KNOWLEDGE-BASED MONTE-CARLO TREE SEARCH IN HAVANNAH

J.A. Stankiewicz

Master Thesis DKE 11-05

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. M.H.M. Winands
Dr. ir. J.W.H.M. Uiterwijk
Dr. M.P.D. Schadd
H. Baier, M.Sc.

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
July 2011

Preface

The research of this thesis has been performed as part of the Artificial Intelligence Master programme at the Department of Knowledge Engineering at Maastricht University, under the supervision of Dr. Mark Winands.

Havannah is a game which is known to be difficult for computers, due to the nature of the game. Traditional search algorithms such as $\alpha\beta$-search, generally give poor results. Attempts with Monte-Carlo Tree Search have been more successful. As the goal of the game is to form certain patterns, the subject of this thesis is how to use knowledge about these patterns, in order to improve the performance of a Monte-Carlo Tree Search player for Havannah.

There are a number people who I would like to thank. First, I would like to thank Dr. Mark Winands for supervising this thesis. The feedback that he gave me during the research was useful and it helped me to overcome several problems I had encountered. I would also like to thank Bart Joosten and Joscha-David Fossel for creating the game engine in which the enhancements discussed in this thesis were implemented. Finally, I would also like to thank my friends and family for their support during the period I was working on this thesis.

Jan Stankiewicz
Sittard, June 2011

Summary

The research discussed in this master thesis investigates how the performance of a Monte-Carlo Tree Search (MCTS) player for Havannah can be improved, by using knowledge about patterns during the search. The problem statement of this thesis is therefore: "*How can one use knowledge in an MCTS engine for Havannah, in order to improve its performance?*".

To answer this question, the research is split into two parts. The first part aims to improve the balance between exploration and exploitation during the *selection* step of the MCTS algorithm. A better balance means that less time is wasted on trying moves which are unlikely to be good anyway. The balance is improved by initializing the visit and win counts of new nodes in the MCTS tree, using pattern knowledge. By biasing the selection towards certain moves, a significant improvement in the performance is achieved.

The second part of the research aims to improve the play-out of the MCTS algorithm. Using local patterns to guide the play-out does not lead to an improvement in the performance. The Last-Good-Reply (LGR) technique generally gives a significant improvement, although it depends on which LGR variant is used. Using N-grams to guide the play-out also results in a significant increase in the win percentage. Combining N-grams with LGR leads to a further improvement, although in most cases, it is not significantly better than using only N-grams.

Experiments show that the best overall performance is achieved when combining visit and win count initialization with LGR and N-grams. In the best case, a win percentage of 77.5% can be obtained against the original Havannah program.

Contents

Preface iii

Summary v

Contents vii

List of Figures ix

List of Tables xi

1 Introduction 1
- 1.1 Games and AI 1
- 1.2 The Rules of Havannah 2
- 1.3 Havannah and AI 2
- 1.4 Problem Statement and Research Questions 4
- 1.5 Thesis Overview 5

2 Monte-Carlo Tree Search 7
- 2.1 The MCTS Algorithm 7
  - 2.1.1 Selection 7
  - 2.1.2 Expansion 8
  - 2.1.3 Simulation 8
  - 2.1.4 Backpropagation 9
- 2.2 Enhancements to MCTS 9
  - 2.2.1 Rapid Action-Value Estimation 9
  - 2.2.2 Progressive History 10
  - 2.2.3 Havannah-Mate 10
  - 2.2.4 Last-Good-Reply 10
  - 2.2.5 N-grams 11

3 Pattern Knowledge 13
- 3.1 Patterns in Havannah 13
- 3.2 Local Patterns 14
- 3.3 Assigning Weights 15
- 3.4 Dead Cells 17
- 3.5 Roulette Wheel Selection 17
- 3.6 ϵ-Greedy Selection 18
- 3.7 Initializing Visit and Win Counts 18
  - 3.7.1 Joint and Neighbour Moves 18
  - 3.7.2 Local Connections 19
  - 3.7.3 Edge and Corner Connections 20

4 Experiments and Discussion **21**
4.1 Experimental Setup 21
4.2 Local Patterns During Play-out 21
4.3 Last-Good-Reply 23
4.4 N-grams 24
4.4.1 Thresholding 25
4.4.2 Averaging 26
4.4.3 N-grams Combined With LGR 26
4.5 Dead Cells 26
4.6 Initializing Visit and Win Counts 27
4.6.1 Joint and Neighbour Moves 27
4.6.2 Local Connections 28
4.6.3 Edge and Corner Connections 28
4.6.4 Combination 29
4.7 Summary 31

5 Conclusions **33**
5.1 Answering the Research Questions 33
5.2 Answering the Problem Statement 34
5.3 Future Research 35

References **37**

Samenvatting **39**

List of Figures

1.1 The three possible connections to win the game. 2

2.1 The four steps of the Monte-Carlo Tree Search algorithm. 8
2.2 A simulation in MCTS. 10

3.1 An example of a fork frame. 13
3.2 Example of what appears to be, but is actually not a ring frame. 14
3.3 An example of a ring frame. 14
3.4 An example pattern with indicated cell numbers. 15
3.5 Distribution of weights on a base-5 board. 16
3.6 An example of a pattern with a high weight. 16
3.7 Dead cells in Havannah. 17
3.8 Roulette Wheel Selection. 18
3.9 A joint move and two neighbour moves. 19
3.10 Groups of surrounding stones. 19
3.11 Edge and corner connections. 20

4.1 Pattern frequencies on a base-5 board. 22
4.2 Weight distribution of the 50 most frequent patterns. 22

List of Tables

1.1 State-space and game-tree complexity of Havannah and several other games. 3

4.1 Pattern play-out win percentages. 21
4.2 Performance of the four LGR variants, with 1 second thinking time. 23
4.3 Performance of the four LGR variants, with 3 seconds thinking time. 23
4.4 Performance of the four LGR variants, with 5 seconds thinking time. 23
4.5 Performance of the four LGR variants, with 1 second thinking time and LGR reset. 24
4.6 Performance of the four LGR variants, with 3 seconds thinking time and LGR reset. . . . 24
4.7 Performance of the four LGR variants, with 5 seconds thinking time and LGR reset. . . . 24
4.8 The win percentages of N-grams for different values of γ. 25
4.9 The influence of applying thresholds to N-grams. 25
4.10 The win percentages of N-gram2 with averaging. 26
4.11 The win percentages of LGR combined with N-grams for $\gamma = 0$. 26
4.12 The performance of MCTS when omitting dead cells, with pattern play-out. 26
4.13 The performance of MCTS when omitting dead cells, with random play-out. 27
4.14 Biasing the selection towards joint and neighbour moves. 27
4.15 Biasing the selection towards joint and neighbour moves, with the addition of LGRF-2 and N-gram1. 27
4.16 Biasing the selection and simulation towards joint and neighbour moves. 28
4.17 Biasing the selection towards local connections. 28
4.18 Biasing the selection towards local connections, with the addition of LGRF-2 and N-gram1. 28
4.19 Biasing the selection towards edge and corner connections. 28
4.20 Biasing the selection towards edge and corner connections, with the addition of LGRF-2 and N-gram1. 29
4.21 Biasing towards edge and corner connections, when also taking into account the location of the proposed move itself. 29
4.22 The performance of MCTS when the three visit and win count initialization methods are combined. 30
4.23 The performance of MCTS when the three visit and win count initialization methods are combined, with the addition of LGRF-2 and N-gram1. 30
4.24 The performance the combination of the three methods, when playing against a player where the selection is only biased towards joint and neighbour moves. 30

Chapter 1

Introduction

This chapter describes the rules of Havannah and gives an overview of the related research. First, a brief overview of games and AI is given. The second section explains the rules of Havannah, after which the third section gives an overview of related research. Finally, the problem statement and research questions of this thesis are discussed and an overview of the thesis is given.

Chapter contents: Introduction — Games and AI, The Rules of Havannah, Havannah and AI, Problem Statement and Research Questions, Thesis Overview

1.1 Games and AI

Ever since the invention of computer technology, people have wondered whether it would be possible for a computer to play games on such a level that it would outperform humans, or at least to be a solid opponent. This has proven to be quite a challenge. Various games have been the domain of research throughout the years, with certain games giving better results than others.

Games can be categorized according to several properties. First, there are 2-player vs. multi-player games. Second, there are games with perfect information and imperfect information. Perfect information refers to whether the player has access to all the information about a game. For example, Chess is a perfect-information game, because the player can see all the pieces at all times. Imperfect information means that several variables are hidden from the player. Poker is an example of an imperfect-information game, because the player only knows his own cards, but does not know the cards of his opponents. Finally, there are deterministic and non-deterministic games. A deterministic game is a game where, given all the available information, the outcome of each possible action is known. For example, Chess is a deterministic game, while Poker is a non-deterministic game.

Halfway through the previous century, Shannon (1950) and Turing (1953) were the first to describe a chess playing computer program. Research carried out in the following decades eventually resulted in the famous Deep Blue chess computer, which beat world champion Garry Kasparov in 1997 (Hsu, 2002). Other games, such as Awari (Bal and Romein, 2003), Checkers (Schaeffer, 2007) and Connect Four (Allis, 1988) have been solved, which means that, when every player plays optimally, the outcome of the game and the strategy to achieve it is known.

There are also games, such as Go and Havannah, which are difficult for computers. Computer programs for these games perform quite weakly compared to human players, especially on larger board sizes. Such games generally have a large number of possible moves, and their game states are often difficult to evaluate. These are the main reasons why $\alpha\beta$-search performs poorly when applied to these games, as its performance is dependent on the quality of the evaluation function and the width of the game tree.

1.2 The Rules of Havannah

Havannah is a turn-based two-player deterministic perfect-information game invented by Christian Freeling in 1979 (Freeling, 2003). It is played on a hexagonal board, often with a *base* of 10, meaning that each side has a length of 10 cells. One player uses white stones; the other player uses black stones. The player who plays with white stones starts the game. Each turn, a player places one stone of his colour on an empty cell. The goal is to form one of the three possible winning connections. These connections are shown in Figure 1.1.

- **Bridge**: A connection that connects any two corner cells of the board.
- **Fork**: A connection that connects three sides. Corner cells do not count as side cells.
- **Ring**: A connection that surrounds at least one cell. The cells surrounded by a ring may be empty or occupied by white or black stones.

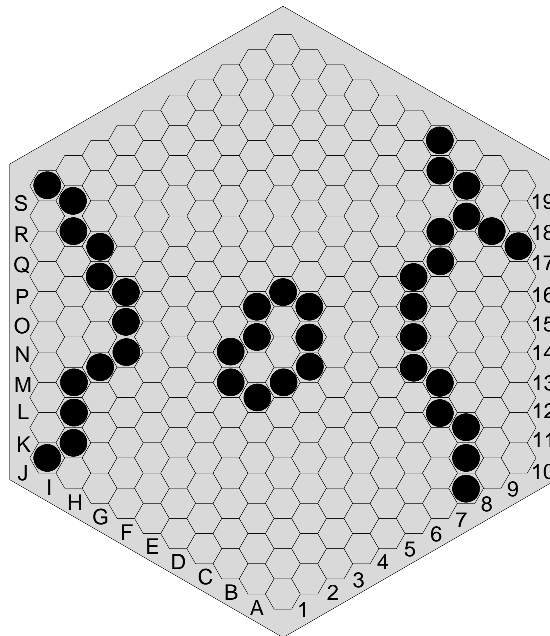

Figure 1.1: The three possible connections to win the game. From left to right: a bridge, a ring and a fork.

Because White has an advantage being the starting player, the game is often started using the swap rule. One of the players places a white stone on the board after which the other player may decide whether he or she will play as white or black. It is possible for the game to end in a draw, although this is quite unlikely.

1.3 Havannah and AI

There are several aspects that make Havannah a difficult game for computers (Teytaud and Teytaud, 2010a).

- No straightforward evaluation function. It is hard to determine how useful a particular cell is. A cell that does not look useful now, may turn out to be useful in the future and vice versa.
- No general pruning rule for moves, due to a similar reason as described above.

- Large branching factor. On a board with a base of 10, there are already 271 possible opening moves, which implies that Havannah has a wide game tree.

These aspects are the reason why common search algorithms, such as $\alpha\beta$-search (Knuth and Moore, 1975), give poor results. As a comparison, Table 1.1 shows the state-space and game-tree complexity of Havannah with base-10 and several other games (Van den Herik, Uiterwijk, and Van Rijswijck, 2002; Joosten, 2009).

Game	State-space complexity	Game-tree complexity
Awari	10^{12}	10^{32}
Checkers	10^{21}	10^{31}
Chess	10^{46}	10^{123}
Chinese Chess	10^{48}	10^{150}
Connect-Four	10^{14}	10^{21}
Dakon-6	10^{15}	10^{33}
Domineering (8×8)	10^{15}	10^{27}
Draughts	10^{30}	10^{54}
Go (19×19)	10^{172}	10^{360}
Go-Moku (15×15)	10^{105}	10^{70}
Havannah (Base-10)	10^{127}	10^{157}
Hex (11×11)	10^{57}	10^{98}
Kalah (6,4)	10^{13}	10^{18}
Nine Men's Morris	10^{10}	10^{50}
Othello	10^{28}	10^{58}
Pentominoes	10^{12}	10^{18}
Qubic	10^{30}	10^{34}
Renju (15×15)	10^{105}	10^{70}
Shogi	10^{71}	10^{226}

Table 1.1: State-space and game-tree complexity of Havannah and several other games.

Among the games shown in the table, Havannah is one of the more complex games. Go is the only game with both a higher state-space and game-tree complexity.

While common $\alpha\beta$-algorithms generally do not give satisfactory results, there have been several attempts based on Monte-Carlo Tree Search (MCTS) (Coulom, 2007; Kocsis and Szepesvári, 2006). The MCTS algorithm does random simulations of the game in order to determine which move to play. It consists of 4 steps which are repeated until no time is left (Chaslot *et al.*, 2008). The first step is *selection*, in which a node is selected that is not part of the MCTS tree yet. The node is then added to the tree in the *expansion* step. The third step is *simulation* (also called *play-out*), in which a game is simulated from that node until the end by playing random moves. The last step is *backpropagation*, where the result of the simulated game is backpropagated up to the root of the MCTS tree. MCTS is discussed in detail in Chapter 2.

Teytaud and Teytaud (2010a) introduced MCTS based on Upper Confidence Bounds in Havannah. Experiments show that the number of simulations per move has a significant impact on the performance. A player doing $2n$ simulations wins about 75% of the games played against a player doing only n simulations. Additions such as Progressive Widening (Chaslot *et al.*, 2008) and Rapid Action Value Estimates (RAVE) (Gelly and Silver, 2007) are used as well. The former gives a little improvement in the success rate. The addition of RAVE though shows a significant increase in the percentage of games won. A further improvement of RAVE applied to Havannah is discussed by Rimmel, Teytaud, and Teytaud (2011).

Another improvement is given by Teytaud and Teytaud (2010b), where decisive moves are applied during the search. Whenever there is a winning move, that move is played regardless of the other possible moves. Experiments show win percentages in the range of 80% to almost 100%.

Several more additions for an MCTS player in Havannah are discussed by Lorentz (2011). One is to try to find moves near stones already on the board, thus avoiding playing in empty areas. Another method

is the use of the Killer RAVE heuristic, where only the most important moves are used for computing RAVE values. Each of these additions results in a significant increase in the percentage of wins.

Experiments with a Nearest Neighbour preference are discussed by Joosten (2009). This preference ranks moves according to their proximity to the last move played during the simulation step. However, no significant improvement has been achieved with this preference.

Fossel (2010) used Progressive History and Extended RAVE to improve the selection strategies, giving a win percentage of approximately 60%. Experiments with RAVE during the simulation step, or biasing the simulation towards corner cells or cells in proximity of the previous move, do not give significant improvements.

Related games in which MCTS has been applied are for instance Hex and Go. Both are games for which no straightforward evaluation function is known. Like in Havannah, the goal in Hex is to form a connection. Van Rijswijk (2006) and Arneson, Hayward, and Henderson (2010) applied local patterns in Hex to detect captured regions and dead cells, i.e. cells that are not of any use to either player for the rest of the game. This gives a significant improvement in the percentage of wins.

Chaslot (2010) discussed how MCTS can be improved by incorporating knowledge into the *selection* and *simulation* strategy, using Go as the test domain. Experiments show that progressive strategies improve the *selection* strategy, regardless of the board size. Urgency-based and sequence-like simulations improve the simulation strategy significantly.

1.4 Problem Statement and Research Questions

The goal of this thesis is to improve the performance of MCTS for the game Havannah, by incorporating knowledge into the algorithm. Because the goal of the game is to form a particular pattern, knowledge about how to construct or recognize parts of such patterns is of great importance. In the context of MCTS, such knowledge, when used in the *selection* and *simulation* steps, allows for a more directed search towards reasonable moves, which in turn means that less time is lost on exploring weak moves. It is expected that this will lead to an improvement of the performance of the AI player. The problem statement of this thesis is therefore:

- *How can one use knowledge in an MCTS engine for Havannah, in order to improve its performance?*

The corresponding three research questions are as follows:

1. *How can local patterns be matched in an efficient way?*

   The first research question focuses on how the matching of local patterns should work and how it can be implemented as efficiently as possible. The performance of any MCTS engine is dependent on the number of computations needed for each simulation. The fewer computations are required, the more simulations can be done within a given time frame, which in turn means that the moves are chosen based on more reliable data. Therefore it is crucial that the matching of local patterns is computationally inexpensive.

2. *How can pattern knowledge be used in the selection step to improve the balance between exploration and exploitation?*

   Knowledge about how to create patterns can be used to tune the balance between exploration and exploitation during the *selection* step of the MCTS algorithm. A better balance increases the performance of the MCTS algorithm. It is expected that by using such knowledge, the search can be guided in such a way that less time is squandered by trying moves that are unlikely to be useful.

3. *How can pattern knowledge be used in the simulation step to improve the play-out strategies?*

   Similarly to the previous question, using knowledge about patterns in the simulation step results in a more directed simulation. Again, the emphasis is on reasonable moves, rather than purely random ones, which is expected to result in an improvement in performance.

The MCTS engine used for this thesis is based on the work of Joosten (2009). The engine is already enhanced with RAVE, Progressive History, Extended RAVE and Havannah-Mate (Fossel, 2010).

1.5 Thesis Overview

Chapter 2 explains the general MCTS algorithm and several of its enhancements. First, an outline of the MCTS algorithm is given. The chapter then continues with describing several general and domain-specific enhancements to the MCTS algorithm. These enhancements are RAVE, Progressive History, Havannah-Mate, Last-Good-Reply and N-grams.

Chapter 3 describes how pattern knowledge is used in the MCTS engine. First, the chapter discusses patterns in Havannah and how they can be formed in an efficient way. The chapter continues by describing how local patterns are represented and how they are used to guide the search. Furthermore, the chapter discusses several ways by which the quality of a potential move can be determined based on the type of move and its connection properties.

Chapter 4 discusses the conducted experiments and their results. First, experiments with local patterns are discussed. The chapter continues with experiments with LGR, N-grams and a combination of the two. Finally, several experiments are discussed where the search is guided according to the type of move and its connection properties.

Chapter 5 answers the research questions and problem statement, based on the results of the experiments. Finally, several recommendations for future research are mentioned.

Chapter 2

Monte-Carlo Tree Search

This chapter explains the Monte-Carlo Tree Search algorithm. First an overview is given of each step. The chapter then continues with discussing several general and domain-specific enhancements to the MCTS engine. These enhancements are RAVE, Progressive History, Havannah-Mate, Last-Good-Reply and N-grams.

Chapter contents: Monte-Carlo Tree Search — The MCTS Algorithm, Enhancements to MCTS

2.1 The MCTS Algorithm

Monte-Carlo Tree Search (MCTS) is a best-first search algorithm which combines Monte-Carlo evaluation (MCE) with tree search (Coulom, 2007; Kocsis and Szepesvári, 2006). The basic idea behind MCEs is to evaluate a particular game state S according to the result of n simulations. In each of these simulations, moves are played randomly starting in S until the end of the game is reached. A value is assigned to the result of the game, such as $+1$ for win, -1 for loss and 0 for draw. The average of these values is the evaluation of S.

While MCEs have proven to give good results in various games, a major drawback is the computation time needed to obtain a reliable evaluation of a certain game state (Chaslot, 2010). Usually one needs thousands of simulations before an evaluation of S is of any significance. MCTS tries to overcome this drawback, by guiding the search in such a way that the amount of time lost by trying poor moves is minimized.

The MCTS algorithm works by iterating over four steps, until time runs out. These steps are depicted in Figure 2.1 (Chaslot, 2010).

The algorithm slowly builds a tree, where the root node represents the current game state. Each node not only holds information about the game state it represents, but also keeps track of the number of times it has been visited and the number of times a simulation through this node led to a victory or loss. The first step is *selection* in which a node is selected which is not part of the MCTS tree yet. This node is then added in the *expansion* step. From there, a *simulation* is performed by playing random moves. The result of the simulated game is then *backpropagated* to the root. If there is still time left, another iteration of these steps is executed. The four steps are explained in more detail in the subsequent subsections.

2.1.1 Selection

The selection step consists of applying a *selection strategy* in a recursive fashion, starting from the root node of the MCTS tree, until a node is reached which is not part of the MCTS tree yet. Each time the children of a node are examined to determine which child node is the best candidate to be selected next. The selection strategy should provide a proper balance between exploitation and exploration.

Exploitation means that nodes with a high value are preferred over nodes with a low value. When there is too much exploitation, the search tends to be too narrow, i.e. it only searches through a relatively

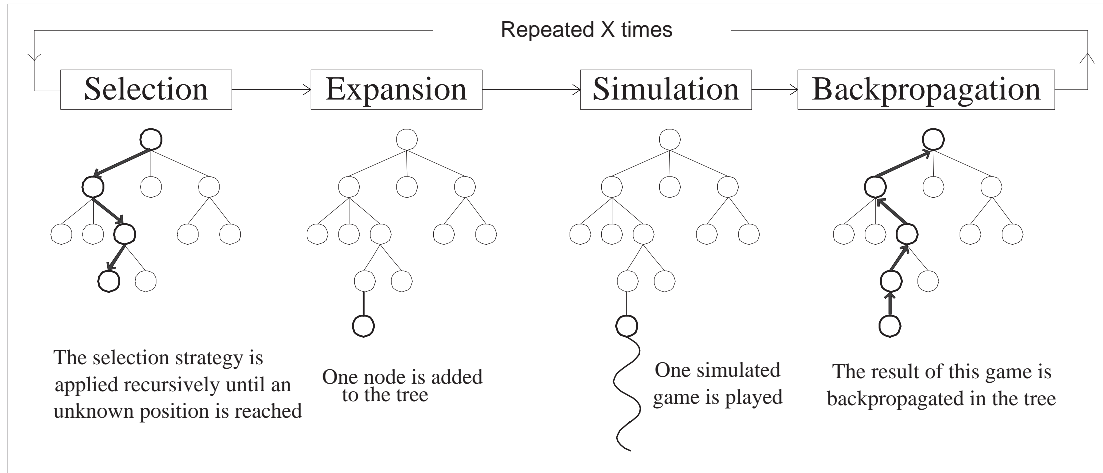

Figure 2.1: The four steps of the Monte-Carlo Tree Search algorithm.

small part of the tree. As a result, the moves which seem good during the search are probably suboptimal. However, moves that seem poor in the early stages of the search, may actually be good moves, but simply appear to be poor because they have not been tried often enough. Those moves need more exploration. Rather than selecting moves based on their value, the idea of exploration is to bias the search towards nodes which have not been visited often enough yet, in order to get a better estimate of their true value. For example, a move which has been tried a hundred times needs less investigation than a move which has been tried only a couple of times. However, too much exploration is not good either, because the MCTS tree gets too wide and valuable search time is lost on exploring moves which are unlikely to be good anyway.

The most commonly used formula to calculate which child node k of node p should be selected, is the UCT (Upper Confidence bounds applied to Trees) formula proposed by Kocsis and Szepesvári (2006), shown in Equation 2.1.

$$k \in argmax_{i \in I} \left(v_i + C \cdot \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{2.1}$$

In the formula, I is the set of all children of node p. The variable v_i is the average reward of node i so far. The variables n_p and n_i denote the number of visits to node p and child node i respectively. The constant C determines the balance between exploitation and exploration. A low value for C will shift the focus towards exploitation as the selection of a node will depend mainly on its value, whereas a high value will do the opposite. The value of C has to be determined experimentally.

2.1.2 Expansion

After the last node has been selected, it is added to the MCTS tree. Although there exist other ways of expanding the tree, such as adding multiple nodes, usually it is sufficient to only add the last selected node (Coulom, 2007). Other expansion methods require more memory, while generally giving only small improvements in the performance.

2.1.3 Simulation

After the tree has been expanded with a new node, a random simulation is performed from that new node until the end of the game. This simulation is also known as the *play-out*.

The moves in the simulation can be chosen completely randomly or pseudo-randomly using knowledge about the game. Choosing moves randomly allows a faster simulation and therefore more iterations of the MCTS algorithm can be done within a given time limit. However, purely random moves are not a good representation of how a good player would play the game. By using knowledge during the simulation, more accurate estimates of the true value of a move can be obtained. This of course requires additional

computation time, which means that fewer iterations of the algorithm can be executed within the same time limit. It is therefore important to find a good balance between speed and accuracy.

Similarly to the *selection* step, there is also a trade-off between exploration and exploitation. If the simulation is too random, it means that the search is too exploratory and the results of the simulation are unreliable because of poor moves. On the other hand, too much exploitation will result in a narrow search tree, which means that the estimates of the true value of a move will be inaccurate.

2.1.4 Backpropagation

The last step of the algorithm propagates the result of the simulated game back to the root. The backpropagation consists of updating the value of every node on the path from the node by which the MCTS tree got expanded up to the root of the tree. Although the value of a node can be computed in various ways, the most common one is to take the average of all the results of simulated games made through that node and dividing it by the number of times the node has been visited. Other methods generally do not improve the performance (Chaslot, 2010). One notable exception is MCTS-Solver (Winands, Björnsson, and Saito, 2008).

The idea of MCTS-Solver is to not only backpropagate using average results, but also to propagate game-theoretical values ∞ and $-\infty$ for won and lost games, respectively. If the value of one of the children of node n is ∞, then there is no need to further inspect node n, as it is always possible to make a win move from node n. Node n is therefore proven to be a win node. Similarly, if all children of n have the value $-\infty$, it means that there is no win move possible from n. Node n is therefore proven to be a loss node. Winands *et al.* (2008) showed that MCTS-Solver gives a 65% win percentage in the game Lines of Action when playing against an MCTS player which only uses average results during the backpropagation. MCTS-Solver has also been used in Hex (Arneson *et al.*, 2010), applied to Connect Four and to solve Seki in the game of Go (Cazenave and Saffidine, 2011).

2.2 Enhancements to MCTS

This section discusses several enhancements which improve the performance of MCTS in Havannah. They are extensions of the general selection and simulation strategies discussed in the previous section. These enhancements have already been implemented by Joosten (2009) and Fossel (2010) in the Havannah engine used in this thesis.

2.2.1 Rapid Action-Value Estimation

Rapid Action-Value Estimation (RAVE) (Gelly and Silver, 2007) combines the All-Moves-As-First heuristic (AMAF) (Brügmann, 1993) with UCT. The idea of AMAF is to treat each move in the simulation to be as important as the first move. For each child i of node n, a value r_i is computed. This value is the average result of those simulations starting from n in which move i has been played. Thus, move i does not need to be played directly in state n, but may also be played at a later stage.

The resulting formula, which is an adapted version of Equation 2.1, is shown in Equation 2.2 (Teytaud and Teytaud, 2010a).

$$k \in argmax_{i \in I} \left(\alpha_i^{'} \cdot v_i + \alpha_i \cdot r_i + C \cdot \sqrt{\frac{\ln n_p}{n_i}} \right) \tag{2.2}$$

In the above mentioned formula $\alpha_i = R/(R + n_i)$ and $\alpha_i^{'} = 1 - \alpha_i$. The constant R controls for how long AMAF influences the selection of a node. A low value indicates that the influence of AMAF quickly decreases the more often node i is visited, while a high value indicates that AMAF still has influence even if node i has been visited a significant number of times.

Teytaud and Teytaud (2010a) showed that RAVE improves the performance of a Havannah-playing MCTS agent significantly, with win percentages of 60% up to 100% depending on the board size.

2.2.2 Progressive History

Progressive History (Nijssen and Winands, 2011) is a combination of Progressive Bias (Chaslot *et al.*, 2008) and the relative History Heuristic (Schaeffer, 1989; Winands *et al.*, 2006). Its aim is to direct the search when there is not sufficient information available to estimate the true value of a node. The formula used to calculate which node should be selected is shown in Equation 2.3a and 2.3b (Fossel, 2010).

$$k \in argmax_{i \in I} \left(v_i + C \cdot \sqrt{\frac{\ln n_p}{n_i}} + h_i \right) \tag{2.3a}$$

$$h_i = \frac{w_i}{n_i} \cdot \frac{W}{l_i + 1} \tag{2.3b}$$

The formula in Equation 2.3a is the same as the formula in Equation 2.1, with the exception of the additional Progressive History variable h_i. The value of this variable depends on two terms. The first term w_i/n_i is the relative History Heuristic and the second term $W/(l_i + 1)$ is the Progressive Bias. The variable w_i is the number of times move i eventually led to a victory, n_i is the number of times move i has been played and l_i denotes the number of times playing move i eventually resulted in a loss. The constant W determines the influence of the relative History Heuristic.

Fossel (2010) showed that for Havannah, Progressive History on its own gives a win percentage of 61% when playing against an MCTS agent which uses only RAVE.

2.2.3 Havannah-Mate

Havannah-Mate (Lorentz, 2011) is an enhancement of the simulation strategy, which checks whether the game can be finished within the next two plies. If this is the case, the corresponding move is always played. The check is performed in the following way. After a random move is chosen, all the free cells adjacent to the connection to which that move belongs, are tried to determine whether a ring, bridge or fork could be completed by playing in these cells. If at least two of these cells complete a winning connection, the game can be ended in the next two turns.

Havannah-Mate can be computationally expensive, as all cells adjacent to existing connections need to be checked. Therefore Havannah-Mate is only applied in the first N/c plies, where N is the total number of cells on the board and c is a constant set experimentally (Fossel, 2010). Lorentz (2011) and Fossel (2010) showed that Havannah-Mate gives a significant improvement in the win percentage.

2.2.4 Last-Good-Reply

Last-Good-Reply (LGR) (Drake, 2009; Baier and Drake, 2010) is an enhancement used during the simulation step of the MCTS algorithm. Rather than applying the default simulation strategy, moves are chosen according to the last good replies to previous moves, based on the results from previous simulations.

It is often the case in Havannah that certain situations are played out locally. This means that if a certain move is a good reply to another move on some board configuration, it will likely be a good reply to that move on different board configurations as well, because it is only the local situation that matters. However, MCTS itself does not 'see' such similar local situations. The goal of LGR is to improve the way in which MCTS handles such local moves and replies.

There are several variants of LGR. The first one is LGR-1, where each of the winner's moves made during the simulation step is stored as the last good reply to the previous move. During the simulation step of future iterations of the MCTS algorithm, the last good reply to the previous move is always played instead of a random move whenever possible. If the last good reply to a certain move is illegal in the current board configuration or if there is no last good reply known to the previous move, the default simulation strategy is used. As an example, consider Figure 2.2, where a sequence of moves made during a simulation is shown.

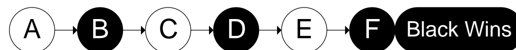

Figure 2.2: A simulation in MCTS.

Because Black is the winner, the LGR-1 table for Black is updated by storing every move by Black as the last good reply to the previous one. For instance, move B is stored as the last good reply to move A. If White will play move A in future simulations, Black will always reply by playing B if possible.

The second variant of LGR is LGR-2. As the name suggests, LGR-2 stores the last good reply to the previous two moves. The advantage of LGR-2 is that the last good replies are based on more relevant samples (Baier and Drake, 2010). During the simulation, the last good reply to the previous two moves is always played whenever possible. If there is no last good reply known for the previous two moves, LGR-1 is tried instead. Therefore, LGR-2 also stores tables for LGR-1 replies.

A third variant is LGR-1 with forgetting, or simply LGRF-1. This works exactly the same like LGR-1, but now the loser's last good replies are deleted if they were played during the simulation. Consider Figure 2.2 again, where White lost the game. For instance, if move C was stored as the last good reply to B for White, it is deleted. Thus, the next time Black plays B, White will chose a move according to the default simulation strategy.

The fourth and last variant is LGRF-2, which is LGR-2 with forgetting. Thus, the last good reply to the previous two moves is stored and after each simulation, the last good replies of the losing player are deleted if they have been played. Experiments with LGR in Havannah are discussed in Chapter 4.

2.2.5 N-grams

The concept of N-grams was originally developed by Shannon (1951), where he discusses how one can predict the next word, given the previous $N-1$ words. Typical applications of N-grams are for instance speech recognition and spelling checkers, where the previous words spoken or written down can help determine what the next word should be. However, N-grams can also be used in the context of games, as shown by Laramée (2002). It is an enhancement to the simulation step of the MCTS algorithm. The idea is somewhat similar to LGR. Again, moves are chosen according to their predecessor, but instead of choosing the last successful reply, the move with the highest win percentage so far among all legal moves is chosen. Thus, for each legal move i, the ratio $w_{i,j}/p_{i,j}$ is calculated, where $w_{i,j}$ is the number of times playing move i in reply to move j led to a win and $p_{i,j}$ is the number of times move i was played in reply to move j. For instance, suppose move A has been played by Black and White can now choose between move B and C. If move B after A led to a win in 80% of the games, while move C after A resulted in a win in 20% of the games, then move B will be chosen.

In order to not make the search too deterministic, the moves are chosen in an ϵ-greedy manner (see Section 3.6). With a probability of $1-\epsilon$ an N-gram move is chosen, while in all other cases, a quasi-random move is chosen based on the default simulation strategy. Furthermore, the values $w_{i,j}$ and $p_{i,j}$ in the N-gram tables are multiplied by a decay factor γ after every move played in the game, where $0 \leq \gamma \leq 1$. This ensures that, as the game progresses, new moves will be tried as well, instead of only playing the same N-gram moves over and over again.

It is also possible to combine N-grams with a certain threshold T. The reason to apply thresholding, is to try to improve the reliability of N-grams. The more often a certain N-gram has been played, the more reliable it is. If the N-gram of a proposed move has been played fewer than T times before, the move is not taken into consideration. If all of the available moves have been played fewer than T times, the default simulation strategy is applied.

Like with LGR, N-grams can be extended to take into account the previous two moves, instead of only the previous move. In order to distinguish between the two for the remainder of this thesis, 'N-gram1' refers to N-grams based on only the previous move, while 'N-gram2' refers to N-grams based on the previous two moves.

Because N-gram1 and N-gram2 are based on different contexts, combining the two may give a better performance than using N-gram1 or N-gram2 on its own. N-gram1 and N-gram2 can be combined using averaging. Rather than choosing moves based purely on N-gram2, the moves are chosen based on the average of the ratios $w_{i,j}/p_{i,j}$ of N-gram1 and N-gram2. Experiments with N-grams, thresholding and averaging are discussed in Chapter 4.

Chapter 3

Pattern Knowledge

This chapter describes how pattern knowledge is used within the MCTS engine. First, the role of patterns in Havannah is discussed. The chapter then describes how local patterns are used to guide the search and how they can be represented efficiently. The fourth section discusses the concept of dead cells. The chapter continues with describing two ways how moves are selected, namely using Roulette Wheel Selection or ϵ-Greedy Selection. The chapter ends with discussing how the type of a potential move and its connection properties are used to direct the search.

Chapter contents: Pattern Knowledge — Patterns in Havannah, Local Patterns, Assigning Weights, Dead Cells, Roulette Wheel Selection, ϵ-Greedy Selection, Initializing Visit and Win Counts

3.1 Patterns in Havannah

The goal of Havannah is to create one of three possible patterns: a fork, a bridge or a ring. Freeling (2003) describes how those patterns can be created in an efficient way by means of 'frames'. These are uncompleted connections which cannot be broken by the opponent. An example of a fork frame is shown in Figure 3.1.

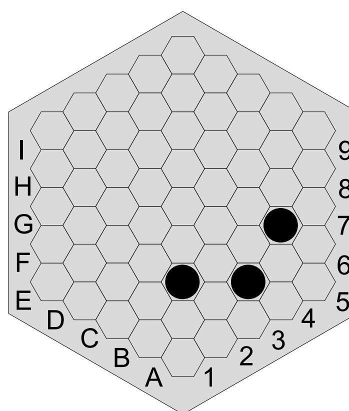

Figure 3.1: An example of a fork frame.

No matter where the opponent plays, Black will always be able to complete the fork. This is possible because of the two unoccupied cells between two black stones. If White would play in one of those two cells, Black simply needs to play in the other one to ensure the connection. A similar story holds for the cells between the black stones and the edges.

Ring frames are a bit more difficult to construct. Figure 3.2a shows what appears to be a frame for a ring.

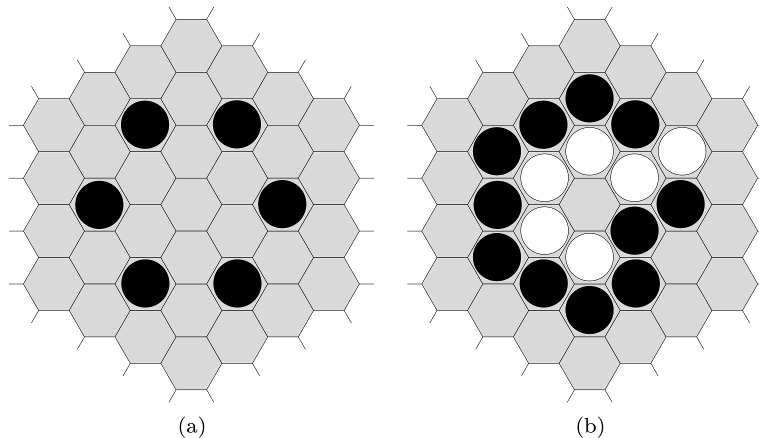

Figure 3.2: Example of what appears to be, but is actually not a ring frame.

The black stones in Figure 3.2a have two unoccupied cells between them. However, White can break this connection from the inside, as shown in Figure 3.2b. If White places stones on the inside, there will eventually be an opportunity for White to form a ring of six cells. At this point, Black will have to prevent this by placing a stone in the cell which would complete the White ring. This will give White a chance to break the black connection. If two of the black stones from Figure 3.2a would have been connected, as shown in Figure 3.3, Black would have had a ring frame.

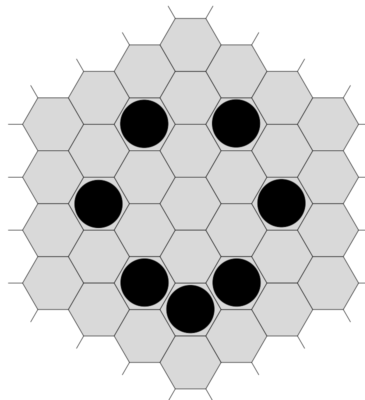

Figure 3.3: An example of a ring frame.

However, determining whether certain stones form a frame can be quite difficult and above all computationally expensive. This means that fewer iterations of the MCTS algorithm can be done, resulting in a decrease in the performance. Therefore, is it necessary to determine the quality of a move in a different way. This chapter discusses several approaches to do this.

3.2 Local Patterns

Chaslot (2010) describes how local 3×3 patterns can be used to guide the search in the game of Go. A similar idea can be used for Havannah. Every time a cell is considered as a move, the six cells surrounding that cell are taken into account as well. These six cells form a certain local pattern. For the remainder of this chapter, the term 'pattern' refers to such a local pattern. The performance of the knowledge-based MCTS algorithm depends on how efficiently those patterns are represented. The most basic way in which a pattern can be represented is by using three 6-bit integers. These integers correspond to the positions

of own stones (i_{own}), opponent stones (i_{opp}) and edge cells (i_{edge}). As an example, let us take the pattern shown in Figure 3.4.

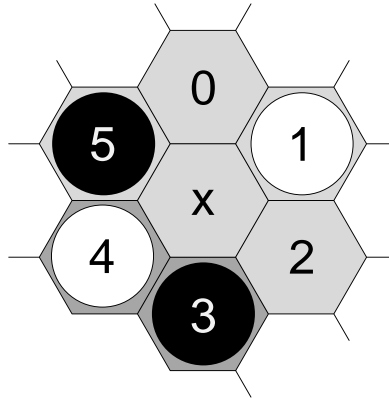

Figure 3.4: An example pattern with indicated cell numbers.

The cell marked with '×' is the move currently under consideration. The surrounding cells are numbered from 0 to 5 and indicate to which bit each cell corresponds. The two darker shaded cells 3 and 4 are cells on the edge of the board.

Suppose White is to move, then the white stones are 'own stones' and the black stones are 'opponent stones'. Cells 1 and 4 are the only cells containing own stones and therefore $i_{own} = 010010 = 18$. Similarly, $i_{opp} = 101000 = 40$ and $i_{edge} = 011000 = 24$.

Each pattern has a number of siblings, i.e. mirrored and rotated versions of the same pattern. By taking siblings into account, the total number of unique possible patterns is 938. Representing patterns as three 6-bit numbers has the advantage that rotation and mirroring can be done quickly. Rotation consists of a circular bit shift, whereby bits are shifted left or right and those bits that are shifted out of the bit string, are reinserted at the other side. For example, a possible rotated version of the pattern from Figure 3.4 can be represented as $i'_{own} = 100100 = 36$, $i'_{opp} = 010001 = 17$ and $i'_{edge} = 110000 = 48$. Mirroring a pattern consists of simply reversing the bit order of the three integers by which it is represented. A mirrored version of the pattern from Figure 3.4 is described by $i''_{own} = 010010 = 18$, $i''_{opp} = 000101 = 5$ and $i''_{edge} = 000110 = 6$.

In order to ensure an efficient lookup for patterns, each pattern is indexed in a 3-dimensional table. The index corresponds to i_{own}, i_{opp} and i_{edge}. For example, the pattern from Figure 3.4 would simply be indexed at $(18, 40, 24)$. Each entry in the table contains information about the weight of the pattern, which is used to direct the search. Section 3.3 discusses how these weights are calculated.

3.3 Assigning Weights

The weight of a pattern should reflect the quality of the corresponding move. Each pattern is therefore weighted by its *transition probability* (Tsuruoka, Yokoyama, and Chikayama, 2002). This probability is calculated according to Equation 3.1.

$$W_p = \frac{n_{played(p)}}{n_{available(p)}} \tag{3.1}$$

W_p denotes the weight of pattern p. The variable $n_{played(p)}$ is the number of turns a move corresponding to pattern p was played, whereas $n_{available(p)}$ denotes the number of turns it was possible to play a move corresponding to pattern p. Thus, the more often it is possible to play a certain pattern and the less it is actually played, the lower the weight of that pattern gets.

The actual values of $n_{played(p)}$ and $n_{available(p)}$ are determined by letting the program play a significant number of games against itself. Each turn, after the best move is chosen using MCTS, the values of $n_{played(p)}$ and $n_{available(p)}$ are updated according to the board configuration in that turn. Updating these values not only means that the table entry for pattern p has to be updated, but also for all of its siblings. After all, the orientation of a pattern does not change its weight. Figure 3.5 shows the distribution of

pattern weights as calculated according to Equation 3.1, based on 5,000 games played on a board with 5 cells per side.

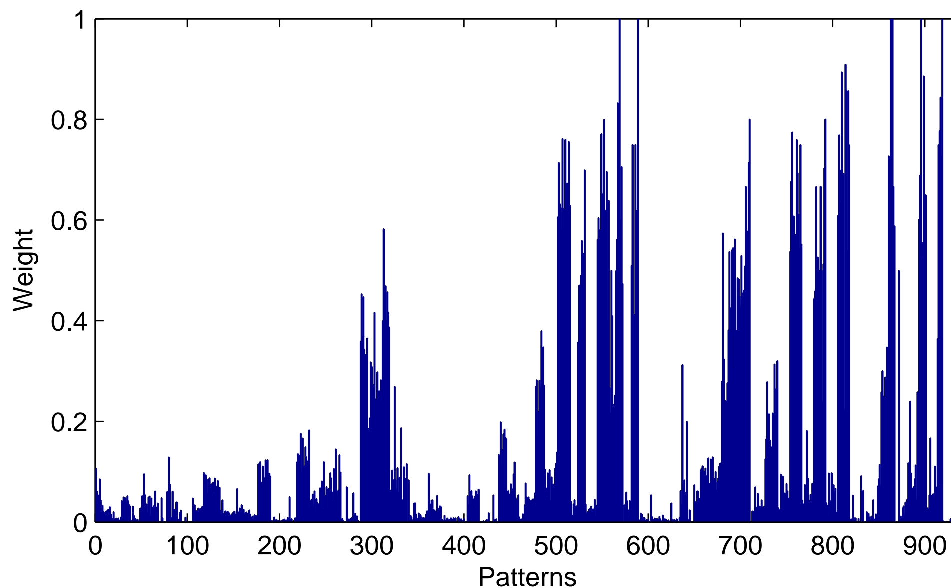

Figure 3.5: Distribution of weights on a base-5 board.

Each bar in Figure 3.5 represents the weight of a unique pattern. As the figure shows, certain patterns seem to be better than others. An example of a pattern corresponding to a good move, is depicted in Figure 3.6.

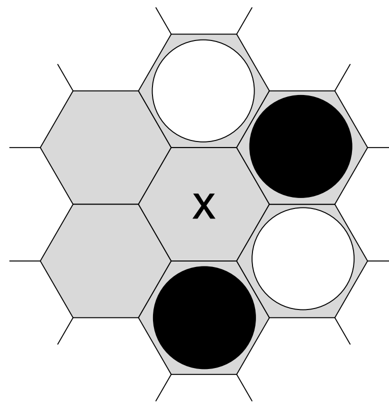

Figure 3.6: An example of a pattern with a high weight.

The pattern corresponds to a good move, because playing in the cell marked '×' connects two stones of the same colour, while at the same time it ensures that the opponent cannot connect his two stones. During the 5,000 games played to determine the weights, this pattern occurred 1154 times and the corresponding move was played 700 times, which gives a relatively high weight of $700/1154 \approx 0.61$.

It could happen that two patterns have a similar weight, even though one of the patterns appeared much more often than the other during the simulations. For example, pattern p_1 could have the weight $50/100 = 0.5$ while pattern p_2 might have the weight $2/4 = 0.5$. Both patterns have the same weight, but p_1 was available 100 times, whereas p_2 was only available 4 times. One could say that therefore the weight of p_1 is more reliable. However, while it seems counterintuitive, the difference between the number of times the patterns were available is not of great importance. If pattern p_2 was only available 4 times during these simulations, it means that the pattern is unlikely to show up anyway.

3.4 Dead Cells

The concept of dead cells was introduced by Van Rijswijk (2006) and Arneson *et al.* (2010) for the game of Hex. A dead cell is a cell which will not be of any use to either player for the rest of the game. Dead cells are important in Hex, as they occur relatively often. In Havannah however, there are only two possible dead cells when using patterns based on the six surrounding cells. These are shown in Figure 3.7.

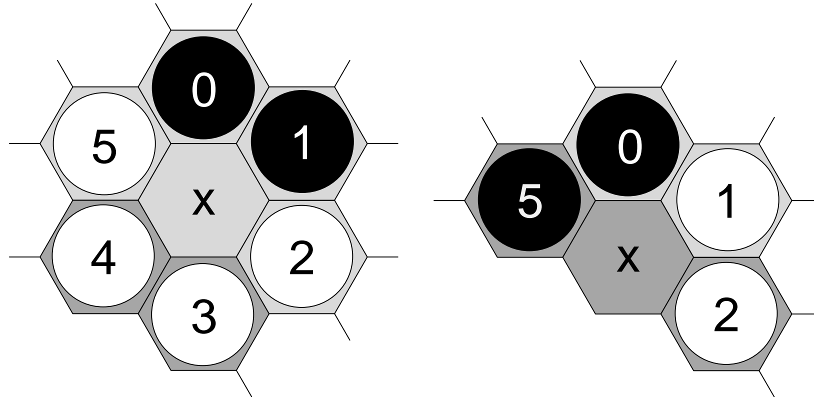

Figure 3.7: Dead cells in Havannah.

Playing in the cell marked '×' is a useless move for both players, as it does not lead to any new connections. These moves are therefore given a weight of 0. Consider the left pattern in Figure 3.7. If White plays in the empty cell, it does not create a new connection between white stones. White is already connected to the side through stones 3 and 4 and these two stones also connect stones 2 and 5. Furthermore, the two black stones prevent the empty cell from being useful for the creation of a white ring. A similar story holds if Black plays in the empty cell. Again, no new black connections are formed and the white cells prevent the empty cell from being useful for the formation of a black ring.

If one or more of the numbered cells would be empty, the cell marked '×' would not be dead anymore, because it would still be possible for one or both of the players to form a new connection through cell '×'. For the same reason, if the colours of the stones in the numbered cells would be different, cell '×' would not be dead either.

The right pattern in Figure 3.7 depicts another dead cell. This is a dead cell for similar reasons as described above. Cell '×' is not of any use for the creation of a new connection between stones of the same colour. However, if for instance cell 1 would be occupied by a black stone, then one could use cell '×' as part of a ring. Similarly, if cell 2 would be occupied by Black, then one could use cell '×' to connect cell 2 with cells 0 and 5.

3.5 Roulette Wheel Selection

After the weights of all available moves have been determined, a move is chosen to be played. One possible way to choose which move to play, is by applying Roulette Wheel Selection (RWS).

RWS takes into account the weights of the patterns. Moves of which the pattern has a higher weight have a higher probability of being chosen. One can compare this to a roulette wheel in a casino. Each pocket on the wheel corresponds to a single move and the size of the pocket corresponds to the weight of a move. As an example, consider Figure 3.8.

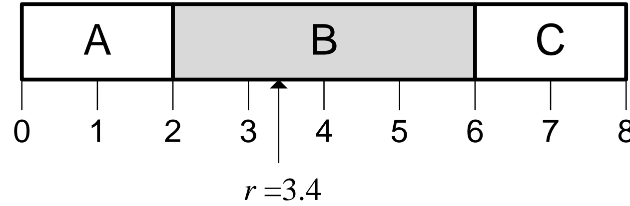

Figure 3.8: Roulette Wheel Selection.

The current player has a choice between move A, B and C with weights 2, 4 and 2, respectively. RWS starts by drawing a random number r from the interval $[0, S]$, where S is the sum of the weights of all available moves. Let us assume that $r = 3.4$. The next step is to subtract the weights of the moves from r one by one, until $r \leq 0$. In this example, subtracting the weight of move A would give $r = 3.4 - 2 = 1.4$, which is still greater than 0. Thus, the subtraction continues with the next move, which is move B. This gives a value of $r = 1.4 - 4 = -2.6$, which is less than zero and therefore the chosen move is B.

3.6 ϵ-Greedy Selection

Another approach by which moves can be chosen, is ϵ-greedy selection (Sutton and Barto, 1998; Sturtevant, 2008), which chooses moves in a slightly more deterministic way than RWS. Rather than choosing a random move with a probability depending on the weight, ϵ-greedy simply chooses the move with the highest weight with a probability of $1 - \epsilon$ and a quasi-random move with a probability ϵ. A common value for ϵ is 0.1, which means that 90% of the time, the move with the highest weight is chosen. It could happen that multiple available moves have highest weight. If this is the case, a random choice is made among those moves.

The ϵ-greedy selection is also applied in combination with N-grams, as discussed in Subsection 2.2.5. The move with the highest win percentage so far is chosen with a probability of $1 - \epsilon$, whereas the default simulation strategy is applied with a probability of ϵ.

3.7 Initializing Visit and Win Counts

Gelly and Silver (2007) proposed the prior knowledge enhancement for the selection step of the MCTS algorithm, where the visit and win counts of new nodes in the MCTS tree are initialized to certain values, based on the properties of the move to which the node corresponds. It is basically an adaptation of the UCT formula, as shown in Equation 3.2.

$$k \in argmax_{i \in I} \left(\frac{v_i n_i + \alpha_i}{n_i + \beta_i} + C \cdot \sqrt{\frac{\ln n_p}{n_i + \beta_i}} \right) \tag{3.2}$$

The additional parameters α_i and β_i are the win count bonus and visit count bonus, respectively, which are based on the properties of the move corresponding to node i. By adding such bonus to the win and visit count of MCTS nodes, the selection can be biased towards certain moves.

This section discusses three methods how the values of α_i and β_i can be chosen. First, Subsection 3.7.1 describes how the selection can be biased towards joint moves and neighbour moves. Subsection 3.7.2 then discusses how local connections can be used to guide the selection. Finally, Subsection 3.7.3 describes how the selection is biased towards edge and corner connections.

3.7.1 Joint and Neighbour Moves

Lorentz (2011) proposed to initialize the visit and win counts of new nodes in such a way, that the selection is biased towards 'joint moves' and 'neighbour moves'. Joint moves are moves which are located two cells from a stone of the current player and where the two cells between are empty. An example is shown in Figure 3.9, where the joint move is marked with 'J', viewed from White's perspective. The nodes corresponding to such moves are initialized with a visit count of 40 and a win count of 30. Neighbour

moves are simply moves adjacent to a cell of the same colour, marked with 'N' in Figure 3.9. They get initialized with a visit count of 40 and a win count of 40. All other moves are initialized with a visit count of 40 and a win count of 5. Experiments with a biased selection towards joint and neighbour moves are discussed in Section 4.6.1.

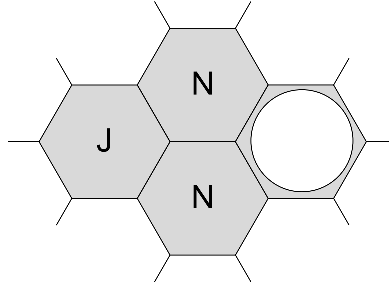

Figure 3.9: A joint move and two neighbour moves.

3.7.2 Local Connections

The idea of initializing visit and win counts of new nodes may also be combined with the quality of the proposed move with respect to connections, rather than the type of move. After all, Havannah is a game in which connections play an important role. For instance, one could argue that it is better to play a move which connects two chains, rather than a move which is not connected to anything.

To determine the connection quality of a move, the number of local groups of the current player's stones surrounding the proposed move is counted. Stones which are connected with each other form one group. As an example, consider Figure 3.10.

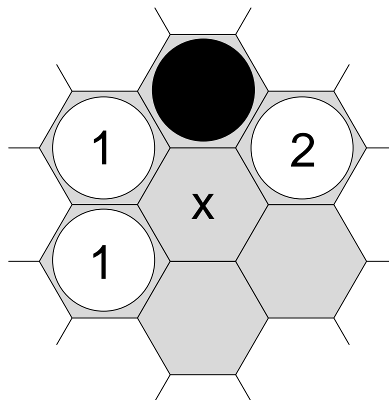

Figure 3.10: Groups of surrounding stones.

Assuming that White is to play, there are two local groups of white surrounding stones in Figure 3.10. They are indicated by the number of the group to which the stone belongs. The stones marked with '1' belong to the same group, as they are connected with each other. The stone marked with '2' is a separate group. If one would play move '×', it would thus form a connection between local two groups. Of course, it could be the case that the two groups are actually connected outside this local area, in which case playing move '×' would simply complete a ring.

Because the maximum number of local groups is three, namely three groups of one stone, the visit and win counts are initialized as follows. Each node gets an initial visit count of 30. Nodes with 0 groups surrounding the corresponding move get an initial win count of 0. Nodes with 1, 2, and 3 local groups surrounding the corresponding move get initial win counts of 10, 20 and 30, respectively.

3.7.3 Edge and Corner Connections

Another option is to count the number of edges and corners a proposed move would indirectly be connected to. The idea is to try to direct the search towards the formation of forks and bridges. For example, consider Figure 3.11.

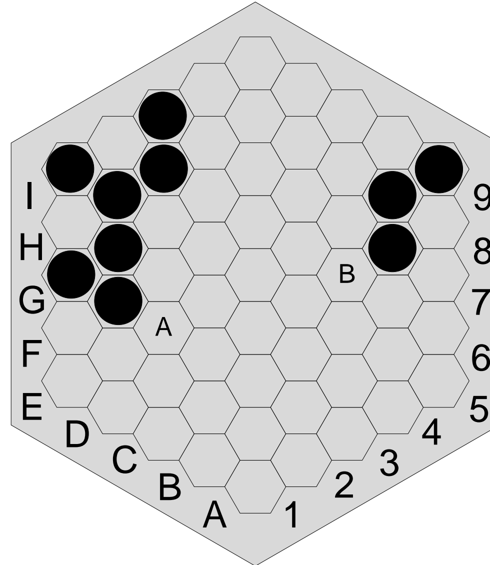

Figure 3.11: Edge and corner connections.

Move 'A' on cell E3 is indirectly connected to one corner and two edges, whereas move 'B' on cell D6 is only connected to one corner. Move 'A' is therefore likely to be better than move 'B'. The MCTS engine already keeps track to which chain each stone belongs. This means that the only additional calculations are to determine which chains the proposed move is connected to and then check each of the edge and corner cells whether they belong to one of those chains as well.

If the number of corners to which a move is connected is greater than 1, or if the number of edges to which is it connected is greater than 2, it means that playing that move would win the game. In this case, the initial visit and win count of the corresponding node is set to 1 and 1000, respectively. This ensures that that node is always selected in future iterations of the MCTS algorithm. In all other cases, the initial visit count is set to 30. The initial win count is equal to the number of edges or corners to which the proposed move is connected, multiplied by 10. Thus, the initial win count can be 30 at most, namely for one corner and two edges.

Chapter 4

Experiments and Discussion

This chapter discusses the experiments carried out using the techniques and enhancements described in the previous chapters. First, the performance when using local patterns in the play-out is tested. The chapter continues with the results of experiments with Last-Good-Reply, N-grams and the influence of skipping dead cells. Finally, experiments visit and win count initialization are discussed.

Chapter contents: Experiments – Experimental Setup, Local Patterns During Play-out, Last-Good-Reply, N-Grams, Dead Cells, Initializing Visit and Win Counts, Summary

4.1 Experimental Setup

The experiments discussed in this chapter were performed on a 2.4GHz AMD Opteron CPU with 8GB RAM. Each of the experiments was done in a self-play setup. Unless stated otherwise, each experiment consists of $1,000$ games on a board with 5 cells per side, with a thinking time of 1 second. Both players use RAVE with parameters $R = 50$ and $C = 0.4$ and Havannah-Mate. Because White has a starting advantage, the experiment is split into two parts. During the first part, the enhancements discussed in the previous chapters are applied to White and during the second part, they are applied to Black. The non-enhanced player simply plays random moves as its play-out strategy. The performance is then calculated as the average percentage of wins of the enhanced players.

4.2 Local Patterns During Play-out

In order to determine the performance of MCTS when using local patterns to guide the play-out, $1,000$ games were played with the setup as described in Section 4.1. For the remainder of this thesis, 'pattern play-out' refers to a play-out which is guided by local patterns. The moves of the pattern play-out were chosen using Roulette Wheel Selection. The win percentages are shown in Table 4.1. The number between brackets indicates the 95% confidence interval for the average result.

	White Pattern Play-out	Black Pattern Play-out	Average
Percentage of wins	37.2%	33.0 %	35.1% (±3.0)

Table 4.1: Pattern play-out win percentages.

As the table shows, pattern play-out by itself does not lead to an improvement of the performance of MCTS. A quick test showed that the additional time needed for pattern play-outs is negligible. A player using pattern play-out performs almost as many simulations within the thinking time as a player which does not use pattern play-out.

A closer examination of the weight distribution gives a possible explanation for the poor performance. Figure 4.1 below shows how often each of those pattern was encountered during the 5,000 games from which the weights were calculated.

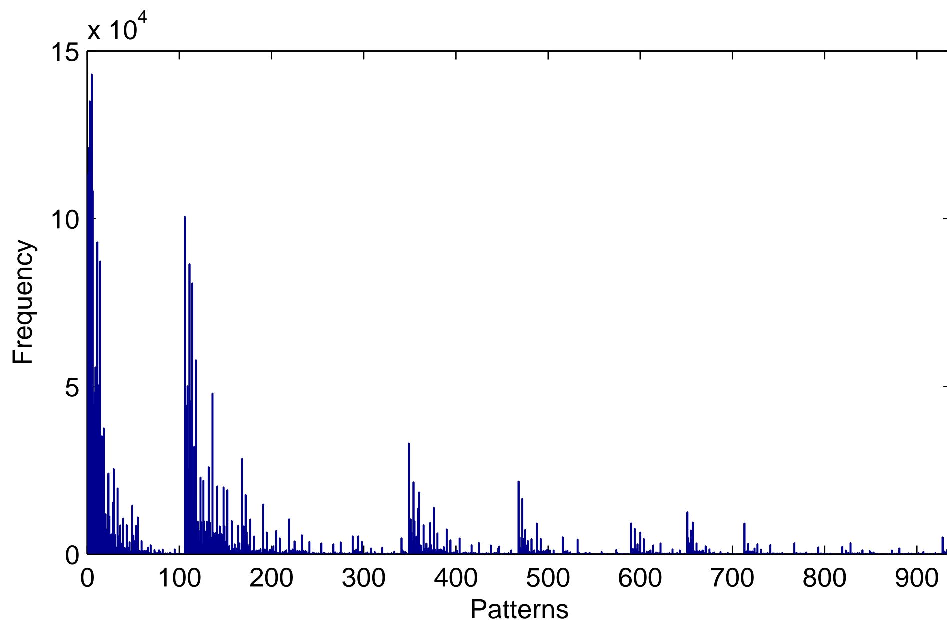

Figure 4.1: Pattern frequencies on a base-5 board.

When comparing Figure 4.1 and Figure 3.5, it seems that most of the patterns with a high weight do not show up often, while those patterns that do appear often seem to have similar weights. In order to verify this, the weight distribution of the 50 most frequent patterns is shown in Figure 4.2.

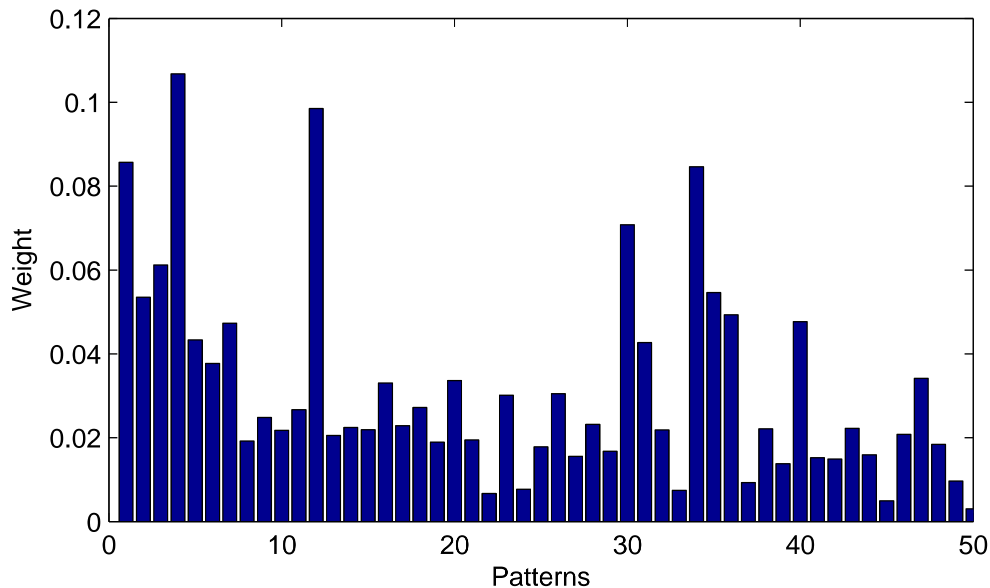

Figure 4.2: Weight distribution of the 50 most frequent patterns.

The patterns in Figure 4.2 are ordered from most frequent to least frequent. It appears that there is some level of distinction between the patterns, although it is not much. There are several peaks, but most patterns seem to have a similar weight.

One could argue that perhaps the six cells that surround a certain cell, are insufficient to determine the quality of the corresponding move, because the pattern is too local. A possible solution would be to not only take into account the neighbours of a cell, but also to take into account the neighbours of those neighbours, i.e. any cells which are within two cells from the proposed move are taken into account. This ensures that the patterns are less local, although it requires more computation time and memory.

4.3 Last-Good-Reply

The performance of Last-Good-Reply was tested with the default setup described in Section 4.1. For this first set of experiments, the contents of the LGR tables of the previous turn were used as the initial LGR tables for the current move. The results are shown in Table 4.2.

	White LGR	Black LGR	Average
LGR-1	65.8%	49.4%	57.6% (±3.1)
LGR-2	62.4%	47.8%	55.1% (±3.1)
LGRF-1	65.8%	58.0%	61.9% (±3.0)
LGRF-2	59.0%	49.6%	54.3% (±3.1)

Table 4.2: Performance of the four LGR variants, with 1 second thinking time.

As the table shows, LGR generally improves the performance of the MCTS engine. LGR-1 and LGR-2 seem to perform equally well given the confidence interval, with win percentages of 57.6% and 55.1%, respectively. Forgetting poor replies seems to give a slight improvement when added to LGR-1, but when added to LGR-2, the performance appears to be the same. LGRF-1 gives a win percentage of 61.9% while LGRF-2 only wins 54.3% of the games.

Quite remarkably, LGR-2 and LGRF-2 do not perform better than LGR-1 and LGRF-1. In particular, the difference between the performance of LGRF-2 and LGRF-1 is quite significant. It appears that taking a larger context into account when using last good replies, does not lead to a better performance of the MCTS engine.

To determine how the four LGR variants perform when using more thinking time per turn, the experiments were repeated with the same parameters, with a thinking time of 3 and 5 seconds. Again, LGR tables of the previous turn were used as the initial LGR tables for the current turn. The results are shown in Table 4.3 and Table 4.4, respectively.

	White LGR	Black LGR	Average
LGR-1	60.6%	47.4%	54.0% (±3.1)
LGR-2	62.2%	43.4%	52.8% (±3.1)
LGRF-1	62.6%	51.2%	56.9% (±3.1)
LGRF-2	61.8%	47.4%	54.6% (±3.1)

Table 4.3: Performance of the four LGR variants, with 3 seconds thinking time.

	White LGR	Black LGR	Average
LGR-1	58.2%	44.0%	51.1% (±3.1)
LGR-2	61.4%	41.8%	51.6% (±3.1)
LGRF-1	64.2%	45.8%	55.0% (±3.1)
LGRF-2	59.8%	44.4%	52.1% (±3.1)

Table 4.4: Performance of the four LGR variants, with 5 seconds thinking time.

The tables show that increasing the thinking time does not improve the performance. In fact, there seems to be a slight decrease in the performance when the thinking time is increased, for each of the LGR variants. Even if the thinking time is increased, forgetting poor replies still seems to make a difference, mainly for the case of LGR-1 and LGRF-1. Both LGR-2 and LGRF-2 keep performing worse than LGR-1 and LGRF-1 when the thinking time is increased.

A possible reason why more thinking time results in a decrease in the performance, may be that the non-enhanced MCTS player, i.e. the opponent of the player using LGR, is simply more influenced by the amount of time allowed for each turn. Increasing the thinking time might improve the performance of the non-enhanced player more than it does improve the performance of the LGR player, which means that when they play against each other, the LGR player wins fewer games.

The experiments were repeated, again with a thinking time of 1, 3 and 5 seconds. This time however, the LGR tables were reset for each turn, to see whether starting with empty tables would have a positive of negative influence on the performance, if any. The results for 1, 3 and 5 seconds thinking time are shown in Table 4.5, Table 4.6 and Table 4.7, respectively.

	White LGR	Black LGR	Average
LGR-1	62.2%	50.0%	56.1% (±3.1)
LGR-2	60.4%	52.8%	56.6% (±3.1)
LGRF-1	62.8%	56.2%	59.5% (±3.0)
LGRF-2	59.8%	51.6%	55.7% (±3.1)

Table 4.5: Performance of the four LGR variants, with 1 second thinking time and LGR reset.

	White LGR	Black LGR	Average
LGR-1	60.4%	49.2%	54.8% (±3.1)
LGR-2	65.2%	46.0%	55.6% (±3.1)
LGRF-1	68.6%	48.6%	58.6% (±3.1)
LGRF-2	60.0%	47.4%	53.7% (±3.1)

Table 4.6: Performance of the four LGR variants, with 3 seconds thinking time and LGR reset.

	White LGR	Black LGR	Average
LGR-1	56.6%	46.8%	51.7% (±3.1)
LGR-2	60.6%	45.0%	52.8% (±3.1)
LGRF-1	63.6%	49.2%	56.4% (±3.1)
LGRF-2	56.8%	45.8%	51.3% (±3.1)

Table 4.7: Performance of the four LGR variants, with 5 seconds thinking time and LGR reset.

When comparing these tables with the previous results, it becomes clear that resetting the tables does not have any influence on the performance of the MCTS player. Each of the results lies within the confidence interval of its no-reset equivalent.

4.4 N-grams

N-grams were tested with the default setup described in Section 4.1. The first experiment was to determine the best value of the decay factor γ. The experiment was done for N-grams based on only the previous move (N-gram1) and N-grams based on the previous two moves (N-gram2). In the case of N-gram2, if no N-gram was available, N-gram1 was used instead. The N-grams were chosen using ϵ-greedy selection, with $\epsilon = 0.1$. The average win percentages are shown in Table 4.8.

γ	N-gram1	N-gram2
1.0	48.9% (±3.1)	52.5% (±3.1)
0.9	48.2% (±3.1)	54.0% (±3.1)
0.8	50.7% (±3.1)	52.5% (±3.1)
0.75	50.8% (±3.1)	57.3% (±3.1)
0.5	50.4% (±3.1)	53.6% (±3.1)
0.25	57.3% (±3.1)	56.7% (±3.1)
0.1	59.4% (±3.0)	56.9% (±3.1)
0.05	57.0% (±3.1)	54.8% (±3.1)
0.0	60.2% (±3.0)	61.3% (±3.0)

Table 4.8: The win percentages of N-grams for different values of γ.

As it turns out, the best value for γ appears to be 0, which simply means that the N-gram tables are completely reset for each turn. The resulting win percentages for N-gram1 and N-gram2 are 60.2% and 61.3%, respectively. This is a rather surprising result, as one would expect that the extra information gathered in the previous turn of the current player would be useful in the current turn. A possible explanation why a lower value for γ gives better results, is that the N-gram tables of the previous turn are based on a search that started two plies higher in the game-tree. This means that the old tables may simply not be accurate enough to be used as the initial tables of the current turn.

Another observation is that N-gram2 seems to perform better than N-gram1, but only for higher values of γ. When the value of γ is decreased, the performance of N-gram2 drops to a similar level as N-gram1 and sometimes it even performs worse.

4.4.1 Thresholding

The second set of experiments with N-grams was done to determine the influence of thresholding. Thus, a move was only considered if its N-gram had been played more than T times before. In the case of N-gram2, if the N-gram was played fewer than T times, the N-gram1 of the move was tried instead. If the N-gram1 was also below the threshold, the move was not considered at all. If none of the N-grams of the available moves had been played more than T times before, a move was chosen using the default simulation strategy. Based on the results from Table 4.8, the experiment was done with a value of $\gamma = 0$. The results are given in Table 4.9.

T	N-gram1	N-gram2
0	60.2% (±3.0)	61.3% (±3.0)
3	57.6% (±3.1)	63.6% (±3.0)
5	55.8% (±3.1)	58.2% (±3.1)
10	49.1% (±3.1)	47.6% (±3.1)

Table 4.9: The influence of applying thresholds to N-grams.

The row for $T = 0$ corresponds to the case where no thresholding is applied. As the table shows, thresholding does not seem to have a positive influence on the performance. In fact, the higher the threshold, the lower the performance seems to get for each of the N-gram variants. A possible explanation why this is the case, is that most N-grams might have never reached the threshold within the 1 second thinking time allowed for each turn and therefore a lot of moves were never considered. When a threshold is applied, it means that for a certain amount of time, only one move will be considered from all available moves, namely the first move of which the N-gram was played more than T times. As long as the N-grams of the other moves do not reach the threshold, that one move will always be chosen. The only way for the other moves to reach the threshold, is if their N-grams are played often enough when the default simulation strategy is applied by ϵ-greedy. The threshold makes the search too deterministic, resulting in a decrease in the performance.

4.4.2 Averaging

The third set of experiments with N-grams were performed to determine the influence of averaging N-gram1 and N-gram2, rather than using only N-gram2. Again, the experiment was run with $\gamma = 0$. The results for different thresholds are shown in Table 4.10.

T	Win Percentage
0	60.7% (±3.0)
3	58.7% (±3.1)
5	54.0% (±3.1)
10	50.7% (±3.1)

Table 4.10: The win percentages of N-gram2 with averaging.

Table 4.10 reveals that using the average of both N-grams instead of only N-gram2 generally does not give a significant improvement when compared to the entries for $\gamma = 0$ in Table 4.8. Again, applying a threshold only decreases the performance. When no threshold is applied, the result is within the confidence interval of the 61.3% mentioned in Table 4.8.

4.4.3 N-grams Combined With LGR

The fourth set of experiments was done to determine the performance when N-grams are combined with Last-Good-Reply. The moves in the play-out are chosen as follows. First, the last good reply is tried. If none exists or if it is illegal, the move is chosen using N-grams with a probability of 0.9, thus $\epsilon = 0.1$. Otherwise, the default simulation strategy is applied. Again, the decay factor was set to $\gamma = 0$. No thresholding or averaging was applied to the N-grams. The results are shown in Table 4.11.

	N-gram1	N-gram2
LGR-1	62.0% (±3.0)	65.0% (±3.0)
LGR-2	61.0% (±3.0)	60.2% (±3.0)
LGRF-1	62.9% (±3.0)	65.6% (±2.9)
LGRF-2	65.9% (±2.9)	62.2% (±3.0)

Table 4.11: The win percentages of LGR combined with N-grams for $\gamma = 0$.

As the table shows, combining LGR with N-grams overall gives a better result. Given the confidence intervals, there seems to be little difference between the performance of the different combinations of LGR and N-grams. When comparing Table 4.11 and Table 4.8, one may conclude that N-grams on their own generally perform just as good as the combination of LGR and N-grams. Thus, adding LGR does not give a significant additional increase in the performance in most cases, although most combinations perform better than LGR on its own. It appears that N-gram1 can best be combined with LGRF-2, while N-gram2 works best with LGR-1 or LGRF-1. These are the only cases which seem to give a significant improvement, when compared to Table 4.8. However, the difference with the performance of the other combinations is marginal.

4.5 Dead Cells

To test the performance of MCTS when applying knowledge about dead cells, an experiment was performed where dead cells were omitted when considering which move to play, both in the selection and simulation steps of the MCTS algorithm. The experiment was executed using the default setup with pattern play-out. The result in given in Table 4.12.

	White	Black	Average
Percentage of wins	38.6%	32.0%	35.3% (±3.0)

Table 4.12: The performance of MCTS when omitting dead cells, with pattern play-out.

When comparing Table 4.12 with Table 4.1, it becomes clear that knowledge about dead cells does not improve the performance significantly when pattern play-out is used. However, to see the influence of dead cells on their own, the experiment was repeated, but with both players using a random playout. The results are shown in Table 4.13

	White	Black	Average
Percentage of wins	52.6%	47.6%	50.1% (±3.1)

Table 4.13: The performance of MCTS when omitting dead cells, with random play-out.

As the table indicates, ignoring dead cells does not lead to an improvement with random play-out either. The two dead cell patterns discussed in Section 3.4 simply do not show up often enough to make any difference. This is because the composition of corresponding patterns simply has a relatively low probability of occurring, when compared to for instance a pattern with only one black or white stone in the neighbouring cells of the proposed move. Therefore, there is little difference in the performance.

4.6 Initializing Visit and Win Counts

To test the performance when visit and win count initialization is used, four sets of experiments were constructed. The first set of experiments was done to determine the performance when the selection is biased towards joint and neighbour moves. The results are discussed in Subsection 4.6.1. The second and third set of experiments were done to determine the performance when the selection is biased towards moves with a high number of local connections and moves with a high number of edge and corner connections. These results are discussed in Subsection 4.6.2 and Subsection 4.6.3, respectively. Finally, a combination of the three methods to initialize visit and win counts was tried. The results are shown in Subsection 4.6.4.

4.6.1 Joint and Neighbour Moves

The influence of biasing the selection towards joint and neighbour moves was tested using the default setup, with the required parameters set to the values described in Subsection 3.7.1. Thus, all new nodes were initialized with a visit count of 40. Joint moves were given a win count of 30, neighbour moves a win count of 40 and all other move a win count of 5. The results are shown in Table 4.14.

	White	Black	Average
Percentage of wins	76.2%	62.2%	69.2% (±2.9)

Table 4.14: Biasing the selection towards joint and neighbour moves.

As the table shows, biasing the selection towards joint and neighbour moves improves the performance greatly, with a win percentage of 69.2%, which is close to the 67.5% that Lorentz (2011) achieved. Additional tweaking of the parameters might even give a higher win percentage.

Because the results were quite encouraging, the experiment was repeated with the addition of LGRF-2 and N-gram1, based on the results from Table 4.11. The results are shown in Table 4.15.

	White	Black	Average
Percentage of wins	83.0%	72.0%	77.5% (±2.6)

Table 4.15: Biasing the selection towards joint and neighbour moves, with the addition of LGRF-2 and N-gram1.

By adding LGRF-2 and N-gram1 to the play-out, the performance increases to 77.5%. This is a significant boost compared to the 69.2% from Table 4.14.

The idea of biasing towards joint moves and neighbour moves was also extended to the play-out. Joint moves were given a weight of 30, neighbour moves a weight of 40 and all other moves a weight of 5. The

moves were then chosen using Roulette Wheel Selection. The result of biasing both the selection and simulation steps towards joint and neighbour moves, is shown in Table 4.16.

	White	Black	Average
Percentage of wins	59.8%	53.0%	56.4% (±3.1)

Table 4.16: Biasing the selection and simulation towards joint and neighbour moves.

It is clear that biasing the play-out towards joint and neighbour moves, decreases the performance. As shown in Table 4.14, directing the selection towards joint and neighbour moves gives a win percentage of 69.2%. However, when joint and neighbour moves are also used in the play-out, the performance drops to 56.4%. The reason for this decrease in performance is most likely the computation cost of determining whether a cell corresponds to a joint or neighbour move.

4.6.2 Local Connections

The experiments with biasing the selection towards local connections, were run using the default setup, with the parameter values described in Subsection 3.7.2. Thus, all nodes were initialized with a visit count of 30. The nodes of which the corresponding move was connected to 0 surrounding groups, were given an initial win count of 0. Nodes of which the corresponding move was connected to 1, 2 or 3 surrounding groups, were given initial win counts of 10, 20 and 30, respectively. The results are shown in Table 4.17.

	White	Black	Average
Percentage of wins	65.6%	57.0%	61.3% (±3.0)

Table 4.17: Biasing the selection towards local connections.

The table indicates that when directing the selection towards local connections, a win percentage of 61.3% is achieved. Compared to Table 4.14, these results are less than when biasing towards joint and neighbour moves, although additional tweaking of the initial visit and win counts could give a boost in the performance. The values of the parameters described in Section 3.7.2 were chosen intuitively.

Like with the tests from Section 4.6.1, these experiments were repeated with the addition of LGRF-2 and N-gram1. The results are shown in Table 4.18.

	White	Black	Average
Percentage of wins	75.8%	64.2%	70.0% (±2.8)

Table 4.18: Biasing the selection towards local connections, with the addition of LGRF-2 and N-gram1.

Again, adding LGRF-2 and N-gram1 to the play-out results in a significant increase compared to the result in Table 4.17. The win percentage increases from 61.3% to 70.0%. However, biasing the selection towards joint and neighbour moves still performs significantly better.

4.6.3 Edge and Corner Connections

A similar set of experiments was done with biasing the selection towards edge and corner connections. For these experiments, the initial visit and win counts were set to those described in Subsection 3.7.3. Thus if a proposed move would be connected to more than 1 corner or more than 2 edges, the initial visit and win counts were set to 1 and 1000, respectively. In all other cases, the initial visit count was set to 30, while the initial win count was set to the number of edges or corners to which the proposed move would be connected, multiplied by 10. The results are shown in Table 4.19.

	White	Black	Average
Percentage of wins	67.2%	50.2%	58.7% (±3.1)

Table 4.19: Biasing the selection towards edge and corner connections.

As the table shows, biasing the selection towards edge and corner connections does increase the win percentage, although not as much as biasing towards joint and neighbour moves or local connections. The difference between the performance of biasing towards local connections and biasing towards edge and corners cells, is quite small however. The experiments were also repeated with the addition of LGRF-2 and N-gram1. These results are shown in Table 4.20.

	White	Black	Average
Percentage of wins	74.4%	62.2%	68.3% (±2.9)

Table 4.20: Biasing the selection towards edge and corner connections, with the addition of LGRF-2 and N-gram1.

As with the previous two methods, adding LGRF-2 and N-gram1 increases the performance significantly. Again, additional parameter tweaking could give an additional increase. The parameter values used during these experiments assume that corners are equally important as sides. Thus, a move which connects to an edge is considered to be just as good as a move which connects to a corner. However, preferring corners over edges or vice versa might lead to different results. Furthermore, the values of the initial win counts are set in a linear fashion. That is, for every edge or corner, a fixed amount of 10 is added. Setting the values in a non-linear way might give different results as well.

In the experiments of Table 4.19 and Table 4.20, the cell of the proposed move itself was not taken into consideration as to whether it was located on an edge or a corner. Thus, a proposed move on for instance an edge, would not be counted as being connected to that edge. To see whether it would make a difference if the cells of the moves themselves were also taken into consideration, the experiments from Table 4.19 and Table 4.20 were repeated. The results are shown in Table 4.21.

	White	Black	Average
Plain	69.2%	53.8%	61.5% (±3.0)
With LGRF-2 and N-gram1	68.6%	53.2%	60.9% (±3.0)

Table 4.21: Biasing towards edge and corner connections, when also taking into account the location of the proposed move itself.

When the location of the proposed move is also taken into consideration, the win percentage achieved is 61.5%, which is just within the confidence interval of the 58.7% mentioned in Table 4.19. However, when LGRF-2 and N-gram1 are added, the performance only reaches 60.9%, which is much worse than the 68.3% from Table 4.20, which is quite remarkable. A possible explanation why adding LGRF-2 and N-gram1 makes no difference, is that the selection is now also biased towards moves which are located on the edges and corners themselves, because they are initialized with a higher win count than moves in the middle of the board. Especially during the early stages of the game, when there are no chains formed yet, the selection chooses moves on edges or corners. It could be that LGRF-2 and N-gram1 have too little influence to guide the search in a different direction, i.e. to let the selection step also choose moves which are not on the edge or corner.

Based on the results from Table 4.21, one may therefore conclude that it is better not to take into account the location of the proposed move itself.

4.6.4 Combination

The last set of experiments with visit and win count initialization was based on a combination of the three methods described in the previous subsections. Thus, the selection was biased towards joint and neighbour moves, local connections and corner and edge connections. The three methods were combined in a cumulative way. The initial visit count for each node was set to 100, which is the combined initial visit count of the three methods. The initial win count was determined by combining the relevant initial visit counts of the three methods. For instance, a node of which the move was a neighbour move, connected locally to 2 groups and 2 edges, was given an initial win count of $40 + 20 + 20 = 80$. Nodes of which the move would be connected to more than 1 corner or 2 edges, were again given an initial visit count of 1 and a win count of 1000. The win percentage of this combination is shown in Table 4.22.

	White	Black	Average
Percentage of wins	79.8%	67.0%	73.4% (±2.7)

Table 4.22: The performance of MCTS when the three visit and win count initialization methods are combined.

As the table indicates, combining the three methods gives good results. The combination works better than any of the three methods on their own. Like with each of the three methods, the combination was also tested with the addition of LGRF-2 and N-gram1. The results are shown in Table 4.23.

	White	Black	Average
Percentage of wins	82.8%	72.2%	77.5% (±2.6)

Table 4.23: The performance of MCTS when the three visit and win count initialization methods are combined, with the addition of LGRF-2 and N-gram1.

The addition of LGRF-2 and N-gram1 gives an increase in the performance, although the increase is not as large as with the three methods individually. In fact, a comparison of Table 4.23 and Table 4.15 shows when LGRF-2 and N-gram1 are added, the combination performs just as good as the first method, where the selection is biased towards joint and neighbour moves, when playing against the non-enhanced MCTS player. However, to determine whether the combination is really just as good as the first method individually, another experiment was performed. During this experiment, one player played using the combination while the other played using a bias towards joint and neighbour moves only. White played using the combination in the first 500 games, while Black used the combination in the last 500 games. Both players used LGRF-2 and N-gram1. The results are shown in Table 4.24.

	White Using Combination	Black Using Combination	Average
Percentage of wins	62.2%	41.0%	51.6% (±3.1)

Table 4.24: The performance the combination of the three methods, when playing against a player where the selection is only biased towards joint and neighbour moves.

The table shows that the win percentage seems to be slightly in favour of the combination, although it is a close call. One may therefore conclude that when LGRF-2 and N-gram1 are added, the combination of the three visit and win count initialization methods indeed performs equally well as biasing the selection only towards joint and neighbour moves.

Because the method is a combination of the previous three, is it the most expensive by far. Nonetheless, the results are quite promising, despite the fact that the three methods were combined in a rather naive way. Again, tweaking the parameter values might give an increase in the performance. This can be quite a challenge however, because there are many possible options to do this. For instance, one might say that the number of edge and corner connections is more important than whether the move is a joint or neighbour move. In that case, the initial visit count and win count values for edge and corner connections should be increased. This ensures that the edge/corner connection property of a move gets a higher 'weight' within the combination of the three methods.

Another option could be to let the importance of each of the three components be dynamic. For example, during the initial stages of the game, there are few chains or no chains at all. It therefore makes sense to bias the selection only towards joint and neighbour moves during the first n turns. This also saves computation time needed to determine the number of local and edge/corner connections, which means that more iterations of the MCTS algorithm can be executed. As the game progresses and more and bigger chains are formed, the importance of local connections and edge/corner connections is then gradually increased.

Finally, another option is of course to tweak the initial visit and win counts of each of the three methods individually, as described in the previous subsections. However, for each of the options mentioned, there is no real guideline for what the actual values of the parameters should be. These would have to be determined by trial and error.

4.7 Summary

The experiments discussed in the previous sections showed mixed results. Using local patterns to guide the play-out only achieved a win percentage of 35.1%. A possible cause for this poor performance could be that the patterns made up of the six surrounding cells are too local. Guiding the play-out by larger patterns may give different results.

Using Last-Good-Reply generally improves the performance. Forgetting poor replies seems to improve the performance, but mainly for the case of LGR-1 and LGRF-1. Using LGR-2 and LGRF-2 instead of LGR-1 and LGRF-1 does not lead to a higher win percentage. When the thinking time is increased, the performance of LGR decreases. Resetting the LGR seems to have no influence on the performance whatsoever.

N-grams generally gave a significant improvement in the performance, depending on the decay factor. N-gram2 performed better than N-gram1, but only for high values of γ. The best performance was achieved with $\gamma = 0$, where both N-gram1 and N-gram2 achieved a win percentage of slightly over 60%. Adding thresholding and averaging did not improve the performance.

The combination of N-grams and LGR gave good results overall. However, in most cases, N-grams on their own perform equally well as N-grams combined with LGR. The only exceptions are N-gram1 with LGRF-2 and N-gram2 with LGR-1 or LGRF-1. These give win percentages of 65.9%, 65.0% and 65.6%, respectively.

Ignoring dead cells during the selection and simulation did not give any improvement. A win percentage of 35.3% was achieved, which is similar to that of pattern play-out.

Visit and win count initialization gave the best results. Biasing the selection towards joint and neighbour moves resulted in a win percentage of 69.2%. With the addition of LGRF-2 and N-gram1, this was increased to 77.5%. Biasing both the selection and simulation towards joint and neighbour moves only resulted in a win percentage of 56.4%, which is most likely caused by the additional computation time during the simulation.

Biasing the selection towards local connections gave a win percentage of 61.3% on its own and 70.0% when LGRF-2 and N-gram1 were added. A bias towards edge and corner connections resulted in a win percentage of 58.7% on its own and 68.3% with the addition of LGRF-2 and N-gram1.

Finally, a combination of the three visit and win count initialization methods was tried. This resulted in a win percentage of 73.4% on its own, which is better than any of the three method individually. However, when LGRF-2 and N-gram1 were added, the performance only increased to 77.5%, which is equally good as biasing only towards joint and neighbour moves, with the addition of LGRF-2 and N-gram1.

Chapter 5

Conclusions

This chapter provides conclusions based on the results of the experiments and gives recommendations for future research. First, each of the three research questions is answered, according to the results of the experiments. The chapter then continues with answering the problem statement of this thesis. Finally, several recommendations for future research are discussed in the last section.

Chapter contents: Conclusions — Answering the Research Questions, Answering the Problem Statement, Future Research

5.1 Answering the Research Questions

This section answers each of the three research questions of this thesis.

1. *How can local patterns be matched in an efficient way?*

   The matching of local patterns should be computationally inexpensive, such that enough iterations of the MCTS algorithm can be executed within a given time frame, making the choice of which move to play more reliable. To achieve this, each local pattern is represented by three 6-bit integers, where the bits correspond to the locations of the player's own stones and the opponent's stones, as well as the locations of edge cells. Using three 6-bit numbers has the additional benefit that siblings of patterns can be found easily, by performing a circular bit shift or reversing the bit string corresponding to each integer. Because each set of three 6-bit integers describes a unique pattern, the weight of the pattern can be looked up efficiently in a 3-dimensional table. A quick test confirmed that the required computation time for local pattern matching is indeed negligible. A player which uses pattern play-out performs almost as many iterations of the MCTS algorithm as a player who does not use pattern play-out, within the same time frame.

2. *How can pattern knowledge be used in the selection step to improve the balance between exploration and exploitation?*

   A good balance between exploration and exploitation is important for the performance of the MCTS algorithm, as a better balance means that less time is wasted on exploring poor moves. The balance between exploration and exploitation can be improved by biasing the selection towards particular moves, according to certain properties of those moves. These properties are based on knowledge about how the patterns in Havannah are constructed. The selection bias is achieved by initializing the visit and win counts of new nodes in the MCTS tree to a certain value. Three approaches to bias the selection, have been discussed in Section 3.7 of this thesis.

   First, the selection can be biased towards joint and neighbour moves. The main idea is that it is generally better to play moves in the vicinity of other stones of the current player. Biasing the selection towards such moves leads to a win percentage of 69.2% on its own and when also adding LGRF-2 and N-gram1 to the play-out, the performance increases to 77.5%. Another option is to

bias the selection towards moves with a high number of local connections. Moves that connect for instance two chains are preferred over moves which are connected to only one chain or are not connected to anything at all. The performance achieved with such a bias, is 61.3%. With the addition of LGRF-2 and N-gram1, the win percentage increases to 70.0%. The third option discussed in this thesis, is to bias the selection based on the number of edges and corners to which a move is indirectly connected. Moves that are connected to multiple edges or corners are preferred over moves which are connected to only one edge or corner. The resulting win percentage achieved with this bias is 58.7%. Adding LGRF-2 and N-gram1 increases the performance to 68.3%.

It is also possible to combine the three properties to bias the selection. For the experiments discussed in Subsection 4.6.4, the three methods have been combined in a cumulative fashion. The win percentage achieved is 73.4%. This is better than the performance of any of the three visit and win count initialization methods individually. However, when LGRF-2 and N-gram1 are added, the performance only increases to 77.5%, which is just as good as biasing the selection only towards joint and neighbour moves, with the addition of LGRF-2 and N-gram1. It should be noted that the actual values of the initial visit and win counts, were determined intuitively. Different values might lead to an increase in the performance. However, these values would have to be determined by trial and error.

3. *How can pattern knowledge be used in the simulation step to improve the play-out strategies?*

   Using local patterns to guide the play-out results in a decrease in the performance. The win percentage achieved is only 35.1%. Additional tests showed that the weight distribution of the 50 most frequent patterns does not give enough distinction between the patterns. A possible reason for the poor performance might be that the patterns based on the six surrounding cells are simply too local. Using larger patterns might give an improvement in the performance, although it requires additional computation time and memory. Ignoring dead cells does not lead to a difference in the performance, because they do not occur often. However, if larger patterns would be used, there might be more possible dead cells.

   Last-Good-Reply and N-grams, which make use of the locality of moves and patterns in Havannah, generally give a significant improvement in the performance. Using LGR-1 results in a win percentage of 57.6%. When forgetting poor replies, the performance is increased to 61.9%. LGR-2 and LGRF-2 do not perform better than LGR-1 and LGRF-1, with a win percentage of 55.1% and 54.3% respectively.

   Both N-gram1 and N-gram2 perform better with a low decay factor γ. In fact, $\gamma = 0$ turns out to be optimal, with a win percentage of 60.2% and 61.3% for N-gram1 and N-gram2, respectively. Using thresholding and averaging does not give an improvement in the performance. When combining N-grams with LGR, a win percentage of more than 60% is achieved for each of the combinations. However, given the performance of N-grams on their own, it can be concluded that adding LGR does give an additional increase in the performance in most cases. The only exception is N-gram1 combined with LGRF-2 and N-gram2 combined with LGR-1 or LGRF-1. These give a win percentage of more than 65%.

   Finally, biasing not only the selection, but also the simulation towards joint and neighbour moves, performs worse than biasing only the selection. This is caused by the computation time required to determine whether a move is a joint move or a neighbour move.

5.2 Answering the Problem Statement

Once the research questions have been answered, the problem statement can be answered as well.

- *How can one use knowledge in an MCTS engine for Havannah, in order to improve its performance?*

The answer to this question can be split up into two parts. First, one can use knowledge about patterns to improve the balance between exploration and exploitation in the selection step of the MCTS algorithm. This is achieved by initializing the visit and win counts of new nodes in the MCTS tree to certain values, which are based on the properties of the move to which the node corresponds. By biasing

the selection towards joint/neighbour moves, local connections and edge/corner connections, a significant improvement in the performance is obtained.

The second part of the answer lies is the improvement of the simulation step. Using local patterns based on the six surrounding cells, does not lead to an improvement of the performance. However, adding LGR and N-grams does lead to an improvement, both when they are used individually and in combination with each other. LGRF-2 and N-gram1 seem to be a strong combination.

The best overall performance is achieved when visit and win count initialization is combined with LGRF-2 and N-gram1. Experiments showed that a win percentage of 77.5% can be achieved.

5.3 Future Research

Although this thesis has propsed several ways by which the performance of an MCTS engine for Havannah can be improved, there are still several ideas that may give an additional improvement.

The first potential improvement is tweaking the parameter values for visit and win count initialization. The values used for the experiments in this thesis, were determined intuitively. Using different initial visit and win counts may give better results. Furthermore, the combination of the three visit and win count initialization methods may be improved. One may consider altering the importance of each of the three methods within the combination. For instance, considering the edge/corner connection property of a move to be more important, might increase the performance of the combination. Another idea is to let the importance of each of the three methods be dynamic with respect to current stage of the game. For example, during the first stages of the game, one could bias the selection only towards joint and neighbour moves, because there are almost no chains yet on the board. As the game progresses and chains are formed, the importance of local and edge/corner connections may be increased while that of joint and neighbour moves is decreased.

Another suggestion is to try using larger local patterns, for instance patterns consisting of all the cells within a distance of two cells from the proposed move. This creates a larger context for the local patterns, which means that a local pattern contains more information about the surrounding area of the proposed move. Furthermore, using larger local patterns may lead to more possible dead cell patterns. The main disadvantages of larger local patterns are the additional computation time and memory requirements.

Furthermore, letting the weights of the patterns depend on the stage of the game, might also give an improvement of the performance. It could be that the moves corresponding to certain patterns are good in the early stages of the game, but become weaker as the game progresses. However, determining in which stage the game is, can be quite difficult. Moreover, one would have to determine first the number of stages into which the game is divided.

References

Allis, L. V. (1988). A Knowledge-based Approach of Connect-Four. M.Sc. thesis, Vrije Universiteit, Amsterdam, The Netherlands. [1]

Arneson, B., Hayward, R. B., and Henderson, P. (2010). Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 251–258. [4, 9, 17]

Baier, H. and Drake, P. (2010). The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 2, No. 4, pp. 303–309. [10, 11]

Bal, H. E. and Romein, J. W. (2003). Solving Awari with Parallel Retrograde Analysis. *IEEE Computer*, Vol. 36, No. 10, pp. 26–33. [1]

Brügmann, B. (1993). Monte Carlo Go. Technical report, Physics Department, Syracuse University, Syracuse, NY, USA. [9]

Cazenave, T. and Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 93–104. Springer-Verlag, Heidelberg, Germany. [9]

Chaslot, G. M. J-B. (2010). *Monte-Carlo Tree Search*. Ph.D. thesis, Maastricht University, Maastricht, The Netherlands. [4, 7, 9, 14]

Chaslot, G. M. J-B., Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J. 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. [3, 10]

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)* (eds. H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers), Vol. 4630 of *LNCS*, pp. 72–83. Springer-Verlag, Heidelberg, Germany. [3, 7, 8]

Drake, P. (2009). The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA*, Vol. 32, No. 4, pp. 221–227. [10]

Fossel, J. D. (2010). Monte-Carlo Tree Search Applied to the Game of Havannah. B.Sc. thesis, Maastricht University, Maastricht, The Netherlands. [4, 9, 10]

Freeling, C. (2003). Introducing Havannah. *Abstract Games*, Vol. 14, pp. 14–20. [2, 13]

Gelly, S. and Silver, D. (2007). Combining Online and Offline Knowledge in UCT. *Proceedings of the 24th International Conference on Machine Learning* (ed. Z. Ghahramani), ICML '07, pp. 273–280, ACM Press, New York, USA. [3, 9, 18]

Herik, H. J. van den, Uiterwijk, J. W. H. M., and Rijswijck, J. van (2002). Games Solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311. [3]

Hsu, F.-h. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion.* Princeton University Press, Princeton, NJ, USA. [1]

Joosten, B. (2009). Creating a Havannah Playing Agent. B.Sc. thesis, Maastricht University, Maastricht, The Netherlands. [3, 4, 9]

Knuth, D. E. and Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [3]

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. *Proceedings of the 2006 European Conference on Machine Learning (ECML-06)*, Vol. 4212 of *LNCS*, pp. 282–293, Springer-Verlag, Heidelberg, Germany. [3, 7, 8]

Laramée, F. D. (2002). Using N-Gram Statistical Models to Predict Player Behavior. *AI Game Programming Wisdom* (ed. S. Rabin), pp. 596–601. Charles River Media, Hingham, MA, USA. [11]

Lorentz, R. J. (2011). Improving Monte-Carlo Tree Search in Havannah. *Computers and Games (CG 2010)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 105–115. Springer-Verlag, Heidelberg, Germany. [3, 10, 18, 27]

Nijssen, J. A. M. and Winands, M. H. M. (2011). Enhancements for Multi-Player Monte-Carlo Tree Search. *Computers and Games (CG 2010)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 238–249. Springer-Verlag, Heidelberg, Germany. [10]

Rijswijk, J. van (2006). *Set Colouring Games*. Ph.D. thesis, University of Alberta, Alberta, Canada. [4, 17]

Rimmel, A., Teytaud, F., and Teytaud, O. (2011). Biasing Monte-Carlo Simulations Through RAVE Values. *Computers and Games (CG 2010)* (eds. H. J. van den Herik, H. Iida, and A. Plaat), Vol. 6515 of *LNCS*, pp. 59–68. Springer-Verlag, Heidelberg, Germany. [3]

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 11, No. 11, pp. 1203–1212. [10]

Schaeffer, J. (2007). Checkers is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [1]

Shannon, C. E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–275. [1]

Shannon, C. E. (1951). Predication and Entropy of Printed English. *The Bell System Technical Journal*, Vol. 30, No. 1, pp. 50–64. [11]

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-player Games. *ICGA Journal*, Vol. 31, No. 1, pp. 195–208. [18]

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA. [18]

Teytaud, F. and Teytaud, O. (2010a). Creating an Upper-Confidence-Tree Program for Havannah. *Advances in Computer Games (ACG 2009)* (eds. H. J. van den Herik and P. Spronck), Vol. 6048 of *LNCS*, pp. 65–74. Springer, Heidelberg, Germany. [2, 3, 9]

Teytaud, F. and Teytaud, O. (2010b). On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG 2010)*, pp. 359–364. [3]

Tsuruoka, Y., Yokoyama, D., and Chikayama, T. (2002). Game-Tree Search Algorithm Based on Realization Probability. *ICGA*, Vol. 25, No. 3, pp. 132–144. [15]

Turing, A. M. (1953). Digital Computers Applied to Games. *Faster Than Thought* (ed. B. V. Bowden), pp. 286–295. Pitman Publishing, London, United Kingdom. [1]

Winands, M. H. M., Werf, E. C. D. van der, Herik, H. J. van den, and Uiterwijk, J. W. H. M. (2006). The Relative History Heuristic. *Computers and Games (CG 2004)* (eds. H.J. van den Herik, Y. Björnsson, and N. S. Netanyahu), Vol. 3846 of *LNCS*, pp. 262–272. Springer-Verlag, Heidelberg, Germany. [10]

Winands, M. H. M., Björnsson, Y., and Saito, J-T. (2008). Monte-Carlo Tree Search Solver. *Computers and Games (CG 2008)* (eds. H. J. van den Herik, X. Xu, Z. Ma, and M. H. M. Winands), Vol. 5131 of *LNCS*, pp. 25–36. Springer-Verlag, Heidelberg, Germany. [9]

Samenvatting

In deze masterscriptie wordt onderzocht hoe de prestatie van een Monte-Carlo Tree Search (MCTS) speler voor het spel Havannah kan worden verbeterd, door gebruik te maken van kennis over patronen gedurende het zoeken. De probleemstelling van deze scriptie luidt daarom als volgt: "*Hoe kan men kennis gebruiken in een MCTS programma voor Havannah zodat the prestatie verbetert?*".

Om deze vraag te beantwoorden, is het onderzoek opgedeeld in twee delen. Het eerste deel is er op gericht om de balans tussen exploratie en exploitatie tijdens de *selectie* stap van het MCTS algoritme te verbeteren. Een betere balans betekent dat minder tijd wordt verspild met het proberen van zetten die waarschijnlijk niet goed zijn. Deze balans wordt verbeterd door de visit en win counts van nieuwe knopen in the MCTS boom te initialiseren aan de hand van kennis over patronen. Door de selectie richting bepaalde zetten te sturen, kan een significante verbetering in de prestatie van MCTS worden bereikt.

Het tweede deel van het onderzoek is er op gericht om de *play-out* van het MCTS algoritme te verbeteren. Het gebruik van locale patronen om de play-out te sturen, leidt niet tot een verbetering. Last-Good-Reply (LGR) geeft over het algemeen wel een verbetering, al hangt het ervan af welke variant gebruikt wordt. N-grams verbeteren de prestatie eveneens. Een combinatie van N-grams en LGR leidt tot een verdere verbetering van de prestatie, alhoewel in de meeste gevallen deze verbetering niet significant hoger is dan wanneer alleen N-grams gebruikt worden.

Experimenten tonen aan dat de beste verbetering wordt bereikt wanneer het initialiseren van visit en win counts wordt gecombineerd met LGR en N-grams. In het beste geval kan een winst percentage van 77.5% worden bereikt tegen het originele Havannah programma.