Analysis and Implementation of the Game OnTop

Robert Briesemeister

Master Thesis DKE 09-25

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Knowledge Engineering
of the Maastricht University

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Artificial Intelligence
November 2009

Preface

This thesis is the result of my last project for the study of Artificial Intelligence. It was performed at the Department of Knowledge Engineering at Maastricht University. The goal was to analyse the game OnTop and to create a challenging computer player. To achieve this goal the search techniques Expectimax and Monte-Carlo were investigated in detail.

I wish to thank several people for supporting me actively and passively during the time of the thesis work. First of all, I would like thank my supervisor Dr. ir. Jos Uiterwijk for his patience during the last months. Especially I would like to say thank you for reading and providing feedback on my thesis and for the productive discussions on new approaches. Furthermore, his lectures on Intelligent Search Technique gave a good introduction and background for this thesis. I also would like to thank Dr. Mark Winands, who was also involved in the course mentioned above, for his explanations of Monte- Carlo Search and all related improvements.

Last but not least, I would like to thank my family and my friends for their continuous support and motivation during the whole studies. A special thank is given to my friend Susanne Kretschmer for her additional reading and commenting on my thesis and Cathleen Heyden and Frank Fleischhack who volunteered to be the human opponents to play against the computer player in the game OnTop.

Robert Briesemeister
Aachen, November 2009

Abstract

In the field of artificial intelligence (AI) solving games and creating challenging computer implementations is of high significance. Probably the most famous AI, Deep Blue, was developed for chess and beat world-class players like Kasparov.

This thesis focuses on the analysis and implementation of AI for the modern board game OnTop. OnTop contains perfect information and chance events and is categorised as a non-deterministic game. Chance events occur when drawing a random tile. Due to the fact that the individual tile types have different numbers, the chance events are not uniformly distributed. OnTop can be played with up to 4 players. This thesis focuses on the 2-player variation.

The thesis starts with an introduction to the game OnTop and follows with the game-tree and state-space complexity. After that the two search techniques Expectimax and Monte-Carlo search are described in detail. In addition, some improvements and enhancements are presented for both techniques. With respect to Expectimax these are Star1, Star2 and Star2.5 of the *-Minimax algorithm family. Concerning Monte-Carlo Search we have investigated Monte-Carlo Tree Search with Upper Confidence bounds applied to Trees and some improved techniques such as parallelisation.

The results show that for the *-Minimax algorithms Star2 performs best. Star2 is able to reduce the number of searched nodes for a certain search depth up to 93% compared to Expectimax. The experiments also revealed that move ordering influences the number of nodes serached. Moreover, it was not possible for Star2.5 to perform better than Star2. For Monte Carlo the investigated improvement Monte-Carlo Tree Search (MCTS) reaches the best results. The strength of MCTS was furthermore improved by root parallelisation. The parallelisation approach increased the number of simulated games at least up to 55%. The games between Star2 and MCTS show that MCTS performs much better. This experiment additionally shows that having the upper hand in the beginning of the game is no guarantee of winning the game in the end. So MCTS wins 80% of the simulated games.

Contents

List of Figures

List of Tables

List of Algorithm Listings

Chapter 1

Introduction

This chapter provides a general introduction to stochastic board games (section 1.1). In section 1.2 the problem statement and the related research questions are listed. Finally, in section 1.3 an outline of this thesis is given.

1.1 Domain of Stochastic Games

The influence of artificial intelligence (AI) is visible in many domains of daily life. The content of this thesis focuses on the domain of games and more specifically on stochastic board games. Games are a good abstraction of domains in which agents follow different goals. In OnTop, which forms the focus of this thesis, every player's goal is to win a game and consequently the player wants the opponent to lose.

There are also several ways to categorise games. Table 1.1 shows different types of games and some examples.

	perfect	**imperfect**
deterministic	chess, Go	stratego
non-deterministic	backgammon, OnTop, Carcassone	poker

Table 1.1: Game categories.

Most research done so far focused on deterministic games with perfect information. In such games every player can see at every time all the information about the game. In this way the player has a fully observable state. The first implementation of a computer player was also done for this type of game. In 1950 Claude Shannon[34] and Alan Turing [8] designed the first chess program. The artificial intelligence created for this program was the basis for the strongest chess computer machine called DEEP BLUE [24]. Two other famous research games in this domain are Checkers [33] and Go [20, 28].

Deterministic games with imperfect information are characterised by a non-observable game state and no elements of chance. For example in the game Stratego the player has no knowledge about the units' position of the opponent until an attack is done.

An example of non-deterministic games with imperfect information is Poker. In Poker the players have no knowledge about the cards of their opponents. Furthermore, the result of a Poker game depends on chance events which occur when extra cards like community cards are added to the game. The Poker game has been also the subject of several research studies [4, 18].

The game OnTop is an example of a non-deterministic game with perfect information. Games in this domain contain an element of chance; examples include throwing a dice or drawing a random card. Some research has already been done in this domain, for example on the games Backgammon [21] and Carcassonne [23].

Due to the great success of research in deterministic games, future research on non-deterministic games is likely to grow in importance.

The most commonly used implementation approaches which can be applied to non-deterministic games are Monte-Carlo Search and Expectimax. An alternative technique is reinforcement learning [15]. The Monte-Carlo Search is based on several randomly played games for a possible move. Ultimately, the winning rate of the simulated games for a move depends on the players' choice. Alternatively, the Expectimax approach evaluates a game state in the future and derives from this evaluation the best move. Both techniques will be applied to OnTop in this thesis.

1.2 Problem Statement and Research Questions

The motivation for artificial intelligence is to create a program which can autonomously solve a given problem. In this case the general problem is to play a game as good as possible. The more specific problem statement for this thesis is the following:

Can we build an effective and efficient computer player for OnTop?

To find an answer to this statement several research questions arise. The thesis will attempt to answer the following five research questions.

1. *What is the complexity of the game OnTop?*

To answer this question the state-space complexity and the game-tree complexity need to be computed.

2. *Can Expectimax be used for the implementation of OnTop?*

This approach is a well-known technique for stochastic games. It was also applied to Carcassonne [23]. For this reason it will be applied to OnTop, too.

3. *Can Monte-Carlo search be used for the implementation of OnTop?*

Monte Carlo is a newer approach for implementing a computer player. It has an easy theoretic background which simplifies the implementation. In the research done so far it led to good results. [14, 25, 40]

4. How can the investigated techniques be improved?

Depending on the results from the research with regards to the most effective implementation, this thesis will attempt to provide suggestions for improvements and enhancements.

5. Which improved technique of Monte-Carlo and Expectimax is most effective?

To get an overview of the quality and the strength of both implementations, they are used to play against each other. The effectiveness and efficiency can be derived from the winning rate of an implementation. Other facts like the reached points per game or used time per turn can also also be a factor in answering this question.

1.3 Thesis Outline

The outline of this thesis is as follows:

- Chapter 1 contains a general introduction to probabilistic games and applicable techniques. Furthermore, this chapter gives an overview of the problem statement and the research questions.
- Chapter 2 gives an explanation of the modern board game OnTop. Additionally the reader is introduced to the rules and the game play of OnTop. Finally, two strategies are described briefly.
- Chapter 3 deals with a complete complexity analysis of OnTop. For this purpose the state-space complexity and the game-tree complexity are investigated. Moreover, the results of this investigation are compared to other board games.
- Chapter 4 describes two possible approaches for the implementation of a computer player for a game of chance, i.e., Expectimax and Monte-Carlo Search. Furthermore, some improvements for these techniques are presented.
- Chapter 5 reports the experimental results of the two implemented approaches. This chapter also compares the effectiveness of the two approaches against each other.
- Chapter 6 gives the conclusions of this thesis. This chapter evaluates the research questions and answers the problem statement. Moreover, suggestions for future research are given.

Chapter 2

The Game OnTop

OnTop is a tile-laying game with a stochastic feature. It is in an abstract way comparable with the game of Carcassonne.

The game has been published by KOSMOS in 2006 and was invented by Günther Burkhardt. The name of the game is derived from the scoring position of a player during a match. The game can be played by two to four players. Furthermore, the possibility of a team play exists. In this case two players will play together and sum up their points. For this thesis the two-player version is considered.

2.1 Gameplay

The game contains 34 diamond-shaped tiles in five variations, 21 point stones for every player and a fixed playing field. The different tile types and their numbers are shown in figure 2.1.

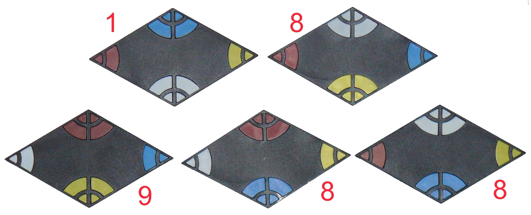


Figure 2.1: Overview of the tiles in OnTop. The numbers indicate how many tiles of this type can appear during the game.

At the beginning of a game two triangle-shaped tiles are placed on fixed positions on the playing field as shown in Figure 2.2. During the game such triangle tiles will also be placed on unplayable positions. The corners of these triangle shaped tiles are not taken into account for evaluation. In every round each player draws a tile and places it on the playing field. A tile can only be placed on an edge of an existing tile (see Figure 2.3). The players can get points for closed circles or for circle parts which are blocked by the playing field. If such a state is reached the circle is evaluated. Only the player with the largest share of the circle gets points. The scoring of a circle is described in the next section.

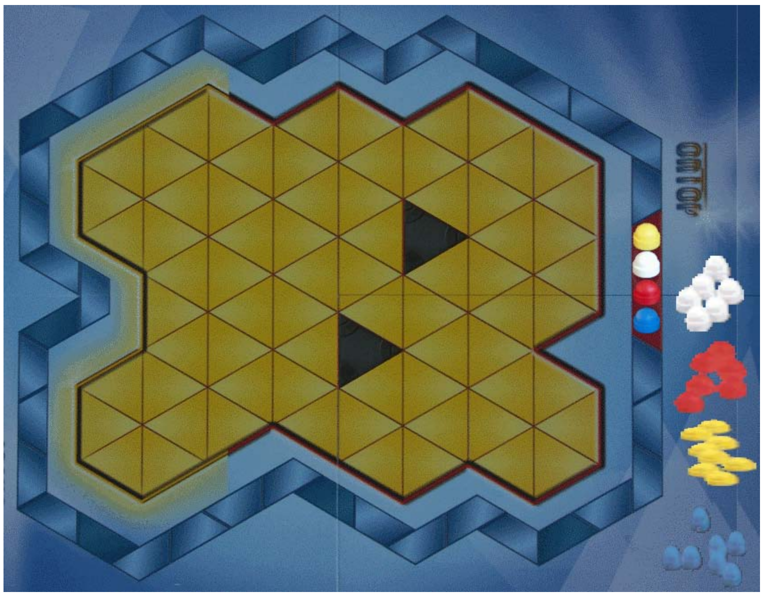


Figure 2.2: Starting position and game components.

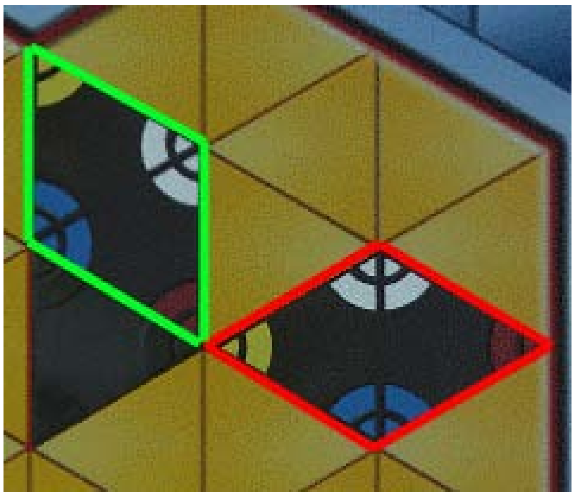


Figure 2.3: Allowed (green border) and not allowed (red border) tile laying.

During the evaluation of a circle a player which is involved in the evaluation can place one point stone, of which he has 20 pieces, on the circle. The point stone on top of a the player-stone-stack of an evaluated circle indicates the winner of the circle. One additionally point stone is always at the border to indicate the number of points obtained so far. The colours which are not assigned to a player are taken in account, too. Since four different colours are involved in the game also if only two players play the game, we differ between active and passive players. Active players participate in the game and will earn points whereas passive players are only involved when putting down point stones to evaluate a closed circle.

The game has two main terminal states. The first one is reached if there is no further possibility to place a tile, due to all positions on the game field being blocked. In this case the active player with the most points wins. The second terminal position is reached if one of the players has distributed all his point stones on the playing field. If the player who distributed all his stones first is an active player he wins "prematurely". If a passive player distributed all his point stones the active player with the most points wins.

2.2 Scoring and Stone Reduction

To terminate a game a player must either (1) use the last possible tile space or (2) use up all his point stones. Both states can be enforced by closed circles. However, not every time a circle is closed the player will be given points or distribute point stones. Two different possibilities for a terminal position of type (1) are shown in figures 2.4 and 2.5. For a better view in both figures the actually distributed point stones are removed. An example for a terminal position of type (2) is shown in figure 2.6.

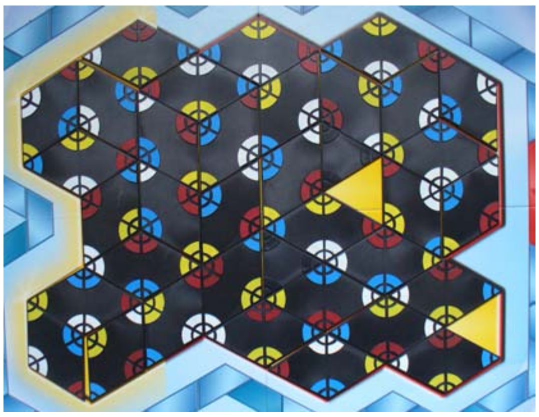


Figure 2.4: Terminal position of type (1) when all tiles are used.

Due to the shape of a tile, a tile contains two corners which can influence a circle evaluation by $\frac{2}{6}$ and two corners which can influence a circle evaluation by $\frac{1}{6}$. A circle only gives points to a player if their own colour has occupied the majority of the circle. The number of points for a closed circle is calculated as the number of colours relating to the second most occupied part of the circle plus one for the dominating colour of the circle. The colour which occupies the third most part of a circle is not taken into account for the scoring. If a circle contains a black corner, this corner is not involved to the circle evaluation, because black is not related to an active or passive player. The player on top with the most occupied part of the circle can earn four points at most for a closed circle on the normal playing field. There is one exception relating to the

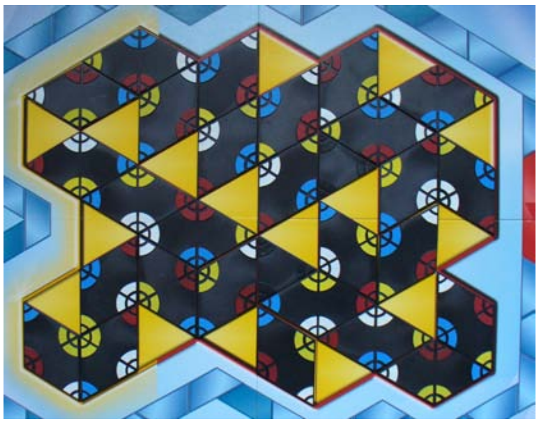


Figure 2.5: A possible terminal position of type (1) when six tiles were not used.

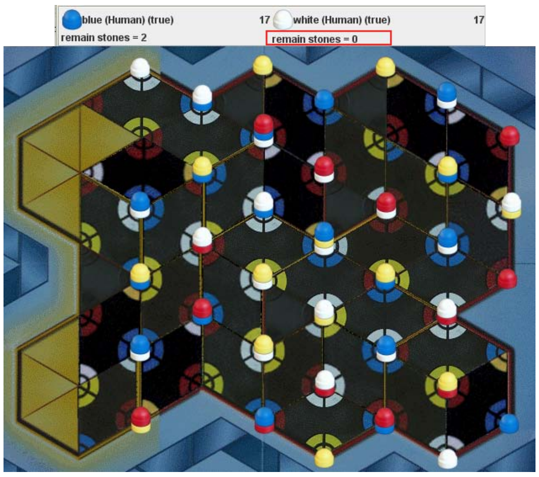


Figure 2.6: A possible terminal position of type (2).

left border of the playing field. This border is gold coloured and doubles the points of a closed circle. Hence, it is possible to earn at most six points at this border.

All players that are taken into account for the scoring will also place one of its point stone on the closed circle. The colour which wins the circle is placed on top of the point-stone-stack. A few circle configurations are listed in table 2.1. This table shows that a player can at most earn 6 points for a circle.

If a game is finished because all positions on the game field are blocked, a terminal position is reached and the winner has to be determined. At this state the points earned P_{EARNED} during the game are subtracted from the remaining point stones R_{STONES}. The final score of a player is thus determined as follows:

$$Player_{RESULT} = P_{EARNED} - R_{STONES}$$

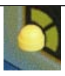
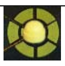
color combination	**gold border**	**points**	**picture**
2 x blue, white, yellow	true	6	
2 x yellow, blue, black	true	4	
2 x red, white, blue, yellow,black	false	4	
2 x yellow, white, blue, 2 x black	false	3	
3 x white, red, blue, black	false	3	
2 x yellow	true	2	
3 x yellow, 2 x white, black	false	2	
6 x yellow	false	1	
red, white	true	0	
3 x red, 3 x blue	false	0	
2 x red, 2 x yellow, 2 x blue	false	0	

Table 2.1: Some scoring possibilities.

2.3 Strategies

With the knowledge of the two types of terminal positions, two strategies to win the game can be derived.

The first strategy is to win by points. To win by points the player must earn as many points as possible and try to decrease the potential points of the opponents. In this case you must manipulate the circles in which the opponents are interested in such a way that the opponents get no advantage. An advantage for your opponent could be to earn more points than you or to lose point stones. Therefore the player should try to manipulate the circle in such a way that the scoring of the circle is zero or that the opponent is not taken into account for the scoring evaluation of the circle.

The second strategy to win the game is to be the first who gets rid of all point stones. This means a player wins prematurely without any attention to the score. Therefore it is necessary to pay as much focus as possible on available scoring evaluations. For this strategy, the player must only pay attention to the risk that other players may win the game prematurely before he gets a chance to do so himself.

Chapter 3

Complexity Analysis of OnTop

This chapter examines the complexity of OnTop in order to give the reader an insight into the difficulty of the game. In general, the complexity of a game is determined by two different features: the state-space complexity and the game-tree complexity [1]. The calculated complexity can be used to compare this game to other games.

In section 3.1 the game-tree complexity is calculated. Section 3.2 determines the state-space complexity of OnTop. Concluding in section 3.3 is the comparison of the complexity of OnTop to other games.

3.1 Game-Tree Complexity

The game-tree complexity is determined by the number of different games which can be played. This number is reflected in the number of leaf nodes of the game tree. To determine the size of the game tree two values are necessary: the branching factor and the game length. The branching factor is the number of possible moves per turn from which a player can chose and the game length indicates the number of turns which are needed to reach a terminal position of the game. The precision for the value of the game-tree complexity greatly depends on the settings of a game. In many cases it is impossible to calculate an exact value, which means that it will only be possible to calculate an estimate of the game-tree complexity.

The length of the game depends on the shape of the game field combined with the shape of the game tiles and the position the game tiles have on the game field. Since the game field exists of 72 triangles and a game tile consists of two triangles it is possible that a tile position can block spaces on the game field and reduce the number of useful triangle positions by one, two or three. An example of this case is shown in figure 3.1. To demonstrate the waste of space only a part of the game field is demonstrated. The left picture shows a full utilisation of space and the right picture shows waste of space. In this special case two triangles can not be used to place a tile. For this reason it is for example possible that a game finishes already after 27 turns (see figure 2.5.

However, this case is only achievable if the game does not end prematurely. If the game ends prematurely it could also be possible that the game ends even quicker.

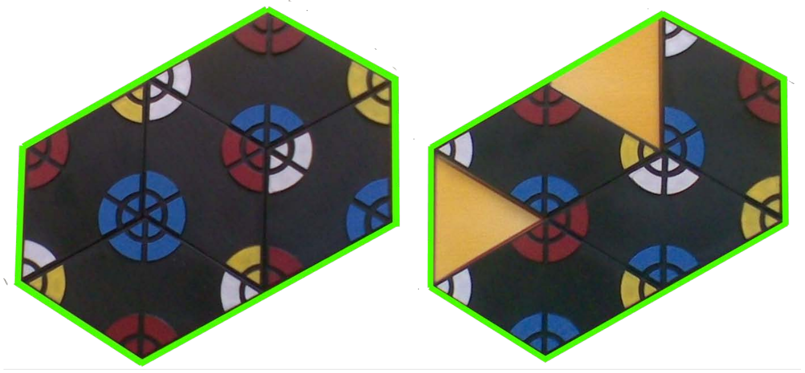


Figure 3.1: Waste of space.

The branching factor of the game depends on the number of available edges of placed tiles and indicates the number of possible moves. The number of moves for a certain state of the game field is related to the alignment of the actual tile. There are six different alignments for a tile as shown in figure 3.2. At the beginning of the game two triangles are placed on the game field. A tile can be placed on an edge of such a triangle in four different alignments. Therefore we can derive that the branching factor at the beginning is always 24.

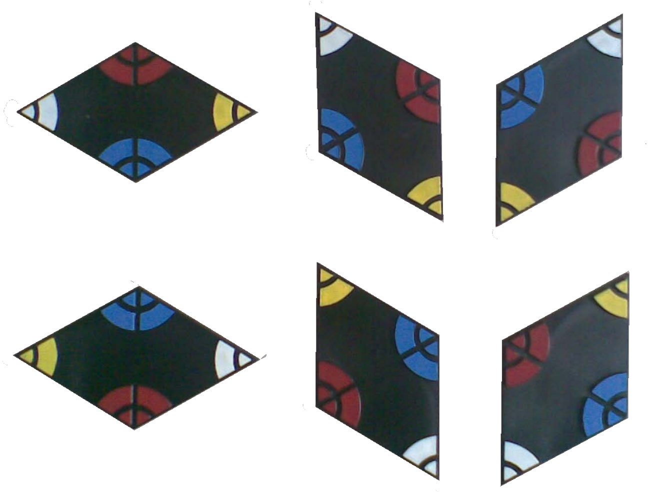


Figure 3.2: Tile alignments.

In addition the number of chance events must be included in the computation of the game-tree complexity. This value is the number of different tiles which can be drawn. There are five different tiles. However, the distribution over all tiles is not the same (see figure 2.1).

The information about the branching factor, the game length and the number of possible chance events can be used to build the following equation to compute the game-tree complexity for games which include an element of chance [30].

$$C_{GT} = (branching\ factor)^{game\ length} \times (chance\ events)^{game\ length} \quad (3.1)$$

For OnTop it is impossible to calculate an exact value as neither the branching factor nor the game length can be determined exactly.

For this reason an average value for all three values must be determined in order to calculate an estimate of the game-tree complexity. These average values are derived out of stored values from 1400 games.

Figure 3.3 shows the number of plies needed to finish a game. The average value for the game length derived from all game length values is 31.36.

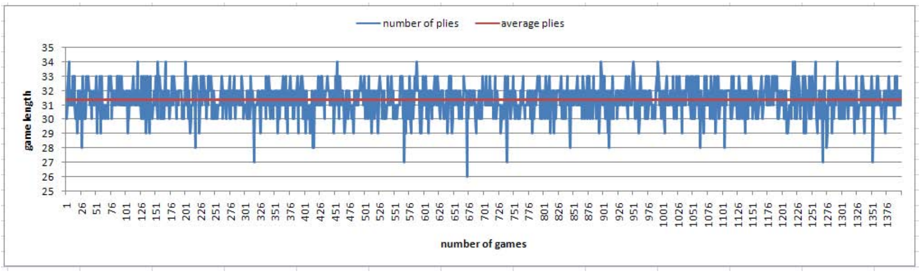


Figure 3.3: Game length of 1400 played games.

The results of the stored values related to the branching factor are demonstrated in figure 3.4. This figure shows a graph for the maximal, minimal and average branching factor stored for a certain ply out of all games. In all three graphs the branching factors increase up to 7 plies. After this point the branching factors decrease with the exception of the maximum-values graph which has its peak at 12 and 15 plies. The reason for the rise of the branching factors is that the space for tile positions increases each time a tile is placed until the game field borders get near to the placed tiles. After that the number of plies decreases. From the average graph we derived the value 23.77 for the overall average branching factor.

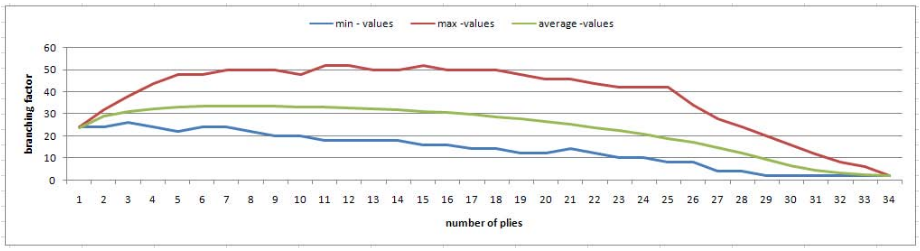


Figure 3.4: Branching factor per turn.

The last relevant value is the number of chance events which decreases during the game. Figure 3.5 shows three graphs related to the number of chance events during a game. These graphs are based on several games. Derived from this figure we use 4.16 for the average number of chance events. This value is computed as the average value of the values of the average graph.

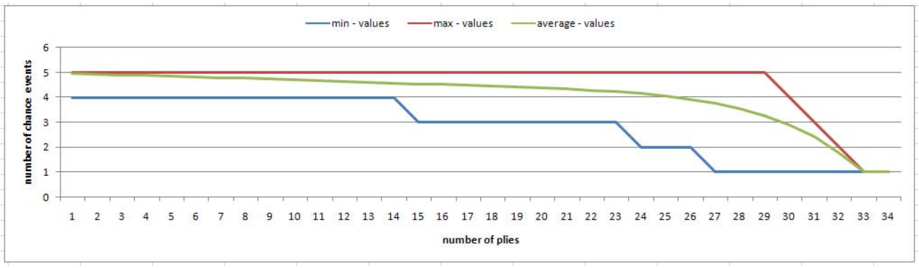


Figure 3.5: Number of chance events per turn.

Concluding, the game-tree complexity can be computed as follows:

$$C_{GT} = 23.77^{31.36} \times 4.16^{31.36} \approx 3.7 \cdot 10^{62} \tag{3.2}$$

3.2 State-Space Complexity

The state-space complexity estimates the number of all legal game positions which can occur during a game. As it is not feasible to calculate an exact value of the complexity, an approximation of an upper bound is used.

At the beginning of a game two triangle-shaped tiles are placed on the game field. Each triangle has three edges where the next tile can be placed. Furthermore, each tile can be placed in four different alignments to the edge of the triangle and all five tile types are available. For this reason it is possible to calculate the exact number of a possible board configuration when one tile is placed:

$$\underbrace{2(triangles) \times 3(edges)}_{placement\ possibilities} \times \underbrace{5}_{tile\ types} \times \underbrace{4}_{alignments} = 120 \tag{3.3}$$

From equation 3.3 we can derive that we need a value for the placement possibilities, the number of available tile types and a number for the possible number of alignments for a tile. For all possible numbers of involved tiles the board configuration must be calculated and summed.

For simplicity the value 4 is used for the alignment. This will also include illegal placements.

At the beginning of a game the number of free edges increases, because every time a tile is placed on an edge the possibilities to place a new tile increases by two. This is because the edge where the tile is placed changes their status to blocked but the new tile has three more open edges. After 7 plies the number of free edges decreases every ply. This is because the space is limited by the game field borders and every time a triangle is placed the board space becomes narrower. We use the average branching factor as value for the placement possibilities.

The last factor which influences the calculation is the number of different available tile types at every state of the game. Therefore the same value like in equation 3.2 for the number of chance events is used.

ply	configurations	ply	configurations	ply	configurations
1	395.68	12	$1.47 \cdot 10^{31}$	23	$5.48 \cdot 10^{59}$
2	$1.57 \cdot 10^{05}$	13	$5.83 \cdot 10^{33}$	24	$2.17 \cdot 10^{62}$
3	$6.20 \cdot 10^{07}$	14	$2.30 \cdot 10^{36}$	25	$8.58 \cdot 10^{64}$
4	$2.45 \cdot 10^{10}$	15	$9.12 \cdot 10^{38}$	26	$3.40 \cdot 10^{67}$
5	$9.70 \cdot 10^{12}$	16	$3.61 \cdot 10^{41}$	27	$1.34 \cdot 10^{70}$
6	$3.84 \cdot 10^{15}$	17	$1.42 \cdot 10^{44}$	28	$5.32 \cdot 10^{72}$
7	$1.52 \cdot 10^{18}$	18	$5.65 \cdot 10^{46}$	29	$2.10 \cdot 10^{75}$
8	$6.00 \cdot 10^{20}$	19	$2.24 \cdot 10^{49}$	30	$8.32 \cdot 10^{77}$
9	$2.38 \cdot 10^{23}$	20	$8.85 \cdot 10^{51}$	31	$3.30 \cdot 10^{80}$
10	$9.40 \cdot 10^{25}$	21	$3.50 \cdot 10^{54}$	32	$1.30 \cdot 10^{83}$
11	$3.72 \cdot 10^{28}$	22	$1.39 \cdot 10^{57}$	33	$5.16 \cdot 10^{85}$
				34	$2.04 \cdot 10^{88}$
					$2.046 \cdot 10^{88}$

Table 3.1: Number of board configurations per ply.

With these approximated values we can compute the number of different board configurations dependent on the number of played tiles, where n is the number of plies.

$$C^n_{SS} = (23.77 \times 4.16 \times 4)^n \tag{3.4}$$

The results of equation 3.4 are listed in table 3.1. The result for the game-state complexity is the sum of all board configurations:

$$C_{SS} = \sum_{i=1}^{n} C^i_{SS} \tag{3.5}$$

The estimated value of $2.045 \cdot 10^{88}$ is an overestimated upper bound, because it also includes illegal positions. However, the number of illegal positions is expected to be only a small fraction of this.

3.3 Comparison with Other Games

This last section of chapter 3 gives an overview of the classification of OnTop compared to other games. Figure 3.6 shows several other game complexities. OnTop is one of two games where the game-tree complexity is lower than the state-space complexity. This fact is partly related to the overestimation of the state-space complexity and the fact that the calculation of the game-tree complexity uses an average value for the length of a game. The calculated average value of 31.36 reduces the maximum possible game length by 2.64. Therefore the game-tree complexity does not contain the game-tree configurations for the search depths of 33 and 34. However, the computed state-space complexity does contain the board configurations for the cases that 33 or 34 tiles are placed on the game field, which results in a higher state-space complexity.

Based on figure 3.6 OnTop is comparable in complexity with the game of Go-Moku (15x15). Due to the high game-tree complexity, the valuation of a full game tree search is not possible. Moreover, the state-space complexity makes OnTop not solvable.

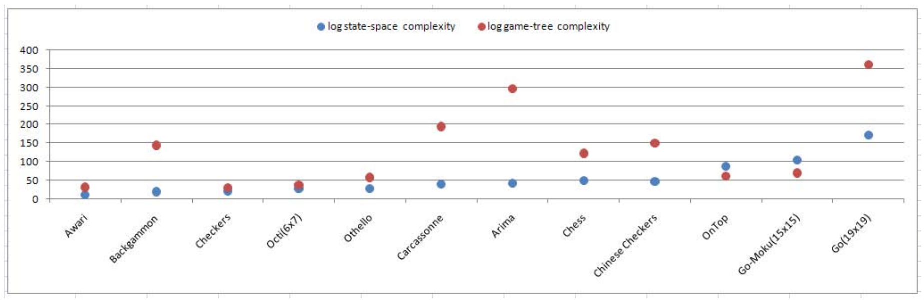


Figure 3.6: An overview of game complexities for several games.

Chapter 4

Search Techniques for Games of Chance

Due to the fact that the research related to non-deterministic board games is not as advanced as the research for deterministic games, there are also fewer approaches for implementation.

This chapter presents two search techniques which are implemented for OnTop. Section 4.1 describes the *Expectimax* search and section 4.2 describes the Monte-Carlo Search. In detail these sections provide information about related research, its implementation and possible improvements.

Both approaches will be applied to the 2-player variation of OnTop.

4.1 Expectimax

The Expectimax search is a modification of the Minimax algorithm. It expands the search tree with chance nodes which are added after every min and max node of the game tree.

4.1.1 Introduction

Expectimax search is a full-width tree search like Minimax search [22]. The difference between both search types is that Minimax only considers move nodes whereas Expectimax also considers chance nodes. A move node is related to a deterministic action done by a player and a chance node represents a chance event which occurs during the game (see section 4.1.3). The result of a Minimax search is based on the evaluation of the leaf nodes of the search tree. A leaf node of a search tree is reached when a certain search-depth limitation is reached. However, it is also possible that a search tree is not deeply investigated at a certain moment when a tree node represents a terminal position of the game before the maximal search depth is reached. During the search the decision process to choose the best node is changed after every layer. The player who initiated the search is the player to move (corresponding with the root node of the search tree). He will choose the move node with the best value. For this reason, the nodes related to the root player's choice are called max nodes. In the next layer the search considers the opponent's move. The opponent will choose the move which minimizes the move value of the root player. For this reason the nodes which are related to the opponent's choice are called min nodes. After these two steps a search depth of two is reached. This procedure will be repeated until the maximum search depth is reached. The search complexity of a Minimax tree is $O(b^d)$ where b is the number of branches of a node and d is the number of layers of the search tree. When d equals 0 only the root node is taken into account. An example of a Minimax search tree for a search depth of 3 is shown in figure 4.1.

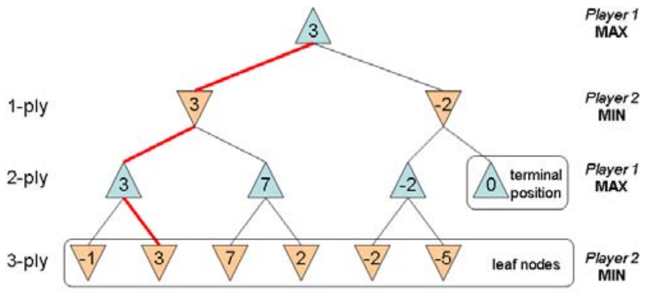


Figure 4.1: Minimax search tree.

4.1.2 Related Research

The implementation for stochastic games is an area which is less investigated. However, quite some research has been done for the board game Backgammon. For this game Expectimax and two algorithms of the *-Minimax family were applied [21], i.e., Star1 and Star2. These algorithms were proposed by Bruce Ballard in 1983 [3]. The research results for Backgammon indicate that the *-Minimax algorithms outperform Expectimax. The best results were documented for Star2. Ballard furthermore mentions that performance is not always improved by deeper search [3]. Hauk, Buro and Schaeffer describe a possible improvement by the use of the Star2.5 algorithm family [22].

Moreover, Star1, Star2 and StarETC were applied to the game of dice. As for Backgammon, the game of dice also indicates that Star2 outperforms Expectimax and Star1. It was also possible for StarETC to improve upon Star2 [38].

Recently, Star1, Star2 and Star2.5 were applied to the board game Carcassonne, which was also mentioned for future work in the research of the game of dice. In the case of Carcassonne, Star2.5 proved to be the most efficient implementation as it outperformed the other *-Minimax algorithms, as well as the Monte-Carlo and Monte-Carlo Tree Search implementation [23].

4.1.3 Implementation

A search tree for Expectimax consists of maximum and minimum nodes like a Minimax search tree. In order to investigate non-deterministic games effectively, the Expectimax search tree needs to contain additional chance nodes. These chance nodes can in principle occur at any time in a game. For this reason it is possible that a tree layer does not only consist of min, max or chance nodes, but of a combination of sll three types of nodes. However, the following research does not investigate this type of chance tree. For OnTop a chance node always follows a non-terminal min or max node. Ballard refers to this type of trees as regular trees. A comparison of both tree structures is shown in figure 4.2. The addition of chance nodes also changes the complexity computation. The search complexity of Expectimax trees is computed by $O(b^d c^d)$, where b is the number of branches for min and max nodes, c is the number of branches for chance nodes and d is the search depth.

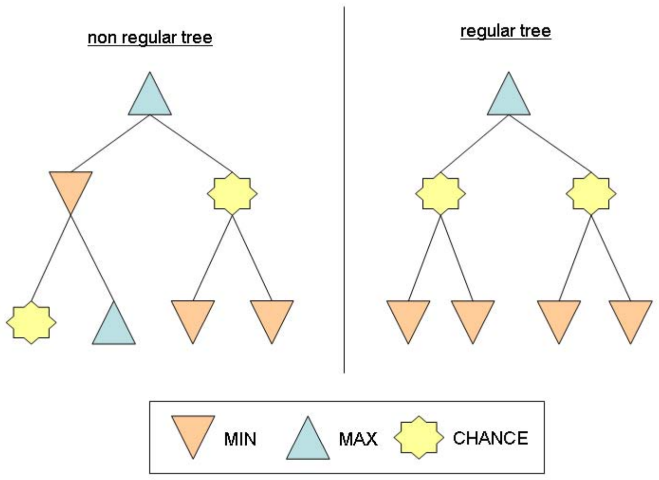


Figure 4.2: Tree structures.

In OnTop a chance event occurs when a player draws a tile. The probabilities for the existing tiles are not uniformly distributed. To compute the value of a chance node we use the sum of the product of the probability *P(s)* that a certain state s will be reached times the utility *U(s)* that this state is reached. This results in the following formula for an expected value of a chance node s_node:

$$Expectimax(s\_node) = \sum_i P(child_i) \times U(child_i) \tag{4.1}$$

Figure 4.3 shows an example of an Expectimax tree. This tree contains two max layers and one min layer and is comparable to a 3-ply Minimax tree. However, with the addition of the chance events the width of the tree grows drastically. For this research, a search until depth n indicates that the stochastic game tree contains n chance layers where *n-1* chance events occur. Figure 4.3 pictures a search until depth 3. The right branch of the root is not detailed for the figure. An edge from a min or max node to a chance node describes a possible move which can be done with the actual tile. An edge from a chance node to a min or max node describes the possibility that a certain tile is drawn. For this reason an evaluation takes place at the chance layer before the chance event occurs.

The value of the left chance node in the first chance layer is computed as $\frac{2}{6} \times (1) + \frac{4}{6} \times (-2) = -1$. For the calculation of the value for a chance node equation 4.1 is used. Figure 4.3 furthermore demonstrates that it is not possible to derive the path to the best leaf node after the first chance node is reached. For this reason it is not possible to use a principal variation search to improve the search result. It is only possible to select the best move for the actual player.

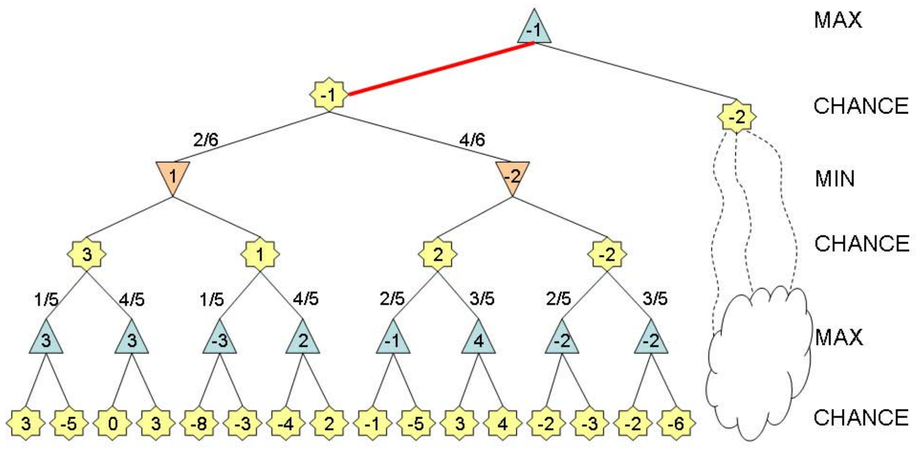


Figure 4.3: An Expectimax search tree.

The pseudocode for Expectimax is given in listing 4.1. It contains the query of a board evaluation, the execution of a chance event and the calculation of a value for a move node.

```
double expectimax(int depth) {
    score = 0;
    if (isTerminalPosition() || depth == 0) {
        return evaluation();
    }
    for (i = 0; i < numberOfTileTypesInDeck(); i++) {
        setNextPlayer();                //set next player to move
        setNewActualTileOfType(i);      //do chance event
        value = negamax(depth);
        resetActualTileOfType(i);       //undo chance event
        setPreviousPlayer();            //set actual player to move
        score += tileProbability(i) * value;
    }
    return score;
}
```

Listing 4.1: Expectimax Algorithm.

The implemented algorithm does not use the minimax notation. For simplicity a `Negamax` algorithm is applied. Using `Negamax` the algorithm does not differ between min and max nodes. This algorithm always maximises the values. Each time a value of a chance node is back-propagated the algebraic sign is switched (see listing 4.2). Therefore the evaluation of a leaf is done from the point of view from the player to move at the leaf node.

```
double negamax(int depth) {
    score = -INFINITY;
    for (i = 0; i < numberOfPosMoves; i++) {
        makeMove(possibleMoves(i));     //place actual tile
        placeSingleBlackTriangles();    //fill unplayable positions
        evaluateClosedCircles();
        value = -expectimax(depth - 1);
        removeLastCircleEvaluation();
        removeSingleBlackTriangles();
        undoMove(posMoves.get(i));      //remove placed tile
        score = max(value,score);
    }
    return score;
}
```

Listing 4.2: Negamax Algorithm.

4.1.4 *-Minimax Algorithms

The *-Minimax algorithms improve the Expectimax search. They were also introduced by Bruce Ballard [2, 3]. These algorithms apply several pruning techniques to the search tree. They try to reduce the number of searched nodes of a game tree. We introduced pruning for Minimax trees in section 4.1.4.1. Section 4.1.4.2, 4.1.4.3 and 4.1.4.4 describe the applicability of pruning techniques to Expectimax.

4.1.4.1 Introduction

The pruning technique for Expectimax is based on a pruning algorithm for Minimax trees. For Minimax trees it is possible to use Alpha-Beta search which can cut off several branches of a tree which need no further investigation. Alpha-Beta search uses a search window which is updated every time a node value is back-propagated. If the value of a child node falls outside a search window, the remaining child nodes need not be investigated and cut-offs occur. A search window is described by a lower and an upper bound.

An example is pictured in figure 4.4. This example uses the same game tree as figure 4.1. It demonstrates that the pruning technique decreases the number of examined nodes of a game tree, while the best move and the path to the best leaf node is still the same. The greatest cut-off occurs at the second branch of the second child of the root node. There the child node knows that he would only choose another branch if one of his successor nodes have a value less than or equal to -2. However, the child node furthermore knows that the root node will only choose a node with a value higher than or equal to 3. For this reason the child node can finish its search because he will only choose a value less than or equal to -2 which is never greater than or equal to 3.

The number of cut-offs which can occur during Alpha-Beta search depends on move ordering. A good move ordering ensures that the best nodes are examined first and more cot-offs will occur.

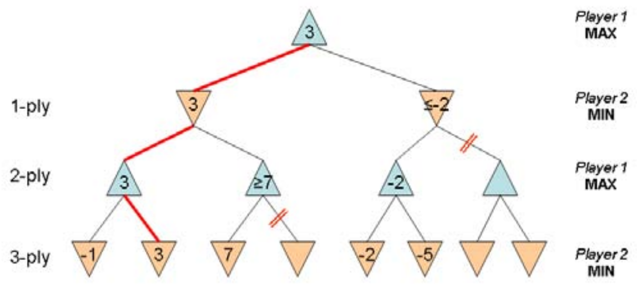


Figure 4.4: Alpha-beta tree.

4.1.4.2 Star1

The first pruning technique for Expectimax is Star1. This technique uses an $\alpha\beta$ window like Alpha-Beta search but it is only possible to have cut-offs at chance nodes. The α and β values for the search window are computed with reference to a maximum possible evaluation value U and a minimum possible evaluation value L. During the tree search the values for the search window are updated with the values of the explored nodes.

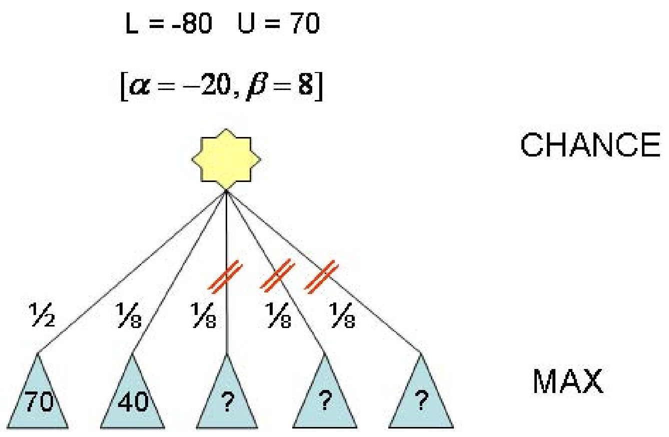


Figure 4.5: A cut-off with Star1 search.

Figure 4.5 shows a cut-off example with Star1. In this example the lower evaluation bound L is -80 and the upper evaluation bound U is 70. Furthermore, the $\alpha\beta$ window for the shown chance node is $[-20, 8]$. Star1 has already examined the first two child nodes of the chance node and the algorithm computes an estimated value for the chance node. If this value is between the search window the computation of an exact value for the chance node would be resumed with the investigation of the next child node of the chance node. If the estimated value is not between the search window a cut-off occurs. The calculation for an estimated value for the chance node contains of two steps. Firstly, the value for the visited nodes is calculated by multiplying the node value with the probability and then summed across all visited nodes. Secondly, the same is done for the non-visited nodes and added to the value derived from the first step. But these node values which do not exist are replaced by the upper or lower bound. In this case we use the lower bound. The estimated chance value is then calculated as follows: $0.5 \times 70 + 0.125 \times 40 + 0.125 \times (-80) + 0.125 \times (-80) + 0.125 \times (-80) = 10$. The result is higher than the upper bound β of 8 for the chance node. This shows that we assume that the remaining nodes will be evaluated with the minimum evaluation score, the node value would not lie in the search window and we can end the search for this chance node.

In general we must prove for a cut-off that the expected value for a chance node will not be between the given search window of α and β. To calculate an upper bound for a chance node after visiting i child nodes we use the already calculated values of the $i-1$ nodes and set the values for the remaining child nodes to the minimum evaluation value L. For the calculation of a lower bound we use the maximum evaluation value U for the remaining nodes. If we assume a uniform chance node distribution, the following equations could be used:

$$\frac{1}{N}(\underbrace{(V_1+...+V_{i-1})}_{Values\ seen}+\underbrace{V_i}_{Current\ value}+\underbrace{U\times(N-i)}_{Values\ to\ come})\leq alpha \tag{4.2}$$

$$\frac{1}{N}(\underbrace{(V_1+...+V_{i-1})}_{Values\ seen}+\underbrace{V_i}_{Current\ value}+\underbrace{L\times(N-i)}_{Values\ to\ come})\geq beta \tag{4.3}$$

For a deeper investigation of the ith child node it is possible to calculate a new alpha value A_i by rearranging equation 4.2 and a new beta value B_i by rearranging equation 4.3:

$$A_i = N\times alpha-(V_1+...+V_{i-1})-U\times(N-i) \tag{4.4}$$

$$B_i = N\times beta-(V_1+...+V_{i-1})-L\times(N-i) \tag{4.5}$$

The result of $(V_1+...+V_{i-1})$ is the exact value for the $i-1$ examined child nodes and $U\times(N-i)$ and $L\times(N-i)$ represents the best/worst case assumption for the values of the remaining nodes. However, these formulas can not be applied to OnTop, because OnTop has a non-uniform distribution of the chance events.

For OnTop the probability P_i of each node must be included explicitly [3, 22]. In the equations 4.2 and 4.3 the uniform distribution was indicated by dividing by N which is the total number of child nodes. However, the addition of the single probabilities leads to:

$$(P_1\times V_1+...+P_{i-1}\times V_{i-1})+P_i\times V_i+U\times(1-P_1-...-P_i)\leq alpha \tag{4.6}$$

$$(P_1\times V_1+...+P_{i-1}\times V_{i-1})+P_i\times V_i+L\times(1-P_1-...-P_i)\geq beta \tag{4.7}$$

These equations can be rearranged to compute a new alpha and a new beta value for a deeper investigation of a child node:

$$A_i=\frac{alpha-(P_1\times V_1+...+P_{i-1}\times V_{i-1})-U\times(1-P_1-...-P_i)}{P_i} \tag{4.8}$$

$$B_i=\frac{beta-(P_1\times V_1+...+P_{i-1}\times V_{i-1})-L\times(1-P_1-...-P_i)}{P_i} \tag{4.9}$$

For a more efficient computation in a procedural implementation it is possible to update certain parts of the equations 4.6 and 4.7 after each investigated child node. If the search enters a chance node, no child node is investigated and $(P_1\times V_1+...+P_{i-1}\times V_{i-1})$ can be initialized by $predecessor\_score_1 = 0$. After each investigated child node this value can be updated by $predecessor\_score_{i+1} = predecessor\_score_i + P_i\times V_i$. On the other side we assume that $succ\_prob = (1-P_1-...-P_i)$. The initial value is $succ\_prob_0 = 1$ and the update of $succ\_prob$ is done by $succ\_prob_i = succ\_prob_{i-1} - P_i$. With reference to these two assumptions the equations 4.8 and 4.9 can be rewritten as:

$$A_i=\frac{(alpha-predecessor\_score_i-U\times succ\_prob_i)}{P_i} \tag{4.10}$$

$$B_i=\frac{(beta-predecessor\_score_i-L\times succ\_prob_i)}{P_i} \tag{4.11}$$

Listing 4.3 shows the implementation of Star1 in which we use the derived formulas. The proof for the compliance of the $\alpha\beta$ search window is done in the used `Negamax` implementation which will work in the same way as `Expectimax` except to the window proof.

```
double star1(double alpha, double beta, int depth) {
    score = 0;
    if (isTerminalPosition() || depth == 0) {
        return evaluation();
    }
    predecessor_scores = 0;
    succ_prob = 1;
    for (i = 0; i < numberOfTileTypesInDeckSortedByPobability(); i++) {
        probability = tileProbability(i);
        succ_prob -= probability;
        cur_alpha = (alpha-U_BOUND*succ_prob-predecessor_scores)/probability;
        cur_beta = (beta-L_BOUND*succ_prob-predecessor_scores)/probability;
        ax = max(L_BOUND,cur_alpha);
        bx = max(U_BOUND,cur_beta);
        setNextPlayer();                    //set next player to move
        setNewActualTileOfType(i);          //do chance event
        score = negamax(ax, bx, depth);
        resetActualTileOfType(i);           //undo chance event
        setPreviousPlayer();                //set actual player to move
        if (score >= cur_beta)  return beta;
        if (score <= cur_alpha) return alpha;
        predecessor_scores+=probability*score;
    }
    return predecessor_scores;
}
```

Listing 4.3: Star1 search.

Star1 reduces the number of investigated nodes for a chance node but as it always thinks that the worst case occurs, it provides a lower pruning efficiency. Furthermore, Star1 has no knowledge about the type of the following nodes and is therefore pessimistic and agnostic.

4.1.4.3 Star2

The described Star1 algorithm makes no assumption about the type of following tree nodes and can be applied to any search tree with chance events. Star2 is an extension to Star1 which considers that a regular game tree is investigated (see figure 4.2). With the used `Negamax` algorithm the tree layers differ between max and chance nodes. With this knowledge the number of investigated nodes can be reduced by more efficient cut-offs. The advantage of Star2 is a preliminary probing phase.

During the probing phase a single successor of each child node is searched. If we determine during this speculative play that the probing value fails the β value, a further search of all child nodes is not necessary. If the probing value does not fail the β value, we would continue with the search phase of Star1 but we would narrow the $\alpha\beta$ window with the help of the probing values.

With a good move ordering it is furthermore possible to influence the quality of the probing phase. A good move ordering is achieved by choosing the best move first. By choosing the best move the probability of exceeding the search window during the probing phase is maximised. For chance nodes the best opportunity is to arrange them by their probability.

The implementation of the probing phase is presented in listing 4.4. The method `probing` works similar to a `Negamax` implementation except that it only searches one child.

```
double star2(double alpha, double beta, int depth){
    score = 0;
    if (isTerminalPosition() || depth == 0) {
        return evaluation();
    }
    predecessor_scores = 0;
    succ_prob = 1;
    cur_w=0;
    cur_alpha = (alpha-U_BOUND*(1-tileProbability(0)))/tileProbability(0);
    ax = max(L_BOUND,cur_alpha);
    for (int i = 0; i < numberOfTileTypesInDeckSortedByPobability(); i++) {
        probability = tileProbability(i);
        succ_prob-=probability;
        cur_beta = (beta-L_BOUND*succ_prob-predecessor_scores)/probability;
        bx = min(U_BOUND,cur_beta);
        setNextPlayer();                    //set next player to move
        setNewActualTileOfType(i);          //do chance event
        score = probing(depth, ax, bx);
        resetActualTileOfType(i);           //undo chance event
        setPreviousPlayer();
        if (score >= cur_beta) {
            return beta;
        }
        w[i] = probability*score;
        predecessor_scores+=probability*score;
    }
    cur_w = predecessor_scores;
    //Star1 search phase
    predecessor_scores=0;
    succ_pob=1;
    ....

}
```

Listing 4.4: Star2 search.

The search phase of Star2 is quite similar to Star1 except for the computation of B_i. When the search phase of Star2 occurs the probing values do not cause a cut-off but the results W_i can be used to narrow the search window. Therefore equation 4.11 needs to be modified to:

$$B_i = \frac{beta - predecessor\_score_i - W_i}{P_i} \tag{4.12}$$

where $W_i = (W_{i+1} + ... + W_N)$ is the sum of the probed values for nodes not investigated so far. Therefore line 12 of listing 4.3 must be changed to $cur\_beta = (beta - cur_w - predecessor\_scores)/probability$ and the line $cur_w -= probeScore(i) \times probability(i)$ must be added before line 12. It is possible that Star2 performs worse than Star1 when the probing phase gives no advantage and the number of additionally searched nodes increases.

4.1.4.4 Star2.5

The Star2.5 algorithm excels by its improved probing phase. Star2.5 increases the number of searched nodes during the probing phase and searches at least 2 child nodes.

Ballard introduced the Star2.5 as an additional modification to Star2 [3]. He mentioned two variations. The first variation, called 'cyclic Star2.5', searches the first child of each node and proves whether the probing value exceeds β. Then the second node is included into the search and so on. This procedure ends when a cut-off occurs or a certain number of child nodes are included in the search.

To reduce the number of probing phases Ballard explained a second method called 'sequential Star2.5'. This method searches for each probing phase by a given number of child nodes of a node. For both methods the maximum number of child nodes searched is indicated as *probing factor*.

The implementation of this sequential approach is presented in listing 4.5. The same implementation is used for Star2 except for the loop with the probing factor.

```
double probing(int probingFactor, int depth, double alpha, double beta) {
    sortMoves = moveOrdering(posMoves);
    for (int i = 0; i<probingfactor && i < length; i++) {
        makeMove(sortMoves.get(i));      //place actual tile
        placeSingleBlackTriangles();     //fill unplayable positions
        evaluateClosedCircles();
        score = -star2(-beta, -alpha, depth-1);
        removeLastCircleEvaluation();
        removeSingleBlackTriangles();
        undoMove(sortMoves.get(i));      //remove placed tile
        if (tmpScore >= beta) return beta
        if (tmpScore > alpha) alpha = score;
    }
    return alpha;
}
```

Listing 4.5: Probing in Star2.5.

4.2 Monte Carlo

Another approach to implement a player for OnTop is the Monte-Carlo Search. This approach is useful for a system with uncertainty. In artificial intelligence this technique performs many simulations for a system. For this purpose the average value of the number of observations is taken. An advantage of this approach is that no strategic knowledge is needed. The only information needed is how to play the game to perform simulations and scoring terminal positions.

4.2.1 Related Research

In the beginning Monte-Carlo-based techniques were used in physics and mathematics. The name "Monte-Carlo method" was originally related to the Polish mathematician Stanislaw Ulam 1945. He worked together with John von Neuman on the Manhattan Project during the World War II [27]. His invention of the Monte-Carlo method was connected to the game Solitaire [16].

However, it was only in the last few years that the relevance of Monte-Carlo Search in games increased and the results became sufficient. The best example is the implementation of a computer player called MAVEN [36] for the imperfect-information game Scrabble. This player plays better than the best human players [35]. Another game worth mentioning is Backgammon, for which a challenging player implementation called TD-Gammon was created, which also has the potential to beat top human players [37]. Monte-Carlo Search is also applied to the game of Go [7]. However, due to Go's large complexity a perfect computer player for Go has not been built yet and no applied technique has yet achieved to beat a top human player [29].

An improvement to Monte-Carlo Search is the is the Monte-Carlo Tree Search (MCTS) which is a best-first search that "is guided by the results of

Monte Carlo simulations" [40] which is applied to the game of Go [14], too. Other implementation examples are Clobber [19] and Amazons [25]. A further enhancement of MCTS is UCT (Upper Confidence bound applied to Trees) which tries to balance searches (exploitation versus exploration). UCT is a selection technique to traverse a search tree. This approach has been applied to Go [17]. UCT improved the strength of computer players of Go to a significant degree. For Go 9 x 9 five of the six best programs use UCT [39].

4.2.2 Implementation

The implementation of the Monte-Carlo algorithm consists of three steps. These are: (1) determine all possible moves; (2) produce random sample games and evaluate each of them; (3) choose the move with the highest winning probability.

The starting point of each Monte-Carlo Search is the current state of the game. For this state all legal possible moves are determined. For OnTop there are at least two possible moves per turn because if a tile can be placed it is also possible to place the same tile rotated by 180 degrees as another move. For the next step the Monte-Carlo technique plays out random games for every possible move. The number of random games for a move depends on the time the player has to search for the best move, the number of possibilities and the progress of the game. If we have a search time of 3000 milliseconds, 24 possible moves for the current state and each game simulation needs 5 milliseconds, we could do 25 simulations per move. However, with each turn also the length of a simulation decreases and thus the time for a simulation decreases, too.

The result of a simulated game is determined by an evaluation of the end state of the simulation. The evaluation determines if the simulated game was a win, loss or draw for the actual player. For this general evaluation the evaluation returns 1 for win, -1 for a loss and 0 for a draw. Furthermore, there is the possibility to improve the information of the evaluation. Therefore the evaluation includes and returns the margin of the simulated game. If the actual player won the simulated game, the margin would be the difference between his own points and the points of the second-best player. If the actual player did not win the game the margin is calculated as the difference between the points of the player who wins the game and his own points. The returned value for this simulation is added to a score value for the simulated move. The pseudocode is shown in listing 4.6.

```
Move MCSearch(){
    node_i = 1;
    while(timeLeft){
        posMovesData[node_i] = play_game_simulation(node_i);
        node_i += 1;
        node_i = node_1 modulo posMoves_size();
    }
    return select_Move_with_highest_winrate(posMovesData);
}
```

Listing 4.6: Monte-Carlo Search

If the Monte Carlo player runs out of search time, he chooses the move with the highest winning rate $BEST_{WR}$. The winning rate WR for a move i is calculated for each move by dividing the score of the move MS by the number of simulations for the move NS:

$$WR_i = \frac{MS_i}{NS_i}$$

$$BEST_{WR} = \max_{1 \leq i \leq N} WR_i$$

where N is the number of possible moves.

4.2.3 Monte-Carlo Tree Search (MCTS)

Monte-Carlo Tree Search (MCTS) is a best-first search algorithm which is based on the Monte-Carlo Search. The normal Monte-Carlo Search simulates random games for all possible moves from the root and scores the game on the basis of the average outcome of the played simulations for a move.

MCTS plays random games, too. However, these random games are also related to states after some plies. Therefore MCTS creates a game tree which is expanded depending on the results of simulated games. For this reason, the distribution of played games should be higher for moves with a higher win probability.

In contrast to Monte-Carlo Search, MCTS take into consideration chance events. Chance events with a higher probability will be searched first. The constructed game tree contains two different types of nodes: chance nodes and move nodes. Both node types store at least two values: the number of visits and a score value which indicates the obtained score after all simulations are done from the current node. Both values are used to walk through the game tree to find the most promising leaf node outcome for a further simulation.

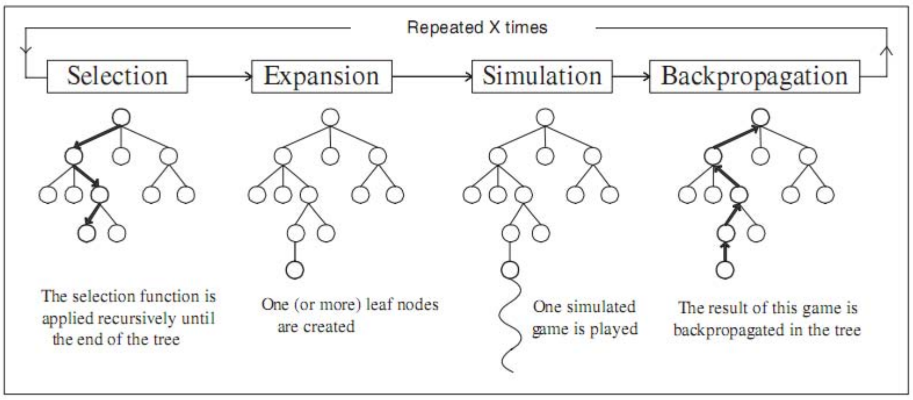

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until the end of the tree
One (or more) leaf nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Figure 4.6: Scheme of Monte-Carlo Tree Search.

Monte-Carlo Tree Search consists of four steps, which are repeated as long as there is time left. A short overview of the four steps is pictured in figure 4.6 (taken form [12]). (1) During the *selection step* the algorithm traverses recursively through the game tree until a leaf node L is reached. (2) The *expansion step* stores one (or more) child nodes C of L in the game tree. (3) The *simulation step* simulates the game from the expanded node C until the end of the game is reached. (4) Finally, the result of the simulated game is back-propagated through the tree in the *backpropagation step*. When the time is up, the best child node is chosen to play (see section 4.2.4.1). A more detailed description of the four steps mentioned above is given in the following paragraphs.

Selection: The first step of MCTS is to select the best node to traverse the tree to determine the best leaf node. Therefore it is important to differentiate between chance nodes and move nodes. If the child nodes are chance nodes, the selection of a chance node is done randomly. However, the non-uniform distribution of chance events influences the probability to choose a certain tile. If the child nodes are move nodes, a selection strategy is applied. The selection strategy controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best results so far (exploitation). On the other hand, also the less promising moves must still be tried out, due to the uncertainty of the evaluation (exploration). This problem is similar to the Multi-Armed Bandit problem [13].

The chosen selection strategy for this implementation is the Upper Confidence bound applied to Trees (UCT) [26]. This strategy chooses the move i which maximises the following formula:

$$wr_i + C \times \sqrt{\frac{ln\ v_p}{v_i}} \tag{4.13}$$

where wr_i is the winning rate of node i, v_p is the number of visits of the parent node of i and v_i is the visit count of i. The winning rate for i is computed by dividing the number of wins after visiting i divided by v_i. C is a coefficient, which has to been chosen experimentally. A high C is linked to a high degree of exploration and an increased number of investigated game tree paths. A small C increases the risk of potentially neglecting a good move, hence the degree of exploitation is higher.

Another feature which is added to the selection strategy compares the number of visits to the threshold k, i.e., when the number of visits of the current node is smaller than a threshold k the selection of a child node is done randomly. This means that only after k visits of a node the UCT- selection is applied. If UCT tries to compute equation 4.13 for a node which was not visited yet, the maximal evaluation value is returned. For this reason it is possible that not all nodes of a tree layer are investigated.

Expansion: During the expansion step it is decided which node is added to the game tree. This decision depends on the selection step. The simplest rule, proposed by [14], is to expand one node per simulation. The node expanded corresponds to the first position which has not been stored yet.

Simulation: When a new move node is added to the game tree the simulation step starts. Hence, the remaining moves to finish the game can be done on a pure random basis or pseudo-randomly. The same applies to the draw of a tile. If the moves are done pseudo-randomly it is possible to guide the simulation to even better results. The game simulation returns an evaluation value for the finished game. This value can either be a relative value for win, draw or loss like -1, 0, 1 or an absolute value which indicates, for example, the point difference between the players.

Backpropagation: The last step back-propagates the result of a game simulation through the game tree. Each node which is visited during the backpropagation updates its visit count and its value which indicates the winning score after searching this node.

4.2.4 Enhancements

This section investigates possible enhancements of MCTS, with the aim to improve the results and its efficiency.

4.2.4.1 Final Move Selection

Once the available time is elapsed the AI must choose the best possible move. To achieve this, the information produced from the MCTS is used. There are several ways to select the best move:

- (a) *Max Child:* select the move with the highest visit count
- (b) *Robust child:* select the move with the highest winning rate
- (c) *Robust-max child:* select the move which fulfills (a) and (b)

In [12, 40] a *secure child* node was mentioned, which is not investigated for this research due to the limited time frame available. The implemented MCTS first applies (c). If such a child node does not exist then (a) or (b) is chosen. During the tests we figured out that selecting the move with the highest winning rate (b) performs best when no robust-max child exists.

4.2.4.2 Progressive Bias with History Heuristic

In order to improve the selection strategy it is possible to use a progressive-bias strategy. The progressive-bias strategy includes some degree of domain knowledge of the actually used selection strategy. The first games selected by MCTS are significantly influenced by the addition of the heuristic knowledge. The intention is, that with a higher number of simulations the knowledge-based selection strategy converges to a UCT selection strategy [12]. After adding a heuristic value MCTS selects the node which maximises the following formula:

$$wr_i + C \times \sqrt{\frac{ln\ v_p}{v_i}} + f(v_i) \tag{4.14}$$

For OnTop $f(v_i)$ is calculated as $\frac{H_i}{v_i+1}$ where H_i is a value representing the heuristic knowledge [12]. The other variables of equation 4.14 are the same as for equation 4.13.

The value of H_i depends on statistic values based on previously done simulations. Therefore, during the simulation phase information will be stored for all placed tiles, including where they are placed and how often they were placed on a certain position. When the simulation phase for a node ends, the result of the simulation is added to all involved moves of the simulation. This heuristic is comparable with the history heuristic [32]. The value which can be selected from the stored values gives a relative indication about the winning probability of a tile when it is placed on a certain position with a certain orientation. For this purpose, the reached score is divided by the number of placements. This winning probability is indicated by H_i. To store the relevant information, a four-dimensional array is used. One dimension is used for the tile type, one for the orientation and two for the occupied triangle positions on the game field.

The knowledge of this heuristic increases by the number of simulations carried out. It is possible that at the beginning of a game the selection strategy can hardly benefit from the heuristic value. However, if the values are stored during the whole game, the benefit increases with every ply. A disadvantage of this heuristic is that we have only a value for a single tile on the board but we have no knowledge of the board configuration that lead to the value for a tile. For this reason it is also possible that a good heuristic value for a tile can lead to an unfavourable decision.

4.2.4.3 Dynamic Simulation Cut

During the simulation phase of MCTS a game is played randomly from a certain state until a terminal position is reached. It is also possible that the game score of one player indicates a unique winner n plies before the simulation ends. This case occurs if the winner or loser of a game at a certain state is the same at each possible terminal position. A similar approach is also mentioned by Bouzy as “mercy rule” [5]. Therefore, it would be efficient to end the simulation and use the saved time for further simulations. In order to achieve a real improvement, the function has to test whether a player’s score indicates, with significant reliability, a unique winner before the simulation ends. This function should be fast and simple.

For OnTop it is possible to compute the score difference of both players. If this value is higher than the number of tiles n which can still be placed (number of remaining moves to finish the game) times SA, the function could end the simulation and return the score. SA is a coefficient indicating the number of points a player can score with a placed tile. This value can be amended experimentally. A specific example is shown in figure 4.7, where $n = 4$ and $SA = 3$. In this example, there are four moves before the simulation can reach a terminal position and we investigate whether it is necessary to continue the simulation. Player one has 30 points and player two has 10 points. For this reason player one is the winner at this position. With an estimated average score SA of 3 player two can at most get 22 points during the last 4 plies. This estimated score is still lower than the actual score of player one and a cut-off occurs.

4.2.5 Parallelisation Approach

The main quality criterion of the Monte-Carlo Search is the number of simulations done in a specific time frame. To increase the number of simulations it is possible to increase the time the player is allowed to search for the best move. However, this approach destroys at a certain moment the illusion of a human-like behaviour. A more promising approach is parallelisation.

4.2.5.1 Monte-Carlo Search

For the Monte-Carlo Search the parallelisation approach uses several threads to compute the result of several simulations at the same time. In the best

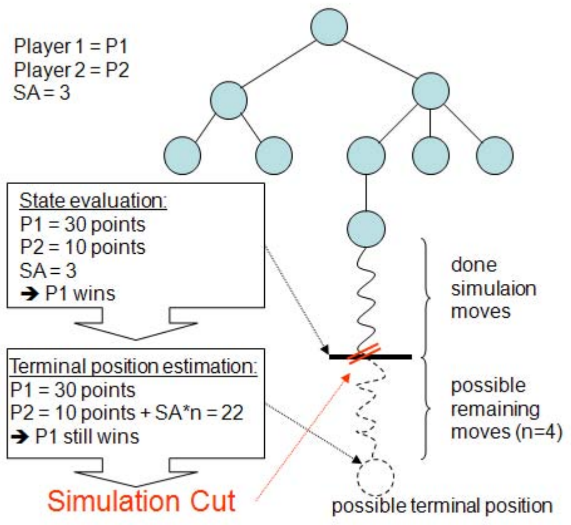


Figure 4.7: Simulation cut for MCTS.

scenario the number of simulations with parallelisation is the number of simulations without parallelisation times the number of threads. The effect on the implementation is that a thread does not choose any next possible move but the next possible move which is not blocked by another thread. If each thread needs every time the same amount of time for a simulation, the next possible move it would choose for a simulation is the actual move index m_{index} increased by the number of threads th_{size} modulo the number of possible moves m_{size}:

$$next\_move\_index = (m_{index} + th_{size})\ mod\ m_{size}$$

4.2.5.2 Monte-Carlo Tree Search

There are several parallelisation techniques which can improve the efficiency of MCTS [10, 11, 9]: leaf parallelisation, root parallelisation, and tree parallelisation. The difference between these three techniques concerns the part of the search tree which is parallelised. Figure 4.8 (taken from [11]) pictures the three techniques.

Leaf Parallelisation: One of the easiest ways to parallelise MCTS is *leaf parallelisation*. This technique starts the parallelisation process with the simulation phase of MCTS. At the beginning the main algorithm traverses down the search tree until a leaf node is reached and one or more child nodes are added to the tree. During the next step n independent simulations are started from the leaf node, where n is the number of threads used for the simulations. Only after all threads finished their simulations, the main thread starts to back-propagate a computed average result of the simulation threads. This technique was introduced by Cazenave and Jouandeau [9]. The advantage of the *leaf parallelisation* is the easy implementation process. However, there are also disadvantages;

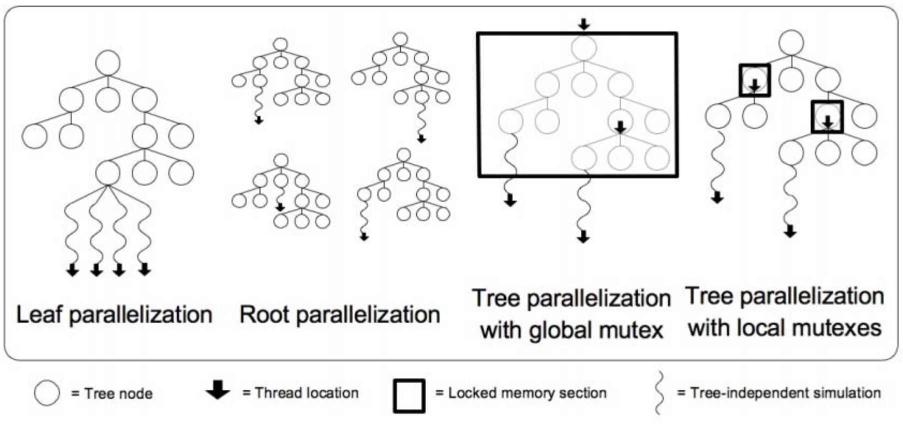

Leaf parallelization
Root parallelization
Tree parallelization with global mutex
Tree parallelization with local mutexes
= Tree node
= Thread location
= Locked memory section
= Tree-independent simulation

Figure 4.8: (a) Leaf parallelisation (b) Root parallelisation (c) Tree parallelisation with global mutex and (d) with local mutexes.

for instance the required time of a thread to finish a simulation is highly unpredictable. The main algorithm must wait until the last thread finishes his simulation, even if $n - 1$ threads are ready and only one thread is still running [11]. This means that the advantage of *dynamic simulation cuts* may be lost (see section 4.2.4.3). Another disadvantage is that each thread operates independently and does not share information. For example, if more than $\frac{n}{2}$ threads have already finished their simulation and their results lead to a loss, it would be highly probable that the results of the remaining threads will also lead to a loss. In such a case it would be preferable to terminate the remaining threads before they finish their simulations in order to use the time more efficiently [11].

Root Parallelisation: The second parallelisation technique, which was also proposed by Cazenave under the name "single-run" parallelisation is *root parallelisation*. For this technique each thread manages one MCTS tree in parallel and the threads do not share information with each other. When no further simulation time is left the results of the child nodes of the root node of each MCTS tree are merged together and the best move is selected based on the results of all MCTS trees [11, 9]. The implementation effort is a little bit higher than for the leaf-parallelisation technique. One disadvantage is, that more memory space is required, which might increase by a factor of n (n= number of threads), because each thread stores an independent game tree in memory.

Tree Parallelisation: Chaslot, Winands and van den Herik introduced a third technique called *tree parallelisation* [11]. For this technique, all threads use a shared game tree and each thread is able to exchange information of the tree. To prevent that a thread *th1* will exchange information which will then be used by thread *th2*, the threads can lock certain parts of the search tree from time to time with mutexes. In order to improve the performance Chaslot, Winands and van den Herik propose the methods "mutex location" and "virtual loss".

Chapter 5

Experiments and Results

In order to be able to recommend the best search technique for OnTop, the following sections give an overview of the tests carried out and the results obtained. Section 5.2 describes tests related to the implemented Expectimax and *-Minimax algorithms. Section 5.3 presents the test results of Monte-Carlo and Monte-Carlo Tree Search. Concluding, the best player configurations of section 5.2 and 5.3 are tested against each other in section 5.4. Finally, section 5.5 compares the level of play of the best AI player with that of some human players.

5.1 Settings

This chapter describes some configuration possibilities for an Expectimax and Monte-Carlo player. For this purpose, 10 board configurations are defined in order to be able to compare the results of Expectimax, Star1, Star2, Star2.5, Monte-Carlo and Monte-Carlo Tree Search.[1] The following table 5.1 shows some additional information about these configurations. The table contains the number of possible moves a player can make in the position and the progress of the game. All results, which are related to these board configurations, assume it is the blue players's move. The configuration set consists of opening, midgame and endgame positions.

	1	2	3	4	5	6	7	8	9	10
pos. move	24	38	42	32	18	46	52	24	34	10
game turn	1	8	9	16	25	7	8	24	18	29

Table 5.1: Information about board configurations.

5.2 Expectimax

Section 5.2.2 shows the influence of a move ordering on the number of investigated moves. The construction of the evaluation function used and the weight-

[1] All 10 board configurations are listed in Appendix A.

ings of certain features are described in section 5.2.1. A comparison of investigated nodes between Expectimax, Star1 and Star2 is listed in section 5.2.3 followed by an estimation for the best lower and upper bound configurations for all *-Minimax algorithms. The last test is done for the best probing factor for Star2.5. Concluding, the best configuration derived from the results of this section is presented in section 5.2.6.

5.2.1 Evaluation Function

The value of a certain state of a game is computed by an evaluation function which contains several features f_i to evaluate a state. The influence of the different features to the value of a state differs. Therefore, the value of a feature is weighted by w_i. Finally, the value of a state is computed by the sum of all weighted features of a state. Table 5.2 shows different weighting configurations for the different features. As features the point difference, the stone difference and a point potential value are used. When the player knows that he wins or loses, this fact is scored additionally. The point potential is a value which indicates the probability that the player scores points in the future. The potential value is the sum of the percentages of its own colour on the open circles divided by the number of open circles where the players colour is involved in.

For testing, five different Star2 configurations played against an Expectimax player. They played 200 simulations for each combination where each player had three seconds to choose the best move. It is striking that all configurations in general try to finish the game by the highest score value. This is also evidenced by the low percentage for a premature win. The last configuration is the only one where the focus was set to the stone difference which should lead to more premature wins. This is also observed, as evidenced by the higher rate of premature wins. However, this has also deteriorated the probability of winning. Due to this fact the Star2 player played worse than the Expectimax player. The best probability of winning was reached with configuration 4 which does not include a potential score.

	configuration				
features f_i	1	2	3	4	5
point difference	3	6	1	5	1
stone difference	1	3	0.25	2	3
score potential	1	1	0.05	0	2
win score	100	100	100	100	100
win rate	**59%**	**70%**	**64%**	**75%**	**48%**
win prematurely	17%	14.2%	13.3%	18.6%	25%

Table 5.2: Winning rate for different weighting configurations for the used evaluation features.

5.2.2 Move Ordering

A possibility to save search time would be to search the best move first. Even though it is not possible to know the best move before the search starts, it is possible to increase the probability of the best move being searched first. For this purpose, the move ordering is applied to the possible moves. The move ordering evaluates the probability to get points by choosing a certain move. Therefore, we look to all open circles which are influenced by a move. If after playing a tile the related circles have a high share of the actual player's colour, the probability to get points after the move or after further moves would be higher than the other way around. An example for two possible moves is given in figure 5.1. Both pictures show a possible move (green bordered) for player white and the currently percentage of its colour parts of the open circles influence by this move. Move ordering can only be applied to Star1, Star2 and Star2.5 where cut-offs can occur.

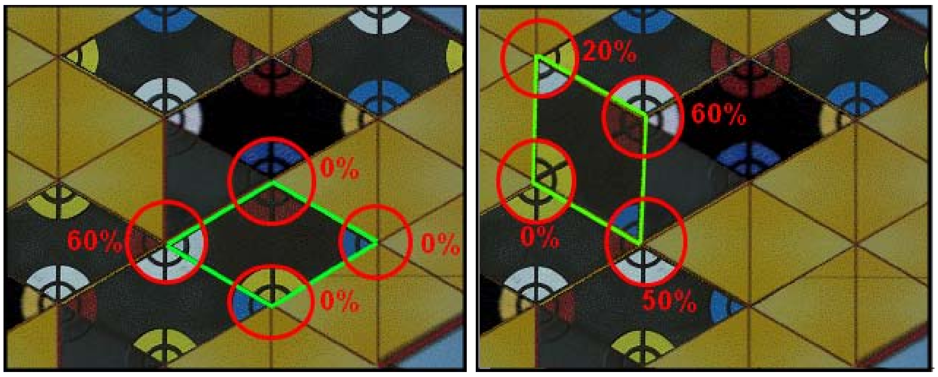


Figure 5.1: Move-ordering example.

The results on move ordering are given in tables 5.3 and 5.4. The tables contain the percentage reduction of searched nodes without move ordering $\overline{mo}$ and with move ordering mo. The column '$gain_{mo}$' lists the percentage reduction of searched nodes with move ordering in contrast to the searched nodes without move ordering. The value is calculated as follows:

$$gain_{mo} = \frac{\overline{mo}-mo}{100-\overline{mo}}$$

All results are related to the board configuration set.

Table 5.3 shows the results for a search depth of two. The highest average gain is reached for Star2. However, the results also demonstrate that it is possible that the number of nodes searched is increased with move ordering. This happens for configuration 1 for Star2 and configuration 10 for Star1.

For table 5.4 the search depth is set to 3. The results show for Star2 that with increased search depth, the average node reduction can be further increased from 39.98 to 60.89. However, this behaviour does not occur for Star1. Additionally, the percentage reduction for the configured boards 8, 9 and 10 deteriorated compared to the search depth of 2.

5.2.3 Node Investigations

In order to determine the advantage of pruning in stochastic game trees, Expectimax, Star1 and Star2 are applied to the test board configurations. For this test the number of investigated nodes and the search time is stored. The

board	Exp vs. Star1			Exp vs. Star2		
states	$\overline{mo}$	mo	$gain_{mo}$	$\overline{mo}$	mo	$gain_{mo}$
1	48.14	49.36	2.36	85.81	84.24	-11.07
2	71.03	82.25	38.72	82.77	97.23	83.94
3	71.89	82.87	39.05	84.85	97.5	83.48
4	66.34	70.41	12.07	91.14	93.73	29.18
5	80.09	92.61	62.87	83.14	93.36	60.62
6	71.39	77.38	20.94	89.86	95.84	58.97
7	67.81	71.39	11.14	87.97	92.26	35.61
8	86.71	90.28	26.87	89.15	94.33	47.77
9	87.37	90.10	29.53	93.43	93.98	8.34
10	37.39	37.33	-0.09	49.17	50.66	2.49
			24.35			**39.98**

Table 5.3: Reduction of nodes investigated with move ordering with a search depth of 2.

board	Exp vs. Star1			Exp vs. Star2		
states	$\overline{mo}$	mo	$gain_{mo}$	$\overline{mo}$	mo	$gain_{mo}$
1	62.77	67.17	11.81	90.31	95.87	57.39
2	77.59	86.03	37.64	93.07	99.52	93.06
3	79.09	85.08	28.68	94.18	99.41	89.85
4	74.16	77.41	12.56	97.50	99.02	60.71
5	82.51	88.63	34.99	92.44	97.58	68.02
6	79.23	83.18	19.04	97.01	99.37	79.07
7	77.47	80.46	13.27	97.52	98.96	57.93
8	75.28	76.12	3.4	92.11	95.84	47.24
9	84.6	86.58	12.84	95.76	97.97	52.23
10	31.91	32.08	0.24	48.84	50.55	3.35
			17.45			**60.89**

Table 5.4: Reduction of nodes investigated with move ordering with a search depth of 3.

number of investigated nodes is the same for every search technique applied to a board configuration, but the used time depends on the utilisation of the CPU used. For this reason, the search time is only mentioned in specific cases.

Figures 5.2, 5.3 and 5.4 below compare the results of all three search techniques, where figure 5.2 presents the results for search depth 1, figure 5.3 for search depth 2 and figure 5.4 for search depth 3. A search depth n includes n chance events. Furthermore, table 5.5 presents the maximum, minimum and average gain for the investigated nodes of Star1 and Star2 compared to Expectimax.

Figure 5.2 shows that the *-Minimax algorithms decrease the number of investigated nodes even for a small depth. As expected Star2 shows the best performance and reaches an average search gain of 78.57%. However, with regards to the board configurations two and three, Star1 outperforms Star2, which probably is due to the additional nodes that are searched during the probing phase.

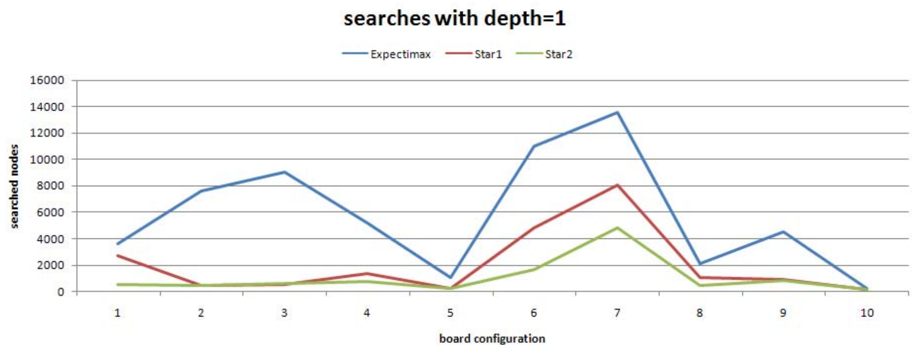


Figure 5.2: Number of investigated nodes for several board configurations with a search depth of 1.

With a search depth of 2 it is no longer possible for Star1 to outperform Star2 (see figure 5.3).The comparison of the search depth shows that Star2 shows an average gain of 10% whereas Star1 only shows a gain of 8%.

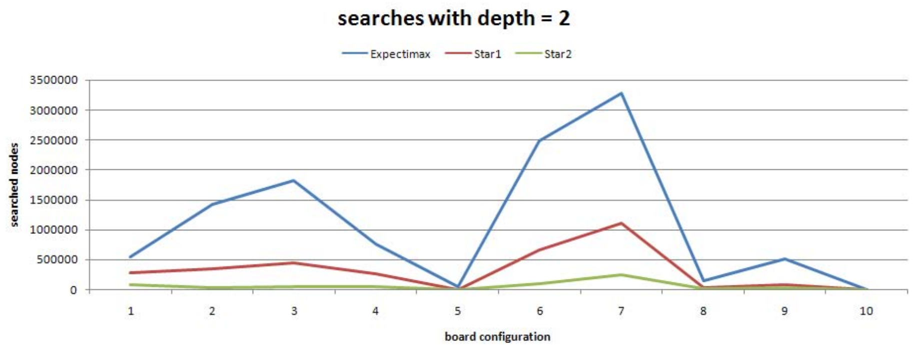


Figure 5.3: Number of investigated nodes for several board configurations with a search depth of 2.

For search depth 3 the average gain is furthermore improved by 5% for Star1 and Star2. The maximal search gain is reached with Star2 for configuration 2 with a gain of 99.48%. This is a reduction from 264,576,932 nodes to 1,366,584 nodes. Furthermore, Expectimax, Star1 and Star2 choose every time the same move for a certain board configuration with a certain search depth. An interesting fact is that the performance of Star1 is more similar to Star2 as it is to Expectimax. This was also the case in [3, 23].

At the beginning of this subsection we mentioned that the search time cannot be regarded as a significant evaluation feature for the performance of the search techniques. Nevertheless some results of the used search time are presented in table 5.6. These results are taken from a search with depth 2 for three different board configurations.

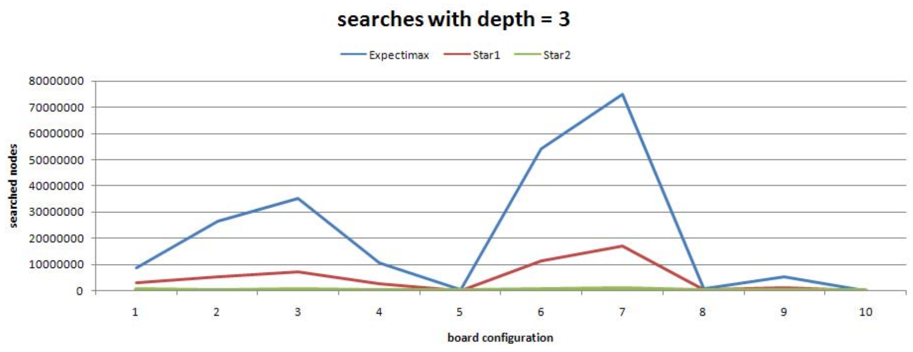


Figure 5.4: Number of investigated nodes for several board configurations with a search depth of 3.

		depth = 1		depth = 2		depth = 3	
		Star1	Star2	Star1	Star2	Star1	Star2
gain	MIN	18.27%	37.98%	32.74%	50.66%	29.37%	50.55%
gain	AVG	60.23%	78.57%	68.71%	88.98%	73.39%	93.37%
gain	MAX	94.07%	93.92%	87.39%	97.46%	85.71%	99.48%

Table 5.5: Minimal, maximal and average gain of nodes investigated with Star1 and Star2 for different search depths.

	position = 10			position = 4			position = 7		
	nodes	time	gain	nodes	time	gain	nodes	time	gain
Star1	1216	31	50%	268092	4406	54.37%	1106999	14265	52.4%
Star2	892	31	50%	48490	1062	89%	254663	4609	84.62%

Table 5.6: Time gain of Star1 and Star2 compared to Expectimax with a search depth of 2.

In most cases, the reduction in time for a search was more than 50%. This fact is also highlighted in table 5.6. The gain values for the time reduction are related to an Expectimax search. The node values are only listed for a better classification of the position. Furthermore, the time was measured in milliseconds. Nevertheless for a search depth of 1 it also occurs that Star1 and Star2 have shown no time reduction.

Concluding, the *-Minimax algorithms outperform Expectimax significantly in search time and with regards to the number of investigated nodes. The results of Star2 were the best. With deeper search and on midgame positions the best performances were reached from Star1 and Star2. For a human-like behaviour it should be possible for Star2 to search at least a depth of 2 in each position. However, a deeper search is probably also possible for most positions.

5.2.4 Bounds Configuration for *-Minimax

The number of cut-offs which occur during the search depend on the given search window. The bounds of the search window should be the minimal and maximal possible evaluation value of a state. For the implementation the initial bounds are calculated before every search. The bounds depend on the search depth and an average value APT which indicates the points per placed tile. The bounds are calculated as follows:

$$L\_BOUND = (own\_score - opponent\_score) + (own\_stones - opponent\_stones) - APT \times searchDepth$$

$$U\_BOUND = (own\_score - opponent\_score) + (own\_stones - opponent\_stones) + APT \times searchDepth$$

With a smaller APT value the number of searched nodes during the search decreases. This is the result of the narrowed search window which leads to faster cut-offs. This can save time for a possible deeper search. However, the results of table 5.7 show that a small APT value decreases the winning rate. The results for an APT value of 2, 3, 4 and 5 are nearly the same. Also the number of searched nodes has hardly any influence.

	APT value				
time (ms)	1	2	3	4	5
3000	56.9%	64.5%	66%	65.5%	63%
5000	60.6%	67.5%	69%	71%	65%

Table 5.7: Influence of assumed average tile scores for *-Minimax bounds.

5.2.5 Optimal Probing Factor for Star2.5

To improve Star2 it is possible to investigate more child nodes during the probing phase which should increase the probability to reach a cut-off during the probing phase or an earlier cut-off during the search phase. The number of child nodes searched during the probing phase is called probing factor (pf). The tables 5.8, 5.9 and 5.10 show the results for the 10 test positions with different probing factors for the search depths one, two and three. In each table the column Star2 shows the number of visited nodes with Star2. The columns for the probing factors show the gain of investigated nodes. For the purpose of this investigation a positive value indicates a search reduction.

Table 5.8 shows the results for a search depth of 1. For a probing factor of 4 it is possible to reduce the number of investigated nodes for the board configurations 5, 6, 7 and 9. The other tested probing factors only cause a node reduction for two board configurations.

For a search depth of 2 it was possible to reduce the number of investigated nodes for the board configurations 1, 5 and 7 with a probing factor of 2 (see table 5.9). However, for this *probing factor* the number of visited nodes for the configurations 2, 3 and 4 increases by more than 15% compared to the highest reduction of 18% for configuration 1.

	Star2	$pf = 2$	$pf = 3$	$pf = 4$	$pf = 5$
1	*495*	-22	-101	-191	-281
2	*461*	-6	-10	-15	-20
3	*597*	-6	-12	-17	-22
4	*729*	-12	-38	-98	-113
5	*238*	**31**	**20**	**9**	-2
6	*1620*	-124	-202	**145**	**136**
7	*4825*	**840**	**935**	**774**	**661**
8	*411*	-9	-53	-104	-155
9	*849*	-5	-43	**1**	-31
10	*129*	-11	-29	-52	-56
total		676	467	452	117

Table 5.8: Node reduction with Star2.5 for different probing factors with a search depth of 1.

	Star2	pf = 2	pf = 3	pf = 4	pf = 5
1	*87051*	**16356**	**14466**	**17943**	**29815**
2	*40234*	-6907	-15827	-20511	-27094
3	*46334*	-11094	-20143	-23689	-28263
4	*48490*	-7859	-10801	-18261	-25645
5	*3637*	**238**	-72	-416	-782
6	*106943*	-2976	-23823	-23435	-26908
7	*254663*	**34518**	**35424**	**32207**	**7563**
8	*10399*	-762	-2786	-4183	-6374
9	*36506*	-892	-4535	-7098	-13547
10	*892*	-298	-636	-900	-852
total		20324	-28733	-48343	-92087

Table 5.9: Node reduction with Star2.5 for different probing factors with a search depth of 2.

Table 5.10 shows the results for a search depth of 3, which can also be reached with Star2 for certain positions in acceptable circumstances. However, it was only possible to reach a search reduction of 0.8% for configuration 7 which has the highest number of investigated nodes with Star2. This position is searched by Star2 in 4600 ms (see table 5.6). For this reason the node reduction was only efficient if the allowed search time is higher 4600 ms.

Concluding, Star2.5 does not give an efficient improvement to Star2. With peeper search depth the number of additionally searched nodes with Star2.5 increases significantly and it is not possible to save time. But if a probing factor should be chosen, the value 2 is the best choice.

5.2.6 And The Winner Is?

Finally, we investigated that Star2 performs best and Star2.5 does not improve the performance. For the evaluation function configuration 4 is used. With a

	Star2	pf = 2	pf = 3	pf = 4	pf = 5
1	*3695994*	-71543	-253170	-554303	-1006383
2	*1366584*	-582898	-1173697	-1619286	-2338154
3	*2250696*	-804694	-1612099	-2219613	-3152343
4	*1080500*	-276811	-597127	-1006476	-1361580
5	*54087*	-13457	-27792	-46050	-68521
6	*3472337*	-733414	-1860638	-2042892	-3162608
7	*7936733*	**67910**	-1488914	-2466523	-3128425
8	*360387*	-151806	-283755	-370721	-568822
9	*1069885*	-296099	-557258	-895593	-1339332
10	*894*	-305	-658	-934	-886
	total	-2720031	-7855108	-11222391	-19279397

Table 5.10: Node reduction with Star2.5 for different probing factors with a search depth of 3.

dynamic lower and upper bounds calculation the player tries to adapt as best as he can to the current situation.

5.3 Monte Carlo and MCTS

This section shows the results of the tests that have been carried out in order to achieve the best configuration with regard to MC and MCTS. Section 5.3.1 represents the results for two different evaluation possibilities for a simulated game. The results of different UCT settings are listed in section 5.3.2. Section 5.3.3 describes the tests and results for the enhancements explained in section 4.2.4. Section 5.3.4 illustrates the advantage of parallelisation with regards to achieving an increased number of simulations for a given time. Concluding, the best player configuration is presented in section 5.3.5.

5.3.1 Evaluation

In order to choose the best move Monte-Carlo Search and Monte-Carlo Tree Search evaluate randomly played simulations for possible moves. The evaluation is done when the simulation ends and a terminal position is reached. The two evaluation possibilities were explained in detail in section 4.2.2. A relative evaluation takes place when only an indication for a win (1), draw (0) or a loss (-1) is returned. For an absolute evaluation the differece in score of the players is compared. To decide which evaluation approach is the better one 200 games with four different playing times were simulated. The playing time indicates how much time a player has to select the best move. These tests were made for Monte-Carlo Search and for Monte-Carlo Tree Search separately. The results are shown in table 5.11.

MCr and MCTSr use the relative evaluation and MCa and MCTSa use the absolute evaluation. The results show that with increasing playing time per turn the winning rate increases for the player who uses the relative evaluation. Only for Monte-Carlo Search and a playing time of 1000 ms was it possible for the player with the absolute evaluation to win. Due to this unique result, a relative evaluation is used for further experiments.

time	MCr	MCa	MCTSr	MCTSa
1000	44%	56%	56%	44%
3000	53%	47%	69%	31%
5000	56%	44%	71%	29%
7000	65%	35%	75%	25%
	54.5%	**45.5%**	**67.75%**	**32.25%**

Table 5.11: The influence of absolute and relative scoring for the winning rate.

Table 5.12 shows how often which type of terminal position was reached with the different evaluation approaches. Terminal position of type (1) ($t1$) is reached when a player wins by points and terminal position of type (2) ($t2$) is reached when the game ends prematurely. For MC and MCTS the player with the relative evaluation reached on average the terminal position of type (1) twice as often as the player with the absolute evaluation.

	MCr		MCa		MCTSr		MCTSa	
time	t1	t2	t1	t2	t1	t2	t1	t2
1000	69	19	99	13	94	18	80	8
3000	72	27	86	15	117	22	58	3
5000	86	27	77	10	129	13	56	2
7000	97	34	60	9	123	28	42	7
	75.2%	**24.8%**	**87.3%**	**12.7%**	**85.1%**	**14.9%**	**92.2%**	**7.8%**

Table 5.12: Reached terminal positions with absolute and relative scoring.

5.3.2 UCT Selection

The UCT formula contains a constant factor C which balances the exploitation and exploration of the selection strategy. The implemented algorithm uses the square root of C for the calculation. Furthermore, a threshold value is added to the selection strategy which indicates how often the selection strategy selects a purely random move until the behaviour is changed to the selection by the UCT formula. The winning rates of all possible combinations of C and the chosen threshold are shown in table 5.13. The threshold values listed in table 5.13 are relative to the percentage of child nodes taken, e.g., with a percentage of 0.5 and 34 child nodes, the threshold is 17. Due to this fact, the threshold value is determined dynamically for each node. For each combination 100 games were simulated and the MCTS player played against a Monte-Carlo player.

The results show that MCTS outperforms MC with some exceptions for a large C. Furthermore we derive that with an increasing C the winning rate decreases. On average the best results were reached for a C of 0.5. The threshold value does not significantly influence the winning rate for the best C. Nevertheless, the threshold value improves the winning rate for larger C values and makes the winning rate more stable.

	threshold				
C	0	0.25	0.5	0.75	1
0.5	84%	77%	85%	81%	83%
1	88%	79%	79%	80%	85%
2	69%	78%	79%	79%	84%
3	63%	75%	74%	75%	75%
5	57%	77%	74%	70%	73%
7	38%	71%	67%	69%	75%
10	47%	60%	68%	66%	74%
15	36%	45%	60%	63%	70%

Table 5.13: Winning rate of MCTS agains MC with different C and threshold vakues for MCTS

5.3.3 Modifications

These experiments test the enhancements described in section 4.2.3.

Heuristic

To improve the quality of a selected move of the UCT selection and during the simulation phase, we test a history heuristic. During the simulation step it is possible to choose the next move pseudo-randomly by the help of the stored values of the history heuristic. Table 5.14 presents the results for this technique. For this experiment 200 games between two MCTS players were simulated, where one player used pseudo-randomly selected moves during the simulation step and the other one used randomly selected moves. This experiment was done for four different move times to see whether more search time would improve the quality of the history-heuristic values and hence the winning rate by increasing the quality of the pseudo-randomly selected moves. However, the results of table 5.14 illustrate that the history heuristic does not improve the performance of the MCTS player. Due to this fact, we continue to select moves during the simulation step purely at random. A second option to take

	milliseconds			
	1000	3000	5000	7000
win rate	40.5%	44%	44.5%	40%

Table 5.14: Winning rate with pseudo-random moves during the simulation step.

advantage of the history heuristic is to use these values for the selection step as a heuristic value (see section 4.2.4.2). For this purpose the three best configurations of section 5.3.2 are chosen and tested with the adapted selection formula. The results of these tests are listed in table 5.13.

Adding a heuristic value to the UCT selection does not increase the winning rate of MCTS. Both tests show that the information of the history heuristic gives no advantage but instead lowers the winning rate. For this reason, we

time (ms)	C=0 ,t=1	C=0.5 ,t=0.5	C=1 ,t=1
3000	50%	43%	40%

Table 5.15: Winning rate with a progressive-bias strategy.

ignore the addition of a heuristic value for further experiments.

Dynamic Simulation Cut

With the dynamic simulation cut we want to abbreviate the simulation step and increase the number of simulations. The increased number of simulations should furthermore improve the quality of the ultimately chosen move. The results for this experiment are shown in table 5.16.

For each SA value 200 games were simulated to obtain a winning rate, where a MCTS player with a dynamic simulation cut played against a MCTS player without a dynamic simulation cut. The results confirm the assumption that a simulation cut increases the number of simulations. Nevertheless, the results also show that this technique does not improve the winning rate. A likely reason for the lowered winning rate is that the premature cut lowers the certainty of a real game end. Hence, the dynamic simulation cut is not the best choice for OnTop to improve its winning rate and will not be applied to the final configuration.

	SA=1	SA=2	SA=3	SA=4
win rate	44.5%	47%	48.5%	47%
gain of number of simulations	18%	8%	-1%	-2,55%

Table 5.16: Winning rate for applied dynamic simulation cut with different average tile scores.

5.3.4 Parallelisation

To test the advantage of parallelisation we investigated several simulations done for the ten predefined board positions and compared these to the number of simulations done of a non-parallelised player. For Monte-Carlo Search we tested plain parallelisation and for Monte-Carlo Tree Search we implemented the leaf and root parallelisation. In both cases we increased the number of simulations for all test positions. These tests were run on an Intel dual core CPU with 2.33 GHz and 2 GB of RAM.

MC: Parallelisation

The initial idea for the parallelisation of Monte-Carlo Search and Monte-Carlo Tree Search was to increase the number of simulations for a certain playing time. Figure 5.5 pictures the number of investigated nodes of a parallelised Monte-Carlo player for different numbers of used threads. The results do not confirm the original assumption of a much higher number of investigated nodes. The best results were reached for two and four threads which increased the number of searched nodes up to 33%. However, for endgame positions, (board configuration 5, 8 and 10) where a simulation only includes six to eight further plies, the number of investigated nodes is decreased by parallelisation.

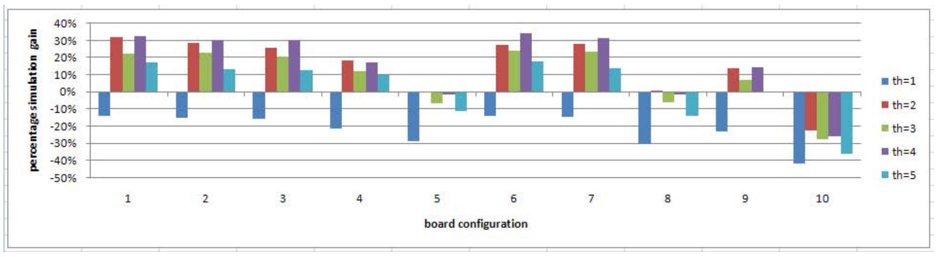


Figure 5.5: The advantage of parallelisation for Monte-Carlo Search.

MCTS: Leaf Parallelisation

The leaf parallelisation of MCTS takes place on a leaf node of the search tree, where a number of threads were started and each thread performs a single simulation. The results for leaf parallelisation are illustrated in figure 5.6. The best results were reached with 5 threads. However, 3 and 4 threads perform quite well, too. After 5 threads the simulation gain decreases greatly compared to the gain with 5 threads. However figure 5.6 does not show the results for test position ten, because the results have strongly influenced the figure. With one thread the gain was 0%, with two threads 99%, with three threads 190%, with four threads 283%, with five threads 376% and with six threads 450%.

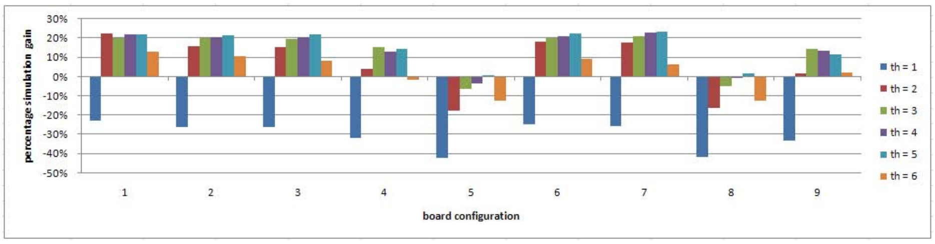


Figure 5.6: The advantage of leaf parallelisation for Monte-Carlo Tree Search.

MCTS: Root Parallelisation

For the root parallelisation each thread handles its own search tree in memory. The results for this parallelisation technique are illustrated in figure 5.7. The figure shows quite clearly that the best results were reached with two threads. Therefore we have a gain of simulations between 57% and 74% and the gain is stable for all configurations.

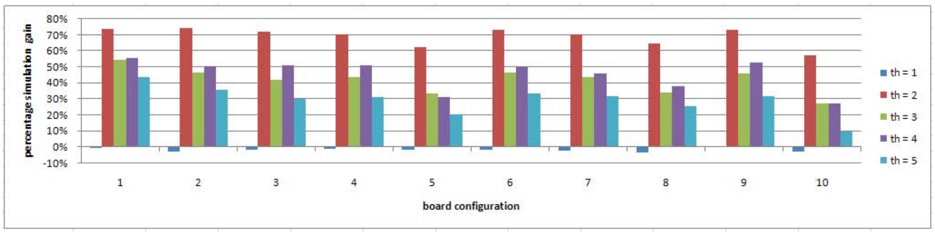


Figure 5.7: The advantage of root parallelisation for Monte-Carlo Tree Search.

5.3.5 And The Winner Is?

The previous experiments had shown that MCST outperforms MC. We also realised that the additional enhancements of MCTS do not improve MCTS. The best improvement turned out to be the root parallelisation. Concluding, the best configuration is a MCTS player with a UCT value of 1, a threshold value of 0 and an applied root parallelisation with two threads. This configuration uses purely at random selected moves during the simulation step and does not use the dynamic simulation cut.

5.4 Expectimax vs. Monte Carlo

Finally, the best player configuration of section 5.2, which is described in subsection 5.2.6, played against the best player configuration of section 5.3 which is described in subsection 5.3.5. For this experiment we simulated 1400 games and each player had 5 seconds to select the best move. The 5 seconds should increase the probability that Star2 searches at least to a depth of 3 for all game positions. The results are illustrated in figure 5.8 and table 5.17. Table 5.17 shows that MCTS absolutely outperforms Star2. A winning rate of 80% is a unique sign for the strength of MCTS. An interesting fact is that 13.2% of the games of Star2 were wins by reaching terminal position of type (2), due to the fact that the results of all other experiments related to the reached terminal positions indicate that terminal position one is preferred over terminal position of type (2).

	MCTS	Star2
win rate	80%	20%
terminal position type (1)	66.4%	6.8%
terminal position type (2)	13.6%	13.2%

Table 5.17: The winning rate distribution of the best player configurations.

Figure 5.8 illustrates the score of both players during the game. These points are calculated by subtracting the remaining stones from the earned points of each player. To illustrate this, we draw a graph which shows the games that MCTS won and a graph which shows the games that Star2 won. The graphs are based on average values for all simulated games and show that Star2 had the better start every time. Nevertheless, it lost most of the games because after more than half of the game MCTS earned more points than Star2. The graphs show that MCST does not attempt to aggressively collect points. Instead, MCTS tries to win by focusing on a wide distribution of their colour in order to keep many options to earn points.

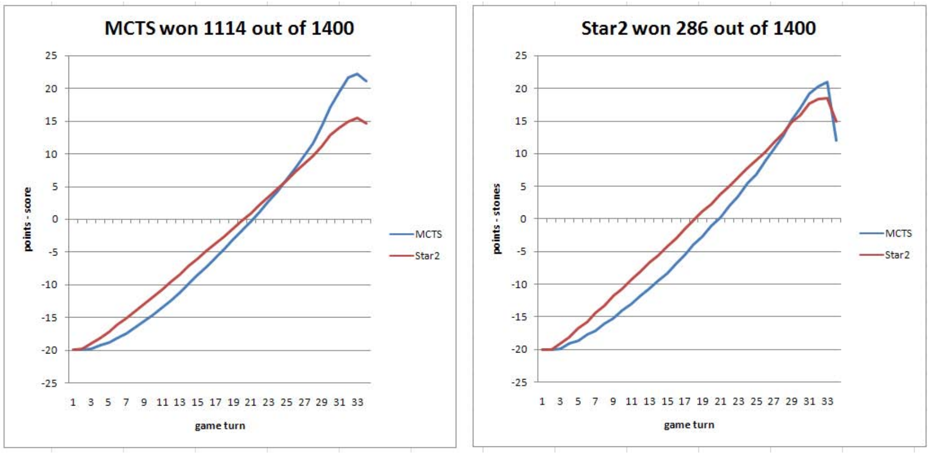


Figure 5.8: Score progress for won and lost games of MCTS against Star2.

5.5 Computer Player against Human Opponents

This section gives an impression about the strength of the computer players against human players. Therefore two advanced human players (Robert Briesemeister, Cathleen Heyden) and one intermediate player (Frank Fleischhack) played against the two player configurations which were involved in the tests of section 5.4. The results of these games are presented in table 5.18. The results are given as m/n, with the meaning that the program won m games out of n.

The results show that Star2 reached a higher winning rate against humans than against MCTS. However, MCTS performs best and reached a winning rate of 82%. The advanced players were able to challenge one of the computer players. Due to this fact, it is quite interessting that both advanced player does not challenge the same player. The experince of the human players was that the strategy to beat the computer player should be to close the open circles of the opponent without any advantage for the opponent. This strategy focuses on responding to the actions of the opponent and not to increase the points of their own.

	MCTS	Star2
Cathleen	4/8	6/8
Frank	12/12	4/7
Robert	7/8	3/8
winning rate	82%	56%

Table 5.18: Winning rate of MCTS and Star2 against human players.

Chapter 6

Conclusions and Future Research

This chapter presents the final conclusions of our research and recommendations for future research. Section 6.1 and 6.2 answers the research questions and the problem statement. Finally, section 6.3 gives some possibilities for future research.

6.1 Answering the Research Questions

In section 1.2 the following research questions were stated:

1. *What is the complexity of the game OnTop?*

This question was answered in chapter 3, where we calculated the game-tree complexity as $O(10^{64})$ and the state-space complexity as $O(10^{77})$. Both numbers are only estimates, because the calculation was done with average values for the branching factors and an average game length of 1400 simulated games. Furthermore, the state-space complexity includes some illegal positions.

2. *Can Expectimax be used for its implementation?*

Expectimax is a full-width search, which was investigated first in order to build a computer player for OnTop. Due to the large width of the search tree, which results from the high branching factor of OnTop, the search of Expectimax usually reached a depth of no more than 2. The quality of the results of Expectimax strongly depends on the evaluation function. If the features and their weightings are reliable the evaluation function leads to good results.

3. *Can Monte-Carlo Search be used for its implementation?*

Monte-Carlo Search can be used for the implementation of OnTop. It requires no strategic knowledge and evaluates a position using simulations to the end of the game. For OnTop the behaviour appears quite passive at the beginning, i.e., appears as if Monte-Carlo Search has no chance to win. However, during the game the strength of Monte-Carlo grows and reaches impressive results.

4. How can the investigated techniques be improved?

The results of section 5.4 demonstrate that the MCTS configuration of section 5.3.5 performs best. MCTS won 80% of the 1400 simulated games. The strength of Star2 is in the first plies where it leads each time. Nevertheless Star2 fails to keep the lead in the game.

5. Which improved technique of Monte-Carlo and Expectimax is most effective?

There are several techniques to improve Expectimax and Monte-Carlo. Expectimax can be adapted to a pruning algorithm like Alpha-Beta search. The derived algorithms are Star1 and Star2. Both algorithms do not further investigate the full width of the search tree. They cut unpromising tree paths with the help of a search window. Star2 also contains a probing phase to get more accurate search-window bounds which are determined by the search of a single child node. If more than one child node is searched during the probing phase Star2 becomes Star2.5.

An improvement to Monte-Carlo Search is Monte-Carlo Tree Search. This technique builds a search tree depending on game simulations. To select the most promising leaf node for further simulations a UCT (Upper Confidence bounds applied to Trees) selection strategy is applied on the search tree. The evaluation of a simulation returns an indication of the outcome of the game. In order to improve the accuracy of MCTS we increase the number of simulations with root parallelisation.

6.2 Answering the Problem Statement

Now that all research questions are answered, we can also answer the problem statement given in section 1.2:

Can we build an effective and efficient computer player for OnTop?

The results shows that it is possible to build a computer player which performs quite well. The most convincing algorithm was Monte-Carlo Tree Search with root parallelisation. We used a UCT value of 1 and a threshold of 0% of the child nodes. Due to this fact we use every time the selection strategy. Nevertheless, the results of Star2 were not bad. However, Star2 wasted too much colour potential for the first moves without planning for the complete game. With regard to MCTS, this disadvantage was compensated by game simulations which gave a rough estimate of the entire game.

6.3 Future Research

There are several areas for potential future research. For instance, the complexity analysis for the state-space and game-tree complexity can be calculated more accurately. The estimated values can be replaced by more precise values derived from a built formula so that even the illegal positions are excluded from the calculation.

Furthermore, the investigated search algorithms can be more fine-tuned and enhanced. In order to improve the Star algorithms the evaluation function can be enhanced with more features and the weightings can be more fine-tuned. Our test results showed that Star2 seems to lose the game at the beginning of the game, where it still scores points but does not plan ahead. A possibility to lower or remove this disadvantage could be to give Star2 more search time at the beginning when the branching factor increases and lower the search time at the end when the branching factor decreases. To achieve this, the tests must be adapted with a total game time as opposed to the time per move which was used in our tests. However, there is also another pruning technique which can be investigated, ChanceProbCut [31].

A possibility to enhance Monte-Carlo Tree Search is to add more meaningful knowledge to the search. This can be done with transposition tables which in contrast to the implemented heuristic table save information of the entire board configuration with the help of a Zobrist key [6].

Derived from the results of section 5.4 it is recommended to investigate a combination of Star2 and MCTS. For this purpose, Star2 earns points until a certain threshold is reached, after which MCTS takes over to save the lead and win the game.

Additionally, it is recommended that the future research investigates a multi-player variation of OnTop, because OnTop can be played with up to 4 players.

Bibliography

[1] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, Netherlands, 1994.

[2] B. W. Ballard. A Search Procedure for Perfect Information Games of Chance: Its Formulation and Analysis. In D. L. Waltz, editor, *Proceedings of the 1st Annual National Conference on Artificial Intelligence. Stanford University, August 18-21*, pages 111–114. AAAI Press, 1982.

[3] B. W. Ballard. The *-Minimax Search Procedure for Trees Containing Chance Nodes. *Artificial Intelligence*, 21(3):327–350, 1983.

[4] D. Billings, A. Davidson, T. Schauenberg, N. Burch, M. H. Bowling, R. C. Holte, J. Schaeffer, and D. Szafron. Game-Tree Search with Adaptation in Stochastic Imperfect-Information Games. In H. J. van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 21–34. Springer, 2004.

[5] B. Bouzy. Old-fashioned Computer Go vs Monte-Carlo Go. http://ewh.ieee.org/cmte/cis/mtsc/ieeecis/tutorial2007/Bruno_Bouzy_2007.pdf, Powerpoint presentation at Honolulu, Hawaii, 2007.

[6] D.M. Breuker, J. W.H.M. Uiterwijk, and H.J. van den Herik. Replacement Schemes for Transposition Tables. *ICCA Journal*, 17(4):183–193, 1994.

[7] B. Brügemann. Monte Carlo Go. page 13, October 1993. Max-Planck-Institut of Physics, München, Germany, http://www.ideanest.com/vegos/MonteCarloGo.pdf.

[8] G. Van Brummelen and M. Kinyon, editors. *Mathematics and the Historian's Craft*, chapter 11, pages 297–328. CMS Books in Mathematics. New York, NY : Springer Science+Business Media, Inc., 2005.

[9] T. Cazenave and N. Jouandeau. On the Parallelization of UCT. In *Proceedings of the Computer Games Workshop 2007(CGW 2007)*, pages 93–101, Universiteit Maastricht, Maastricht, The Netherlands, June 2007.

[10] T. Cazenave and N. Jouandeau. A Parallel Monte-Carlo Tree Search Algorithm. In H. J. van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*,

volume 5131 of *Lecture Notes in Computer Science*, pages 72–80. Springer, 2008.

[11] G. M.J.-B. Chaslot, M. H.M. Winands, and H. J. van den Herik. Parallel Monte-Carlo Tree Search. In H. J. van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, volume 5131 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 2008.

[12] G. M.J.-B. Chaslot, M. H.M. Winands, H. J. van den Herik, J.W.H.M. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation (NMNC)*, 4(03):343–357, 2008.

[13] P.-A. Coquelin and R. Munos. Bandit Algorithms for Tree Search. *Technical Report 6141, INRIA*, 2007.

[14] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In J. H. van den Herik, P. Ciancarini, and (jeroen) H. H. L. M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630/2007 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[15] F. A. Dahl. The Lagging Anchor Algorithm: Reinforcement Learning in Two-Player Zero-Sum Games with Imperfect Ifnformation. *Machine Learning*, 49(1):5–37, 2002.

[16] R. Eckard. Stan Ulam, John von Neumann, and the Monte Carlo Method. In *Los Alamos Science Special Issue*, number 15, page 11. 1987.

[17] S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[18] A. Gilpin. Algorithms for abstracting and solving imperfect information games. Master's thesis, Carnegie Mellon University - Computer Science Department, April 2007.

[19] A. Grebennik. Monte Carlo Method in the Game of Clobber. Technical report, Tartu University, 2005. BSc thesis.

[20] K. Hafner. In an Ancient Game, Computing's Future. *The New York Times*, pages 1–4, July 2002. http://www.nytimes.com/2002/08/01/technology/in-an-ancient-game-computing-s-future.html.

[21] T. Hauk, M. Buro, and J. Schaeffer. *-Minimax Performance in Backgammon. In H. J. van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 41–66. Springer, 2004.

[22] T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-Minimax Search. In H. J. van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2004.

[23] C. Heyden. Implementing a computer player for carcassone. Master's thesis, Maastricht University, 2009.

[24] F.-H. Hsu. *Behind Deep Blue.* Princeton University Press, Princeton, NJ, USA, 2002.

[25] J. Kloetzer, H. Ida, and B. Bouzy. The Monte-Carlo Approach in Amazons. In *Proceedings of the Computer Games Workshop 2007(CGW 2007)*, pages 185–192, Universiteit Maastricht, Maastricht, The Netherlands, 2007.

[26] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, pages 282–293, 2006.

[27] Nicholas Metropolis. The beginning of the Monte Carlo method. *Los Alamos Science*, 15:125–130, 1987.

[28] X. Niu and M. Müller. An Improved Safety Solver for Computer Go. In H. J. van den Herik, Yngvi Björnsson, and Nathan S. Netanyahu, editors, *Computers and Games, 4th International Conference, CG 2004, Ramat-Gan, Israel, July 5-7, 2004, Revised Papers*, volume 3846 of *Lecture Notes in Computer Science*, pages 97–112. Springer, 2004.

[29] P. Rajkumar. A Survey of Monte-Carlo Techniques in Games. *Master's Scholarly Paper.*

[30] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[31] M.P.D. Schadd, M. H.M. Winands, and J. W.H.M. Uiterwijk. Forward Pruning in Chance Nodes. In *IEEE Symposium on Computational Intelligence and Games (CIG'09)*, pages 178–185, 2009.

[32] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.

[33] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. Reviving The Game of Checkers. In *Games of No Chance*, pages 119–136. Cambridge University Press, 1991.

[34] C. E. Shannon. Programming a computer for playing chess. In *Computer chess compendium*, pages 2–13, New York, NY, USA, 1988. Springer-Verlag New York, Inc. ISBN 0-387-91331-9.

[35] B. Sheppard. *Towards Perfect Play of Scrabble.* PhD thesis, Institute for Knowledge and Agent Technology (IKAT), Universiteit Maastricht, 2002.

[36] B. Sheppard. World-championship-caliber Scrabble. *Artif. Intell.*, 134(1-2):241–275, 2002.

[37] G. Tesauro and G. R. Galperin. *On-line policy improvement using Monte-Carlo Search.* Cambridge, 1996.

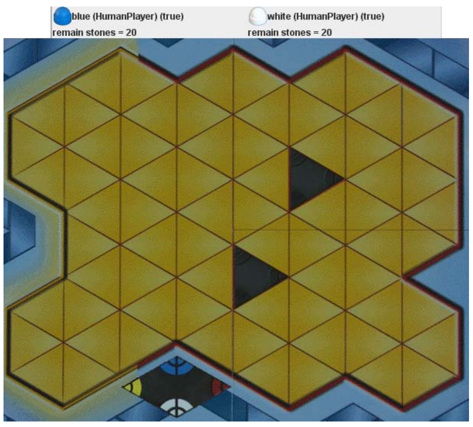
[38] J. Veness. Expectimax Enhancements for Stochastic Game Players. Master's thesis, The university of New South Wales, 2006.

[39] M. Wächter. Uct: Selektive Monte-Carlo-Simulation in Spielbäumen. 2008. http://www.ke.tu-darmstadt.de/lehre/ss08/challenge/Ausarbeitungen/Waechter.pdf.

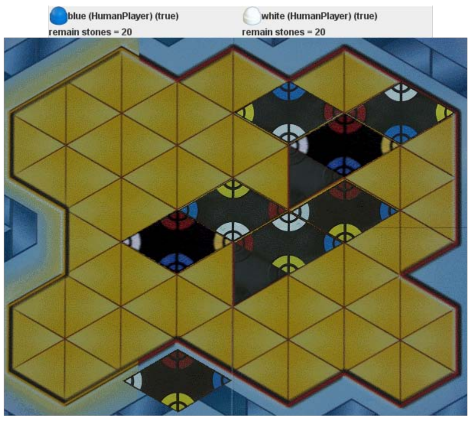
[40] M. H. M. Winands, Y. Björnsson, and J.-T. Saito. Monte-Carlo Tree Search Solver. In H. J. van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2008.

Appendix A
The 10 test positions

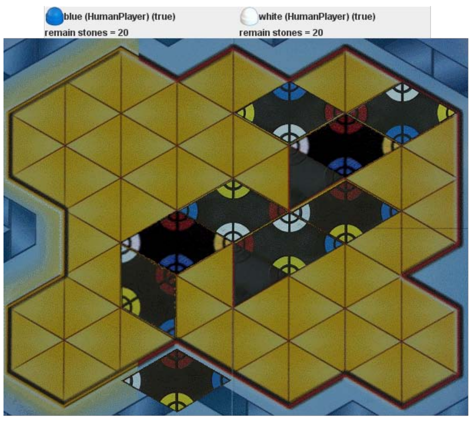
Figure A.1: The subfigures below indicates the 10 predefined positions which were used for testing.



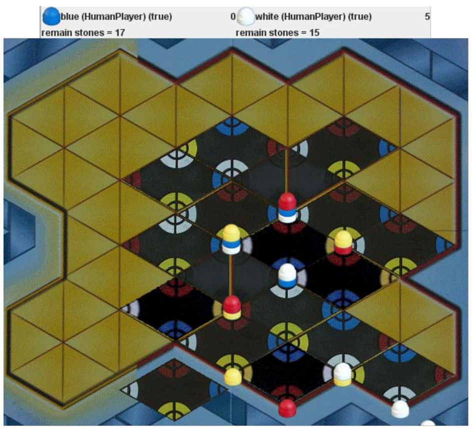
Game state 1.



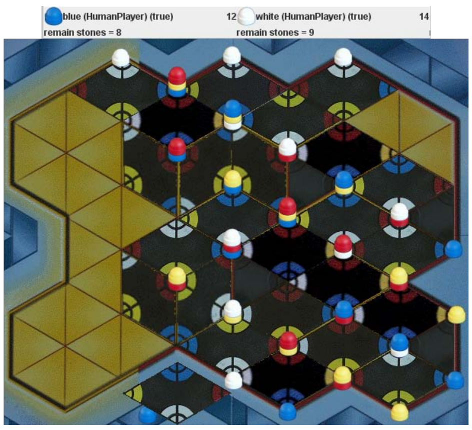
Game state 2.



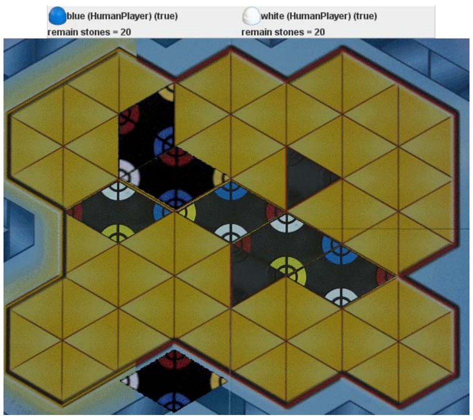
Game state 3.



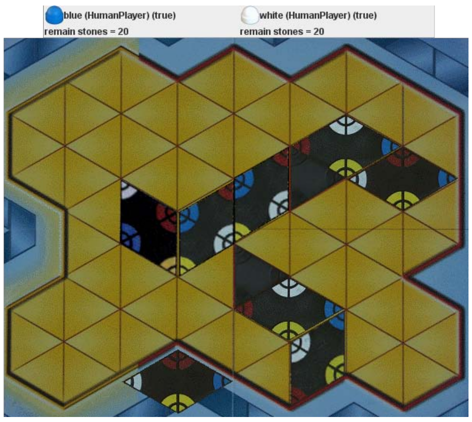
Game state 4.



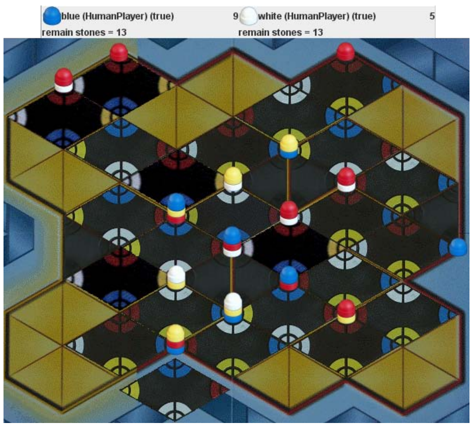
Game state 5.



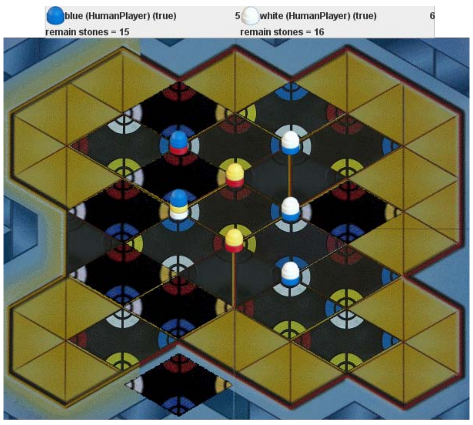
Game state 6.



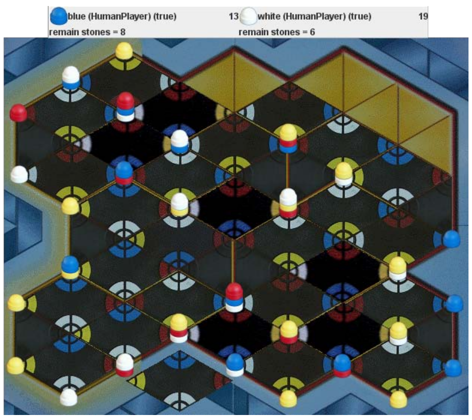
Game state 7.



Game state 8.



Game state 9.



Game state 10.