AN MCTS AGENT FOR EINSTEIN WÜRFELT NICHT!

Emanuel Oster

Master Thesis DKE 15-19

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Thesis committee:

Dr. Mark H.M. Winands
Dr. ir. Jos W.H.M. Uiterwijk

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
July 8, 2015

Preface

This thesis was written at the Department of Knowledge Engineering of Maastricht University. The thesis discusses the implementation of “EinStein würfelt nicht!” in Monte-Carlo Tree Search. I would like to express my thanks to everyone who helped me during the time of writing this thesis. Special thanks go to Dr. Mark Winands for all his useful hints and tips. Next I would like to thank my girlfriend Carina for telling me when I have to split my sentences, for her support and for enduring me during this time. More thanks go to my friends for cheering me up and providing me with distractions when I needed them. I also thank my parents for all their support and for making this possible (Danke Mama und Papa!).

Emanuel Oster
Aachen, July 2015

Abstract

Since the invention of the computer, attempts have been made to create programs, which are able to compete against humans in games. For a long time, several variants of the Minimax algorithm dominated the field of board games. However, the recent invention of the Monte-Carlo Tree Search (MCTS) algorithm allowed computer players, to drastically improve their performance in some games, in which Minimax did not perform well. This thesis tries to answers the question, how MCTS can be used to form a strong player in the game of "EinStein würfelt nicht!".

"EinStein würfelt nicht!" (EWN) is a board game invented by Ingo Althöfer in 2005. On a 5×5 board, two players try to be the first who reach the opposite corner, while movement is restricted by a die roll performed each turn. Despite its relatively simple rules, EWN is complex enough to be researched in the field of Artificial Intelligence

MCTS is a fairly recent best-first search algorithm that is best known for its performance in Go. Many different enhancements have been proposed so far, from which several are examined in this thesis for the case of EWN. It is described how MCTS has to be modified to work with the die rolls of EWN and which approaches have been used to reduce memory consumption.

Several enhancements and combinations of them are assessed to form the strongest-possible EWN agent. The enhancements are playout strategy, Prior Knowledge, Progressive History, MAST, Variance Reduction and Quality-based Rewards. The most beneficial enhancements seem to be Lorentz's (2012) playout strategy and Prior Knowledge. The resulting agent uses the both these enhancements and is then compared to the state-of-the-art EWN agent MeinStein. In the performed experiments, the performance of both agents has shown to be on equal footing.

Contents

Chapter 1

Introduction

This chapter gives an overview over AI in games and search algorithms. It states the problem statement and defines the research questions, which are the basis for this master thesis.

Chapter contents: Introduction — Games and AI, Search, problem statement and research questions.

1.1 Games and AI

Games have been an important pastime for thousands of years. While there exist several games that can be played with only one person (such as Solitaire), most games are dependent on more than one player and cannot be played alone. This can lead to a problem if no other players are available at the moment, which led to the idea to replace them with an artificial counterpart. An example of an early attempt to create such an artificial player is the Turk, which gave the impression to be able to play chess fully autonomously through complex clockwork mechanics. In fact, the Turk housed a human player instead and the clockworks were only used by that player to move the chess figures (Schaffer, 1999).

The first real automated players have been made possible through the invention of the computer. Since then, more and more games have been adapted for the computer and often featured also computer players to avoid the need for human competitive players. With constantly increasing computation power and better algorithms, these agents grew stronger throughout the last years. One major breakthrough was the defeat of the then-incumbent world chess champion Garry Kasparov by IBM's chess computer Deep Blue in a six-games match in 1997 (Campbell, Hoane Jr, and Hsu, 2002).

One of the major advancements in the recent past has been the development of agents, which are able to compete against expert-level players in Go on smaller boards. Before that, Go was one of few classic board games that has eluded the attempts to create a strong AI player. Even now, competing against expert-level players on the standard 19×19 board proves difficult (Browne *et al.*, 2012).

1.1.1 Games with Chance

An important subdomain are games with a chance factor, such as card games or games including dice. An example for this kind of game is Backgammon, which has also been adapted to computers. Already in 1979, an AI agent managed to defeat the then-incumbent world champion Luigi Villa in 7 out of 8 matches. This was also the first time that a computer program was able to defeat any world champion in any game (Tesauro, 1995).

Another game including chance is Can't Stop, in which a player can take several turns consecutively. However, depending on the die rolls it is possible that the progress of all these turns is lost if that mechanic is used too often. The player constantly has to estimate if taking another turn is worth the risk of losing all previous progress (Ren Fang, Glenn, and Kruskal, 2008).

A more recent example of a game with chance elements is "EinStein würfelt nicht!", in which the possible moves are influenced by a die roll. It is easy to learn but still allows for tactics and strategy,

which makes it interesting for tournaments. The game is actively played on the online gaming platform Little Golem (2015).

1.2 Search

In order to calculate the best move of a game, an agent has to examine the different possibilities and evaluate their strength as a move. Since this evaluation is dependent on the future moves of the opponent and the agent itself, they have to be considered as well by building a whole tree of possible future moves. Depending on the complexity of the game, the tree can become very large and traversing every branch can take longer than the allotted thinking time for the agent. In order to analyze such a tree, several search algorithms have been developed, some of which are shortly presented hereafter.

The **Minimax** algorithm (Von Neumann and Morgenstern, 1944) first reduces the size of the tree by limiting its maximal depth. An evaluation function then gives each leaf node an estimated value of how good that state is for the agent. Assuming that both players are playing as strong as possible, this information is then back-propagated by assigning each node the best value of all child nodes with respect to the current player.

In order to reduce the number of nodes, which have to be considered by the Minimax algorithm, the **$\alpha\beta$-pruning** algorithm (Knuth and Moore, 1975) has been introduced. It tries to reduce the number of nodes to consider by cutting away any branches of the tree, which are proven to be strictly worse than the best already found sibling node.

To adapt the Minimax algorithm to games with a chance element, the **Expectimax** algorithm (Michie, 1966) has been developed. It adds chance nodes to the tree, which represent each chance element of the game. These chance nodes are not assigned the best value of its child nodes, but their average value instead.

1.2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) (Coulom, 2007; Kocsis and Szepesvári, 2006) is a fairly new approach in game AI and has been described in detail in Browne *et al.* (2012). The basic idea of the algorithm is to randomly play out some games and examine promising moves more closely. This is repeated for a predefined time and the most promising move is chosen to be executed. Its major advantage over the Minimax algorithm and its enhancements is that it does not require an evaluation function. This can be important, because for some games such as Go (Coulom, 2007) an evaluation function can be difficult to design. Other domains for MCTS include real-time games such as Ms. Pac-Man (Pepels *et al.*, 2014) as well as games with imperfect information like Scotland Yard (Nijssen and Winands, 2012).

While MCTS has been researched in depth for deterministic games, using it for games with a chance element is a fairly new research topic and requires further investigation (Lanctot *et al.*, 2013).

1.3 Problem Statement and Research Questions

The chance element in "EinStein würfelt nicht!" has a strong impact on the players' decision but still allows for tactics and strategy (Lorentz, 2012). This makes it suitable for investigating the performance of MCTS in a game with a chance element. From that, the following problem statement can be derived:

How can we develop an MCTS agent for "EinStein würfelt nicht!", which performs as strong as possible and in a feasible amount of time?

This leads to the following four research questions:

1. *How can MCTS be made suitable for games with chance elements?*

   As explained above, using MCTS for games with a chance element is an unexplored territory and has to be investigated further. How can the die roll of EWN be incorporated into MCTS in such a way, that it represents the game realistically?

2. *What, if any, is the benefit of Variance Reduction in the case of EWN?*

   The game tree branches of MCTS are not necessarily comparable directly due to the chance elements of EWN. Variance Reduction tries to make these branches more comparable and it has to be assessed, if this is beneficial for the game of EWN.

3. *What, if any, is the benefit of playout strategies in the case if EWN?*

   More realistic playouts during the playout help MCTS to evaluate the branches of the game tree. Playout strategies try to play the game more realistically and it has to be evaluated, which strategies work the best in the case of EWN.

4. *What, if any, is the benefit of Selection strategies for MCTS in the case of EWN?*

   MCTS needs a number of games per branch, to give a first realistic estimate of which branch is the most promising one. Selection strategies try to guide MCTS into selecting potentially more promising branches more often in this stage. It has to be explored, how this benefits MCTS in EWN.

1.4 Thesis Outline

In the following, the outline of the thesis is described.

Chapter 1 gives an introduction to Games and AI and search algorithms, and "EinStein würfelt nicht!". This introduction then leads to the problem statement and the four research questions.
Chapter 2 describes the game "EinStein würfelt nicht!" in detail. History and background of the game is given, followed by an explanation of the rules. Afterwards, certain strategies are discussed before the chapter concludes with an overview of the game's complexity.
Chapter 3 gives a description of the MCTS algorithm and explains the enhancements, which are added to it. The chapter discusses MCTS and explains each algorithm step. Afterwards, the enhancements playout strategy, Prior Knowledge, Progressive History, MAST, Variance Reduction and Quality-based Rewards are explained.
Chapter 4 discusses how MCTS and each of the enhancements have been adapted for EWN. The chapter begins with an explanation on how MCTS has been modified to work with the chance events of EWN. Following, the structure of a node is explained. The remainder of the chapter describes how each enhancement introduced in Chapter 3 is used in the context of EWN.
Chapter 5 presents the experiments, which were done and analyzes their results. The experimental setup is described, followed by experiments for tuning UCT's C value and diminishing returns experiments. The added benefit of the tree, when compared to Flat Monte-Carlo is also examined. Afterwards, the enhancements described in Chapter 4 are assessed. The chapter finished with an experiment comparing the best agent against MeinStein, a strong Expectimax based agent.
Chapter 6 gives the conclusion of the thesis and outlines possible future research. The findings of experiments are concluded and the research questions and the problem statement are answered. Possible ideas for future research are proposed and explained.

Chapter 2

EinStein Würfelt Nicht!

This chapter describes the game "EinStein würfelt nicht!" with all its rules and also gives an overview over the strategies to consider when playing the game. Finally the complexity of the game is discussed.

Chapter contents: EinStein Würfelt Nicht! — Rules, Strategies and Complexity.

2.1 Background of the Game

The game "EinStein würfelt nicht!" (EWN) is a two-player board game by Ingo Althöfer in 2005. The name of the game means "EinStein does not roll dice". The first word, "EinStein", refers to Albert Einstein but also means "one stone", which is expressed by the capitalized letter "s". The name also refers to a statement of Albert Einstein, which was later abbreviated to "God does not roll dice". In addition, the name describes one of the rules of the game, saying that one does not have to roll dice with only one stone/token left. The game is the official game of a German exhibition focusing on Einstein during the Einstein Year 2005.

The game has already been researched under different aspects. Lorentz (2012) investigated the differences in performance between an MCTS agent and a pure Monte-Carlo search agent. Except for a basic playout and selection strategy, no additional enhancements to the MCTS agent have been discussed. Turner (2012) investigated endgame situations with up to 7 pieces left on the board and compared the results to the moves of different EWN agents during a tournament at the 16^{th} *Computer Olympiad* in Tilburg.

2.2 Rules

EWN is a board game for two players and is played on 5×5 board. Each player starts with six tokens numbered from 1 to 6. Player 1 starts in the upper-left corner and player 2 in the lower-right corner, arranging their tokens in that corner as they like. Figure 2.1 shows a possible board position after placing all tokens.

The goal of the game is to be the first player to reach the opposite corner with a token. Every turn is executed by first rolling a six-sided die. Afterwards the token that has the same number as the current die roll is moved. For the player starting in the upper-left corner, the directions down, right or diagonally down-right are allowed, and vice versa for the other player. If that square is already occupied, the existing token is removed from the game and replaced by the token, which has just been moved. This is even true if the previous token belongs to the same player.

As a result, it is possible that the rolled number of the die does not have a corresponding token. In such a case the player can choose between the next-higher or next-lower available token. If a player has only one token left, he does not need to roll, since every result would lead to the same token. A player automatically loses if he has no tokens left.

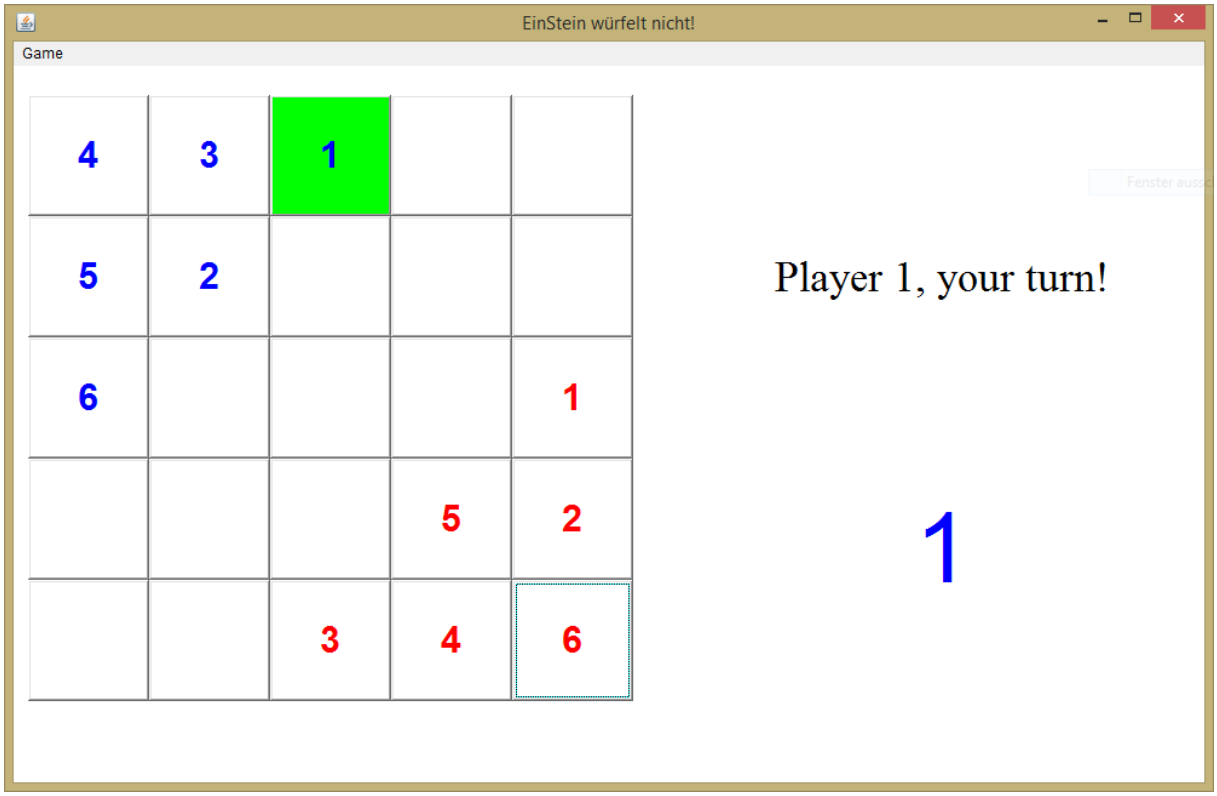

Figure 2.1: EinStein würfelt nicht! board after placing all tokens

Variant: Backwards Capture
In the standard rules it is not possible to move backwards with a token. The Backwards Capture variant of the game alters that rule by allowing a token to move backwards, if that move would capture another token.

Variant: Black Hole
In this variant, the center square of the board is a "black hole". In game terms this means that if a player moves his token onto that square, it is immediately removed from the game. This makes it easier for the players to get rid of their tokens but also blocks one of the direct paths to the goal.

2.3 Strategies

Capturing
When moving a token in "EinStein würfelt nicht!", one of the most important questions to consider is, whether another token should be captured or not. This highly depends on the current board position. On the one hand, the fewer tokens a player has, the higher the probability is that he can move the token he would like to move. On the other hand, having few tokens left increases the danger of being completely captured by the other player and thus losing the game. For capturing tokens of the opponent, the opposite has to be considered. Capturing these tokens increases the probability of the opponent to move his desired token. It can still be a good move though, for example if there is a token close to the goal, which would allow the other player to win soon.

Moving
The second important question is whether one should move directly toward the goal or take a detour. Similarly to the capturing, this also depends on the current board state. Moving closer to the goal minimizes the remaining number of moves the player has to take in order to reach the goal. On the contrary

it can be beneficial to take a detour such that the token is harder to capture for the other player.

Considering Chances
In addition to these considerations, it is also important to notice the probability with which a specific token will move in the next turn. For instance, if the opponent has only tokens 5 and 6 left, the probability for token 5 to move is $\frac{5}{6}$ while for token 6 only $\frac{1}{6}$. If the player wants to avoid being captured, but has to move towards one of these two tokens, it is likely better to move towards token 6, as this token will less likely move in the opponent's next turn.

2.4 Complexity

State-Space Complexity
The state-space complexity describes the number of possible board positions reachable from the initial setup of the board (Allis, 1994). An upper bound for this complexity can be given by calculating each possible board combination for all possible numbers of tokens on the board. Shannon (1950) describes the following formula in order to calculate all possible board combinations with a fixed number of tokens:

$$C = \frac{n!}{(n-k)! \times i_1! \times i_2! \times ... \times i_m!}$$

Here, n is the number of squares on the board, k is the number of tokens on the board, and i_x is the number of identical tokens (for instance the 8 pawns of the same color in chess). In EWN every token is unique, meaning that each $i_x!$ would be 1 in the above formula and can be omitted. In order to get all possible board combinations, the results for each number of remaining tokens have to be added up, resulting in the following formula for EWN:

$$SSC \leq \sum_{k=1}^{12} \frac{25!}{(25-k)!} \approx 2.68 \times 10^{15}$$

This result contains board combinations impossible to achieve with legal moves and does not consider symmetry, meaning that it can only give an upper bound for the state-space complexity. The results confirm the calculations of Turner (2012).

Game-Tree Complexity
The game-tree complexity is described as the average branching factor to the average length of the game (Allis, 1994). Self-play experiments have shown that the average branching factor including chance outcomes for EWN is ∼16.10 and the average game length is ∼18.51. Thus, the game-tree complexity for EWN is:

$$GTC = 16.10^{18.51} \approx 2.18 \times 10^{22}$$

2.5 Chanciness

In the domain of games, chanciness describes by what extend the outcome of a game is influenced by its chance elements. Erdmann (2009) has developed a method to measure this chanciness in games and applied that method to EWN among others. In general, chanciness is a value between 0 and 1, where 0 means that the outcome of a game only depends on the skill of the players and 1 means that the outcome of the game only depends on chance. It has been shown that the chanciness not only depends on the game but also on the players playing that game.

In the case of EWN, different player strengths have been simulated by using Flat Monte-Carlo agents with various numbers of simulations. Experiments have been performed with equally strong players and with increasingly stronger players. The results of Erdmann (2009) regarding EWN's chanciness are shown in Table 2.1.

The first block of results shows the chanciness if both players are equally strong. The chanciness increases with rising strength of the players. The next two blocks of results show that chanciness decreases, as one player becomes increasingly stronger. The chanciness is also smaller, if the stronger player is the

first player to move. The last result shows that chanciness is rather high, if one player is stronger than the other and the weaker player is already fairly strong. These results indicate that it could be difficult to prove the superiority of a player, if both players perform already on a high level.

Player 1 Simulations	**Player 2 Simulations**	**Chanciness**
1	1	0.51 ± 0.028
8	8	0.62 ± 0.025
64	64	0.71 ± 0.022
512	512	0.84 ± 0.015
1	8	0.44 ± 0.023
1	64	0.35 ± 0.020
1	512	0.30 ± 0.017
8	1	0.43 ± 0.021
64	1	0.33 ± 0.018
512	1	0.24 ± 0.017
512	1024	0.82 ± 0.013

Table 2.1: Chanciness in EWN (Erdmann, 2009)

Chapter 3

Monte-Carlo Tree Search

This chapter describes the Monte-Carlo Tree Search algorithm including a description of every main step as well as the UCT formula.

Chapter contents: Monte-Carlo Tree Search — Overview, Algorithm Steps and UCT

3.1 Overview

Monte-Carlo Tree Search (MCTS) (Coulom, 2007; Kocsis and Szepesvári, 2006) is a best-first search algorithm. It combines the idea of Monte-Carlo sampling with a tree in order to enhance performance. In basic Monte-Carlo sampling, the outcome of currently available actions is estimated by repeatedly solving each state after a given action at random. The idea is that the higher the number of simulations is, the higher the accuracy of the predictions is as well. In MCTS, a tree is added in order to make use of knowledge about future events.

The nodes of this tree represent a state, while the edges represent the action, which was used in order to reach that state from the previous state. Each node stores information about the times it was visited as well as the cumulative score for all simulations of the node's state and all following states. The algorithm is split up into four steps, which are described in the following. In addition, Algorithm 1 provides a pseudo code version of the algorithm.

MCTS has been shown to perform successfully in various games. Examples for such games include Go (Gelly *et al.*, 2012), Amazons (Lorentz, 2008) and Hex (Arneson, Hayward, and Henderson, 2010). Especially in Go, MCTS has marked a new era for computer players. Previously, Go was one of few classic games in which human players outperformed computer players even on small boards. MCTS was the first approach that was able to beat expert level human players on smaller Go boards. However, playing against such players on the standard 19×19 board still proves difficult for MCTS (Browne *et al.*, 2012).

3.2 Algorithm Steps

The following algorithm steps are based on Chaslot *et al.* (2008), but the expansion step was modified. Each step is visualized in Figure 3.1. The four steps are repeated either until a certain time has passed or until a predefined number of loops is executed.

Selection
During the selection step, a leaf node of the tree is selected. In order to do so, the algorithm starts at the root of the tree and analyzes all children of it. Normally, the analysis assigns each child a rating based on the number of wins and visits, but other criteria can also influence that rating. Afterwards, the child that fits the criteria of the analysis the best is chosen and the procedure is repeated for that node. This continues until the chosen node is a leaf node. In the end, the node, which is most suited to be investigated more closely, is selected.

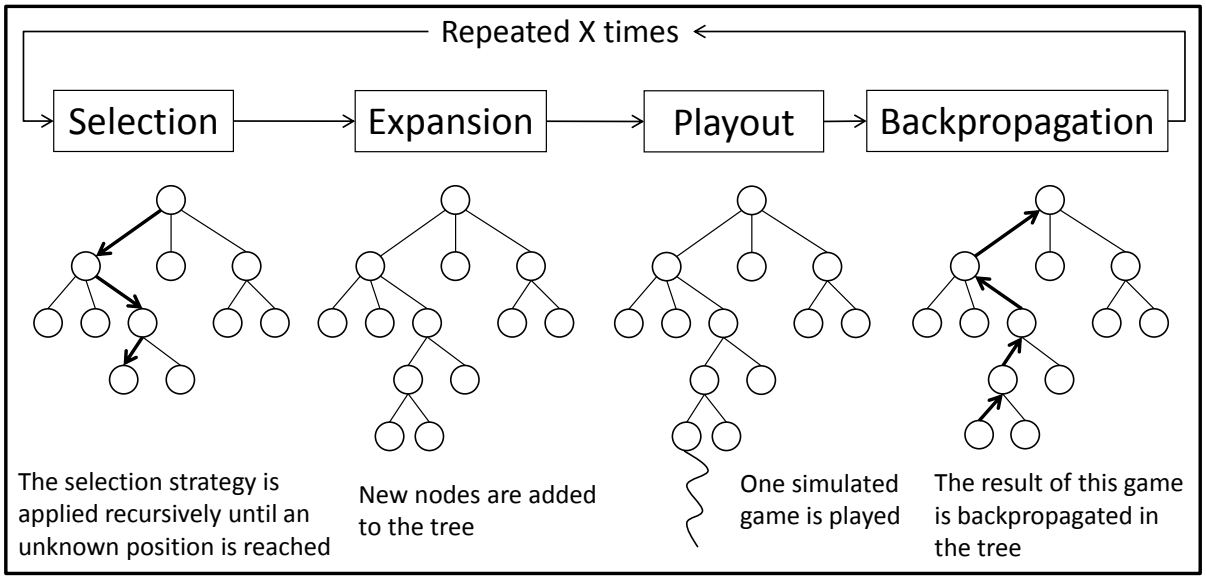

Figure 3.1: Visual representation of the MCTS algorithm steps

Expansion
In the expansion step, the selected leaf node is expanded. In this thesis, this means, that each possible move in the state of the selected node is added as a child to that node. In Chaslot *et al.* (2008), only one node is added during this step instead.

Playout
In the playout step, one of the new children added during the expansion step is simulated until a terminal state is reached. This can be done by randomly picking and executing one of the possible moves, but it is also possible to give moves different weights or even use more elaborate strategies.

Backpropagation
During the backpropagation step, the result of the playout is backpropagated through the tree. Starting from the node that has just been simulated, the visit count of the node is increased by one. The score of the node is also updated with the playout result from the parent node's point of view. This means that, if in the playout Player 1 won, and it is the turn of Player 1 at the parent node, then the score of the current node is updated by a win score. The score of a single game lies in the bounds of $[0, 1]$. This procedure is repeated for each parent node until the root node is reached.

3.3 Upper Confidence Bound Applied to Trees

As described above, each node is assigned a rating in order to indicate how suited it is to be investigated more closely. One of the most common methods to assign such a rating is Upper Confidence Bound Applied to Trees (UCT) (Kocsis and Szepesvári, 2006). The following formula describes UCT:

$$UCT = \frac{s_n}{v_n} + C\sqrt{\frac{\ln v_p}{v_n}}$$

In the above formula, s_n is the score of node n, v_n is the number of times node n was visited, v_p is the number of times the parent node of node n was visited and C is a constant. The first part of the formula represents how well the current node performed on average in the previous simulations. The second part of the formula is a counterbalance for exploration. It favors less often visited nodes and as such forces MCTS to also consider nodes, which currently have a worse score, than the sibling with the best score. The impact of the second part can be manipulated through the constant C.

This formula does not work if a node is not visited yet, because the denominator would be 0 in this case. This issue can be resolved by giving such nodes a default value such as 1 or ∞. In this thesis

Algorithm 1 MCTS

```
1: function MCTS(Node root)
2:     startTime ← currentTime
3:     while startTime + thinkingTime > currentTime do
4:         currentNode ← root

5:         while currentNode has children do                          ▷ Selection
6:             bestChild ← first child of currentNode
7:             for all children of currentNode as child do
8:                 if child.calculateUCT() > bestNode.calculateUCT() then
9:                     bestNode ← child
10:                end if
11:            end for
12:            currentNode ← bestChild
13:        end while

14:        currentState ← game state in currentNode                   ▷ Expansion
15:        newChildren ← create Nodes for each possible move in currentState
16:        currentNode.children ← newChildren

17:        currentNode ← one child of currentNode                     ▷ Playout
18:        currentState ← game state in currentNode
19:        while ¬currentState.gameEnded() do
20:            possibleMoves ← currentState.getMoves()
21:            chosenMove ← choose move from possibleMoves
22:            currentState.executeMove(chosenMove)
23:        end while
24:        result ← currentState.getResult()

25:        while currentNode has parent do                            ▷ Backpropagation
26:            currentNode.visits + +
27:            currentScore ← result from currentNode.parent's point of view
28:            currentNode.score ← currentNode.score + currentScore
29:            currentNode ← currentNode.parent
30:        end while

31:    end while
32:    bestChild ← first child of root
33:    for all children of root as child do
34:        if child.visits > bestChild.visits then
35:            bestChild ← child
36:        end if
37:    end for
38:    return bestChild
39: end function
```

however, the following modified formula is used:

$$UCT = \frac{s_n}{v_n + 10^{-6}} + C\sqrt{\frac{\ln v_p}{v_n + 10^{-6}}} + \text{rand}(0, 10^{-6})$$

This formula extends the standard UCT formula by adding a small value to the denominator of both fractions. In the case of $v_n = 0$, the first fraction will be 0 (as $s_n = 0$ if $v_n = 0$). The second fraction will also be 0 for $v_p = 1$ but much larger for $v_p > 1$. This has the effect that unvisited nodes are forced to be explored at least once. In the cases where $v_n > 0$ the small addition to the denominator has a negligible effect. The second modification to the formula is to add a small random number in the end. This number has the effect that ties between two nodes with the same UCT value are resolved at random, countering a bias introduced by the node order.

3.4 Playout Strategy

As mentioned in Section 3.2, there are different approaches for the playout step. For instance, instead of choosing a move each turn completely at random, different moves can be assigned different weights, based on an evaluation function. Another approach can be to use a search algorithm in the playout as well. The goal of this approach is to increase the accuracy of the playout, which probably increases the accuracy of the node's rating as well, given sufficient time. However, not every playout strategy that performs well on its own increases the performance of MCTS (Bouzy and Chaslot, 2006; Lorentz, 2012). As a result, playout strategies have to be compared using MCTS and not on their own.

3.5 Prior Knowledge

Each time new nodes are added to the tree during the expansion step, MCTS needs several playouts until a first estimation of these nodes is possible. Prior knowledge tries to reduce the time needed for this first estimation by assigning each new node a predefined value depending on the move, which leads to that node (Gelly and Silver, 2007). As predefined values, the node gets a number of visits $v_{prior} > 0$ and a score s_{prior} depending on a heuristic function, where $0 \leq s_{prior} \leq v_{prior}$. These numbers are not backpropagated but are only used to influence the selection step at the current level of the tree. Should the heuristic estimation not match the actual strength of the current move, it will be disproven by MCTS over time, as the proportion of the predefined values decreases with each visit of the node. The higher the initial number v is chosen, the longer it takes MCTS to disprove an inaccurate estimation.

3.6 Progressive History

The Progressive History enhancement (Nijssen and Winands, 2011) tries to improve the performance of the selection step. In order to do so, the number of wins and playouts are remembered independently for each move for each player. The data for this can come from the tree as well as from the moves performed during the playout step. Progressive history is based on the idea that a move that often leads to a win, is likely to be a good move in the current situation as well. In order to make use of that knowledge, Progressive History modifies the UCT formula by adding another part to it:

$$UCT_{PH} = \frac{s_n}{v_n} + C\sqrt{\frac{\ln v_p}{v_n}} + \frac{s_{PHn}}{v_{PHn}} \times \frac{W}{v_n - s_n + 1}$$

In the above formula, the meaning of s_n, v_n, v_p and C is identical to the one in Section 3.3. For the new elements it holds that s_{PHn} is the score of the move that leads to node n within the Progressive History table, v_{PHn} is the number of visits of the move that leads to node n within the Progressive History table and W is the factor, with which the impact of the Progressive History can be modified. The denominator $v_n - s_n + 1$ describes the number of losses of node n (plus 1 to avoid dividing by 0). In this way, the influence of the Progressive History part decreases for badly performing moves.

3.7 Move-Average Sampling Technique (MAST)

MAST (Finnsson and Björnsson, 2008) is a different approach to improve the accuracy of the playout step. In MAST, the same information as in the Progressive History enhancement is stored. Instead of using that information during the selection step, it is applied during the playout step. In this thesis, this is done by using an ε-greedy approach. This means, that the best move according to the Progressive History table will be chosen with $p = 1 - \varepsilon$ probability. For the remaining $p = \varepsilon$ probability, a random move is chosen (Tak, Winands, and Bjornsson, 2012).

3.8 Variance Reduction

In trees with chance elements, different branches of the tree are not necessarily comparable because different chance outcomes were used for each branch. Variance reduction (Veness, Lanctot, and Bowling, 2011) tries to solve this problem by reusing chance outcomes from other branches. One approach to achieve this is to have a table of dice rolls for each level of the tree. Each time, a node is reached the n^{th} time, the n^{th} entry of the table that belongs to the level of the node is used as the current chance outcome. If it does not exist yet, it is generated and stored in the table. In a more simplified version of this approach, only one table for all nodes of the tree is used, regardless of the node's level (Cowling, Powley, and Whitehouse, 2012). This means, that each time any node is visited for the n^{th} time, the n^{th} entry of the table is used.

Instead of using different tables for each level of the tree, it is also possible to use a table for sequences of chance outcomes. Here, the different branches are only distinguished at the root, meaning that each child node of the root is a different branch. Each time one of the branches is chosen for the n^{th} time, the n^{th} sequence of chance outcomes is used for the following tree traversal. As before, if an entry does not exist yet, it is generated and stored in the table.

3.9 Quality-based Rewards

In games, it is often possible to evaluate if a player barely won or if he dominated the game. For instance consider a chess game. A game could have ended with both players having only two pawns left, plus the king of the winner. In such a game, it is unlikely that the winner played significantly stronger than the opponent. However, another game could have ended with the losing player having only two pawns left. The winner, though, only lost 3 pawns and a bishop. In such a game it is likely that the winner played considerably stronger than the opponent.

The concept of Quality-based Rewards tries to make use of these differences (Pepels *et al.*, 2014). If a game during the playout step ended with a dominant player, then that playout should get a higher weight for backpropagation. In order to do so, the Quality-based Rewards approach calculates a bonus to the reward of a playout. This is based on the result, the average of previous results and their standard deviation. The final reward is calculated as follows.

$$r_b = r + \text{sgn}(r) \times a \times b(\lambda_q), \qquad r \in \{-1, 0, 1\}$$

In this formula, r_b is the adjusted reward, r is the initial reward (loss, draw, win), a is a scalar, which has to be determined empirically, and $b(\lambda_q)$ is the bonus added to the reward. $b(\lambda_q)$ is defined as

$$b(\lambda_q) = -1 + \frac{2}{1 + e^{-k\lambda_q}}$$

where k is a constant to be determined empirically and λ_q is

$$\lambda_q = \frac{q - \bar{Q}^\tau}{\hat{\sigma}_Q^\tau}, \qquad q \in (0, 1)$$

Here, q describes the quality of the playout, $\bar{Q}^\tau$ is the average of all past quality values for winning player τ and $\hat{\sigma}_Q^\tau$ is the standard deviation of all past quality values for winning player τ.

Chapter 4

MCTS and EinStein Würfelt Nicht!

This chapter describes how MCTS has been adapted to work with EWN and how the enhancements described in Chapter 3 are used in the context of EWN.

Chapter contents: MCTS and EinStein Würfelt Nicht! — Implementation Details and Enhancement Specifications

4.1 Chance

As discussed in Section 2.2, a die roll determines each turn, which tokens are available for movement. This rule has to be addressed within MCTS as well. Each time MCTS chooses a move, the available moves have to be limited to those corresponding to a temporary die roll. This affects the available nodes during the selection step, as well as possible moves during the playout step. To resolve this problem, each node has a flag for every number 1-6. During creation of the node, it is checked, which die rolls will lead to that node, and the corresponding flag is set to true. Every time a child has to be selected, a random number is generated, and subsequently only the children are available whose matching flag is true. During the playout step, it is sufficient to create only the moves belonging to a random die roll each turn.

As an example, consider a board situation in which Player 1 has tokens 1, 4 and 6 left, as shown in Figure 4.1. When MCTS tries to access the child nodes in this situation, a random die roll is generated. In this example, a 3 is generated. Instead of returning each child node, it is checked, which nodes are associated with the die roll 3. In this case, the left four nodes are returned. The rightmost node is still a child node of the current node, but is kept invisible from MCTS due to the current die roll.

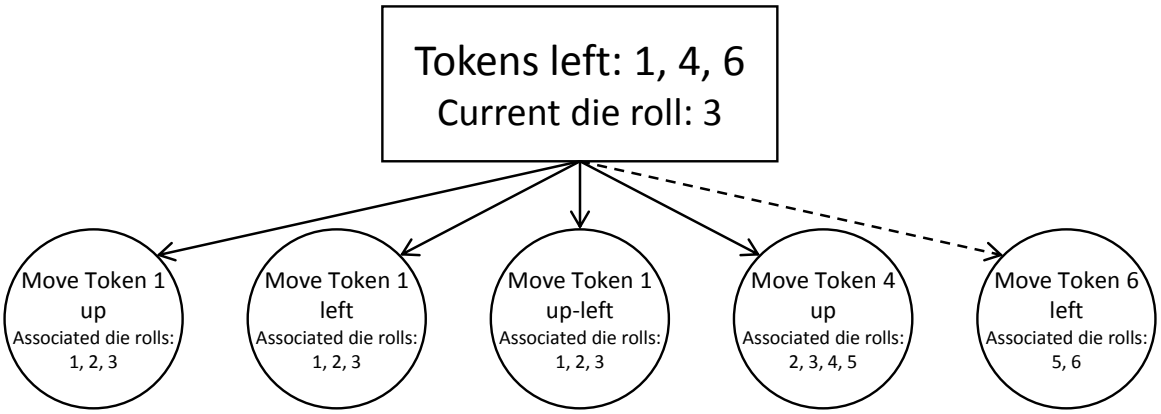

Figure 4.1: Limited access to nodes depending on die roll

4.2 Node Representation

Because of the decision to generate all possible children during the expansion step (see Section 3.2) and the relatively short average game length of EWN, the engine generates a high number of nodes. As computer memory is limited, the nodes had to be designed lightweight.

Necessary for MCTS are values for the number of wins and visits as well as information of the parent node and child nodes. Instead of using an array to store references to all child nodes, a single reference to the first child node is used. In addition, a reference to the next sibling node is stored. From this information, a sequence of all child nodes can be reconstructed.

For the expansion step, the board state is needed as well. However, storing this information in a node consumes a large amount of memory. Instead, only the move that led to the node is stored. With the initial board state and the sequence of moves leading to the current one, the board state can be reconstructed.

Finally, information about the die rolls that lead to the node is needed, as described in the previous section. This can be achieved by representing each possible die number as a Boolean variable, signaling whether that number will lead to the node or not. However, in Java, a Boolean variable consumes 8 Bit of memory. To address this problem, a Byte variable is used instead. By using bit shifting operations, this variable can function as a memory efficient representation of 8 Boolean variables. As there are only 6 possible die roll outcomes, 2 more boolean values could be stored without consuming any more memory. These values are used to indicate if the move leading to the node is a capturing move, and which player executed the move. While this information could be easily restored from other existing data, it is still useful to reduce overhead. If a move is a capturing move is an information used for Progressive History and MAST, as explained in Section 4.5. Knowing, which player executed a specific move can is relevant on several occasions and is therefore useful to be accessed in a convenient way. Figure 4.2 gives a visual overview of the byte variable allocation used in a node.

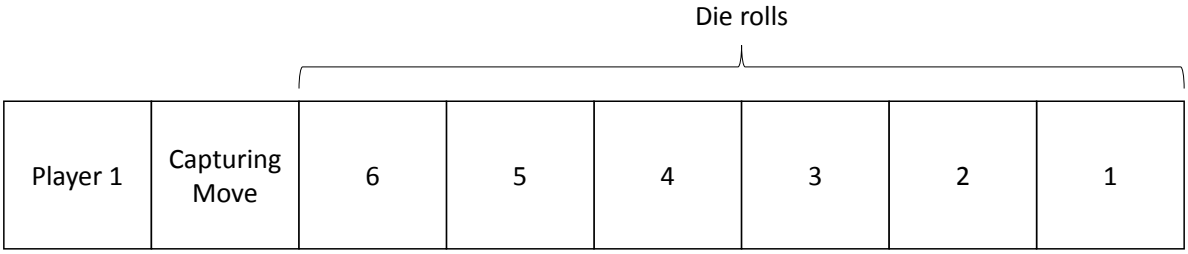

Figure 4.2: Byte variable allocation

4.3 Playout Strategy

Section 3.4 discussed the advantages of a playout strategy to improve the accuracy of the playout step. Lorentz (2012) proposes a simple strategy, which favors capturing moves and moves that place a token strictly closer to the goal over the remaining ones.

The idea behind this approach is, that in most cases, the direct way is favorable, as taking another way only makes sense to evade an opponent's token. However, this is only useful in specific situations, which is the reason why these kind of moves receive a lower weight. Capturing moves are also favored as they are useful in most situations, regardless of the token's player. Having less tokens on the board increases the probability that specific token can be moved in future turns, while capturing the opponent's tokens often prevents them to come too close to the goal.

This strategy is accomplished by doubling the probability of such moves to be chosen during each turn of the playout. In addition, if one of the moves is a win in one, it is chosen regardless of its probability. Otherwise the move is chosen with the roulette-wheel approach (cf. Powley, Whitehouse, and Cowling, 2013). First, all weights for the currently available moves are added up. Then this value is multiplied by a random number between 0 and 1. Finally, the weight for each move is subtracted again from this value. Once the value becomes negative, the move, which just subtracted its weight, is chosen to be executed.

4.4 Prior Knowledge

In Section 3.5 the advantages of initializing new nodes with Prior Knowledge were discussed. Lorentz (2012) proposes a similar strategy to the strategy presented in the previous section by favoring capturing moves and moves leading closer to the goal. Each new node is initialized with a visit count of 100. Depending on the nature of the move, which led to the new node, the win counter is initialized with a different number. The standard value for a node is 40 wins. If the move brought the token closer to the goal, a higher value is used depending on how close the token is to the goal after the move. Capturing moves receive a value of 65. If a move fulfills both criteria, it gets the higher value and 5 bonus points. Table 4.1 gives an overview of the values used for each possible move.

Move not leading closer to goal			
Capturing Move		65	
Non-Capturing Move		40	
Move leading closer to goal			
		Non-Capturing Move	Capturing Move
Distance to goal after move	0	100	105
	1	90	95
	2	75	80
	3	60	70

Table 4.1: Prior Knowledge Values

4.5 Progressive History and MAST

Implementing Progressive History and MAST for EWN was accomplished by modifying the playout and backpropagation steps. Two tables are used to store the history information: One table stores the cumulative score for each move, the other counts how often that move has been executed. During the playout step, each move that has been executed is remembered. After the playout is finished, the history tables are updated for each move. A move is distinguished by were the token came from, were it went and whether it captured another token.

To find the correct index within the tables, a number representing the move is generated. This is done by multiplying the numbers for $from_{Column}$, $from_{Row}$, to_{Column} and to_{Row} with 1000, 100, 10 and 1 respectively. This has the effect that each of these numbers becomes a different digit in a four digit number. In addition, 10000 is added, if the move was a capturing move. For instance, a capturing move from square (2,4) to square (1,4) would receive the number 12414. It is not necessary to distinguish the two players, because they cannot execute the same move, as moving backwards is not allowed in EWN. This is not a memory efficient way to save the moves but has the advantage that the index remains human-readable. The total size of one table is 14444 and uses ∼113KB of memory in Java. Since the tables are static, this memory consumption is negligible on today's computers.

The moves performed during the selection step are also taken into account by a modified backpropagation step. During the traversal back to the root, it also updates the history tables for moves performed during the selection step. The tables are always cleared at the beginning of each simulation.

To use this information for Progressive History, the modified UCT formula described in Section 3.6 is used during the selection step. For MAST, an ε-greedy approach uses the history data to choose each move in the playout step.

4.6 Variance Reduction

Section 3.8 explained, that Variance Reduction reuses die rolls of other parts of the search tree. Two of the proposed methods were implemented. The first method uses Variance Reduction applied to each node individually, the second method applies Variance Reduction to the nodes of each level of the tree. Both variants need information about how often a node has been visited, which is already an integral part of MCTS and can be reused.

For the first variant, an ArrayList was used to keep track of the previous die rolls. Each time a node is accessed, it gets assigned the n^{th} entry of the ArrayList as the current die roll, where n is the visit counter of the node. If the ArrayList is shorter than n, a random die roll is generated and appended.

For the second variant, a two-dimensional ArrayList is used instead. The first dimension describes the level of the node and the second dimension describes the past die rolls for that level. When a node is accessed, the l^{th} entry of the n^{th} ArrayList is used, where l is the level of the node within the tree and n is the number of visits. l can be determined by counting the number of predecessors of the node. If l is larger than the number of ArrayLists in the first dimension, a new one is added. If n is larger than the size of the l^{th} ArrayList, it is expanded as described above.

4.7 Quality-based Rewards

The implementation of Quality-based Rewards for EWN needs a quality measurement of the game, as described in Section 3.9. Counting the number of remaining tokens on the board is unlikely a suitable quality measurement, as having fewer tokens is often favorable over having more tokens. Simultaneously, having few tokens increases the risk of losing, if the opponent captures all remaining tokens. It is highly depending on the board situation, if more or fewer tokens are desirable, which is the reason why this approach is unlikely suitable as a quality measurement. Instead it is proposed to count the number of turns, the losing player would still need at least to reach the goal. This approach gives a bonus to playouts, in which the losing player would need several more turns to reach the goal, as this was likely the result of a strong play.

Another possible approach is to use the game length. While Pepels *et al.* (2014) propose to give particularly short games a bonus, this might be counterproductive in EWN. A short game in EWN is often caused by a series of favorable die rolls instead of a strong play. Therefore, it is proposed to downgrade the reward of such playouts as they are not presenting a meaningful result.

4.8 MeinStein

MeinStein (cf. Krabbenbos, 2015) is an EWN agent written by Theo van der Storm and won the 16$^{\text{th}}$ Computer Olympiad in EWN (Turner, 2012). As such it is suitable as a benchmark for evaluating the performance of the agent written for this thesis. MeinStein uses an Expectimax algorithm (Michie, 1966) with iterative deepening. The minimal depth is 6 and the maximal depth is 20. A time constraint is also used, which might prevent MeinStein from reaching the maximal search depth. In its standard setting, the time constraint is set to 3.5 seconds. A transposition table is used to enhance the performance. The size of the transposition table is normally calculated dynamically based on the available memory, but has been fixed to 4,000,000 entries for the experiments in this thesis. Table entries are only replaced if the new entry comes from a deeper search than the already existing entry. MeinStein does not use move ordering.

In order to test against MeinStein, it has been modified by removing all GUI elements. MeinStein provides the ability to set the current state of the board using a modified version of the Forsyth-Edwards-Notation (FEN) for chess. Each time it is MeinStein's turn, the current board state is exported to that modified FEN and then imported into MeinStein. After finishing its calculations, MeinStein returns the move that has to be performed.

Because MeinStein uses the Expectimax algorithm, it uses an evaluation function to estimate a certain board state. Using an ε-greedy approach, this evaluation function could also be used in the playout step of MCTS. To use this approach, the current board state is imported into MeinStein as described above. Afterwards, MeinStein's Expectimax algorithm is used with a search depth of 1. Effectively, this evaluates each possible move using the evaluation function and returns the move with the highest estimated winning probability. At the same time, this approach ensures that the evaluation function is used as intended.

Chapter 5

Experiment Results

This chapter describes the setup used for the experiments and discusses the results for the performed experiments to assess MCTS and its enhancements.

Chapter contents: Experiment Results — Experimental setup and result discussion

5.1 Experimental Setup

Unless otherwise noted, each experiment has been performed with the following setup. Both agents were given a thinking time of 1s per turn. 1000 games were performed per experiment, with sides switched after each game to avoid first or second player bias. As a result, each agent plays 500 games as Player 1 and 500 games as Player 2. The results generally show the win ratio of Player 1 in percent and indicate the confidence bounds for a 95% confidence level.

All agents are written in Java. The experiments have been performed using CentOS 5.11 and Java 1.7.0_40 64-Bit. The following hardware has been used.

- 2 x AMD Dual-Core Opteron F 2216, 2.4 GHz, 95 Watt (max. 2 Opteron Socket F Processors)
- 8 GB DDR2 DIMM, reg. ECC (4 DIMMs, max. 32 GB, 16 DIMMs)
- NVIDIA nForce4 2200 Professional chipset
- 2x PCI-E x8 slots via riser card (full height, half length)
- 80 GB hot-swap SATA hard drive, max. 2 hot-swap hard drives
- DVD-ROM
- Onboard dual Broadcom BCM5721 Gigabit Ethernet
- Onboard XGI Z9s VGA 32 MB DDR2 graphics
- 1 U rackmount chassis incl. mounting rails
- 500 Watt power supply

On the given setup, the agent written for this thesis performs on average ∼44000 simulations in the first second and on the first turn of a game, when using the playout strategy and Prior Knowledge enhancements.

5.1.1 Starting Positions

The rules for EWN make it not entirely clear how the process of arranging the players' tokens at the beginning of the game takes place. Four variants are possible. First, it could be that the players place their tokens alternating. Second, it could be that one player places all his tokens on the board and then the second player places all his tokens on the board. Third, both players could place their tokens on the board in secret, using a piece of paper or similarly to hide their starting position. In the last variant, which Is used in this thesis, both players always place their starting tokens at random. This variant ensures that no bias resulting from the starting position is introduced.

5.2 Tuning C

Within the UCT formula (see Section 3.3), C is used to balance the exploration factor. As initial value for C, $\sqrt{2}$ has been chosen. In a next step, two agents with the playout strategy and Prior Knowledge enhancements played against each other, where one player always used $C = \sqrt{2}$ and the other player used a different value for C. Figure 5.1 shows the winning percentage of the player using various C values, including standard deviation. As seen, modifying the C values does not change the performance of the agent by much. Interestingly, even a greedy approach performs only slightly worse. This could be due to the fact, that a small random number is added to each UCT calculation to break ties. If MCTS would become stuck in a branch, in which it loses most of the times, the small random number will be big enough to choose a different branch. Because the performance did not change significantly for the other C values, the initial value of $\sqrt{2}$ was not changed for the other experiments.

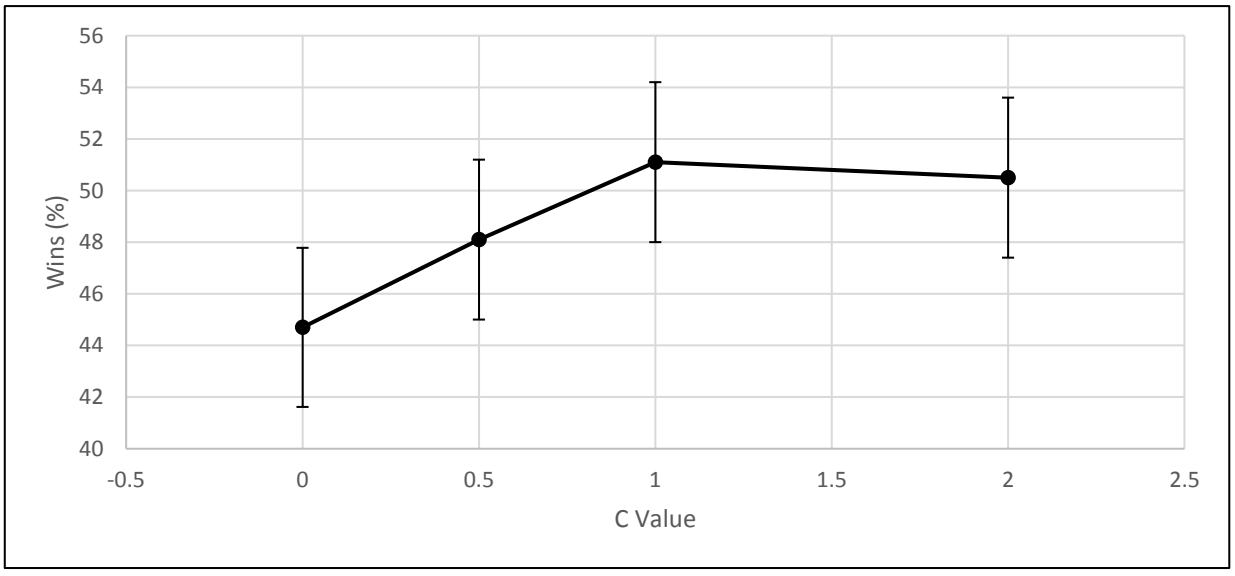

Figure 5.1: Experiment results: C values

5.3 Diminishing Returns

In order to determine the influence of an increased number of simulations, several experiments have been performed. Both players used the same MCTS agent, which has been limited to a fixed number of simulations instead of a time limit per turn. For the experiments, the number of simulations was set to 500, 1000, 2000, 4000, 8000 and 16000. Each instance was tested against all instances with more simulations. This experiment has been performed with three different variants of the MCTS agent. These are the plain MCTS agent without enhancements, the MCTS agent with the Prior Knowledge enhancement, and the MCTS agent with the Prior Knowledge and playout strategy enhancements.

Tables 5.1, 5.2 and 5.3 show the results of these experiments. The tables show the winning percentages of the column player. Figures 5.2, 5.3 and 5.4 also give a visual representation of these results. As

seen in Table 5.1, if there is no knowledge added to MCTS, doubling the number of simulations is not sufficient to increase the performance noticeably. However, after quadrupling the number of simulations and thereafter, performance increases. This experiment has been performed with 32000 simulations in addition to the numbers mentioned before. This was done to verify that the unexpectedly high number of wins of 8000 simulations against 16000 simulations does not denote an upcoming new trend.

In Table 5.2, Prior Knowledge was added to both agents. This led to a significant improvement in performance for doubling the number of simulations. This may indicate that the Prior Knowledge does indeed guide MCTS into promising nodes. As a result, the higher number of simulations is likely not wasted on moves, which do not perform well but have to be disproven. For Table 5.3, the playout strategy was added in addition to the Prior Knowledge. The results do not differ significantly from Table 5.2. In both tables, the effect on the performance decreases, the more simulations are used. Finally, doubling from 8000 to 16000 simulations does not seem to influence the performance anymore, as it is expected by the law of the diminishing returns (Heinz, 2001; Robilliard, Fonlupt, and Teytaud, 2014).

	1000	**2000**	**4000**	**8000**	**16000**	**32000**
500	50.8	55.1	55.4	59.8	60.6	63.9
1000		50.2	51.4	56.4	57.3	62.2
2000			51.0	53.2	54.8	64.1
4000				50.1	53.7	61.4
8000					55.0	53.1
16000						51.0

Table 5.1: Diminishing return results without enhancements

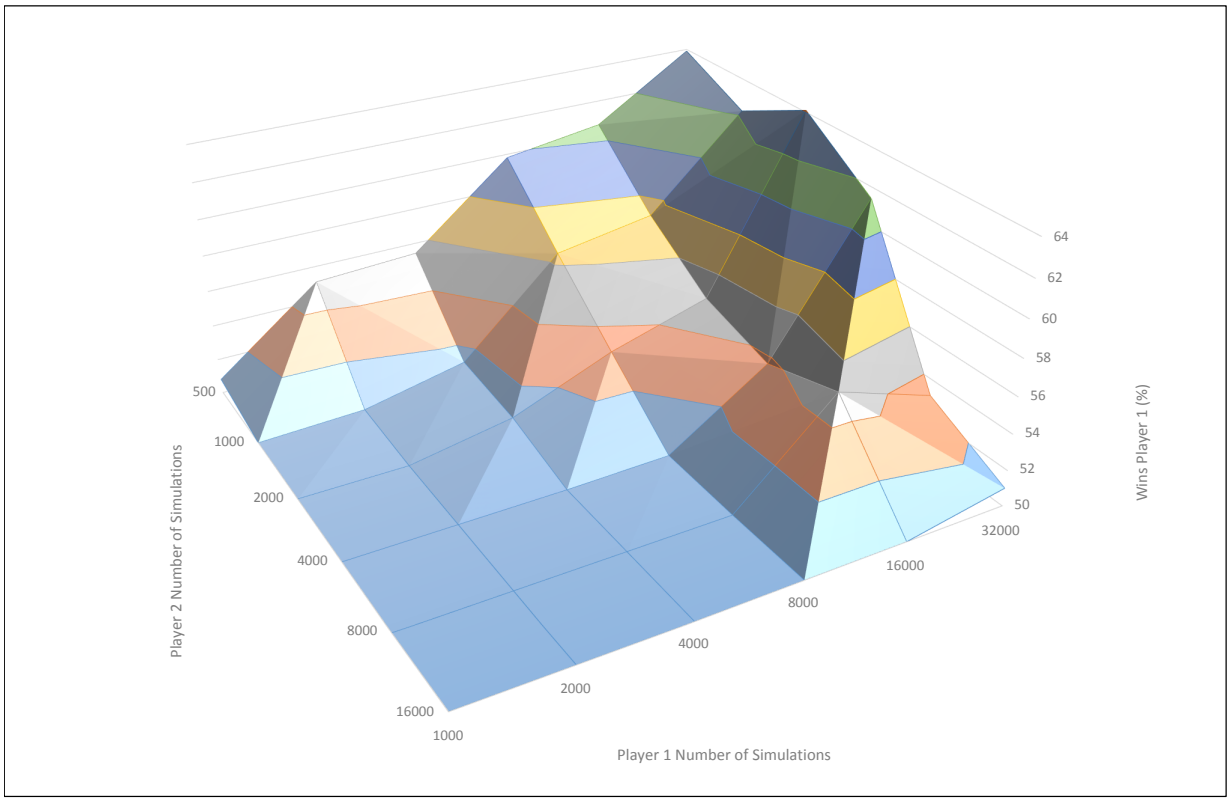

Figure 5.2: Visual representation of Table 5.1

	1000	**2000**	**4000**	**8000**	**16000**
500	61.4	65.5	69.5	69.2	71.8
1000		57.5	62.3	64.7	65.3
2000			54.1	58.5	60.7
4000				52.6	59.4
8000					49.5

Table 5.2: Diminishing return results with prior knowledge enhancement

	1000	**2000**	**4000**	**8000**	**16000**
500	59.3	63.9	68.8	68.5	70.1
1000		57.2	61.4	62.5	63.3
2000			50.8	54.7	59.7
4000				54.1	53.7
8000					47.9

Table 5.3: Diminishing return results with Prior Knowledge and playout strategy enhancements

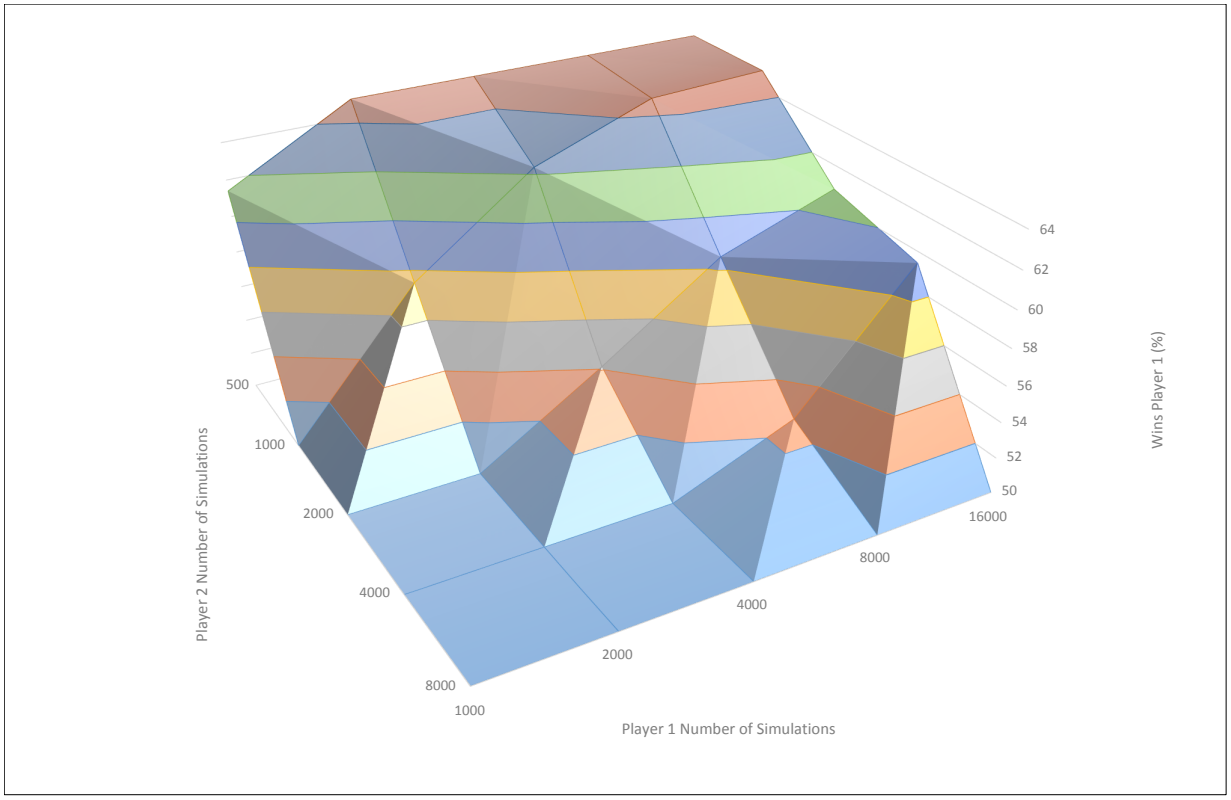

Figure 5.3: Visual representation of Table 5.2

5.4 Flat Monte-Carlo

In this section, two experiments were performed to evaluate the added benefit of the tree when compared to Flat Monte-Carlo. In the first experiment, both agents used no enhancements. In the second experiment, the playout strategy and Prior Knowledge enhancements were added to MCTS. Prior Knowledge has not been added to the Flat Monte-Carlo agent, since it will not have much influence because it initializes the nodes with 100 visits, while each node will be visited several thousand times. The results of these experiments can be found in Table 5.4. As expected, MCTS performs significantly better than Flat Monte-Carlo when using no enhancements. With enhancements, the difference is even higher. This indicates, that the added domain knowledge does help MCTS to assess the different branches of the tree more effectively. Similar results were shown by Lorentz (2012), where MCTS competed against Flat Monte-Carlo with 30 seconds of thinking time.

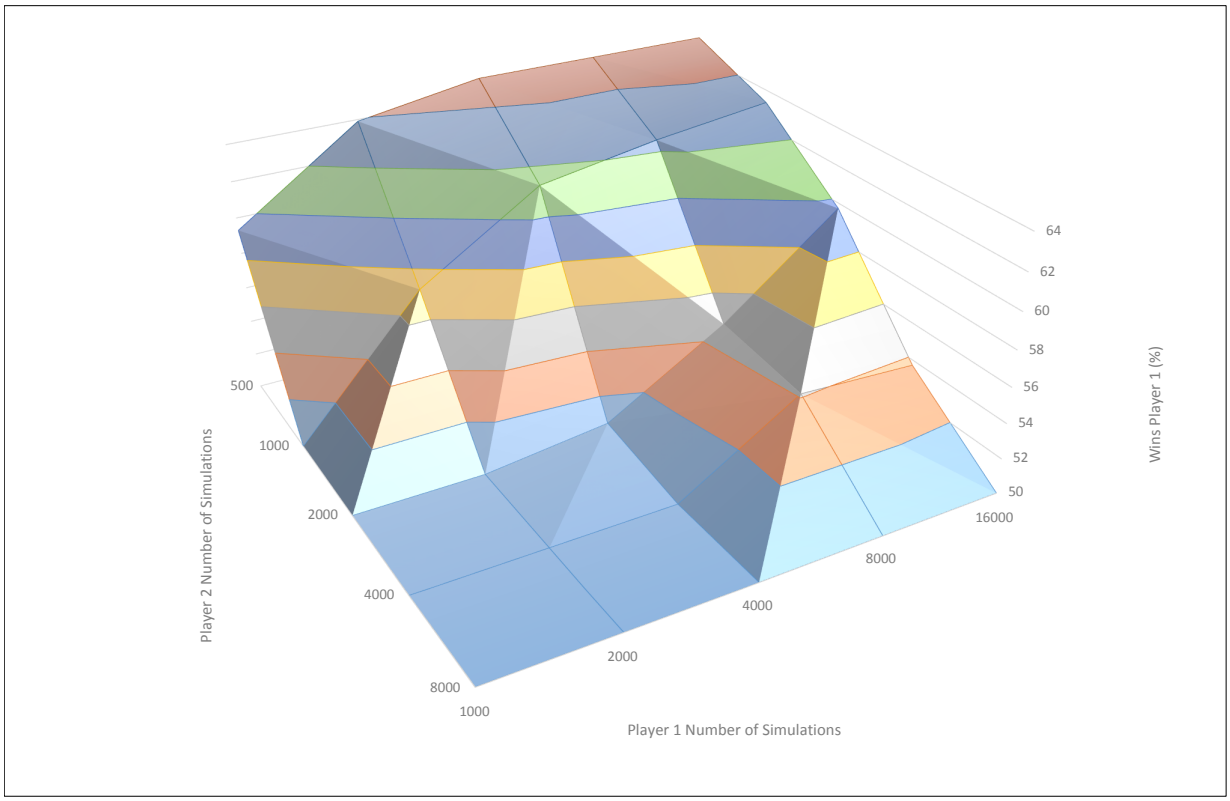

Figure 5.4: Visual representation of Table 5.3

Player 1	**Player 2**	**Wins Player 1 (%)**
MCTS No enhancements	Flat Monte-Carlo No enhancements	58.0 ± 3.1
MCTS + Playout strategy + Prior Knowledge	Flat Monte-Carlo + Playout strategy	62.6 ± 3.0

Table 5.4: Experiment results: MCTS vs. Flat Monte-Carlo

5.5 Playout Strategy and Prior Knowledge

The experiments in this section try to determine the influence of the playout strategy and Prior Knowledge implementation proposed by Lorentz (2012). In order to do so, four experiments were performed as shown in Table 5.5. The first two rows show, how the playout strategy and Prior Knowledge enhancements compare to a MCTS agent without any enhancements. As seen, both enhancements improve the performance of the agent significantly with the playout strategy performing a bit better than Prior Knowledge.

In a second step, the influence of the combination of both enhancements has been tested against the two enhancements alone. These results show that adding additional domain knowledge does not improve the performance as much, if other domain knowledge is already available. In the case of combining Prior Knowledge with the playout strategy, the confidence bounds do not give hard empirical evidence that the combination performs better than only the slayout strategy. However, the combination performs slightly better than only Prior Knowledge. These results indicate again that the playout strategy has a slightly larger impact on performance than Prior Knowledge.

Player 1	**Player 2**	**Wins Player 1 (%)**
Playout strategy	No enhancement	61.1 ± 3.0
Prior Knowledge	No enhancement	58.5 ± 3.1
Playout strategy + Prior Knowledge	Playout strategy	52.3 ± 3.1
Playout strategy + Prior Knowledge	Prior Knowledge	53.8 ± 3.1

Table 5.5: Experiment results: Playout strategy and Prior Knowledge in MCTS

5.6 Progressive History and MAST

In this section, the influence of using history data in the selection and playout steps is examined. At first, a series of experiments has been performed to find a suitable value for the W constant of the modified UCT formula in Progressive History. As initial value for W, 1 has been chosen. Afterwards two agents with the Progressive History enhancements, one with $W = 1$ and one with another value for W were tested against each other. The results for these experiments are shown in Figure 5.5. As can be seen, no significant difference can be observed for these different values, so the initial value of 1 has remained for further experiments.

Five experiments were performed to determine how Progressive History increase the performance of MCTS in EWN. Three experiments with different ε values have been performed to do the same for MAST. Table 5.6 shows the results of these experiments.

As seen, an agent using Progressive History performs slightly stronger than an agent without any enhancements. In the next step, it has been examined how Progressive History relates to the Prior Knowledge enhancement. It can be seen that the combination of Prior Knowledge and Progressive History performs significantly stronger than Progressive History alone. However, when compared to the Prior Knowledge enhancement, no significant enhancement can be observed for combining both approaches. These three experiments indicate, that Progressive History gives an improvement in performance when used solitary, but does not add to the performance of Prior Knowledge.

Prior knowledge however does add to the performance of Progressive History. From this, it can be assumed that the initialization of nodes by the Prior Knowledge enhancement already influences the selection in a similar way as Progressive History. Prior Knowledge has the advantage that it works without any previous simulations. This might explain why it adds to the performance of Progressive History, which needs several simulations before its tables show meaningful statistics. To confirm this assumption, it has also been tested, how Progressive History compares to Prior Knowledge. The result shows that Prior Knowledge performs slightly worse than Progressive History, which supports the previous assumption.

Another approach that has been tested is the combination of Progressive History and the playout strategy enhancement. As before, there seems to be no difference in the agents' performance when comparing Progressive History to no Progressive History.

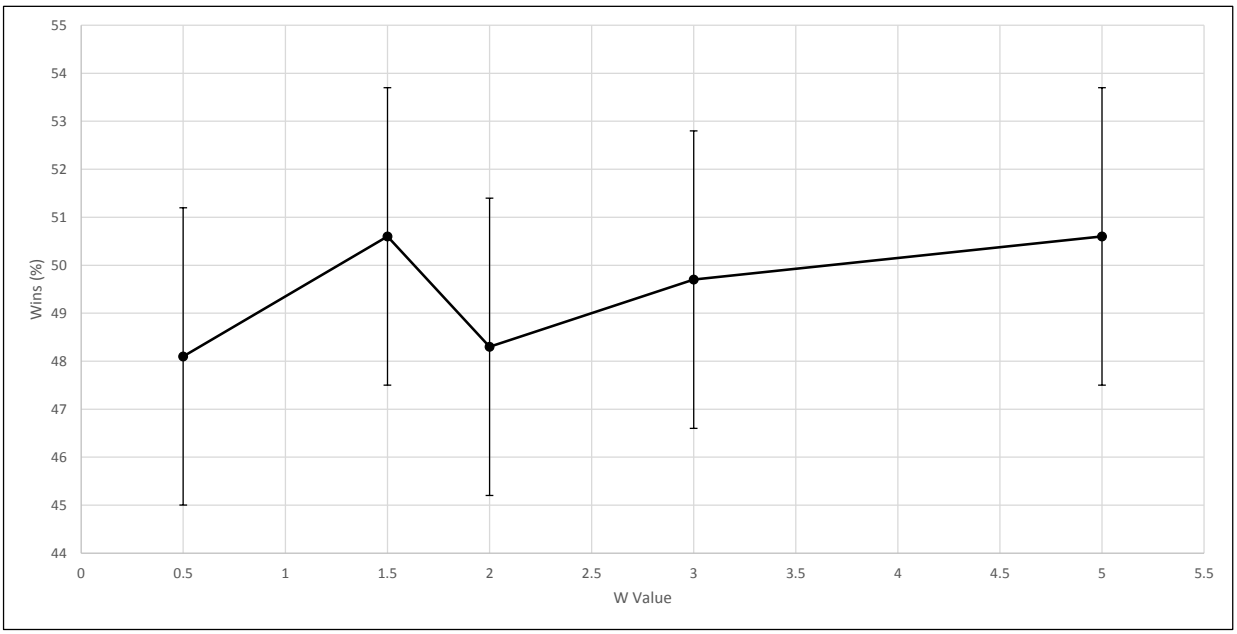

Figure 5.5: Experiment results: W values

The last performed experiments compare an agent with the MAST enhancement to an agent with no enhancements. As MAST was implemented using an ε-greedy approach, three different values have been tried out for ε. The results show no improvement in performance, independently from the chosen value for ε. This indicates that this approach does not work in the context of EWN. The reason for this might lie within the relatively small number of possible moves in the game and in the high simulation number. The moves might be not distinctive enough for MAST to recognize meaningful differences. In EWN, it is hard to find a move that is a bad move in most situations. The strength of a move is very dependent on the current board situation. Also the table could be saturated due to the high simulation number.

Player 1	**Player 2**	**Wins Player 1 (%)**
Progressive History	No enhancement	54.7 ± 3.1
Prior Knowledge + Progressive History	Progressive History	58.0 ± 3.1
Prior Knowledge + Progressive History	Prior Knowledge	51.3 ± 3.1
Progressive History	Prior Knowledge	46.9 ± 3.1
Playout strategy + Progressive History	Playout strategy	49.6 ± 3.1
MAST $\varepsilon = 0$	No enhancement	49.4 ± 3.1
MAST $\varepsilon = 0.05$		49.5 ± 3.1
MAST $\varepsilon = 0.1$		49.7 ± 3.1

Table 5.6: Experiment results: Progressive History and MAST in MCTS

5.7 Variance Reduction

This section discusses the results regarding the Variance Reduction experiments. Two different versions of Variance Reduction have been tested. The first version applies Variance Reduction to each node on the same level, the second version applies Variance Reduction to each node of the tree. These versions are referenced as Variance Reduction per level and Variance Reduction per node respectively. The experiment results can be found in Table 5.7.

At first, both versions have been tested against an agent with no enhancements and thinking time of 1s. The results for this experiment were not distinct at first, which is why the number of experiments has been raised to 10000 in this case. As seen, both versions of Variance Reduction do not seem to improve the performance of MCTS. In the next step, Variance Reduction has been added to an agent with the Prior

Knowledge and playout strategy enhancements and tested against an agent without Variance Reduction. This experiment has been performed to find out if Variance Reduction interacts positively with domain knowledge. Again, the results show no significant change in the performance of MCTS. The last step consisted of an experiment in which both agents haven been limited to 100 simulations per turn. This experiment has been performed to find out, if a high number of simulations reduce the influence of Variance Reduction. As with the previous results, there is no noticeable improvement in performance due to Variance Reduction.

The reason for these results might be explained with the nature of EWN. The upper levels of the search tree are visited numerous times. This has the effect that on average, each branch already used each die roll equally often. In this case, Variance Reduction is not needed to compare different branches of the tree meaningfully. In the lower levels of the search tree, the effect of Variance Reduction should increase, as these branches have been visited only few times. However, in the lower levels of tree, the game also progressed further. In the case of EWN, this means that most likely, several tokens have been already removed from the game. This results in a decreased influence of the die rolls, which subsequently decreases the importance of Variance Reduction.

Player 1	**Player 2**	**Thinking time/ simulation limit**	**Wins Player 1 (%)**
Variance Reduction per level	No enhancement	1s	50.8 ± 1.0
Variance Reduction per node	No enhancement	1s	50.4 ± 1.0
Variance Reduction per level + Prior Knowledge + Playout strategy	Prior Knowledge + Playout strategy	1s	47.3 ± 3.1
Variance Reduction per node + Prior Knowledge + Playout strategy	Prior Knowledge + Playout strategy	1s	50.2 ± 3.1
Variance Reduction per level + Prior Knowledge + Playout strategy	Prior Knowledge + Playout strategy	100 simulations	48.0 ± 3.1
Variance Reduction per node + Prior Knowledge + Playout strategy	Prior Knowledge + Playout strategy	100 simulations	48.8 ± 3.1

Table 5.7: Experiment results: Variance Reduction in MCTS

5.8 Quality-based Rewards

This section's experiments have been performed to assess the performance of Quality-based Rewards. Both agents are using the playout strategy enhancement to make the playouts resemble real games more closely. This should help the Quality-based Reward enhancement to identify exceptional playouts. Three approaches have been used to give bonus rewards. The first approach counts the minimal number of turns, the losing player needs to reach the goal. The second and third approach uses the total game length of every playout, where the second approach rewards below-average long games and the third approach rewards above-average long games.

The results for all three experiments show no significant change in performance when adding Quality-based Rewards. The reason for this could lie within the nature of the game EWN. Figures 5.6 shows the distribution of the game length. 5.7 shows the number of turns the losing player would still have to reach the goal. Both figures also give the average value and standard deviation. As seen, the game length rarely deviates more than 3 turns from the standard deviation. For the number of the turns left the losing player would need to reach the goal, almost all playouts lie within the standard deviation. It is possible that there is not sufficient variety in these values for Quality-based Rewards to take effect.

Player 1	**Player 2**	**Wins Player 1 (%)**
Quality-based reward for many turns left + Playout strategy	Playout strategy	49.5 ± 3.1
Quality-based reward for small game length + Playout strategy	Playout strategy	50.2 ± 3.1
Quality-based reward for large game length + Playout strategy	Playout strategy	49.9 ± 3.1

Table 5.8: Experiment results: Quality-based Rewards in MCTS

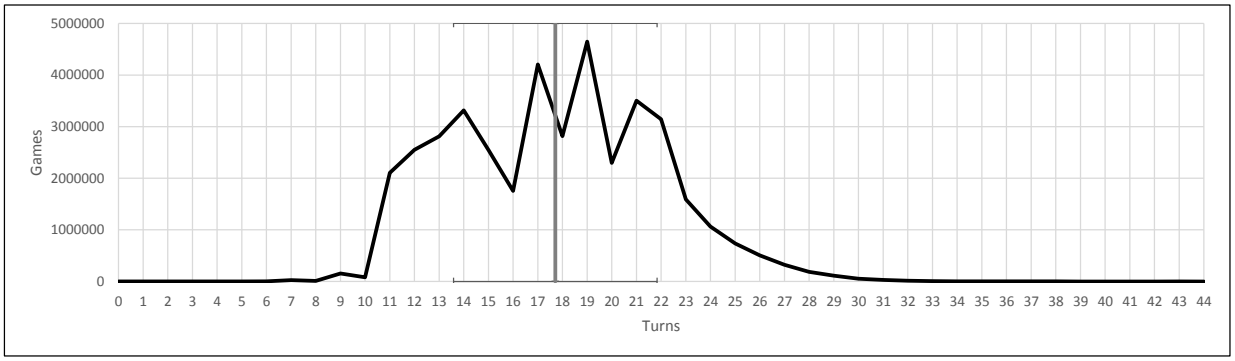

Figure 5.6: Total game length in playouts

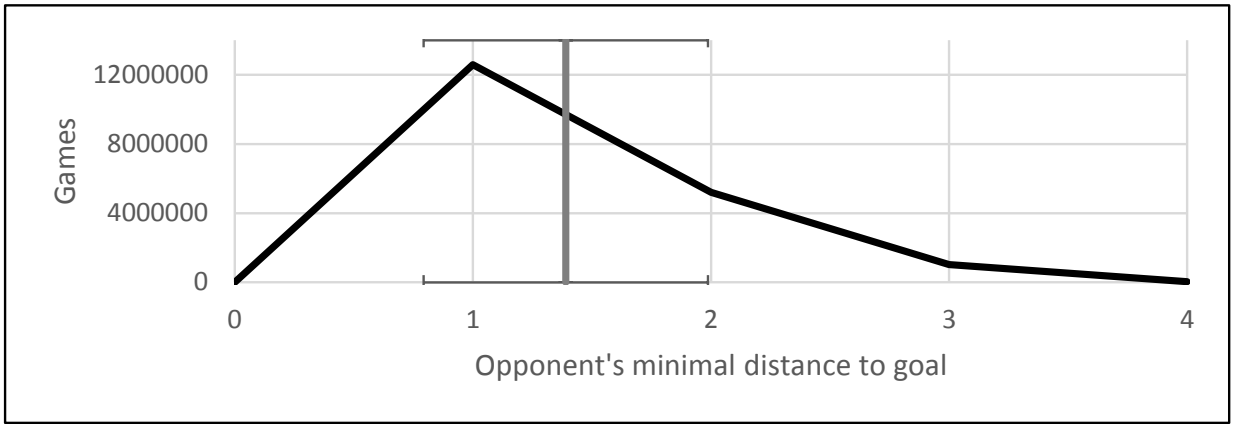

Figure 5.7: Number of turns the losing player would need to reach goal

5.9 MeinStein's Evaluation Function for Playouts

The experiments in this section are aimed towards evaluating MeinStein's evaluation function used in the playout step in an ε-greedy way. The value for ε has been chosen to be 0.05. The results of these experiments can be found in Table 5.9. The results show, that the evaluation function increases the performance of MCTS when compared to no enhancements. When compared against an agent with the playout strategy enhancement, no significant difference in performance can be observed, indicating that the playout strategy and MeinStein's evaluation function often choose the same move during the playouts. When Prior Knowledge is added to both agents, no difference can be observed between MeinStein's evaluation function and the playout strategy.

It has also been examined how much the playout enhancements influence the simulation number of the agent. Table 5.10 shows the average number of simulations in the first second and on the first turn of a game. The results show that the playout strategy has only a small influence on the simulation number, reducing it by 7.5%. MeinStein's evaluation function however reduces the number of simulations drastically, by 85%. This indicates that the evaluation function is well suited to estimate the strength of certain moves. This can be seen from the fact that the agent with the playout strategy and the agent with MeinStein's evaluation function perform equally even though the agent with the evaluation function has much fewer simulations.

Player 1	**Player 2**	**Wins Player 1 (%)**
MeinStein Evaluation Function	No enhancement	53.3 ± 3.1
MeinStein Evaluation Function	Playout strategy	53.0 ± 3.1
Prior Knowledge + MeinStein Evaluation Function	Playout strategy + Prior Knowledge	49.1 ± 3.1

Table 5.9: Experiment results: MeinStein's Evaluation Function

Enhancement	**Number of Simulations**	**Decrease (%)**
No enhancement	51942	0
Playout strategy	47808	7.5
MeinStein Evaluation Function	7837	85

Table 5.10: Number of Simulations for different enhancements, first second, first turn

5.10 MeinStein

In this section, MeinStein is tested against the strongest version of the agent written for this thesis. That agent uses the Prior Knowledge and playout strategy enhancements. The experiments have been performed with thinking times of 1, 3.5 and 5 seconds for both sides. Table 5.11 shows the results of these experiments. The results show, that the agent written for this thesis can compete with MeinStein. Different thinking times do not seem to influence this result significantly. For MeinStein, the reason for that probably lies within the search depth, which is reached in the given time limit. For the first turn, the reached search depth is 8 for 1s and 3.5s. For 5s, the search depth is also 8.3 on average. This means, that in most cases, MeinStein is not able to benefit from the additional thinking time. The MCTS agent is maybe not able to perform better with higher thinking times because of the law of the diminishing returns.

In a second step, experiments were performed to assess, how the two aforementioned enhancements influence the performance when compared to MeinStein. The experiment results show, that both enhancements significantly improve the performance of the MCTS agent. This shows, that these enhancements do not only work when used in self-play, but also against entirely independent agents.

Player 1	Player 2	Thinking Time	Wins Player 1 (%)
MeinStein	Prior Knowledge + Playout strategy	1s	47.8 ± 3.1
		3.5s	49.3 ± 3.1
		5s	49.9 ± 3.1
	No enhancement	1s	57.5 ± 3.1
	Playout strategy		50.5 ± 3.1
	Prior Knowledge		53.2 ± 3.1

Table 5.11: Experiment results: MeinStein vs. MCTS

5.11 Discussion

This section gives discussion of all previous experiments. It has been determined that it is difficult to find enhancements for MCTS, which significantly improve its performance in EWN. The only enhancements that are proven to work for EWN are the Prior Knowledge and playout strategy enhancements introduced by Lorentz (2012). Other enhancements like Variance Reduction or MAST do not seem to influence the performance of MCTS significantly. Only Progressive History shows a significant improvement of this performance when used in isolation. Once combined with either of the domain knowledge enhancements, this improvement cannot be observed anymore.

The reason for these results is likely to lie within the high chanciness of the game, as discussed in Section 2.5. As mentioned there, once both players play reasonably strong, the outcome of the game is primarily influenced by chance and not the players' skill (Erdmann, 2009). This means that if a player would play much stronger than the opponent, he would probably still win only a few more games. Assuming, a player would win on average 51% of all games, 27000 matches would be needed to be 95% certain that the stronger player actually won the majority of the matches. With 6700 games, it is only 50% certain that the stronger player actually won more games (Turner, 2012). Summarizing, this means that it might be that some of the enhancements tested previously are actually working, but many more games would be needed to prove that. Still, the playout strategy and Prior Knowledge enhancements show that some enhancements can increase significantly the performance of MCTS in EWN.

Chapter 6

Conclusion and Future Research

This chapter gives a summary to this thesis and answers the problem statement and research questions from Chapter 1. Afterwards, ideas for future research within this topic are discussed.

Chapter contents: Conclusion, Answer of Research Questions and Problem Statement, and Future Research

6.1 Summary

An MCTS Agent for EinStein würfelt nicht! tried to find out, how MCTS can be adapted to work in game with chance events such as EWN, and which enhancements can be used to improve its performance. Therefore, an introduction of the general topic was given, describing the influence of computer players in games in the past. Different search techniques were introduced before the problem statement and research questions have been formulated.

In the next chapter, the history and rules of EWN have been explained. The distinct features of the game were given, including a description of strategies to consider when playing the game. The upper bound for the state-space complexity of the game has been shown to be $\sim 2.68 \times 10^{15}$ and the game-tree complexity has been shown to be $\sim 2.18 \times 10^{22}$. The chanciness of the game was introduced and it has been concluded that the difference in performance of two strong players can be difficult to prove.

The third chapter has given a detailed overview of the MCTS algorithm. The chapter started with introducing MCTS with its history and some of its accomplishments. Next, each algorithm step and the UCT formula have been described. The rest of that chapter explained the concepts of various enhancements to MCTS, including the playout strategy, Prior Knowledge and Variance Reduction.

The next chapter focused on how MCTS and its enhancements have been adapted for EWN. At first, it has been described how MCTS was modified to properly handle the die rolls of EWN. After that, the structure of a single node has been discussed and it has been explained how the node has been designed to be memory efficient. Afterwards, the specific adaptation of the different enhancements was described. The chapter ends with the introduction of MeinStein, and how it can be used to design a playout strategy.

Chapter 5 was used to give the results of various experiments and explains there meaning. The chapter begins with a description of the experimental setup, providing hardware and software specifications. Next, is has been shown, that the exploration factor of the UCT formula has a smaller impact on the performance than expected, concluding that values between 1 and 2 all lead to relatively equal results.

Afterwards, the results of several experiments regarding diminishing returns have been discussed. The experiments showed that with more domain knowledge, higher simulation numbers are more beneficial. As expected, at some point the law of the diminishing returns has been observed.

The next experiments have proven that MCTS outperforms Flat Monte-Carlo. It was shown that this effect increases when domain knowledge is added. The effect of domain knowledge has also been proven in the following section, were it was shown that both the playout strategy and Prior Knowledge enhancements work for EWN.

In the following experiments, it has been shown that Progressive History can improve the performance of MCTS when used in isolation, but does not show noticeable improvements when used in combination with domain knowledge. The second enhancement using history knowledge, MAST, has been proven to not alter the performance of MCTS.

Variance reduction was then examined in the next section. Various experiments have shown that Variance Reduction does not work in the case of EWN, which was explained by the nature of the game. Similar results have been described for the Quality-based Rewards enhancement. It has been shown that the board configuration at the end of the game and the game length are not feasible as quality measurements.

In the last part of Chapter 5, MeinStein's evaluation function has been assessed as a playout strategy and MeinStein was used as a benchmark for the agent written for this thesis. It has been shown that MeinStein's evaluation function slightly improves the performance of an agent without other enhancements but does not outperform an agent with the playout strategy. It has been shown that the agent written for this thesis performs similar to MeinStein for various time constraints.

In the end, it can be concluded that a combination of the playout strategy and Prior Knowledge enhancements works best for EWN. Some other enhancements work only when used in isolation but there effect diminishes when tried to combine with the aforementioned enhancements. A likely explanation for this lies within the high chanciness as described in Section 2.5. Using both domain knowledge enhancements, the agent already performs strong enough that a subtle increase in performance due to other enhancements is lost to the high chanciness.

6.2 Answering the Research Questions

This section will answer the research questions of this thesis, as given in Section 1.3.

How can MCTS be made suitable for games with chance elements?
In the case of EWN, the chance element was a die roll. To incorporate this die roll into MCTS, the nodes have to be expanded such that they can indicate if a certain die roll would lead to them. The selection step also has to be modified. Before a child is selected, a random die roll is generated. When choosing a child node, only the children are considered, which are fitting the previously made die roll.

What, if any, is the benefit of Variance Reduction in the case of EWN?
In the case of EWN, Variance Reduction does not seem to be beneficial for an MCTS agent. In EWN, the importance of the die roll decreases with the game's progression. At the beginning of the game, where the die roll is still important, Variance Reduction has no benefit because a higher number of simulations already makes the branches comparable.

What, if any, is the benefit of playout strategies in the case if EWN?
Playout strategies have proven to be the most beneficial for MCTS and EWN. In every performed experiment using the proposed playout stratgegy of Lorentz (2012), it significantly improved the performance of the agent independently from other used enhancements. However, other playout related enhancements such as MAST or MeinStein's evaluation function do not seem to give a further improvement.

What, if any, is the benefit of Selection strategies for MCTS in the case of EWN?
For Selection strategies, Progressive History has shown that it can improve the performance of MCTS in EWN when used in isolation. However, when added to an agent with domain knowledge, no significant improvement was observed. Prior Knowledge initialization as introduced by Lorentz (2012) has been shown to enhance the performance of MCTS in combination with all other tested enhancements. Quality-based Rewards do not seem to influence the agent's performance in EWN.

6.3 Answering the Problem Statement

In this section, the problem statement from Section 1.3 is answered.

How can we develop an MCTS agent for "EinStein würfelt nicht!", which performs as strong as possible and in a feasible amount of time?
Developing an MCTS agent for EWN consisted of two parts. First, MCTS had to be adapted to work with EWN. This was done by modifying MCTS to work with die rolls and optimizing it to be memory and computational efficient. The second step consisted of implementing various enhancements and assessing their performance. The enhancements, which significantly improved the agent's performance, were the playout strategy and Prior Knowledge enhancements. Both have been added to form the strongest possible agent based on the experimental findings. Benchmarking that agent against the state-of-the-art EWN agent MeinStein has proven that MCTS can compete with established Minimax algorithms in EWN.

6.4 Future Research

In this section, ideas for future research are proposed. While the topic was researched in depth, there are still research opportunities available.

As noted previously, the high chanciness of EWN makes it difficult to recognize, if one player is superior to the other player. It is possible, that some of the discussed enhancements actually give a small advantage so it could be beneficial to run all experiments more often to discover such advantages. Especially Progressive history could be interesting to test more extensively, as it actually improved the agent's performance when compared to no enhancements.

It is also possible that some enhancements only work with higher thinking times. It could also be reassessed how this affects the performance of the agent against MeinStein, as it is possible that MCTS and Expectimax react differently to higher thinking times.

Another possible idea for future research is to tweak more of the values used in the various enhancements. Reasonable approaches would be to tune the values for the playout strategy and Prior Knowledge enhancements, as these enhancements are the most promising ones, but the W value for the Progressive History enhancement would also be a good choice, as that enhancement seemed to increase performance to some extent, as mentioned above.

There are also some variants of the game with minor rule changes. On Little Golem (2015) there is a variant introduced, in which moving backwards is allowed for capturing. Another variant on Little Golem forces the players to remove any token that was moved to the center square of the board. It could be interesting to see, how these rule changes affect MCTS and its enhancements.

There is also a variant on a 6×6 board with 10 tokens (2-12, without 7), where two six-sided dice are rolled per turn. In this variant, the starting positions could be far more important, as some tokens (such as token 6) have a higher probability of being moved due to the normal distribution of the two die rolls.

References

Allis, L. Victor (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, University of Limburg, Maastricht, The Netherlands. [7]

Arneson, B., Hayward, R.B., and Henderson, P. (2010). Monte Carlo Tree Search in Hex. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 2, No. 4, pp. 251–258. [9]

Bouzy, Bruno and Chaslot, Guillaume M.J-B. (2006). Monte-Carlo Go Reinforcement Learning Experiments. *IEEE 2006 Symposium on Computational Intelligence in Games* (eds. Sushil J. Louis and Graham Kendall), p. 187–194, IEEE, USA, Reno. [12]

Browne, Cameron, Powley, Edward, Whitehouse, Daniel, Lucas, Simon, Cowling, Peter I., Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–49. [1, 2, 9]

Campbell, Murray, Hoane Jr, A. Joseph, and Hsu, Feng-hsiung (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1, pp. 57–83. [1]

Chaslot, Guillaume M.J-B., Winands, Mark H.M., Herik, H. Jaap van den, Uiterwijk, Jos W.H.M., and Bouzy, B. (2008). Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [9, 10]

Coulom, Rémi (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computer and Games: 5th International Conference* (eds. Paolo Ciancarini, H. Jaap van den Herik, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83, Springer. [2, 9]

Cowling, Peter I., Powley, Edward J., and Whitehouse, Daniel (2012). Information Set Monte Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 120–143. [13]

Erdmann, Jakob (2009). Chanciness: Towards a Characterization of Chance in Games. *ICGA Journal*, Vol. 32, No. 4, pp. 187–205. [7, 8, 29, 39]

Fang, Haw ren, Glenn, James, and Kruskal, Clyde P. (2008). Retrograde Approximation Algorithms for Jeopardy Stochastic Games. *ICGA Journal*, Vol. 31, No. 2, pp. 77–96. [1]

Finnsson, Hilmar and Björnsson, Yngvi (2008). Simulation-Based Approach to General Game Playing. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008* (eds. Dieter Fox and Carla P. Gomes), pp. 259–264, AAAI Press, Chicago, Illinois, USA. [13]

Gelly, Sylvain and Silver, David (2007). Combining Online and Offline Knowledge in UCT. *Proceedings of the International Conference on Machine Learning (ICML)* (ed. Zoubin Ghahramani), pp. 273–280, ACM. [12]

Gelly, Sylvain, Kocsis, Levente, Schoenauer, Marc, Sebag, Michèle, Silver, David, Szepesvári, Csaba, and Teytaud, Olivier (2012). The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, Vol. 55, No. 3, pp. 106–113. [9]

Heinz, Ernst A. (2001). Self-Play, Deep Search and Diminishing Returns. *ICGA Journal*, Vol. 24, No. 2, pp. 75–79. [21]

Knuth, Donald E. and Moore, Ronald W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [2]

Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293, Springer. [2, 9, 10]

Krabbenbos, Jan (2015). Stormy's Corner. `http://www.csvn.nl/index.php/download/stormys-corner`. [Online; accessed 09.06.2015]. [18]

Lanctot, Marc, Saffidine, Abdallah, Veness, Joel, Archibald, Chris, and Winands, Mark H.M. (2013). Monte Carlo *-Minimax Search. *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 580–586. [2]

(2015). Little Golem. `http://www.littlegolem.net/jsp/games/gamedetail.jsp?gtid=einstein`. [Online; accessed 30.06.2015]. [2, 33]

Lorentz, Richard J. (2008). Amazons Discover Monte-Carlo. *Computers and Games* (eds. H.Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 13–24. Springer. [9]

Lorentz, Richard J. (2012). An MCTS Program to Play EinStein Würfelt Nicht! *Advances in Computer Games* (eds. H. Jaap van den Herik and Aske Plaat), Vol. 7168 of *Lecture Notes in Computer Science*, pp. 52–59, Springer. [v, 2, 5, 12, 16, 17, 22, 24, 29, 32]

Michie, Donald (1966). Game-playing and Game-Learning Automata. *Advances in Programming and Non-Numerical Computation* (ed. Leslie Fox), pp. 183–200. Pergamon Press. [2, 18]

Neumann, John von and Morgenstern, Oskar (1944). *Theory of Games and Economic Behavior*. Princeton University Press. [2]

Nijssen, J.A.M. and Winands, Mark H.M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat), Vol. 6151 of *Lecture Notes in Computer Science (LNCS)*, pp. 238–249, Springer, Berlin Heidelberg, Germany. [12]

Nijssen, Pim and Winands, Mark H.M. (2012). Monte Carlo Tree Search for the Hide-and-Seek Game Scotland Yard. *Computational Intelligence and AI in Games, IEEE Transactions on*, Vol. 4, No. 4, pp. 282–294. [2]

Pepels, Tom, Tak, Mandy J.W., Lanctot, Marc, and Winands, Mark H.M. (2014). Quality-based Rewards for Monte-Carlo Tree Search Simulations. *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)* (eds. Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan), Vol. 263 of *Frontiers in Artificial Intelligence and Applications*, pp. 705–710. [2, 13, 18]

Powley, Edward J., Whitehouse, Daniel, and Cowling, Peter I. (2013). Bandits all the way down: UCB1 as a simulation policy in Monte Carlo Tree Search. *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pp. 81–88, IEEE, Niagara Falls, Ontario, Canada. [16]

Robilliard, Denis, Fonlupt, Cyril, and Teytaud, Fabien (2014). Monte-Carlo Tree Search for the Game of "7 Wonders". *Computer Games* (eds. Tristan Cazenave, Mark H. M. Winands, and Yngvi Björnsson), Vol. 504 of *Communications in Computer and Information Science*, pp. 64–77. [21]

Schaffer, Simon (1999). Enlightened Automata. *The Sciences in Enlightened Europe* (eds. William Clark, Jan Golinski, and Simon Schaffer), pp. 126–165, University of Chicago Press. [1]

Shannon, Claude E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 314, pp. 256–275. [7]

Tak, Mandy J.W., Winands, Mark H.M., and Bjornsson, Yngvi (2012). N-Grams and the Last-Good-Reply Policy applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 2, pp. 73 – 83. [13]

Tesauro, Gerald (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. [1]

Turner, Wesley (2012). EinStein Würfelt Nicht - An Analysis of Endgame Play. *ICGA Journal*, Vol. 35, No. 2, pp. 94–102. [5, 7, 18, 29]

Veness, Joel, Lanctot, Marc, and Bowling, Michael (2011). Variance Reduction in Monte-Carlo Tree Search. *Advances in Neural Information Processing Systems 24 (NIPS)* (eds. J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, and K.Q. Weinberger), pp. 1836–1844, NIPS. [13]

List of Tables

List of Figures