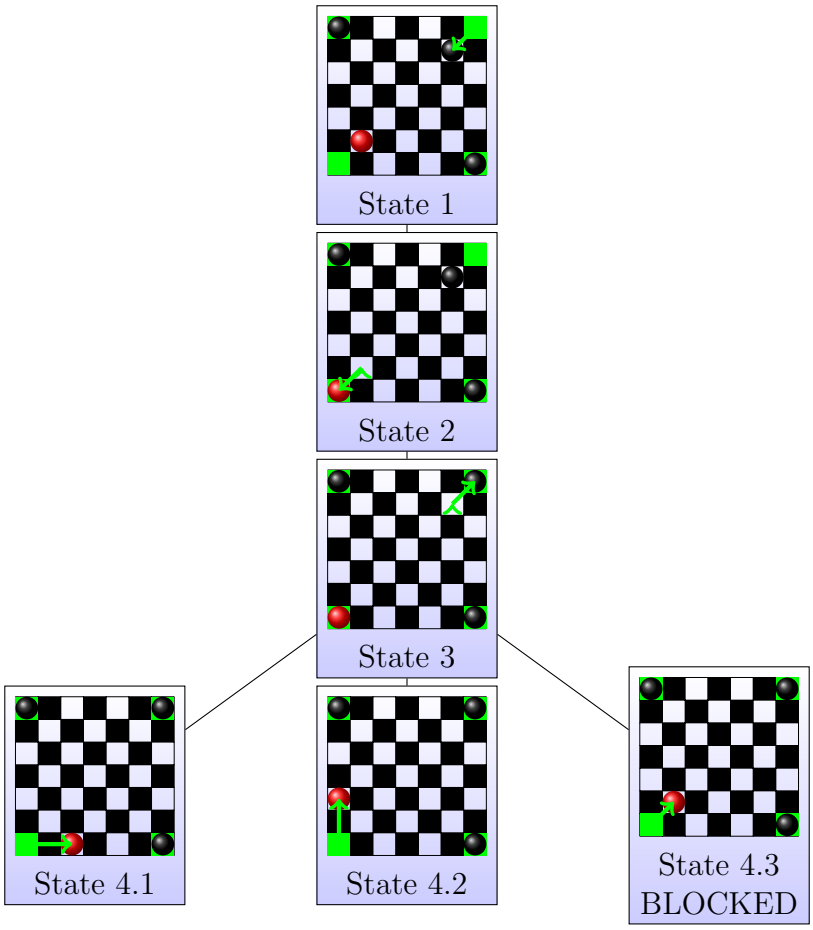

Figure 9: Example of a blocked state and so a move that is not allowed. State 4.1 would be exactly the same as the State that came before State 1. Since the list of states was not interrupted by a *spawn* or *attack* move, State 4.3 is blocked and the move leading to it not allowed since it would be the same board state as that before State 1. Despite being part of the game, for this example we do not show the number of spawned of captured pieces since it is not relevant to understand blocking game state repetitions. As discussed in this Section an *attacking* or *spawning* moves, which changes the number of spawned/captured pieces, represent changes in game state that cannot be reversed.

3 Related Work

The game DOMINANCE was developed by C. M. Perry and published by The Game Crafter in January 2012 [4]. The official game rules can be found at boardgamegeek.com and the game can be played online at tabletopia.com.
As of writing, no papers have yet been published on this game. Since no papers have been published on this game, we will divide this Section into three subsections looking at our implemented algorithms, search algorithms applied to games and theoretic considerations.

3.1 Search Methods

In this paper we will discuss several strategic agents, including Monte Carlo, Nega-max and MCTS. Russell and Norvig describe several algorithms and types of agents in their text book "Artifical Intelligence: a Modern Approach" [11]. Our implementation of Monte Carlo and Nega-max are based on the descriptions of these algorithms presented by Russell and Norvig.
In Browne et al. the workings of MCTS are described [2]. We used their descriptions of MCTS to create an implementation of MCTS for DOMINANCE. The function we use to evaluate nodes in our search tree is based on the Upper Confidence Bound described by Browne et al.. Our assumed constant value for this function are taken from Kocsis and Szepesvári's paper "Bandit Based Monte-Carlo Planning [9]. This constant value is used to balance exploitation of our tree search with exploration.
From the selected algorithms (described in Section 4) we will try to find the one that is best suited to DOMINANCE and discuss how to find the best variable values for each algorithm.

3.2 Search Methods Applied to Games

DOMINANCE is described as "Checkers for the 21st Century" [4] and can apparently easily be compared to checkers. The article *Checkers is Solved* [12] looks into solving the game of checkers by replacing a heuristic function with "perfection". The main differences between DOMINANCE and checkers are that the board is smaller and begins empty. The game mechanics and goals are described in Section 2.
A good example of a strategic agent that has really made a name for itself is Alpha Zero. Alpha Zero has been able to beat the world champion Go and Shogi [13]. Alpha Zero uses a combination of a MCTS algorithm which uses a Neural Network to apply value to a leaf node. Recently Alpha Zero has been adapted for Poker. Despite us not using the same type of agent we did choose to include Alpha Zero in this Section since we do implement a MCTS agent and Alpha Zero is a very popular strategic agent.

3.3 Theoretic Considerations and Games

In the article "Awari is Solved" Romein and Bal show that the (originally) African game has in total more that 800 billion positions in its state space [10]. Databases of "perfect play" for each stone position up to 48-stones were created with retrograde analysis, starting from the final position and searching backwards to the initial position. With theses databases, Romein and Bal were able to solve Awari and find that if both players were to play perfectly the game would end in a draw.

This is reflected in the game theoretic value, which represents the outcome of a game when both players play perfectly.
Schaeffer et al.'s article [12] was able to solve checkers, prove that there are about 500 million possible state positions and show that if both players were to play perfectly, checkers too would end in a draw. These articles show that both awari and checkers do not give one player an advantage or disadvantage. In this paper we will also look at an advantage the starting player has over their opponent in DOMINANCE.
"Games Solved, now and in the future" [6] looked into solving two-player divergent and convergent games. The article suggests decision complexity to be a more important factor when solving a game than state-space complexity and that knowledge based methods work best for games with low decision complexity while brute-force works best for games with low state-space complexity. It also states that with convergent games, like checkers, retrograde analysis has the "greatest impact on solving" the game. The same should apply to DOMINANCE but we do not consider retrograde analysis in this paper. This paper will mainly focus on knowledge based methods to solve dominance.
Despite this thesis not going into depth about the complexity of DOMINANCE, it is important to keep complexity in mind when comparing strategic agents. "Computing a Perfect Strategy for n x n Chess Requires Time Exponential in n" shows that for chess, another complex game, on a nxn board the time required to calculate a perfect strategy is at least exponential in n [5]. Exponential complexity is extremely costly when it comes to time.
Hoogeboom et al. suggest in their paper "Acyclic Constrain Logic and Games" the use of non-deterministic constraint logic framework to reduce the board game Dou Shou Qi, showing it to be PSpace-hard [7]. The reduction of games can be used to better understand and analyze games. In this paper we will not be using this technique.

4 Strategic Agents

In this section we will describe the different strategic agents we will compare while playing DOMINANCE. The different agents that will be investigated are random players, players using the Monte Carlo algorithm, a player using the Nega-max algorithm and finally players using a Monte Carlo Tree Search (MCTS) algorithm.
The interesting aspects of these different agents will be highlighted and we will describe how collected data can help us tune the initial algorithm variables to create a better strategic agent. The slightly altered strategic agent will then be analyzed in comparison to the initial strategic agent and other analyzed algorithms.
It is important to note that for each strategic agent the game is fully observable. This means that each player's "sensors give it access to the complete state of the environment at each point in time" [11, p. 42]. This complete state will include the position of every piece on the board, how many spawnable pieces are off the board for each player and whose turn it is. If needed the number of pieces that have been dominated and removed from gameplay can be calculated by subtracting the number of on- plus offboard (but still in play) pieces from the starting number of pieces, in this case 10.

The goal of these different strategic agents is to help us find which pieces of information provided in a fully observable state combined with which types of moves increase our chances of winning the game or delaying our inevitable defeat.

4.1 Random Players

Our simplest agent is the Pure Random player. This agent creates a list of all possible moves and randomly chooses one of these moves, with all moves being equally likely. We assume that the game is not biased towards the Black or Red player and will therefore assume that neither player is in a (possible) winning state at the start of the game. In Section 5.2 we will show that the game is, based on results from playing two Pure Random players against each other, statistically unbiased at the start of a game for Pure Random players but there is a Scholar's mate for DOMINANCE. Of course, as our strategic agents become more sophisticated, the tables are likely to tip towards the more sophisticated agent.
As players spawn pieces onto the board their possible moves will probably begin to favor *non-attacking* of onboard pieces over *spawning*. Since each onboard piece can move in up to eight different directions and the number of spawn points being at most four, the further into the game the more likely the Pure Random player will choose to move an onboard piece over spawning a piece onto the board.
The Pure Random Player treats all possible moves as equal, but not all *types* of moves as equal. As the list of possible moves is filled with *movement moves*, the *spawn moves* become a minority, limiting the agent since it will become less likely to spawn new pieces onto the board. This is also likely to apply to *attack moves*. This might lead to undesirable gameplay (despite state repetition not being allowed, as described in Section 2.8) or suboptimal gameplay. To alleviate this problem a weighted or parameterized Random player was implemented: Parameterized Random.
We have defined three categories of moves: *spawn*, *attack* and *non-attacking* moves. The Parameterized Random player will divide all possible moves into lists based on these categories; one list

containing *attack* moves, the second containing *spawn* moves and the last containing *non-attacking* moves. Each list will be assigned a value γ, δ or ω respectively. These values will define a ratio and thus dictate the chances of a given type of move being selected. If a certain type of move is not available its ratio value is set to 0 and so equivalently dividing its selection chances over the other lists.
It is important to note that some moves can be classified as belonging to more than one list. Every *attack* move is either a *spawn* or *non-attacking* move. These moves will be placed in both lists. If we do not place *attack* moves in both associated lists we will actually be showing an active bias towards our *attack* list, actively reducing the value of our *non-attacking* and *spawn* lists. How to choose to avoid bias in a situation like this is tricky and will be discussed further in Section 6.
Since each *attack* move is part of either our *spawn* moves list or our *non-attacking* moves list we would ideally define $\delta > 0$ and $\omega > 0$ so that all possible moves are allowed and the player will not get stuck due to not being able to choose a possible move.
This might not be desirable for testing so certain exceptions are made. If there are two lists to choose from of which both associated values are 0, the chances one of these lists will be selected will be 50-50. If, based on associated value, we have to select from two empty lists we will randomly choose a move from the third and non-empty list. This should avoid a situation where the player has to choose a non-existent move or from an empty move list.
By comparing these Random agents, we hope to try and find the types of moves that significantly increase our chances of winning. The best selected Random agent will later be used in our Monte Carlo and MCTS algorithms.

4.1.1 Choosing Values for the Parameterized Random Player

Finding the ideal ratio can be emperically established. Of course we assume that a higher value for γ and δ over ω would increase our chances of winning but ω might be more important than we expected. To try and find the ideal values for γ, δ and ω we tried all combinations of $0 \leq \gamma < 6$, $0 \leq \delta < 6$ and $0 \leq \omega < 6$ for our Parameterized Random player as Red against a Black Basic Monte Carlo player (described in Section 4.2) where $k = 10$. There will be $n = 10$ games played for each value of γ, δ and ω. Our values for k and n are extremely low in this case, but this is due to time limitations and will be further discussed in 6.
We will choose our value for ω which returned the most perfect win rate (where 10 out of 10 games are won for that value ω) for all combinations of $0 \leq \gamma < 6$ and $0 \leq \delta < 6$. Our values for γ and δ will be selected based on the values γ_ω and δ_ω at the center of the largest perfect win rate cluster for our selected value ω.
The results of this experiment are presented in Section 5.3.1 with the "ideal" combination being $\gamma = 4$, $\delta = 2$ and $\omega = 0$. This describes a ratio of $4 : 2 : 0$ for $\gamma : \delta : \omega$ (attack : spawn : movement). A *non-attacking* move will thus only be selected if there is no possible *spawn* or *attack* move available.

4.2 Monte Carlo Players

The second strategic agent to be implemented is the Monte Carlo player, using a randomized sampling algorithm. This agent uses Monte Carlo game search to select a move with the highest probability of winning based on direct sampling of each move [11, p. 530]. It does this by playing k

random games for each possible move, where a Pure Random player plays against another Random player. The agent then selects the move where, based on the k randomly played games per move, it won the most times and thus hopefully the move with which it is most likely to win. Since the game cannot end in a tie, the Random players will always return a win or a loss.
As we will see in Section 5.3, Parameterized Random player (described in Section 4.1) generally wins against a Pure Random player and reduces the average number of performed moves. Since γ, δ, ω-Random has proven itself a better (and quicker) strategy than Pure-Random, we will also implement an Parameterized Monte Carlo (also referred to as Parameterized Monte Carlo) agent which uses Parameterized Random for its k simulations instead of Pure Random. This way we hope to reduce the amount of work done by Monte Carlo by reducing work done during simulations while making the simulations more effective.
With these agents we will experiment with different values for k. A visualization of this algorithm is presented in Figure 10.

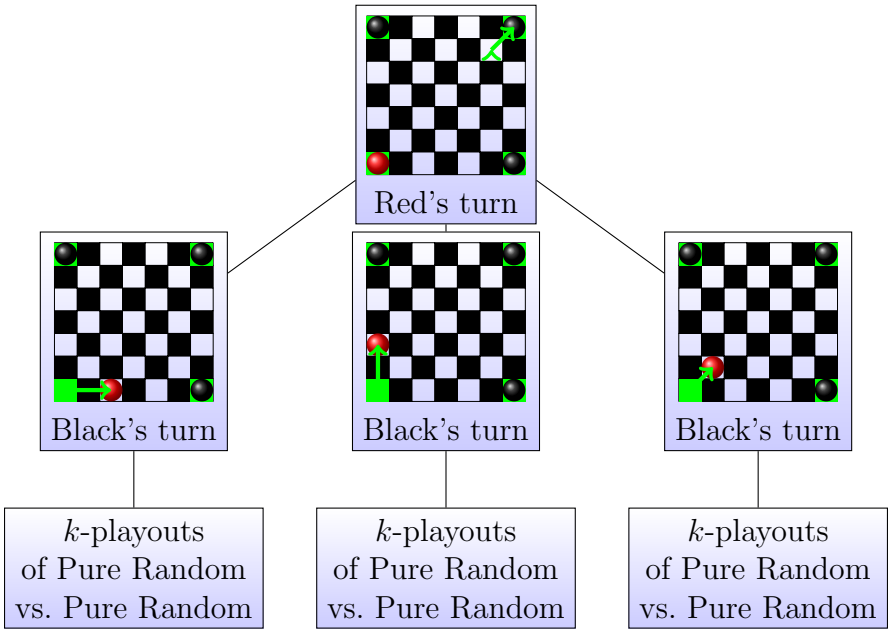

Figure 10: Our Basic Monte Carlo algorithm. In this case the Red player will expand every directly possible move and perform k random games for every move. The Red player will then select the move which returned the greatest win rate out of the k games played.

4.3 Nega-max Player

The third strategic agent will follow the Nega-max algorithm which is an alteration on the mini-max algorithm [11, p. 165]. When using Mini-max or Nega-max every gamestate is assigned a certain heuristic value. The goal of the player is to choose the move that is able to increase their own heuristic value as much as possible and so decrease the opponents heuristic value. Mini-max "uses a simple recursive computation of the minimax values [heuristic values] of each successor state, directly implementing the defining [heuristic] equations" [11, p. 165]. This process is followed down

a game tree containing all possible moves and the states they lead to. This tree goes down to a predefined depth d or until it reaches a leaf node, which is an end-state.
When using Nega-max the goal, exactly like mini-max, is to increase your own utility as much as possible. The utility of each state is defined by a heuristic function, which is explained further in Section 4.3.1. The only difference is that Nega-max does not need to define a function for both players but instead just takes the negative value of the heuristic of the opposition, following a decision tree as presented below in Figure 11.

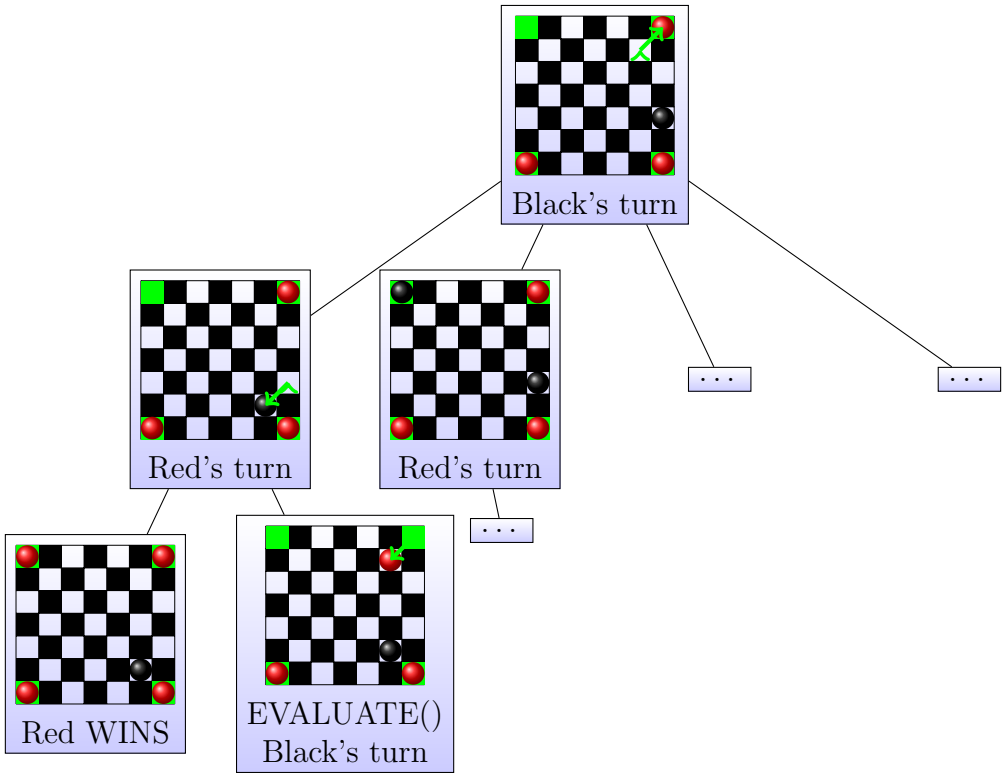

Figure 11: Nega-max tree for depth $d = 2$. Our Red player will always choose the move which is able to increase its heuristic value the most (which in turn reduces the Black players heuristic value the most). The Black player will not allow this and chooses another sub-tree. The green arrows show the move taken to get to this state.

4.3.1 Defining the Heuristic Function

Based on our results in Section 5.3 we assume that having many pieces on the game board increase your chances of winning, so does spawning and attacking more often than your opponent. The heuristic value should accurately reflect a game situation. A high heuristic value will reflect a high number of your onboard pieces and a low number of your opponent's onboard pieces.
We decided to use this to define our simple heuristic value h for a given game state as

$$h = (my\ pieces\ on\ board\ *\ 3) - (opponents\ pieces\ on\ board\ *\ 5)$$

By assigning your opponents pieces a value with an absolute value higher than your own, you stimulate attack moves or blocking spawn moves. This heuristic definition should give a balance between stimulating spawn and attack moves by favoring reducing the number of pieces your opponent has on the board while stimulating you to spawn your own pieces.

4.4 Monte Carlo Tree Search Player

Our third strategic agent will employ a Monte Carlo Tree Search algorithm to choose the most appropriate move. This algorithm combines a Monte Carlo algorithm, as described in Section 4.2, with an exploration and exploitation technique. We end up building a tree of states from moves, expanding the tree nodes based on random sampling and exploration versus exploitation equation [2]. Kocsis and Szepesvári [9] suggested that we should expand the node with the highest value for

$$\left(\frac{w_i}{n_i}\right) + c \cdot \sqrt{\frac{\ln(N_i)}{n_i}},$$

where w_i is the number of wins associated with the node after the i-th move for that node, n_i is the number of times the node was looked at after the i-th move, N_i is the number total number of simulations down the tree, and $c = \sqrt{2}$ representing the exploration parameter.
The first part of the equation represents the win ratio for the node being looked at and the second part represents the stimulation to further explore other moves. The results of an explored node is propagated back up the tree to the root node.
Starting at the root node, which represents the current game state, we use the formula described above to follow the tree down to a leaf node. This leaf node will then be expanded further, selecting one of the possible moves from that game state randomly.
When testing a leaf node we will use a Parameterized Random agent since our Parameterized Random agent is able to beat our Pure Random agent. Due to time limitations we chose not to implement a MCTS agent that uses Basic/Parameterized Monte Carlo or Nega-max for simulations. Since our MCTS agent using Parameterized Random for simulations out performs all of our other agents (see Section 5.6), we will also try to see if it still performs better while only running simulations till a given depth, using the Nega-max heuristic function to assign a value to a node that is not an end-state. This will greatly reduce processing time while still allowing for a reliable MCTS agent.

5 Results

In this chapter we will have the different agents described in Section 4 play against each other. Each agent will play n games against itself and other agents, where n will be defined based on the type of agents being played against each other and our time constraints during experimentation. For each experiment, the agents will play the predefined n games as both the Red and Black player against their opponent.
During experimentation we will collect data (discussed in Section 5.1) and use this data to deduce effectiveness of one agent over the other. This data will also be used to help make decisions when defining pre-set variables for different agents. For example, how to decide on the k-value used when experimenting with Monte Carlo, if Monte Carlo should use our Parameterized Random players instead of a Pure Random player for simulations and how to define values for the $\gamma\delta\omega$ ratio.
By combining all of this information we will try to understand why certain agents perform better than others and which type of move will help increase an agent's chance of winning.

5.1 Collected Data

During experimentation we collect several statistics for every agent over the predefined n games. These include the number of wins, average number of performed moves, average of total number of possible moves during a games, maximum number of possible moves from one state, minimum number of possible moves from a state, average number of performed spawn moves, average number of performed attack moves, and finally the minimum and maximum number of performed moves before an end-state was reached.
It is important to note that when we collect data on the minimum number m of possible moves to be performed, we only look at values where $m > 0$, because $m = 0$ corresponds to an end-state. Being a characteristic of a lost game, we will not include this in our findings.
When analyzing the results of the experiments we will refer to the collected data we deem relevant to the given agents being analyzed. The collected raw data will not be presented this paper itself.

5.2 Pure Random Player

The first thing we can do with our Pure Random agent is test if there is a Red-Black discrepancy, where the starting player (Red) might be at an advantage over the agent playing as the Black player. This can be done by playing two Pure Random agents against each other and see if the Red player wins significantly more often than the Black player. Or the other way around, in which case the Black player has an inherent advantage over the Red player.
For this experiment we define $n = 1,000,000$.

Table 5.1: 1,000,000 games Pure Random vs. Pure Random. Results when a Pure Random agent plays 1,000,000 games against another Pure Random agent.

	Pure Random (Red)	*Pure Random (Black)*
wins	503,370	496,630
pref. moves	16.777	16.777
nr. pos. moves	545.862	545.472
max. nr. pos. moves	46	44
min. nr. pos. moves	1	1
nr. perf. spawn	9.522	9.510
nr. perf. attack	7.017	7.005
min. nr. perf. moves	4	3
max. nr. perf. moves	318	318

By using a frequentist approach we can calculate the confidence interval for a proportion [1]. With this data we can then see if the win rate for Red hints at a bias towards the Red player.
Assuming an unbiased game will, like tossing a coin, have a 50-50 playout between two pure random agents, we use a one-sample, one-tailed t-test with a confidence level of 99% to see if the difference between our expected win rate and our observed win rate is significant.
A t-test does make some assumptions [1]. First, this test assumes the data is obtained using randomization. Since a new game is played per simulation and both agents use randomization, we assume this to be true. Secondly, a quantitative variable is being measured and finally that this variable is assumed to have a normal population distribution. Our quantitative variable is number of games won and we assume a normal population distribution of the sampling distribution due to the Central Limit Theorem [1].
After performing our significance test can conclude with a 99% confidence level that there is a significant bias in favor of the Red player at the start of the game. This shows that there is a difference in who starts the game when playing two Pure Random agents against each other: Red appears to have a slight advantage over Black. Statistically significant, this discrepancy can also be noticed by the minimum number of performed moves (4 for Red and 3 for Black).

5.2.1 Dominance's Scholar's Mate

Red can both win and lose the game after Red has performed 4 moves and this allows us to find the quickest winning strategy. This strategy focuses on spawning onto the spawn points while the opposition only spawns once and after that moves their one spawned piece around the board. By blocking all of the spawn points while the opposition only has one piece on the board, the opposing play becomes unable to exert dominance and loses.
Figure 12 gives an example of the Red player using the quickest winning strategy. This helps us understand why the Red plays is observed to have a slightly higher win rate than the black player. By following the game presented, we can calculate the chances of a player selecting that type of move. It should be noted that if no other moves are possible than *spawn* moves, the probability of performing a *spawn* move is 1. The probability of Red's first move being a *spawn* move is 1 and the same counts for Black's first move.
Using this logic we can multiply the probability of the selected move being performed to calculate

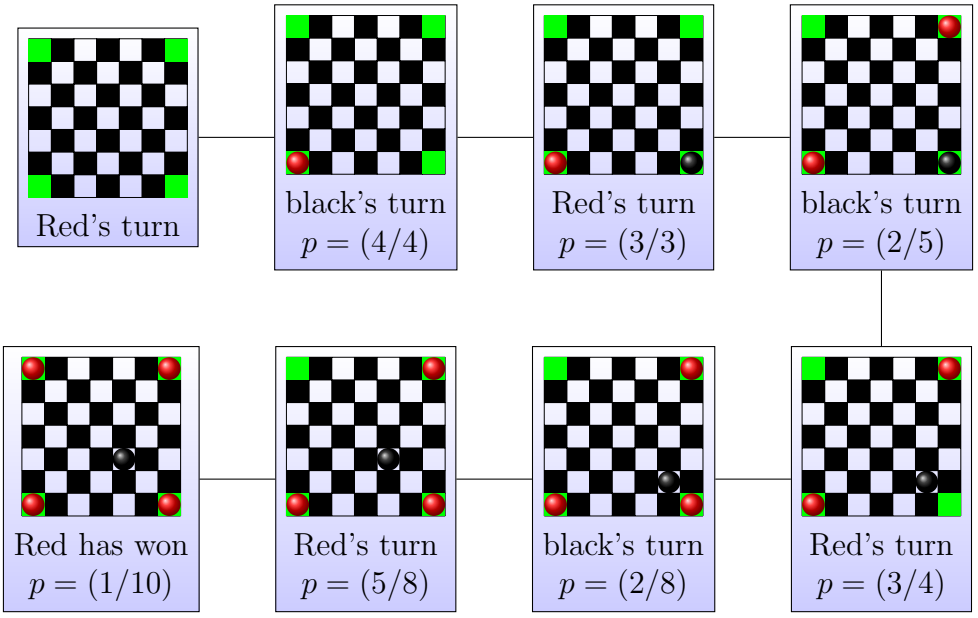

Figure 12: Example of the quickest way to win for the Red player. This is equivalent to the Scholar's mate in chess. The game starts top left, with an empty game board and it being Red's turn. Each state shows whose turn it is and the probability p of the last type of move being performed out of all moves that were possible. The Red player wins in four turns by blocking all of the spawn points while the Black player only has one piece on the board.

the chances of DOMINANCE'S version of the Scholar's mate when playing two Pure Random Players against each other. The final game state presented in a winning state for Red since it is Black's turn while Black only has one piece on the board and all spawn points are blocked by Red, see Section 2.7.
The chances of this occurring as described above is $\frac{4}{4} * \frac{3}{3} * \frac{2}{5} * \frac{3}{4} * \frac{2}{8} * \frac{5}{8} * \frac{1}{10} = 0.00469$, which is 0.469%. This perhaps explains the slight advantage the Red Player appeared to have when playing two Pure Random Players against each other.
There are of course alterations which slightly change the probability, for example if the one Black piece were to move up the side of the board instead of towards the center the board. For the purposes of this paper we will only provide this example and not go in on other examples.

5.3 Parameterized Random Player

The first step when testing the Parameterized Random agent is to define its values for γ, δ and ω. To try and find the optimum values we will test all combinations of $0 \leq \gamma < 6$, $0 \leq \delta < 6$ and $0 \leq \omega < 6$, integer values, against an agent using Basic Monte Carlo as described in Section 4.2.
The next step will be to test this agent (with our defined values for γ, δ and ω) against our Pure Random player, described in Section 4.1.

5.3.1 Finding Values for Weights

To find the "ideal" ratio of γ, δ and ω we will play $n = 10$ games for all combinations of $0 \leq \gamma < 6$, $0 \leq \delta < 6$ and $0 \leq \omega < 6$ against a Basic Monte Carlo player where $k = 10$. Our Parameterized Random agent will assume the role of the Red player and the Black player will use Monte Carlo. This gives our Parameterized Random agent a very slight inherent advantage as discussed in Section 5.2, but since we *assume* Monte Carlo to be the more strategic agent we did not see this as a problem. Due to time restrictions we have chosen a very low value for both k and n.
We have divided our data collection into 6 groups with different values for ω. So ω is a constant in each group, while γ and δ vary between $0 \leq \gamma < 6$ and $0 \leq \delta < 6$. We will then select the value for ω whose group had the highest number of perfect wins (10 out of 10 victories for varrying combinations of γ and δ). From this group the values γ and δ will be selected that is at the center of the largest cluster with radius 2.

Table 5.2. The number of perfect wins (10 out of 10 victories for combinations of γ and δ) for given constant values of ω and varying values of $0 \leq \gamma < 6$ and $0 \leq \delta < 6$.

Value Gamma	*Number of perfect wins*
0	27
1	12
2	7
3	4
4	3
5	1

From the results in Table 5.2, we can conclude that with $\omega = 0$ the Parameterized Random player has the highest win rate. This is equivalent to always choosing a *spawn* or *attack* move over a *non-attacking* move. The next step is to find reliable values for γ and ω.

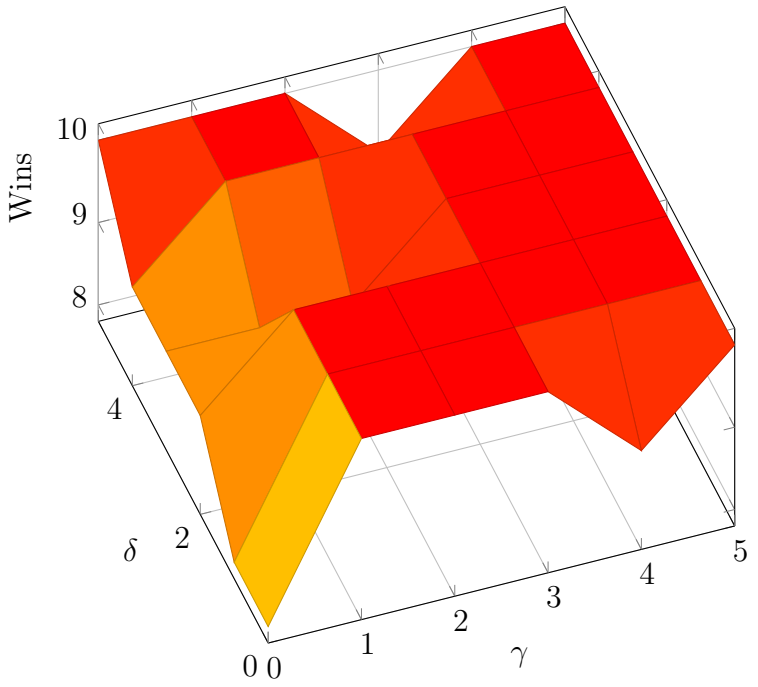

Figure 13: 3D plot of the number of wins out of 10 (y-axis) for varying values of $0 \leq \gamma \leq 5$ and $0 \leq \delta \leq 5$ while $\omega = 0$.

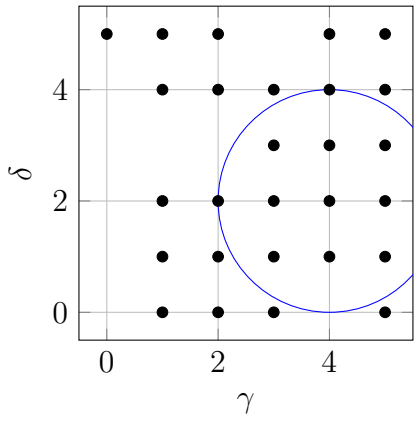

Figure 14: Scatter plot of perfect wins (10 out of 10 victories for combinations of γ_ω and δ_ω) for given constant values of ω and varying values of $0 \leq \gamma < 6$ and $0 \leq \delta < 6$. The circle marks a radius of 2 around the combination of values $\gamma = 4$ and $\delta = 2$ which is the center of the largest winning cluster.

Based on the data in Figure 13 and Figure 14, we will choose our values $\gamma = 4$, $\delta = 2$ and $\omega = 0$.

5.3.2 Parameterized Random vs. Pure Random

With our values of $\omega = 0$, $\gamma = 4$ and $\delta = 2$ we can start comparing our agent to itself and our Pure Random player.

Table 5.3: 1,000,000 games Pure Random vs Parameterized Random. Results when playing a Pure Random agent against a Parameterized Random Agent. 1,000,000 games were played with Pure Random as the Red player and another 1,000,000 games with Pure Random as the Black player.

	Pure Random (Red)	*Parameterized Random (Black)*
wins	7	999,993
pref. moves	7.652	7.652
nr. pos. moves	50.402	82.784
max. nr. pos. moves	28	44
min. nr. pos. moves	1	2
nr. perf. spawn	2.109	5.393
nr. perf. attack	0.087	1.246
min. nr. perf. moves	4	4
max. nr. perf. moves	71	71
	Parameterized Random (Red)	*Pure Random (Black)*
wins	999,998	2
pref. moves	7.005	6.005
nr. pos. moves	74.330	35.568
max. nr. pos. moves	43	26
min. nr. pos. moves	2	1
nr. perf. spawn	5.103	1.820
nr. perf. attack	0.932	0.065
min. nr. perf. moves	4	3
max. nr. perf. moves	62	61

From the data in Table 5.3 it is clear that our Parameterized Random player performs better than our Pure Random player. Our Parameterized Random agent reduces the total number of performed moves, from an average of 16.78 in Section 5.2, to around 7.65 when playing as Black and 7 when playing as Red against a Pure Random agent.
This Parameterized Random agent focuses on spawning and attacking. This way it spawns as many pieces as needed (on average around 5) and limits the number of *attack* moves needed (on average around 1) per game.

Table 5.4: 1,000,000 games Parameterized Random vs Parameterized Random. Results of 1,000,000 games of Parameterized Random against Parameterized Random.

	Parameterized Random (Red)	*Parameterized Random (Black)*
wins	470,314	529,686
pref. moves	16.777	16.777
nr. pos. moves	299.906	307.254
max. nr. pos. moves	42	40
min. nr. pos. moves	1	1
nr. perf. spawn	8.361	8.511
nr. perf. attack	5.342	5.464
min. nr. perf. moves	6	6
max. nr. perf. moves	90	89

Based on our results from Table 5.4 we can notice that our Parameterized Random agent has an advantage over itself when playing as Black. By performing the same statistical test as in Section 5.2 we know that the discrepancy between the win rate and our expected win rate for both players (500000) is most likely not due to chance. When playing two Parameterized Random agents against each other, the Black player has an inherent advantage.

5.4 Monte Carlo Player

Similar to our Parameterized Random agent, before experimenting with our Basic Monte Carlo agent (described in Section 4.2) agent we have to define our parameters, in this case k for both Monte Carlo agents. After having defined our value $k_{basicMC}$ for the Basic Monte Carlo agent we can continue to define another paramenter $k_{Par.MC}$ for our Monte Carlo agent using Parameterized Random.
We can then continue to compare our Pure Random, Parameterized Random, Basic Monte Carlo and Parameterized Monte Carlo agents.

5.4.1 Defining k for our Monte Carlo Agents

To try and find an ideal value for k we played $n = 50$ games of Basic Monte Carlo against our Parameterized Random ($\gamma = 4$, $\delta = 2$, $\omega = 0$) for values $k \in \{10, 25, 50, 75, 100, 125, 150, 175, 200, 250, 300\}$. This same process is repeated for our Parameterized Monte Carlo agent. We chose to use a relatively low value for n due to time constrictions.
With this data we will try to find the lowest viable value for k for both Monte Carlo agents.

The data from Figure 15 shows a clear trend: the higher the value for k, the higher the win rate for our Basic Monte Carlo agent against an Parameterized Random agent. Due to time limitations we chose to define $k_{basicMC} = 200$ for our Basic Monte Carlo agent. With $k_{basicMC} = 200$ the Basic Monte Carlo agent wins more than 50% of the time against our Parameterized Random agent.
Similar to what we did in Section 5.2 we can uses a t-test to see if the slightly higher win rate is significant. Based on these statistics we can say with 99% certainty that Basic Monte Carlo might be able to beat our Parameterized Random agent, but Parameterized Random might still be beter. Simply put, our sample size is too small to definitively conclude that Basic Monte Carlo performs

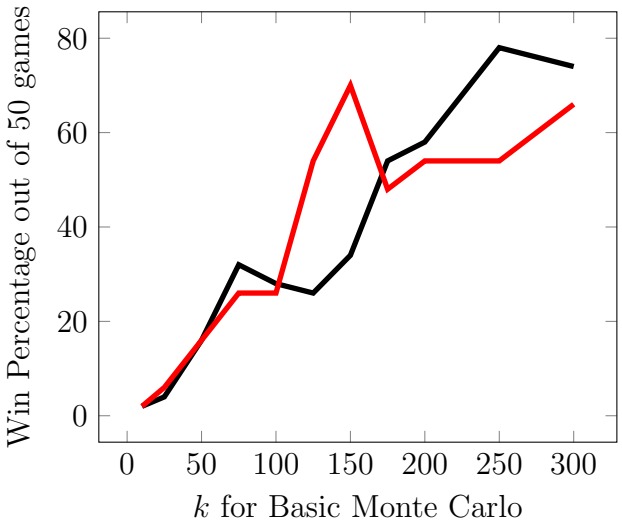

Figure 15: Plot showing the win percentage out of 50 games for varying values of k for Basic Monte Carlo against Parameterized Random ($\alpha = 4$, $\beta = 2$, $\gamma = 0$). ━ represents the results when Monte Carlo was Black and ━ when Monte Carlo was Red.

better than Parameterized Random, or vice versa. That $k_{basicMC} > 200$ is shown to perform better is discussed in Section 6.

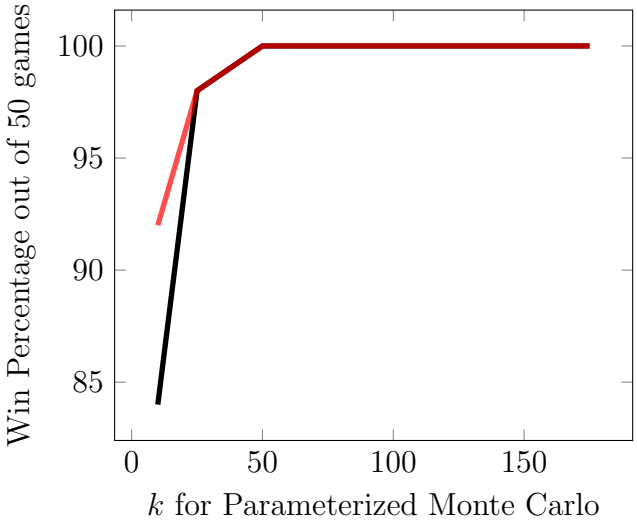

Figure 16: Plot showing the win percentage of 50 games for varying values of k for Parameterized Monte Carlo against Parameterized Random ($\alpha = 4$, $\beta = 2$, $\gamma = 0$). ━ represents the results when Parameterized Monte Carlo was Black and ━ when Parameterized Monte Carlo was Red.

From Figure 16, we see that our Parameterized Monte Carlo agent clearly performs better than our Basic Monte Carlo agent. Even with a low value of $k_{Par.MC}$ this agent still has a win rate of above

80% against our Parameterized Random agent.
Since our Basic Monte Carlo agent with $k_{basicMC} = 200$ has a win rate of more than 50% against the Parameterized Random player, we will also test our Parameterized Monte Carlo agent for varying values of $k_{Par.MC}$ against Basic Monte Carlo with $k_{basicMC} = 200$.

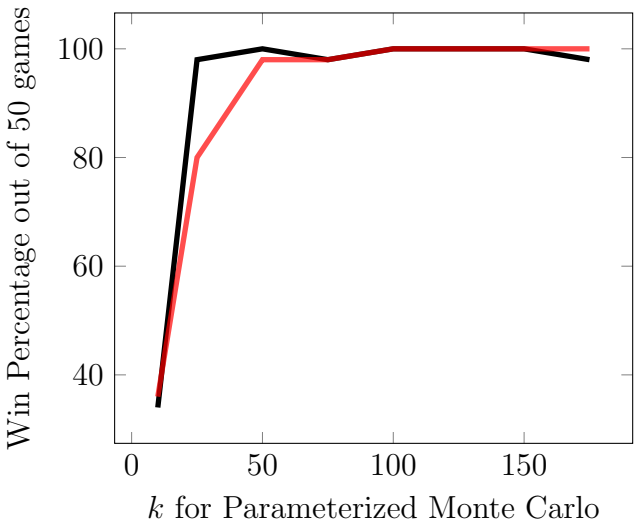

Figure 17: Plot showing the win percentage of 50 games for varying values of k for ABG-Monte Carlo against Basic Monte Carlo with $k_{basicMC} = 200$. ━ represents the results when Parameterized Monte Carlo was Black and ━ when Parameterized Monte Carlo was Red.

The data presented in Figure 17 suggests that Parameterized Monte Carlo is generally able to beat Basic Monte Carlo if $k_{Par.MC} \geq 50$. Basic Monte Carlo is sometimes able to beat Parameterized Monte Carlo, even if $50 \leq k_{Par.MC} \leq 175$.
Based on the results of playing Parameterized Monte Carlo against our Parameterized Random and Basic Monte Carlo agent, we can be fairly certain that our Parameterized Monte Carlo will be able to beat both other agents with $k_{Par.MC} = 100$.

5.5 Nega-max Player

Our Nega-max agent actively uses a heuristic function (described in Section 4.3.1) to make choices as to which move can lead to a winning state.
For this agent we will experiment with different search depths d for our Nega-max agent against our Parameterized Monte Carlo agent. The results of these experiments are presented in this Section. Based on these results we will try to find the minimum depth d needed for our Nega-max agent to always be a winning agent against our (parameterized) Random and (parameterized) Monte Carlo agents.
From the data in Figure 18 we can conclude that the higher our value for d the better Nega-max is able to perform, and that our Parameterized Monte Carlo agent is also generally able to beat

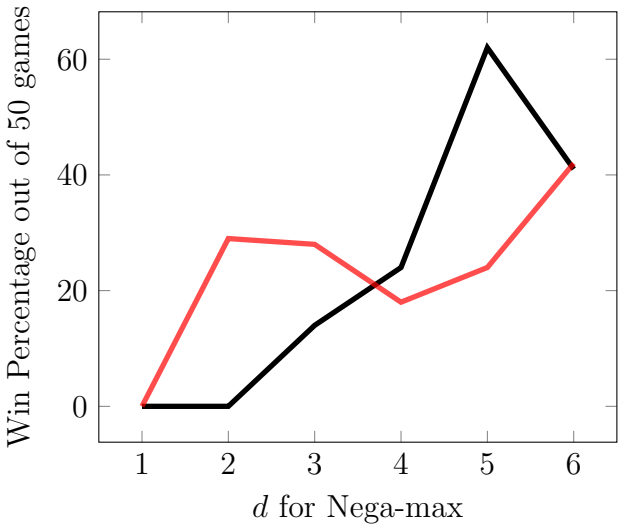

Figure 18: Plot showing the win percentage of 50 games for Nega-max with varrying values for search depth d against Parameterized Monte Carlo with $k_{Par.MC} = 100$. ━ represents the results when Nega-max was Black and ━ when Nega-max was Red.

Nega-max with $d \leq 6$ for more than 50% of the games. The exception is $d = 5$ when the win rate of our Black Nega-max agent sharply increases. The data is presented in Table 5.5.

Table 5.5: 50 games Nega-max($d = 6$) vs Par. Monte Carlo. Results of playing Parameterized Monte Carlo against Nega-max. 50 games where played with Nega-max as Black and another 50 games were played with Nega-max as Red.

	Nega-max(Red)	*Par. Monte Carlo (Black)*
wins	21	29
pref. moves	55.5	55.08
nr. pos. moves	713.24	800.86
max. nr. pos. moves	36	32
min. nr. pos. moves	2	1
min. nr. perf. moves	12	11
max. nr. perf. moves	164	164
	Par. Monte Carlo (Red)	*Nega-max (Black)*
wins	29	21
pref. moves	57.6	57.02
nr. pos. moves	892.9	693.16
max. nr. pos. moves	32	32
min. nr. pos. moves	1	1
min. nr. perf. moves	13	13
max. nr. perf. moves	198	197

Due to time limitations we were not able to test values for $d > 6$. We can see from Table 5.5 that the average number of possible moves from any state visited during experimentation by our Nega-max agent with $d = 6$ is 12.6. This is found by dividing the number of possible moves by the number of performed moves for both Nega-max as Black and Red. The average was then taken from these two statistics.
For Nega-max with $d = 6$ this already suggests on average visiting roughly $12^6 \approx 3 \cdot 10^6$ states before choosing a move to perform. The large number of possible states to visit in a decision tree explored by Nega-max forces us to choose a relatively low value for d due to time restrictions. We will choose $d = 3$ for further experimentation with Nega-max. This limitation is further explored in Section 6. As we can see from Figure 19 our Nega-max agent with $d = 3$ is always able to beat both

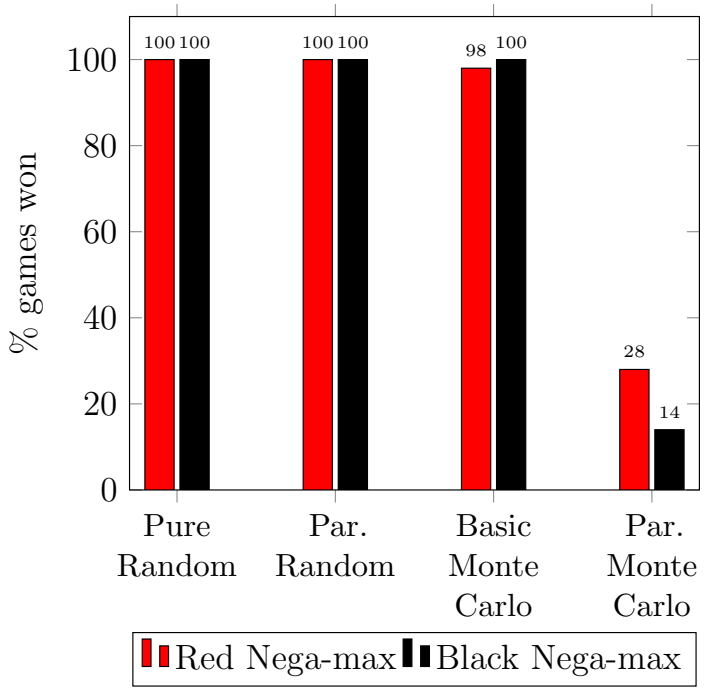

Figure 19: Bar graph showing win percentage (out of 50 games) for Nega-max with $d = 3$ as Black and Red against our (Parameterized) Random and (Parameterized) Monte Carlo agents.

our Random agents and almost always beats our Basic Monte Carlo agent. The Parameterized Monte Carlo agent is generally able to beat Nega-max.

5.6 Monte Carlo Tree Search Player

During experimentation with MCTS we will start by trying to find an appropriate value for c and a suitable total number of simulations needed for MCTS, k_{mcts}. We will do this by playing our MCTS agent using Parameterized Random for simulations against our Nega-max with $d = 3$ agent to try and define k_{mcts}. We are looking for an appropriate value k_{mcts} which is able to perform as well as or better than our Parameterized Monte Carlo agent does against Nega-max with $d = 3$.
We will first play our MCTS agent against Nega-max to try and define $k_{mcts} \leq 1800$ while $c = \sqrt{2}$. Since our Parameterized Monte Carlo agent has to investigate on average 17.8 moves per turn

against Nega-max with $d = 3$ and it performs $k_{Par.MC} = 100$ playouts per move, the maximum we will consider as a suitable value for k_{mcts} is $k_{mcts} = 17.8 \cdot 100 = 1780 \approx 1800$.
To fine-tune our value for c we will play our MCTS agent using Parameterized Random for simulations against our Parameterized Monte Carlo player with our already defined value for k_{mcts}. Since our MCTS agent using Parameterized Random for simulations out performs all of our other agents, we will also try to see if it still performs better while only running simulations till a given depth, using a heuristic function to assign a value to a node that is not an end-state. This will greatly reduce processing time while still allowing for a reliable MCTS agent.

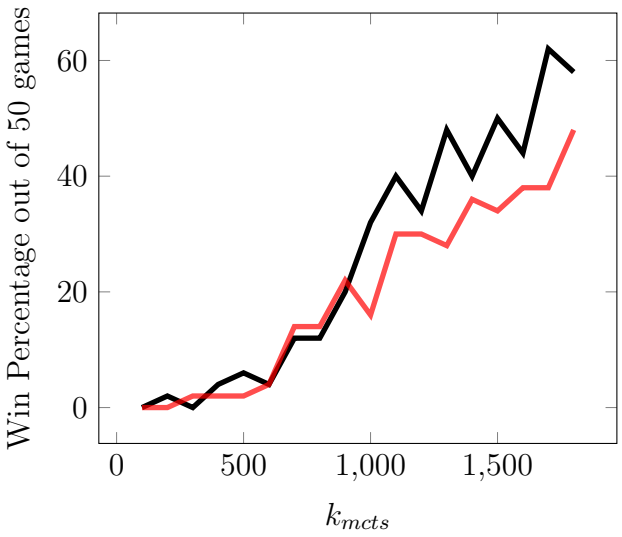

Figure 20: Plot showing the win percentage of 50 games for MCTS using Parameterized Random for simulations with varrying values for k_{mcts} against Nega-max with $d = 3$. ━ represents the results when MCTS was Black and ━ when MCTS was Red.

Figure 20 shows that the higher our value for k_{mcts} the higher the win rate for our MCTS agent against Nega-max $d = 3$. Based on this data we will define our $k_{mcts} = 1800$. Since we know that Parameterized Monte Carlo is better than our Nega-max agent, we will not test MCTS against Parameterized Monte Carlo.
When trying to define our value for c we will play our MCTS agent using Pure Random for simulations against our Basic Monte Carlo player with our already defined value for $k_{mcts} = 1800$. This is to try and find a balance between exploration and exploitation.
From the data in Figure 21 we notice a significant difference between focusing on exploitation over exploration. The lower our value for c the higher the win rate from MCTS against negamax. The highest win rate was with $c = 0.2$ where MCTS had a win rate of up to 90% when Black and 82% when Red.
With our defined values for $k_{mcts} = 1800$ and $c = 0.2$ we will test our MCTS agent, using Parameterized Random during simulations, against our Parameterized Monte Carlo agent. The results of this experiment are presented in Table 5.6.

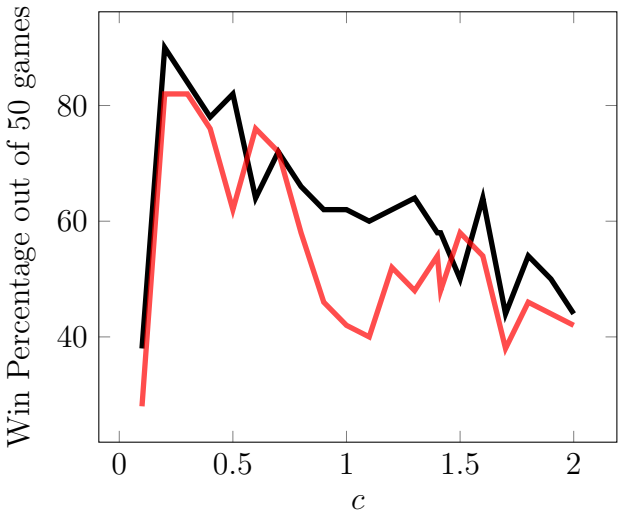

Figure 21: Plot showing the win percentage of 50 games for MCTS using Parameterized Random for simulations and $k_{mcts} = 1800$ with varrying values for c against Nega-max with $d = 3$. ━ represents the results when MCTS was Black and ━ when MCTS was Red.

Table 5.6: 50 games MCTS vs Parameterized Monte Carlo. Results when playing MCTS against Parameterized Monte Carlo. 50 games were played with MCTS as the Red player and another 50 with MCTS as the Black player.

	MCTS (Red)	*Parameterized Monte Carlo (Black)*
wins	29	21
pref. moves	60.02	59.44
nr. pos. moves	1102.74	714.58
max. nr. pos. moves	41	30
min. nr. pos. moves	1	1
nr. perf. spawn	9.54	9.4
nr. perf. attack	6.54	6.3
min. nr. perf. moves	8	8
max. nr. perf. moves	192	191
	Parameterized Monte Carlo (Red)	*MCTS (Black)*
wins	23	27
pref. moves	59.32	58.86
nr. pos. moves	745.94	1055.36
max. nr. pos. moves	30	37
min. nr. pos. moves	1	2
nr. perf. spawn	9.42	9.8
nr. perf. attack	6.46	6.72
min. nr. perf. moves	22	22
max. nr. perf. moves	151	151

Our MCTS agent appears to apply a winning strategy against our Parameterized Monte Carlo agent. Knowing that MCTS performs better than Parameterized Monte Carlo, we will try to reduce the amount of work performed by our MCTS agent by setting a limit on the depth simulations can playout to (d_{mcts}). Simulations will then no longer necessarily playout until an end-state has been found.
With an average of 59 ply in games between MCTS (using Parameterized Random during simulations) we will experiment with our value $d_{mcts} \leq 40$. When d_{mcts} has been reached in a simulation we will use our heuristic function (described in Section 4.3.1) to evaluate if this is a winning simulation. From the data in Figure 22 we can see a general trend, the higher our value for d_{mcts} the higher

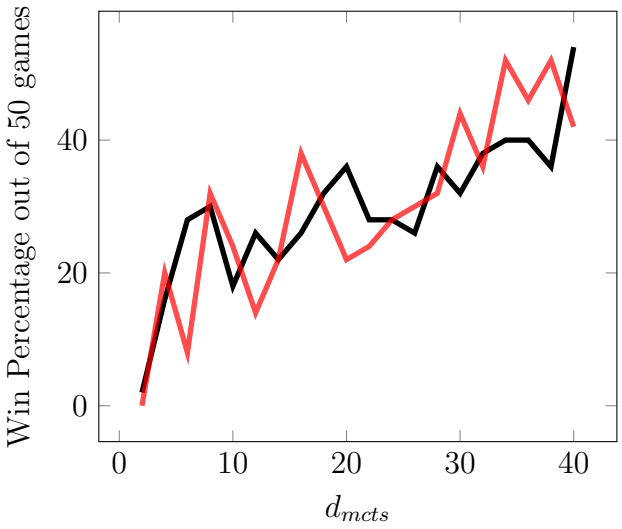

Figure 22: Plot showing the win percentage of 50 games for MCTS using Parameterized Random for simulations, $k_{mcts} = 1800$ and $c = 0.2$ with varrying values for d_{mcts} against Parameterized Monte Carlo. ━ represents the results when MCTS was Black and ━ when MCTS was Red.

our win rate for MCTS against our Parameterized Monte Carlo agent. With a value of $d_{mcts} = 40$ we were able to get our win rate for MCTS to around 50%.

6 Conclusion and Future Work

In this thesis we compare several different agents with each other to try and find which stategic agent performs the best for the game DOMINANCE and why. We can conclude that our Parameterized Monte Carlo and MCTS agents perform the best of all of our agents. This is in line with the a win percentage of above 70% even when playing against our Nega-max agent.
By dividing the list of possible moves into three different *types* of moves, assigning a ratio value to each type, we were able to improve our random player by making a Parameterized Random player. This Parameterized Random player was able to reduce the average number of ply per game (compared to our Pure Random agent) and was shown to beat our Pure Random agent. This simple division of moves into three lists greatly reduced the amount of work done per simulation (by reducing ply per game) and allowed our agent to focus on the types of moves that often go hand in hand with a winning strategy.
Three random sampling agents were implemented: a Basic Monte Carlo agent, a Monte Carlo agent that used the Parameterized Random player for sampling and a MCTS agent that also used the Parameterized Random player. A Nega-max agent that uses a heuristic function to assign values to game states was also implemented. During experimentation we found that the agents that parameterized the different types of moves performed better.
By limiting the simulation depth of simulations for MCTS we were able to do less work while achieving similar results. Although full simulations (until an end-state has been found) do appear to increase win rates for our MCTS agent. While limiting depth does reduce work, it appears to also reduce performance.
It also appears that our MCTS agent performs better when emphasizing exploitation over exploration. By testing different values for our constant c, which stimulates exploration when trying to find a node to expand, we noticed that smaller values for c increase our win rate. This suggests that our MCTS agent has to focus on exploitation over exploration.
For further research, it would be interesting to look at improving the testing platform to allow for more in-depth testing. We ended up using very low values for n and $k_{basicMC}$ when trying to find values for γ, δ and ω in Section 5.3. By using larger values for n and $k_{basicMC}$ we might have been able to get preciser values for the γ:δ:ω ratio and allow for better performance. Due to time limitations we were also only able to test relatively low values for $k_{basicMC}$, $k_{Par.MC}$, k_{mcts}, d and d_{mcts}. It would also be interesting to experiment with different heuristic functions.
Another interesting option for further research is to combine our developed ratio for the different *types* of moves with Nega-max to only expanding nodes in the decision tree that apply a move *type* that have the highest associated ratio value.
For further research we have made our C++ program available on GitHub[2].

[2]Full details: https://github.com/akannangara/dominance

References

[1] Agresti, A., Finlay, B., Miller, J. (2018). Statistical Methods for the Social Sciences 5th edition. Pearson.

[2] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. IEEE Trans. Comput. Intellig. and AI in Games 4: 1–43 (2012).

[3] BoardGameGeek. Dominance. https://boardgamegeek.com/boardgame/119216/dominance [online; accessed June 24, 2019].

[4] Thegamecrafter.com. Dominance. https://www.thegamecrafter.com/games/dominance [online; accessed June 24, 2019].

[5] Fraenkel, A. S., Lichtenstein, D. (1981). Computing a Perfect Strategy for n x n Chess Requires Time Exponential in n. J. Comb. Theory, Ser. A 31(2), pp. 199-214.

[6] Herik, H. J. van den, Uiterwijk, J. W. H. M., Rijswijck, J. van (2002). Games Solved: Now and in the Future. Artificial Intelligence, 134(1-2), pp. 277-311.

[7] Hoogeboom, H. J., Kosters, W. A., Rijn, J. N. van, Vis, J. K. (2014). Acyclic Constraint Logic and Games. CoRR abs/1604.05487.

[8] Knuth, D. and Moore, R. (1975). An Analysis of Alpha-beta Pruning. Artificial Intelligence, 6(4), pp. 293–326.

[9] Kocsis, L. and Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. ECML 2006, pp. 282–293.

[10] Romein, J. W. and Bal, H. E. (2002). Awari is Solved. ICGA Journal, 25(3), pp. 162–165.

[11] Russell, S. and Norvig, P. (2010). Artifical Intelligence: a Modern Approach. 3rd edition. Pearson.

[12] Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., Sutphen, S. (2007). Checkers is Solved. Science, 317(5844), 1518–1522.

[13] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Hassabis, D. (2018). A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go Through Self-play. Science, 362(6419), 1140–1144.